Algorithms for programmers
ideas and source code

This document is work in progress: read the “important remarks” near the beginning

Jörg Arndt
arndt@jjj.de

Draft version\(^1\) of 2008-August-17

\(^1\)The latest version and the accompanying software is online at http://www.jjj.de/fxt/
## Contents

### Important remarks about this document

### Low level algorithms

#### 1 Bit wizardry

1.1 Trivia .......................................................... 3
1.2 Operations on individual bits ........................................ 8
1.3 Operations on low bits or blocks of a word ...................... 9
1.4 Extraction of ones, zeros, or blocks near transitions ............ 12
1.5 Computing the index of a single set bit .......................... 14
1.6 Operations on high bits or blocks of a word .................... 16
1.7 Functions related to the base-2 logarithm ....................... 18
1.8 Counting bits and blocks of a word ............................ 19
1.9 Counting bits of many words .................................. 22
1.10 Words as bit sets ............................................. 24
1.11 Avoiding branches ............................................ 25
1.12 Bit-wise rotation of a word .................................... 29
1.13 Functions related to bit-wise rotation and binary necklaces .... 30
1.14 Reversing the bits of a word ................................... 35
1.15 Bit-wise zip .................................................. 40
1.16 Gray code and parity .......................................... 42
1.17 Bit sequency ................................................ 48
1.18 Powers of the Gray code ...................................... 50
1.19 Invertible transforms on words .................................. 51
1.20 Space filling curves ............................................ 57
1.21 Scanning for zero bytes ....................................... 60
1.22 2-adic inverse and square root ................................ 61
1.23 Radix −2 representation ...................................... 63
1.24 A sparse signed binary representation ......................... 66
1.25 Generating bit combinations ................................... 67
1.26 Generating bit subsets of a given word ......................... 71
1.27 Binary words as subsets in lexicographic order ................. 73
1.28 Minimal-change bit combinations ............................... 78
1.29 Fibonacci words ............................................. 80
1.30 Binary words and parentheses strings * .......................... 83
1.31 Permutations via primitives ................................... 86
1.32 CPU instructions often missed ................................ 88

#### 2 Permutations

2.1 The revbin permutation ......................................... 91
2.2 The radix permutation ......................................... 95
## CONTENTS

2.3 In-place matrix transposition ................................................. 96  
2.4 The triple reversion technique * ............................................. 97  
2.5 The zip permutation .......................................................... 99  
2.6 The reversed zip permutation ................................................ 101  
2.7 The XOR permutation .......................................................... 102  
2.8 The Gray code permutation .................................................... 103  
2.9 The reversed Gray code permutation ....................................... 107  
2.10 General permutations and their operations ............................... 109

3 Sorting and searching ................................................................... 121  
3.1 Sorting .................................................................................. 121  
3.2 Binary search ......................................................................... 123  
3.3 Index sorting ......................................................................... 124  
3.4 Pointer sorting ....................................................................... 126  
3.5 Sorting by a supplied comparison function ................................. 127  
3.6 Determination of unique elements .......................................... 130  
3.7 Unique elements with inexact types ....................................... 131  
3.8 Determination of equivalence classes ..................................... 133  
3.9 Determination of monotonicity and convexity * ....................... 137  
3.10 Heapsort ............................................................................. 140  
3.11 Counting sort and radix sort ................................................ 140  
3.12 Searching in unsorted arrays ................................................ 143

4 Data structures ........................................................................... 147  
4.1 Stack (LIFO) ......................................................................... 147  
4.2 Ring buffer ........................................................................... 149  
4.3 Queue (FIFO) ......................................................................... 150  
4.4 Deque (double-ended queue) .................................................. 152  
4.5 Heap and priority queue .......................................................... 154  
4.6 Bit-array ............................................................................... 158  
4.7 Left-right array .................................................................... 160  
4.8 Finite state machines .............................................................. 163  
4.9 Emulation of coroutines .......................................................... 165

II Combinatorial generation .......................................................... 169

5 Conventions and considerations .................................................. 171  
5.1 About representations and orders ......................................... 171  
5.2 Ranking, unranking, and counting ......................................... 172  
5.3 Characteristics of the algorithms .......................................... 172  
5.4 Optimization techniques .......................................................... 172  
5.5 The implementations, demo-programs, and timings .................. 173

6 Combinations .............................................................................. 175  
6.1 Lexicographic and co-lexicographic order ............................... 176  
6.2 Order by prefix shifts (cool-lex) ............................................. 180  
6.3 Minimal-change order .............................................................. 181  
6.4 The Eades-McKay strong minimal-change order ...................... 182  
6.5 Two-close orderings via endo/enup moves ............................... 186  
6.6 Recursive generation of certain orderings ............................... 191

7 Compositions .............................................................................. 193  
7.1 Co-lexicographic order ............................................................. 193
## CONTENTS

7.2 Co-lexicographic order for compositions into exactly \( k \) parts ............................................. 195
7.3 Compositions and combinations ................................................. 197
7.4 Minimal-change orders .................................................... 198

8 Subsets ........................................................................ 201
8.1 Lexicographic order ............................................................. 201
8.2 Minimal-change order ......................................................... 204
8.3 Ordering with De Bruijn sequences ........................................ 207
8.4 Shifts-order for subsets ....................................................... 208
8.5 \( k \)-subsets where \( k \) lies in a given range ............................... 210

9 Mixed radix numbers ............................................................. 219
9.1 Counting (lexicographic) order ............................................... 219
9.2 Gray code order ............................................................... 222
9.3 gsex order .................................................................. 226
9.4 endo order .................................................................. 229
9.5 Gray code for endo order ................................................... 230

10 Permutations .................................................................. 233
10.1 Lexicographic order ............................................................. 233
10.2 Co-lexicographic order ....................................................... 235
10.3 Factorial representations of permutations ................................. 236
10.4 An order from reversing prefixes ........................................... 246
10.5 Minimal-change order (Heap’s algorithm) ................................. 249
10.6 Lipski’s Minimal-change orders .............................................. 251
10.7 Strong minimal-change order (Trotter’s algorithm) ..................... 254
10.8 Minimal-change orders from factorial numbers ......................... 259
10.9 Orders where the smallest element always moves right ................. 265
10.10 Single track orders ............................................................ 271
10.11 Star-transposition order ...................................................... 275
10.12 Derangement order ............................................................ 277
10.13 Recursive algorithm for cyclic permutations ............................... 279
10.14 Minimal-change order for cyclic permutations ............................ 281
10.15 Permutations with special properties ....................................... 283

11 Subsets and permutations of a multiset .................................... 291
11.1 Subsets of a multiset ............................................................ 291
11.2 Permutations of a multiset .................................................... 292

12 Gray codes for strings with restrictions ..................................... 299
12.1 Fibonacci words ................................................................ 300
12.2 Generalized Fibonacci words ................................................. 302
12.3 Digit \( x \) followed by at least \( x \) zeros ...................................... 305
12.4 Generalized Pell words ......................................................... 306
12.5 Sparse signed binary words ................................................. 308
12.6 Strings with no two successive nonzero digits ......................... 310
12.7 Strings with no two successive zeros ..................................... 312
12.8 Binary strings without substrings 1x1 or 1xy1 ............................ 313

13 Parenthesis strings .............................................................. 317
13.1 Co-lexicographic order ........................................................ 317
13.2 Gray code via restricted growth strings .................................... 319
13.3 The number of parenthesis strings: Catalan numbers ................. 324
13.4 Increment-\( i \) RGS and \( k \)-ary trees .............................................. 325
14 Integer partitions 331
  14.1 Solution of a generalized problem 331
  14.2 Iterative algorithm 333
  14.3 Partitions into \( m \) parts 334
  14.4 The number of integer partitions 336

15 Set partitions 341
  15.1 Recursive generation 341
  15.2 The number of set partitions: Stirling set numbers and Bell numbers 345
  15.3 Restricted growth strings 347

16 A string substitution engine 357

17 Necklaces and Lyndon words 361
  17.1 Generating all necklaces 362
  17.2 The number of binary necklaces 369
  17.3 The number of binary necklaces with fixed content 370

18 Hadamard and conference matrices 373
  18.1 Hadamard matrices via LFSR 373
  18.2 Hadamard matrices via conference matrices 375
  18.3 Conference matrices via finite fields 377

19 Searching paths in directed graphs 381
  19.1 Representation of digraphs 382
  19.2 Searching full paths 383
  19.3 Conditional search 388
  19.4 Edge sorting and lucky paths 392
  19.5 Gray codes for Lyndon words 394

III Fast orthogonal transforms 401

20 The Fourier transform 403
  20.1 The discrete Fourier transform 403
  20.2 Summary of definitions of Fourier transforms 404
  20.3 Radix-2 FFT algorithms 406
  20.4 Saving trigonometric computations 411
  20.5 Higher radix FFT algorithms 413
  20.6 Split-radix Fourier transforms 421
  20.7 Symmetries of the Fourier transform 424
  20.8 Inverse FFT for free 425
  20.9 Real valued Fourier transforms 426
  20.10 Multidimensional Fourier transforms 433
  20.11 The matrix Fourier algorithm (MFA) 434

21 Algorithms for fast convolution 437
  21.1 Convolution 437
  21.2 Correlation 442
  21.3 Weighted Fourier transforms and convolutions 445
  21.4 Convolution using the MFA 447
  21.5 The z-transform (ZT) 450
  21.6 Prime length FFTs 453

22 The Walsh transform and its relatives 457
22.1 The Walsh transform: Walsh-Kronecker basis ........................................... 457
22.2 Eigenvectors of the Walsh transform * ......................................................... 460
22.3 The Kronecker product ................................................................................. 461
22.4 A variant of the Walsh transform * .............................................................. 463
22.5 Higher radix Walsh transforms ................................................................. 465
22.6 Localized Walsh transforms .................................................................... 469
22.7 Dyadic (XOR) convolution ...................................................................... 473
22.8 The Walsh transform: Walsh-Paley basis ................................................ 475
22.9 Sequency ordered Walsh transforms .......................................................... 476
22.10 Slant transform ......................................................................................... 482
22.11 Arithmetic transform ............................................................................... 483
22.12 Reed-Muller transform ............................................................................ 486
22.13 The OR-convolution, and the AND-convolution ..................................... 489

23 The Haar transform ...................................................................................... 493
23.1 The `standard' Haar transform .................................................................. 493
23.2 In-place Haar transform ........................................................................... 495
23.3 Non-normalized Haar transforms .............................................................. 497
23.4 Transposed Haar transforms .................................................................... 499
23.5 The reversed Haar transform ................................................................... 501
23.6 Relations between Walsh and Haar transforms ....................................... 503
23.7 Nonstandard splitting schemes * .............................................................. 506

24 The Hartley transform .................................................................................. 511
24.1 Definition and symmetries ....................................................................... 511
24.2 Radix-2 FHT algorithms .......................................................................... 512
24.3 Complex FT by HT .................................................................................... 517
24.4 Complex FT by complex HT and vice versa ........................................... 518
24.5 Real FT by HT and vice versa .................................................................. 519
24.6 Higher radix FHT algorithms ................................................................... 520
24.7 Convolution via FHT ................................................................................ 521
24.8 Negacyclic convolution via FHT ............................................................... 525
24.9 Localized FHT algorithms ......................................................................... 525
24.10 Two-dimensional FHTs ........................................................................... 527
24.11 Discrete cosine transform (DCT) by HT ................................................. 528
24.12 Discrete sine transform (DST) by DCT .................................................. 529
24.13 Automatic generation of transform code ............................................... 530
24.14 Eigenvectors of the Fourier and Hartley transform * .............................. 532

25 Number theoretic transforms (NTTs) ............................................................ 535
25.1 Prime moduli for NTTs ............................................................................. 535
25.2 Implementation of NTTs ........................................................................... 537
25.3 Convolution with NTTs ............................................................................ 542

26 Fast wavelet transforms ............................................................................... 543
26.1 Wavelet filters .......................................................................................... 543
26.2 Implementation ......................................................................................... 545
26.3 Moment conditions .................................................................................. 546

IV Fast arithmetic ............................................................................................. 549

27 Fast multiplication and exponentiation ........................................................ 551
27.1 Asymptotics of algorithms ...................................................................... 551
CONTENTS

27.2 Splitting schemes for multiplication .................................................. 552
27.3 Fast multiplication via FFT ................................................................. 560
27.4 Radix/precision considerations with FFT multiplication ......................... 562
27.5 The sum-of-digits test ..................................................................... 564
27.6 Binary exponentiation ..................................................................... 565

28 Root extraction ...................................................................................... 569
28.1 Division, square root and cube root .................................................... 569
28.2 Root extraction for rationals ............................................................... 572
28.3 Divisionless iterations for the inverse $a$-th root ................................ 575
28.4 Initial approximations for iterations .................................................... 578
28.5 Some applications of the matrix square root ....................................... 579
28.6 Goldschmidt’s algorithm ................................................................... 584
28.7 Products for the $a$-th root ................................................................. 586
28.8 Divisionless iterations for polynomial roots ........................................ 589

29 Iterations for the inversion of a function ................................................. 591
29.1 Iterations and their rate of convergence .............................................. 591
29.2 Schröder’s formula ....................................................................... 592
29.3 Schröder’s formula and series reversion ............................................ 593
29.4 Householder’s formula .......................................................... 596
29.5 Dealing with multiple roots ............................................................... 597
29.6 More iterations ........................................................................ 599
29.7 Convergence improvement by the delta squared process ...................... 601

30 The arithmetic-geometric mean (AGM) ................................................. 603
30.1 The AGM .................................................................................. 603
30.2 The elliptic functions $K$ and $E$ ....................................................... 605
30.3 AGM-type algorithms for hypergeometric functions ......................... 608
30.4 Computation of $\pi$ ................................................................ 612
30.5 Arctangent relations for $\pi$ ............................................................... 620

31 Logarithm and exponential function ...................................................... 627
31.1 Logarithm ............................................................................. 627
31.2 Exponential function .......................................................... 633
31.3 Logarithm and exponential function of power series ....................... 636
31.4 Simultaneous computation of logarithms of small primes .................... 638

32 Numerical evaluation of power series ................................................... 641
32.1 The binary splitting algorithm for rational series .............................. 641
32.2 Rectangular schemes for evaluation of power series ......................... 647
32.3 The magic sumalt algorithm for alternating series ......................... 651

33 Computing the elementary functions with limited resources ................. 655
33.1 Shift-and-add algorithms for $\log_b(x)$ and $b^x$ ............................... 655
33.2 CORDIC algorithms .......................................................... 660

34 Recurrences and Chebyshev polynomials ............................................ 667
34.1 Recurrences ........................................................................ 667
34.2 Chebyshev polynomials ............................................................. 677

35 Cyclotomic polynomials, Hypergeometric functions, and continued fractions ................................................. 687
35.1 Cyclotomic polynomials, Möbius inversion, Lambert series ................ 687
35.2 Hypergeometric functions ............................................................ 696
35.3 Continued fractions ................................................................. 714
## Contents

### 36 Synthetic Iterations *

36.1 A variation of the iteration for the inverse .................................................. 723
36.2 An iteration related to the Thue constant ....................................................... 727
36.3 An iteration related to the Golay-Rudin-Shapiro sequence ............................. 728
36.4 Iterations related to the ruler function ......................................................... 730
36.5 An iteration related to the period-doubling sequence ..................................... 732
36.6 An iteration from substitution rules with sign ................................................ 736
36.7 Iterations related to the sum of digits ........................................................... 736
36.8 Iterations related to the binary Gray code ..................................................... 738
36.9 A function that encodes the Hilbert curve ...................................................... 744
36.10 Sparse variants of the inverse ........................................................................ 747
36.11 An iteration related to the Fibonacci numbers .............................................. 750
36.12 Iterations related to the Pell numbers ............................................................ 754

### V Algorithms for finite fields

37 Modular arithmetic and some number theory ................................................. 763

37.1 Implementation of the arithmetic operations .................................................... 763
37.2 Modular reduction with structured primes ....................................................... 769
37.3 The sieve of Eratosthenes ............................................................................. 769
37.4 The order of an element .............................................................................. 771
37.5 Prime modulus: the field $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p = \text{GF}(p)$ .................. 772
37.6 Composite modulus: the ring $\mathbb{Z}/m\mathbb{Z}$ .................................................... 773
37.7 The Chinese Remainder Theorem (CRT) ....................................................... 778
37.8 Quadratic residues ..................................................................................... 780
37.9 Computation of a square root modulo $m$ ..................................................... 783
37.10 The Rabin-Miller test for compositeness ...................................................... 785
37.11 Proving primality ..................................................................................... 791
37.12 Complex moduli: $\text{GF}(p^2)$ .................................................................. 803
37.13 Solving the Pell equation ........................................................................... 811
37.14 Multiplication of hypercomplex numbers * ................................................ 814

### 38 Binary polynomials

38.1 The basic arithmetical operations .................................................................. 819
38.2 Multiplication for polynomials of high degree ................................................. 824
38.3 Modular arithmetic with binary polynomials ................................................... 830
38.4 Irreducible polynomials .............................................................................. 834
38.5 Primitive polynomials ............................................................................. 838
38.6 The number of irreducible and primitive polynomials .................................... 840
38.7 Transformations that preserve irreducibility ................................................... 842
38.8 Self-reciprocal polynomials ....................................................................... 843
38.9 Irreducible and primitive polynomials of special forms * .............................. 846
38.10 Generating irreducible polynomials from Lyndon words .............................. 852
38.11 Irreducible and cyclotomic polynomials * .................................................... 854
38.12 Factorization of binary polynomials ............................................................ 855

### 39 Shift registers

39.1 Linear feedback shift registers (LFSR) .......................................................... 861
39.2 Galois and Fibonacci setup ....................................................................... 864
39.3 Error detection by hashing: the CRC ............................................................. 865
39.4 Generating all revbin pairs ...................................................................... 870
39.5 The number of m-sequences and De Bruijn sequences ................................. 870
39.6 Auto correlation of m-sequences ................................................................ 872

[fxtbook draft of 2008-August-17]
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>39.7</td>
<td>Feedback carry shift register (FCSR)</td>
<td>873</td>
</tr>
<tr>
<td>39.8</td>
<td>Linear hybrid cellular automata (LHCA)</td>
<td>875</td>
</tr>
<tr>
<td>39.9</td>
<td>Additive linear hybrid cellular automata</td>
<td>880</td>
</tr>
<tr>
<td>40</td>
<td>Binary finite fields: GF($2^n$)</td>
<td>885</td>
</tr>
<tr>
<td>40.1</td>
<td>Arithmetic and basic properties</td>
<td>885</td>
</tr>
<tr>
<td>40.2</td>
<td>Minimal polynomials</td>
<td>891</td>
</tr>
<tr>
<td>40.3</td>
<td>Fast computation of the trace vector</td>
<td>893</td>
</tr>
<tr>
<td>40.4</td>
<td>Solving quadratic equations</td>
<td>895</td>
</tr>
<tr>
<td>40.5</td>
<td>Representation by matrices *</td>
<td>898</td>
</tr>
<tr>
<td>40.6</td>
<td>Representation by normal bases</td>
<td>899</td>
</tr>
<tr>
<td>40.7</td>
<td>Conversion between normal and polynomial representation</td>
<td>909</td>
</tr>
<tr>
<td>40.8</td>
<td>Optimal normal bases (ONB)</td>
<td>910</td>
</tr>
<tr>
<td>40.9</td>
<td>Gaussian normal bases</td>
<td>913</td>
</tr>
</tbody>
</table>

**A** The electronic version of the book | 921  |
**B** Machine used for benchmarking | 923  |
**C** The pseudo language Sprache | 925  |
**D** The pari/gp language | 927  |
**Bibliography** | 935  |
**Index** | 953  |
Important remarks about this document

This is a draft of a book about selected algorithms. The audience in mind are programmers who are interested in the treated algorithms and actually want to create and understand working and reasonably optimized code.

The style varies somewhat which I do not consider bad per se: While some topics (as fast Fourier transforms) need a clear and explicit introduction others (like the bit wizardry chapter) seem to be best presented by basically showing the code with just a few comments.

The pseudo language Sprache is used when I see a clear advantage to do so, mainly when the corresponding C++ does not appear to be self explanatory. Larger pieces of code are presented in C++. C programmers do not need to be shocked by the ‘++‘ as only a rather minimal set of the C++ features is used. Some of the code, especially in part 3 (Arithmetical algorithms), is given in the pari/gp language as the use of other languages would likely bury the idea in technicalities.

A printable version of this book will always stay online for free download. The referenced sources are online as part of FXT (fast transforms and low level routines [20]) and hfloat (high precision floating point algorithms [21]).

The reader is welcome to criticize and suggest improvements. Please name the draft version (date) with your feedback! This version is of 2008-August-17. Note that you can copy and paste from the PDF and DVI versions. Thanks go to those1 who helped to improve this document so far!

In case you want to cite this document, please avoid referencing individual chapters or sections as their numbers (and titles) may change.

Enjoy reading!

Legal matters

This book is copyright © by its author, Jörg Arndt. Redistributing or selling this book in printed or in electronic form is prohibited. This book must not be mirrored on the Internet. Using this book as promotional material is prohibited. CiteSeer (http://citeseer.ist.psu.edu/cs/ and its mirrors) is allowed to keep a copy of this book in its database.

1 In particular Igal Aharonovich, Max Alekseyev, Marcus Blackburn, Nathan Bullock, Dominique Delande, Torsten Finke, Sean Furlong, Almaz Gaifullin, Pedro Gimeno, Alexander Glyzov, Andreas Grünbacher, Christoph Haenel, Tony Hardie-Bick, László Hars, Jeff Hurchalla, Gideon Klimer, Richard B. Kreckel, Gál László, Dirk Lattermann, Avery Lee, Brent Lehman, Marc Lehmann, Paul C. Leopardi, John Lien, Mirko Liss, Fred Lunnon, Johannes Middeke, Doug Moore, Andrew Morris, David Nalepa, Matthew Oliver, Michael Roby Wetherfield, Jim White, Vinnie Winkler, John Youngquist, Rui Zhang, and Paul Zimmermann.
“Why make things difficult, when it is possible to make them cryptic and totally illogical, with just a little bit more effort?”

– Aksel Peter Jørgensen
Part I

Low level algorithms
Chapter 1

Bit wizardry

We present low-level functions that operate on the bits of a binary word. It is sometimes not obvious what a particular function is good for. However, if you happen to have an application for it then the program using it may run significantly faster.

The C-type \texttt{unsigned long} is abbreviated as \texttt{ulong} as defined in \cite{fxt:ftxtypes.h}. It is assumed that \texttt{BITS\_PER\_LONG} reflects the size of an \texttt{unsigned long}. It is defined in \cite{fxt:bits/bitsperlong.h} and usually equals the machine word size: it equals 32 on 32-bit architectures and 64 on 64-bit machines. Further, the quantity \texttt{BYTES\_PER\_LONG} shall reflect the number of bytes in a machine word: it equals \texttt{BITS\_PER\_LONG} divided by eight. For some functions it is assumed that \texttt{long} and \texttt{ulong} have the same number of bits.

Many functions will only work on machines that use two’s complement.

The examples of assembler code are for the x86 and the AMD64 architecture. They should be simple enough to be understandable for readers who know assembler for any CPU.

1.1 Trivia

1.1.1 Little endian versus big endian

The order in which the bytes of an integer are stored in memory can start with the least significant byte (\textit{little endian} machine) or with the most significant byte (\textit{big endian} machine). The hexadecimal number \texttt{0x0D0C0B0A} will be stored in the following manner when memory addresses grow from left to right:

\begin{verbatim}
addr:  z  z+1  z+2  z+3  // big endian
mem:  0D  0C  0B  0A  // big endian
mem:  0A  0B  0C  0D  // little endian
\end{verbatim}

The difference becomes visible when you cast pointers. Let \texttt{V} be the 32-bit integer with the value above. Then the result of \texttt{char c = *(char *)&V;} will be \texttt{0x0A} (value modulo 256) on a little endian machine but \texttt{0xOD} (value divided by \texttt{2^{24}}) on a big endian machine. Though friends of big endian way sometimes refer to little endian as ‘wrong endian’, the desired result of the shown pointer cast is much more often the modulo operation.

Whenever words are serialized into bytes, for example with transfer over a network or to a disk one will need two code versions, one for big endian, and one for small endian machines. The C-type \texttt{union} (with words and bytes) may also require separate treatment for big and little endian architectures.
1.1.2 Size of pointer is size of long

On sane architectures a pointer fits into a type long integer. When programming for a 32-bit architecture (where the size of int and long coincide) casting pointers to integers (and back) will work. The same code will fail on 64-bit machines. If you have to cast pointers to an integer type, cast them to long. For portable code better avoid casting pointers to integer types.

1.1.3 Shifts and division

With two’s complement arithmetic (that is: on likely every computer you’ll ever touch) division and multiplication by powers of two is right and left shift, respectively. This is true for unsigned types and for multiplication (left shift) with signed types. Division with signed types rounds toward zero, as one would expect, but right shift is a division (by a power of two) that rounds to minus infinity:

```c
int a = -1;
int c = a >> 1;  // c == -1
int d = a / 2;  // d == 0
```

The compiler still uses a shift instruction for the division, but with a ‘fix’ for negative values:

```c
9:test.cc @ int foo(int a)
10:test.cc @ {
  285 0003 8B442410 movl 16(%esp),%eax // move argument to %eax
  289 0007 89C1 movl %eax,%ecx
  290 0009 D1F9 sarl $1,%ecx // fix: %ecx=(%ecx<0?1:0)
  293 000b 89C2 movl %eax,%edx
  294 000d C1EA1F shrl $31,%edx // fix: add one if a<0
  295 0010 01D0 addl %edx,%eax // fix: add one if a<0
  296 0012 D1F8 sarl $1,%eax
```

For unsigned types the shift would suffice. One more reason to use unsigned types whenever possible.

The assembler listing was generated from C code via the following commands:

```sh
# create assembler code:
c++ -S -fverbose-asm -g -O2 test.cc -o test.s
# create asm interlaced with source lines:
as -alhnd test.s > test.lst
```

There are two types of right shifts: a so-called logical and an arithmetical shift. The logical version (shrl in the above fragment) always fills the higher bits with zeros, corresponding to division of unsigned types. The arithmetical shift (sarl in the above fragment) fills in ones or zeros, according to the most significant bit of the original word.

Computing remainders modulo a power of two with unsigned types is equivalent to a bit-and:

```c
ulong a = b % 32;  // == b & (32-1)
```

All of the above is done by the compiler’s optimization wherever possible.

Division by (compile time) constants can be replaced by multiplications and shifts. The magic machinery inside the compiler does it for you. A division by the constant 10 is compiled to:

```c
5:test.cc @ ulong foo(ulong a)
6:test.cc @ {
  7:test.cc @ ulong b = a / 10;
  290 0000 8B442404 movl 4(%esp),%eax
  291 0004 F7250000 mull .LC33 // value == 0xcccccccc
  292 000a 89D0 movl %edx,%eax
  293 000c C1EB03 shrl %s3,%eax
```

Thereby it is sometimes reasonable to have separate code branches with explicit special values. Similar optimizations can be used for the modulo operation if the modulus is a compile time constant. For example, using modulus 10,000:

```c
8:test.cc @ ulong foo(ulong a)
9:test.cc @ {
  53 0000 8B4C2404 movl 4(%esp),%ecx
```

[fxtbook draft of 2008-August-17]
1.1: Trivia

```
10: test.cc  0  ulong b = a % 10000;
57 0004 89CB  movl %ecx,%eax
58 0006 F7E00000  mull .LC0  // value == 0xd1b71759
59 000c 89D0  movl %edx,%eax
60 000e C1E0  shrw $13,%eax
61 0010 88C0  imull $10000,%eax,%eax
62 0012 29C4  subl %eax,%ecx
63 0014 89C8  movl %ecx,%eax
```

Algorithms to replace divisions by a constant by multiplications and shifts are given in [135], see also [270].

We note that the C standard leaves the behavior of a right shift of a signed integer as ‘implementation-defined’. The described behavior (that a negative value remains negative after right shift) is the de facto standard of all modern C compilers.

1.1.4 A pitfall (two’s complement)

```
Figure 1.1-A: With two’s complement there is one nonzero value that is its own negative.
```

In two’s complement zero is not the only number that is equal to its negative. The value with just the highest bit set (the most negative value) also has this property. Figure 1.1-A (the output of [FXT: bits/gotcha-demo.cc]) shows the situation for words of sixteen bits. Now we know why innocent looking code like

```c
if ( x<0 ) x = -x;
// assume x positive here (WRONG!)
```

can simply fail.

1.1.5 Another pitfall (shifts in the C-language)

A shift by more than BITS_PER_LONG−1 is undefined by the C-standard. Therefore the following function can fail if k is zero:

```
inline ulong first_comb(ulong k)
// Return the first combination of (i.e. smallest word with) k bits,
// i.e. 00..001111..1 (k low bits set)
{
ulong t = ~0UL >> ( BITS_PER_LONG - k );
return t;
}
```

Compilers usually emit just a shift instruction which on certain CPUs does not give zero if the shift is equal to or greater than BITS_PER_LONG. This is why the line

```c
if ( k==0 ) t = 0; // shift with BITS_PER_LONG is undefined
```

[fxtbook draft of 2008-August-17]
has to be inserted just before the `return` statement.

### 1.1.6 Shortcuts

To test whether at least one of \(a\) and \(b\) equals zero use `if ( !(a && b) )`. This works for signed and unsigned integers. Checking whether both are zero can be done using `if ( (a|b)==0 )`. This obviously generalizes for several variables as `if ( (a|b|c|..|z)==0 )`. Test whether exactly one of two variables is zero using `if ( (!a) ^ (!b) )`.

### 1.1.7 Toggling between values

In order to toggle an integer \(x\) between two values \(a\) and \(b\) use:

```plaintext
pre-calculate: t = a ^ b;
toggle: x ^= t; // a <-> b
```

the equivalent trick for floating point types is

```plaintext
pre-calculate: t = a + b;
toggle: x = t - x;
```

Here an overflow could occur with \(a\) and \(b\) in the legal range (but both close to overflow). This should, however, not happen with sane programs.

### 1.1.8 Next or previous even or odd value

Compute the next or previous even or odd value via [FXT: bits/evenodd.h]:

```plaintext
1 static inline ulong next_even(ulong x) { return x+2-(x&1); }
2 static inline ulong prev_even(ulong x) { return x-2+(x&1); }
3 static inline ulong next_odd(ulong x) { return x+1+(x&1); }
4 static inline ulong prev_odd(ulong x) { return x-1-(x&1); }
```

The following functions return the unmodified argument if it has the required property, else the nearest such value:

```plaintext
1 static inline ulong next0_even(ulong x) { return x+(x&1); }
2 static inline ulong prev0_even(ulong x) { return x-(x&1); }
3 static inline ulong next0_odd(ulong x) { return x+1-(x&1); }
4 static inline ulong prev0_odd(ulong x) { return (x-1)|1; }
```

Pedro Gimeno gives [priv.comm.] the following optimized versions:

```plaintext
1 static inline ulong next_even(ulong x) { return (x|1)+1; }
2 static inline ulong prev_even(ulong x) { return (x-1)&~1; }
3 static inline ulong next_odd(ulong x) { return (x+1)|1; }
4 static inline ulong prev_odd(ulong x) { return (x&~1)-1; }
```

### 1.1.9 Integer versus float multiplication

The floating point multiplier gives the highest bits of the product. Integer multiplication gives the result modulo \(2^b\) where \(b\) is the number of bits of the integer type used. As an example we square the number 1010101 using a 32-bit integer type and floating point types with 24-bit and 53-bit mantissa:
1.1: Trivia

\[
a = 11111111\ \\
a\cdot a = 12345678987654321 \quad \text{// true result}
\]
\[
a\cdot a = 1653732529 \quad \text{// result with 32-bit integer multiplication}
\]
\[
(a\cdot a)\% (2^{32}) = 1653732529 \quad \text{// ... which is modulo (2^{bits\_per\_int})}
\]
\[
a\cdot a = 1.2345679481405440e+16 \quad \text{// result with float multiplication (24 bit mantissa)}
\]
\[
a\cdot a = 1.2345678987654320e+16 \quad \text{// result with float multiplication (53 bit mantissa)}
\]

1.1.10 Double precision float to signed integer conversion

Conversion of double precision floats that have a 53-bit mantissa to signed integers via [13, p.52-53]

```c
#define DOUBLE2INT(i, d) { double t = ((d) + 6755399441055744.0); i = *((int *)(&t)); }
double x;
int i = 123;
DOUBLE2INT(i, x);
```
can be a faster alternative to

```c
double x = 123.0;
int i;
i = x;
```
The constant used is 6755399441055744 = 2^{52} + 2^{51}. The method is machine dependent as it relies on the binary representation of the floating point mantissa. Here it is assumed that, firstly, the floating point number has a 53-bit mantissa with the most significant bit (that is always one with normalized numbers) omitted, and secondly, the address of the number points to the mantissa.

1.1.11 Optimization considerations

Never assume that some code is the ‘fastest possible’, there always another trick that can still improve performance. Many factors can have an influence on performance, like number of CPU registers or cost of branches. Code that performs well on one machine might perform badly on another. The old trick to swap variables without using a temporary

```c
// a=0, b=0  a=0, b=1  a=1, b=0  a=1, b=1
a ^= b; // 0 0 1 1 1 0 0 1
b ^= a; // 0 0 1 0 1 1 0 1
a ^= b; // 0 0 1 0 0 1 1 1
```
equivalent to:

```c
tmp = a; a = b; b = tmp;
```
is pretty much out of fashion today. However in some specific context (like extreme register pressure) it may be the way to go.

The only way to find out which version of a function is faster is to actually do profiling (timing). The performance does depend on the stream of instructions before the machine code (we assume that all of these low-level functions get inlined). Studying the generated CPU instructions does help to understand what is going on but can never replace profiling.

The code surrounding a specific function can have a massive impact on performance. That is, benchmarks for just the isolated routine can only give a rough indication. Profile your application and also test whether the second best (when isolated) routine is the fastest.

Never just replace the unoptimized version of some code fragment when introducing a streamlined one. Keep the original in the source. In case something nasty happens (think of low level software failures when porting to a different platform) you’ll be very thankful for the chance to temporarily use the slow but correct version.

Study the optimization recommendations for your CPU (like [13] for the AMD64). It doesn’t hurt to see the corresponding documentation also for other architectures.
Proper documentation is an absolute must for optimized code, just assume that nobody will be able to read and understand it from the supplied source alone. The experience of not being able to understand code oneself has written some time ago helps a lot in this matter.

More techniques for optimization are given in section 5.4 on page 172.

### 1.2 Operations on individual bits

#### 1.2.1 Testing, setting, and deleting bits

The following functions should be self explanatory. Following the spirit of the C language there is no check whether the indices used are out of bounds. That is, if any index is greater or equal to BITS_PER_LONG, the result is undefined [FXT: bits/bittest.h]:

```c
inline ulong test_bit(ulong a, ulong i)
// Return zero if bit[i] is zero, 
// else return one-bit word with bit[i] set.
{
    return (a & (1UL << i));
}
```

The following version returns either zero or one:

```c
static inline bool test_bit01(ulong a, ulong i)
// Return whether bit[i] is set.
{
    return ( 0 != test_bit(a, i) );
}
```

Functions for setting, clearing, and changing a bit are:

```c
inline ulong set_bit(ulong a, ulong i)
// Return a with bit[i] set.
{
    return (a | (1UL << i));
}
```

```c
inline ulong delete_bit(ulong a, ulong i)
// Return a with bit[i] cleared.
{
    return (a & ~(1UL << i));
}
```

```c
inline ulong change_bit(ulong a, ulong i)
// Return a with bit[i] changed.
{
    return (a ^ (1UL << i));
}
```

#### 1.2.2 Copying a bit

In order to copy a bit from one position to another we generate a one exactly if the bits at the two positions differ. Then an XOR changes the target bit if needed [FXT: bits/bitcopy.h]:

```c
inline ulong copy_bit(ulong a, ulong isrc, ulong idst)
// Copy bit at [isrc] to position [idst].
// Return the modified word.
{
    ulong x = ((a>>isrc) ^ (a>>idst)) & 1; // one if bits differ
    a ^= (x<<idst); // change if bits differ
}
```

The situation is more tricky if the bit positions are given as (one bit) masks:

```c
inline ulong mask_copy_bit(ulong a, ulong msrc, ulong mdst)
// Copy bit according at src-mask (msrc)
// to the bit according to the dest-mask (mdst).
// Both msrc and mdst must have exactly one bit set.
```
1.3: Operations on low bits or blocks of a word

```c
{  ulong x = mdst;
  if ( msrc & a ) x = 0; // zero if source bit set
  x ^= mdst; // ==mdst if source bit set, else zero
  a &= ~mdst; // clear dest bit
  a |= x;
  return a;
}
```

The compiler generates branch-free code as the conditional assignment is compiled to a `cmov` (conditional move) assembler instruction. If one or both masks have several bits set the routine will set all bits of `mdst` if any of the bits in `msrc` is one else clear all bits of `mdst`.

### 1.2.3 Swapping two bits

A function to swap two bits of a word is [FXT: `bits/bitswap.h`]:

```c
static inline ulong bit_swap(ulong a, ulong k1, ulong k2)
// Return a with bits at positions [k1] and [k2] swapped.
// k1==k2 is allowed (a is unchanged then)
{  ulong x = ((a>>k1) ^ (a>>k2)) & 1; // one if bits differ
  a ^= (x<<k2); // change if bits differ
  a ^= (x<<k1); // change if bits differ
  return a;
}
```

When it is known that the bits do have different values then the following routine can be used:

```c
static inline ulong bit_swap_01(ulong a, ulong k1, ulong k2)
// Return a with bits at positions [k1] and [k2] swapped.
// Bits must have different values (!)
// (i.e. one is zero, the other one)
// k1==k2 is allowed (a is unchanged then)
{  return a ^ ( (1UL<<k1) ^ (1UL<<k2) );
}
```

### 1.3 Operations on low bits or blocks of a word

The underlying idea of functions that operate on the lowest set bit is that addition and subtraction of 1 always changes a burst of bits at the lower end of the word. The following functions are given in [FXT: `bits/bitlow.h`].

#### 1.3.1 Isolating, setting, and deleting the lowest one

Isolation of the lowest set bit is achieved via

```c
static inline ulong lowest_one(ulong x)
// Return word where only the lowest set bit in x is set.
// Return 0 if no bit is set.
{  return x & -x; // use: -x == ~x + 1
}
```

The lowest zero (unset bit) of some word `x` is then trivially isolated using the equivalent of `lowest_one(~x)`:

```c
static inline ulong lowest_zero(ulong x)
// Return word where only the lowest unset bit in x is set.
// Return 0 if all bits are set.
{  x = ~x;
  return x & ~x;
}
```
Alternatively, one can use either of

\[
\begin{align*}
\text{return } & (x \oplus (x+1)) \& \sim x; \\
\text{return } & ((x \oplus (x+1)) >> 1 ) + 1;
\end{align*}
\]

The sequence of returned values for \(x = 0, 1, \ldots\) is the binary ruler function, the highest power of two that divides \(x + 1\):

\[
\begin{array}{c|c}
\text{x:} & \text{lowest_zero(x)} \\
\hline
0: & \ldots \ldots \ldots 1 \ldots 1 \\
1: & \ldots \ldots \ldots 1 \ldots 1 \\
2: & \ldots \ldots \ldots 1 \ldots 1 \\
3: & \ldots \ldots \ldots 1 \ldots 1 \\
4: & \ldots \ldots \ldots 1 \ldots 1 \\
5: & \ldots \ldots \ldots 1 \ldots 1 \\
6: & \ldots \ldots \ldots 1 \ldots 1 \\
7: & \ldots \ldots \ldots 1 \ldots 1 \\
8: & \ldots \ldots \ldots 1 \ldots 1 \\
9: & \ldots \ldots \ldots 1 \ldots 1 \\
10: & \ldots \ldots \ldots 1 \ldots 1 \\
\end{array}
\]

Clearing the lowest set bit in a word can be achieved via

\[
\begin{align*}
\text{static inline ulong delete_lowest_one(ulong x) \\
// Return word where the lowest bit set in x is cleared. \\
// Return 0 for input == 0. \\
{ \\
\quad \text{return} \ x \ & (x-1); \\
}\}
\]

while setting the lowest unset bit is done by

\[
\begin{align*}
\text{static inline ulong set_lowest_zero(ulong x) \\
// Return word where the lowest unset bit in x is set. \\
// Return ~0 for input == ~0. \\
{ \\
\quad \text{return} \ x \ | \ (x+1); \\
}\}
\]

### 1.3.2 Computing the index of the lowest one

Extracting the index (position) of the lowest bit is easy when the corresponding assembler instruction is used [FXT: bits/bitasm-amd64.h]:

\[
\begin{align*}
\text{static inline ulong asm_bsf(ulong x) \\
// Bit Scan Forward \\
{ \\
\quad \text{asm} \ ("bsfq \%0, \%0" : \"=r" \ (x) : \"0" \ (x)); \\
\quad \text{return} \ x; \\
}\}
\end{align*}
\]

Without the assembler instruction an algorithm that involves proportional \(\log_2(\text{BITS\_PER\_LONG})\) operations can be used. The function can be implemented as follows (suggested by Nathan Bullock [priv.comm.], 64-bit version):

\[
\begin{align*}
\text{static inline ulong lowest_one_idx(ulong x) \\
// Return index of lowest bit set. \\
// Examples: \\
// ***1 --\> 0 \\
// **10 --\> 1 \\
// *100 --\> 2 \\
\hline
\text{// Return 0 (also) if no bit is set.} \\
{ \\
\quad \text{ulong} \ r = 0; \\
\quad \text{x} \ &= \ -x; \ // \text{isolate lowest bit} \\
\quad \text{if} \ ( \ x \ & \ 0xffffffff00000000UL ) \ r += 32; \\
\quad \text{if} \ ( \ x \ & \ 0xffffffff00000000UL ) \ r += 16; \\
\quad \text{if} \ ( \ x \ & \ 0xffff000000000000UL ) \ r += 8; \\
\quad \text{if} \ ( \ x \ & \ 0xff0f0f0f0f0f0f0fUL ) \ r += 4; \\
\quad \text{if} \ ( \ x \ & \ 0xc000000000000000UL ) \ r += 2; \\
\quad \text{if} \ ( \ x \ & \ 0xaaaaaaaaaaaaaaaaUL ) \ r += 1; \\
\text{#endif} \ // BIT\_USE\_ASM \\
\quad \text{return} \ r; \\
\}
\end{align*}
\]
The function returns zero for two inputs, one and zero. If a special value for the input zero is needed then add a statement like

```c
if ( 1>=x ) return x-1; // 0 if 1, ~0 if 0
```

as first line of the function.

The following function returns the parity of the index of the lowest set bit in a binary word

```c
static inline ulong lowest_one_idx_parity(ulong x)
{
    x &= -x; // isolate lowest bit
    return 0 != (x & 0xaaaaaaaaaaaaaaaaUL);
}
```

The sequence of values for \( x = 0, 1, 2, \ldots \) is

```
0010001010001000100010001000100010001010100010...
```

This is the complement of the period-doubling sequence, entry A035263 of [245]. See section 36.5.1 on page 733 for the connection to the towers of Hanoi puzzle.

### 1.3.3 Isolating blocks of zeros or ones at the low end

Isolate the burst of low ones as follows:

```c
static inline ulong low_ones(ulong x)
// Return word where all the (low end) ones are set.
// Example: 01011011 --> 00000011
// Return 0 if lowest bit is zero:
// 10110110 --> 0
{
    x = ~x;
    x &= -x;
    --x;
    return x;
}
```

The isolation of the low zeros is slightly cheaper:

```c
static inline ulong low_zeros(ulong x)
// Return word where all the (low end) zeros are set.
// Example: 01011000 --> 00000111
// Return 0 if all bits are set.
{
    x &= -x;
    --x;
    return x;
}
```

Isolation of the lowest block of ones (which may have zeros to the right of it) can be achieved via:

```c
static inline ulong lowest_block(ulong x)
// Isolate lowest block of ones.
// e.g.: 
// x = *****011100
// l = 0000001100
// y = 0000011100
// ret = 0000011100
// x = *****011100
// y = 0000011100
// ret = 0000011100
{  
    ulong l = x & -x; // lowest bit
    ulong y = x + l;
    x ^= y;
    return x & (x>>1);
}
```

### 1.3.4 Creating a transition at the lowest one

Occasionally one wants to set a rising or falling edge at the position of the lowest bit:
1.4 Extraction of ones, zeros, or blocks near transitions

We give functions for the creation or extraction of bit-blocks, and the isolation of values near transitions. The term ‘one’ is used for a set bit, and ‘zero’ for an unset bit. A transition is a place where adjacent bits have different values. A block is a group of adjacent bits of the same value.

1.4.1 Creating blocks of ones

The following functions are given in [FXT: bits/bitblock.h].

```c
static inline ulong bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
  ulong x = (1UL<<n) - 1;
  return x << p;
}
```

A version with indices wrapping around is

```c
static inline ulong cyclic_bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// The result is possibly wrapped around the word boundary.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
  ulong x = (1UL<<n) - 1;
  return (x<<p) | (x>>(BITS_PER_LONG-p));
}
```

1.4.2 Finding isolated ones or zeros

For the following functions [FXT: bits/bit-isolate.h]. We assume that the outside bits are all zero

```c
static inline ulong single_ones(ulong x)
// Return word with only the isolated ones of x set.
{
  return x & ~( (x<<1) | (x>>1) );
}
```

```c
static inline ulong single_zeros(ulong x)
// Return word with only the isolated zeros of x set.
{
  return single_ones(~x);
}
```
1.4: Extraction of ones, zeros, or blocks near transitions

1.4.3 Isolating single ones or zeros at the word boundary

1 static inline ulong border_ones(ulong x)
2 // Return word where only those ones of x are set that lie next to a zero.
3 {
4    return x & ~( (x<<1) & (x>>1) );
5 }

1 static inline ulong border_values(ulong x)
2 // Return word where those bits of x are set that lie on a transition.
3 {
4    ulong g = x ^ (x>>1);
5    g |= (g<<1);
6    return g | (x & 1);
7 }

1.4.4 Isolating transitions

1 static inline ulong high_border_ones(ulong x)
2 // Return word where only those ones of x are set that lie right to (i.e. in the next lower bin of) a zero.
3 {
4    return x & ( x ^ (x>>1) );
5 }

1 static inline ulong low_border_ones(ulong x)
2 // Return word where only those ones of x are set that lie left to (i.e. in the next higher bin of) a zero.
3 {
4    ulong t = x & ( (x<<1) ^ (x>>1) ); // block_border_ones()
5    return t & (x>>1);
6 }

1.4.5 Isolating ones or zeros at block boundaries

1 static inline ulong block_border_ones(ulong x)
2 // Return word where only those ones of x are set that are at the border of a block of at least 2 bits.
3 {
4    return x & ( (x<<1) ^ (x>>1) );
5 }

1 static inline ulong low_block_border_ones(ulong x)
2 // Return word where only those bits of x are set that are at left of a border of a block of at least 2 bits.
3 {
4    ulong t = x & ( (x<<1) ^ (x>>1) ); // block_border_ones()
5    return t & (x>>1);
6 }

1 static inline ulong high_block_border_ones(ulong x)
2 // Return word where only those bits of x are set that are at right of a border of a block of at least 2 bits.
3 {
4    ulong t = x & ( (x<<1) ^ (x>>1) ); // block_border_ones()
5    return t & (x<<1);
6 }

1 static inline ulong block_ones(ulong x)
2 // Return word where only those bits of x are set that are part of a block of at least 2 bits.
3 {
4    return x & ( (x<<1) | (x>>1) );
5 }
1.4.6 Isolating the interior of bit blocks

```cpp
static inline ulong block_values(ulong x) {
  // Return word where only those bits of x are set
  // that do not lie next to an opposite value.
  return ~single_values(x);
}

static inline ulong interior_ones(ulong x) {
  // Return word where only those bits of x are set
  // that do not have a zero to their left or right.
  return x & ( (x<<1) & (x>>1) );
}

static inline ulong interior_values(ulong x) {
  // Return word where only those values of x are set
  // that do have a transitions on both sides.
  return ~border_values(x);
}
```

1.5 Computing the index of a single set bit

In the function `lowest_one_idx()` given in section 1.3.2 on page 10 we first isolated the lowest one of a word `x` by first setting `x&=-x`. At this point, `x` contains just one set bit (or `x==0`). The following lines in the routine implement an algorithm that computes the index of the only bit set. This section gives some alternative techniques to compute the index of the one in a single-bit word.

1.5.1 Cohen’s trick

A nice trick is presented in [91]: for `N`-bit words find a number `m` such that all powers of two are different modulo `m`. That is, the order of two modulo `m` must be greater or equal to `N`. We use a table `mt[]` of size `m` that contains the powers of two: `mt[(2**j) mod m] = j` for `j > 0`. To look up the index of a one-bit-word `x` it is reduced modulo `m` and `mt[x]` is returned.

```
const ulong m = 11;  // the modulus
ulong mt[m+1];
static void mt_setup() {
  k = 0 1 2 3 4 5 6 7
  mt[k]= 0 0 1 2 4 8 9 7
  x= ......1 = 1 x % m= 1 ==> lookup = 0
  x= .......1 = 2 x % m= 2 ==> lookup = 1
  x= ......1.. = 4 x % m= 4 ==> lookup = 2
  x= ......1... = 8 x % m= 8 ==> lookup = 3
  x= ......1.... = 16 x % m= 5 ==> lookup = 4
  x= ......1..... = 32 x % m= 10 ==> lookup = 5
  x= ......11.... = 64 x % m= 9 ==> lookup = 6
  x= ......11..... = 128 x % m= 7 ==> lookup = 7
```

Figure 1.5-A: Determination of the position of a single bit with 8-bit words.

We demonstrate the method for `N = 8` where `m = 11` is the smallest number with the required property. The setup routine for the table is

```cpp
const ulong m = 11;  // the modulus
ulong mt[m+1];
static void mt_setup() {
  ...
```
1.5: Computing the index of a single set bit

```
mt[0] = 0; // special value for the zero word
ulong t = 1;
for (ulong i=1; i<m; ++i)
{
    mt[t] = i-1;
    t *= 2;
    if ( t>=m ) t -= m; // modular reduction
}
```

The entry in \( mt[0] \) will be accessed when the input is the zero word. One can use a special value that will be returned for input zero. Here we simply used zero in order to always have the same return value as with \( \text{lowest_one_idx}() \). The computation of the index can then be achieved by

```
inline ulong m_lowest_one_idx(ulong x)
{
    x &= -x; // isolate lowest bit
    x %= m; // power of two modulo m
    return mt[x]; // lookup
}
```

The code is given in the demo [FXT: bits/modular-lookup-demo.cc], the output with \( N = 8 \) (edited for size) is shown in figure 1.5-A. The following moduli \( m(N) \) can be used for \( N \)-bit words:

\[
N: 4 8 16 32 64 128 256 512 1024
m: 5 11 19 37 67 131 269 523 1061
\]

The modulus \( m(N) \) is the smallest prime greater than \( N \) such that 2 is a primitive root modulo \( m(N) \).

1.5.2 Using De Bruijn sequences

The following method (given in [185]) is even more elegant, it uses binary De Bruijn sequences of size \( N \). A binary De Bruijn sequence of length \( 2^N \) contains all binary words of length \( N \), see section 39.1 on page 861. These are the sequences for 32 and 64 bit, as binary words:

```
#define BITS_PER_LONG 32
const ulong db = 0x4653ADFUL; // == 00000100011001000111011101111
const ulong s = 32-5;
#else
const ulong db = 0x218A392CD3D5DBFUL; // == 0000010001100100011101110111011111
const ulong s = 64-6;
#endif
```

```
db=...1.111 (De Bruijn sequence)  
k = 0 1 2 3 4 5 6 7  
dbt[k] = 0 0 0 0 1 1 1 1  
Lowest bit == 0: x = .......1 db * x = ....1.111 shifted = ........ == 0 ==> lookup = 0  
Lowest bit == 1: x = .......1 db * x = ....1.111 shifted = .......1 == 1 ==> lookup = 1  
Lowest bit == 2: x = .......1 db * x = ....1.111 shifted = .......1 == 2 ==> lookup = 2  
Lowest bit == 3: x = .......1 db * x = ....1.111 shifted = .......1 == 5 ==> lookup = 3  
Lowest bit == 4: x = .......1 db * x = ....1.111 shifted = .......1 == 3 ==> lookup = 4  
Lowest bit == 5: x = .......1 db * x = ....1.111 shifted = .......1 == 7 ==> lookup = 5  
Lowest bit == 6: x = .......1 db * x = ....1.111 shifted = .......1 == 6 ==> lookup = 6  
Lowest bit == 7: x = .......1 db * x = ....1.111 shifted = .......1 == 4 ==> lookup = 7  
```

**Figure 1.5-B:** Computing the position of the single set bit in 8-bit words with a De Bruijn sequence.

Let \( w_i \) be the \( i \)-th sub-word from the left (high end). We create a table such that the entry with index \( w_i \) points to \( i \):

```
ulong db[BITS_PER_LONG];  
static void dbt_setup()
{
    for (ulong i=0; i<BITS_PER_LONG; ++i) dbt[ (db<<i)>>s ] = i;
}
```

The computation of the index involves a multiplication and a table lookup:

```
inline ulong db_lowest_one_idx(ulong x)
{
    x &= -x; // isolate lowest bit
    x %= m; // power of two modulo m
    return mt[x]; // lookup
}
```


```c
x &= -x; // isolate lowest bit
x *= db; // multiplication by a power of two is a shift
x >>= s; // use log_2(BITS_PER_LONG) highest bits
return dbt[x]; // lookup
```

The used sequences must start with at least $\log_2(N) - 1$ zeros because in the line $x *= db$ the word $x$ is shifted (not rotated). The code is given in the demo [FXT: bits/debruijn-lookup-demo.cc], the output with $N = 8$ (edited for size, dots denote zeros) is shown in figure 1.5-B.

### 1.5.3 Using floating point numbers

Floating point numbers are normalized so that the highest bit in the mantissa is one. Therefore if one converts an integer into a float then the position of the highest set bit can be read off the exponent. By isolating the lowest bit before that operation the index can be found by the same trick. However, the conversion between integers and floats is usually slow. Further, the technique is highly machine dependent.

### 1.6 Operations on high bits or blocks of a word

For the functions operating on the highest bit there is not a way as trivial as with the equivalent task with the lower end of the word. With a bit-reverse CPU-instruction available life would be significantly easier. However, almost no CPU seems to have it. The following functions are given in [FXT: bits/bithigh.h].

#### 1.6.1 Isolating the highest one and finding its index

Isolation of the highest set bit is achieved via the bit-scan instruction when it is available [FXT: bits/bitasm-i386.h]:

```c
\textbf{asm\_bar}:

```c
1
2
3
4
5

\textbf{highest\_one\_01edge}:

```c
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

When no bit-scan instruction is available we can use the following auxiliary function:

```c
\textbf{highest\_one\_01edge}:

```c
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

```

so the resulting code is

```c
\textbf{highest\_one}:

```c
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

[fxtbook draft of 2008-August-17]
1.6: Operations on high bits or blocks of a word

```c
#else
    x = highest_one_01edge(x);
    return x ^ (x>>1);
#endif // BITS_USE_ASM
}
```

---

![Binary representation](image)

---

**Figure 1.6-A:** Operations on the highest and lowest bits (and blocks) of a binary word for two different 32-bit input words. Dots denote zeros.

Finding the index of the highest set bit uses the equivalent algorithm as with the lowest set bit:

```c
static inline ulong highest_one_idx(ulong x)
{
    // Return index of highest bit set.
    // Return 0 if no bit is set.
    #if defined BITS_USE_ASM
        return asm_bsr(x);
    #else // BITS_USE_ASM
        ulong r = 0;
        #if BITS_PER_LONG >= 64
            if (x & 0xffffffff00000000UL) { x >>= 32; r += 32; }
        #endif
        if (x & 0xffff0000UL) { x >>= 16; r += 16; }
        if (x & 0x0000ff00UL) { x >>= 8; r += 8; }
        if (x & 0x000000f0UL) { x >>= 4; r += 4; }
        if (x & 0x0000000cUL) { x >>= 2; r += 2; }
        if (x & 0x00000002UL) { r += 1; }
        return r;
    #endif // BITS_USE_ASM
}
```
1.6.2 Isolating the highest block of ones or zeros

Isolation of the high zeros can be achieved with the function

```c
static inline ulong high_zeros(ulong x)
// Return word where all the (high end) zeros are set.
// e.g.: 00011001 --> 11100000
// Returns 0 if highest bit is set:
// 11011001 --> 00000000
{
    x |= x>>1;
    x |= x>>2;
    x |= x>>4;
    x |= x>>8;
    x |= x>>16;
    #if BITS_PER_LONG >= 64
    x |= x>>32;
    #endif
    return ~x;
}
```

The high bits can be isolated using arithmetical right shifts

```c
static inline ulong high_ones(ulong x)
// Return word where all the (high end) ones are set.
// e.g. 11001011 --> 11000000
// Returns 0 if highest bit is zero:
// 01110110 --> 00000000
{
    long y = (long)x;
    y &= y>>1;
    y &= y>>2;
    y &= y>>4;
    y &= y>>8;
    y &= y>>16;
    #if BITS_PER_LONG >= 64
    y &= y>>32;
    #endif
    return (ulong)y;
}
```

If arithmetical shifts are more expensive than unsigned shifts, then use

```c
static inline ulong high_ones(ulong x)
{
    return high_zeros(~x);
}
```

A demonstration of selected functions operating on the highest or lowest bit (or block) of binary words is given in [FXT: bits/bithilo-demo.cc]. A part of the output is shown in figure 1.6-A.

1.7 Functions related to the base-2 logarithm

The following functions are given in [FXT: bits/bit2pow.h]. A function that returns \( \lfloor \log_2(x) \rfloor \) can be implemented using the obvious algorithm:

```c
static inline ulong ld(ulong x)
// Return k so that 2^k <= x < 2^(k+1)
// If x==0 then 0 is returned (!)
{
    ulong k = 0;
    while ( x>>=1 ) { ++k; }
    return k;
}
```

The result is the same as returned by `highest_one_idx()`, so one can use

```c
static inline ulong ld(ulong x)
{
    return highest_one_idx(x);
}
```
The bit-wise algorithm can be faster if the average result is known to be small.

The function `one_bit_q()` can be used to determine whether its argument is a power of two:

```c
static inline bool one_bit_q(ulong x)
// Return whether x \in \{1,2,4,8,16,\ldots\}
{
ulong m = x-1;
return (((x^m)>>1) == m);
}
```

The following function does the same except that it returns `true` also for the zero argument:

```c
static inline bool is_pow_of_2(ulong x)
// Return whether x == 0(!) or x == 2**k
{
return !(x & (x-1));
}
```

With FFTs where the length of the transform is often restricted to powers of two the following functions are useful:

```c
static inline ulong next_pow_of_2(ulong x)
// Return x if x=2**k
// else return 2**ceil(log_2(x))
// Exception: returns 0 for x==0
{
if ( is_pow_of_2(x) ) return x;
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
#if BITS_PER_LONG == 64
x |= x >> 32;
#endif
return x + 1;
}
```

```c
static inline ulong next_exp_of_2(ulong x)
// Return k if x=2**k else return k+1.
// Exception: returns 0 for x==0.
{
if ( x <= 1 ) return 0;
return ld(x-1) + 1;
}
```

The following version should be faster if inline assembler is used for `ld()`:

```c
static inline ulong next_pow_of_2(ulong x)
{
if ( is_pow_of_2(x) ) return x;
ulong n = 1UL<<ld(x); // n<x
x |= x >> 1; // n<x
return x + 1;
}
```

### 1.8 Counting bits and blocks of a word

The following functions count the ones in a binary word. They need proportional to \( \log_2(\text{BITS\_PER\_LONG}) \) operations [FXT: bits/bitcount.h]:

```c
static inline ulong bit_count(ulong x)
// Return number of bits set
{
x = (0x55555555UL & x) + (0x55555555UL & (x>> 1)); // 0-2 in 2 bits
x = (0x33333333UL & x) + (0x33333333UL & (x>> 2)); // 0-4 in 4 bits
x = (0x0f0f0f0fUL & x) + (0x0f0f0f0fUL & (x>> 4)); // 0-8 in 8 bits
x = (0x00ff00ffUL & x) + (0x00ff00ffUL & (x>> 8)); // 0-16 in 16 bits
x = (0x0000ffffUL & x) + (0x0000ffffUL & (x>>16)); // 0-31 in 32 bits
return x;
}
```

The underlying idea is to do a search via bit masks. The code can be improved to either
Chapter 1: Bit wizardry

1 \( x = ((x>>1) & 0x55555555UL) + (x & 0x55555555UL); \) // 0-2 in 2 bits
2 \( x = ((x>>2) & 0x33333333UL) + (x & 0x33333333UL); \) // 0-4 in 4 bits
3 \( x = ((x>>4) + x) & 0x0f0f0f0fUL; \) // 0-8 in 4 bits
4 \( x += x>> 8; \) // 0-16 in 8 bits
5 \( x += x>>16; \) // 0-32 in 8 bits
6 return x & 0xff;

or (taken from [11])

1 \( x -= (x>>1) & 0x55555555UL; \)
2 \( x = ((x>>2) & 0x33333333UL) + (x & 0x33333333UL); \)
3 \( x = ((x>>4) + x) & 0x0f0f0f0fUL; \)
4 \( x *= 0x01010101UL; \)
5 return x>>24;

Which of the latter two versions is faster mainly depends on the speed of integer multiplication.

For 64-bit words the masks have to be adapted and one more step must be added (example corresponding to the second variant above):

1 \( x = ((x>>1) & 0x5555555555555555UL) + (x & 0x5555555555555555UL); \) // 0-2 in 2 bits
2 \( x = ((x>>2) & 0x3333333333333333UL) + (x & 0x3333333333333333UL); \) // 0-4 in 4 bits
3 \( x = ((x>>4) + x) & 0x0f0f0f0f0f0f0f0fUL; \) // 0-8 in 4 bits
4 \( x += x>> 8; \) // 0-16 in 8 bits
5 \( x += x>>16; \) // 0-32 in 8 bits
6 \( x += x>>32; \) // 0-64 in 8 bits
7 return x & 0xff;

The following code for 32-bit words (given by Johan Rönblom [priv.comm.]) may be advantageous if loading constants is expensive:

1 inline uint CountBits32(uint a)
2 {
3 uint mask = 0111111111UL; // Note: octal constant
4 a = (a - ((a&~mask)>>1)) - ((a>>2)&mask);
5 a += a>>3;
6 a = (a & 070707) + ((a>>18) & 070707);
7 a *= 010101;
8 return ((a>>12) & 0x3f);
9 }

With a table of 256 bytes, holding the bit-counts of the numbers 0...255, the bits can be counted as follows:

1 ulong bit_count(ulong x)
2 {
3 unsigned char ct = 0;
4 ct += tab[ x & 0xff ]; x >>= 8;
5 ct += tab[ x & 0xff ]; x >>= 8;
6 --snip-- /* BYTES_PER_LONG times */
7 ct += tab[ x & 0xff ];
8 return ct;
9 }

However, while table driven methods tend to excel in synthetic benchmarks they can be very slow if they cause cache misses.

We give a curious method to count the bits of a word of a special form:

1 static inline ulong bit_count_01(ulong x)
2 // Return number of bits in a word
3 // for words of the special form 00...0001...11
4 {
5 ulong ct = 0;
6 ulong a;
7 #if BITS_PER_LONG == 64
8 a = (x & (1UL<<32)) >> (32-5); // test bit 32
9 x >>= a; ct += a;
10 #endif
11 a = (x & (1UL<<16)) >> (16-4); // test bit 16
12 x >>= a; ct += a;
13 a = (x & (1UL<<8)) >> (8-3); // test bit 8
14 x >>= a; ct += a;
15 a = (x & (1UL<<4)) >> (4-2); // test bit 4
16 x >>= a; ct += a;
17 
}
1.8: Counting bits and blocks of a word

1.8.1 Sparse counting

When the (average input) word is known to have only a few bits set the following sparse count variant can be advantageous:

```c
static inline ulong bit_count_sparse(ulong x)
// Return number of bits set.
{
    ulong n = 0;
    while ( x ) { ++n; x &= (x-1); }
    return n;
}
```

The loop will execute once for each set bit. Partial unrolling of the loop should be an improvement for most cases:

```c
ulong n = 0;
do {
    n += (x!=0); x &= (x-1);
    n += (x!=0); x &= (x-1);
    n += (x!=0); x &= (x-1);
    n += (x!=0); x &= (x-1);
} while ( x );
return n;
```

If the number of bits is close to the maximum then use the given routine with the complement:

```c
static inline ulong bit_count_dense(ulong x)
// Return number of bits set. The loop (of bit_count_sparse()) will execute once for each unset bit (i.e. zero) of x.
{
    return BITS_PER_LONG - bit_count_sparse( ~x );
}
```

1.8.2 Counting blocks

The number of bit-blocks in a binary word can be computed by the following function:

```c
static inline ulong bit_block_count(ulong x)
// Return number of bit blocks.
// E.g.:
// ..1..1111...111. -> 3
// ...1..1111...111 -> 3
// ......1....1.1.. -> 3
// ..........111.1111 -> 2
{
    return (x & 1) + bit_count( (x^(x>>1)) ) / 2;
}
```

Similarly, the number of blocks with two or more bits can be counted via:

```c
static inline ulong bit_block_ge2_count(ulong x)
// Return number of bit blocks with at least 2 bits.
// E.g.:
// ..1...1111...111. -> 2
// .........11111....11111 -> 2
{
    return (x & 1) + bit_count( (x^(x>>1)) ) / 2;
}
```
1.8.3 GCC builtins *

Newer versions of the C compiler of the GNU Compiler Collection (GCC [120], starting with version 3.4) offer a function __builtin_popcountl(ulong) that counts the bits of an unsigned long integer.

We list a few such functions, taken from [121]:

```
int __builtin_ffs (unsigned int x)
  Returns one plus the index of the least significant 1-bit of x,
  or if x is zero, returns zero.

int __builtin_clz (unsigned int x)
  Returns the number of leading 0-bits in x, starting at the
  most significant bit position. If x is 0, the result is undefined.

int __builtin_ctz (unsigned int x)
  Returns the number of trailing 0-bits in x, starting at the
  least significant bit position. If x is 0, the result is undefined.

int __builtin_popcount (unsigned int x)
  Returns the number of 1-bits in x.

int __builtin_parity (unsigned int x)
  Returns the parity of x, i.e. the number of 1-bits in x modulo 2.
```

The names of corresponding versions for arguments of type unsigned long are obtained by adding 'l' (ell) to the names.

1.9 Counting bits of many words

```
x[ 0]=11111111 a0=11111111 a1=........ a2=........ a3=........ a4=........
x[ 1]=11111111 a0=........ a1=11111111 a2=........ a3=........ a4=........
x[ 2]=11111111 a0=11111111 a1=11111111 a2=........ a3=........ a4=........
x[ 3]=11111111 a0=........ a1=........ a2=11111111 a3=........ a4=........
x[ 4]=11111111 a0=11111111 a1=........ a2=11111111 a3=........ a4=........
x[ 5]=11111111 a0=........ a1=11111111 a2=11111111 a3=........ a4=........
x[ 6]=11111111 a0=11111111 a1=11111111 a2=11111111 a3=........ a4=........
x[ 7]=11111111 a0=........ a1=........ a2=........ a3=11111111 a4=........
x[ 8]=11111111 a0=11111111 a1=........ a2=........ a3=11111111 a4=........
x[ 9]=11111111 a0=........ a1=........ a2=........ a3=........ a4=11111111
```

**Figure 1.9-A:** Counting the bits of an array (where all bits are set) via vertical addition.

For counting the bits in a long array the technique of *vertical addition* can be useful. For ordinary addition the following relation holds:

\[ a + b = ((a^b) + ((a&b)<<1)) \]

The carry term \((a&b)\) is propagated to the left. We can replace this ‘horizontal’ propagation by a ‘vertical’ one, that is, propagation into another word. The idea can be implemented as follows [FXT: bits/bitcount-v-demo.cc]:

```
ulong bit_count_leq31(const ulong *x, ulong n)
  // Return sum(j=0, n-1, bit_count(x[j]))
  // Must have n<=31
  
  ulong a0=0, a1=0, a2=0, a3=0, a4=0;
```
1.9: Counting bits of many words

for (ulong k=0; k<n; ++k)
{
    ulong cy = x[k];
    { ulong t = a0 & cy; a0 ^= cy; cy = t; }
    { ulong t = a1 & cy; a1 ^= cy; cy = t; }
    { ulong t = a2 & cy; a2 ^= cy; cy = t; }
    { ulong t = a3 & cy; a3 ^= cy; cy = t; }
    { a4 ^= cy; }
    // [ PRINT x[k], a0, a1, a2, a3, a4 ]
}

ulong b = bit_count(a0);
b += (bit_count(a1)<<1);
b += (bit_count(a2)<<2);
b += (bit_count(a3)<<3);
b += (bit_count(a4)<<4);
return b;
}

Each of the macros \texttt{VV} gives three machine instructions, namely AND, XOR, and one MOVE (assignment). The routine for the user is

```c
ulong bit_count_v(const ulong *x, ulong n)
{
    ulong b = 0;
    const ulong *xe = x + n + 1;
    while ( x+15 < xe ) // process blocks of 15 elements
        b += (bit_count(*x)<<i);
    return b;
}
```

Reading the columns as binary numbers tells us that in all positions of all words there were a total of $17 = 10001_2$ bits. The remaining instructions compute the total bit-count.

After some simplifications and loop-unrolling a routine for counting the bits of 15 words can be given as

```c
static inline ulong bit_count_v15(const ulong *x)
// Return sum(j=0, 14, bit_count(x[j])
// Technique is "vertical" addition.
{
#define VV(A) { ulong t = A & cy; A ^= cy; cy = t; }
ulong a1, a2, a3;
ulong a0=x[0];
{ ulong cy = x[ 1]; VV(a0); a1 = cy; }
{ ulong cy = x[ 2]; VV(a0); a1 ^= cy; }
{ ulong cy = x[ 3]; VV(a0); VV(a1); a2 = cy; }
{ ulong cy = x[ 4]; VV(a0); VV(a1); a2 ^= cy; }
{ ulong cy = x[ 5]; VV(a0); VV(a1); a2 ^= cy; }
{ ulong cy = x[ 6]; VV(a0); VV(a1); a2 ^= cy; }
{ ulong cy = x[ 7]; VV(a0); VV(a1); VV(a2); a3 = cy; }
{ ulong cy = x[ 8]; VV(a0); VV(a1); VV(a2); a3 ^= cy; }
{ ulong cy = x[ 9]; VV(a0); VV(a1); VV(a2); a3 ^= cy; }
{ ulong cy = x[10]; VV(a0); VV(a1); VV(a2); a3 ^= cy; }
{ ulong cy = x[11]; VV(a0); VV(a1); VV(a2); a3 ^= cy; }
{ ulong cy = x[12]; VV(a0); VV(a1); VV(a2); a3 ^= cy; }
{ ulong cy = x[13]; VV(a0); VV(a1); VV(a2); a3 ^= cy; }
{ ulong cy = x[14]; VV(a0); VV(a1); VV(a2); a3 ^= cy; }
#undef VV
ulong b = bit_count(a0);
b += (bit_count(a1)<<1);
b += (bit_count(a2)<<2);
b += (bit_count(a3)<<3);
return b;
}
```
24

Chapter 1: Bit wizardry

8     { b += bit_count_v15(x);
9         x += 15;
10     }
11
12     // process remaining elements:
13     const ulong r = (ulong)(xe-xe-1);
14     for (ulong k=0; k<r; ++k) b+=bit_count(x[k]);
15     return b;
16
17
Compared to the obvious method of bit-counting
1
ulong bit_count_v2(const ulong *x, ulong n)
2 {
3     ulong b = 0;
4     for (ulong k=0; k<n; ++k) b += bit_count(x[k]);
5     return b;
6 }

our routine uses roughly 30 percent less time when an array of 100,000,000 words is processed. There
are many possible modifications of the method. If the bit-count routine is rather slow one may want
to avoid the four calls to it after the processing of every 15 words. Instead, the variables a0, . . . , a3 could
be added (vertically!) to an array of more elements. If that array has n elements then only with each
block of 2^n − 1 words n calls to the bit-count routine are necessary.

1.10 Words as bit sets

1.10.1 Testing whether bit-subset

The following function tests whether a word u, as a bit-set, is a subset of the bit-set given as the word e
[FXT: bits/bitsubsetq.h]:

1     inline bool is_subset(ulong u, ulong e)
2         // Return whether u is a bit-subset of e.
3     {
4         return ( (u & e)==u );
5     }

Should u contain any bits not set in e then these bits are deleted in the AND-operation and the test for
equality will fail.

A proper subset is a subset of e is a subset \not= e:

1     inline bool is_proper_subset(ulong u, ulong e)
2         // Return whether u is a proper bit-subset of e.
3     {
4         return ( (u<e) && ((u & e)==u) );
5     }

The generated machine code contains a branch:

101     xorl %eax, %eax # prephitmp.71
102     cmpq %rsi, %rdi # e, u
103     jae .L6 #, /* branch to end of function */
104     andq %rdi, %rsi # u, e
106     xorl %eax, %eax # prephitmp.71
107     cmpq %rdi, %rsi # u, e
108     sete %al #, prephitmp.71

Replace the boolean operator ‘&&’ by the bit-wise operator ‘&’ to obtain branch-free machine code:

101     cmpq %rsi, %rdi # e, u
102     setb %al #, tmp63
103     andq %rdi, %rsi # u, e
105     cmpq %rdi, %rsi # u, e
106     sete %dl #, tmp66
107     addl %edx, %eax # tmp66, tmp63
1.11: Avoiding branches

Branches are expensive operations with many CPUs, especially if the CPU pipeline is very long. A trick that is often useful is to replace

```c
if ( (x<0) || (x>m) ) { ... }
```

where `x` might be a signed integer, by

```c
if ( (unsigned)x > m ) { ... }
```

To test whether a point `(x, y)` lies outside square box whose size `m` is a power of two one should, instead of the sequence

```c
if ( (x<0) || (x>m) || (y<0) || (y>m) ) { ... }
```

we use

```c
if ( (unsigned)x > m ) { ... }
```

```c
if ( (unsigned)y > m ) { ... }
```

```c
if ( (unsigned)x < m ) { ... }
```

```c
if ( (unsigned)y < m ) { ... }
```

```c
if ( (unsigned)x > m ) { ... }
```

```c
if ( (unsigned)y > m ) { ... }
```

Bit arrays of arbitrary size are discussed in section 4.6 on page 158.

1.11.2 Testing whether an element is in a given set

We determine whether a given number is an element of a given set (which must be a subset of the set `{0, 1, 2, ..., BITS_PER_LONG-1}`). For example, to determine whether `x` is a prime less than 32, one can use the function

```c
ulong m = (1UL<<2) | (1UL<<3) | (1UL<<5) | ... | (1UL<<31); // precomputed
static inline ulong is_tiny_prime(ulong x)
{
    return m & (1UL << x);
}
```

The same idea can be applied to look up tiny factors [FXT: bits/tinyfactors.h]:

```c
static inline bool is_tiny_factor(ulong x, ulong d)
// For x,d < BITS_PER_LONG (!)
// return whether d divides x (1 and x included as divisors)
// no need to check whether d==0
{
    return ( 0 != ( (tiny_factors_tab[x]>>d) & 1 ) );
}
```

The function uses the precomputed array [FXT: bits/tinyfactors.cc]:

```c
extern const ulong tiny_factors_tab[] =
{
    0x0UL, // x = 0: ( bits: ........)
    0x2UL, // x = 1: 1 ( bits: ......1.)
    0x6UL, // x = 2: 1 2 ( bits: .....11.)
    0xaUL, // x = 3: 1 3 ( bits: ....1.1.)
    0x16UL, // x = 4: 1 2 4 ( bits: ...1.11.)
    0x22UL, // x = 5: 1 5 ( bits: ..1...1.)
    0x4eUL, // x = 6: 1 2 3 6 ( bits: ..11.11.)
    0x82UL, // x = 7: 1 7 ( bits: 1.....1.)
    0x116UL, // x = 8: 1 2 4 8 ( bits: .1..111.)
    0x20aUL, // x = 9: 1 3 9
    ...
    0x2000000000002UL, // x = 29: 1 29
    0x4000846eUL, // x = 30: 1 2 3 5 6 10 15 30
    0x80000002UL, // x = 31: 1 31
    #if ( BITS_PER_LONG > 32 )
    0x100010116UL, // x = 32: 1 2 4 8 16 32
    0x20000080aUL, // x = 33: 1 2 3 7 9 21 63
    #endif // ( BITS_PER_LONG > 32 )
    ...
};
```

```c
Bit arrays of arbitrary size are discussed in section 4.6 on page 158.
```

[fxtbook draft of 2008-August-17]
better use
if ( (x|y) > m ) { ... }

The following functions are given in [FXT:bits/branchless.h]. This function returns \( \max(0, x) \). That is, zero is returned for negative input, else the unmodified input:

```c
1 static inline long max0(long x)
2 {
3     return x & ~(x >> (BITS_PER_LONG-1));
4 }
```

There is no restriction on the input range. The trick used is that with negative \( x \) the arithmetic shift will give a word of all ones which is then negated and the AND-operation deletes all bits. The following routine computes \( \min(0, x) \):

```c
1 static inline long min0(long x)
2 {
3     return x & (x >> (BITS_PER_LONG-1));
4 }
```

A routine for the computation of the average \( (x + y)/2 \) of two arguments \( x \) and \( y \) is

```c
1 static inline ulong average(ulong x, ulong y)
2 {
3     return (x & y) + ((x ^ y) >> 1);
4 }
```

The function gives the correct value even if \( (x + y) \) does not fit into a machine word. If it is known that \( x \geq y \) then one can use the simpler statement \( \text{return } y + (x - y)/2 \).

The following \texttt{upos_*()} functions only work for a limited range. The highest bit must not be set as it is used to emulate the carry flag. Branchless computation of the absolute difference \( |a - b| \):

```c
1 static inline ulong upos_abs_diff(ulong a, ulong b)
2 {
3     long d1 = b - a;
4     long d2 = (d1 & (d1>>(BITS_PER_LONG-1)))<<1;
5     return d1 - d2; // == (b - d) - (a + d);
6 }
```

Sorting of the arguments:

```c
1 static inline void upos_sort2(ulong &a, ulong &b)
2 {
3     long d = b - a;
4     d &= (d>>(BITS_PER_LONG-1));
5     a += d;
6     b -= d;
7 }
```

Johan Rönnow gives [priv.comm.] the following versions for signed integer minimum, maximum, and absolute value, that can be advantageous for PPC (G4) CPUs:

```c
1 #define B1 (BITS_PER_LONG-1) // bits of signed int minus one
2 #define MINI(x,y) (((x) & (((int)((x)-(y)))>>B1)) + ((y) & ~(((int)((x)-(y)))>>B1)))
3 #define MAXI(x,y) (((x) & ~(((int)((x)-(y)))>>B1)) + ((y) & (((int)((x)-(y)))>>B1)))
4 #define ABSI(x) (((x) & ~(((int)(x))>>B1)) - ((x) & (((int)(x)))>>B1)))
```

### 1.11.1 Conditional swap

The following statement is compiled with a branch:

```c
if ( a<b ) { ulong t=a; a=b; b=t; } // swap if a < b
```

Here: \( a \) in \%rcx, \( b \) in \%rdx

```
62 000e 4889C8 movq %rcx, %rax # X, X
68 0011 4839D1 cmpq %rdx, %rcx # X, X
```

[fxtbook draft of 2008-August-17]
1.11: Avoiding branches

An equivalent branchless version uses a conditional move:

```c
{ ulong x=a^b; if (a>=b) { x=0; } a^=x; b^=x; } // swap if a < b
```

We'd like to have fewer instructions. But with

```c
{ ulong ta=a; if (a<b) {a=b; b=ta;} } // swap if a < b
```

the machine code is identical to the first version. Let's try [FXT: bits/swap.h]:

```c
1 static inline void cswap_lt(ulong &a, ulong &b)
2 // Branchless equivalent to:
3 // if ( a<b ) { ulong t=a; a=b; b=t; } // swap if a < b
4 {
5   asm volatile("movq %0, %%r15 \n" // t=a
6     "cmpq %0, %1 \n" // cmp a, b
7     "cmovae %1, %0 \n" // cond a=b
8     "cmovae %r15, %1 \n" // cond b=t
9     "" //
10    :="x" (a), "x"= (b) // output
11    :="0" (a), "1" (b) // input
12    ;"r15" // clobber
13    );
14 }
```

Now the machine code looks better:

```c
// Here: a in %rax, b in %rdx
111 0027 4989C7 movq %rax, %r15 # tmp71
112 002a 4839C2 cmpq %rax, %rdx # tmp71, tmp72
113 002d 480F43C2 cmovae %rdx, %rax # tmp72, tmp71
114 0031 490F43D7 cmovae %r15, %rdx # tmp72
```

Clearly, the relative speed of the three versions depends on the machine used. But it also turns out to be dependent on the surrounding code. We use bubble sort for benchmarking:

```c
1 void bubble_sort(ulong *f, ulong n)
2 {
3    while ( n-- > 1 )
4        for (ulong k=0; k<n; ++k) cswap_NN(f[k], f[k+1]);
5 }
```

Where we use the three versions of conditional swap for cswap_NN(). We sort an array of length 2\textsuperscript{15} twice with each version, once starting with an already sorted array and once with an array that is sorted in descending order. The 'plain' version wins:

```c
cswap_gt_plain(f[k], f[k+1]); // 3.58s
```

This is due to the fact that the compiler bypasses the store when no swap happens:

```c
103 0020 488B4808 movq 8(%rax), %rcx, %rdx #, tmp71
104 0024 488B10 movq (%rax), %rax, %rdx #* f, tmp72
105 0027 4839D1 cmpq %rdx, %rax # tmp72, tmp71
106 002a 7307 jae .L7 #,
108 002c 48895008 movq %rdx, 8(%rax) # tmp72,
109 0030 488908 movq %rcx, (%rax) # tmp71,* f
```

If we change the inner loop to

```c
for (ulong k=0; k<n; ++k) {
```
Chapter 1: Bit wizardry

```c
ulong a = f[k], b = f[k+1];
cswap_NN(a, b);
f[k] = a; f[k+1] = b;
```

then we obtain:

```c
cswap_gt_plain(a, b); // 5.78s
cswap_gt_xor(a, b); // 6.22s
cswap_gt(a, b); // 4.68s
```

Our innocent looking change in the code prevented the compiler from doing its nice trick. The XOR version is (within timing precision) as slow as before. The assembler version wins because the data is already in registers. We learn that profiling is an absolute must.

### 1.11.2 Your compiler may be smarter than you thought

The machine code generated for

```c
x = x & ~(x >> (BITS_PER_LONG-1)); // max0()
```

is

```c
35: 48 99 cqto
37: 48 83 c4 08 add $0x8,%rsp // stack adjustment
3f: 48 f7 d2 not %rdx
3e: 48 21 d0 and %rdx,%rax
```

The variable `x` resides in the register `rAX` both at start and end of the function. The compiler uses a special (AMD64) instruction `cqto`. Quoting [12]:

Copies the sign bit in the rAX register to all bits of the rDX register. The effect of this instruction is to convert a signed word, doubleword, or quadword in the rAX register into a signed doubleword, quadword, or double-quadword in the rDX:rAX registers. This action helps avoid overflow problems in signed number arithmetic.

Now the equivalent

```c
x = ( x<0 ? 0 : x ); // max0() "simple minded"
```

is compiled to:

```c
35: ba 00 00 00 00 mov $0x0,%edx
3a: 48 85 c0 test %rax,%rax
3d: 48 0f 48 c2 cmovs %rdx,%rax // note %edx is %rdx
```

A conditional move (`cmovs`) instruction is used here. That is, our optimized version is (on my machine) actually worse than the straightforward equivalent.

A second example is a function to adjust a given value when it lies outside a given range [FXT: `bits/branchless.h`):

```c
1 static inline long clip_range(long x, long mi, long ma)
2 // Code equivalent to (for mi<=ma):
3 // if ( x<mi ) x = mi;
4 // else if ( x>ma ) x = ma;
5 {
6    x -= mi;
7    x = clip_range0(x, ma-mi);
8    x += mi;
9    return x;
10 }
```

The auxiliary function used involves one branch:

```c
1 static inline long clip_range0(long x, long m)
2 // Code equivalent (for m>0) to:
3 // if ( x<0 ) x = 0;
4 // else if ( x>m ) x = m;
5 // return x;
6 {
7    if ( (ulong)x > (ulong)m ) x = m & ~(x >> (BITS_PER_LONG-1));
```
1.12: Bit-wise rotation of a word

```c
8    return x;
9 }
```

The generated machine code is

```
0: 48 89 f8 mov %rdi,%rax
3: 48 29 f2 sub %rsi,%rdx
6: 31 c9 xor %ecx,%ecx
8: 48 29 f0 sub %rsi,%rax
b: 78 0a js 17 <_Z2CLlll+0x17> // the branch
d: 48 39 d0 cmp %rdx,%rax
t: 48 29 f2 sub %rdx,%rax
b: 48 0f 4e c8 cmovle %rdx,%rcx
f: 48 29 f0 sub %rsi,%rdx
12: 48 39 d0 cmp %rdx,%rax
15: 48 0f 4f c2 cmovg %rdx,%rax
19: 48 01 f0 add %rsi,%rax
```

Now we replace the code by

```c
1 inline long clip_range(long x, long mi, long ma)
2 {
3     x -= mi;
4     if ( x<0 ) x = 0;
5     // else // commented out to make (compiled) function really branchless
6     {
7         ma -= mi;
8         if ( x>ma ) x = ma;
9     }
10     x += mi;
11 }
```

Then the compiler generates branchless code:

```
0: 48 89 f8 mov %rdi,%rax
3: b9 00 00 00 00 mov $0x0,%ecx
6: 48 29 f0 sub %rsi,%rax
b: 48 0f 48 c1 cmovs %rcx,%rax
f: 48 29 f2 sub %rsi,%rdx
12: 48 39 d0 cmp %rdx,%rax
15: 48 0f 4f c2 cmovg %rdx,%rax
19: 48 01 f0 add %rsi,%rax
```

Still, with CPUs that do not have a conditional move instruction (or some branchless equivalent of it) the techniques shown in this section can be useful.

### 1.12 Bit-wise rotation of a word

Neither C nor C++ have a statement for bit-wise rotation of a binary word (which may be considered a missing feature). The operation can be ‘emulated’ via [FXT:bits/bitrotate.h]:

```c
1 static inline ulong bit_rotate_left(ulong x, ulong r)
2 // Return word rotated r bits to the left
3 // (i.e. toward the most significant bit)
4 {
5     return (x<<r) | (x>>(BITS_PER_LONG-r));
6 }
```

As already mentioned, GCC emits exactly the CPU instruction that is meant here, even with non-constant argument r. Well done, GCC folks! Explicit use of the corresponding assembler instruction should not do any harm:

```c
1 static inline ulong bit_rotate_right(ulong x, ulong r)
2 // Return word rotated r bits to the right
3 // (i.e. toward the least significant bit)
4 {
5     #if defined BITS_USE_ASM // use x86 asm code
6         return asm_ror(x, r);
7     #else
8         return (x>>r) | (x<<(BITS_PER_LONG-r));
9     #endif
10 }
```

where we used [FXT:bits/bitasm-amd64.h]:

[fxtbook draft of 2008-August-17]
static inline ulong asm_ror(ulong x, ulong r)
{
    asm("rorq %llx, %0" : "=r" (x) : "o" (x), "c" (r));
    return x;
}

Rotations using only a part of the word length are achieved by

static inline ulong bit_rotate_left(ulong x, ulong r, ulong ldn)
// Return ldn-bit word rotated r bits to the left
// (i.e. toward the most significant bit)
// Must have 0 <= r <= ldn
{
    ulong m = ~0UL >> (BITS_PER_LONG - ldn);
    x &= m;
    x = (x<<r) | (x>>(ldn-r));
    x &= m;
    return x;
}

and

static inline ulong bit_rotate_right(ulong x, ulong r, ulong ldn)
// Return ldn-bit word rotated r bits to the right
// (i.e. toward the least significant bit)
// Must have 0 <= r <= ldn
{
    ulong m = ~0UL >> (BITS_PER_LONG - ldn);
    x &= m;
    x = (x>>r) | (x<<(ldn-r));
    x &= m;
    return x;
}

Finally, the functions

static inline ulong bit_rotate_sgn(ulong x, long r, ulong ldn)
// Positive r --> shift away from element zero
{
    if ( r > 0 ) return bit_rotate_left(x, (ulong)r, ldn);
    else return bit_rotate_right(x, (ulong)-r, ldn);
}

and (full-word version)

static inline ulong bit_rotate_sgn(ulong x, long r)
// Positive r --> shift away from element zero
{
    if ( r > 0 ) return bit_rotate_left(x, (ulong)r);
    else return bit_rotate_right(x, (ulong)-r);
}

are sometimes convenient.

1.13 Functions related to bit-wise rotation and binary necklaces

We give several functions related to cyclic rotations of binary words and a class to generate binary necklaces.

1.13.1 Cyclic matching, minimum, and maximum

The following function determines whether there is a cyclic right shift of its second argument so that it matches the first argument. It is given in [FXT: bits/bitcyclic-match.h]:

static inline ulong bit_cyclic_match(ulong x, ulong y)
// Return r if x=rotate_right(y, r) else return ~0UL.
// In other words: return
// how often the right arg must be rotated right (to match the left)
// or, equivalently:
1.13: Functions related to bit-wise rotation and binary necklaces

```c
// how often the left arg must be rotated left (to match the right)
{ ulong r = 0;
do {
if ( x==y ) return r;
y = bit_rotate_right(y, 1);
}
while ( ++r < BITS_PER_LONG );
return ~0UL;
}
```

The functions shown work on the full length of the words, equivalents for the sub-word of the lowest ldn bits are given in the respective files. Just one example:

```c
static inline ulong bit_cyclic_match(ulong x, ulong y, ulong ldn)
// Return r if x==rotate_right(y, r, ldn) else return ~0UL
// (using ldn-bit words)
{
ulong r = 0;
do {
if ( x==y ) return r;
y = bit_rotate_right(y, 1, ldn);
}
while ( ++r < ldn );
return ~0UL;
}
```

The minimum among all cyclic shifts of a word can be computed via the following function given in [FXT: bits/bitcyclic-minmax.h]:

```c
static inline ulong bit_cyclic_min(ulong x)
// Return minimum of all rotations of x
{
ulong r = 1;
ulong m = x;
do {
x = bit_rotate_right(x, 1);
if ( x<m ) m = x;
}
while ( ++r < BITS_PER_LONG );
return m;
}
```

1.13.2 Cyclic period and binary necklaces

Selecting from all $n$-bit words those that are equal to their cyclic minimum gives the sequence of the binary length-$n$ necklaces, see chapter 17 on page 361. For example, with 6-bit words we obtain:

<table>
<thead>
<tr>
<th>word</th>
<th>period</th>
<th>word</th>
<th>period</th>
</tr>
</thead>
<tbody>
<tr>
<td>.....</td>
<td>1</td>
<td>..111</td>
<td>6</td>
</tr>
<tr>
<td>....1</td>
<td>6</td>
<td>......</td>
<td>1</td>
</tr>
<tr>
<td>...11</td>
<td>6</td>
<td>...111</td>
<td>1</td>
</tr>
<tr>
<td>...111</td>
<td>6</td>
<td>....11</td>
<td>1</td>
</tr>
</tbody>
</table>

The values in each right column can be computed using [FXT: bits/bitcyclic-period.h]:

```c
static inline ulong bit_cyclic_period(ulong x, ulong ldn)
// Return minimal positive bit-rotation that transforms x into itself.
// (using ldn-bit words)
// The returned value is a divisor of ldn.
{
ulong y = bit_rotate_right(x, 1, ldn);
return bit_cyclic_match(x, y, ldn) + 1;
}
```

It is possible to completely avoid the rotation of partial words: let $d$ be a divisor of the word length $n$. Then the rightmost $(n-1)d$ bits of the word obtained as $x^*(x>>d)$ are zero exactly if the word has
period \( d \). Thereby we can use the following function body:

```c
ulong sl = BITS_PER_LONG-ldn;
for (ulong s=1; s<ldn; ++s)
{
    ++sl;
    if ( 0==( (x^(x>>s)) << sl ) ) return s;
}
return ldn;
```

Testing for periods that are not divisors of the word length can be avoided as follows:

```c
ulong f = tiny_factors_tab[ldn];
ulong sl = BITS_PER_LONG-ldn;
for (ulong s=1; s<ldn; ++s)
{
    ++sl;
    f >>= 1;
    if ( 0==(f&1) ) continue;
    if ( 0==( (x^(x>>s)) << sl ) ) return s;
}
return ldn;
```

The table of tiny factors used is shown in section 1.10.2 on page 25.

The version for \( ldn==BITS_PER_LONG \) can be optimized similarly:

```c
static inline ulong bit_cyclic_period(ulong x)
// Return minimal positive bit-rotation that transforms x into itself.
// (same as bit_cyclic_period(x, BITS_PER_LONG) )
// // The returned value is a divisor of the word length,
// // i.e. 1,2,4,8,...,BITS_PER_LONG.
{
    ulong r = 1;
    do
    {
        ulong y = bit_rotate_right(x, r);
        if ( x==y ) return r;
        r <<= 1;
    } while ( r < BITS_PER_LONG );
    return r; // == BITS_PER_LONG
}
```

### 1.13.3 Generating all binary necklaces

We can generate all necklaces by the FKM algorithm given in section 17.1.1 on page 362. Here we specialize the method for binary words. The words generated are the cyclic maxima [FXT: class bit_necklace in bits/bit-necklace.h]:

```c
class bit_necklace
{
public:
    ulong a_; // necklace
    ulong j_; // period of the necklace
    ulong n2_; // bit representing n: n2==2**(n-1)
    ulong j2_; // bit representing j: j2==2**(j-1)
    ulong n_; // number of bits in words
    ulong mm_; // mask of n ones
    ulong tfb_; // for fast factor lookup
public:
    bit_necklace(ulong n) { init(n); }
    ~bit_necklace() { ; }
    void init(ulong n)
    {
        if ( 0==n ) n = 1; // avoid hang
        if ( n>=BITS_PER_LONG ) n = BITS_PER_LONG;
        n_ = n;
    }
```
1.13: Functions related to bit-wise rotation and binary necklaces

```c
n2_ = 1UL<<(n-1);
mm_ = (~0UL) >> (BITS_PER_LONG-n);
tfb_ = tiny_factors_tab[n] >> 1;
tfb_ |= n2_; // needed for n==BITS_PER_LONG

```

```c
first();
}
```  

```c
void first()
{
a_ = 0;
j_ = 1;
j2_ = 1;
}
```  

```c
ulong data() const { return a_; }
ulong period() const { return j_; }
```

The method for computing the successor is

```c
ulong next()
// Create next necklace.
// Return the period, zero when current necklace is last.
{
    if ( a_==mm_ ) { first(); return 0; }
    do
    {
        // next lines compute index of highest zero, same result as
        // j_ = highest_zero_idx( a_ ^ (~mm_) );
        // but the direct computation is faster:
        j_ = n_ - 1;
        ulong jb = 1UL << j_;
        while ( 0!=(a_ & jb) ) { --j_; jb>>=1; }
        j2_ = 1UL << j_;
        ++j_;
        a_ |= j2_;
        a_ = bit_copy_periodic(a_, j_, n_);
    } while ( 0==(tfb_ & j2_) ); // necklaces only
    return j_;
}
```

It uses the following function for periodic copying [FXT: bits/bitperiodic.h]:

```c
static inline ulong bit_copy_periodic(ulong a, ulong p, ulong ldn)
// Return word that consists of the lowest p bits of a repeated
// in the lowest ldn bits (higher bits are zero).
// E.g.: if p==3, ldn=7 and a=*****xyz (8-bit), the return 0zxyzxyz.
// Must have p>0 and ldn>0.
{
    a &=( ~0UL >> (BITS_PER_LONG-p) );
    for (ulong s=p; s<ldn; s<<=1) { a |= (a<s); }
    a &=( ~0UL >> (BITS_PER_LONG-ldn) );
    return a;
}
```

Finally, we can easily detect whether a necklace is a Lyndon word:

```c
ulong is_lyndon_word() const { return (j2_ & n2_); }
```

```c
ulong next_lyn()
// Create next Lyndon word.
// Return the period (==n), zero when current necklace is last.
{
    if ( a_==mm_ ) { first(); return 0; }
    do { next(); } while ( !is_lyndon_word() );
    return n_;
}
```

About 54 million necklaces per second are generated (with \( n = 32 \)), corresponding to a rate of 112 M/s for pre-necklaces [FXT: bits/bit-necklace-demo.cc].
1.13.4 Computing the cyclic distance

A function to compute the cyclic distance between two words [FXT: bits/bitcyclic-dist.h] is:

```c
inline ulong bit_cyclic_dist(ulong a, ulong b)
// Return minimal bitcount of (t ^ b)
// where t runs through the cyclic rotations of a.
{
    ulong d = ~0UL;
    ulong t = a;
    do
    {
        ulong z = t ^ b;
        ulong e = bit_count(z);
        if ( e < d ) d = e;
        t = bit_rotate_right(t, 1);
    } while ( t!=a );
    return d;
}
```

If the arguments are cyclic shifts of each other then zero is returned. A version for partial words is

```c
inline ulong bit_cyclic_dist(ulong a, ulong b, ulong ldn)
{
    ulong d = ~0UL;
    const ulong m = (~0UL>>(BITS_PER_LONG-ldn));
    b &= m;
    a &= m;
    ulong t = a;
    do
    {
        ulong z = t ^ b;
        ulong e = bit_count(z);
        if ( e < d ) d = e;
        t = bit_rotate_right(t, 1, ldn);
    } while ( t!=a );
    return d;
}
```

1.13.5 Cyclic XOR and its inverse

The functions [FXT: bits/bitcyclic-xor.h]

```c
static inline ulong bit_cyclic_rxor(ulong x)
{
    return x ^ bit_rotate_right(x, 1);
}
```

and

```c
static inline ulong bit_cyclic_lxor(ulong x)
{
    return x ^ bit_rotate_left(x, 1);
}
```

return a word whose number of set bits is even. A word and its complement produce the same result.

The inverse functions need no rotation at all, the inverse of `bit_cyclic_rxor()` is the inverse Gray code (see section [1.16](#) on page [42]):

```c
static inline ulong bit_cyclic_inv_rxor(ulong x)
// Return v so that bit_cyclic_rxor(v) == x.
{
    return inverse_gray_code(x);
}
```

The argument `x` must have an even number of bits. If this is the case then the lowest bit of the result is zero. The complement of the returned value is also an inverse of `bit_cyclic_rxor()`.

The inverse of `bit_cyclic_lxor()` is the inverse reversed code (see section [1.16.6](#) on page [47]):

```c
static inline ulong bit_cyclic_inv_lxor(ulong x)
// Return v so that bit_cyclic_lxor(v) == x.
{
    return inverse_reversed_code(x);
}
```
1.14 Reversing the bits of a word

The bits of a binary word can efficiently be reversed by a sequence of steps that reverse the order of certain blocks. For 16-bit words, we need \(4 = \log_2(16)\) such steps [FXT: bits/revbin-steps-demo.cc]:

\[
\begin{array}{cccccccccccccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & a & b & c & d & e & f \\
1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 & 9 & 8 & b & a & d & c & f & e \\
3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 & b & a & 9 & 8 & f & e & d & c \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & f & e & d & c & b & a & 9 & 8 \\
f & e & d & c & b & a & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]

---

1.14.1 Swapping adjacent bit blocks

We need a couple of auxiliary functions given in [FXT: bits/bitswap.h]. Pairs of adjacent bits can be swapped via

\[
\begin{array}{cccccccccccccccccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & a & b & c & d & e & f \\
1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 & 9 & 8 & b & a & d & c & f & e \\
3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 & b & a & 9 & 8 & f & e & d & c \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & f & e & d & c & b & a & 9 & 8 \\
f & e & d & c & b & a & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]

---

When swapping half-words (here for 32-bit architectures)
Chapter 1: Bit wizardry

1. static inline ulong bit_swap_16(ulong x)
2. // Return x with groups of 16 bits swapped.
3. {
4.   ulong m = 0x0000ffffUL;
5.   return ((x & m) << 16) | ((x & (m<<16)) >> 16);
6. }

We could also use the bit-rotate function from section 1.12 on page 29, or
return (x << 16) | (x >> 16);

The GCC compiler recognizes that the whole operation is equivalent to a (left or right) word rotation
and indeed emits just a single rotate instruction.

1.14.2 Bit-reversing binary words

The following is a function to reverse the bits of a binary word [FXT: bits/revbin.h]:

1. static inline ulong revbin(ulong x)
2. // Return x with reversed bit order.
3. {
4.   x = bit_swap_1(x);
5.   x = bit_swap_2(x);
6.   x = bit_swap_4(x);
7.   x = bit_swap_8(x);
8.   x = bit_swap_16(x);
9.   #if BITS_PER_LONG >= 64
10.    x = bit_swap_32(x);
11.   #endif
12.   return x;
13. }

The steps after bit_swap_4() correspond to a byte-reverse operation. This operation is just one assembler
instruction for many CPUs (‘bswap’). The inline assembler with GCC for AMD64 CPUs is given in [FXT:
bits/bitasm-amd64.h]:

1. static inline ulong asm_bswap(ulong x)
2. {
3.   asm ("bswap %0" : =r" (x) : "0" (x));
4.   return x;
5. }

We use it for byte reversion when available:

1. static inline ulong bswap(ulong x)
2. // Return word with reversed byte order.
3. {
4.   #ifdef BITS_USE_ASM
5.     x = asm_bswap(x);
6.   #else
7.     x = bit_swap_8(x);
8.     x = bit_swap_16(x);
9.     #if BITS_PER_LONG >= 64
10.        x = bit_swap_32(x);
11.     #endif
12.   #endif
13. // def BITS_USE_ASM
14.   return x;
15. }

The function actually used for bit reversion is good for both 32 and 64 bit words:

1. static inline ulong revbin(ulong x)
2. {
3.   x = bit_swap_1(x);
4.   x = bit_swap_2(x);
5.   x = bit_swap_4(x);
6.   x = bswap(x);
7.   return x;
8. }

One can generate the masks in the process as follows:
### 1.14: Reversing the bits of a word

```c
static inline ulong revbin(ulong x)
{
  ulong s = BITS_PER_LONG >> 1;
  ulong m = "0UL >> s;
  while (s)
  {
    x = ( (x & m) << s ) ^ ( (x & (~m)) >> s );
  
    s >>= 1;
    m ^= (m<<s);
  }
  return x;
}
```

Note that the above function will not always beat the obvious, bit-wise algorithm:

```c
static inline ulong revbin(ulong x)
{
  ulong r = 0, ldn = BITS_PER_LONG;
  while ( ldn-- != 0 )
  {
    r <<= 1;
    r += (x&1);
    x >>= 1;
  }
  return r;
}
```

Therefore the function

```c
static inline ulong revbin(ulong x, ulong ldn)
// Return word with the ldn least significant bits
// (i.e. bit_0 ... bit_{ldn-1}) of x reversed,
// the other bits are set to zero.
{
  return revbin(x) >> (BITS_PER_LONG-ldn);
}
```

should only be used when \( ldn \) is not too small, else be replaced by the trivial algorithm.

One can also use table lookups methods so that, for example, eight bits are reversed at a time using a 256-byte table. We give the routine for full words:

```c
unsigned char revbin_tab[256]; // reversed 8-bit words
ulong revbin_t(ulong x)
{
  ulong r = revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  #if BYTES_PER_LONG > 4
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  #endif
  r <<= 8; r |= revbin_tab[ x ];
  return r;
}
```

The routine can be optimized by unrolling to avoid all branches:

```c
static inline ulong revbin_t(ulong x)
{
  ulong r = revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  #if BYTES_PER_LONG > 4
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
  #endif
  r <<= 8; r |= revbin_tab[ x ];
  return r;
}
```

However, reversing the first \( 2^{30} \) binary words with this routine takes (on a 64-bit machine) longer than with the routine using the \texttt{bit\_swap\_NN()} calls, see [FXT: bits/revbin-tab-demo.cc].

[fxtbook draft of 2008-August-17]
1.14.3 Generating the bit-reversed words in order

If the bit-reversed words have to be generated in the (reversed) counting order then there is a significantly cheaper way to do the update [FXT: bits/revbin-upd.h]:

```c
static inline ulong revbin_upd(ulong r, ulong h)
// Let n=2**ldn and h=n/2.
// Then, with r == revbin(x, ldn) at entry, return revbin(x+1, ldn)
// Note: routine will hang if called with r the all-ones word
{
    while ( !((r^=h)&h) ) h >>= 1;
    return r;
}
```

Now assume we want to generate the bit-reversed words of all \( N = 2^n \) words smaller than \( 2^n \). The total number of branches with the while-loop can be estimated by observing that for half of the updates just one bit changes, for a quarter two bits change, three bits change for one eighth of all updates, and so on. Thereby the loop executes less than \( 2N \) times:

\[
N \left( \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots + \frac{\log_2(N)}{N} \right) = N \sum_{j=1}^{\log_2(N)} \frac{j}{2^j} < 2N \quad (1.14-1)
\]

Observing that the updates that involve a single bit change occur at every second step we can avoid half of all branches.

For large values of \( N \) the following method can be significantly faster if a fast routine is available for the computation of the least significant bit in a word. The underlying observation is that for a fixed word of size \( n \) there are just \( n \) different patterns of bit-changes with incrementing. We generate a lookup table of the bit-reversed patterns, `utab[]`, an array of BITS_PER_LONG elements:

```c
inline void make_revbin_upd_tab(ulong ldn)
// Initialize lookup table used by revbin_tupd()
{
    utab[0] = 1UL<<(ldn-1);
    for (ulong k=1; k<ldn; ++k) utab[k] = utab[k-1] | (utab[k-1]>>1);
}
```

The change patterns for \( n = 5 \) start as

<table>
<thead>
<tr>
<th>pattern</th>
<th>reversed pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.....</td>
</tr>
<tr>
<td></td>
<td>11...</td>
</tr>
<tr>
<td></td>
<td>111.</td>
</tr>
<tr>
<td></td>
<td>1111.</td>
</tr>
<tr>
<td></td>
<td>11111</td>
</tr>
</tbody>
</table>

The crucial observation is that the pattern with \( x \) set bits is used for the update of \( k \) to \( k+1 \) when the lowest zero of \( k \) is at index \( x-1 \):

- `utab[0]` = 1..... used when the lowest zero of \( k \) is at index 0
- `utab[1]` = 11... used when the lowest zero of \( k \) is at index 1
- `utab[2]` = 111.. used when the lowest zero of \( k \) is at index 2
- `utab[3]` = 1111. used when the lowest zero of \( k \) is at index 3
- `utab[4]` = 11111 used when the lowest zero of \( k \) is at index 4

The update routine can now be implemented as

```c
inline ulong revbin_tupd(ulong r, ulong k)
// Let r=revbin(k, ldn) then
// return revbin(k+1, ldn).
// NOTE 1: need to call make_revbin_upd_tab(ldn) before usage
// where ldn=log_2(n)
// NOTE 2: different argument structure than revbin_upd()
{
    k = lowest_one_idx(~k); // lowest zero idx
    r ^= utab[k];
    return r;
}
```
The revbin-update routines are used for the revbin permutation described in section 2.1.

<table>
<thead>
<tr>
<th></th>
<th>30 bits</th>
<th>16 bits</th>
<th>8 bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update, bit-wise</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>revbin_upd()</td>
</tr>
<tr>
<td>Update, table</td>
<td>0.99</td>
<td>1.08</td>
<td>1.15</td>
<td>revbin_tupd()</td>
</tr>
<tr>
<td>Full, masks</td>
<td>0.74</td>
<td>0.81</td>
<td>0.86</td>
<td>revbin()</td>
</tr>
<tr>
<td>Full, 8-bit table</td>
<td>1.77</td>
<td>1.94</td>
<td>2.06</td>
<td>revbin_t()</td>
</tr>
<tr>
<td>Full32, 8-bit table</td>
<td>0.83</td>
<td>0.90</td>
<td>0.96</td>
<td>revbin_t_le32()</td>
</tr>
<tr>
<td>Full16, 8-bit table</td>
<td>—</td>
<td>0.54</td>
<td>0.58</td>
<td>revbin_t_le16()</td>
</tr>
<tr>
<td>Full, generated masks</td>
<td>2.97</td>
<td>3.25</td>
<td>3.45</td>
<td>page 36</td>
</tr>
<tr>
<td>Full, bit-wise</td>
<td>8.76</td>
<td>5.77</td>
<td>2.50</td>
<td>page 37</td>
</tr>
</tbody>
</table>

**Figure 1.14-A:** Relative performance of the revbin-update and (full) revbin routines. The timing of the bit-wise update routine is normalized to one. Values in each column should be compared, smaller values correspond to faster routines. A column labeled “N bits” gives the timing for reversing the N least significant bits of a word.

The relative performance of the different revbin routines is shown in figure 1.14-A. As a surprise, the full-word revbin function is consistently faster than both of the update routines. This is mainly due to the fact that the machine used (see appendix B on page 923) has a byte swap instruction. As the performance of table lookups is highly machine dependent your results can be very different.

### 1.14.4 Alternative techniques for in-order generation

The following loop, due to Brent Lehmann [priv.comm.], also generates the bit-reversed words in succession:

```c
ulong n = 32; // a power of two
ulong p = 0, s = 0, n2 = 2*n;
do {
    // here: s is the bit-reversed word
    p += 2;
    s ^= n - (n / (p&-p));
} while ( p<n2 );
```

The revbin-increment is branchless but involves a division which usually is an expensive operation. With a fast bit-scan function the loop should be replaced by

```c
do {
    p += 1;
    s ^= n - (n >> (lowest_one_idx(p)+1));
} while ( p<n );
```

A recursive algorithm for the generation of the bit-reversed words in order is given in [FXT: bits/revbin-rec-demo.cc]:

```c
ulong N;
void revbin_rec(ulong f, ulong n) {
    // visit( f )
    for (ulong m=N>>1; m>n; m>>=1) revbin_rec(f+m, m);
}
```

One has to call revbin_rec(0, 0) to generate all N-bit bit-reversed words.

A technique to generate all revbin pairs in a pseudo random order is given in section 39.4 on page 870.
1.15 Bit-wise zip

The bit-wise zip (bit-zip) operation moves the lower half bits to even indices and higher half bits to odd indices. For example, with 8-bit words the permutation of bits is (see section 2.5 on page 99):

\[
\begin{bmatrix}
  a & b & c & d \\
  A & B & C & D
\end{bmatrix} \rightarrow \begin{bmatrix}
  a & A & b & B \\
  c & C & d & D
\end{bmatrix}
\]

A straightforward implementation is

```c
ulong bit_zip(ulong a, ulong b)
{
    ulong x = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        x |= (a & m) << s;
        ++s;
        x |= (b & m) << s;
        m <<= 1;
    }
    return x;
}
```

Its inverse (bit-unzip) moves even indexed bits to the lower half-word and odd indexed bits to the higher half-word:

```c
void bit_unzip(ulong x, ulong &a, ulong &b)
{
    a = 0; b = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        a |= (x & m) >> s;
        ++s;
        m <<= 1;
        b |= (x & m) >> s;
    }
}
```

For a faster implementation we will use the `butterfly_*()`-functions which are defined in [FXT: bits/bitbutterfly.h] (64-bit version):

```c
static inline ulong butterfly_4(ulong x)
{
    const ulong ml = 0x0f000f000f000f00UL;
    const ulong s = 4;
    const ulong mr = ml >> s;
    const ulong t = ((x & ml) >> s) | ((x & mr) << s);
    x = (x & ~(ml | mr)) | t;
    return x;
}
```

The following version of the function may look more elegant but is actually slower:

```c
static inline ulong butterfly_4(ulong x)
{
    const ulong m = 0x0ff00ff00ff00ff0UL;
    ulong c = x & m;
    c ^= (c<<4) ^ (c>>4);
    c &= m;
    return x ^ c;
}
```

The optimized versions of the bit-zip and bit-unzip routines are [FXT: bits/bitzip.h]:

```c
static inline ulong bit_zip(ulong x)
{
    #if BITS_PER_LONG == 64
        x = butterfly_16(x);
    #endif
    x = butterfly_8(x);
    x = butterfly_4(x);
    x = butterfly_2(x);
    x = butterfly_1(x);
}
```

[fxtbook draft of 2008-August-17]
1.15: Bit-wise zip

```c
10 return x;
11 }
```

and

```c
1 static inline ulong bit_unzip(ulong x)
2 {
3   x = butterfly_1(x);
4   x = butterfly_2(x);
5   x = butterfly_4(x);
6   x = butterfly_8(x);
7 #if BITS_PER_LONG == 64
8   x = butterfly_16(x);
9 #endif
10 return x;
11 }
```

Laszlo Hars suggests [priv.comm.] the following routine (version for 32-bit words), which can be obtained by making the compile-time constants explicit:

```c
1 static inline uint32 bit_zip(uint32 x)
2 {
3   x = ((x & 0x0000ff00) << 8) | ((x >> 8) & 0x0000ff00) | (x & 0xff0000ff);
4   x = ((x & 0x00f000f0) << 4) | ((x >> 4) & 0x00f000f0) | (x & 0xf00ff00f);
5   x = ((x & 0x0c0c0c0c) << 2) | ((x >> 2) & 0x0c0c0c0c) | (x & 0xc3c3c3c3);
6   x = ((x & 0x22222222) << 1) | ((x >> 1) & 0x22222222) | (x & 0x99999999);
7 return x;
8 }
```

A bit-zip version for words whose higher half is zero is (64-bit version)

```c
1 static inline ulong bit_zip0(ulong x)
2 // Return word with lower half bits in even indices.
3 {
4   x = (x | (x<<16)) & 0x0000ffff0000ffffUL;
5   x = (x | (x<<8)) & 0x00ff00ff00ff00ffUL;
6   x = (x | (x<<4)) & 0x0f0f0f0f0f0f0f0fUL;
7   x = (x | (x<<2)) & 0x3333333333333333UL;
8   x = (x | (x<<1)) & 0x5555555555555555UL;
9 return x;
10 }
```

Its inverse is

```c
1 static inline ulong bit_unzip0(ulong x)
2 // Bits at odd positions must be zero.
3 {
4   x = (x | (x>>1)) & 0x3333333333333333UL;
5   x = (x | (x>>2)) & 0x00ff00ff00ff00ffUL;
6   x = (x | (x>>4)) & 0x0f0f0f0f0f0f0f0fUL;
7   x = (x | (x>>8)) & 0x3333333333333333UL;
8   x = (x | (x>>16)) & 0x5555555555555555UL;
9 return x;
10 }
```

The simple structure of the routines suggests to try the following versions of bit-zip and its inverse:

```c
1 static inline ulong bit_zip(ulong x)
2 {
3   ulong y = (x >> 32);
4   x &= 0xffffffffUL;
5   y = (y | (y<<16)) & 0x0000ffff0000ffffUL;
6   y = (y | (y<<8)) & 0x0000ffff0000ffffUL;
7   y = (y | (y<<4)) & 0x0000ffff0000ffffUL;
8   y = (y | (y<<2)) & 0x0000ffff0000ffffUL;
9   y = (y | (y<<1)) & 0x0000ffff0000ffffUL;
10   x |= (y<<1);
11 return x;
12 }
```

```c
1 static inline ulong bit_unzip(ulong x)
2 {
3   ulong y = (x >> 1) & 0x5555555555555555UL;
4   ```
Chapter 1: Bit wizardry

As the statements involving the variables \(x\) and \(y\) are independent the CPU-internal parallelism can be used. However, these versions turn out to be slightly slower than the ones given before.

The following function moves the bits of the lower half-word of \(x\) into the even positions of \(lo\), and the bits of the higher half-word into \(hi\) (two versions given):

```
#define BPLH (BITS_PER_LONG/2)

static inline void bit_zip2(ulong x, ulong &lo, ulong &hi)
{
#if 1
    x = bit_zip(x);
    lo = x & 0x5555555555555555UL;
    hi = (x>>1) & 0x5555555555555555UL;
#else
    hi = bit_zip0( x >> BPLH );
    lo = bit_zip0( (x << BPLH) >> (BPLH) );
#endif
}
```

The inverse function is

```
static inline ulong bit_unzip2(ulong lo, ulong hi)
// Inverse of bit_zip2(x, lo, hi).
{
#if 1
    return bit_unzip( (hi<<1) | lo );
#else
    return bit_unzip0(lo) | (bit_unzip0(hi) << BPLH);
#endif
}
```

Functions that zip/unzip the bits of the lower half of two words are

```
static inline ulong bit_zip2(ulong x, ulong y)
// Two-word version:
// only the lower half of x and y are merged
{
    return bit_zip( (y<<BPLH) + x );
}
```

and (64-bit version)

```
static inline void bit_unzip2(ulong t, ulong &x, ulong &y)
// Two-word version:
// only the lower half of x and y are filled
{
    t = bit_unzip(t);
    y = t >> BPLH;
    x = t & 0x0000000000000000UL;
}
```

1.16 Gray code and parity

The Gray code of a binary word can easily be computed by

```
static inline ulong gray_code(ulong x)
{
```

Gray codes of consecutive values differ in one bit. Squared Gray codes of consecutive values differ in one or two bits. Gray codes of values that have a difference of a power of two differ in two bits. Gray codes of even/odd values have an even/odd number of bits set, respectively. This is demonstrated in [FXT: bits/gray2-demo.cc], whose output is given in figure 1.16-A.

In order to produce a random value with an even/odd number of bits set, set the lowest bit of a random number to zero/one, respectively, and take the Gray code.

Computing the inverse Gray code is slightly more expensive. As the Gray code is the ‘bit-wise difference modulo 2’ we can computing the inverse as ‘bit-wise sums modulo 2’:

```
static inline ulong inverse_gray_code(ulong x)
{
    // VERSION 1 (integration modulo 2):
    ulong h=1, r=0;
    do
    {
        if ( x & 1 ) r^=h;
        x >>= 1;
        h = (h<<1)+1;
    } while ( x!=0 );
    return r;
}
```

Applying the Gray code twice is identical to $x \oplus x \gg 2$; applying it four times is $x \oplus x \gg 4$; and the idea holds for all powers of two. This leads to the most efficient way to compute the inverse Gray code:

```
// VERSION 2 (apply graycode BITS_PER_LONG-1 times):
ulong r = BITS_PER_LONG;
while ( --r ) x ^= x>>1;
return x;
```
Chapter 1: Bit wizardry

1.16.1 The parity of a binary word

The *parity* of a word is its bit-count modulo two. The lowest bit of the inverse Gray code of a word contains the parity of the word. Thereby we compute the parity as [FXT: bits/parity.h]:

```c
static inline ulong parity(ulong x)
// return 1 if the number of set bits is even, else 0
{
    return inverse_gray_code(x) & 1;
}
```

Each bit of the inverse Gray code contains the parity of the partial input left from it (including itself).

Be warned that the parity flag of many CPUs is the complement of the above. With the x86-architecture the parity bit also only takes in account the lowest byte. One can use the following routine [FXT: bits/bitasm-i386.h]:

```c
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>16);
    x ^= (x>>8);
    asm (...);
    return x;
}
```

The equivalent code for the AMD64 CPU is [FXT: bits/bitasm-amd64.h]:

```c
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>32);
    x ^= (x>>16);
    x ^= (x>>8);
    asm (...);
    return x;
}
```

1.16.2 Byte-wise Gray code and parity

A byte-wise Gray code can be computed using (32-bit version)

```c
static inline ulong byte_gray_code(ulong x)
// Return the Gray code of bytes in parallel
{
    return x ^ ((x & 0xfefefefe)>>1);
}
```

Its inverse is

```c
static inline ulong byte_inverse_gray_code(ulong x)
// Return the inverse Gray code of bytes in parallel
{
```
4  x ^= ((x & 0xfeedeefeUL) >> 1);
5  x ^= ((x & 0xcf0cf0cfUL) >> 2);
6  x ^= ((x & 0xf0f0f0f0UL) >> 4);
7  return x;
8 }

Thereby
1 static inline ulong byte_parity(ulong x)
2 { // Return the parities of bytes in parallel
3   return byte_inverse_gray_code(x) & 0x01010101UL;
4 }

1.16.3 Incrementing (counting) in Gray code

<table>
<thead>
<tr>
<th>k:</th>
<th>g(k)</th>
<th>g(2*k)</th>
<th>g(k)</th>
<th>diff</th>
<th>p</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>.......</td>
<td>.......</td>
<td>.....</td>
<td>+1</td>
<td>1</td>
<td>{}</td>
</tr>
<tr>
<td>1:</td>
<td>......1</td>
<td>.....11</td>
<td>.....1</td>
<td>+1</td>
<td>1</td>
<td>{0}</td>
</tr>
<tr>
<td>2:</td>
<td>......11</td>
<td>..1111</td>
<td>..111</td>
<td>+1</td>
<td>1</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>3:</td>
<td>.....1.</td>
<td>..11.1</td>
<td>..11.</td>
<td>+1</td>
<td>1</td>
<td>{0, 2}</td>
</tr>
<tr>
<td>4:</td>
<td>......11</td>
<td>..1111</td>
<td>..111</td>
<td>+1</td>
<td>1</td>
<td>{0, 1, 2}</td>
</tr>
<tr>
<td>5:</td>
<td>.....11</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 1, 2, 3}</td>
</tr>
<tr>
<td>6:</td>
<td>.....11</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 1, 2, 3}</td>
</tr>
<tr>
<td>7:</td>
<td>......1</td>
<td>..11</td>
<td>..1</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>8:</td>
<td>......1</td>
<td>..11</td>
<td>..1</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>9:</td>
<td>......11</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>10:</td>
<td>..1111</td>
<td>..1111</td>
<td>..111</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>11:</td>
<td>..1111</td>
<td>..1111</td>
<td>..111</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>12:</td>
<td>..1111</td>
<td>..1111</td>
<td>..111</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>13:</td>
<td>..111</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>14:</td>
<td>..111</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>15:</td>
<td>..111</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>16:</td>
<td>..111</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>17:</td>
<td>..111</td>
<td>..111</td>
<td>..11</td>
<td>+1</td>
<td>1</td>
<td>{0, 3}</td>
</tr>
</tbody>
</table>

Figure 1.16-B: The Gray code equals the Gray code of doubled value shifted to the right once. Equiv-
    alently, we can separate the lowest bit which equals the parity of the other bits. The last column shows
    that the changes with each increment always happen one position left of the rightmost bit.

Let g(k) be the Gray code of a number k. We are interested in efficiently generating g(k + 1). Using the
observation shown in figure 1.16-B we can implement a fast Gray counter if we use a spare bit to keep
track of the parity of the Gray code word. The following routine does this [FXT: bits/nextgray.h]:

inline ulong next_gray2(ulong x)
// With input x==gray_code(2*k) the return is gray_code(2*k+2).
// Let x1 be the word x shifted right once
// and i1 its inverse Gray code.
// Let r1 be the return r shifted right once.
// Then r1 = gray_code(i1+1).
// That is, we have a Gray code counter.
// The argument must have an even number of bits.
{
  x ^= 1;
  x ^= (lowest_one(x) << 1);
  return x;
}

Start with x=0, increment with x=next_gray2(pg) and use the words g=x>>1:

ulong x = 0;
for (ulong k=0; k<n2; ++k)
  { ulong g = x>>1;
    x = next_gray2(x);
    // here: g == gray_code(k);  
}
Chapter 1: Bit wizardry

This is shown in [FXT: bits/bit-nextgray-demo.cc].

To start at an arbitrary (Gray code) value \( g \) compute

\[
x = (g \ll 1) \oplus \text{parity}(g)
\]

in order to use the statement \( x = \text{next\_gray2}(x) \) for later increments.

If one works with a set whose elements are the set bits in the Gray code then the parity is the set size \( k \) modulo two. The increment can then be achieved as follows: if \( k \) is even then, if the first element is zero, then remove it, else prepend the element zero. If \( k \) is odd then, if the first element equals the second minus one, then remove the second element, else insert at the second position the element equal to the first element plus one. Further, the decrement is obtained by simply swapping the actions for even and odd parity.

If one works with an array that contains the elements of the set it is more convenient to actually do the described operations at the end of the array. This leads to the (loopless) algorithm for subsets in minimal-change order that is given in section 8.2 on page 204.

1.16.4 The Thue-Morse sequence

The sequence of parities of the binary words,

\[
0110100110100101101001100110100110010110011010011001011001101001100101100110100110010110011010011001011001101001100101100110100110010110011010011001011001101001...
\]

is called the Thue-Morse sequence (entry \([A010060]\) of \([245]\)). It appears in various seemingly unrelated contexts, see \([8]\) and section 36.1 on page 723.

The sequence can be generated with [FXT: class thue_morse in bits/thue-morse.h]

```cpp
class thue_morse
{
    public:
        ulong k_;
        ulong tm_; 
    public:
        thue_morse(ulong k) { init(k); }
        ~thue_morse() { ; }
        ulong init(ulong k)
        {
            k_ = k;
            tm_ = parity(k_);
            return tm_; 
        }
        ulong data() { return tm_; }
        ulong next()
        {
            ulong x = k_  ^ (k_ + 1);
            x ^= x>>1; // highest bit that changed with increment
            x &= 0x5555555555555555UL; // 64-bit version
            tm_ ^= ( x!=0 ); // change if highest changed bit was at even index
            return tm_; 
        }
};
```

The rate of generation is about 435 million per second (5 cycles per update) [FXT: bits/thue-morse-demo.cc].

1.16.5 The Golay-Rudin-Shapiro sequence *

The function [FXT: bits/grsnegative.h]
1.16: Gray code and parity

returns one for indices where the Golay-Rudin-Shapiro sequence (or GRS sequence) has a negative value. The function returns one for \( x \) in the sequence

\[
3, 6, 11, 12, 13, 15, 19, 22, 24, 25, 26, 30, 35, 38, 43, 44, 45, 47, 48, 49, 50, 52, 53, 55, 59, 60, 61, 63, 67, 70, 75, 76, 77, 79, 83, 86, 88, 89, 90, 94, 96, 97, 98, 100, 101, 103, 104, 105, 106, 110, 115, 118, 120, 121, 122, 126, 131, 134, 139, 140, \ldots
\]

This is sequence A020985 of [245], see also section 36.3 on page 728.

The sequence can be obtained by starting with two ones and in each step appending the left half and the negated right half of the values so far, see figure 1.16-C.

The algorithm counts the bit-pairs modulo 2. Note that the sequence [1111] contains three bit-pairs: [11..], [.11.] and [..11]. The function proves to be useful in specialized versions of the fast Fourier- and Walsh transform, see section 22.4 on page 463.

1.16.6 The reversed Gray code

We define the reversed Gray code to be the bit-reversed word of the Gray code of the bit-reversed word. That is,

\[
\text{rev_gray_code}(x) := \text{revbin}(\text{gray_code}(\text{revbin}(x)))
\]
It turns out that the corresponding functions are identical to the Gray code versions up to the reversed shift operations (C-language operators ‘\&\&’ replaced by ‘\&\&‘). Thereby, computing the reversed Gray code is as easy as [FXT: bits/revgraycode.h]:

```
1 static inline ulong rev_gray_code(ulong x)
2 {
3     return x ^ (x\ll1);
4 }
```

Its inverse is

```
1 static inline ulong inverse_rev_gray_code(ulong x)
2 {
3     // use: rev_gray ** BITS_PER_LONG == id:
4     x ^= x\ll1; // rev_gray ** 1
5     x ^= x\ll2; // rev_gray ** 2
6     x ^= x\ll4; // rev_gray ** 4
7     x ^= x\ll8; // rev_gray ** 8
8     x ^= x\ll16; // rev_gray ** 16
9     // here: x = rev_gray**31(input)
10    // note: the statements can be reordered at will
11    #if BITS_PER_LONG >= 64
12       x ^= x\ll32; // for 64bit words
13    #endif
14    return x;
15 }
```

Some examples with 32-bit words are shown in figure 1.16-D. The inverse reversed Gray code contains at each bit position the parity of all bits of the input right from it, including the bit itself. Especially, the word parity can be found in the highest bit of the inverse reversed Gray code.

The reversed Gray code preserves the lowest set bit while the Gray code preserves the highest.

Let \(G^{-1}\) and \(E^{-1}\) be the inverse Gray- and reversed Gray code of \(X\), respectively. Then the bit-wise sum (XOR) of \(G^{-1}\) and \(E^{-1}\) equals \(X\) if the parity of \(X\) is zero, else it equals the complement \(X\).

We note that taking the reversed Gray code of a binary word corresponds to multiplication with the binary polynomial \(x + 1\), and the inverse reversed Gray code is a method for fast exact division by \(x + 1\), see section 38.1.6 on page 823.

### 1.17 Bit sequency

Functions concerned with the sequency (number of zero-one transitions) are given in [FXT: bits/bitsequency.h]. Sequency counting:

```
1 static inline ulong bit_sequency(ulong x)
2 {
3     return bit_count( gray_code(x) );
4 }
```

The function assumes that all bits to the left of the word are zero, and all bits to right are equal to the lowest bit. For example, the sequency of the 8-bit word \(00\ldots0011111\) is one. To take the lowest bit into account, add it to the sequency (then all sequencies are even).

The minimal binary word with given sequency can be computed as follows:

```
1 static inline ulong first_sequency(ulong k)
2 { // Return the first (i.e. smallest) word with sequency k,
3     // e.g. 00..00010101010 (seq 8)
4     // e.g. 00..00101010101 (seq 9)
5     // Must have: 0 <= k <= BITS_PER_LONG
6     return inverse_gray_code( first_comb(k) );
7 }
```

A faster version is (32-bit branch only):
1.17: Bit sequency

The maximal binary word with given sequency can be computed via

```c
1 static inline ulong last_sequency(ulong k)
2 // Return the last (i.e. biggest) word with sequency k.
3 {
4    return inverse_gray_code( last_comb(k) );
5 }
```

The functions `first_comb(k)` and `last_comb(k)` return a word with `k` bits set at the low and high end, respectively (see section 1.25 on page 67).

Generation of all words with a given sequency, starting with the smallest, can be achieved with a function that computes the next word with the same sequency:

```c
1 static inline ulong next_sequency(ulong x)
2 // Return smallest integer with highest bit at greater or equal
3 // position than the highest bit of x that has the same number
4 // of zero-one transitions (sequency) as x.
5 // The value of the lowest bit is conserved.
6 // Zero is returned when there is no further sequence.
7 {
8    x = gray_code(x);
9    x = next_colex_comb(x);
10    x = inverse_gray_code(x);
11    return x;
12 }
```

The list of all 6-bit words ordered by sequency is shown in figure 1.17-A. It was created with the program [FXT: bits/bitsequency-demo.cc].

We note that the sequency of a word can be ‘complemented’ as follows (32-bit version):

```c
1 static inline ulong complement_sequency(ulong x)
```
// Return word whose sequency is BITS_PER_LONG - s
// where s is the sequency of x
{
    return x ^ 0xaaaaaaaaUL;
}

1.18 Powers of the Gray code

The Gray code is a bit-wise linear transform of a binary word. The $2^k$-th power of the Gray code of $x$ can be computed as $x ^ (x>>k)$. The $e$-th power can be computed as the bit-wise sum of the powers corresponding to the bits in the exponent. This motivates [FXT: bits/graypower.h]:

```c
inline ulong gray_pow(ulong x, ulong e)
// Return (gray_code**e)(x)
// gray_pow(x, 1) == gray_code(x)
// gray_pow(x, BITS_PER_LONG-1) == inverse_gray_code(x)
{
    e &= (BITS_PER_LONG-1); // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e )
    {
        if ( e & 1 ) x ^= x >> s; // gray ** s
        s <<= 1;
        e >>= 1;
    }
    return x;
}
```

The Gray code $g = [g_0, g_1, ..., g_7]$ of an 8-bit binary word $x = [x_0, x_1, ..., x_7]$ can be expressed as a matrix multiplication over GF(2) (dots for zeros):

$$
g = G x
$$

The powers of the Gray code correspond to multiplication with powers of the matrix $G$, shown in figure[1.18-A](bottom). The powers of the inverse Gray code for $N$-bit words (where $N$ is a power of two) can be computed by the relation $G^e G^{N-e} = G^N = id$.

```c
inline ulong inverse_gray_pow(ulong x, ulong e)
// Return (inverse_gray_code**(e))(x)
// == (gray_code**(-e))(x)
// inverse_gray_pow(x, 1) == inverse_gray_code(x)
// inverse_gray_pow(x, BITS_PER_LONG-1) == gray_code(x)
```
The matrices corresponding to the powers of the reversed Gray code are shown in figure 1.18-A (bottom). We just have to reverse the shift operator in the functions:

```cpp
// Return (rev_gray_code**e)(x)
inline ulong rev_gray_pow(ulong x, ulong e) {
    e &= (BITS_PER_LONG-1); // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e ) {
        if ( e & 1 ) x ^= x << s; // rev_gray ** s
        s <<= 1;
        e >>= 1;
    }
    return x;
}
```

The inverse function is

```cpp
// Return (inverse_rev_gray_code**(e))(x)
inline ulong inverse_rev_gray_pow(ulong x, ulong e) {
    return rev_gray_pow(x, -e);
}
```

### 1.19 Invertible transforms on words

The functions presented in this section are invertible ‘transforms’ on binary words. The names are chosen as ‘some code’, emphasizing the result of the transforms, similar to the convention used with the name ‘Gray code’. The functions are given in [FXT: bits/bittransforms.h].

In the transform (blue code)

```cpp
// Return blue_code(a)
inline ulong blue_code(ulong a) {
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    do {
        a ^= ( (a&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    } while ( s );
    return a;
}
```

the masks ‘m’ are (32-bit binary)

```
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
```

The same masks are used in the yellow code

```cpp
// Return yellow_code(a)
inline ulong yellow_code(ulong a) {
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    do {
        a ^= ( (a&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    } while ( s );
    return a;
}
```
Both involve a computational work $\sim \log_2(b)$ where $b$ is the number of bits per word (BITS_PER_LONG). The blue code can be used as a fast implementation for the composition of a binary polynomial with $x + 1$, see section [83.7.2 on page 343](#). Note the names ‘blue code’ etc. are ad hoc terminology and not standard.

<table>
<thead>
<tr>
<th>blue</th>
<th>yellow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:......0*</td>
<td>01111111111111111111111111111111 32</td>
</tr>
<tr>
<td>1:......1*</td>
<td>11111111111111111111111111111111 64</td>
</tr>
<tr>
<td>2:....1</td>
<td>11111111111111111111111111111111 128</td>
</tr>
<tr>
<td>3:....1</td>
<td>11111111111111111111111111111111 256</td>
</tr>
<tr>
<td>4:....1</td>
<td>11111111111111111111111111111111 512</td>
</tr>
<tr>
<td>5:....1</td>
<td>11111111111111111111111111111111 1024</td>
</tr>
<tr>
<td>6:....1</td>
<td>11111111111111111111111111111111 2048</td>
</tr>
<tr>
<td>7:....1</td>
<td>11111111111111111111111111111111 4096</td>
</tr>
<tr>
<td>8:....1</td>
<td>11111111111111111111111111111111 8192</td>
</tr>
<tr>
<td>9:....1</td>
<td>11111111111111111111111111111111 16384</td>
</tr>
<tr>
<td>10:....1</td>
<td>11111111111111111111111111111111 32768</td>
</tr>
<tr>
<td>11:....1</td>
<td>11111111111111111111111111111111 65536</td>
</tr>
<tr>
<td>12:....1</td>
<td>11111111111111111111111111111111 131072</td>
</tr>
<tr>
<td>13:....1</td>
<td>11111111111111111111111111111111 262144</td>
</tr>
<tr>
<td>14:....1</td>
<td>11111111111111111111111111111111 524288</td>
</tr>
<tr>
<td>15:....1</td>
<td>11111111111111111111111111111111 1048576</td>
</tr>
<tr>
<td>16:....1</td>
<td>11111111111111111111111111111111 2097152</td>
</tr>
<tr>
<td>17:....1</td>
<td>11111111111111111111111111111111 4194304</td>
</tr>
<tr>
<td>18:....1</td>
<td>11111111111111111111111111111111 8388608</td>
</tr>
<tr>
<td>19:....1</td>
<td>11111111111111111111111111111111 16777216</td>
</tr>
<tr>
<td>20:....1</td>
<td>11111111111111111111111111111111 33554432</td>
</tr>
<tr>
<td>21:....1</td>
<td>11111111111111111111111111111111 67108864</td>
</tr>
<tr>
<td>22:....1</td>
<td>11111111111111111111111111111111 134217728</td>
</tr>
<tr>
<td>23:....1</td>
<td>11111111111111111111111111111111 268435456</td>
</tr>
<tr>
<td>24:....1</td>
<td>11111111111111111111111111111111 536870912</td>
</tr>
<tr>
<td>25:....1</td>
<td>11111111111111111111111111111111 1073741824</td>
</tr>
<tr>
<td>26:....1</td>
<td>11111111111111111111111111111111 2147483648</td>
</tr>
<tr>
<td>27:....1</td>
<td>11111111111111111111111111111111 4294967296</td>
</tr>
<tr>
<td>28:....1</td>
<td>11111111111111111111111111111111 8589934592</td>
</tr>
<tr>
<td>29:....1</td>
<td>11111111111111111111111111111111 17179869184</td>
</tr>
<tr>
<td>30:....1</td>
<td>11111111111111111111111111111111 34359738368</td>
</tr>
<tr>
<td>31:....1</td>
<td>11111111111111111111111111111111 68719476736</td>
</tr>
</tbody>
</table>

**Figure 1.19-A:** Blue and yellow transforms of the binary words 0,1,...,32. Bit-counts are shown at the right of each column. Fixed points are marked with asterisks.

The transforms of the binary words up to 31 are shown in figure [1.19-A](#), the lists were created with the program [FXT: bits/bittransforms-blue-demo.cc](#). The parity of $B(a)$ is equal to the lowest bit of $a$. Up to the $a = 47$ the bit-count varies by $\pm 1$ between successive values of $B(a)$, the transition $B(47) \rightarrow B(48)$ changes the bit-count by 3. The sequence of the indices $a$ where the bit-count changes by more than one is


The yellow code might be a good candidate for ‘randomization’ of binary words. The blue code maps any range $[0\ldots2^k - 1]$ onto itself. Both the blue code and the yellow code are involutions (self-inverse).

The transforms *(red code)*

```c
inline ulong red_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    do
    {
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u<<s);
        a ^= (v>>s);
        s >>= 1;
        m ^= (m<<s);
    } while ( s );
    return a;
}
```

and *(green code)*

```c
inline ulong green_code(ulong a)
{
    // ...
}
```

[fxtbook draft of 2008-August-17]
1.19: Invertible transforms on words

The sequence of fixed points of the blue code is (entry A118666 of [245]):

0, 1, 6, 7, 18, 19, 20, 21, 106, 107, 108, 109, 120, 121, 126, 127, 258, 259, ...

If \( f \) is a fixed point then \( f \) XOR 1 is also a fixed point. Further, \( 2 ( f \) XOR \( 2 f ) \) is a fixed point. These facts can be cast into a function that returns a unique fixed point for each argument [FXT: bits/blue-fixed-points.h]:

```
inline ulong blue_fixed_point(ulong s)
{
    .......
           11111111111111111111111111111111111111111
           11111111111111111111111111111111111111111
           11111111111111111111111111111111111111111
           11111111111111111111111111111111111111111
    do
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u>>s);
        a ^= (v<<s);
        m ^= (m>>s);
    while ( s );
    return a;
}
```

use the masks

```
.............11111111111111111111111111111111111111111
.............11111111111111111111111111111111111111111
.............11111111111111111111111111111111111111111
.............11111111111111111111111111111111111111111
```

The transforms of the binary words up to 31 are shown in figure 1.19-B, which was created with the program [FXT: bits/bittransforms-red-demo.cc].

1.19.1 Fixed points of the blue code

The transforms of the binary words up to 31 are shown in figure 1.19-B, which was created with the program [FXT: bits/bittransforms-red-demo.cc].

```
Figure 1.19-B: Red and green transforms of the binary words 0,1,...,32. Bit-counts are shown at the right of each column.
```

<table>
<thead>
<tr>
<th>red</th>
<th>green</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>1:</td>
</tr>
<tr>
<td>1:</td>
<td>1:</td>
</tr>
<tr>
<td>2:</td>
<td>1:</td>
</tr>
<tr>
<td>3:</td>
<td>1:</td>
</tr>
<tr>
<td>4:</td>
<td>1:</td>
</tr>
<tr>
<td>5:</td>
<td>1:</td>
</tr>
<tr>
<td>6:</td>
<td>1:</td>
</tr>
<tr>
<td>7:</td>
<td>1:</td>
</tr>
<tr>
<td>8:</td>
<td>1:</td>
</tr>
<tr>
<td>9:</td>
<td>1:</td>
</tr>
<tr>
<td>10:</td>
<td>1:</td>
</tr>
<tr>
<td>11:</td>
<td>1:</td>
</tr>
<tr>
<td>12:</td>
<td>1:</td>
</tr>
<tr>
<td>13:</td>
<td>1:</td>
</tr>
<tr>
<td>14:</td>
<td>1:</td>
</tr>
<tr>
<td>15:</td>
<td>1:</td>
</tr>
<tr>
<td>16:</td>
<td>1:</td>
</tr>
<tr>
<td>17:</td>
<td>1:</td>
</tr>
<tr>
<td>18:</td>
<td>1:</td>
</tr>
<tr>
<td>19:</td>
<td>1:</td>
</tr>
<tr>
<td>20:</td>
<td>1:</td>
</tr>
<tr>
<td>21:</td>
<td>1:</td>
</tr>
<tr>
<td>22:</td>
<td>1:</td>
</tr>
<tr>
<td>23:</td>
<td>1:</td>
</tr>
<tr>
<td>24:</td>
<td>1:</td>
</tr>
<tr>
<td>25:</td>
<td>1:</td>
</tr>
<tr>
<td>26:</td>
<td>1:</td>
</tr>
<tr>
<td>27:</td>
<td>1:</td>
</tr>
<tr>
<td>28:</td>
<td>1:</td>
</tr>
<tr>
<td>29:</td>
<td>1:</td>
</tr>
<tr>
<td>30:</td>
<td>1:</td>
</tr>
<tr>
<td>31:</td>
<td>1:</td>
</tr>
</tbody>
</table>
```
Chapter 1: Bit wizardry

Figure 1.19-C: The first fixed points of the blue code. The highest bit of all fixed points lies at an even index. There are $2^{n/2}$ fixed points with highest bit at index $n$.

```c
if ( 0==s ) return 0;
ulong f = 1;
while ( s>1 )
{
    f ^= (f<<1);
    f <<= 1;
    f |= (s&1);
    s >>= 1;
    f = inverse_rev_gray_code(f); // == bitpol_div(f, 3);
}
return f;
```

Now write $f(x)$ for the binary polynomial corresponding to $f$ (see chapter 38 on page 819). if $f(x)$ is a fixed point (that is, $Bf(x) = f(x+1) = f(x)$), then both $(x^2 + x)f(x)$ and $1 + (x^2 + x)f(x)$ are fixed points. The function `blue_fixed_point()` repeatedly multiplies by $x^2 + x$ and adds one if the corresponding bit of the argument is set.

The output for the first few arguments is shown in figure 1.19-C. Note that the fixed points are not in ascending order. The list was created by the program [FXT: `bits/bittransforms-blue-fp-demo.cc`].

The inverse function uses the fact that polynomial division by $x + 1$ can be achieved with the inverse reversed Gray code (see section 1.16.6 on page 47) if the polynomial is divisible by $x + 1$:}

```c
inline ulong blue_fixed_point_idx(ulong f)
// Inverse of blue_fixed_point()
{
    ulong s = 1;
    while ( f )
    { s <<= 1;
        s ^= (f & 1);
        f >>= 1;
        f = inverse_rev_gray_code(f); // == bitpol_div(f, 3);
    }
    return s >> 1;
}
```

1.19.2 Relations between the transforms

We write $B$ for the blue code (transform), $Y$ for the yellow code and $r$ for bit-reversal (the `revbin`-function). We have the following relations between $B$ and $Y$:

$$
B = YrY = rYr \quad (1.19-1a)
$$
$$
Y = BrB = rBr \quad (1.19-1b)
$$
$$
r = YBY = BYY \quad (1.19-1c)
$$

[fxtbook draft of 2008-August-17]
1.19: Invertible transforms on words

As said, $B$ and $Y$ are self-inverse:

$$B^{-1} = B, \quad BB = \text{id} \quad (1.19-2a)$$
$$Y^{-1} = Y, \quad YY = \text{id} \quad (1.19-2b)$$

The red code and the green code are not involutions (‘square roots of identity’) but third roots of identity (Using $R$ for the red code, $E$ for the green code):

$$RRR = \text{id}, \quad R^{-1} = RR = E \quad (1.19-3a)$$
$$EEE = \text{id}, \quad E^{-1} = EE = R \quad (1.19-3b)$$
$$RE = ER = \text{id} \quad (1.19-3c)$$

Figure 1.19-D shows the multiplication table. The $R$ in the third column of the second row says that $rB = R$. The letter $i$ is used for identity (id). An asterisk says that $xy \neq yx$.

By construction we have

$$R = rB \quad (1.19-4a)$$
$$E = rY \quad (1.19-4b)$$

Relations between $R$ and $E$ are:

$$R = ERE = rEr \quad (1.19-5a)$$
$$E = RRr = rRr \quad (1.19-5b)$$
$$R = RER \quad (1.19-5c)$$
$$E = ERE \quad (1.19-5d)$$

For the bit-reversal one has

$$r = YR = RB = BE = EY \quad (1.19-6)$$

Some products for the transforms are

$$B = RY = YE = RBR = EBE \quad (1.19-7a)$$
$$Y = E = BR = YR = EYE \quad (1.19-7b)$$
$$R = BY = BEB = YEY \quad (1.19-7c)$$
$$E = YB = BRB = YRY \quad (1.19-7d)$$

Some triple products that give the identical transform are

$$\text{id} = BYE = RYB \quad (1.19-8a)$$
$$\text{id} = EBY = BRY \quad (1.19-8b)$$
$$\text{id} = YEB = YBR \quad (1.19-8c)$$
1.19.3 Relations to Gray code and reversed Gray code

Write $g$ for the Gray code, then:

\[
\begin{align*}
g B g B &= \text{id} \quad (1.19-9a) \\
g B g &= B \quad (1.19-9b) \\
g^{-1} B g^{-1} &= B \quad (1.19-9c) \\
g B &= B g^{-1} \quad (1.19-9d)
\end{align*}
\]

Let $S_k$ be the operator that rotates a word by $k$ bits (bit zero is moved to position $k$, use [FXT: bit_rotate_sgn() in [bits/bitrotate.h]) then

\[
\begin{align*}
Y S_{+1} Y &= g \quad (1.19-10a) \\
Y S_{-1} Y &= g^{-1} \quad (1.19-10b) \\
Y S_k Y &= g^k \quad (1.19-10c)
\end{align*}
\]

Shift in the sequency domain is bit-wise derivative in time domain. Relation 1.19-10c, together with an algorithm to generate the cycle leaders of the Gray permutation (section 2.8.1 on page 104) gives a curious method to generate the binary necklaces whose length is a power of two, described in section 17.1.6 on page 367. Let $e$ be the operator for the reversed Gray code, then

\[
\begin{align*}
B S_{+1} B &= e^{-1} \quad (1.19-11a) \\
B S_{-1} B &= e \quad (1.19-11b) \\
B S_k B &= e^{-k} \quad (1.19-11c)
\end{align*}
\]

1.19.4 More transforms by symbolic powering

The idea of powering a transform (as with the Gray code, see section 1.18 on page 50) can be applied to the ‘color’-transforms as exemplified for the blue code:

```c
inline ulong blue_xcode(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1); // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        if ( x & 1 ) a ^= ( (a&m) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m>>s);
    }
    return a;
}
```

The result is not the power of the blue code which would be pretty boring as $BB = \text{id}$. Instead the transform (and the equivalents for $Y$, $R$ and $E$, see [FXT: bitxtransforms.h]) are more interesting; all relations between the transforms are still valid, if the symbolic exponent is identical with all terms in the relation. For example, we had $BB = \text{id}$, now $B^x B^x = \text{id}$ is true for all $x$. Similarly, $EE = R$ now has to be $E^x E^x = R^x$. That is, we have \text{BITS_PER_LONG} different versions of our four transforms that share their properties with the ‘simple’ versions. Among them are \text{BITS_PER_LONG} transforms $B^x$ and $Y^x$ that are involutions and $E^x$ and $R^x$ that are third roots of the identity: $E^x E^x E^x = R^x R^x R^x = \text{id}$.

While not powers of the simple versions, we still have $B^0 = Y^0 = R^0 = E^0 = \text{id}$. Further, let $e$ be the ‘exponent’ of all ones and $Z$ be any of the transforms, then $Z^e = Z$, Writing ‘$+$’ for the XOR operation, then $Z^e Z^y = Z^{x+y}$ and so $Z^e Z^y = Z$ whenever $x + y = e$. 

[fxtbook draft of 2008-August-17]
1.19.5 The building blocks of the transforms

Consider the following transforms on two-bit words where addition is bit-wise (that is, XOR):

\[
\begin{align*}
\text{id}_2 v &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \\
\text{r}_2 v &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix} \\
\text{B}_2 v &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ b \end{bmatrix} \\
\text{Y}_2 v &= \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \\
\text{R}_2 v &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a+b \end{bmatrix} \\
\text{E}_2 v &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ a \end{bmatrix}
\end{align*}
\]

(1.19-12a)

(1.19-12b)

(1.19-12c)

(1.19-12d)

(1.19-12e)

(1.19-12f)

It can easily be verified that for these the same relations hold as for id, r, B, Y, R, E. In fact the ‘color-transforms’, bit-reversion and (trivially) identity are the transforms obtained as repeated Kronecker-products of the matrices (see section 22.3 on page 461). The transforms are linear over GF(2):

\[
Z(\alpha a + \beta b) = \alpha Z(a) + \beta Z(b)
\]

(1.19-13)

The corresponding version of the bit-reversal is [FXT: xrevbin() in bits/revbin.h]:

```
inline ulong xrevbin(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        if ( x & 1 ) a = ( (a & m) << s ) ^ ( (a & (~m)) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m<<s);
    }
    return a;
}
```

Then, for example, \( R^x = r^x B^2 \) (see relation 1.19-4a on page 55). The yellow code is the bit-wise Reed-Muller transform (described in section 22.12 on page 486) of a binary word. The symbolic powering is equivalent to selecting individual levels of the transform.

### 1.20 Space filling curves

#### 1.20.1 The Hilbert curve

A rendering of the Hilbert curve is shown in figure 1.20-A. An efficient algorithm to compute the direction of the \( n \)-th move of the Hilbert curve is based on the parity of the number of threes in the radix-4 representation of \( n \) (see section 36.9.1 on page 745).

Let \( d_x \) and \( d_y \) correspond to the moves at step \( n \) in the Hilbert curve. Then \( d_x, d_y \in \{-1, 0, +1\} \) and exactly one of them is zero. Thereby for both \( p := d_x + d_y \) and \( m := d_x - d_y \) we have \( p, m \in \{-1, +1\} \).

The following function computes \( p \) and returns 0, 1 if \( p = -1, +1 \), respectively [FXT: bits/hilbert.h]:
Chapter 1: Bit wizardry

Figure 1.20-A: The Hilbert curve.

dx+dy: ++-+++-+++-+++----++++-+++-+++----++++-+++-+++----++++-+++-+++----+---+---+---++++-
dx-dy: +----+++-+++-+++-++++---+---+----++++---+---+----++++---+---+--
dir: >^<^^>v>^>vv<v>>^>v>>^<^>^<<v<^^^>v>>^<^>^<<v<^<<v>vv<^<v<^^>^<turn: 0--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+-+

Figure 1.20-B: Moves and turns of the Hilbert curve.

```c
inline ulong hilbert_p(ulong t)
{
    ulong d = (t & 0x5555555555555555UL) & ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
    return parity( d );
}
```

The function can be slightly optimized as follows (64-bit version only):

```c
inline ulong hilbert_p(ulong t)
{
    t &= ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
    t ^= t>>2;
    t ^= t>>4;
    t ^= t>>8;
    t ^= t>>16;
    t ^= t>>32;
    return t & 1;
}
```

The value of \( m \) can be computed as:

```c
inline ulong hilbert_m(ulong t)
{
    return hilbert_p( -t );
}
```

It remains to merge the values of \( p \) and \( m \) into a two-bit value \( d \) that encodes the direction of the move:

```c
inline ulong hilbert_dir(ulong t)
{
    ulong p = hilbert_p(t);
    ulong m = hilbert_m(t);
    ulong d = p ^ (m<<1);
}
```
To print the value of \( d \) symbolically, one can use the C++ statement
\[
\text{cout} \ll \text{("v^<")}[d];
\]
The turn \( u \) between steps can be computed as
\[
\begin{align*}
\text{inline int hilbert_turn(ulong t)} & \quad \text{// Return the turn (left or right) with the steps} \\
& \quad \text{// \( t \) and \( t-1 \) of the Hilbert curve.} \\
& \quad \text{// Returned value is} \\
& \quad \text{// 0 for no turn} \\
& \quad \text{// +1 for right turn} \\
& \quad \text{// -1 for left turn} \\
& \{ \\
& \quad \text{ulong d1 = hilbert_dir(t);} \\
& \quad \text{ulong d2 = hilbert_dir(t-1);} \\
& \quad \text{d1 ^= (d1>>1);} \\
& \quad \text{d2 ^= (d2>>1);} \\
& \quad \text{ulong u = d1 - d2;} \\
& \quad \text{// at this point, symbolically: cout \ll \text{("+-0+-")}[ u + 3 ];} \\
& \quad \text{if ( 0==u ) return 0;} \\
& \quad \text{if ( (long)u<0 ) u += 4;} \\
& \quad \text{return (1==u ? +1 : -1);} \\
& \}
\end{align*}
\]
To print the value of \( u \) symbolically, one can use the statement
\[
\text{cout} \ll \text{("-0+")}[d+1];
\]
The values of \( p \) and \( m \), followed by the direction and turn of the Hilbert curve are shown in figure 1.20-B.
The list was created with the program [FXT: bits/hilbert-moves-demo.cc]. Figure 1.20-A was created with the program [FXT: bits/hilbert-texpic-demo.cc]. A finite state machine to transform to and from linear coordinates is given in section 4.8 on page 163. More information about the Hilbert curve can be found in [270] ch.14.

### 1.20.2 The Z-order

A 2-dimensional space-filling curve that traverses all points in each quadrant before it enters the next can be obtained by the Z-order. Figure 1.20-C shows a rendering of the Z-order curve, it was created with the program [FXT: bits/zorder-texpic-demo.cc]. The conversion between a linear parameter to a pair coordinates achieved by separating the bits at the even and odd indices [FXT: bits/zorder.h].

\[\text{inline void lin2zorder(ulong t, ulong \&x, ulong \&y)} \{ \text{bit_unzip2(t, x, y);} \} \]

The routine \text{bit_unzip2()} is described in section 1.13 on page 40. The inverse is \text{inline ulong zorder2lin(ulong x, ulong y)} \{ \text{return bit_zip2(x, y);} \}.

From any coordinate pair the next pair can be computed with the following (constant amortized time) routine:
\[
\begin{align*}
\text{inline void zorder_next(ulong \&x, ulong \&y)} & \quad \{ \\
& \}
\end{align*}
\]
Chapter 1: Bit wizardry

The previous pair is obtained similarly:

```c
inline void zorder_prev(ulong &x, ulong &y)
{
    ulong b = 1;
    do
    {
        x ^= b; b &= ~x;
        y ^= b; b &= ~y;
        b <<= 1;
    } while ( b );
}
```

The routines are written in a way that generalizes trivially to more dimensions:

```c
inline void zorder3d_next(ulong &x, ulong &y, ulong &z)
{
    ulong b = 1;
    do
    {
        x ^= b; b &= ~x;
        y ^= b; b &= ~y;
        z ^= b; b &= ~z;
        b <<= 1;
    } while ( b );
}
```

Unlike the Hilbert curve there are steps where the curve advances more than one unit.

### 1.21 Scanning for zero bytes

The following function (32-bit version) determines if any sub-byte of the argument is zero from [FXT: bits/zerobyte.h]:

```c
static inline ulong contains_zero_byte(ulong x)
{
    return ((x-0x01010101UL)^x) & (~x) & 0x80808080UL;
}
```

It returns zero when `x` contains no zero-byte and nonzero when it does. The idea is to subtract one from each of the bytes and then look for bytes where the borrow propagated all the way to the most significant bit. In order to scan for other values than zero (e.g. 0xa5) one can use `contains_zero_byte( x ^ 0xa5a5a5a5UL )`.

Using the simplified version

```c
    return ((x-0x01010101UL) ^ x) & 0x80808080UL;
```
1.22: 2-adic inverse and square root

1.22.1 Computation of the inverse

The 2-adic inverse can be computed using an iteration (see section 28.1.5 on page 571) with quadratic convergence. The number to be inverted has to be odd [FXT: bits/bit2adic.h]:

```c
inline ulong inv2adic(ulong x)
// Return inverse modulo 2**BITS_PER_LONG
// x must be odd
// The number of correct bits is doubled with each step
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 3, 6, 12, 24, 48, 96, ... bits (or better)
{
    if ( 0==(x&1) ) return 0; // not invertible
    ulong i = x; // correct to three bits at least
    ulong p;
    do
    {
        p = i * x;
        i *= (2UL - p);
    } while ( p!=1 );
    return i;
}
```

Let \( m \) be the modulus (a power of two), then the computed value \( i \) is the inverse of \( x \) modulo \( m \): \( i \equiv x^{-1} \mod m \). It can be used for the so-called exact division: to compute the quotient \( a/x \) for a number \( a \) that is known to be divisible by \( x \), simply multiply by \( i \). This works because \( a = bx \) (\( a \) is divisible by \( x \)), so \( ai \equiv bxi \equiv b \mod m \).
### 1.22.2 Exact division by \( C = 2^k \pm 1 \)

We use the relation (for power series)

\[
\frac{A}{C} = \frac{A}{1 - Y} = A(1 + Y)(1 + Y^2)(1 + Y^4) \cdots (1 + Y^{2^n}) \mod x^{2^{n+1}} \tag{1.22-1}
\]

where \( Y = 1 - C \). The relation can be used for efficient exact division over \( \mathbb{Z} \) by \( C = 2^k \pm 1 \). For \( C = 2^k + 1 \) use

\[
\frac{A}{C} = A(1 - 2^k)(1 + 2^{k+2})(1 + 2^{k+4}) \cdots (1 + 2^{k+2^n}) \mod 2^N \tag{1.22-2}
\]

where \( k2^n \geq N \). For \( C = 2^k - 1 \) use \((A/C = -A/(-C))\)

\[
\frac{A}{C} = -A(1 + 2^k)(1 + 2^{k+2})(1 + 2^{k+4}) \cdots (1 + 2^{k+2^n}) \mod 2^N \tag{1.22-3}
\]

The equivalent method for exact division by polynomials (over GF(2)) is given in section 38.1.6 on page 823.

#### 1.22.3 Computation of the square root

<table>
<thead>
<tr>
<th>x inv</th>
<th>sqrt</th>
<th>x inv</th>
<th>sqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>...............................1</td>
<td>= 1</td>
<td>...............................1</td>
<td>= 5</td>
</tr>
<tr>
<td>...............................1</td>
<td>= -1</td>
<td>...............................1</td>
<td>= -5</td>
</tr>
<tr>
<td>...............................1</td>
<td>= -2</td>
<td>...............................1</td>
<td>= -6</td>
</tr>
<tr>
<td>...............................1</td>
<td>= -3</td>
<td>...............................1</td>
<td>= -7</td>
</tr>
<tr>
<td>...............................1</td>
<td>= -4</td>
<td>...............................1</td>
<td>= -9</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 1</td>
<td>...............................1</td>
<td>= 2</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 2</td>
<td>...............................1</td>
<td>= 3</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 3</td>
<td>...............................1</td>
<td>= 4</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 4</td>
<td>...............................1</td>
<td>= 5</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 5</td>
<td>...............................1</td>
<td>= 6</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 6</td>
<td>...............................1</td>
<td>= 7</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 7</td>
<td>...............................1</td>
<td>= 8</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 8</td>
<td>...............................1</td>
<td>= 9</td>
</tr>
<tr>
<td>...............................1</td>
<td>= 9</td>
<td>...............................1</td>
<td>= 1</td>
</tr>
</tbody>
</table>

Figure 1.22-A: Examples of the 2-adic inverse and square root of \( x \) where \(-9 \leq x \leq +9\). Where no inverse or square root is given, it does not exist.

With the inverse square root we choose the start value to match \( \lfloor d/2 \rfloor + 1 \) as that guarantees four bits of initial precision. Moreover, we get control to which of the two possible values the inverse square root is finally reached. The argument modulo 8 has to be equal to one.

```c
inline ulong invsqrt2adic(ulong d)
// Return inverse square root modulo 2**BITS_PER_LONG
// Must have: d==1 mod 8
// The number of correct bits is doubled with each step
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 4, 8, 16, 32, 64, ... bits (or better)
{
    if ( 1 != (d&7) ) return 0; // no inverse sqrt
    // start value: if d == ****10001 ==> x := ****1001
    ulong x = (d >> 1) | 1;
    ulong p, y;
    do
        y = x;
        p = (3 - d * y * y);
        x = (y * p) >> 1;
    while( (p & 7) != 1 );
    return x;
}
```

[fxtbook draft of 2008-August-17]
1.23: Radix $-2$ representation

The square root can be obtained by a final multiplication with $d$:

```c
inline ulong sqrt2adic(ulong d)
// Return square root modulo 2**BITS_PER_LONG
// Must have: d==1 mod 8 or d==4 mod 32, d==16 mod 128
// ... d==4*k mod 4**(k+3)
// Result undefined if condition does not hold
{
    if ( 0==d ) return 0;
    ulong s = 0;
    while ( 0==(d&1) ) { d >>= 1; ++s; }
    d *= invsqrt2adic(d);
    d <<= (s>>1);
    return d;
}
```

Note that the 2-adic square root is something completely different from the integer square root in general. If the argument $d$ is a perfect square then the result equals $\pm \sqrt{d}$. The output of the program [FXT: bits/bit2adic-demo.cc] is shown in figure 1.22-A. For further information on 2-adic (more generally $p$-adic) numbers see [172], [111], and also [168].

1.23 Radix $-2$ representation

The radix $-2$ representation of a number $n$ is

$$n = \sum_{k=0}^{\infty} t_k (-2)^k$$

where the $t_k$ are zero or one. For integers $n$ the sum is terminating: the highest nonzero $t_k$ is at most two positions beyond the highest bit of the binary representation of the absolute value of $n$ (with two’s complement).

1.23.1 Conversion from binary

Item 128 of [33] gives a surprisingly simple algorithm to obtain the coefficients $t_k$ of the radix $-2$ representation of a binary number [FXT: bits/negbin.h]:

```c
inline ulong bin2neg(ulong x)
// binary --> radix(-2)
{
    const ulong m = 0xaaaaaaaaUL; // 32 bit version
    x += m;
    x ^= m;
    return x;
}
```

An example:

$$14 \rightarrow \ldots 1 \ldots 1 = 16 - 2 = (-2)^4 + (-2)^1$$

The inverse routine is obtained by executing the inverse of the two steps in reversed order:

```c
inline ulong neg2bin(ulong x)
// radix(-2) --> binary
// inverse of bin2neg()
{
    const ulong m = 0xaaaaaaaaUL; // 32-bit version
    x ^= m;
    x -= m;
    return x;
}
```
Chapter 1: Bit wizardry

<table>
<thead>
<tr>
<th>k</th>
<th>bin(k)</th>
<th>m = bin2neg(k)</th>
<th>g = gray(m)</th>
<th>dec(g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>0 &lt;= 0</td>
</tr>
<tr>
<td>1</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>1 &lt;= 1</td>
</tr>
<tr>
<td>4</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>2 &lt;= 4</td>
</tr>
<tr>
<td>5</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>3 &lt;= 5</td>
</tr>
<tr>
<td>7</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>5 &lt;= 7</td>
</tr>
<tr>
<td>8</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>6 &lt;= 8</td>
</tr>
<tr>
<td>10</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>7 &lt;= 10</td>
</tr>
<tr>
<td>12</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>8 &lt;= 12</td>
</tr>
<tr>
<td>14</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>9 &lt;= 14</td>
</tr>
<tr>
<td>16</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>10 &lt;= 16</td>
</tr>
<tr>
<td>17</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>11 &lt;= 17</td>
</tr>
<tr>
<td>19</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>12 &lt;= 19</td>
</tr>
<tr>
<td>20</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>13 &lt;= 20</td>
</tr>
<tr>
<td>21</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>14 &lt;= 21</td>
</tr>
<tr>
<td>22</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>15 &lt;= 22</td>
</tr>
<tr>
<td>23</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>16 &lt;= 23</td>
</tr>
<tr>
<td>24</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>17 &lt;= 24</td>
</tr>
<tr>
<td>25</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>18 &lt;= 25</td>
</tr>
<tr>
<td>26</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>19 &lt;= 26</td>
</tr>
<tr>
<td>27</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>20 &lt;= 27</td>
</tr>
<tr>
<td>28</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>21 &lt;= 28</td>
</tr>
<tr>
<td>29</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>22 &lt;= 29</td>
</tr>
<tr>
<td>30</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>23 &lt;= 30</td>
</tr>
<tr>
<td>31</td>
<td>.......1</td>
<td>.......1</td>
<td>.......1</td>
<td>24 &lt;= 31</td>
</tr>
</tbody>
</table>

Figure 1.23-A: Radix −2 representations and their Gray codes. Lines ending in ‘<=N’ indicate that all values less or equal N occur in the last column up to that point.

Figure 1.23-A shows the output of the program [FXT: bits/negbin-demo.cc]. The sequence of Gray codes of the radix −2 representation is a Gray code for the numbers in the range 0, ..., k for the following values of k (entry A002450 of [245]):

\[ k = 1, 5, 21, 85, 341, 1365, 5461, 21845, 87381, 349525, 1398101, \ldots, (4^n - 1)/3 \]

1.23.2 Fixed points of the conversion *

<table>
<thead>
<tr>
<th>0</th>
<th>.......1</th>
<th>64: .......1</th>
<th>256: .......1</th>
<th>320: .......1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.......1</td>
<td>61: .......1</td>
<td>257: .......1</td>
<td>321: .......1</td>
</tr>
<tr>
<td>4</td>
<td>.......1</td>
<td>68: .......1</td>
<td>260: .......1</td>
<td>324: .......1</td>
</tr>
<tr>
<td>5</td>
<td>.......1</td>
<td>69: .......1</td>
<td>261: .......1</td>
<td>325: .......1</td>
</tr>
<tr>
<td>15</td>
<td>.......1</td>
<td>81: .......1</td>
<td>273: .......1</td>
<td>337: .......1</td>
</tr>
<tr>
<td>20</td>
<td>.......1</td>
<td>84: .......1</td>
<td>276: .......1</td>
<td>340: .......1</td>
</tr>
<tr>
<td>21</td>
<td>.......1</td>
<td>85: .......1</td>
<td>277: .......1</td>
<td>341: .......1</td>
</tr>
</tbody>
</table>

Figure 1.23-B: The fixed points of the conversion and their binary representations (dots denote zeros).

The sequence of fixed points of the conversion starts as

0, 1, 4, 5, 16, 17, 20, 21, 64, 65, 68, 69, 80, 81, 84, 85, 256, ... 

The binary representations have ones only at even positions (see figure 1.23-B). This is the Moser – De Bruijn sequence, entry A000695 of [245]. The generating function of the sequence is

\[ \frac{1}{1-x} \sum_{j=0}^{\infty} \frac{4^j x^{2j}}{1+x^{2j}} = x + 4x^2 + 5x^3 + 16x^4 + 17x^5 + 20x^6 + \ldots \]  

The sequence also appears as exponents in the power series

\[ \prod_{k=0}^{\infty} (1 + x^{4^k}) = 1 + x + x^4 + x^5 + x^{16} + x^{17} + x^{20} + x^{21} + x^{64} + x^{65} + x^{68} + \ldots \]  

[fxtbook draft of 2008-August-17]
The $k$-th fixed point can be obtained moving all bits of the binary representation of $k$ to position $2x$ where $x \geq 0$ is the index of the bit under consideration:

```c
inline ulong negbin_fixed_point(ulong k)
{
    return bit_zip0(k);
}
```

The used function is given in section 1.15 on page 40. The sequence of radix $-2$ representations of $0, 1, 2, \ldots$, interpreted as binary numbers, is entry A005351 of [245]:

$0,6,7,4,5,26,27,24,25,30,31,28,29,18,19,16,17,22,23,20,21,106,107,104,105,110,111,\ldots$

The corresponding sequence for the negative numbers $-1, -2, -3, \ldots$ is entry A005352:

$3,2,13,12,15,14,9,8,11,10,53,52,55,54,49,48,51,50,61,60,63,62,57,56,59,58,37,36,39,38,\ldots$

More information about ‘non-standard’ representations of numbers can be found in [172].

### 1.23.3 Generating negbin words in order

![Figure 1.23-C: Radix $-2$ representations of the numbers $0\ldots+63$ (top), and $0\ldots-63$ (bottom).](image)

A radix $-2$ representation can be incremented by the function [FXT: bits/negbin.h] (32-bit versions in what follows):

```c
inline ulong next_negbin(ulong x)
// With x the radix(-2) representation of n
// return radix(-2) representation of n+1.
{
    const ulong m = 0xaaaaaaaaUL;
    x ^= m;
    ++x;
    x ^= m;
    return x;
}
```

A version without constants is

```c
ulong s = x << 1;
ulong y = x ^ s;
y += 1;
s ^= y;
return s;
```

Decrementing can be done via

```c
inline ulong prev_negbin(ulong x)
// With x the radix(-2) representation of n
// return radix(-2) representation of n-1.
{
    const ulong m = 0xaaaaaaaaUL;
    x ^= m;
    --x;
    x ^= m;
    return x;
}
```
The functions are quite fast, about 440 million words per second are generated (5 cycles per increment or decrement). Figure 1.23-C shows the generated words in forward (top) and backward (bottom) order, it was created with the program [FXT: bits/negbin2-demo.cc].

### 1.24 A sparse signed binary representation

An algorithm to compute a representation of a number \( x \) as

\[
x = \sum_{k=0}^{\infty} s_k \cdot 2^k \quad \text{where} \quad s_k \in \{-1, 0, +1\}
\]

such that two consecutive digits \( s_k, s_{k+1} \) are never simultaneously nonzero is given in [221]. Figure 1.24-A gives the representation of several small numbers, it is the output of [FXT: bits/bin2naf-demo.cc].

We can convert the binary representation of \( x \) into a pair of binary numbers that correspond to the positive and negative digits [FXT: bits/bin2naf.h].

```c
inline void bin2naf(ulong x, ulong &np, ulong &nm)
// Compute (nonadjacent form, NAF) signed binary representation of x:
// the unique representation of x as
// x = \sum_{k}(d_k \cdot 2^k) \text{ where } d_j \in \{-1, 0, +1\}
// and no two adjacent digits d_j, d_{j+1} are both nonzero.
```
1.25: Generating bit combinations

6 // np has bits j set where \( d_j = +1 \)
7 // nm has bits j set where \( d_j = -1 \)
8 // Thereby: \( x = np - nm \)
9 {
10 ulong xh = x >> 1; // x/2
11 ulong x3 = x + xh; // 3*x/2
12 ulong c = xh ^ x3;
13 np = x3 & c;
14 nm = xh & c;
15 }

Converting back to binary is trivial:

1 inline ulong naf2bin(ulong np, ulong nm)
2 {
3 return ( np - nm );
4 }

The representation is one example of a nonadjacent form (NAF). A method for the computation of certain nonadjacent forms (\( w \)-NAF) is given in [209]. A Gray code for the signed binary words is described in section 12.5 on page 308.

![Figure 1.24-B: The numbers whose negative part in the NAF representation is zero.](image)

When a binary word contains no consecutive ones then the negative part of the NAF representation is zero. The sequence of values is \([0, 1, 2, 4, 5, 8, 9, 10, 16, \ldots]\), entry A003714 of [245], see figure 1.24-B.

The numbers are called the Fibbinary numbers.

1.25 Generating bit combinations

1.25.1 Co-lexicographic (colex) order

Given a binary word with \( k \) bits set the following routine computes the binary word that is the next combination of \( k \) bits in co-lexicographic order. In the co-lexicographic order the reversed sets are sorted, see figure 1.25-A.

The method to determine the successor is to determine the lowest block of ones and move its highest bit one position up. The rest of the block is then moved to the low end of the word [FXT: bits/bitcombcolex.h]:

1 inline ulong next_colex_comb(ulong x)
2 {
3 ulong r = x & -x; // lowest set bit
4 x += r; // replace lowest block by a one left to it
5

[fxtbook draft of 2008-August-17]
word = set = set (reversed)
...
111 = { 0, 1, 2 } = { 2, 1, 0 }
1... = { 0, 1, 3 } = { 3, 1, 0 }
1.1 = { 0, 2, 3 } = { 3, 2, 0 }
111. = { 1, 2, 3 } = { 3, 2, 1 }
1... = { 0, 1, 4 } = { 4, 1, 0 }
1.1. = { 0, 2, 4 } = { 4, 2, 0 }
11. = { 0, 3, 4 } = { 4, 2, 1 }
11.1 = { 1, 3, 4 } = { 4, 3, 1 }
111. = { 2, 3, 4 } = { 4, 3, 2 }
1... = { 0, 1, 5 } = { 5, 1, 0 }
1.1. = { 0, 2, 5 } = { 5, 2, 0 }
1... = { 1, 2, 5 } = { 5, 2, 1 }
1.1. = { 0, 3, 5 } = { 5, 3, 0 }
1.1. = { 1, 3, 5 } = { 5, 3, 1 }
11. = { 2, 3, 5 } = { 5, 3, 2 }
11.1 = { 0, 4, 5 } = { 5, 4, 0 }
11.1 = { 1, 4, 5 } = { 5, 4, 1 }
111. = { 2, 4, 5 } = { 5, 4, 2 }
1111 = { 0, 4, 5 } = { 5, 4, 3 }

Figure 1.25-A: Combinations \( \binom{6}{3} \) in co-lexicographic order. The reversed sets are sorted.

6    if ( 0==x ) return 0; // input was last combination
7    ulong z = x & -x; // first zero beyond lowest block
8    z -= r; // lowest block (cf. lowest_block())
9    while ( 0==(z&1) ) { z >>= 1; } // move block to low end of word
10   return x | (z>>1); // need one bit less of low block
11 }

One could replace the while-loop by a bit scan and shift combination. The combinations \( \binom{32}{20} \) are generated at a rate of about 142 million per second. The rate is about 120 M/s for the combinations \( \binom{32}{12} \), the rate with \( \binom{60}{7} \) is 70 M/s, and with \( \binom{60}{53} \) it is 160 M/s.

The following routine computes the predecessor of a combination:

1    inline ulong prev_colex_comb(ulong x)
2    { // Inverse of next_colex_comb()
3        x = next_colex_comb( ~x ); // first zero beyond lowest block
4        z = r; // lowest block (cf. lowest_block())
5        while ( 0==(z&1) ) { z >>= 1; } // move block to low end of word
6        return x | (z>>1); // need one bit less of low block
7    }

The first and last combination can be computed via

1    inline ulong first_comb(ulong k)
2    { // Return the first combination of (i.e. smallest word with) k bits,
3        // i.e. 00...001111..1 (k low bits set)
4        // Must have: 0 <= k <= BITS_PER_LONG
5            ulong t = ~0UL >> ( BITS_PER_LONG - k );
6        if ( k==0 ) t = 0; // shift with BITS_PER_LONG is undefined
7            return t;
8    }

and

1    inline ulong last_comb(ulong k, ulong n=BITS_PER_LONG)
2    { // return the last combination of (biggest n-bit word with) k bits
3        // i.e. 1111...100..00 (k high bits set)
4        // Must have: 0 <= k <= n <= BITS_PER_LONG
5            return first_comb(k) << (n - k);
6    }

The if-statement in first_comb() is needed because a shift by more than BITS_PER_LONG−1 is undefined by the C-standard, see section 1.1.5 on page 5.
1.25: Generating bit combinations

A variant of the presented (colex-) algorithm appears in [33 item 175]. The variant used here avoids the division of the HAKMEM-version and was given by Doug Moore and Glenn Rhoads. The listing in figure 1.25-A can be created with the program [FXT: bits/bitcombcolex-demo.cc]:

```cpp
ulong n = 6, k = 3;
ulong last = last_comb(k, n);
ulong g = first_comb(k);
ulong gg = 0;
do {
    // visit combination given as word g
    gg = g;
    g = next_colex_comb(g);
} while ( gg!=last );
```

1.25.2 Lexicographic (lex) order

The binary words corresponding to lexicographic order are obtained from the bit-reversed complements of the words corresponding to the co-lexicographic order of combinations \( \binom{n}{k} \). Note a more precise term for the order is subset-lex (for sets written with elements in increasing order). The sequence is identical to the delta-set-colex order backwards.

The program [FXT: bits/bitcomblex-demo.cc] shows how to compute the subset-lex sequence efficiently:

```cpp
ulong n = 5, k = 3;
ulong x = first_comb(n-k); // first colex (n-k choose n)
const ulong m = first_comb(n); // aux mask
const ulong l = last_comb(k, n); // last colex
ulong ct = 0;
ulong y;
do {
    y = revbin(~x, n) & m; // lex order
    // visit combination given as word y
    x = next_colex_comb(x);
} while ( y != 1 );
```

The bit-reversal routine `revbin()` is shown in section 1.14 on page 35. Sections 6.1.1 on page 176 and section 6.1.2 give iterative algorithms for combinations (represented by arrays) in lex and colex order, respectively.
Chapter 1: Bit wizardry

1.25.3 Shifts-order

Figure 1.25-C shows combinations in shifts-order. The order for combinations \( \binom{n}{k} \) is obtained from the shifts-order for subsets (section 8.4 on page 208) by discarding all subsets whose number of elements are \( \neq k \) and reversing the list order. The first combination is \([1^k0^{n-k}]\), and the successor is obtained as follows (see figure 1.25-D):

1. Easy case: if the rightmost one is not in position zero (least significant bit) then shift the word to the right and return the combination.
2. Finished?: if the combination is the last one (\([0^n]\), \([0^{n-1}]\), \([10^{n-k1^{k-1}}]\)) then return zero.
3. Shift back: shift the word to the left such that the leftmost one is in the leftmost position (this can be a no-op).
4. Simple split: if the rightmost one is not the least significant bit then move it one position to the right and return the combination.
5. Split second block: move the rightmost bit of the second block (from the right) of ones one position to the right and attach the lowest block of ones and return the combination.

An implementation is given in [FXT: bits/bitcombshifts.h]:

```plaintext
class bit_comb_shifts
{
  ulong x_; // the combination
  ulong s_; // how far shifted to the right
  ulong n_, k_; // combinations (n choose k)
  ulong last_; // last combination

  public:
    bit_comb_shifts(ulong n, ulong k)
    {
      n_ = n; k_ = k;
      first();
    }
```

Figure 1.25-C: Combinations \( \binom{n}{k} \), for \( k = 1, 2, 3, 4 \) in shifts-order.
1.26 Generating bit subsets of a given word

1.26.1 Counting order

In order to generate all bit-subsets of a binary word one can use the sparse counting idea shown in section 1.8.1 on page 21. The implementation is [FXT: class bit_subset in bits/bitsubset.h]:

```cpp
class bit_subset {
public:
  ulong u_; // current subset
  ulong v_; // the full set
};
```

The combinations \( \binom{32}{20} \) are generated at a rate of about 150 M/s, for the combinations \( \binom{32}{12} \) the rate is about 220 M/s [FXT: bits/bitcombshifts-demo.cc]. The rate with the combinations \( \binom{60}{7} \) is 415 M/s, and with \( \binom{60}{53} \) it is 110 M/s, the generation is very fast for the sparse case.
Chapter 1: Bit wizardry

```cpp
bit_subset(ulong v) : u_(0), v_(v) { ; }
~bit_subset() { ; }
ulong current() const { return u_; }
ulong prev() { u_ = (u_ - v_) & v_; return u_; }
ulong first(ulong v) { v_=v; u_=0; return u_; }
ulong first() { first(v_); return u_; }
ulong last(ulong v) { v_=v; u_=v; return u_; }
ulong last() { last(v_); return u_; }
}

With the word [...] the following sequence of words is produced by subsequent next()-calls:

```
....1.
...1.1.
...11:1.
...11:1.
```

A block of ones at the right will result in the binary counting sequence. About 1.1 billion subsets per second are generated with both next() and prev() [FXT: bits/bitsubset-demo.cc].

1.26.2 Gray code order

We use a method to isolate the changing bit from counting order that does not depend on shifting:

```
******0111 = u
******1000 = u+1
00000001111 = (u+1) ^ u
00000001000 = ((u+1) ^ u) & (u+1) <== bit to change
```

The method still works if the lowest one are separated by any amount of zeros. In fact, we want to find the single bit that changed from zero to one. The bits that are switched from zero to one in the transition from the word `A` to `B` can also be isolated via `X=B&~A`. The implementation is [FXT: class bit_subset in bits/bitsubset-gray.h]:

```cpp
class bit_subset_gray
{
public:
    bit_subset S_; // subsets in Gray code order
    ulong g_; // highest bit in S_.v_; needed for the prev() method

public:
    bit_subset_gray(ulong v) : S_(v), g_(0), h_(highest_one(v)) { ; }
    ~bit_subset_gray() { ; }
    ulong current() const { return g_; }
    ulong next()
    {
        ulong u0 = S_.current();
        if ( u0 == S_.v_ ) return first();
        ulong u1 = S_.next();
        ulong x = ~u0 & u1;
        g_ ^= x;
        return g_;
    }
    ulong first(ulong v) { S_.first(v); g_=highest_one(v); return g_; }
    ulong first() { S_.first(); g_=0; return g_; }
    [--snip--]
}
```

With the word [...] the following sequence of words is produced by subsequent next()-calls:

```
....1.
...1.1.
...11:1.
...11:1.
```

[fxtbook draft of 2008-August-17]
A block of ones at the right will result in the binary Gray code sequence, see [FXT: bits/bitsubset-gray-demo.cc]. The method `prev()` computes the previous word in the sequence, note the swapped roles of the variables \( u_0 \) and \( u_1 \):

```cpp
ulong prev()
{
    ulong u1 = S_.current();
    if ( u1 == 0 ) return last();
    ulong u0 = S_.prev();
    ulong x = ~u0 & u1;
    g_ ^= x;
    return g_;
}
```

About 365 million subsets are generated with both `next()` and `prev()`.

### 1.27 Binary words as subsets in lexicographic order

#### 1.27.1 Next and previous word in lexicographic order

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1 | 1. | . | .. | =
| 2 | 11. | . | .. | =
| 3 | 111. | . | .. | =
| 4 | 1111 | . | .. | =
| 5 | 11.1 | . | .. | =
| 6 | 1.1 | . | .. | =
| 7 | 1.11 | . | .. | =
| 8 | 1..1 | . | .. | =
| 9 | ..1 | . | .. | =
| 10 | ..11 | . | .. | =
| 11 | ..111 | . | .. | =
| 12 | ..1111 | . | .. | =
| 13 | ...1 | . | .. | =
| 14 | ...11 | . | .. | =
| 15 | ...111 | . | .. | =
|   | 8 | {0} |   |   |
|   | 12 | {0, 1} |   |   |
|   | 14 | {0, 1, 2} |   |   |
|   | 15 | {0, 1, 2, 3} |   |   |
|   | 13 | {0, 1, 3} |   |   |
|   | 10 | {0, 2} |   |   |
|   | 11 | {0, 2, 3} |   |   |
|   | 9 | {0, 3} |   |   |
|   | 4 | {1} |   |   |
|   | 6 | {1, 2} |   |   |
|   | 7 | {1, 2, 3} |   |   |
|   | 5 | {1, 3} |   |   |
|   | 2 | {2} |   |   |
|   | 3 | {2, 3} |   |   |
|   | 1 | {3} |   |   |

**Figure 1.27-A:** Binary words corresponding to non-empty subsets of the 4-element set in lexicographic order with respect to subsets. Note the first element of the subsets corresponds to the highest set bit.

The (bit-reversed) binary words in lexicographic order with respect to the subsets shown in figure 1.27-A can be generated by successive calls to the following function [FXT: bits/bitlex.h]:

```cpp
static inline ulong next_lexrev(ulong x)
// Return next word in subset-lex order.
{
    ulong x0 = x & -x; // lowest bit
    if ( 1!=x0 ) // easy case: set bit right of lowest bit
    {
        x0 >>= 1;
        x ^= x0; // set rightmost bit
    }
    else // lowest bit at word end
    {
        x = ~x0; // delete lowest bit
        x0 = x & -x; // new lowest bit ...
        x0 >>= 1; // ... is moved one to the right
        x -= x0;
        return x;
    }
}
```
Chapter 1: Bit wizardry

<table>
<thead>
<tr>
<th>[0: \ldots \ldots = 0 \ast]</th>
<th>16: \ldots \ldots = 17</th>
<th>32: \ldots \ldots = 33</th>
<th>48: \ldots \ldots = 51</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: \ldots 1 = 1 \ast</td>
<td>17: \ldots 1 = 19</td>
<td>33: \ldots 1 = 35</td>
<td>49: \ldots 1 = 50</td>
</tr>
<tr>
<td>2: \ldots \ldots = 1</td>
<td>18: \ldots \ldots = 18 \ast</td>
<td>34: \ldots \ldots = 34 \ast</td>
<td>50: \ldots \ldots = 53</td>
</tr>
<tr>
<td>3: \ldots 1 = 2</td>
<td>19: \ldots 11 = 21</td>
<td>35: \ldots 11 = 37 \ast</td>
<td>51: \ldots 1111 = 55</td>
</tr>
<tr>
<td>4: \ldots \ldots = 5</td>
<td>20: \ldots 111 = 23</td>
<td>36: \ldots 111 = 39</td>
<td>52: \ldots 111 = 54</td>
</tr>
<tr>
<td>5: \ldots 11 = 7</td>
<td>21: \ldots 111 = 27</td>
<td>37: \ldots 111 = 38</td>
<td>53: \ldots 111 = 52</td>
</tr>
<tr>
<td>6: \ldots 111 = 6 \ast</td>
<td>22: \ldots 111 = 20</td>
<td>38: \ldots 111 = 36</td>
<td>54: \ldots 1111 = 57</td>
</tr>
<tr>
<td>7: \ldots 1 = 4</td>
<td>23: \ldots 1111 = 25</td>
<td>39: \ldots 1111 = 41</td>
<td>55: \ldots 1111 = 59</td>
</tr>
<tr>
<td>8: \ldots \ldots = 10</td>
<td>24: \ldots 11111 = 27</td>
<td>40: \ldots 11111 = 43</td>
<td>56: \ldots 11111 = 58</td>
</tr>
<tr>
<td>9: \ldots 11 = 11</td>
<td>25: \ldots 11111 = 26</td>
<td>41: \ldots 11111 = 42</td>
<td>57: \ldots 111111 = 61</td>
</tr>
<tr>
<td>10: \ldots 111 = 10 \ast</td>
<td>26: \ldots 111111 = 29</td>
<td>42: \ldots 111111 = 45</td>
<td>58: \ldots 1111111 = 63</td>
</tr>
<tr>
<td>11: \ldots 1111 = 13</td>
<td>27: \ldots 1111111 = 31</td>
<td>43: \ldots 1111111 = 47</td>
<td>59: \ldots 11111111 = 62</td>
</tr>
<tr>
<td>12: \ldots 11111 = 15</td>
<td>28: \ldots 11111111 = 30</td>
<td>44: \ldots 11111111 = 46</td>
<td>60: \ldots 111111111 = 61</td>
</tr>
<tr>
<td>13: \ldots 111111 = 14</td>
<td>29: \ldots 111111111 = 31</td>
<td>45: \ldots 111111111 = 46</td>
<td>61: \ldots 1111111111 = 62</td>
</tr>
<tr>
<td>14: \ldots 1111111 = 12</td>
<td>30: \ldots 1111111111 = 28</td>
<td>46: \ldots 1111111111 = 44</td>
<td>62: \ldots 11111111111 = 63</td>
</tr>
<tr>
<td>15: \ldots 11111111 = 8</td>
<td>31: \ldots 11111111111 = 24</td>
<td>47: \ldots 11111111111 = 40</td>
<td>63: \ldots 111111111111 = 64</td>
</tr>
<tr>
<td>16: \ldots 111111111 = 4</td>
<td>32: \ldots 111111111111 = 16</td>
<td>48: \ldots 111111111111 = 49</td>
<td>64: \ldots 1111111111111 = 65</td>
</tr>
</tbody>
</table>

Figure 1.27-B: Binary words corresponding to the subsets of the 6-element set, as generated by `prev_lexrev()`. Fixed points are marked with asterisk.

The bit-reversed representation was chosen because the isolation of the lowest bit is often cheaper than the same operation on the highest bit. Starting with a one-bit word at position $n-1$ one generates the $2^n$ bit-subsets of length $n$. The function is used as follows [FXT: bits/bitlex-demo.cc]:

```cpp
ulong n = 4; // n-bit binary words
ulong x = 1UL<<(n-1); // first subset
do {
    // visit word x
} while ( (x=next_lexrev(x)) );
```

The following function goes backward:

```cpp
static inline ulong prev_lexrev(ulong x)
// Return previous word in subset-lex order.
{
    ulong x0 = x & -x; // lowest bit
    if ( x & (x0<<1) ) // easy case: next higher bit is set
        { x ^= x0; // delete lowest bit return x; }
    else
        { x += x0; // move lowest bit to the left x |= 1; // set rightmost bit return x; }
}
```

Starting with zero one obtains, by repeated calls to `prev_lexrev()`, a sequence of words that just before the $2^n$-th call has visited every word of length $n$. The generated sequence of words corresponding to subsets of the 6-element set is shown in figure 1.27-B. The sequence 1, 3, 2, 5, 7, 6, 4, 9, . . . in the right column is entry A108918 of [245].

The rate of generation using `next()` is about 274 million per second, and about 253 million per second with `next()`. An equivalent routine for arrays is given in section 8.1.2 on page 202. The routines turn out to be useful for a special version of fast Walsh transforms described in section 22.6.3 on page 472.

### 1.27.2 Conversion between binary- and lex-ordered words

A little contemplation on the structure of the binary words in lexicographic order leads to the routine that allows random access to the $k$-th lex-rev word (unrank algorithm) [FXT: bits/bitlex.h]:

```cpp
inline ulong negidx2lexrev(ulong k)
{
    ulong z = 0;
    ulong h = highest_one(k);
    while ( k )
```
Let the inverse function be $T(x)$, then we have $T(0) = 0$ and, with $h(x)$ being the highest power of two not greater than $x$,

$$T(x) = h(x) - 1 + \begin{cases} T(x - h(x)) & \text{if } x - h(x) \neq 0 \\ h(x) & \text{else} \end{cases} \tag{1.27-1}$$

We obtain the rank algorithm by starting with the lowest bit:

```c
inline ulong lexrev2negidx(ulong x)
{
    if ( 0==x ) return 0;
    ulong h = x & -x; // lowest bit
    ulong r = (h-1);
    while ( x^=h )
    {
        r += (h-1);
        h = x & -x; // next higher bit
    }
    r += h; // highest bit
    return r;
}
```

### 1.27.3 Minimal decompositions into terms $2^k - 1$

The least number of terms needed in the sum $x = \sum_k 2^k - 1$ equals the number of bits of the lex-word as shown in figure 1.27-C. The number can be computed as

```c
    c = bit_count( negidx2lexrev( x ) );
```

---

**Figure 1.27-C:** Binary words in subset-lex order and their bit counts (left columns). The least number of terms of the form $2^k - 1$ needed in the sum $x = \sum_k 2^k - 1$ (right columns) equals the bit count.
Alternatively, one can subtract the greatest integer of the form $2^k - 1$ until $x$ is zero and count the number of subtractions. The sequence of these numbers is entry A100661 of [245]:

$$1, 2, 1, 2, 3, 2, 1, 2, 3, 2, 3, 4, 3, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 2, 1, 2, 3, 2, 3, \ldots$$

The following function can be used to compute the sequence:

```c
1 void S(ulong f, ulong n) // A100661
2 {
3     static int s = 0;
4     ++s;
5     cout << s << ",";
6     for (ulong m=1; m<n; m<<=1) S(f+m, m);
7     --s;
8     cout << s << ",";
9 }
```

When called with arguments $f = 0$ and $n = 2^k$ it prints the first $2^k + 1 - 1$ numbers of the sequence followed by a zero.

A generating function of the sequence is given by

$$Z(x) := \frac{-1 + 2(1-x) \prod_{n=1}^{\infty} (1 + x^{2^n} - 1)}{(1-x)^2} = 1 + 2x + x^2 + 2x^3 + 3x^4 + 2x^5 + x^6 + 2x^7 + 3x^8 + 2x^9 + 3x^{10} + 4x^{11} + 3x^{12} + 2x^{13} + \ldots$$

### 1.27.4 The sequence of fixed points *

<table>
<thead>
<tr>
<th>0:</th>
<th>514:</th>
<th>.1.11...111.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>540:</td>
<td>.1.111111111.</td>
</tr>
<tr>
<td>10:</td>
<td>.11.</td>
<td>[--snip--]</td>
</tr>
<tr>
<td>18:</td>
<td>.11.</td>
<td>1556: .1.11.</td>
</tr>
<tr>
<td>34:</td>
<td>.11.</td>
<td>1972: .11.</td>
</tr>
<tr>
<td>60:</td>
<td>.111.</td>
<td>1604: .11.1.</td>
</tr>
<tr>
<td>66:</td>
<td>.11.</td>
<td>1668: .11.11.</td>
</tr>
<tr>
<td>92:</td>
<td>.11.</td>
<td>1796: .11.</td>
</tr>
<tr>
<td>116:</td>
<td>.111111.</td>
<td>2050: .11.</td>
</tr>
<tr>
<td>130:</td>
<td>.111111.</td>
<td>2076: .111111.</td>
</tr>
<tr>
<td>156:</td>
<td>.111111.</td>
<td>2092: .111111.</td>
</tr>
<tr>
<td>172:</td>
<td>.111111.</td>
<td>2100: .111111.</td>
</tr>
<tr>
<td>180:</td>
<td>.111111.</td>
<td>2124: .111111.</td>
</tr>
<tr>
<td>236:</td>
<td>.111111.</td>
<td>2132: .111111.</td>
</tr>
<tr>
<td>212:</td>
<td>.111111.</td>
<td>2148: .111111.</td>
</tr>
<tr>
<td>228:</td>
<td>.111111.</td>
<td>[--snip--]</td>
</tr>
<tr>
<td>258:</td>
<td>.111111.</td>
<td>4644: .111111.</td>
</tr>
<tr>
<td>284:</td>
<td>.111111.</td>
<td>4676: .111111.</td>
</tr>
<tr>
<td>300:</td>
<td>.111111.</td>
<td>4740: .111111.</td>
</tr>
<tr>
<td>308:</td>
<td>.111111.</td>
<td>4868: .111111.</td>
</tr>
<tr>
<td>332:</td>
<td>.111111.</td>
<td>5112: .111111.</td>
</tr>
<tr>
<td>392:</td>
<td>.111111.</td>
<td>5188: .111111.</td>
</tr>
<tr>
<td>404:</td>
<td>.111111.</td>
<td>5252: .111111.</td>
</tr>
<tr>
<td>452:</td>
<td>.111111.</td>
<td>5380: .111111.</td>
</tr>
</tbody>
</table>

Figure 1.27-D: Fixed points of the binary to lex-rev conversion.

The sequence of fixed points of the conversion to and from indices is 0, 1, 6, 10, 18, 34, 60, 66, 92, 108, 116, 130, 156, 172, 180, 204, 212, 228, 258, 284, 300, 308, 332, 340, 356, 396, 404, 420, 452, 514, 540, 556, \ldots is sequence A079471 of [245]. Their values as bit patterns are shown in figure 1.27-D. The crucial observation is that a word is a fixed point exactly if (it equals zero or) its bit-count equals $2^j$ where $j$ is the index of the lowest set bit.

Now we can find out whether $x$ is a fixed point of the sequence by the following function:

```c
1 static inline bool is_lexrev_fixed_point(ulong x)
2 // Return whether x is a fixed point in the prev_lexrev() - sequence
```
3 \{ 
4 \quad \text{if ( } x \& 1 \text{ )} 
5 \quad \quad \text{if ( } 1==x \text{ ) return true;} 
6 \quad \quad \text{else return false;} 
7 \} 
8 \text{else} 
9 \quad \quad \text{ulong } w = \text{bit}_{-}\text{count}(x); 
10 \quad \quad \text{if ( } w >_{-}\text{rev}(w \& -w) \text{ ) return false;} 
11 \quad \quad \text{if ( 0==x ) return true;} 
12 \quad \quad \text{return } 0 \neq ( (x \& -x) \& w ); 
13 \} 
14 \}

One can also use either of the following tests:

$$x == \text{negidx}\text{2lexrev}(x)$$
$$x == \text{lexrev}\text{2negidx}(x)$$

### 1.27.5 Recursive generation and relation to a power series *

The following function generates the bit-reversed binary words in reversed lexicographic order:

```c
void C(ulong f, ulong n, ulong w)
{
  for (ulong m=1; m<n; m<<=1) C(f+m, m, w^m);
  print_bin(" ", w, 10); // visit
}
```

Calling `C(0, 64, 0)` we obtain the list of words shown in figure 1.27-B with the all-zeros word moved to the last position. A slight modification of the function

```c
void A(ulong f, ulong n)
{
  cout << "1,"
  for (ulong m=1; m<n; m<<=1) A(f+m, m);
  cout << "0,"
}
```

generates the power series (sequence [A079559](http://oeis.org/A079559) of [245])

$$\prod_{n=1}^{\infty} \left(1 + x^{2^n-1}\right) = 1 + x + x^3 + x^4 + x^7 + x^8 + x^{10} + x^{11} + x^{15} + x^{16} + \ldots \quad (1.27-3)$$

Calling `A(0, 32)` we obtain:

$$1,1,0,1,1,0,0,1,1,0,1,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0,1,0,0,0,0,0, \ldots$$

Indeed, the lowest bit of the $k$-th word of the bit-reversed sequence in reversed lexicographic order equals the $(k-1)$-st coefficient in the power series. The sequence can also be generated with a string substitution engine (see chapter 16 on page 357).

```
Start: 1
Rules: 0 --> 0
   1 --> 110
: (#=2)
 1: (#=4)
 110
 2: (#=8)
  1101100
 3: (#=16)
 110110011011000
 4: (#=32)
 110110011011000110110011011000
 5: (#=64)
 1101100110110001101100110110001101100110110001101100110110001101100110110000
```

We note that the sequence of sums, prepended by one,

$$1 + x \prod_{n=1}^{\infty} \frac{(1 + x^{2^n-1})}{1 - x} = 1 + 1 + 2 x^2 + 2 x^3 + 3 x^4 + 4 x^5 + 4 x^6 + \ldots \quad (1.27-4)$$
has series coefficients

\[ 1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 6, 7, 8, 8, 8, 8, 9, 10, 10, 11, 12, 12, 12, 13, \ldots \]

This sequence is entry [A046699] of [245]. We have \( a(1) = a(2) = 1 \) and the sequence satisfies the peculiar recurrence

\[ a(n) = a(n-a(n-1)) + a(n-1-a(n-2)) \quad \text{for} \quad n > 2 \quad (1.27-5) \]

### 1.28 Minimal-change bit combinations

The wonderful routine [FXT: bits/bitcombminchange.h]

```c
1 static inline ulong igc_next_minchange_comb(ulong x)
2 { // Return the inverse Gray code of the next combination in minimal-change order.
3     // Input must be the inverse Gray code of the current combination.
4     ulong g = rev_gray_code(x);
5     ulong i = 2;
6     ulong cb; // == candidate bits;
7     do
8     { //
9         ulong y = (x & ~(i-1)) + i;
10        ulong j = lowest_one(y) << 1;
11        ulong h = !!(y & j);
12        cb = ((j-h) ^ g) & (j-i);
13        i = j;
14     } while ( 0==cb );
15     return x + lowest_one(cb);
16 }
```

together with

```c
1 static inline ulong igc_last_comb(ulong k, ulong n)
2 { // Return the (inverse Gray code of the) last combination
3     // as in igc_next_minchange_comb()
4     if ( 0==k ) return 0;
5     ulong f = first_sequency(k); // = (by Doug Moore)
6     ulong c = first_comb(n);
7     return c ^ (f>>1);
8     // = (((1UL<<n) - 1) ^ (((1UL<<k) - 1) / 3));
9 }
```

(the function `first_sequency()` is defined in section 1.17 on page 48) can be used as suggested by the routine

```c
1 static inline ulong next_minchange_comb(ulong x, ulong last)
2 { // Not efficient, just to explain the usage of igc_next_minchange_comb()
3     // Must have: last==igc_last_comb(k, n)
4     x = inverse_gray_code(x);
5     if ( x==last ) return 0;
6     x = igc_next_minchange_comb(x);
7     return gray_code(x);
8 }
```

Each combination is different from the preceding one in exactly two positions. For example, using $n = 5$ and $k = 3$ we obtain

\[
\begin{array}{c}
x \quad \text{inverse_gray_code}(x) \\
1.111 \quad .1.1 = \text{first_sequency}(k) \\
11.11 \quad 1.11 \\
11.111 \quad 11.1 \\
111.1 \quad 111.1 \\
11111.1 \quad 11111.1 \\
111111.1 \quad \text{igc_last_comb}(k, n) \\
\end{array}
\]

The same run of bit combinations could be obtained by going through the Gray codes and omitting all words where the bit-count is not equal to $k$. The algorithm shown here, however, is much more efficient.

For reasons of efficiency one may prefer code as

```c
ulong last = igc_last_comb(k, n);
ulong c, nc = first_sequency(k);
do {
    c = nc;
    nc = igc_next_minchange_comb(c);
    ulong g = gray_code(c);
    // Here g contains the bit-combination
} while ( c!=last );
```

which avoids the repeated computation of the inverse Gray code.

The algorithm in `igc_next_minchange_comb()` uses the fact that the difference of two (inverse Gray codes of) successive combinations is always a power of two, see figure 1.28-A. See also [FXT: bits/bitcombminchange-demo.cc]. Using this observation one can derive a different version that checks the pattern of the change:

```c
static inline ulong igc_next_minchange_comb(ulong x)
// Alternative version.
{
    ulong gx = gray_code( x );
    ulong i = 2;
do {
        ulong y = x + i;
        i <<= 1;
        ulong gy = gray_code( y );
        ulong r = gx ^ gy;
        // Check that change consists of exactly one bit
    } while ( c!=last );
```

Figure 1.28-A: Minimal-change combinations, their inverse Gray codes and the differences of the inverse Gray codes. The differences are powers of two.
if ( is_pow_of_2( r & gy ) && is_pow_of_2( r & gx ) ) break;
// is_pow_of_2(x):=((x & -x) == x) returns 1 also for x==0.
// But this cannot happen for both tests at the same time
while ( 1 );
return y;

This version is the fastest, the combinations \( \binom{32}{12} \) are generated at a rate of about 96 million per second, the combinations \( \binom{32}{20} \) at a rate of about 83 million per second.

Here is another version which needs the number of set bits as a second parameter:

```
static inline ulong igc_next_minchange_comb(ulong x, ulong k)
// Alternative version, uses the fact that the difference
// of two successive x is the smallest possible power of 2.
{
ulong y, i = 2;
do
{
y = x + i;
i <<= 1;
} while ( bit_count( gray_code(y) ) != k );
return y;
}
```

The routine will be fast if the CPU used has a bitcount instruction. The necessary modification for the generation of the previous combination is trivial:

```
static inline ulong igc_prev_minchange_comb(ulong x, ulong k)
// Returns the inverse graycode of the previous combination in minimal-change order.
// Input must be the inverse graycode of the current combination.
// With input==first the output is the last for n=BITS_PER_LONG
{
ulong y, i = 2;
do
{
y = x - i;
i <<= 1;
} while ( bit_count( gray_code(y) ) != k );
return y;
}
```

### 1.29 Fibonacci words

A Fibonacci word is a word that does not contain two successive ones. Whether a given binary word is a Fibonacci word can be tested with the function \([FXT: \texttt{bits/fibrep.}\texttt{h}]\):

```
inline bool is_fibrep(ulong f)
{
return ( 0==(f&(f>>1)) );
}
```

#### 1.29.1 Lexicographic order

To generate all Fibonacci words use the following functions from \([FXT: \texttt{bits/fibrep.}\texttt{h}]\). For forward order (see figure \[1.29-A\]):

```
inline ulong next_fibrep(ulong x)
// With x the Fibonacci representation of n
// return Fibonacci representation of n+1.
{
// 2 examples: // ex. 1 // ex. 2
// x == [\star]0 010101 // x == [\star]0 01010
ulong y = x | (x>>1); // y == [\star]? 011111 // y == [\star]? 011111
ulong z = y + 1; // z == [\star]? 100000 // z == [\star]? 10000
```

[fxtbook draft of 2008-August-17]
1.29: Fibonacci words

<table>
<thead>
<tr>
<th>0:</th>
<th>1:</th>
<th>2:</th>
<th>3:</th>
<th>4:</th>
<th>5:</th>
<th>6:</th>
<th>7:</th>
<th>8:</th>
<th>9:</th>
<th>10:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
<td>1..</td>
</tr>
<tr>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
<td>.1.</td>
</tr>
<tr>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
</tr>
<tr>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
<td>1..1</td>
</tr>
<tr>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
<td>.1.1</td>
</tr>
<tr>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
<td>1.1.1</td>
</tr>
<tr>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
<td>.1.1.1</td>
</tr>
<tr>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
<td>1..1.1</td>
</tr>
<tr>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
<td>.1..1.1</td>
</tr>
<tr>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
<td>1.1..1.1</td>
</tr>
</tbody>
</table>

Figure 1.29-A: All 55 Fibonacci words with 8 bits in lexicographic order.

```cpp
9  z = z & -z; // z == [0]0 100000 // z == [0]0 10000
10 x ^= z;   // x == [*]0 110101 // x == [*]0 11010
11 x &= ~(z-1); // x == [0]0 100000 // x == [0]0 10000
12 return x;
```

The routine can be used as shown in [FXT: bits/fibrep2-demo.cc]:

```cpp
ulong n = 7;
const ulong f = 1UL << n;
ulong t = 0;
do {
  // visit(t)
  t = next_fibrep(t);
} while ( t!=f );
```

The reversed order can be obtained via

```cpp
ulong f = 1UL << n;
while ( f ) {
  ulong t = prev_fibrep(f);
  f = t;
  // visit(t)
}
```

which uses the function (64-bit version)

```cpp
inline ulong prev_fibrep(ulong x)
// With x the Fibonacci representation of n
// return Fibonacci representation of n-1.
{
  ulong i = lowest_one_idx(x);
  x ^= (1UL<<i);
  ++i;
  x ^= (0x5555555555555555UL >> (BITS_PER_LONG-i));
  return x;
}
```

The forward version generates about 200 million words per second, the backward version about 135 million words per second.

1.29.2 Gray code order

A Gray code for the binary Fibonacci words (shown in figure 1.29-B) can be obtained by using the Gray code of the radix $\omega$ representations (see section 1.28 on page 65) of binary words whose difference is of the form

```
0: 1: 2: 3: 4:
     1.. | 1.. | 1.. | 1.. | 1.. |
     .1. | .1. | .1. | .1. | .1. |
     1.1 | 1.1 | 1.1 | 1.1 | 1.1 |
     .1.1 | .1.1 | .1.1 | .1.1 | .1.1 |
     1..1 | 1..1 | 1..1 | 1..1 | 1..1 |
     .1..1 | .1..1 | .1..1 | .1..1 | .1..1 |
     1...1 | 1...1 | 1...1 | 1...1 | 1...1 |
```

[fxtbook draft of 2008-August-17]
Chapter 1: Bit wizardry

Table 1.29-B: Gray code for the binary Fibonacci words (rightmost column).

```
<table>
<thead>
<tr>
<th>j</th>
<th>k(j)</th>
<th>k(j)-k(j-1)</th>
<th>x=bin2neg(k)</th>
<th>gray(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
<td>1111</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>1.1</td>
<td>11</td>
<td>11111</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>1.11</td>
<td>1111</td>
<td>1111111</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>1.111</td>
<td>111111</td>
<td>11111111</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>1.1111</td>
<td>1111111</td>
<td>111111111</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>1.11111</td>
<td>11111111</td>
<td>1111111111</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>1.111111</td>
<td>111111111</td>
<td>11111111111</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>1.1111111</td>
<td>1111111111</td>
<td>111111111111</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>1.11111111</td>
<td>11111111111</td>
<td>1111111111111</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>1.111111111</td>
<td>111111111111</td>
<td>11111111111111</td>
<td>31</td>
</tr>
<tr>
<td>12</td>
<td>1.1111111111</td>
<td>1111111111111</td>
<td>111111111111111</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>1.11111111111</td>
<td>11111111111111</td>
<td>1111111111111111</td>
<td>29</td>
</tr>
<tr>
<td>14</td>
<td>1.111111111111</td>
<td>111111111111111</td>
<td>11111111111111111</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>1.1111111111111</td>
<td>1111111111111111</td>
<td>111111111111111111</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>1.11111111111111</td>
<td>11111111111111111</td>
<td>1111111111111111111</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>1.111111111111111</td>
<td>111111111111111111</td>
<td>11111111111111111111</td>
<td>12</td>
</tr>
<tr>
<td>18</td>
<td>1.1111111111111111</td>
<td>1111111111111111111</td>
<td>111111111111111111111</td>
<td>11</td>
</tr>
<tr>
<td>19</td>
<td>1.11111111111111111</td>
<td>11111111111111111111</td>
<td>1111111111111111111111</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>1.111111111111111111</td>
<td>111111111111111111111</td>
<td>11111111111111111111111</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1.1111111111111111111</td>
<td>1111111111111111111111</td>
<td>111111111111111111111111</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>1.11111111111111111111</td>
<td>11111111111111111111111</td>
<td>1111111111111111111111111</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>1.111111111111111111111</td>
<td>111111111111111111111111</td>
<td>11111111111111111111111111</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>1.1111111111111111111111</td>
<td>1111111111111111111111111</td>
<td>111111111111111111111111111</td>
<td>7</td>
</tr>
<tr>
<td>25</td>
<td>1.11111111111111111111111</td>
<td>11111111111111111111111111</td>
<td>1111111111111111111111111111</td>
<td>6</td>
</tr>
<tr>
<td>26</td>
<td>1.111111111111111111111111</td>
<td>111111111111111111111111111</td>
<td>11111111111111111111111111111</td>
<td>19</td>
</tr>
<tr>
<td>27</td>
<td>1.1111111111111111111111111</td>
<td>1111111111111111111111111111</td>
<td>111111111111111111111111111111</td>
<td>15</td>
</tr>
<tr>
<td>28</td>
<td>1.11111111111111111111111111</td>
<td>11111111111111111111111111111</td>
<td>1111111111111111111111111111111</td>
<td>14</td>
</tr>
<tr>
<td>29</td>
<td>1.111111111111111111111111111</td>
<td>111111111111111111111111111111</td>
<td>11111111111111111111111111111111</td>
<td>17</td>
</tr>
<tr>
<td>30</td>
<td>1.1111111111111111111111111111</td>
<td>11111111111111111111111111111111</td>
<td>111111111111111111111111111111111</td>
<td>16</td>
</tr>
</tbody>
</table>

The algorithm is to try these values as increments starting from the least, same as for the minimal-change combination described in section 1.28 on page 78. The next valid word is encountered if it is a valid Fibonacci word, that is, if it does not contain two consecutive set bits. The implementation is [FXT: class bit_fibgray in [bits/bitfibgray.h].

```

```
More than 100 million words per second are generated. The program [FXT: bits/bitfibgray-demo.cc] shows how to use the class, figure 1.29-B was created with it. Section 12.1 on page 300 gives a recursive algorithm for Fibonacci words in Gray code order.

### 1.30 Binary words and parentheses strings *

A subset of the binary words can be interpreted as a (well formed) string of parentheses. The 4-bit binary words that have this property are marked with ‘P’ in figure 1.30-A (left) [FXT: bits/parenword-demo.cc]. The strings are constructed by scanning the word from the low end and printing a ‘(’ with each one and a ‘)’ with each zero. In order to find out when to terminate one adds up +1 for each opening parenthesis and −1 for a closing parenthesis. When the ones in the binary word have been scanned then s closing parentheses have to be added where s is the value of the sum [FXT: bits/parenwords.h]:

```c
inline void parenword2str(ulong x, char *str)
{
    int s = 0;
    ulong j = 0;
    for (j=0; x!=0; ++j)
    {
        s += ( x&1 ? +1 : -1 );
        str[j] = "("[x&1];
        x >>= 1;
    }
    while ( s-- > 0 ) str[j++] = ’)’; // finish string
    str[j] = 0; // terminate string
}
```

---

**Figure 1.30-A:** Left: some of the 4-bit binary words can be interpreted as a string parentheses (marked with ‘P’). Right: all 5-bit words that correspond to well-formed parentheses strings.
Chapter 1: Bit wizardry

The 5-bit binary words that are valid ‘paren words’ together with the corresponding strings are shown in figure 1.30-A (right). Note that the lower bits in the word (right end) correspond to the beginning of the string (left end). If a negative value for the sums occurs at any time of the computation then the word is not a paren word. We use that fact to create a routine that determines whether a word is a paren word:

```cpp
inline bool is_parenword(ulong x)
{
    int s = 0;
    for (ulong j=0; x!=0; ++j)
    {
        s += (x&1 ? +1 : -1);
        if (s<0) break; // invalid word
        x >>= 1;
    }
    return (s>=0);
}
```

The sequence

1, 3, 5, 7, 11, 13, 15, 19, 21, 23, 27, 29, 31, 39, 43, 45, 51, 53, 55, 59, 61, 63, ...

of nonzero integers \( x \) so that \( \text{is}_\text{parenword}(x) \) returns true is entry A036991 of [245]. If we fix the number of paren pairs then the following functions generate the least and biggest valid binary word. The function simply returns a word with a block of \( n \) ones at the low end:

```cpp
inline ulong first_parenword(ulong n)
// Return least binary word corresponding to \( n \) pairs of parens.
// Example, \( n=5 \): .....11111 ((((()))))
{
    return first_comb(n);
}
```

The last paren word is the word with a sequence of \( n \) blocks ’01’ at the low end:

```cpp
inline ulong last_parenword(ulong n)
// Return biggest binary word corresponding to \( n \) pairs of parens.
// Must have: 1 <= \( n \) <= BITS_PER_LONG/2.
// Example, \( n=5 \): .1.1.1.1.1 ()()()()()
{
    return 0x5555555555555555UL >> (BITS_PER_LONG-2*n);
}
```

The sequence of all binary words corresponding to \( n \) pairs of parens in colex order can be generated with the following (slightly cryptic) function:

```cpp
inline ulong next_parenword(ulong x)
// Next (colex order) binary word that is a paren word.
{
    if (x & 2) // Easy case, move highest bit of lowest block to the left:
    {
        ulong b = lowest_zero(x);
        x ^= b;
        x ^= (b>>1);
        return x;
    }
    else // Gather all low "01"s and split lowest nontrivial block:
    {
```
define {  
  if ( 0==(x & (x>>1)) ) return 0;  
  ulong w = 0; // word where the bits are assembled  
  ulong s = 0; // shift for lowest block  
  ulong i = 1; // == lowest_one(x)  
  do // collect low "01"s:  
  {  
    x ^= i;  
    w <<= 1;  
    w |= 1;  
    ++s;  
    i <<= 2; // == lowest_one(x);  
  }  
  while ( 0==(x&(i<<1)) );  
  ulong z = x ^ (x+i); // lowest block  
  x ^= z;  
  z &= (z>>1);  
  z &= (z>>1);  
  w = (z>>s);  
  x |= w;  
  w ^= (z>>s);  
  x |= w;  
  return x;  
}  

The program [FXT: bits/parenword-colex-demo.cc] shows how to create a list of binary words corresponding to \( n \) pairs of parens (code slightly shortened):

\[
\text{ulong } n = 4; \quad \text{// Number of paren pairs} \\
\text{ulong } pn = 2*n+1; \\
\text{char } *str = \text{new char}[n+1]; \quad \text{str}[n] = 0; \\
\text{ulong } x = \text{first_parenword}(n); \\
\text{while ( } x \text{ )} \\
{  
  \text{print_bin(" ", } x, pn);  
  \text{parenword2str(x, str);}  
  \text{cout } << " \text{=} " \text{<< str } << \text{endl; }  
}  
\text{x = next_parenword(x);}  
\]

Its output with \( n = 5 \) is shown in figure 1.30-B. The 1,767,263,190 paren words for \( n = 19 \) are generated at a rate of about 169 million words per second. Chapter 13 on page 317 gives a different formulation of the algorithm.

Knuth gives [174] a very elegant routine for generating the next paren word, the comments are MMIX instructions:

\[
\text{inline ulong next_parenword(ulong } x \text{)} \\
\{  
  \text{const ulong } m0 = -1UL/3; 
  \text{ulong } t = x \text{ } m0; \quad \text{// XOR } t, x, m0; 
  \text{if ( } (t&x)=0 \text{ ) return 0; } \quad \text{// current is last} 
  \text{ulong } u = (t-1) \text{ } t; \quad \text{// SUBU } u, t, 1; \quad \text{XOR } u, t, u; 
  \text{ulong } v = x \text{ } u; \quad \text{// OR } v, x, u; 
  \text{ulong } y = \text{bit_count( } u \text{ } m0 \text{ ); } \quad \text{// SADD } y, u, m0; 
  \text{ulong } w = v + 1; \quad \text{// ADDU } w, v, 1; 
  \text{t = v } \& \text{ } w; \quad \text{// ANDN } t, v, w; 
  \text{y = t } >> \text{ } y; \quad \text{// SRU } y, t, y; 
  \text{y += w; } \quad \text{// ADDU } w, y, y; 
  \text{return } y; 
\}
\]

The routine is slower, however, about 81 million words per second are generated. A bit-count instruction in hardware would speed it up significantly. By treating the case of easy update separately as in the other version, a rate of about 137 million words per second is obtained.
1.31 Permutations via primitives

We give two methods to specify permutations of the bits of a binary word via one or more control words. The methods are suggestions for machine instructions that can serve as primitives for permutations of the bits of a word.

A restricted method

Generalizing the symbolic-powering idea given in section 1.19 on page 51 we can specify a subset of all permutations by selecting bit-blocks of the masks (for 32-bit words)

```
.............1111111111111111........11111111........11111111....1111....1111....1111....1111..11..11..11..11..11..11..11..11.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1
```

Subsets of the blocks of the masks can be determined with the bits of a 32-bit word by considering the highest bit of each block:

```
.............1............... bits 15 ...........1...............1....... bits 7 .......1.......1.......1... bits 3 11 ....1...1...1...1...1...1...1...1. bits 1 5 9 13 ....1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 bits 0 2 4 6 8 10 12 14 ...
```

Thereby one can use all (except for the highest) bits of a word to select the blocks where the bits defined by the block and those left to it should be swapped. A implementation of the implied algorithm can be found in [FXT: bits/bitperm1-demo.cc]. Arrays are used to give more readable code:

```c
void perm1(uchar *a, ulong ldn, const uchar *x)
// Permute a[] according to the 'control word' x[].
// The length of a[] must be 2**ldn.
{
    long n = 1L<<ldn;
    for (long s=n/2; s>0; s/=2)
    {
        for (long k=0; k<n; k+=s+s)
        {
            if ( x[k+s-1]!="0" )
                { // swap regions [a+k,...,a+k+s-1], [a+k+s,...,a+k+2*s-1]:
                    swap(a+k, a+k+s, s);
                }
        }
    }
}
```

No attempt has been made to optimize or parallelize the algorithm. We just explore how useful a machine instruction for the permutation of bits would be.

The program uses a fixed size of 16 bits, an ‘x’ is printed whenever the corresponding bit is set:

```
a=0123456789ABCDEF bits of the input word
x=0010011000110110 control word
8:  2  5  9  13  6x  8  3 10 12  14
```

The control word used leads to the Gray code permutation (see 2.8 on page 103). Assume we use words with \( N \) bits. We cannot (for \( N > 2 \)) obtain all \( N! \) permutations as we can choose between only \( 2^{N-1} \) control words. Now set the word length to \( N := 2^n \). The reachable permutations are those where the intervals \([k \cdot 2^j, \ldots, (k + 1) \cdot 2^j - 1]\) contain all numbers \([p \cdot 2^j, \ldots, (p + 1) \cdot 2^j - 1]\) for all \( j \leq n \) and \( 0 \leq k < 2^{n-j} \), choosing \( p \) for each interval arbitrarily \((0 \leq p < 2^{n-j})\). For example, the lower half of the permuted array must contain a permutation of either the lower or the upper half \((j = n-1)\) and each pair \(a_{2j}, a_{2j+1}\) must contain two elements \(2z, 2z + 1\) \((j = 1)\). The bit-reversal can be obtained using a control word with all bits set. Alas, the (important!) zip permutation (bit-zip, see section 1.15 on page 40) is unreachable.
The inverse permutation is achieved by changing a single line:

```java
for (long s=1; s<n; s+=s)
```

A machine instruction could choose between the two routines via the highest bit in the control word.

## A general method

All permutations of \( N = 2^n \) elements can be obtained using \( n \) control words of \( N \) bits. Assume we have a machine instruction that collects bits according to a control word. An eight bit example:

```plaintext
a = abcdefgh  input data
x = ..1.11.1  control word (dots for zeros)
  cefh  bits of a where x has a one
  abdg  bits of a where x has a zero
  abdefh result, bits separated according to x
```

We need \( n \) such instructions that work on all length-2\( k \) sub-words for \( 1 \leq k \leq n \). For example, the instruction working on half words of a 16-bit word would work as

```plaintext
a = abcdefgh ABCDEFGH  input data
x = ..1.11.1 1111.... control word (dots for zeros)
  cefh ABCD bits of a where x has a one
  abdg EFGH bits of a where x has a zero
  abdefh EFGHABCD result, bits separated according to x
```

Note the bits of the different sub-words are not mixed. Now all permutations can be reached if the control word for the 2\( k \)-bit sub-words have exactly 2\( k - 1 \) bits set in all ranges \([j \cdot 2^k, \ldots, (j+1) \cdot 2^k]\).

A control word together with the specification of the instruction used defines the action taken. The following leads to a swap of adjacent bit pairs

\[
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.  \quad k = 1 \quad (2\text{-bit sub-words})
\]

while this

\[
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.  \quad k = 5 \quad (32\text{-bit sub-words})
\]

results in gathering the even and odd indexed bits in the half-words.

A complete set of permutation primitives for 16-bit words and their effect on a symbolic array of bits (split into groups of four elements for readability) is

\[
\begin{align*}
11111111\ldots & \quad k = 4 \quad \Rightarrow \quad 0123 \quad 4567 \quad 89ab \quad cdef \\
1111\ldots1111\ldots & \quad k = 3 \quad \Rightarrow \quad 1357 \quad 9bdf \quad 0246 \quad 8ace \\
11.11.11.11\ldots & \quad k = 2 \quad \Rightarrow \quad f7b3 \quad 5d91 \quad 6e2a \quad 4c80 \\
1.1.1.1.1.1.1.1 & \quad k = 1 \quad \Rightarrow \quad fedc \quad ba98 \quad 7654 \quad 3210
\end{align*}
\]

The top primitive leads to a swap of the left and right half of the bits, the next to a swap of the halves of the half-words and so on. The obtained permutation is array reversal. Note that we use array notation (least index left) here.

The resulting permutation depends on the order in which the primitives are used. Starting with full words

\[
\begin{align*}
1.1.1.1 \quad 1.1.1.1 & \quad k = 4 \quad \Rightarrow \quad 0123 \quad 4567 \quad 89ab \quad cdef \\
1.1.1.1 \quad 1.1.1.1 & \quad k = 3 \quad \Rightarrow \quad 1357 \quad 9bdf \quad 0246 \quad 8ace \\
1.1.1.1 \quad 1.1.1.1 & \quad k = 2 \quad \Rightarrow \quad f7b3 \quad 5d91 \quad 6e2a \quad 4c80 \\
1.1.1.1 \quad 1.1.1.1 & \quad k = 1 \quad \Rightarrow \quad fedc \quad ba98 \quad 7654 \quad 3210
\end{align*}
\]

The result is different when starting with 2-bit sub-words:

\[
\begin{align*}
1.1.1.1 \quad 1.1.1.1 & \quad k = 1 \quad \Rightarrow \quad 0123 \quad 4567 \quad 89ab \quad cdef \\
1.1.1.1 \quad 1.1.1.1 & \quad k = 2 \quad \Rightarrow \quad 0123 \quad 4567 \quad 8a9b \quad cdef \\
1.1.1.1 \quad 1.1.1.1 & \quad k = 4 \quad \Rightarrow \quad 3715 \quad bf9d \quad 2604 \quad ae8c
\end{align*}
\]

There are \( \binom{2^z}{z} \) possibilities to have \( z \) bits set in a 2\( z \)-bit word. There are 2\( n-k \) length-2\( k \) sub-words in a
Chapter 1: Bit wizardry

2\(^n\)-bit word so the number of valid control words for that step is

\[\left\lfloor \frac{2^k}{2^{k-1}} \right\rfloor^{2^{n-k}}\]

The product of the number of valid words in all steps gives the number of permutations:

\[ (2^n)! = \prod_{k=1}^{n} \left\lfloor \frac{2^k}{2^{k-1}} \right\rfloor^{2^{n-k}} \quad (1.31-1) \]

1.32 CPU instructions often missed

Essential

- Bit-shift and bit-rotate instructions that work properly for shifts greater or equal to the word length: the shift instruction should zero the word, the rotate instruction should take the shift modulo word length. The C-language standards leave the results for these operations undefined and compilers simply emit the corresponding assembler instructions. The resulting CPU dependent behavior is both a source of errors and makes certain optimizations impossible.
- A bit-reverse instruction. A fast byte-swap mitigates the problem, see section 1.14 on page 35.
- Instructions that return the index of highest or lowest set bit in a word.
- Fast conversion from integer to float and double (both directions).
- A fused multiply-add instruction for floats.
- Instructions for the multiplication of complex floating point numbers. For example, with 128-bit (SIMD) registers and 64-bit floats:
  
  ```c
  // Here: R1 == A, B;  R2 == C, D;
  CMUL R1, R2;
  // Here: R2 == A*C-B*D, A*D+B*C;
  ```
- A sum-diff instruction, such as:
  
  ```c
  // Here: R1 == A;  R2 == B;
  SUMDIFF R1, R2;
  // Here: R1 == A+B;  R2 == A-B;
  ```

  It serves as a primitive for fast orthogonal transforms.
- An instruction to swap registers. Even better, a conditional version of that.

Nice to have

- A parity bit for the complete machine word. The parity of a word is the number of bits modulo two, not the complement of it. Even better would be an instruction for the inverse Gray code, see section 1.16 on page 42.
- A bit-count instruction, see section 1.8 on page 19. This would also give the parity at bit zero.
- A random number generator: LHCAs (see section 39.8 on page 875) may be candidates.
- A conditional version of more than just the move instruction, possibly as an instruction prefix.
- An instruction to detect zero bytes in a word, see section 1.21 on page 60. The C-convention is to use a zero byte as string terminator. Performance of the string related functions in the C-library could thereby be increased significantly. Ideally the instruction should exist for different word sizes: 4-byte, 8-byte and 16-byte (possibly using SIMD extensions).
1.32: CPU instructions often missed

- A bit-zip and a bit-unzip instruction, see section 1.15 on page 40. Note this is polynomial squaring over GF(2).

- A bit-gather and a bit-scatter instruction, see [FXT: bits/bitgather.h] and [FXT: bits/bitseparate.h]. This would include bit-zip and its inverse.

- Primitives for permutations of bits, see section 1.31 on page 86.

- Multiplication corresponding to XOR as addition. That is, a multiplication without carries, the one used for polynomials over GF(2). See section 38.1 on page 819 and [FXT: bitpol_mult()] in [bpol/bitpol-arith.h].
Chapter 2

Permutations

We first describe several special permutations that are needed for FFT algorithms, like the revbin permutation and the Gray permutation. We describe permutations in general together with the operations on them, like composition and inversion.

Algorithms for the generation of all permutations of a given number of objects are given in chapter 10. The connection to mixed radix numbers in factorial base is given in section 10.3.

2.1 The revbin permutation

The permutation that swaps elements whose binary indices are mutual reversals is called revbin permutation (sometimes also bit-reversal or bitrev permutation). For example, for length $n = 256$ the element with index $x = 43_{10} = 00101011_2$ is swapped with the element whose index is $\tilde{x} = 11010100_2 = 212_{10}$.

Figure 2.1-A: Permutation matrices of the revbin permutation for sizes 16, 8 and 4. The permutation is self-inverse.

The permutation that swaps elements whose binary indices are mutual reversals is called revbin permutation (sometimes also bit-reversal or bitrev permutation). For example, for length $n = 256$ the element with index $x = 43_{10} = 00101011_2$ is swapped with the element whose index is $\tilde{x} = 11010100_2 = 212_{10}$. Note that $\tilde{x}$ depends on both $x$ and on $n$. Pseudo code for a naive implementation is

```
1  procedure revbin_permute(a[], n)
2    // a[0..n-1] input,result
3    { for x:=0 to n-1
4        { r := revbin(x, n)
5            if r>x then swap(a[x], a[r])
6        }
7    }
```

[fxtbook draft of 2008-August-17]
Chapter 2: Permutations

The condition \( r > x \) before the `swap()` statement makes sure that the swapping is not undone later when the loop variable \( x \) has the value of the present \( r \).

### 2.1.1 Computation using revbin-update

The key ingredient for a fast permutation routine is the observation that we only need to update the bit-reversed values: given \( \tilde{x} \) we can compute \( \tilde{x} + 1 \) efficiently as described in section 1.14.3 on page 38. A faster routine will be of the form

```plaintext
procedure revbin_permute(a[], n)

// a[0..n-1] input,result

if n<=2 return

r := 0 // the reversed 0

for x:=1 to n-1

r := revbin_upd(r, n/2)

if r>x then swap(a[x], a[r])

}

About \((n - \sqrt{n})/2 \) `swap()`-statements will be executed with the revbin permutation of \( n \) elements. That is, almost every element is moved for large \( n \), as there are only few numbers with symmetric bit patterns:

<table>
<thead>
<tr>
<th>( n )</th>
<th>2 # swaps</th>
<th># symm. pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>64</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>2^{10}</td>
<td>992</td>
<td>32</td>
</tr>
<tr>
<td>2^{20}</td>
<td>0.999 · 2^{20}</td>
<td>2^{10}</td>
</tr>
<tr>
<td>( n \rightarrow \infty )</td>
<td>( n - \sqrt{n} )</td>
<td>( \sqrt{n} )</td>
</tr>
</tbody>
</table>

The sequence is entry A045687 of [245]:

0, 2, 4, 12, 24, 56, 112, 238, 480, 992, 1980, 4032, 8064, 16242, 32512, 65280, ...

### 2.1.2 Exploiting the symmetries of the permutation

Symmetry can be used for further optimization: if for even \( x < n/2 \) there is a swap for the pair \( x, \tilde{x} \) then there is also a swap for the pair \( n - 1 - x, n - 1 - \tilde{x} \). As \( x < n/2 \) and \( \tilde{x} < n/2 \) one has \( n - 1 - x > n/2 \) and \( n - 1 - \tilde{x} > n/2 \). That is, the swaps are independent. A routine that uses these observations is

```plaintext
procedure revbin_permute(a[], n)

if n<=2 return

nh := n/2

r := 0 // the reversed 0

x := 1

while x<nh

// x odd:

r := r + nh

swap(a[x], a[r])

x := x + 1

// x even:

r := revbin_upd(r, n/2)

if r>x then

swap(a[x], a[r])

swap(a[n-1-x], a[n-1-r])

}

x := x + 1

[ffxtbook draft of 2008-August-17]
The revbin permutation

The code above can be used to derive an optimized version for zero padded data (used with linear convolution, see section 21.1.4 on page 440):

```c
procedure revbin_permute0(a[], n)
{
    if n<=2  return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        a[r] := a[x]
        a[x] := 0
        x := x + 1
        // x even:
        r := revbin_upd(r, n)
        if r>x then swap(a[x], a[r])
        // Omit swap of a[n-1-x] and a[n-1-r] as both are zero
        x := x + 1
    }
}
```

One can carry the scheme further, distinguishing whether \( x \mod 4 = 0, 1, 2 \) or 3. This is done in the implementation [FXT: revbin_permute() in perm/revbinpermute.h]. The parameters

```c
#define RBP_SYMM 4  // 1, 2, 4 (default is 4)
#define FAST_BIT_SCAN // define if machine has fast bit scan
```

determine how much of the symmetry is used (RBP_SYMM) and which flavor of the revbin-update routine is chosen (FAST_BIT_SCAN). With a fast bit-scan instruction a table driven version is slightly faster. We further define a macro to swap elements:

```c
#define idx_swap(k, r) { ulong kx=(k), rx=(r); swap2(f[kx], f[rx]); }
```

The main routine uses unrolled versions of the revbin permutation for small values of \( n \). These are given in [FXT: perm/shortrevbinpermute.h]. For example, the unrolled routine for \( n = 16 \) is

```c
template <typename Type>
inline void revbin_permute_16(Type *f)
{
    swap2(f[1], f[8]);
    swap2(f[2], f[4]);
    swap2(f[3], f[12]);
    swap2(f[5], f[10]);
    swap2(f[7], f[14]);
    swap2(f[11], f[13]);
}
```

The code was generated with the program [FXT: perm/cycles-demo.cc], see section 2.10.2 on page 111.

The routine `revbin_permute_le_64(f, n)` that is called for \( n \leq 64 \) selects the correct routine for the parameter \( n \):

```c
template <typename Type>
void revbin_permute(Type *f, ulong n)
{
    if ( n<=64 )
    {
        revbin_permute_le_64(f, n);
        return;
    }
    [--snip--]
```

If the table driven update is used, the table and some auxiliary variables have to be initialized:

```c
const ulong nh = (n>>1);
ifndef FAST_BIT_SCAN
make_revbin_upd_tab(nh);
#endif
#ifndef RBP_SYMM >= 2
#define RBP_SYMM >= 2
#endif
```

[fxtbook draft of 2008-August-17]
In what follows we set \( RBP\text{-SYMM} \) to 4 and omit the corresponding preprocessor statements. The main loop is

```c
ulong k = 0, r = 0;
while ( k<n/RBP\text{-SYMM} ) // n\geq16, n/2\geq8, n/4\geq4
{
    // ----- k\%4 == 0:
    if ( r>k )
    {
        idx_swap(k, r); // <nh, <nh 11
        idx_swap(n1\text{-}k, n1\text{-}r); // >nh, >nh 00
        idx_swap(nx1\text{-}k, nx1\text{-}r); // <nh, <nh 11
        idx_swap(nx2\text{-}k, nx2\text{-}r); // >nh, >nh 00
    }
    r ^= nh;
    ++k;
    // ----- k\%4 == 1:
    if ( r>k )
    {
        idx_swap(k, r); // <nh, >nh 10
        idx_swap(n1\text{-}k, n1\text{-}r); // >nh, <nh 01
    }
    #ifdef FAST\_BIT\_SCAN
    r = revbin\_tupd(r, k);
    #else
    r = revbin\_upd(r, nh);
    #endif
    ++k;
    // ----- k\%4 == 2:
    if ( r>k )
    {
        idx_swap(k, r); // <nh, <nh 11
        idx_swap(n1\text{-}k, n1\text{-}r); // >nh, >nh 00
    }
    r ^= nh;
    ++k;
    // ----- k\%4 == 3:
    if ( r>k )
    {
        idx_swap(k, r); // <nh, >nh 10
        idx_swap(nx1\text{-}k, nx1\text{-}r); // <nh, >nh 10
    }
    #ifdef FAST\_BIT\_SCAN
    r = revbin\_tupd(r, k);
    #else
    r = revbin\_upd(r, nh);
    #endif
    ++k;
} // end revbin\_permute()
```

The routines `make\_revbin\_upd\_tab()` for initializing the table and the table driven update routine `revbin\_tupd()` are given in section 1.14.3 on page 38.

It turns out that the routine takes, for large \( n \), about six times of the simple `reverse()` operation that swaps elements \( k \) with \( n - k - k \). Much of the time is spend waiting for memory which suggests that further optimizations would best be attempted with special machine instructions to bypass the cache or with non-temporal writes.

The routine [FXT: `revbin\_permute0()` in `perm/revbinpermute0.h`] is a specialized version optimized for zero padded data. Some memory access can be avoided for that case. For example, revbin-pairs with both indices larger than \( n/2 \) need no processing at all. Therefore the routine is faster than the general version.
2.2: The radix permutation

2.1.3 A pitfall

If, for complex data, one works with separate arrays for the real and imaginary parts one can remove half of the bookkeeping as follows:

```c
procedure revbin_permute(a[], b[], n)
{
  if n<=2  return
  r := 0 // the reversed 0
  for x:=1 to n-1
  {
    r := revbin_upd(r, n/2) // inline me
    if r>x then
      { swap(a[x], a[r])
        swap(b[x], b[r])
      }
  }
}
```

If both real and imaginary part fit into level-1 cache the method can lead to a speedup. However, for large arrays the routine can be drastically slower than two separate calls of the simple method: with FFTs the real and imaginary element for the same index typically lie apart in memory by a power of two, leading to a high percentage of cache misses with large array sizes.

2.2 The radix permutation

The radix permutation is the generalization of the revbin permutation to arbitrary radices. Pairs of elements are swapped when their indices, written in radix \( r \), are reversed. For example, in radix 10 and \( n = 1000 \) the elements with indices 123 and 321 will be swapped. The radix permutation is self-inverse.

C++ code for the radix \( r \) permutation of the array \( f[] \) is given in [FXT: perm/radixpermute.h]. The routine must be called with \( n \) a perfect power of the radix \( r \). Using radix \( r = 2 \) gives the revbin permutation.

```c
extern ulong radix_permute_nt[]; // == 9, 90, 900, ... for r=10
extern ulong radix_permute_kt[]; // == 1, 10, 100, ... for r=10
#define NT radix_permute_nt
#define KT radix_permute_kt

template <typename Type>
void radix_permute(Type *f, ulong n, ulong r)
{
  ulong x = 0;
  NT[0] = r-1;
  KT[0] = 1;
  while ( 1 )
  {
    ulong z = KT[x] * r;
    if ( z>n ) break;
    ++x;
    KT[x] = z;
  
    NT[x] = NT[x-1] * r;
  }
  // here: n == p**x
  for (ulong i=0, j=0; i < n-1; i++)
  {
    if ( i<j ) swap2(f[i], f[j]);
    ulong t = x - 1;
    ulong k = NT[t]; // =\(^{\ast}\) k = (r-1) * n / r;
    while ( k<=j )
    {
      j -= k;
      k = NT[--t]; // =\(^{\ast}\) k /= r;
    }
  }
```

[fxtbook draft of 2008-August-17]
2.3 In-place matrix transposition

Transposing a matrix is easy when it is not done in-place. The simple routine [FXT: \texttt{transpose()} in \texttt{aux2/transpose.h}] does the job:

```cpp
template <typename Type>
void transpose(const Type * restrict f, Type * restrict g, ulong nr, ulong nc)
// Transpose nr x nc matrix f[] into an nc x nr matrix g[].
{
    for (ulong r=0; r<nr; r++)
    {
        ulong isrc = r * nc;
        ulong idst = r;
        for (ulong c=0; c<nc; c++)
        {
            g[idst] = f[isrc];
            isrc += 1;
            idst += nr;
        }
    }
}
```

Matters get more complicated for the in-place equivalent. We have to find the cycles (see section \ref{sec:permutations} on page \pageref{sec:permutations}) of the underlying permutation. To transpose a $n_r \times n_c$-matrix first identify the position $i$ of the entry in row $r$ and column $c$:

$$i = r \cdot n_c + c \quad (2.3-1)$$

After the transposition the element will be at position $i'$ in the transposed $n'_r \times n'_c$-matrix

$$i' = r' \cdot n'_c + c' \quad (2.3-2)$$

We have $r' = c$, $c' = r$, $n'_r = n_c$ and $n'_c = n_r$, so:

$$i' = c \cdot n_r + r \quad (2.3-3)$$

Multiply the last equation by $n_c$

$$i' \cdot n_c = c \cdot n_r \cdot n_c + r \cdot n_c \quad (2.3-4)$$

With $n := n_r \cdot n_c$ and $r \cdot n_c = i - c$ we obtain

$$i' \cdot n_c = c \cdot n + i - c \quad (2.3-5)$$

$$i = i' \cdot n_c - c \cdot (n - 1) \quad (2.3-6)$$

Take the equation modulo $n - 1$ to obtain

$$i \equiv i' \cdot n_c \mod n - 1 \quad (2.3-7)$$

That is, the transposition moves the element $i = i' \cdot n_c$ to position $i'$. Multiply by $n_c$ to obtain the inverse:

$$i \cdot n_r \equiv i' \cdot n_c \cdot n_r \equiv i' \cdot (n - 1 + 1) \equiv i' \quad (2.3-8)$$

That is, element $i$ will be moved to $i' = i \cdot n_r \mod n - 1$. The routine [FXT: \texttt{transpose()} in \texttt{aux2/transpose.h}] uses the a bit-array to keep track of the elements that have been processed so far:
2.4: The triple reversion technique *

2.4.1 Triple reversion

We give a method to rotate an length-\( n \) array by \( s \) positions that does not require any scratch space. The trick is to use \texttt{reverse()} three times as in the following [FXT: \texttt{rotate\_left()} in \texttt{perm/rotate.h}]:

\begin{verbatim}
    [ 1 2 3 4 5 6 7 8 ] original array
    [ 3 2 1 4 5 6 7 8 ] reverse first 3 elements
    [ 3 2 4 1 5 6 7 8 ] reverse last 8-3=5 elements
    [ 4 5 6 7 8 1 2 3 ] reverse whole array
\end{verbatim}

\begin{verbatim}
    [ 1 2 3 4 5 6 7 8 ] original array
    [ 5 4 3 2 1 6 7 8 ] reverse first 8-3=5 elements
    [ 5 4 3 2 1 8 7 6 ] reverse last 3 elements
    [ 6 7 8 1 2 3 4 5 ] reverse whole array
\end{verbatim}

**Figure 2.4-A:** Rotation of an length-8 array by 3 positions to the left (top), and right (bottom).

Note that one should take care of possible overflows in the calculation \( i \cdot n_c \). For the case that \( n \) is a power of two (and so are both \( n_r \) and \( n_c \)) the multiplications modulo \( n - 1 \) are cyclic shifts. Thus any overflow can be avoided and the computation is also significantly cheaper. An implementation is given in [FXT: \texttt{aux2/transpose2.h}].
template <typename Type>
void rotate_left(Type *f, ulong n, ulong s)  
// Rotate towards element #0
// Shift is taken modulo n
{
  if ( s==0 ) return;
  if ( s>=n ) {
    if (n<2) return;
    s %= n;
  }
  reverse(f, s);
  reverse(f+s, n-s);
  reverse(f, n);
}

The technique is usually called the triple reversion trick. For example, left-rotating an 8-element array by 3 positions is achieved by the steps shown in figure 2.4-A (top). A right rotation of an n-element array by s positions is identical to a left rotation by n − s positions (bottom of figure 2.4-A):

template <typename Type>
void rotate_right(Type *f, ulong n, ulong s)  
// Rotate away from element #0
// Shift is taken modulo n
{
  if ( s==0 ) return;
  if ( s>=n ) {
    if (n<2) return;
    s %= n;
  }
  reverse(f, n-s);
  reverse(f+n-s, s);
  reverse(f, n);
}

2.4.2 Matrix transposition with dimensions powers of two

When transposing an \( n_r \times n_c \) matrix where both \( n_r \) and \( n_c \) are powers of two then the rotation is done with the binary indices of the elements. The revbin permutation reverses the complete indices and reversing a few bits at the least significant end is not any harder:

template <typename Type>
void revbin_permute_rows(Type *f, ulong ldn, ulong ldnc)  
// Revbin-permute the length 2**ldnc rows of f[0..2**ldn-1]
// (f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix)
{
  ulong n = 1UL<<ldn;
  ulong nc = 1UL<<ldnc;
  for (ulong k=0; k<n; k+=nc) revbin_permute(f+k, nc);
}

The routine to transpose a matrix is

template <typename Type>
void transpose_by_rbp(Type *f, ulong ldn, ulong ldnc)  
// Transpose f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix
{
  revbin_permute_rows(f, ldn, ldnc);
  ulong n = 1UL<<ldn;
  revbin_permute(f, n);
  revbin_permute_rows(f, ldn, ldn-ldnc);  // ... that is, columns
}

2.4.3 Quadruple reversion

The triple-reversion trick can also be used to swap two blocks in an array: first reverse the three ranges (first blocks, range between blocks, last block), then reverse the range that consists of all three. This is
The *quadruple reversion trick*. The corresponding code is given in \[\text{FXT: perm/swapblocks.h}\]:

\begin{verbatim}
template <typename Type>
void swap_blocks(Type *f, ulong x1, ulong n1, ulong x2, ulong n2)
// Swap the blocks starting at indices x1 and x2
// n1 and n2 are the block lengths
{
  if ( x1>x2 ) { swap2(x1,x2); swap2(n1,n2); }
  f += x1;
  x2 -= x1;
  ulong n = x2 + n2;
  reverse(f, n1);
  reverse(f+n1, n-n1-n2);
  reverse(f+x2, n2);
  reverse(f, n);
}
\end{verbatim}

The elements before \(x1\) and after \(x2+n2\) are not accessed. An example \[\text{FXT: perm/swap-blocks-demo.cc}\] is shown in figure 2.4-B. The effect of \(\text{swap Blocks}(f, x1, n1, x2, n2)\) can be undone with the statement \(\text{swap Blocks}(f, x1, n2, x2+n2-n1, n1)\).

## 2.5 The zip permutation

The *zip permutation* moves the elements from the lower half to the even indices and the elements from the upper half to the odd indices. Symbolically,

\[
\begin{bmatrix}
  a & b & c & d & A & B & C & D \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
  a & A & b & B & c & C & d & D \\
\end{bmatrix}
\]

The size of the array must be even. A routine for the permutation is

\begin{verbatim}
template <typename Type>
void zip(const Type * restrict f, Type * restrict g, ulong n)
\end{verbatim}
Chapter 2: Permutations

When the array size is a power of two we can use a special case of the technique from section 2.4.2 on page 98 compute the zip permutation in-place. We obtain the permutation as transposition of a $2 \times n/2$-matrix [FXT: zip() in perm/zip.h]:

```cpp
template <typename Type>
void zip(Type *f, ulong n)
{
  ulong nh = n/2;
  revbin_permute(f, nh); revbin_permute(f+nh, nh);
  revbin_permute(f, n);
}
```

The inverse of the zip permutation is the unzip permutation. We only give the in-place version based on the revbin permutation, the array size must be a power of two:

```cpp
template <typename Type>
void unzip(Type *f, ulong n)
{
  ulong nh = n/2;
  revbin_permute((Complex *)f, n/2);
  revbin_permute(f, n);
}
```

If we have a type Complex consisting of two doubles lying contiguous in memory we can optimize the procedures:

```cpp
void zip(double *f, long n)
{
  revbin_permute(f, n);
  revbin_permute((Complex *)f, n/2);
}

void unzip(double *f, long n)
{
  revbin_permute((Complex *)f, n/2);
  revbin_permute(f, n);
}
```

For arrays whose size $n$ is not a power of two the in-place zip permutation can be obtained by transposing the data as a $2 \times n/2$ matrix:

$$\text{transpose}(f, 2, n/2); \quad // \Rightarrow \text{zip}(f, n)$$

The routines for in-place transposition are given in section 2.3 on page 96. The inverse is obtained by transposing the data as a $n/2 \times 2$ matrix:

$$\text{transpose}(f, n/2, 2); \quad // \Rightarrow \text{unzip}(f, n)$$

While the above mentioned technique is usually not a gain for doing a transposition it may be used to speed up the revbin permutation itself. Let $U(n)$ be the revbin permutation of a length-$n$ array, $T(n_r, n_c)$ the transposition of the $n_r \times n_c$ matrix and $R(n_c)$ the operation done by revbin_permute_rows() (see section 2.4 on page 97). If $n = n_r \times n_c$ we have

$$T(n_r, n_c) = R(n_r) \cdot U(n) \cdot R(n_c)$$  \hspace{1cm} (2.5-1)

The $R$-operators are their own inverses while $T$ is not self inverse unless $n_r = n_c$.

$$U(n) = R(n_r) \cdot T(n_r, n_c) \cdot R(n_c)$$  \hspace{1cm} (2.5-2)

There is a degree of freedom in this formula: for fixed $n = n_r \times n_c$ one can choose one of $n_r$ and $n_c$. 

[fxtbook draft of 2008-August-17]
2.6 The reversed zip permutation

A permutation closely related to the zip permutation is the reversed zip permutation. It moves the lower half of an array to the even indices and the upper half to the odd indices in reversed order. Symbolically,

\[
\begin{bmatrix}
  a & b & c & d & A & B & C & D \\
\end{bmatrix} \rightarrow
\begin{bmatrix}
  a & D & b & C & c & B & d & A \\
\end{bmatrix}
\]

A C++ routine is [FXT: zip_rev() in perm/ziprev.h]:

```cpp
1 template<typename Type>
2 void zip_rev(const Type * restrict x, Type * restrict y, ulong n)
3 // n must be even
4 {
5    const ulong nh = n/2;
6    for (ulong k=0, k2=0; k<nh; k++, k2+=2) y[k2] = x[k];
7    for (ulong k=nh, k2=n-1; k<n; k++, k2-=2) y[k2] = x[k];
8 }
```

The in-place version can, if the array length is a power of two, be implemented as

```cpp
1 template<typename Type>
2 void zip_rev(Type *x, ulong n)
3 // n must be a power of two
4 {
5    const ulong nh = n/2;
6    reverse(x+nh, nh);
```
The inverse permutation [FXT: unzip_rev() in perm/ziprev.h] can be implemented as

```cpp
template <typename Type>
void unzip_rev(const Type * restrict x, Type * restrict y, ulong n)
// n must be even
{
    const ulong nh = n/2;
    for (ulong k=0, k2=0; k<nh; k++, k2+=2) y[k] = x[k2];
    for (ulong k=nh, k2=n-1; k<n; k++, k2-=2) y[k] = x[k2];
}
```

The in-place version is

```cpp
template <typename Type>
void unzip_rev(Type *x, ulong n)
// n must be a power of two
{
    const ulong nh = n/2;
    revbin_permute(x, n);
    revbin_permute(x, nh); revbin_permute(x+nh, nh);
    reverse(x+nh, nh);
}
```

The given permutation is used in an algorithm where the cosine transform is computed using the Hartley transform, see section 24.11 on page 528.

We write $Z$ and $Z^{-1}$ for the zip permutation and its inverse, $\overline{Z}$ and $\overline{Z}^{-1}$ for the reversed zip permutation and its inverse, and $R$ for the revbin permutation. Then the following relations hold:

\[
Z = R Z^{-1} R = \overline{Z} Z^{-1} \overline{Z} \quad (2.6-1a) \\
\overline{Z} = R \overline{Z}^{-1} R = Z \overline{Z}^{-1} Z \quad (2.6-1b) \\
Z^{-1} = R Z R = \overline{Z}^{-1} Z \overline{Z} \quad (2.6-1c) \\
\overline{Z}^{-1} = R \overline{Z} R = Z^{-1} Z \overline{Z} \quad (2.6-1d)
\]

### 2.7 The XOR permutation

![Permutation matrices of the XOR permutation for length 8 with parameter $x = 0 \ldots 7$. Compare to the table for the dyadic convolution shown in figure 22.7-A on page 474.](image)

[fxtbook draft of 2008-August-17]
The **XOR permutation** may be explained most simply by its trivial implementation: [FXT: xor_permute() in perm/xorpermute.h]:

```cpp
template <typename Type>
void xor_permute(Type *f, ulong n, ulong x)
{
    if ( 0==x ) return;
    for (ulong k=0; k<n; ++k)
    {
        ulong r = k^x;
        if ( r>k ) swap2(f[r], f[k]);
    }
}
```

The XOR permutation is self-inverse. The array length \(n\) must be divisible by the smallest power of two that is greater than \(x\). For example, \(n\) must be even if \(x = 1\), \(n\) must be divisible by four if \(x = 2\) or \(x = 3\). With \(n\) a power of two and \(x < n\) one is on the safe side.

The XOR permutation contains a few other permutations as important special cases (for simplicity assume that the array length \(n\) is a power of two): when the third argument \(x\) equals \(n-1\) then the permutation is the reversion, with \(x = 1\) neighboring even and odd indexed elements are swapped, with \(x = n/2\) the upper and the lower half of the array are swapped.

One has

\[
X_a X_b = X_b X_a = X_c \quad \text{where} \quad c = a \oplus b
\] (2.7-1)

For the special case \(a = b\) the relation expresses the self-inverse property as \(X_0\) is the identity. The XOR permutation often occurs in relations between other permutations where we will use the symbol \(X_x\), the subscript denoting the third argument in the given routine.

### 2.8 The Gray code permutation

The **Gray code permutation** (or simply **Gray permutation**) reorders (length-2^\(n\)) arrays according to the binary Gray code described in section 1.16 on page 42. A routine for the permutation is [FXT: perm/graypermute.h]:

```cpp
template <typename Type>
inline void gray_permute(const Type *f, Type * restrict g, ulong n)
// Put Gray permutation of f[] to g[], i.e. g[gray_code(k)] == f[k]
{
    for (ulong k=0; k<n; ++k) g[gray_code(k)] = f[k];
}
```

| 0: [ * ] | 0: [ * ] |
| 1: [ * ] | 1: [ * ] |
| 2: [ ] | 2: [ ] |
| 3: [ * ] | 3: [ ] |
| 4: [ ] | 4: [ * ] |
| 5: [ ] | 5: [ ] |
| 6: [ * ] | 6: [ ] |
| 7: [ ] | 7: [ ] |
| 8: [ ] | 8: [ ] |
| 9: [ ] | 9: [ ] |
| 10: [ ] | 10: [ ] |
| 11: [ ] | 11: [ ] |
| 12: [ ] | 12: [ ] |
| 13: [ ] | 13: [ ] |
| 14: [ ] | 14: [ ] |
| 15: [ ] | 15: [ ] |

**Figure 2.8-A:** Permutation matrices of the Gray code permutation (left) and its inverse (right).

The **Gray code permuta**tion (or simply **Gray permutation**) reorders (length-2^n) arrays according to the binary Gray code described in section 1.16 on page 42. A routine for the permutation is [FXT: perm/graypermute.h]:

```cpp
template <typename Type>
inline void gray_permute(const Type *f, Type * restrict g, ulong n)
// Put Gray permutation of f[] to g[], i.e. g[gray_code(k)] == f[k]
{
    for (ulong k=0; k<n; ++k) g[gray_code(k)] = f[k];
}
```
Its inverse is

```cpp
template <typename Type>
inline void inverse_gray_permute(const Type *f, Type * restrict g, ulong n)
    // Put inverse Gray permutation of f[] to g[], i.e. g[k] == f[gray_code(k)]
    // (same as: g[inverse_gray_code(k)] == f[k])
{
    for (ulong k=0; k<n; ++k) g[k] = f[gray_code(k)];
}
```

We again use calls to `gray_code()` because they are cheaper than the computation of `inverse_gray_code()`.

We now give in-place versions of the above routines that offer very good performance. It is necessary to identify the cycle leaders (see section 2.10 on page 109) of the permutation and find an efficient way to generate them.

### 2.8.1 Cycles of the permutation

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Elements</th>
<th>Cycle Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>( 2, 3)</td>
<td>#=2</td>
</tr>
<tr>
<td>1:</td>
<td>( 4, 7, 5, 6)</td>
<td>#=4</td>
</tr>
<tr>
<td>2:</td>
<td>( 8, 15, 10, 12)</td>
<td>#=4</td>
</tr>
<tr>
<td>3:</td>
<td>( 9, 14, 11, 13)</td>
<td>#=4</td>
</tr>
<tr>
<td>4:</td>
<td>(16, 31, 21, 25, 17, 30, 20, 24)</td>
<td>#=8</td>
</tr>
<tr>
<td>5:</td>
<td>(18, 28, 23, 26, 19, 29, 22, 27)</td>
<td>#=8</td>
</tr>
<tr>
<td>6:</td>
<td>(32, 63, 42, 51, 34, 60, 40, 48)</td>
<td>#=8</td>
</tr>
<tr>
<td>7:</td>
<td>(33, 62, 43, 50, 35, 61, 41, 49)</td>
<td>#=8</td>
</tr>
<tr>
<td>8:</td>
<td>(36, 56, 47, 53, 38, 59, 45, 54)</td>
<td>#=8</td>
</tr>
<tr>
<td>9:</td>
<td>(37, 57, 46, 52, 39, 58, 44, 55)</td>
<td>#=8</td>
</tr>
<tr>
<td>10:</td>
<td>(64,127, 85,102, 68,120, 80, 96)</td>
<td>#=8</td>
</tr>
<tr>
<td>11:</td>
<td>(65,126, 84,103, 69,121, 81, 97)</td>
<td>#=8</td>
</tr>
<tr>
<td>12:</td>
<td>(66,124, 87,101, 70,123, 82, 99)</td>
<td>#=8</td>
</tr>
<tr>
<td>13:</td>
<td>(67,125, 86,100, 71,122, 83, 98)</td>
<td>#=8</td>
</tr>
<tr>
<td>14:</td>
<td>(72,112, 95,106, 76,119, 90,108)</td>
<td>#=8</td>
</tr>
<tr>
<td>15:</td>
<td>(73,113, 94,107, 77,118, 91,109)</td>
<td>#=8</td>
</tr>
<tr>
<td>16:</td>
<td>(74,115, 93,105, 78,116, 88,111)</td>
<td>#=8</td>
</tr>
<tr>
<td>17:</td>
<td>(75,114, 92,104, 79,117, 89,110)</td>
<td>#=8</td>
</tr>
<tr>
<td>126 elements in 18 nontrivial cycles.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle lengths: 2 ... 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 fixed points: [0, 1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 2.8-B: Cycles of the Gray code permutation of length 128.](image)

It is instructive to study the complementary masks that occur for cycles (see section 2.10 on page 109) of different lengths. The cycles of the Gray code permutation for length 128 are shown in figure 2.8-B. No structure is immediately visible. However, one can generate the cycle maxima as follows: for each range $2^k \ldots 2^{k+1} - 1$ generate a bit-mask $z$ that is obtained from the $k+1$ leftmost bits of the infinite word that has ones at positions 0, 1, 2, 4, 8, ... $, 2^i, \ldots$:

```
[11101000100000010000000001000000000001000 ...]
```

An example: for $k = 6$ we have $z = [1110100]$. Then take $v$ to be $k+1$ leftmost bits of the complement, $v = [0001011]$ in our example. Now the set of words $c = z + s$ where $s$ is a subset of $v$ contains exactly one element of each cycle in the range $2^k \ldots 2^{k+1} = 64 \ldots 127$:

```
<table>
<thead>
<tr>
<th>c</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>117</td>
<td>1</td>
</tr>
<tr>
<td>118</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>122</td>
<td>1</td>
</tr>
<tr>
<td>123</td>
<td>1</td>
</tr>
<tr>
<td>125</td>
<td>1</td>
</tr>
<tr>
<td>126</td>
<td>1</td>
</tr>
</tbody>
</table>

maxima = z XOR subsets(v) where
```

\[ z = .111.1 \]

\[ v = \ldots.11 \]
The list can be generated with the program [FXT: perm/permgray-leaders-demo.cc] which uses the routine [FXT: class gray_cycle_leaders in comb/gray-cycle-leaders.h]:

```cpp
class gray_cycle_leaders
// Generate cycle leaders for Gray code permutation
// where highest bit is at position ldn.
{
public:
  bit_subset b_; // in-place routines
  ulong za_; // mask for cycle maxima
  ulong zi_; // mask for cycle minima
  ulong len_; // cycle length
  ulong num_; // number of cycles
public:
  gray_cycle_leaders(ulong ldn) // 0<=ldn<BITS_PER_LONG
  : b_(0)
  {
    init(ldn);
  }
  ~gray_cycle_leaders() {};
  void init(ulong ldn)
  {
    za_ = 1;
    ulong cz = 0; // ~z
    len_ = 1;
    num_ = 1;
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
      za_ <<= 1;
      cz <<= 1;
      if ( is_pow_of_2(ldm) )
      {
        ++za_;
        len_ <<= 1;
      }
      else
      {
        ++cz;
        num_ <<= 1;
      }
    }
    zi_ = 1UL << ldn;
    b_.first(cz);
  }
  ulong current_max() const { return b_.current() | za_; }
  ulong current_min() const { return b_.current() | zi_; }
  bool next()
  {
    return ( 0!=b_.next() );
  }
  ulong num_cycles() const { return num_; }
  ulong cycle_length() const { return len_; }
};
```

The implementation uses the bit-subset class described in section 1.26 on page 71.

### 2.8.2 In-place routines

The in-place versions of the permutation routines are obtained by inlining the generation of the cycle leaders. The forward version is [FXT: perm/graypermute.h]:

```cpp
template <typename Type>
void gray_permute(Type *f, ulong n)
{
  ulong z = 1; // mask for cycle maxima
```
Chapter 2: Permutations

 ulong v = 0; // ~z
 ulong cl = 1; // cycle length
 for (ulong ldm=1, m=2; m<n; ++ldm, m<<=1)
 {
  z <<= 1;
  v <<= 1;
  if ( is_pow_of_2(ldm) )
  {
    ++z;
    cl <<= 1;
  }
  else ++v;
  bit_subset b(v);
  do
  {
    // --- do cycle: ---
    ulong i = z | b.next(); // start of cycle
    Type t = f[i]; // save start value
    ulong g = gray_code(i); // next in cycle
    for (ulong k=cl-1; k!=0; --k)
    {
      Type tt = f[g];
      f[g] = t;
      t = tt;
      g = gray_code(g);
    }
    f[i] = t;
    // --- end (do cycle) ---
  }
  while ( b.current() );
}

The function \texttt{is\_pow\_of\_2()} is described in section 1.7 on page 18. The inverse routine differs only in the block that processes the cycles:

 template<typename Type>
 void inverse_gray_permute(Type *f, ulong n)
 {
  // --- do cycle: ---
  ulong i = z | b.next(); // start of cycle
  Type t = f[i]; // save start value
  ulong g = gray_code(i); // next in cycle
  for (ulong k=cl-1; k!=0; --k)
  {
    f[i] = f[g];
    i = g;
    g = gray_code(i);
  }
  f[i] = t;
  // --- end (do cycle) ---

 2.8.3 Performance of the routines

 We use the convention that the speed of the trivial (and completely cache-friendly, therefore running at memory bandwidth) \texttt{reverse()} is 1.0, our hereby declared time unit for comparison [FXT: perm/reverse.h]. A little benchmark gives for 16 MB arrays:

 arg 1: 21 == ldn [Using 2**ldn elements] default=21
 arg 2: 10 == rep [Number of repetitions] default=10
 Memsize = 16384 kiloByte == 2097152 doubles

 \begin{verbatim}
 reverse(f,n); dt= 0.0103524 MB/s= 1546 rel= 1
 revbin_permute(f,n); dt= 0.0674235 MB/s= 237 rel= 6.51282
 revbin_permute0(f,n); dt= 0.061507 MB/s= 260 rel= 5.94131
 gray_permute(f,n); dt= 0.0155019 MB/s= 1032 rel= 1.49742
 inverse_gray_permute(f,n); dt= 0.0150641 MB/s= 1062 rel= 1.45512
 \end{verbatim}
2.9: The reversed Gray code permutation

The revbin permutation takes about 6 units, due to its memory access pattern that is very problematic with respect to cache usage. The Gray code permutation needs only 1.50 units. The difference gets bigger for machines with relatively (to the CPU) slow memory. The Gray code permutation can be used to speed up fast transforms of large lengths a power of two, notably the Walsh transform, see chapter 22 on page 457.

The bandwidth of the reverse() is about 1500 MB/sec which should be compared to the output of a memory testing program, revealing that it actually runs at about the bandwidth of copying via a simple loop using pointers to doubles:

```
avg: 16777216 [ 0]"memcpy" 2522.084 MB/s
avg: 16777216 [ 1]"char *" 471.873 MB/s
avg: 16777216 [ 2]"short *" 711.853 MB/s
avg: 16777216 [ 3]"int *" 956.682 MB/s
avg: 16777216 [ 4]"long *" 1514.360 MB/s
avg: 16777216 [ 5]"long * (4x unrolled)" 1330.786 MB/s
avg: 16777216 [ 6]"int64 *" 1329.902 MB/s
avg: 16777216 [ 7]"double *" 1329.507 MB/s  // <--=
avg: 16777216 [ 8]"double * (4x unrolled)" 1325.437 MB/s
```

The relative speeds are quite different for small arrays. Using a size of 16 kB (2048 doubles) we obtain

```
arg 1: 11 == ldn  [Using 2**ldn elements] default=21
arg 2: 100000 == rep [Number of repetitions] default=512
Memsize = 16 kiloByte == 2048 doubles

reverse(f,n); dt=1.88726e-06 MB/s= 8279 rel= 1
revbin_permute0(f,n); dt=3.22166e-06 MB/s= 4850 rel= 1.70706
revbin_permute(f,n); dt=2.69212e-06 MB/s= 5804 rel= 1.42647
gray_permute(f,n); dt=4.75155e-06 MB/s= 3288 rel= 2.51769
inverse_gray_permute(f,n); dt=3.69237e-06 MB/s= 4232 rel= 1.95647
```

Due to the small size, the cache problems are gone. The memory benchmark gives

```
arg 1: 11 == ldn  [Using 2**ldn elements] default=21
arg 2: 100000 == rep [Number of repetitions] default=512
Memsize = 16 kiloByte == 2048 doubles

avg: 16384 [ 0]"memcpy" 3290.353 MB/s
avg: 16384 [ 1]"char *" 572.922 MB/s
avg: 16384 [ 2]"short *" 973.552 MB/s
avg: 16384 [ 3]"int *" 1495.920 MB/s
avg: 16384 [ 4]"long *" 3560.506 MB/s
avg: 16384 [ 5]"long * (4x unrolled)" 3220.792 MB/s
avg: 16384 [ 6]"int64 *" 2498.137 MB/s
avg: 16384 [ 7]"double *" 2498.285 MB/s  // <--=
avg: 16384 [ 8]"double * (4x unrolled)" 2319.784 MB/s
```

2.9 The reversed Gray code permutation

![Permutation matrices of the reversed Gray code permutation (left) and its inverse (right).](image-url)
Chapter 2: Permutations

The reversed Gray code permutation of a length-$n$ array is obtained by permuting the elements in the way that the Gray code permutation would permute the upper half of an array of length $2n$. The array size $n$ must be a power of two. An implementation is [FXT: perm/grayrevpermute.h]:

```cpp
template <typename Type>
inline void gray_rev_permute(const Type *f, Type * restrict g, ulong n)
  // gray_rev_permute() =^=
  // { reverse(); gray_permute(); } 
  { 
    for (ulong k=0, m=n-1; k<n; ++k, --m) g[gray_code(m)] = f[k];
  }
```

All cycles have the same length, the cycles with $n = 64$ elements are

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0, 63, 21, 38, 4, 56, 16, 32)</td>
</tr>
<tr>
<td>1</td>
<td>(1, 62, 20, 39, 5, 57, 17, 33)</td>
</tr>
<tr>
<td>2</td>
<td>(2, 60, 23, 37, 6, 59, 18, 35)</td>
</tr>
<tr>
<td>3</td>
<td>(3, 61, 22, 36, 7, 58, 19, 34)</td>
</tr>
<tr>
<td>4</td>
<td>(8, 48, 31, 42, 12, 55, 26, 44)</td>
</tr>
<tr>
<td>5</td>
<td>(9, 49, 30, 43, 13, 54, 27, 45)</td>
</tr>
<tr>
<td>6</td>
<td>(10, 51, 29, 41, 14, 52, 24, 47)</td>
</tr>
<tr>
<td>7</td>
<td>(11, 50, 28, 40, 15, 53, 23, 46)</td>
</tr>
</tbody>
</table>

64 elements in 8 nontrivial cycles. Cycle length is == 8
No fixed points.

If 64 is added to the indices then the cycles in the upper half of the array as in `gray_permute(f, 128)` are reproduced, by construction.

The in-place version of the routine is

```cpp
template <typename Type>
void gray_rev_permute(Type *f, ulong n)
  // n must be a power of two, n<=BITS_PER_LONG-2 
  { 
    ulong z = 1; // mask for cycle maxima 
    ulong v = 0; // ~z 
    ulong cl = 1; // cycle length 
    ulong ldm, m; 
    for (ldm=1, m=2; m<=n; ++ldm, m<<=1)
      { 
        z <<= 1; v <<= 1; 
        if ( is_pow_of_2(ldm) ) { ++z; cl<<=1; } 
        else ++v; 
      } 
    f -= n; 
    ulong tv = v, tu = 0; // cf. bitsubset.h 
    do 
      { 
        tu = (tu-tv) & tv; 
        ulong i = z | tu; // start of cycle 
        // --- do cycle: --- 
        ulong g = gray_code(i); 
        Type t = f[i]; 
        for (ulong k=cl-1; k!=0; --k)
          { 
            Type tt = f[g]; 
            f[g] = t; 
            t = tt; 
            g = gray_code(g); 
          } 
        f[g] = t; 
        // --- end (do cycle) --- 
      } while ( tu );
  }
```

The inverse differs only in the way the cycles are processed:

```cpp
template <typename Type>
void inverse_gray_rev_permute(Type *f, ulong n)
  { --snip-- }
```
Let $r$ be the reversion and $h$ the swap of the upper and the lower half of an array, we have

\[
\overline{G} = Gr = hG \\
\overline{G}^{-1} = rG^{-1} \\
\overline{G}^{-1}G = G^{-1}\overline{G} = r = X_{n-1} \\
G\overline{G}^{-1} = \overline{G}G^{-1} = h = X_{n/2}
\]

The symbol $X$ denotes the XOR permutation from section 2.7 on page 102.

### 2.10 General permutations and their operations

We study permutations in general with the operations such as composition and inversion.

#### 2.10.1 Basic definitions and operations

A straightforward way to represent a permutation is to consider the array of indices that for the original (unpermuted) data would be the length-$n$ canonical sequence 0, 1, 2, ..., $n-1$. The mentioned trivial sequence represents the 'do-nothing' permutation or identity. The concept is best described by the routine that applies a given permutation $x$ on an array of data $f$: after the routine has finished the array $g$ will contain the elements of $f$ reordered according to $x$.

```cpp
template <typename Type>
void apply_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the permutation x[] on f[]
// i.e. set g[k] <-- f[x[k]] \forall k
{
    for (ulong k=0; k<n; ++k) g[k] = f[x[k]];
}
```

For example, the permutation represented by

```
x = [ 7 6 3 2 5 1 0 4 ]
```

and the input data

```
f = [ A B a d C a f e ]
```

will produce

```
g = [ e f d a a B A C ]
```

Routines that test various properties of permutations are given in [FXT: perm/permq.h]. To check whether a given permutation is the identity is trivial:

```cpp
bool is_identity(const ulong *f, ulong n)
// Return whether f[] is the identical permutation,
// i.e. whether f[k]==k for all k= 0...n-1
{
    for (ulong k=0; k<n; ++k) if ( f[k] != k ) return false;
    return true;
}
```

A fixed point of a permutation is an index where the element is not moved:

```
// --- do cycle: ---
Type t = f[i];  // save start value
ulong g = gray_code(i); // next in cycle
for (ulong k=cl-1; k!=0; --k)
{
    f[i] = f[g];
    i = g;
    g = gray_code(i);
}
// --- end (do cycle) ---
```
ulong count_fixed_points(const ulong *f, ulong n)
// Return number of fixed points in f[]
{
ulong ct = 0;
for (ulong k=0; k<n; ++k) if ( f[k] == k ) ++ct;
return ct;
}

A derangement is a permutation that has no fixed points. To check whether a permutation is a derangement of identity use:

bool is_derangement(const ulong *f, ulong n)
// Return whether f[] is a derangement of identity,
// i.e. whether f[k]!=k for all k
{
for (ulong k=0; k<n; ++k) if ( f[k] == k ) return false;
return true;
}

Whether two arrays are mutual derangements can be determined by:

bool is_derangement(const ulong *f, const ulong *g, ulong n)
// Return whether f[] is a derangement of g[],
// i.e. whether f[k]!=g[k] for all k
{
for (ulong k=0; k<n; ++k) if ( f[k] == g[k] ) return false;
return true;
}

To check whether a given array really describes a valid permutation one has to verify that each index in the valid range appears exactly once. The bitarray class described in section 4.6 on page 158 allows us to do the job without modification of the input:

bool is_valid_permutation(const ulong *f, ulong n, bitarray *bp/*=0*/)
// Return whether all values 0...n-1 appear exactly once,
// i.e. whether f represents a permutation of [0,1,...,n-1].
{
// check whether any element is out of range:
for (ulong k=0; k<n; ++k) if ( f[k]>=n ) return false;

// check whether values are unique:
bitarray *tp = bp;
if ( 0==bp ) tp = new bitarray(n); // tags
tp->clear_all();
ulong k;
for (k=0; k<n; ++k)
{
if ( tp->test_set(f[k]) ) break;
}
if ( 0==bp ) delete tp;
return (k==n);
}

We note two rather trivial operations for permutations, computing the complement [FXT: perm/permcomplement.h]

inline void make_complement(const ulong *f, ulong *g, ulong n)
// Set (as permutation) g to the complement of f.
// Can have f==g.
{
for (ulong k=0; k<n; ++k) g[k] = n - 1 - f[k];
}

and the reversal [FXT: perm/reverse.h]

template <typename Type>
inline void reverse(Type *f, ulong n)
// Reverse order of array f.
{
for (ulong k=0, i=n-1; k<i; ++k, --i) swap2(f[k], f[i]);
}
2.10.2 Representation as disjoint cycles

If one wants to do the operation in-place a little bit of thought is required. The idea underlying all subsequent routines working in-place is that every permutation entirely consists of disjoint cycles. A cycle of a permutation is a subset of the indices that is rotated (by one) by the permutation. The term disjoint means that the cycles do not ‘cross’ each other. While this observation is pretty trivial it allows us to do many operations by following the cycles of the permutation, one by one, and doing the necessary operation on each of them. As an example consider the following permutation of an array originally consisting of the (canonical) sequence 0, 1, ..., 15. Extra spaces are inserted for readability:

\[ 0, 1, 3, 2, 7, 6, 4, 5, 15, 14, 12, 13, 8, 9, 11, 10 \]

There are two fixed points (0 and 1) and these cycles:

\[
\begin{align*}
(2 & \leftarrow 3) \\
(4 & \leftarrow 7 \leftarrow 5 \leftarrow 6) \\
(8 & \leftarrow 15 \leftarrow 10 \leftarrow 12) \\
(9 & \leftarrow 14 \leftarrow 11 \leftarrow 13)
\end{align*}
\]

The cycles do ‘wrap around’, for example, the initial 4 of the second cycle goes to position 6, the last element of the second cycle.

Note that the inverse permutation could formally be described by reversing every arrow in each cycle:

\[
\begin{align*}
(2 & \rightarrow 3) \\
(4 & \rightarrow 7 \rightarrow 5 \rightarrow 6) \\
(8 & \rightarrow 15 \rightarrow 10 \rightarrow 12) \\
(9 & \rightarrow 14 \rightarrow 11 \rightarrow 13)
\end{align*}
\]

Equivalently, one can reverse the order of the elements in each cycle:

\[
\begin{align*}
(3 & \leftarrow 2) \\
(6 & \leftarrow 5 \leftarrow 7 \leftarrow 4) \\
(12 & \leftarrow 10 \leftarrow 15 \leftarrow 8) \\
(13 & \leftarrow 11 \leftarrow 14 \leftarrow 9)
\end{align*}
\]

If we begin each cycle with its smallest element the inverse permutation is written as

\[
\begin{align*}
(2 & \leftarrow 3) \\
(4 & \leftarrow 6 \leftarrow 5 \leftarrow 7) \\
(8 & \leftarrow 12 \leftarrow 10 \leftarrow 15) \\
(9 & \leftarrow 13 \leftarrow 11 \leftarrow 14)
\end{align*}
\]

This form is obtained by reversing all elements except the first in each cycle of the (forward) permutation. The last three sets of cycles all describe the same permutation:

\[ 0, 1, 3, 2, 6, 7, 5, 4, 12, 13, 15, 14, 10, 11, 9, 8 \]

The cycles above were printed with [FXT: print_cycles() in perm/printcycles.cc]
Chapter 2: Permutations

The bit-array (see section 4.6 on page 158 for the implementation) is used to keep track of the elements already processed.

A utility class to compute the decomposition of a permutation into cycles is \texttt{class cycles in perm/cycles.h}. A program that shows its usage is \texttt{perm/cycles-demo.cc}, it prints the cycles

\begin{verbatim}
Computing cycles:
 0: ( 2, 3) #=2
 1: ( 4, 7, 5, 6) #=4
 2: ( 8, 15, 10, 12) #=4
 3: ( 9, 14, 11, 13) #=4
14 elements in 4 nontrivial cycles.
cycle lengths: 2 ... 4
number of fixed points = 2
\end{verbatim}

and code for a permutation of given size

\begin{verbatim}
template <typename Type>
inline void foo_perm_16(Type *f)
// unrolled version for length 16
{
  swap2(f[2], f[3]);
}
\end{verbatim}

2.10.3 Cyclic permutations

A permutation consisting of exactly one cycle is called \textit{cyclic}. Whether a given permutation has this property can be tested with \texttt{is_cyclic()} in \texttt{perm/permq.cc}:

\begin{verbatim}
bool is_cyclic(const ulong *f, ulong n)
// Return whether permutation is exactly one cycle.
{
  if ( n<=1 ) return true;
  ulong k = 0, e = 0;
  do { e=f[s]; ++k; } while ( e!=0 );
  return (k==n);
}
\end{verbatim}

The method used is to follow the cycle starting at position zero and counting how long it is. The permutation is cyclic exactly if the length found equals the array length. There are \((n-1)!\) cyclic permutations of \(n\) elements.

2.10.4 Sign and parity of a permutation

Every permutation can be written as a composition of transpositions (cycles of length two). This composition is not unique, but its number modulo two is unique. The \textit{sign} of a permutation is defined to be \(+1\) the number is even and \(-1\) if the number is odd. The minimal number of transpositions whose composition give a cycle of length \(l\) is \(l-1\). So the minimal number of transpositions for a permutation consisting of \(k\) cycles where the length of the \(j\)-th cycle is \(l_j\) equals \(\sum_{j=1}^{k} l_j - 1 = (\sum_{j=1}^{k} l_j) - k\). The sign
corresponds to the homomorphic mapping into the group of the elements \(+1\) and \(-1\) with multiplication as group operation. If we count the transpositions modulo two (corresponding to the mapping into the additive group modulo two) we obtain what may be called the \textit{parity} of a permutation.

### 2.10.5 Compositions of permutations

One can apply several permutations to an array, one by one. The resulting permutation is called the \textit{composition} of the applied permutations. The routines are given in \cite{fxt:perm/permq.cc}. As an example, the check whether some permutation \(g\) is equal to \(f\) applied twice, or \(f\) squared, use:

```c
bool is_square(const ulong *f, const ulong *g, ulong n)
{
    for (ulong k=0; k<n; ++k) if ( g[k] != f[f[k]] ) return false;
    return true;
}
```

Note that in general \(f \cdot g \neq g \cdot f\) for \(f \neq g\), the operation of composition is not commutative.

A permutation \(f\) is said to be the \textit{inverse} of another permutation \(g\) if it undoes its effect, that is \(f \cdot g = \text{id}\):

```c
bool is_inverse(const ulong *f, const ulong *g, ulong n)
{
    for (ulong k=0; k<n; ++k) if ( f[g[k]] != k ) return false;
    return true;
}
```

One has \(g \cdot f = f \cdot g = \text{id}\), in a group the left-inverse is equal to the right-inverse and we can simply call \(g\) ‘the inverse’ of \(f\).

A permutation that is its own inverse (like the revbin permutation) is called an \textit{involution}. Checking that is easy:

```c
bool is_involution(const ulong *f, ulong n)
{
    for (ulong k=0; k<n; ++k) if ( f[f[k]] != k ) return false;
    return true;
}
```

### 2.10.6 Computing the inverse of a permutation

The following routine computes the inverse of a given permutation \cite{fxt:perm/perminvert.cc}:

```c
void make_inverse(const ulong *f, ulong * restrict g, ulong n)
{
    for (ulong k=0; k<n; ++k) g[f[k]] = k;
}
```

For the in-place computation of the inverse we have to reverse each cycle \cite{fxt:perm/perminvert.cc}:

```c
void make_inverse(ulong *f, ulong n, bitarray *bp/*=0*/)
{
    bitarray *tp = bp;
    if ( 0==bp ) tp = new bitarray(n); // tags
    tp->clear_all();
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) ) continue; // already processed
        tp->set(k);
        // invert a cycle:
        for (ulong k=0; k<n; ++k)
        {
            if ( f[f[k]] != k ) continue;
            tp->set(f[k]);
            tp->set(k);
        }
    }
}
```
ulong i = k;
ulong g = f[i]; // next index
while ( 0==(tp->test_set(g)) )
{
    ulong t = f[g];
    f[g] = i;
    i = g;
    g = t;
}
f[g] = i;
if ( 0==bp ) delete tp;

The extra array of tag bits can be avoided by using the highest bit of each word as tag bit. The scheme would fail if any word of the permutation array had the highest bit set. However, on byte-addressable machines such an array will not fit into memory at all (for word sizes of 16 or more bits). To keep the code similar to the version using the bit-array we define

static const ulong s1 = 1UL << (BITS_PER_LONG - 1); // highest bit is tag bit
static const ulong s0 = ~s1; // all bits but tag bit
static inline void SET(ulong *f, ulong k) { f[k&s0] |= s1; }
static inline void CLEAR(ulong *f, ulong k) { f[k&s0] &= s0; }
static inline bool TEST(ulong *f, ulong k) { return (0!=(f[k&s0]&s1)); }

Note that we have to mask out the tag-bit when using the value ‘k’ as index. The routine can then be implemented as

void
make_inverse(ulong *f, ulong n)
// Set (as permutation) f to its own inverse.
// In-place version using highest bits of array as tag-bits.
{
    for (ulong k=0; k<n; ++k)
    {
        if ( TEST(f, k) ) { CLEAR(f, k); continue; } // already processed
        SET(f, k);
        // invert a cycle:
        ulong i = k;
        ulong g = f[i]; // next index
        while ( 0==TEST(f, g) )
        {
            ulong t = f[g];
            f[g] = i;
            SET(f, g);
            i = g;
            g = t;
        }
        f[g] = i;
        CLEAR(f, k); // leave no tag bits set
    }
}

The extra CLEAR() statement at the end removes the tag-bit of the cycle minima. Its effect is that no tag-bits are set after routine has finished. The routine has about the same performance as the bit-array version.

2.10.7 Computing the square of a permutation

For the routine [FXT: perm/permcompose.cc]

void make_square(const ulong *f, ulong * restrict g, ulong n)
// Set (as permutation) g = f * f
{
    for (ulong k=0; k<n; ++k) g[k] = f[f[k]];
}
we obtain the following in-place version:

```c
void make_square(ulong *f, ulong n, bitarray *bp/*=0*/) {
    // Set (as permutation) f = f * f
    // In-place version.
    bitarray *tp = bp;
    if ( 0==bp ) tp = new bitarray(n); // tags
    tp->clear_all();
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) ) continue; // already processed
        tp->set(k);
        // square a cycle:
        ulong i = k;
        ulong t = f[i]; // save
        ulong g = f[i]; // next index
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = f[g];
        }
        f[i] = t;
    }
    if ( 0==bp ) delete tp;
}
```

### 2.10.8 Composing two permutations

The composition of two permutations can be computed as

```c
void compose(const ulong *f, const ulong *g, ulong * restrict h, ulong n) {
    for (ulong k=0; k<n; ++k) h[k] = f[g[k]];
}
```

The following version will be used in the powering routine for permutations:

```c
void compose(const ulong *f, ulong * restrict g, ulong n) {
    for (ulong k=0; k<n; ++k) g[k] = f[g[k]]; // yes, this works
}
```

### 2.10.9 Powers of a permutation

The $e$-th power of a permutation $f$ is computed (and returned in $g$) by a version of the binary exponentiation algorithm described in section 27.6 on page 565 [FXT: perm/permcompose.cc]:

```c
void power(const ulong *f, ulong * restrict g, ulong n, long e, ulong * restrict t/*=0*/) {
    // Set (as permutation) g = f ** e
    if ( e==0 )
    {
        for (ulong k=0; k<n; ++k) g[k] = k;
        return;
    }
    if ( e==1 )
    {
        copy(f, g, n);
        return;
    }
```

[fxtbook draft of 2008-August-17]
Chapter 2: Permutations

2.10.10 Applying permutations to data, in-place

The in-place analogue for the routine \[\text{FXT: perm/permapply.h}\]

```cpp
template<typename Type>
void apply_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0) {
    // Apply the permutation x[] to the array f[]
    // i.e. set f[k] ← f[x[k]] \forall k
    for (ulong k=0; k<n; ++k)
        if (0==bp) { tp = new bitarray(n); // tags
            tp->clear_all();
        }
        if (tp->test_clear(k)) continue; // already processed
```
To apply the inverse of a permutation without actually inverting the permutation itself use

template <typename Type>
void apply_inverse_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the inverse permutation of x[] to the array f[],
// i.e. set g[x[k]] <-- f[k] \forall k
{
  for (ulong k=0; k<n; ++k) g[x[k]] = f[k];
}

The in-place version is

template <typename Type>
void apply_inverse_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
// Apply the inverse permutation of x[] to the array f[]
// i.e. set f[x[k]] <-- f[k] \forall k
// In-place version.
{
  bitarray *tp = bp;
  if ( 0==bp ) tp = new bitarray(n); // tags
  tp->clear_all();
  for (ulong k=0; k<n; ++k)
  {
    if ( tp->test_clear(k) ) continue; // already processed
    tp->set(k);
    // --- do cycle: ---
    ulong i = k; // start of cycle
    Type t = f[i];
    ulong g = x[i];
    while ( 0==(tp->test_set(g)) ) // cf. gray_permute()
    {
      Type tt = f[g];
      f[g] = t;
      t = tt;
      g = x[g];
    }
    f[g] = t;
    // --- end (do cycle) ---
    }
  if ( 0==bp ) delete tp;
}

When a permutation of the set \( S := \{0, 1, \ldots, n-1\} \) is given as a function \( X \) (where \( X(S) = S \) the permutation can be applied to an array \( f \) via [FXT: apply_permutation()] in [perm/permapplyfunc.h].
For example, the statement `apply_permutation(inverse_gray_code, f, g, n)` is equivalent to `gray_permute(f, g, n)`. The inverse routine is

```cpp
template <typename Type>
void apply_inverse_permutation(ulong (*x)(ulong), const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0; k<n; ++k) g[x(k)] = f[k];
}
```

The in-place versions of these routines are identical to the routines that apply permutations given as arrays. Only a tiny change must be made in the processing of the cycles. For example, the fragment

```cpp
void apply_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
{
    for (ulong k=0; k<n; ++k)
    
    ulong i = k; // start of cycle
    Type t = f[i];
    ulong g = x[i]; // <--=
    while ( 0==(tp->test_set(g)) )
    {
        f[i] = f[g];
        i = g;
        g = x[i]; // <--=
    }
    f[i] = t;
}
```

must be changed to (replace `x[i]` by `x(i)`)

```cpp
void apply_permutation(ulong (*x)(ulong), Type *f, ulong n, bitarray *bp=0)
{
    for (ulong k=0; k<n; ++k)
    
    ulong i = k; // start of cycle
    Type t = f[i];
    ulong g = x(i); // <--=
    while ( 0==(tp->test_set(g)) )
    {
        f[i] = f[g];
        i = g;
        g = x(i); // <--=
    }
    f[i] = t;
}
```

### 2.10.11 Random permutations

Routines for random permutations are given in [FXT:perm/permrand.h]. The following routine randomly permutes an array with arbitrary elements:

```cpp
template <typename Type>
void random_permute(Type *f, ulong n)
{
    for (ulong k=1; k<n; ++k)
    
    ulong r = (ulong)rand();
    r ^= r>>16; // avoid using low bits of rand alone
    ulong i = r % (k+1);
    swap2(f[k], f[i]);
}
```

The method is given in [108]. A random permutation can be obtained by applying the function to the canonical sequence:

```cpp
void random_permutation(ulong *f, ulong n)
{
    for (ulong k=0; k<n; ++k) f[k] = k;
    random_permute(f, n);
}
```
We note that a slight modification of the underlying idea can be used for a routine for random selection from a list with only one linear read. Let \( L \) be a list of \( n \) items \( L_1, \ldots, L_n \).

1. Set \( t = L_1 \), set \( k = 1 \).
2. Set \( k = k + 1 \). If \( k > n \) return \( t \).
3. With probability \( 1/k \) set \( t = L_k \).
4. Go to step 2.

Note that one does not need to know \( n \), the number of elements in the list, in advance: replace part 2 of step 2 by “If there are no more elements then return \( t \”).

A routine to apply a random cyclic permutation (as defined in section 2.10.3 on page 112) to an array is

```cpp
template <typename Type>
void random_permute_cyclic(Type *f, ulong n)
// Permute the elements of f by a random cyclic permutation.
{
    for (ulong k=n-1; k>0; --k)
    {
        ulong r = (ulong)rand();
        r ^= r>>16; // avoid using low bits of rand alone
        ulong i = r % k;
        swap2(f[k], f[i]);
    }
}
```

The method is described in [138], see also [283]. A random cyclic permutation can be obtained by applying a random cyclic permutation to the canonical sequence:

```cpp
inline void
random_cyclic_permutation(ulong *f, ulong n)
// Create a random permutation that is cyclic.
{
    for (ulong k=0; k<n; ++k) f[k] = k;
    random_permute_cyclic(f, n);
}
```

The cycle representation of a cyclic permutation can be obtained by applying a random permutation to all elements (of the identical permutation) except for the first element.
Chapter 3

Sorting and searching

In this chapter some practical flavors of sorting algorithms are given. These include plain sorting, sorting index arrays, pointer sorting; all optionally with a supplied comparison function. Massive literature exist about the topic so we will not go into the algorithmic details. Very readable texts are [95] and [241]. In-depth information can be found in [173]. The sorting algorithms used in this chapter are selection sort, quicksort, counting sort and radix sort.

Some algorithms on sorted arrays like binary searching and determination of unique elements are included. Finally, some functions for scanning unsorted arrays are given.

3.1 Sorting

3.1.1 Selection sort

Figure 3.1-A: Sorting the string ‘nowsortme’ with the selection sort algorithm.

There are a several algorithms for sorting that scale with $\sim n^2$ where $n$ is the size of the array to be sorted. Here we use selection sort whose idea is to find the minimum of the array, swap it with the first element and repeat for all elements but the first. A demonstration of the algorithm is shown in figure 3.1-A, this is the output of [FXT: sort/selection-sort-demo.cc]. The implementation is straightforward [FXT: sort/sort.h]:

```cpp
1  template <typename Type>
2  void selection_sort(Type *f, ulong n)
3  // Sort f[] (ascending order).
4  // Algorithm is proportional to $O(n*n)$, use for short arrays only.
5  {
6      for (ulong i=0; i<n; ++i)
7          {
8              Type v = f[i];
9              ulong m = i; // position of minimum
```
Chapter 3: Sorting and searching

A verification routine is always handy:

```cpp
template <typename Type>
bool is_sorted(const Type *f, ulong n)
// Return whether the sequence f[0], f[1], ..., f[n-1]
// is sorted in ascending order.
{
for (ulong k=1; k<n; ++k) if ( f[k-1] > f[k] ) return false;
return true;
}
```

A test for descending order is

```cpp
template <typename Type>
bool is_falling(const Type *f, ulong n)
{
for (ulong k=1; k<n; ++k) if ( f[k-1] < f[k] ) return false;
return true;
}
```

### 3.1.2 Quicksort

The quicksort algorithm is given in [148], it scales \(\sim n \log(n)\) (in the average case). It does not just obsolete the more simple schemes because for small arrays the more simple algorithms are usually faster because of their minimal bookkeeping overhead.

The main ingredient of quicksort is to partition the array. The corresponding routine reorders the array and returns an pivot index \(p\) so that \(\max(f_0, \ldots, f_{p-1}) \leq \min(f_p, \ldots, f_{n-1})\) [FXT: sort/sort.h]:

```cpp
template <typename Type>
ulong partition(Type *f, ulong n)
{
// Avoid worst case with already sorted input:
const Type v = median3(f[0], f[n/2], f[n-1]);
ulong i = 0UL - 1;
ulong j = n;
while ( 1 )
{
    do { ++i; } while ( f[i] < v );
    do { --j; } while ( f[j] > v );
    if ( i < j ) swap2(f[i], f[j]);
else return j;
}
```

The function `median3()` is defined in [FXT: sort/minmaxmed23.h]:

```cpp
template <typename Type>
static inline Type median3(const Type &x, const Type &y, const Type &z)
// Return median of the input values
{
return x < y ? (y < z ? y : (z < x ? z : x)) : (z < x ? z : x));
}
```

The function does 2 or 3 comparisons, depending on the input. One could simply use the element \(f[0]\) as pivot. However, the algorithm will be \(\sim n^2\) (that is, quadratic) when the array is already sorted.
3.2: Binary search

The main reason for sorting may be that a fast search has to be performed repeatedly. The binary search algorithm works by the obvious subdivision of the data [FXT: bsearch() in sort/bsearch.h]:

```cpp
1 template <typename Type>
2 ulong bsearch(const Type *f, ulong n, const Type v)
3 // Return index of first element in f[] that equals v
4 // Return ~0 if there is no such element.
5 // f[] must be sorted in ascending order.
6 // Must have n!=0
7 {
8     ulong nlo=0, nhi=n-1;
9     while ( nlo != nhi )
```
Chapter 3: Sorting and searching

```c
{ ulong t = (nhi+nlo)/2;
  if ( f[t] < v ) nlo = t + 1;
  else nhi = t;
}
if ( f[nhi]==v ) return nhi;
else return ~0UL;
```

The algorithm uses $\sim \log_2(n)$ operations. For very large arrays the algorithm can be improved by selecting the new index $t$ different from midpoint $(nhi+nlo)/2$, dependent of the value sought and the distribution of the values in the array. As a simple example consider an array of floating point numbers that are equally distributed in the interval $[\min(v), \max(v)]$. If the sought value equals $v$ one would want to use the relation

$$\frac{n - \min(n)}{\max(n) - \min(n)} = \frac{v - \min(v)}{\max(v) - \min(v)}$$

where $n$ denotes an index, and $\min(n), \max(n)$ denote the minimal and maximal index of the current interval. Solving for $n$ gives the linear interpolation formula

$$n = \min(n) + \frac{\max(n) - \min(n)}{\max(v) - \min(v)} (v - \min(v))$$

The corresponding interpolation binary search algorithm would select the new subdivision index $t$ according to the given relation. One could even use quadratic interpolation schemes for the selection of $t$. For the majority of practical applications the midpoint version of the binary search will be good enough.

A simple modification of `bsearch` makes it search the first element greater than or equal to $v$: replace the operator `==` in the above code by `>=` and you have it: [FXT: `bsearch_ge()` in `sort/bsearch.h`]. Similar for the `<=` relation: `bsearch_le()`.

Approximate matches are found by [FXT: `bsearch_approx()` in `sort/bsearchapprox.h`]:

```c
template <typename Type>
ulong bsearch_approx(const Type *f, ulong n, const Type v, Type da)
// Return index of first element x in f[] for which |(x-v)| <= da
// f[] must be sorted in ascending order.
// da must be positive.
// Makes sense only with inexact types (float or double).
// Must have n!=0
{
  ulong k = bsearch_ge(f, n, v-da);
  if ( k<n ) k = bsearch_le(f+k, n-k, v+da);
  return k;
}
```

### 3.3 Index sorting

While the ‘plain’ sorting reorder an array $f$ so that, after it has finished, $f_k \leq f_{k+1}$ the following routines sort an array of indices without modifying the actual data. The index-sort routines reorder the indices in an array $x$ such that $x$ applied to $f$ as a permutation (in the sense of section 2.10.10 on page 116) will render $f$ a sorted array [FXT: `sort/sortidx.h`]:

```c
template <typename Type>
void idx_selection_sort(const Type *f, ulong n, ulong *x)
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ..., f[x[n-1]]
// is sorted in ascending order.
// Algorithm is proportional to $O(n^2)$, use for short array only.
{
```
3.3: Index sorting

```c
for (ulong i=0; i<n; ++i)
{
    Type v = f[x[i]];
    ulong m = i; // position-ptr of minimum
    ulong j = n;
    while ( --j > i ) // search (index of) minimum
    {
        if ( f[x[j]]<v )
        {
            m = j;
            v = f[x[m]];
        }
    }
    swap2(x[i], x[m]);
}
```

Apart from the ‘read only’-feature the index-sort routines have the nice property to perfectly work on non-contiguous data. The verification code is

```c
template <typename Type>
bool is_idx_sorted(const Type *f, ulong n, const ulong *x)
// Return whether the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
{
    if ( 0==n ) return 1;
    while ( --n ) // n-1 ... 1
    {
        if ( f[x[n]] < f[x[n-1]] ) break;
    }
    return !n;
}
```

The transformation of the `partition()` routine is straightforward:

```c
template <typename Type>
ulong idx_partition(const Type *f, ulong n, ulong *x)
// rearrange index array, so that for some index p
// max(f[x[0]] ... f[x[p]]) <= min(f[x[p+1]] ... f[x[n-1]])
{
    // Avoid worst case with already sorted input:
    const Type v = median3(*x[0], *x[n/2], *x[n-1], cmp);
    ulong i = 0UL - 1;
    ulong j = n;
    while ( 1 )
    {
        do ++i;
        while ( f[x[i]]<v );
        do --j;
        while ( f[x[j]]>v );
        if ( i<j ) swap2(x[i], x[j]);
        else return j;
    }
}
```

The index-quick sort itself deserves a minute of contemplation comparing it to the plain version:

```c
template <typename Type>
void idx_quick_sort(const Type *f, ulong n, ulong *x)
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
{
    start:
    if ( n<8 ) // parameter: threshold for nonrecursive algorithm
    {
        idx_selection_sort(f, n, x);
    }
    return;
}
```

[fxtbook draft of 2008-August-17]
ulong ln = p + 1;
ulong rn = n - ln;

if ( ln>rn ) // recursion for shorter sub-array
{
    idx_quick_sort(f, rn, x+ln); // f[x[ln]] ... f[x[n-1]] right
    n = ln;
}
else
{
    idx_quick_sort(f, ln, x); // f[x[0]] ... f[x[ln-1]] left
    n = rn;
    x += ln;
}
go to start;

The index-analogues of the binary search algorithms are again straightforward, they are given in [FXT: sort/bsearchidx.h].

### 3.4 Pointer sorting

Pointer sorting is an idea similar to index sorting which is even less restricted than index sort: The data may be unaligned in memory. And overlapping. Or no data at all but port addresses controlling some highly dangerous machinery. Therby pointer sort is the perfect way to highly cryptic and powerful programs that seg-fault when you least expect it.

Just to make the idea clear, the array of indices is replaced by an array of pointers [FXT: sort/sortptr.h].

```cpp
template <typename Type>
void ptr_selection_sort(/*const Type *f,*/ ulong n, const Type **x)
// Sort x[] so that the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order.
// Algorithm is proportional to O(n*n), use for short arrays only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = *x[i];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i ) // search (index of) minimum
        {
            if ( *x[j]<v )
            {
                m = j;
                v = *x[m];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

The first argument (const Type *f) is not necessary with pointer sorting, it is indicated as comment to make the argument structure uniform. The verification routine is

```cpp
template <typename Type>
bool is_ptr_sorted(/*const Type *f,*/ ulong n, Type const*const*x)
// Return whether the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order.
{
    if ( 0==n ) return 1;
    while ( --n ) // n-1 ... 1
    {
        if ( *x[n] < *x[n-1] ) break;
    }
```

3.5 Sorting by a supplied comparison function

The routines in \texttt{[FXT:sort/sortfunc.h]} are similar to the C-quicksort \texttt{qsort} that is part of the standard library. A comparison function \texttt{cmp} has to be supplied by the called so that compound data types can be sorted with respect to some key contained. Citing the manual page for \texttt{qsort}:

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

Note that the numerous calls to \texttt{cmp} do have a negative impact on the performance. With C++ you can provide a comparison ‘function’ for compound data by overloading the operators \texttt{<, <=} and \texttt{>=} and use the plain version. That is, the comparisons are inlined an we are back in performance land. Isn’t C++ nice? As a prototypical example we give the selection sort routine:

```cpp
template <typename Type>
void selection_sort(Type *f, ulong n, int (*cmp)(const Type &, const Type &))
{
    // Sort f[] (ascending order)
    // with respect to comparison function cmp().
    // Algorithm is proportional to O(n*n), use for short array only.
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[i];
        ulong m = i; // position of minimum
        ulong j = n;
        while ( --j > i ) // search (index of) minimum
        {
            if ( cmp(f[j],v) < 0 )
            {
                m = j;
                v = f[m];
            }
        }
        swap2(f[i], f[m]);
    }
}
```

The other routines are rather straightforward translations of the (plain-) sort analogues: replace the comparison operations as follows

<table>
<thead>
<tr>
<th>Condition</th>
<th>\texttt{cmp(a,b)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a &lt; b}</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>\texttt{a &gt; b}</td>
<td>&gt; 0</td>
</tr>
<tr>
<td>\texttt{a == b}</td>
<td>== 0</td>
</tr>
<tr>
<td>\texttt{a &lt;= b}</td>
<td>&lt;= 0</td>
</tr>
<tr>
<td>\texttt{a &gt;= b}</td>
<td>&gt;= 0</td>
</tr>
</tbody>
</table>

For example, the verification routine is

```cpp
template <typename Type>
bool is_sorted(const Type *f, ulong n, int (*cmp)(const Type &, const Type &))
{
    // Return whether the sequence
    // is sorted in ascending order
    // with respect to comparison function cmp().
    if ( 0==n ) return 1;
    while ( --n ) // n-1 ... 1
    {
        if ( cmp(f[n], f[n-1]) < 0 ) break;
    }
    return !n;
}
```
3.5.1 Sorting complex numbers

You want to sort complex numbers? Fine for me, but don’t tell your local mathematician. To see the mathematical problem we ask whether \( i \) is smaller or greater than zero. Assume \( i > 0 \): follows \( i \cdot i > 0 \) (we multiplied with a positive value) which is \( -1 > 0 \) and that is false. So, is \( i < 0 \)? Then \( i \cdot i > 0 \) (multiplication with a negative value, as assumed). So \( -1 > 0 \), oops! The lesson is that there is no way to impose an arrangement on the complex numbers that would justify the usage of the symbols ‘<’ and ‘>’ consistent with the rules to manipulate inequalities.

Nevertheless we can invent a relation that allows us to sort: arranging (sorting) the complex numbers according to their absolute value (modulus) leaves infinitely many numbers in one ‘bucket’, namely all those that have the same distance from zero. However, one could use the modulus as the major ordering parameter, the angle as the minor. Or the real part as the major and the imaginary part as the minor. The latter is realized in

```cpp
static inline int cmp_complex(const Complex &f, const Complex &g)
{
    const double fr = f.real(), gr = g.real();
    if (fr!=gr) return (fr>gr ? +1 : -1);
    const double fi = f.imag(), gi = g.imag();
    if (fi!=gi) return (fi>gi ? +1 : -1);
    return 0;
}
```

This routine, when used as comparison with the function-sort, as in

```cpp
void complex_sort(Complex *f, ulong n)
// major order wrt. real part
// minor order wrt. imag part
{
    quick_sort(f, n, cmp_complex);
}
```

can indeed be the practical tool you had in mind.

3.5.2 Index and pointer sorting

The index sorting routines that use a supplied comparison function are given in [FXT: sort/sortidxfunc.h]:

```cpp
template <typename Type>
void idx_selection_sort(const Type *f, ulong n, ulong *x, int (*cmp)(const Type &, const Type &))
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order
// with respect to comparison function cmp()
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[x[i]];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while (j > i) // search (index of) minimum
        {
            if ( cmp(f[x[j]], v) < 0 )
            {
                m = j;
                v = f[x[m]];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

[fxtbook draft of 2008-August-17]
3.5: Sorting by a supplied comparison function

The verification routine is:

```cpp
template <typename Type>
bool is_idx_sorted(const Type *f, ulong n, const ulong *x,
       int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order
// with respect to comparison function cmp()
{
    if ( 0==n ) return 1;
    while ( --n ) // n-1 ... 1
    {
        if ( cmp(f[x[n]], f[x[n-1]]) < 0 ) break;
    }
    return !n;
}
```

The pointer sorting versions are given in [FXT: sort/sortptrfunc.h]

```cpp
template <typename Type>
void ptr_selection_sort(/*const Type *f,*/ ulong n, const Type **x,
       int (*cmp)(const Type &, const Type &))
// Sort x[] so that the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = *x[i];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i ) // search (index of) minimum
        {
            if ( cmp(*x[j],v)<0 )
            {
                m = j;
                v = *x[m];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

The verification routine is:

```cpp
template <typename Type>
bool is_ptr_sorted(/*const Type *f,*/ ulong n, Type const*const*x,
       int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order
// with respect to comparison function cmp().
{
    if ( 0==n ) return 1;
    while ( --n ) // n-1 ... 1
    {
        if ( cmp(*x[n],*x[n-1])<0 ) break;
    }
    return !n;
}
```

The corresponding versions of the binary search algorithm are given in [FXT: sort/bsearchidxfunc.h] and [FXT: sort/bsearchptrfunc.h].
3.6 Determination of unique elements

We present functions that check whether values in a sorted array are repeated or unique. All routines are taken from [FXT: sort/unique.h]. To test whether all values are unique, use

```cpp
template <typename Type>
ulong test_unique(const Type *f, ulong n)
// For a sorted array test whether all values are unique
// (i.e. whether no value is repeated).
// Return 0 if all values are unique else return index of the second
// element in the first pair found.
{
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] == f[k-1] ) return k; // k != 0
    }
    return 0;
}
```

```cpp
template <typename Type>
ulong is_unique(const Type *f, ulong n)
// Return true if all values are unique, else return false.
{
    return ( 0==test_unique(f, n) );
}
```

Counting the elements that appear just once:

```cpp
template <typename Type>
int unique_count(const Type *f, ulong n)
// For a sorted array return the number of unique values
// the number of (not necessarily distinct) repeated
// values is n - unique_count(f, n);
{
    if ( 1>=n ) return n;
    ulong ct = 1;
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] != f[k-1] ) ++ct;
    }
    return ct;
}
```

Removing repeated elements:

```cpp
template <typename Type>
ulong unique(Type *f, ulong n)
// For a sorted array squeeze all repeated values
// and return the number of unique values.
// Example: [1, 3, 3, 4, 5, 8, 8] --> [1, 3, 4, 5, 8]
// The routine also works for unsorted arrays as long
// as identical elements only appear in contiguous blocks.
// The order is preserved.
{
    ulong u = unique_count(f, n);
    if ( u==n ) return n; // nothing to do
    Type v = f[0];
    for (ulong j=1, k=1; j<u; ++j)
    {
        while ( f[k]==v ) ++k; // search next different element
        v = f[j] = f[k];
    }
    return u;
}
```

The inner `while`-loop does never access an element out of bounds as it is executed only as long as there is at least one remaining change of value inside the array.
3.7 Unique elements with inexact types

Determination of unique elements with inexact types (floats) is a bit tricky as one cannot rely that elements that should be identical are exactly equal. A solution to the problem is to allow for a maximal (absolute) difference within which two contiguous elements will still be considered equal can be provided as additional parameter. We replace equality conditions with a call to [FXT: sort/uniqueapprox.h]

```cpp
template <typename Type>
inline bool approx_equal(Type x1, Type x2, Type da)
// Return whether abs(x2-x1) <= da
// Must have da>=0
{
    Type d = x2 - x1;
    if ( d<=0 ) d = -d;
    if ( d <= da ) return true;
    else return false;
}
```

The verification routine is

```cpp
template <typename Type>
ulong test_unique_approx(const Type *f, ulong n, Type da)
// For a sorted array test whether all values are unique within some tolerance (i.e. whether no value is repeated).
// Return 0 if all values are unique,
// else return index of the second element in the first pair found.
// Makes mostly sense with inexact types (float or double)
{
    if ( da<=0 ) da = -da; // want positive tolerance
    for (ulong k=1; k<n; ++k)
    {
        if ( approx_equal(f[k], f[k-1], da) ) return k; // k != 0
    }
    return 0;
}
```

One subtle point is that the values can slowly ‘drift away’ unnoticed by this implementation: consider a long array where each difference computed has the same sign and is just smaller than da, say it is \(d = 0.6 \cdot da\). The difference of the first and last value then is \(0.6 \cdot (n-1) \cdot d\) which is greater than da for \(n \geq 3\).

The number of unique elements can be counted as follows:

```cpp
template <typename Type>
ulong unique_approx_count(const Type *f, ulong n, Type da)
// For a sorted array return the number of unique values
// the number of (not necessarily distinct) repeated values is n - unique_approx_count(f, n, da);
{
    if ( 1>=n ) return n;
    if ( da<=0 ) da = -da; // Must have positive tolerance
    ulong ct = 1;
    for (ulong k=1; k<n; ++k)
    {
        if ( approx_equal(f[k], f[k-1], da) ) ++ct;
    }
    return ct;
}
```

The following routine removes duplicates:

```cpp
template <typename Type>
ulong unique_approx(Type *f, ulong n, Type da)
// For a sorted array squeeze all repeated (within tolerance da) values and return the number of unique values.
// Example: [1, 3, 3, 4, 5, 8, 8] --> [1, 3, 4, 5, 8]
// The routine also works for unsorted arrays as long as identical elements only appear in contiguous blocks.
// The order is preserved.
{
```
ulong u = unique_approx_count(f, n, da);
if ( u==n ) return n; // nothing to do
if ( da<=0 ) da = -da; // Must have positive tolerance
Type v = f[0];
for (ulong j=1, k=1; j<u; ++j)
{
  // search next different element:
  while ( approx_equal(f[k], v, da) )
  {
    v = f[k]; // avoid problem with slowly drifting values
    ++k;
  }
  v = f[j] = f[k];
}
return u;

A useful preprocessing step (before using test_unique_approx()) is to quantize the elements of an array
[FXT: quantize()] in sort/quantize.h:

```
template <typename Type>
void quantize(Type *f, ulong n, double q)
  // In f[] set each element x to q*floor(1/q*(x+q/2))
  // E.g.: q=1 ==> round to nearest integer
  // q=1/1000 ==> round to nearest multiple of 1/1000
  // For inexact types (float or double).
{
  Type qh = q * 0.5;
  Type q1 = 1.0 / q;
  while ( n-- )
  {
    f[n] = q * floor( q1 * (f[n]+qh) );
  }
}
```

One should use a quantization parameter q that is greater than the value used for da.

A simple demonstration is given in [FXT: sort/unique-demo.cc]:

```
Random values:
0: 0.9727750243 1: 0.2925167845 2: 0.7713576982 3: 0.5267449795 4: 0.7699138366
5: 0.4002286223
Quantization with q=0.01
Quantized & sorted :
0: 0.2900000000 1: 0.4000000000 2: 0.5300000000 3: 0.7700000000 4: 0.9700000000
First REPEATED value at index 4 (and 3)
Unique’d array:
0: 0.2900000000 1: 0.4000000000 2: 0.5300000000 3: 0.7700000000

The routine quantize() turns out to be also useful for the conversion of imprecise data to symbols. For example, the array of floating point values on the left corresponds to the symbolic (numbers used as symbols) table on the right:

```
1.3133  -1.0101  0.79412  -0.71544  9  2  6  3
0.4927750243 0.9727750243 -1.4382  0.79412  5  7  0  9
-1.0101  0.79412  -0.71544  9  2  6  3
0.29064  0.99173  -1.1086  1.2521  0.99173  -1.0101
-1.1086  0.99173  -1.0101  1.2521  0.99173  -1.1086
-0.18003 -1.1086  0.29064  1.3133
```
```
In this example values were considered identical when their absolute difference is less than 10^-3. The symbolic representation can be helpful to recognize structure in imprecise data. The routine is [FXT: sort/symbolify.h]:
```
template <typename Type>
```
3.8: Determination of equivalence classes

Let $S$ be a set and $C := S \times S$ the set of all ordered pairs $(x, y)$ with $x, y \in S$. A binary relation $R$ on $S$ is a subset of $C$. An equivalence relation is a binary relation that has three additional properties:

- reflexive: $x \equiv x \forall x$.
- symmetric: $x \equiv y \iff y \equiv x$.
- transitive: $x \equiv y, y \equiv z \implies x \equiv z$.

Here we wrote $x \equiv y$ for $(x, y) \in R$ where $x, y \in S$.

We want to determine the equivalence classes: an equivalence relation partitions a set into $1 \leq q \leq n$ subsets $E_1, E_2, \ldots, E_q$ so that $x \equiv y$ whenever both $x$ and $y$ are in the same subset but $x \not\equiv y$ if $x$ and $y$ are in different subsets.

For example, the usual equality relation is an equivalence relation, with a set of (different) numbers each number is in its own class. With the equivalence relation that $x \equiv y$ whenever $x - y$ is a multiple of

The example shown was created with the program [FXT: sort/symbolify-demo.cc]. The routines apply_permutation() and apply_inverse_permutation() are given in section 2.10.10 on page 116.

3.8 Determination of equivalence classes

ulong symbolify_by_size(const Type *f, Type * restrict g, ulong n,
Type eps=1e-6, ulong *ix=0)
// From f[] compute an array of 'symbols' g[] (i.e. numbers)
// that represent the different values.
// Values are considered identical if their absolute difference
// is less than eps.
// Symbols are given with respect to sort-order.
// Return number of different values found (after quantize).
// Optionally supply x[] (scratch space for permutations).
{
ulong *x = ix;
if ( 0==ix ) x = new ulong[n];
set_seq(x, n);
idx_quick_sort(f, n, x);
apply_permutation(x, f, g, n);
quantize(g, n, eps);
eps *= 0.5; // some val <1.0
ulong nsym = 1;
ulong z = 0;
Type s = 0.0;
Type el = g[z], lel;
g[z] = s;
for(ulong k=z+1; k<n; ++k)
{
lel = el;
el = g[k];
if ( fabs(el-lel) > eps )
{
++nsym;
s += 1.0;
}
g[k] = s;
}
apply_inverse_permutation(x, g, n);
if ( 0==ix ) delete [] x;
return nsym;
}
some fixed integer \( m \) and the set \( \mathbb{Z} \) of all natural numbers we obtain \( m \) subsets and \( x \equiv y \) if and only if \( x \equiv y \mod m \).

Let \( n \) be the number of elements in \( S \) and \( Q \) be a set so that, on termination of the algorithm, \( Q_j = j \) if \( j \) is the least index so that \( S_j \equiv S_k \) (note that we consider the sets to be in a fixed but arbitrary order here).

We proceed as follows:

1. Put each element in its own equivalence class: \( Q_k := k \) for all \( 0 \leq k < n \)
2. Set \( k := 1 \) (index of the second element).
3. Search downwards for an equivalent element: for \( j = k - 1, \ldots, 0 \) test whether \( S_k \equiv S_j \). If so (at position \( j \)), set \( Q_k = Q_j \) and goto step 4.
4. Set \( k := k + 1 \) and if \( k < n \) goto step 3, else terminate.

We can terminate the search with the first equivalent element found because, if \( j \) is the index of the equivalent, the \( Q_j \) is already minimal.

The lower and upper bounds for the computational cost are \( n \) and \( n^2 \), respectively: the algorithm needs proportional \( n \) operations when all elements are in the same equivalence class and \( n^2 \) operations when each element lies in its own class.

A C++ implementation is \([FXT: equivalence\_classes() in sort/eqivclasses.h]\). The equivalence relation must be supplied as a function \( \text{equiv}_q() \) that returns true when its arguments are equivalent.

```c++
1 template <typename Type>
2 void equivalence_classes(const Type *s, ulong n, bool (*equiv_q)(Type,Type), ulong *q)
3 // Given an equivalence relation '==' (as function equiv_q())
4 // and a set s[] with n elements,
5 // write into q[k] (0<=k<n) the index j of the
6 // first element s[j] so that s[k]==s[j].
7 // For the complexity C: n<=C<=n*n
8 // C=n if each element is in its own class
9 // C=n if all elements are in the same class
10 {
11    for (ulong k=0; k<n; ++k) q[k] = k; // each in own class
12    for (ulong k=1; k<n; ++k)
13    {  
14      ulong j = k;
15      while ( j-- )
16      {
17        if ( equiv_q(s[j], s[k]) )
18          {  
19            q[k] = q[j];
20            break;
21          }
22      }
23    }
24 }
```

### 3.8.1 Examples for equivalence classes

#### 3.8.1.1 Integers modulo \( m \)

Choose an integer \( m \geq 2 \) and let any two integers \( a \) and \( b \) be equivalent if \( a - b \) is an integer multiple of \( m \). We can choose the numbers 0, 1, \ldots, \( m - 1 \) as representatives of the \( m \) classes obtained. Now we can do computations with those classes via the modular arithmetic as described in section 37.1 on page 763. This is easily the most important example of all equivalence classes.

We note that the concept still make sense with a real (that is, possibly non-integral) modulus \( m \). We still put two numbers \( a \) and \( b \) into the same class if \( a - b \) is a integer multiple of \( m \). Finally, the modulus zero leads to the equivalence relation ‘equality’.

[fxtbook draft of 2008-August-17]
3.8.1.2 Binary necklaces

The set $S$ of $n$-bit binary words and the equivalence relation so that two words $x$ and $y$ are equivalent when there is a cyclic shift $h_k(x)$ by $0 \leq k < n$ positions so that $h_k(x) = y$. The relation is supplied as the function [FXT: sort/equivclass-necklaces-demo.cc]:

```c
static ulong b; // number of bits
bool n_equiv_q(ulong x, ulong y) // necklaces
{
    ulong d = bit_cyclic_dist(x, y, b);
    return (0==d);
}
```

The function `bit_cyclic_dist()` is given in section 1.13.4 on page 34. With $n = 4$ we obtain the following list of equivalence classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Words</th>
<th>#</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>....</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1... .1... .1... .1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.1 11. 11... ..11 .11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.1.1 1.1 1... 11.1 .11 .11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1111 1111 1111 .111 .111 .111 .111 .111 .111</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1111 1111 1111 .111 .111 .111 .111 .111 .111</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

These correspond to the binary necklaces of length 4. One usually chooses the cyclic minima (or maxima) among equivalent words as representatives of the classes.

3.8.1.3 Unlabeled binary necklaces

Same set but the equivalence relation is defined to identify two words $x$ and $y$ when there is a cyclic shift $h_k(x)$ by $0 \leq k < b$ positions so that either $h_k(x) = y$ or $h_k(x) = \overline{y}$ where $\overline{y}$ is the complement of $y$:

```c
static ulong mm; // mask to complement
bool nu_equiv_q(ulong x, ulong y) // unlabeled necklaces
{
    ulong d = bit_cyclic_dist(x, y, b);
    if ( 0!=d ) d = bit_cyclic_dist(mm^x, y, b);
    return (0==d);
}
```

With $n = 4$ we obtain:

<table>
<thead>
<tr>
<th>Class</th>
<th>Words</th>
<th>#</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1111</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1111 1111 1111 .111 .111 .111 .111 .111 .111</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1111 1111 1111 .111 .111 .111 .111 .111 .111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1111 1111 1111 .111 .111 .111 .111 .111 .111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1111 1111 1111 .111 .111 .111 .111 .111 .111</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1111 1111 1111 .111 .111 .111 .111 .111 .111</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

These correspond to the unlabeled binary necklaces of length 4.

3.8.1.4 Binary bracelets

The binary bracelets are obtained by identifying two words that are identical up to rotation and possible reversion. The corresponding comparison function is

```c
bool b_equiv_q(ulong x, ulong y) // bracelets
{
    ulong d = bit_cyclic_dist(x, y, b);
    if ( 0!=d ) d = bit_cyclic_dist(revbin(x,b), y, b);
    return (0==d);
}
```

There are six binary bracelets of length 4:

<table>
<thead>
<tr>
<th>Class</th>
<th>Words</th>
<th>#</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>....</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1... .1... .1... .1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11.1 111. 111... ..111 .111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11.1 111. 111... ..111 .111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11.1 111. 111... ..111 .111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11.1 111. 111... ..111 .111</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1111</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

[fxtbook draft of 2008-August-17]
The unlabeled binary bracelets are obtained by additionally allowing for bit-wise complementation:

```cpp
bool bu_equiv_q(ulong x, ulong y) // unlabeled bracelets
{
ulong d = bit_cyclic_dist(x, y, b);
x ^= mm;
if ( 0!=d ) d = bit_cyclic_dist(x, y, b);
x = revbin(x,b);
if ( 0!=d ) d = bit_cyclic_dist(x, y, b);
if ( 0!=d ) d = bit_cyclic_dist(x, y, b);
x ^= mm;
if ( 0!=d ) d = bit_cyclic_dist(x, y, b);
x ^= mm;
if ( 0!=d ) d = bit_cyclic_dist(x, y, b);
return (0==d);
}
```

There are four unlabeled binary bracelets of length 4:

- 0: 1111 .... [#=2]
- 1: 111. 11.1 1.11 1... .111 ...1 ..1. .1.. [#=8]
- 3: .11. 1..1 11.. ..11 [#=4]
- 5: .1.1 1.1. [#=2]

The shown functions are given in [FXT: sort/quivclass-bracelets-demo.cc] which can be used to produce listings of the equivalence classes.

We give the number of binary necklaces ‘N’, bracelets ‘B’, unlabeled necklaces ‘N/U’ and unlabeled bracelets ‘B/U’. The second row gives the sequence number of [245].

<table>
<thead>
<tr>
<th>n</th>
<th>N</th>
<th>B</th>
<th>N/U</th>
<th>B/U</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>13</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>18</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>36</td>
<td>30</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>60</td>
<td>46</td>
<td>30</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>108</td>
<td>78</td>
<td>56</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>188</td>
<td>126</td>
<td>94</td>
<td>63</td>
</tr>
<tr>
<td>12</td>
<td>352</td>
<td>224</td>
<td>180</td>
<td>122</td>
</tr>
<tr>
<td>13</td>
<td>632</td>
<td>380</td>
<td>316</td>
<td>190</td>
</tr>
<tr>
<td>14</td>
<td>1182</td>
<td>687</td>
<td>596</td>
<td>362</td>
</tr>
<tr>
<td>15</td>
<td>2192</td>
<td>1224</td>
<td>1096</td>
<td>612</td>
</tr>
</tbody>
</table>

### 3.8.1.5 Binary words with reversion and complement

The set $S$ of $n$-bit binary words and the equivalence relation identifying two words $x$ and $y$ whenever they are mutual complements or bit-wise reversals.

For example, the equivalence classes with 3-, 4- and 5-bit words are shown in figure 3.8-A. The sequence of numbers of equivalence classes for word-sizes $n$ is (entry A005418 of [245]):

- $n$: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
- #: 1, 2, 3, 6, 10, 20, 36, 72, 136, 272, 528, 1056, 2080, 4160, 8256, 16512, ...

The equivalence classes can be computed with the program [FXT: sort/quivclass-bitstring-demo.cc].

We have chosen examples where the resulting equivalence classes can be verified by inspection. For example, we could create the subsets of equivalent necklaces by simply rotating a given word and marking the so far visited words. Such an approach, however, is not possible in general when the equivalence relation does not have an obvious structure.
3.9 Determination of monotonicity and convexity *

A sequence is called **monotone** if it is either purely ascending or purely descending. This includes the case where subsequent elements are equal. Whether a constant sequence is considered ascending or descending in this context is a matter of convention.

A routine to check for monotonicity is [FXT:sort/monotone.h]:

```cpp
template <typename Type>
int is_monotone(const Type *f, ulong n)
// Return
// +1 for ascending order
// -1 for descending order
// else 0
{
    if (f[1] >= n) return +1;
    ulong k;
    // Implementation...
}
```

An implementation in C++ is given in [FXT:comb/bell-number-demo.cc]. An alternative way to compute the Bell Numbers is shown in section [15.2 on page 345].

---

### Figure 3.8-A: Equivalence classes of binary words where words are identified if either their reversals or complements are equal.

| 0: | 111
| 1: | 1 .1 .1
<table>
<thead>
<tr>
<th>2:</th>
<th>1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 classes with 4-bit words:</td>
<td></td>
</tr>
</tbody>
</table>
| 0: | 111 1111 ...
| 1: | 1 1 1 1... 1111
| 2: | 1 1 1 1 1...
| 3: | 1 1 1 1 1...
| 4: | 1 1 1 1 1...
| 5: | 1 1 1 1 1...
| 6: | 1 1 1 1 1...

---

### 3.8.2 The number of equivalence relations for a set of \( n \) elements

The sequence \( B(n) \) of the number of possible partitionings (and thereby equivalence relations) for the set \( \{1, 2, \ldots, n\} \) starts as \( n \geq 1 \):

\[
1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, \ldots
\]

These are the Bell numbers, sequence [A000110] of [245]. They can be computed easily as indicated in the following table:

<table>
<thead>
<tr>
<th>( n )</th>
<th>( B(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>[ 1 ]</td>
</tr>
<tr>
<td>1:</td>
<td>[ 1, 2 ]</td>
</tr>
<tr>
<td>2:</td>
<td>[ 2, 3, 5 ]</td>
</tr>
<tr>
<td>3:</td>
<td>[ 5, 7, 10, 15 ]</td>
</tr>
<tr>
<td>4:</td>
<td>[ 15, 20, 27, 37, 52 ]</td>
</tr>
<tr>
<td>5:</td>
<td>[ 52, 67, 87, 114, 151, 203 ]</td>
</tr>
<tr>
<td>6:</td>
<td>[B(6), ... ]</td>
</tr>
</tbody>
</table>

The first element in each row is the last element of the previous row, the remaining elements are the sum of their left and upper left neighbors. As pari/gp code:

```plaintext
N=7; v=w=b=vector(N); v[1]=1;
{ for(n=1,N-1,
    b[n] = v[1];
    print(n-1, " ", v); \ \ print row
    w[1] = v[n];
    for(k=2,n+1, w[k]=w[k-1]+v[k-1]);
    v=w;
    ); }
```

An implementation in C++ is given in [FXT:comb/bell-number-demo.cc]. An alternative way to compute the Bell Numbers is shown in section [15.2 on page 345].
Chapter 3: Sorting and searching

for (k=1; k<n; ++k) // skip constant start
{
    if ( f[k] != f[k-1] ) break;
}
if ( k==n ) return +1; // constant is considered ascending here
int s = ( f[k] > f[k-1] ? +1 : -1 );
if ( s>0 ) // was: ascending
{
    // scan for descending pair:
    for ( ; k<n; ++k) if ( f[k] <= f[k-1] ) return 0;
}
else // was: descending
{
    // scan for ascending pair:
    for ( ; k<n; ++k) if ( f[k] >= f[k-1] ) return 0;
}
return s;

A strictly monotone sequence is a monotone sequence that has no identical pairs of elements. The test turns out to be slightly easier:

template <typename Type>
int is_strictly_monotone(const Type *f, ulong n)
// return
// +1 for strictly ascending order
// -1 for strictly descending order
// else 0
{
    if ( 1>=n ) return +1;
    ulong k = 1;
    if ( f[k] == f[k-1] ) return 0;
    int s = ( f[k] > f[k-1] ? +1 : -1 );
    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for ( ; k<n; ++k) if ( f[k] <= f[k-1] ) return 0;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for ( ; k<n; ++k) if ( f[k] >= f[k-1] ) return 0;
    }
    return s;
}

A sequence is called convex if it starts with an ascending part and ends with a descending part. A concave sequence starts with a descending and ends with an ascending part. Whether a monotone sequence is considered convex or concave again is a matter of convention (you have the choice to consider the first or the last element as extremum). Lacking a term that contains both convex and concave the following routine is called is_convex() [FXT: sort/convex.h]:

template <typename Type>
long is_convex(Type *f, ulong n)
// Return
// +val for convex sequence (first rising then falling)
// -val for concave sequence (first falling then rising)
// else 0
// val is the (second) index of the first pair at the point
// where the ordering changes; val>=n iff seq. is monotone.
// Note: a constant sequence is considered any of rising/falling
{
    if ( 1>=n ) return +1;
    ulong k = 1;
    for (k=1; k<n; ++k) // skip constant start
    {

3.9: Determination of monotonicity and convexity *

```cpp
template <typename Type>
long is_strictly_convex(Type *f, ulong n)
{
  // Return
  // +val for strictly convex sequence (i.e. first strictly rising then strictly falling)
  // -val for strictly concave sequence (i.e. first strictly falling then strictly rising)
  // else 0
  // val is the (second) index of the first pair at the point where the ordering changes; val>n iff seq. is strictly monotone.
  if ( n>=n ) return +1;
  ulong k = 1;
  if ( f[k] == f[k-1] ) return 0;
  int s = ( f[k] > f[k-1] ? +1 : -1 );
  if ( s>0 ) // was: ascending
  {
    // scan for descending pair:
    for ( ; k<n; ++k) if ( f[k] <= f[k-1] ) break;
    s = +k;
  }
  else // was: descending
  {
    // scan for ascending pair:
    for ( ; k<n; ++k) if ( f[k] >= f[k-1] ) break;
    s = -k;
  }
  if ( k==n ) return s; // sequence is monotone
  // check that the ordering does not change again:
  if ( s>0 ) // was: ascending --> descending
  {
    // scan for ascending pair:
    for ( ; k<n; ++k) if ( f[k] > f[k-1] ) return 0;
  }
  else // was: descending
  {
    // scan for descending pair:
    for ( ; k<n; ++k) if ( f[k] < f[k-1] ) return 0;
  }
  return s;
}
```

The test for strictly convex (or concave) sequences is:

```cpp
template <typename Type>
long is_strictly_convex(Type *f, ulong n)
{
  // Return
  // +val for strictly convex sequence (i.e. first strictly rising then strictly falling)
  // -val for strictly concave sequence (i.e. first strictly falling then strictly rising)
  // else 0
  // val is the (second) index of the first pair at the point where the ordering changes; val>n iff seq. is strictly monotone.
  if ( n>=n ) return +1;
  ulong k = 1;
  if ( f[k] == f[k-1] ) return 0;
  int s = ( f[k] > f[k-1] ? +1 : -1 );
  if ( s>0 ) // was: ascending
  {
    // scan for descending pair:
    for ( ; k<n; ++k) if ( f[k] <= f[k-1] ) break;
    s = +k;
  }
  else // was: descending
  {
    // scan for ascending pair:
    for ( ; k<n; ++k) if ( f[k] >= f[k-1] ) break;
    s = -k;
  }
  if ( k==n ) return s; // sequence is monotone
  else if ( f[k] == f[k-1] ) return 0;
  // check that the ordering does not change again:
  if ( s>0 ) // was: ascending --> descending
  {
    // scan for ascending pair:
    for ( ; k<n; ++k) if ( f[k] > f[k-1] ) return 0;
  }
  else // was: descending
  {
    // scan for descending pair:
    for ( ; k<n; ++k) if ( f[k] < f[k-1] ) return 0;
  }
  return s;
}
```
3.10 Heapsort

The heapsort algorithm uses the heap data structure introduced in section 4.5.1 on page 154. A heap can be sorted by swapping the first (and biggest) element with the last and ‘repairing’ the array of size $n - 1$ by a call to `heapify1()`. Applying this idea recursively until there is nothing more to sort leads to the routine [FXT: sort/heapsort.h]:

```cpp
1 template <typename Type>
2 void heap_sort_ascending(Type *x, ulong n)
3 // Sort an array that has the heap-property into ascending order.
4 // On return x[] is _not_ a heap anymore.
5 {
6     Type *p = x - 1;
7     for (ulong k=n; k>1; --k)
8         {
9             swap2(p[1], p[k]); // move largest to end of array
10             --n; // remaining array is one element less
11             heapify1(p, n, 1); // restore heap-property
12         }
13 }
```

that needs time $O(n \log(n))$. That is, a call to

```cpp
1 template <typename Type>
2 void heap_sort(Type *x, ulong n)
3 {
4     build_heap(x, n);
5     heap_sort_ascending(x, n);
6 }
```

will sort the array $x[]$ into ascending order. Note that sorting into descending order is not any harder:

```cpp
1 template <typename Type>
2 void heap_sort_descending(Type *x, ulong n)
3 // Sort an array that has the heap-property into descending order.
4 // On return x[] is _not_ a heap anymore.
5 {
6     Type *p = x - 1;
7     for (ulong k=n; k>1; --k)
8         {
9             ++p; --n; // remaining array is one element less
10             heapify1(p, n, 1); // restore heap-property
11         }
12 }
```

A program that demonstrates the algorithm is [FXT: sort/heapsort-demo.cc].

3.11 Counting sort and radix sort

Imagine you want to sort an $n$-element array $F$ of (unsigned) 8-bit values. An sorting algorithm that only uses 2 passes through the data proceeds as follows:

1. Allocate an array $C$ of 256 integers and set all its elements to zero.
3.11: Counting sort and radix sort

2. Count: for \( k = 0, 1, \ldots, n - 1 \) increment \( C[F[k]] \).
   Now \( C[x] \) contains how many bytes in \( F \) have the value \( x \).

3. Set \( r = 0 \). For \( j = 0, 1, \ldots, 255 \)
   set \( k = C[j] \) and write \( j \) to the elements \( F[r], F[r+1], \ldots, F[r+k-1] \) then add \( k \) to \( r \).

For large values of \( n \) this method is significantly faster than any other sorting algorithm. Note that no comparisons are made between the elements of \( F \). Instead they are counted, the algorithm is the counting sort algorithm.

It might seem that the idea applies only to very special cases but with a little care it can be used in more general situations. We modify the method so that we are able to sort also (unsigned) integer variables whose range of values would make the method impractical with respect to a subrange of the bits in each word. We need an array \( G \) that has as many elements as \( F \):

1. Choose any consecutive run of \( b \) bits, these will be represented by a bit mask \( m \). Allocate an array \( C \) of \( 2^b \) integers and set all its elements to zero.

2. Let \( M \) be a function that maps the \((2^b)\) values of interest (the bits masked out by \( m \)) to the range \( 0, 1, \ldots, 2^b - 1 \).

3. Count: for \( k = 0, 1, \ldots, n - 1 \) increment \( C[M(F[k])] \).
   Now \( C[x] \) contains how many values of \( M(F[i]) \) equal \( x \).

4. Cumulate: for \( j = 1, 2, \ldots, 2^b - 1 \) (second to last) add \( C[j - 1] \) to \( C[j] \).
   Now \( C[x] \) contains the number of values \( M(F[i]) \) less or equal to \( x \).

5. Copy: for \( k = n - 1, \ldots, 2, 1, 0 \) (last to first) set \( x := M(F[k]) \), decrement \( C[x] \) then set \( i = C[x] \), then set \( G[i] := F[x] \).

A crucial property of the algorithm is that it is stable: when we use it to sort with respect to a certain bitmask \( m \) and there is more than one element being mapped to the same value then the relative order between these elements is preserved.

Input Counting sort wrt. two lowest bits
\[
\begin{array}{c|c|c|c}
\text{m = ......11} & \text{m = ......11} & \text{m = ......11} & \text{m = ......11} \\
0: & 11111.11< & 0: & 11111.11< \\
...1:1:1 & ...1:1:1 & ...1:1:1 & ...1:1:1 \\
...1:0:1 & ...1:0:1 & ...1:0:1 & ...1:0:1 \\
...1:1:1< & ...1:1:1< & ...1:1:1< & ...1:1:1< \\
...1:1:1 & ...1:1:1 & ...1:1:1 & ...1:1:1 \\
...1:1:1< & ...1:1:1< & ...1:1:1< & ...1:1:1< \\
...1:1:1 & ...1:1:1 & ...1:1:1 & ...1:1:1 \\
\end{array}
\]

Note that relative order of the three words ending with two set bits (marked with ‘<’) is preserved.

A routine that verifies whether an array is sorted with respect to a bit range specified by the variable \( b0 \) and \( m \) is [FXT: sort/radixsort.cc]:

```cpp
bool is_counting_sorted(const ulong *f, ulong n, ulong b0, ulong m)
// Whether f[] is sorted wrt. bits b0,...,b0+z-1
// where z is the number of bits set in m.
// m must contain a single run of bits starting at bit zero.
{
    m <<= b0;
    for (ulong k=1; k<n; ++k)
    { 
        ulong xm = (f[k-1] & m ) >> b0;
        ulong xp = (f[k] & m ) >> b0;
        if ( xm>xp ) return false;
    }
    return true;
}
```

The function \( M \) is the combination of a mask-out and a shift operation. A routine that sorts according to \( b0 \) and \( m \) is:

```cpp
void counting_sort_core(const ulong * restrict f, ulong n, ulong * restrict g, ulong b0, ulong m)
```
Chapter 3: Sorting and searching

3 // Write to g[] the array f[] sorted wrt. bits b0,...,b0+z-1
4 // where z is the number of bits set in m.
5 // m must contain a single run of bits starting at bit zero.
6 {
7   ulong nb = m + 1;
8   m <<= b0;
9   ALLOCA(ulong, cv, nb);
10  for (ulong k=0; k<nb; ++k) cv[k] = 0;
11  // --- count:
12  for (ulong k=0; k<n; ++k)
13  {
14    ulong x = (f[k] & m) >> b0;
15    ++cv[x];
16  }
17  // --- cumulative sums:
18  for (ulong k=1; k<nb; ++k) cv[k] += cv[k-1];
19  // --- reorder:
20  ulong k = n;
21  while ( k-- ) // backwards ==> stable sort
22  {
23    ulong fk = f[k];
24    ulong x = (fk & m) >> b0;
25    --cv[x];
26    ulong i = cv[x];
27    g[i] = fk;
28  }
29 }

<table>
<thead>
<tr>
<th>Input</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = ....11</td>
<td>m = ..11..</td>
<td>m = 11....</td>
<td></td>
</tr>
<tr>
<td>vv</td>
<td>vv</td>
<td>vv</td>
<td></td>
</tr>
<tr>
<td>111.11</td>
<td>.1...</td>
<td>11....</td>
<td>.1...</td>
</tr>
<tr>
<td>.1.1.i</td>
<td>111..</td>
<td>.1...</td>
<td>.1.1.i</td>
</tr>
<tr>
<td>1...1.</td>
<td>1.1.1</td>
<td>11.1</td>
<td>.1.1.1</td>
</tr>
<tr>
<td>11111</td>
<td>.1...</td>
<td>11.1</td>
<td>.1...</td>
</tr>
<tr>
<td>.1.1.i</td>
<td>.1...</td>
<td>11...</td>
<td>.1...</td>
</tr>
<tr>
<td>1.1.i</td>
<td>1111</td>
<td>1111</td>
<td>11.1.i</td>
</tr>
<tr>
<td>.1.1</td>
<td>11111</td>
<td>1111</td>
<td>11111</td>
</tr>
<tr>
<td>11....</td>
<td>1....</td>
<td>1....</td>
<td>1....</td>
</tr>
<tr>
<td>.......</td>
<td>.......</td>
<td>.......</td>
<td>.......</td>
</tr>
</tbody>
</table>

Figure 3.11-A: Radix sort of 10 six-bit values when using two-bit masks.

Now we can apply counting sort to a set of bit masks that cover the whole range. Figure 3.11-A shows an example with 10 six-bit values and 3 two-bit masks, starting from the least significant bits. This is the output of the program [FXT: sort/radixsort-demo.cc].

The routine [FXT: radix_sort()] in [sort/radixsort.cc] uses 8-bit masks to sort unsigned (long) integers:

```c
void
radix_sort(ulong *f, ulong n)
{
  ulong nb = 8; // Number of bits sorted with each step
  ulong tnb = BITS_PER_LONG; // Total number of bits
  ulong *fi = f;
  ulong *g = new ulong[n];
  ulong m = (1UL<<nb) - 1;
  for (ulong k=1, b0=0; b0<tnb; ++k, b0+=nb)
  {
    counting_sort_core(f, n, g, b0, m);
    swap2(f, g);
  }
  if ( f!=fi ) // result is actually in g[]
  {
    swap2(f, g);
    for (ulong k=0; k<n; ++k) f[k] = g[k];
  }
```

[fxtbook draft of 2008-August-17]
There is room for optimization. Using two arrays for counting and combining copying with counting for
the next pass where possible will reduce the number of passes almost by a factor of two.

A version of radix sort that starts from the most significant bits is given in [241].

3.12 Searching in unsorted arrays

We give an overview of the search functions for unordered arrays.

3.12.1 Minimal and maximal elements in unsorted arrays

Sorting an array only to determine the minimal (and/or maximal element) is obviously a bad idea because
these can be found in linear time in the unsorted data.

Corresponding to all flavors of the sorting routines a min() and max() routine is supplied. We exemplify
by giving the versions that determine the minimum.

Finding the minimum in an unsorted array corresponding to the ‘plain’ sorting routine: [FXT: sort/minmax.h]

```cpp
1 template <typename Type>
2 Type inline min(const Type *f, ulong n)
3     // Return minimum of array.
4     {
5         Type v = f[0];
6         while ( n-- ) if ( f[n]<v ) v = f[n];
7         return v;
8     }
```

The index version: [FXT: sort/minmaxidx.h]

```cpp
1 template <typename Type>
2 Type idx_min(const Type *f, ulong n, const ulong *x)
3     // Return minimum (value) of array elements
4     // {f[x[0]], f[x[1]], .. , f[x[n-1]]}
5     {
6         Type v = f[x[0]];
7         while ( n-- ) { if ( f[x[n]]<v ) v = f[x[n]]; }
8         return v;
9     }
```

The pointer version: [FXT: sort/minmaxptr.h]

```cpp
1 template <typename Type>
2 Type ptr_min(const Type *f, ulong n, Type const*const*x)
3     // Return minimum (value) of array elements
4     // {*x[0], *x[1], .. , *x[n-1]}
5     {
6         Type v = *x[0];
7         while ( n-- ) { if ( *x[n]<v ) v = *x[n]; }
8         return v;
9     }
```

The comparison function version: [FXT: sort/minmaxfunc.h]

```cpp
1 template <typename Type>
2 Type min(const Type *f, ulong n, int (*cmp)(const Type &, const Type &))
3     // Return minimum (value) of array elements
4     // wrt. to comparison function
5     {
6         Type v = f[0];
7         while ( n-- ) { if ( cmp(f[n],v) < 0 ) v = f[n]; }
8         return v;
9     }
```
Chapter 3: Sorting and searching

The comparison function with index version: [FXT: sort/minmaxidxfnc.h]

```cpp
template<typename Type>
type idx_min(const Type *f, ulong n, const ulong *x, 
    int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements 
// {f[x[0]], f[x[1]], ..., f[x[n-1]]} 
// with respect to comparison function cmp() 
{
    Type v = f[x[0]]; 
    while ( n-- ) { if ( cmp(f[x[n]], v) < 0 ) v = f[x[n]]; } 
    return v;
}
```

The comparison function with pointer version: [FXT: sort/minmaxptrfunc.h]

```cpp
template<typename Type>
type ptr_min(const Type *f, ulong n, Type const*const*x, 
    int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements 
// {*x[0], *x[1], ..., *x[n-1]} 
// with respect to comparison function cmp(). 
{
    Type v = *x[0]; 
    while ( n-- ) { if ( cmp(*x[n],v)<0 ) v = *x[n]; } 
    return v;
}
```

### 3.12.2 Searching for values

To find the first occurrence of a certain value in an unsorted array one can use the routine [FXT: sort/usearch.h]

```cpp
template<typename Type>
inline ulong first_eq_idx(const Type *f, ulong n, Type v)
// Return index of first element == v 
// Return n if all !=v 
{
    ulong k = 0; 
    while ( (k<n) && (f[k]!=v) ) k++; 
    return k;
}
```

The functions `first_ne_idx()`, `first_ge_idx()` and `first_le_idx()` find the first occurrence of an element unequal (to `v`), greater or equal and less or equal, respectively.

If the last bit of speed matters, one could (see [170, p.267]) replace the shown code by

```cpp
template<typename Type>
inline ulong first_eq_idx(const Type *f, ulong n, Type v)
// Return index of first element == v 
// Return n if all !=v 
{
    Type s = f[n-1]; 
    (Type *)f[n-1] = v; // guarantee that the search stops 
    ulong k = 0; 
    while ( (f[k]!=v) ) k++; 
    return k;
}
```

The speedup is due to the fact that there is only one branch in the inner loop. The technique is not applicable if the writes to the array `f[]` can have any side effects.

Replace the word `first` by last in the names to name the function that detect the last element satisfying the corresponding condition. For example,
3.12: Searching in unsorted arrays

```
1 template<typename Type>
2 inline ulong last_ge_idx(const Type *f, ulong n, Type v)
3 // Return index of last element >= v
4 // Return n if all <v
5 {
6     ulong k = n-1;
7     while ( f[k]<v )
8         { k--; }  // changed from 0==k to 0<v
9         if ( 0==k ) return n;
10     } 
11     return k;
12 }
```

### 3.12.3 Counting values

The functions in [FXT: sort/ucount.h] count certain elements in unsorted arrays.

```
1 template<typename Type>
2 inline ulong eq_count(const Type *f, ulong n, Type v)
3 // Return number of elements that are ==v
4 {
5     ulong ct = 0;
6     while ( n-- ) if ( f[n]==v ) ++ct;
7     return ct;
8 }
```

As above, replace `eq_` in the function name with `ne_`, `ge_` or `le_` to count the number of occurrences of elements that are unequal (to `v`), greater or equal and less or equal, respectively.

### 3.12.4 Searching matches

The routines in [FXT: sort/usearchfunc.h] generalize the functions from [FXT: sort/usearch.h]: A supplied function implements the condition imposed. For example,

```
1 template<typename Type>
2 inline ulong first_idx(const Type *f, ulong n, bool (* func)(Type))
3 // Return index of first element for which func() returns true.
4 // Return n if there is no such element.
5 {
6     ulong k = 0;
7     while ( (k<n) && (!func(f[k])) ) k++;
8     return k;
9 }
```

and `last_idx()` that scans beginning from the greatest index.

The functions

```
1 template<typename Type>
2 inline ulong next_idx(const Type *f, ulong n, bool (* func)(Type), ulong k0)
3 // Like first_idx() but start from k0.
4 {
5     ulong k = k0;
6     while ( (k<n) && (!func(f[k])) ) k++;
7     return k;
8 }
```

and `previous_idx()` determine the next matching element in forward or backward direction.

### 3.12.5 Selecting matches: grep

The routines from [FXT: sort/grep.h] count or select elements from an unordered array for which a condition implemented by a supplied function is true.

The following function counts the matching elements:

[fxtbook draft of 2008-August-17]
template<typename Type>
inline ulong count(const Type *f, ulong n, bool (*func)(Type))
// Return number of elements for which func() returns true.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k) if (func(f[k])) ct++;
    return ct;
}

Discard all non-matches using

template<typename Type>
inline ulong grep(Type *f, ulong n, bool (*func)(Type))
// Delete elements for which func() returns false.
// Return number of elements kept.
{
    ulong k, j;
    for (k=0, j=0; j<n; ++k, ++j)
    {
        f[k] = f[j];
        if (func(f[j])) --k;
    }
    return k;
}

Record the values of the matches using

template<typename Type>
inline ulong grep(const Type *f, ulong n, bool (*func)(Type), Type *g)
// Make g[] the sequence of values for which func() returns true.
// Return number of 'matching' elements found.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k) if (func(f[k])) g[ct++] = f[k];
    return ct;
}

Finally, recording the indices of the matches can be done with

template<typename Type>
inline ulong grep_idx(const Type *f, ulong n, bool (*func)(Type), ulong *x)
// Make x[] the sequence of indices for which func() returns true.
// Return number of 'matching' elements found.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k) if (func(f[k])) x[ct++] = k;
    return ct;
}
Chapter 4

Data structures

We give implementations of selected data structures like stack, ring buffer, queue, double-ended queue (deque), but-array, heap and priority queue. We also describe a finite state engine, the emulation of coroutines, and left-right arrays.

4.1 Stack (LIFO)

A stack (or LIFO, for last-in, first-out) is a data structure that supports the operations: push() to save an entry, pop() to retrieve and remove the entry that was entered last, and peek() to retrieve the element that was entered last without removing it. The method poke() modifies the last entry. An implementation with the option to let the stack grow when necessary is [FXT: class stack in ds/stack.h]:

```cpp
template <typename Type>
class stack
{
public:
    Type *x_; // data
    ulong s_; // size
    ulong p_; // stack pointer (position of next write), top entry @ p-1
    ulong gq_; // grow gq elements if necessary, 0 for "never grow"

public:
    stack(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        p_ = 0; // stack is empty
        gq_ = growq;
    }

    ~stack() { delete [] x_; }

    ulong num() const { return p_; } // Return number of entries.

    ulong push(Type z)
    // Add element z on top of stack.
    // Return size of stack, zero on stack overflow.
    // If gq_ is nonzero the stack grows if needed.
    {
        if ( p_ >= s_ )
            if ( 0==gq_ ) return 0; // overflow
                grow();
        x_[p_] = z;
        ++p_;  
```
Chapter 4: Data structures

return s_
}
ulong pop(Type &z)
// Retrieve top entry and remove it.
// Return number of entries before removing element.
// If empty return zero and leave z is undefined.
{
    ulong ret = p_
    if (0 != p_)
        --p_; z = x_[p_];
    return ret;
}
ulong poke(Type z)
// Modify top entry.
// Return number of entries.
// If empty return zero and do nothing.
{
    if (0 != p_)
        x_[p_-1] = z;
    return p_;
}
ulong peek(Type &z)
// Read top entry, without removing it.
// Return number of entries.
// If empty return zero and leave z undefined.
{
    if (0 != p_)
        z = x_[p_-1];
    return p_;
}

The growth routine is implemented as
private:
void grow()
{
    ulong ns = s_ + gq_; // new size
    x_ = ReAlloc<Type>(x_, ns, s_);
    s_ = ns;
}

here we use the function ReAlloc() that imports the C function realloc().

% man realloc
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

Usage of the C function realloc() can be disabled globally in \[FXT: realloc.h\]:
#define USE_C_REALLOC // comment out to disable use of realloc()
#endif
#ifdef USE_C_REALLOC
    template <typename Type>
    inline Type *ReAlloc<Type> (Type *p, ulong n, ulong /*nold*/)
    {
        return (Type *)realloc((void *)p, n*sizeof(Type));
    }
#endif
else
    template <typename Type>
    inline Type *ReAlloc<Type> (Type *p, ulong n, ulong nold)
    {
        Type *np = new Type[n];
        ulong nc = (nold < n ? nold : n);

4.2 Ring buffer

A **ring buffer** is an array plus read and write operations that wrap around. That is, if the last position of the array is reached writing continues at the begin of the array, thereby erasing the oldest entries. The read operation should start at the oldest entry in the array.

The implementation is [FXT: class ringbuffer in ds/ringbuffer.h]:

```cpp
template<typename Type>
class ringbuffer {
  public:
    Type *x_; // data (ring buffer)
    ulong s_; // allocated size (# of elements)
    ulong n_; // current number of entries in buffer
    ulong wpos_; // next position to write in buffer
    ulong fpos_; // first position to read in buffer
  }

  public:
    ringbuffer(ulong n) {
      s_ = n;
    }
```

A program that shows the working of the stack is [FXT: ds/stack-demo.cc]. An example output where the initial size is 4 and the growths-feature enabled (in steps of 4 elements) is shown in figure 4.1-A.


4.3 Queue (FIFO)

A queue (or FIFO for first-in, first-out) is a data structure that supports the following operations: push() saves an entry, pop() retrieves (and removes) the entry that was entered least recently, and peek() retrieves the least recently entered element without removing it.

A utility class with the optional feature of growing if necessary is [FXT: class queue in ds/queue.h]:

```cpp
template <typename Type>
class queue
{
    public:
        Type *x_; // pointer to data
        ulong s_;  // allocated size (# of elements)
        ulong n_;  // current number of entries in buffer
        ulong wpos_; // next position to write in buffer
        ulong rpos_; // next position to read in buffer
        ulong gq_;  // grow gq elements if necessary, 0 for "never grow"

    public:
        explicit queue(ulong n, ulong growq=0)
        {
            s_ = n;
        }

    private:
        // other members...
};
```

Ring buffers can be useful for storing a constant amount of history-data such as for logging purposes. For that purpose one would enhance the ringbuffer class so that it uses an additional array of (fixed width) strings. The message to log would be copied into the array and the pointer set accordingly. A read should then just return the pointer to the string.
Its working is demonstrated by the program [FXT: ds/queue-demo.cc]. An example output where the initial size is 4 and the growths-feature enabled (in steps of 4 elements) is shown in figure [4.3-A]. Compare to the corresponding figure for the stack, figure [4.1-A] on page 149.
Chapter 4: Data structures

Figure 4.3-A: Inserting and retrieving elements with a queue.

4.4 Deque (double-ended queue)

A deque (for double-ended queue) combines the data structures stack and queue: insertion and deletion in time $O(1)$ is possible both at the first and the last position. An implementation with the option to let the stack grow when necessary is [FXT: class deque in ds/deque.h]

```cpp
template<typename Type>
class deque {
public:
    Type *x_;       // data (ring buffer)
    ulong s_;       // allocated size (# of elements)
    ulong n_;       // current number of entries in buffer
    ulong fpos_;    // position of first element in buffer
    ulong lpos_;    // position of last element in buffer plus one

    // insert_first() will write to (fpos-1)%n
    // insert_last() will write to lpos, n=(lpos-fpos) (mod s)
    // entries are at [fpos, ..., lpos-1] (range may be empty)
    ulong gq_;      // grow gq elements if necessary, 0 for "never grow"

public:
    explicit deque(ulong n, ulong growq=0) {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        fpos_ = 0;
        lpos_ = 0;
        gq_ = growq;
    }
    ~deque() { delete [] x_; }
};
```

[fxtbook draft of 2008-August-17]
4.4: Deque (double-ended queue)

ulong num() const { return n_; }

The insertion at the front and end are implemented as

ulong insert_first(const Type &z)
// Return number of entries after insertion.// Zero is returned on failure// (i.e. space exhausted and 0==gq_)
{
if ( n_ >= s_ )
{
if ( 0==gq_ ) return 0; // growing disabled
grow();
}
--fpos_;
if ( fpos_ == -1UL ) fpos_ = s_ - 1;
if ( fpos_ == -1UL ) fpos_ = s_ - 1;
x_[fpos_] = z;
++n_; return n_;  
}
ulong insert_last(const Type &z)
// Return number of entries after insertion.// Zero is returned on failure// (i.e. space exhausted and 0==gq_)
{
if ( n_ >= s_ )
{
if ( 0==gq_ ) return 0; // growing disabled
grow();
}
x_[lpos_] = z;
++lpos_; return n_;  
}

The extraction methods are

ulong extract_first(Type & z)
// Return number of elements before extract.// Return 0 if extract on empty deque was attempted.
{
if ( 0==n_ ) return 0;
z = x_[fpos_];
++fpos_; if ( fpos_ >= s_ ) fpos_ = 0;
--n_; return n_ + 1;
}
ulong extract_last(Type & z)
// Return number of elements before extract.// Return 0 if extract on empty deque was attempted.
{
if ( 0==n_ ) return 0;
--lpos_; if ( lpos_ == -1UL ) lpos_ = s_ - 1;
z = x_[lpos_];
--n_; return n_ + 1;
}

Reading at the front, end, or an arbitrary index without changing any data:

ulong read_first(Type & z) const
// Read (but don’t remove) first entry.// Return number of elements (i.e. on error return zero).
{
if ( 0==n_ ) return 0;
z = x_[fpos_]; return n_;  
}
ulong read_last(Type & z) const
// Read (but don’t remove) last entry.
// Return number of elements (i.e. on error return zero).
{
    return read(n_-1, z); // ok for n_==0
}
ulong read(ulong k, Type & z) const
// Read entry k (that is, [(fpos_ + k)%s_]).
// Return 0 if k>=n_ else return k+1
{
    if ( k>=n_ ) return 0;
    ulong j = fpos_ + k;
    if ( j>=s_ ) j -= s_;
    z = x_[j];
    return k + 1;
}

private:
void grow()
{
    ulong ns = s_ + gq_; // new size
    // Move read-position to zero:
    rotate_left(x_, s_, fpos_);
    x_ = ReAlloc<Type>(x_, ns, s_);
    fpos_ = 0;
    lpos_ = n_;  // n_ = ns;
}

insert_first( 1) 1
insert_last(51) 1 51
insert_first( 2) 2 1 51
insert_last(52) 2 1 51 52
insert_first( 3) 3 2 1 51 52
insert_last(53) 3 2 1 51 52 53
extract_first()= 3 2 1 51 52 53
insert_first( 4) 4 2 1 51 52 54
insert_last(54) 4 2 1 51 52 54
extract_first()= 4 2 1 51 52 54
extract_last()= 54 2 1 51 52
extract_first()= 2 1 51 52
extract_last()= 52 1 51 52
extract_first()= 1 51
extract_last()= 51
extract_first()= 5
insert_last(55) 5 55
extract_first()= 5 55
extract_last()= 55
extract_first()= (deque is empty)
exract_last()= (deque is empty)
insert_first( 7) 7
insert_last(57) 7 57

Figure 4.4-A: Inserting and retrieving elements with a queue.

Its working is shown in figure 4.4-A which was created with the program [FXT: ds/deque-demo.cc].

4.5 Heap and priority queue

4.5.1 The binary heap

A binary heap is a binary tree where the left and right children are smaller than or equal to their parent node. The following function determines whether a given array is a heap [FXT: ds/heap.h].

template<typename Type>
ulong test_heap(const Type *x, ulong n)
4.5: Heap and priority queue

It turns out that an unordered array of size \( n \) can be reordered to a heap in \( O(n) \) time, see [95, p.145]. The routine

```cpp
template <typename Type>
void build_heap(Type *x, ulong n)
// Reorder data to a heap.
{
    for (ulong j=(n>>1); j>0; --j) heapify1(x-1, n, j);
}
```

does the trick. It uses the following subroutine that runs in time \( O(\log n) \):

```cpp
template <typename Type>
void heapify1(Type *z, ulong n, ulong k)
// Subject to the condition that the trees below the children of node
// k are heaps, move the element z[k] (down) until the tree below node k is a heap.
// Data expected in z[1,2,...,n].
{
    ulong m = k;
    ulong l = (k<<1); // left(k);
    if ( (l <= n) && (z[l] > z[k]) ) m = l;
    ulong r = (k<<1) + 1; // right(k);
    if ( (r <= n) && (z[r] > z[m]) ) m = r;
    if ( m != k )
    {
        swap2(z[k], z[m]);
        heapify1(z, n, m);
    }
}
```

### 4.5.2 Priority queue

Heaps are useful to build a so-called **priority queue**. This is a data structure that supports insertion of an element and extraction of the maximal element it contains both in time \( O(\log(n)) \). A priority queue can be used to schedule an event for a given point in time and return the next pending event.

A new element can be inserted into a heap in \( O(\log(n)) \) time by appending it and moving it towards the root (that is, the first element) as necessary:

```cpp
bool heap_insert(Type *x, ulong n, ulong s, Type t)
// with x[] a heap of current size n
// and max size s (i.e. space for s elements allocated),
// insert t and restore heap-property.
// Return true if successful, else (i.e. if space exhausted) false.
{
    if ( n > s ) return false;
    ++n;
    Type *x1 = x - 1;
    ulong j = n;
    while ( j > 1 )
    {
        ulong k = (j>>1); // k=parent(j)
        if ( x1[k] >= t ) break;
        x1[j] = x1[k];
        j = k;
    }
    x1[j] = t;
    return true;
}
```
Similarly, the maximal element can be removed in time $O(\log(n))$:

```cpp
template <typename Type>
Type heap_extract_max(Type *x, ulong n)
// Return maximal element of heap and restore heap structure.
// Return value is undefined for 0==n.
{
    Type m = x[0];
    if ( 0 != n )
    {
        Type *x1 = x - 1;
        x1[1] = x1[n];
        --n;
        heapify1(x1, n, 1);
    }
    return m;
}
```

Two modifications seem appropriate: Firstly, replace `extract_max()` by a `extract_next()`, leaving it as a compile time option whether to extract the minimal or the maximal element. This is achieved by changing the comparison operators at a few strategic places so that the heap is built either as described above or with its minimum as first element:

```cpp
#if 1
    // next() is the one with the smallest key
    // i.e. extract_next() is extract_min()
#define _CMP_ <
#define _CMPEQ_ <=
#else
    // next() is the one with the biggest key
    // i.e. extract_next() is extract_max()
#define _CMP_ >
#define _CMPEQ_ >=
#endif
```

Secondly, augmenting the elements used by a event description that can be freely defined. [FXT: `class priority_queue` in `ds/priorityqueue.h`]:

```cpp
template <typename Type1, typename Type2>
class priority_queue
{
public:
    Type1 *t1_; // time: t1[1..s] one-based array!
    Type2 *e1_; // events: e1[1..s] one-based array!
    ulong s_; // allocated size (# of elements)
    ulong n_; // current number of events
    ulong gq_; // grow gq elements if necessary, 0 for "never grow"

public:
    priority_queue(ulong n, ulong growq=0)
    {
        s_ = n;
        t1_ = new Type1[s_] - 1;
        e1_ = new Type2[s_] - 1;
        n_ = 0;
        gq_ = growq;
    }

    ~priority_queue()
    {
        delete [] (t1_+1);
        delete [] (e1_+1);
    }

    ulong num() const { return n_; }
    Type1 get_next_t() const { return t1_[1]; }
    Type2 get_next_e() const { return e1_[1]; }
    Type1 get_next(Type2 &e) const
    // No check if empty
    {
        e = e1_[1];
        return t1_[1];
    }
};
```
4.5: Heap and priority queue

Type1 extract_next(Type2 &e)
{
    Type1 m = get_next(e);
    if ( 0 != n_ )
    {
        t1_[1] = t1_[n_]; e1_[1] = e1_[n_];
        --n;
        heapify1(1);
    }
    // else error
    return m;
}

bool insert(const Type1 &t, const Type2 &e)
// Insert event e at time t
// Return true if successful
// else (i.e. space exhausted and 0==gq_) false
{
    if ( n_ >= s_ )
    {
        if ( 0==gq_ ) return false; // growing disabled
        grow();
    }
    ulong j = n_;    
    while ( j > 1 )
    {
        ulong k = (j>>1); // k==parent(j)
        if ( t1_[k] _CMPEQ_ t ) break;
        t1_[j] = t1_[k]; e1_[j] = e1_[k];
        j = k;
    }
    t1_[j] = t;
    e1_[j] = e;
    // return true;
}

The member function reschedule_next() is more efficient than the sequence extract_next(); insert(); as it calls heapify1() only once:

    void reschedule_next(Type1 t)
    {
        t1_[1] = t;
        heapify1(1);
    }

The heapify1() function is tail-recursive, so we make it iterative:

    private:
    void heapify1(ulong k)
    {
        ulong m = k;
        hstart:
        ulong l = (k<<1); // left(k);
        ulong r = l + 1; // right(k);
        if ( (l <= n_) && (t1_[l] _CMPEQ_ t1_[k]) ) m = l;
        if ( (r <= n_) && (t1_[r] _CMPEQ_ t1_[k]) ) m = r;
        if ( m != k )
        {
            swap2(t1_[k], t1_[m]); swap2(e1_[k], e1_[m]);
            // heapify1(m);
            k = m;
            goto hstart; // tail recursion
        }
    }

The second argument of the constructor determines the number of elements added in case of growth, it is disabled (equals zero) by default.
The ReAlloc() routine is described in section 4.1 on page 147.

The program [FXT: ds/priorityqueue-demo.cc] inserts events at random times $0 \leq t < 1$, then extracts all of them. It gives the output shown in figure 4.5-A. A more typical usage would intermix the insertions and extractions.

### 4.6 Bit-array

The use of *bit-arrays* should be obvious: an array of tag values (like ‘seen’ versus ‘unseen’) where all standard data types would be a waste of space. Besides reading and writing individual bits one should implement a convenient search for the next set (or cleared) bit.

The class [FXT: class bitarray in ds/bitarray.h] is used, for example, for lists of small primes [FXT: mod/primes.cc], for in-place transposition routines [FXT: aux2/transpose.h] (see section 2.3 on page 96) and several operations on permutations (see section 2.10.10 on page 116).

```cpp
class bitarray
// Bit-array class mostly for use as memory saving array of boolean values.
// Valid index is 0...nb_-1 (as usual in C arrays).
{
public:
ulong *f_; // bit bucket
ulong n_; // number of bits
ulong nfw_; // number of words where all bits are used, may be zero
ulong mp_; // mask for partially used word if there is one, else zero
// (ones are at the positions of the _unused_ bits)
bool myfq_; // whether f[] was allocated by class
[--snip--]

The constructor allocates memory by default. If the second argument is nonzero it must point to an accessible memory range:

```
The public methods are

```c
// operations on bit n:
ulong test(ulong n) const // Test whether n-th bit set
void set(ulong n) // Set n-th bit
void clear(ulong n) // Clear n-th bit
ulong test_set(ulong n) // Test whether n-th bit is set and set it
ulong test_clear(ulong n) // Test whether n-th bit is set and clear it
ulong test_change(ulong n) // Test whether n-th bit is set and toggle it

// Operations on all bits:
void clear_all() // Clear all bits
void set_all() // Set all bits
int all_set_q() const; // Return whether all bits are set
int all_clear_q() const; // Return whether all bits are clear

// Scanning the array:
ulong next_set_idx(ulong n) const // Return index of next set or value beyond end
ulong next_clear_idx(ulong n) const // Return index of next clear or value beyond end
```

Combined operations like ‘test-and-set-bit’, ‘test-and-clear-bit’, ‘test-and-change-bit’ are often needed in applications that use bit-arrays. This is underlined by the fact that modern CPUs typically have instructions implementing these operations.

The class does not supply overloading of the array-index operator [ ] because the writing variant would cause a performance penalty. One might want to add ‘sparse’-versions of the scan functions (`next_set_idx()` and `next_clear_idx()`) for large bit-arrays with only few bits set or unset.

On the AMD64 architecture the corresponding CPU instructions are used [FXT: `bits/bitasm-amd64.h`]:

```c
static inline ulong asm_bts(ulong *f, ulong i)
{
    ulong ret;
    asm ( "btsq %2, %1 \n"
         "sbbq %0, %0"
    : "=r" (ret)
    : "m" (*f), "r" (i) );
    return ret;
}
```

If no specialized CPU instructions are available the following two macros are used:

```c
#define DIVMOD(n, d, bm) 
ulong d = n / BITS_PER_LONG; 
ulong bm = 1UL << (n % BITS_PER_LONG);
#define DIVMOD_TEST(n, d, bm) 
ulong d = n / BITS_PER_LONG; 
ulong bm = 1UL << (n % BITS_PER_LONG); 
ulong t = bm & f_[d];
```

An example:

```c
ulong test_set(ulong n) const // Test whether n-th bit is set and set it.
{
    ifdef BITS_USE_ASM
        return asm_bts(f_, n);
    else
        DIVMOD_TEST(n, d, bm);
        f_[d] |= bm;
        return t;
    endif
}
```

Performance is still good in that case as the modulo operation and division by `BITS_PER_LONG` (a power of two) are replaced with cheap (bit-and and shift) operations. On the machine described in appendix [B](#)

[fxtbook draft of 2008-August-17]
on page 923 both versions give practically identical performance.

How out of bounds are handled can be defined at the beginning of the header file:

```
#define CHECK 0 // define to disable check of out of bounds access
// #define CHECK 1 // define to handle out of bounds access
// #define CHECK 2 // define to fail with out of bounds access
```

### 4.7 Left-right array

The left-right array (or LR-array) keeps track of a range of indices $0, \ldots, n - 1$. Every index can have two states, free or set. The LR-array implements the following operations in time $O(n \log(n))$: marking the $k$-th free index as set; marking the $k$-th set index as free; for the $i$-th (absolute) index, finding how many indices of the same type (free or set) are left (or right) to it (including or excluding $i$).

The implementation is given as [FXT: class left_right_array in [ls/left-right-array.h]](fxtbook draft of 2008-August-17):

```cpp
    class left_right_array
    {
        ulong *fl_; // Free indices Left (including current element) in bsearch interval
        bool *tg_; // tags: tg[i]==true iff index at index i is free
        ulong n_; // total number of indices
        ulong f_; // number of free indices
    }
```

The arrays used have $n$ elements:

```cpp
    public:
    left_right_array(ulong n)
    {
        n_ = n;
        fl_ = new ulong[n_];
        tg_ = new bool[n_];
        free_all();
    }
```

The initialization routine `free_all()` of the array `fl[]` uses a variation of the binary search algorithm described in section 3.2 on page 123:

```cpp
    public:
    void free_all()
    // Mark all indices of as free.
    {
        f_ = n_;
        for (ulong j=0; j<n_; ++j) tg_[j] = true;
        init_rec(0, n_ - 1);
        fl_[n_ - 1] = 1;
    }
```

[fxtbook draft of 2008-August-17]
The crucial observation is that the set of all intervals occurring with binary search is fixed if the size of
the searched array is fixed. For any interval \([i_0, i_1]\) the element \(f_1[t]\) where \(t = \lfloor (i_0 + i_1)/2 \rfloor\) contains
the number of free positions in \([i_0, t]\). The following method returns the \(k\)-th free index:

```c
ulong get_free_idx(ulong k) const
    // Return the k-th ( 0 <= k < num_free() ) free index.
    // Return ~0UL if k is out of bounds.
    // Work is O(log(n)).
{
    if ( k >= num_free() ) return ~0UL;
    ulong i0 = 0, i1 = n_-1;
    while ( 1 )
    {
        ulong t = (i1+i0)/2;
        if ( (f1_[t] == k+1) && (tg_[t]) ) return t;
        if ( f1_[t] > k ) // left:
            {
                i1 = t;
            }
        else // right:
            {
                i0 = t+1; k-=f1_[t];
            }
    }
}
```

Usually one would have an extra array \(a[]\) where one actually sets the position returned above. If this
is done the data of the LR-array has to be modified accordingly, the following method does this:

```c
ulong get_free_idx_chg(ulong k)
    // Return the k-th ( 0 <= k < num_free() ) free index.
    // Return ~0UL if k is out of bounds.
    // Change the arrays and f1[] and tg[] reflecting
    // that index i will be set afterwards.
    // Work is O(log(n)).
{
    if ( k >= num_free() ) return ~0UL;
    --f_;
    ulong i0 = 0, i1 = n_-1;
    while ( 1 )
    {
        ulong t = (i1+i0)/2;
        if ( (f1_[t] == k+1) && (tg_[t]) )
            {
                --f1_[t];
                tg_[t] = false;
                return t;
            }
        if ( f1_[t] > k ) // left:
            {
                --f1_[t];
                i1 = t;
            }
        else // right:
            {
                i0 = t+1; k-=f1_[t];
            }
    }
}
```

For example, the following program sets alternatingly the first and last free position until no free position
is left [FXT: ds/left-right-array-demo.cc):

```c
ulong n = 9;
ulong *A = new ulong[n];
left_right_array LR(n);
LR.free_all();
// PRINT
for (ulong e=0; e<n; ++e)
{
```
Figure 4.7-A: Alternatingly setting the first and last free position in an LR-array. Asterisks denote free positions, indices \( i \) where \( \text{tg}[i] \) is true.

```c
ulong s = 0; // first free
if (0!=(e&1)) s = LR.num_free()-1; // last free
ulong idx2 = LR.get_free_idx_chg(s); // PRINT
```

Its output is shown in figure 4.7-A. The rate of the method `get_free_idx_chg()` is (very roughly) 2 million per second for large \( n \). The method to free the \( k \)-th set position is

```c
ulong get_set_idx_chg(ulong k)
    // Return the k-th (0 <= k < num_set()) set index.
    // Return ~0UL if k is out of bounds.
    // Change the arrays and fl[] and tg[] reflecting
    // that index \( i \) will be freed afterwards.
    // Work is \( O(\log(n)) \).
{
    if (k >= num_set()) return ~0UL;
    ++f_
;
    ulong i0 = 0, i1 = n_-1;
    while (1)
    {
        ulong t = (i1+i0)/2;
        // how many elements to the left are set:
        ulong slt = t-i0+1 - fl_[t];
        if ( (slt == k+1) && (tg_[t]==false) )
        {
            ++fl_[t];
            tg_[t] = true;
            return t;
        }
        if (slt > k) // left:
        {
            ++fl_[t];
            i1 = t;
        }
        else // right:
        {
            i0 = t+1; k-=slt;
        }
    }
    return ~0UL;
}
```

The method `nfls(i)` returns the number of free indices left of \( i \) (and excluding \( i \)):
4.8: Finite state machines

A finite state machine (FSM) (alternative terms are state engine, finite state automaton, state machine, and finite automaton) in its simplest form can be described as a program that has finite set of valid states and for each state a certain action is taken. In C-syntax:

```c
void state_engine(int state)
{
    while ( state != end )
    {
        // valid states are: st1 ... stn (and end)
```

Based on it are methods to determine the number of free/set indices to the left/right, including/excluding the given index. We omit the out-of-bounds clauses in the following:

```c
ulong num_FLE(ulong i) const
{ return ~0UL; } // out of bounds
ulong i0 = 0, i1 = n_-1;
ulong ns = i; // number of set element left to i (including i)
while ( i )
{
    if ( i==i1 ) break;
    ulong t = (i1+i0)/2;
    if ( i<=t ) // left:
    {
        i1 = t;
    }
    else // right:
    {
        ns -= fl_[t];
        i0 = t+1;
    }
}
return i-ns;
}
ulong num_FLI(ulong i) const
{ return num_FLE(i) + tg_[i]; }
ulong num_FRE(ulong i) const
{ return num_free() - num_FLI(i); }
ulong num_FRI(ulong i) const
{ return num_free() - num_FLE(i); }
ulong num_SLE(ulong i) const
{ return i - num_FLE(i); }
ulong num_SLI(ulong i) const
{ return i - num_FLE(i) + !tg_[i]; }
ulong num_SRE(ulong i) const
{ return num_set() - num_SLI(i); }
ulong num_SRI(ulong i) const
{ return num_set() - i + num_FLE(i); }
```

We use them for the fast conversion of permutations to and form inversion tables in section 10.3.2 on page 239.
switch (state) {
  case st1: state = func1(); break;
  case st2: state = func2(); break;
  case st3: state = func3(); break;
  [--snip--]
  case stn: state = funcn(); break;
  default: blue_smoke(); // invalid state
}
}

int main() {
  // initialize:
  int state = start;
  state_engine(state);
  return 0;
}

As an example we show a state automaton that transforms a linear coordinate \( t \) into the corresponding pair \( (x, y) \) of coordinates of Hilbert's space-filling curve.

Apart from two bits of internal state the FSM processes at each step two bits of input. The array \( htab[] \) serves as lookup table for the next state plus two bits of the result.

The function ([FXT: lin2hilbert() in bits/hilbert.cc]) implements a FSM as suggested in [33], item 115:

```c
void lin2hilbert(ulong t, ulong &x, ulong &y) {
  ulong xv = 0, yv = 0;
  ulong c01 = (0<<2); // (2<<2) for transposed output (swapped x, y)
  for (ulong i=0; i<(BITS_PER_LONG/2); ++i) {
    ulong abi = t >> (BITS_PER_LONG-2);
    t <<= 2;
    ulong st = htab[ (c01<<2) | abi ];
    c01 = st & 3;
    yv <<= 1;
    yv |= ((st>>2) & 1);
    xv <<= 1;
    xv |= (st>>3);
  }
  x = xv; y = yv;
}
```

<table>
<thead>
<tr>
<th>OLD C</th>
<th>NEW C</th>
<th>A B</th>
<th>X Y</th>
<th>C C</th>
<th>OLD C</th>
<th>NEW C</th>
<th>X Y</th>
<th>A B</th>
<th>C C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>0 1</td>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 1</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>1 1</td>
<td>1 0</td>
<td>1 0 0 1</td>
<td>1 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.8-A: The original table from [33] for the finite state machine for the 2-dimensional Hilbert curve (left). All sixteen 4-bit words appear in both the 'OLD' and the 'NEW' column. So the algorithm is invertible. Swap the columns and sort numerically to obtain the two columns at the right, the table for the inverse function.
4.9: Emulation of coroutines

The table used is defined (see figure 4.8-A) as

```c
static const ulong htab[] = {
#define HT(xi,yi,c0,c1) ((xi<<3)+(yi<<2)+(c0<<1)+(c1))

// index == HT(c0,c1,ai,bi)
HT( 0, 0, 1, 0 ),
HT( 0, 1, 0, 0 ),
HT( 1, 1, 0, 0 ),
HT( 1, 0, 0, 1 ),
[--snip--]
HT( 0, 0, 1, 1 ),
HT( 0, 1, 1, 0 )
};
```

As indicated in the code the table maps every four bits c0,c1,ai,bi to four bits xi,yi,c0,c1. The table for the inverse function (again, see figure 4.8-A) is

```c
static const ulong ihtab[] = {
#define IHT(ai,bi,c0,c1) ((ai<<3)+(bi<<2)+(c0<<1)+(c1))

// index == HT(c0,c1,xi,yi)
IHT( 0, 0, 1, 0 ),
IHT( 0, 1, 0, 0 ),
IHT( 1, 1, 0, 1 ),
IHT( 1, 0, 0, 0 ),
[--snip--]
IHT( 0, 1, 1, 1 ),
IHT( 0, 0, 0, 1 )
};
```

The words have to be processed backwards:

```c
ulong hilbert2lin(ulong x, ulong y)
{ // Transform Hilbert x and y to linear coordinate t
  ulong t = 0;
  ulong c01 = 0;
  for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
  {
    t <<= 2;
    ulong xi = x >> (BITS_PER_LONG/2-1);
    xi &= 1;
    ulong yi = y >> (BITS_PER_LONG/2-1);
    yi &= 1;
    ulong xyi = (xi<<1) | yi;
    x <<= 1;
    y <<= 1;
    ulong st = ihtab[ (c01<<2) | xyi ];
    c01 = st & 3;
    t |= (st>>2);
  }
  return t;
}
```

A method to compute the direction (left, right, up or down) at the n-th move of the Hilbert curve is given in section 1.20.1 on page 57. The computation of a function whose series coefficients are ±1 and ±i according to the Hilbert curve is described in section 36.9 on page 744.

### 4.9 Emulation of coroutines

Sometimes it is possible to find an recursive algorithm for solving some problem that is not easily solved iteratively. However the recursive implementations might produce the results in midst of its calling graph. When a utility class providing the results one by one with some `next` call is required there is an apparent problem (with most programming languages): there is only one stack available for function calls. We do not have ‘offline’ functions or coroutines.

As an example consider the following recursive code (given by Glenn Rhoads):

```c
[fxtbook draft of 2008-August-17]
Chapter 4: Data structures

```c
int n = 4;
int a[n+1], q[2*n+1];

void paren(long i, long s)
{
    long k, t;
    if ( i<n )
    {
        for (k=0; k<=i-s; ++k)
        {
            a[i-1] = k;
            t = s + a[i-1];
            q[t + i] = '(';
            paren(i + 1, t); // recursion
            q[t + i] = ')';
        }
    }
    else
    {
        a[i-1] = n - s;
        Visit(); // next set of parens available
    }
}

int main()
{
    paren(0, 0);
    return 0;
}
```

It generates following output (the different ways to group four pairs of parenthesis):

```
(((())))
((()()))
((())())
((()))()
(()(()))
(()()())
(()())()
(())(())
(())()()
()((()))
```

A reasonable way to create coroutines is to rewrite the function as a state engine and use a class [FXT: class coroutine in ds/coroutine.h] that provides two stacks, one for local variables (including arguments) and one for the state of the function:

```c
template<typename Type>
class coroutine
{
public:
    ulong *t_; // State stack
    ulong tp_; // State stack Pointer
    Type *d_; // Data stack
    ulong dp_; // Data stack Pointer
    ulong n_; // size of stacks
}

coroutine(ulong maxdepth)
{
    n_ = maxdepth;
    t_ = new ulong[n_];
    d_ = new Type[n_];
    init();
}
~coroutine()
{
    delete [] d_;
    delete [] t_;}

void init() { dp_=0; tp_=0; }
```
4.9: Emulation of coroutines

// --- state stack:
void stpush(ulong x) { chk_tp(); t_[tp_] = x; ++tp_; }
ulong stpeek() const { chk_tp(1); return t_[tp_-1]; }
void stpoke(ulong x) { chk_tp(1); t_[tp_-1] = x; }
void sttop(ulong ct=1) { tp_=stc; }
void stnext() { chk_tp(1); ++t_[tp_-1]; }
void stnext(ulong x) { chk_tp(1); t_[tp_-1] = x; }
bool more() const { return (tp_!=0); }

// --- stack for variables and args:
void push(Type x) { chk_dp(); d_[dp_] = x; ++dp_; }
Type &peek() { chk_dp(1); return d_[dp_-1]; }
void pop(ulong ct=1) { dp_=stc; }
private:
coroutine & operator = (const coroutine &); // forbidden

The functions chk_tp() and chk_dp() either check for stack overflow or do nothing, as defined by
#define CHECK_STACKS // define to detect stack overflow

Rewriting the function in question (as part of a utility class, given in [FXT: ds/coroutine-paren-demo.cc])
only requires the understanding of the language, not of the algorithm. The process is straightforward
but needs a bit of concentration:

int
paren::next_recursion()
{
    var &V = cr_->peek();
    int &i=V.i, &s=V.s; // args
    int &k=V.k, &t=V.t; // locals

    case 0:
        break; // shortcut: nothing to do
    case 1:
        // loop end ?
        if ( k>i-s ) { break; } // shortcut: nothing to do at end
        cr_->stnext();
    case 2: // start of loop body
        long ii = i-1;
        x[ii] = k;
        t = s + k;
        str[t+i] = '('; // OPEN_CHAR;
        cr_->stnext();
        CORO_CALL( i+1, t, 0, 0 );
        goto redo;
    case 3:
        str[t+i] = ')'; // CLOSE_CHAR;
        ++k;
        // loop end ?
        if ( k>i-s ) { break; } // shortcut: nothing to do at end
        cr_->stpoke(2); goto loop; // shortcut: back to loop body
    default: ;
    }

CORO_RETURN();

[fxtbook draft of 2008-August-17]
Chapter 4: Data structures

```c
if ( cr_->more() ) goto redo;
return 0;
}
```

The following `#define` was used:

```c
// args=(i, s) (k, t)=locals
#define CORO_CALL(vi, vs, vk, vt) \
{ vars V_; V_.i=vi; V_.s=vs; V_.k=vk; V_.t=vt; \ 
  cr_->push(V_); cr_->stpush(ST_INIT); }
```

The constructor pushes the needed variables and parameters on the data stack and the initial state on the state stack:

```c
cor::cor(int nn)
{
    n = (nn>0 ? nn : 1);
    x = new int[n+1];
    ++x;
    str = new char[2*n+1];
    int i = 0;
    for ( ; i<n ; ++i) str[i] = '('; // OPEN_CHAR;
    for ( ; i<2*n; ++i) str[i] = ')'; // CLOSE_CHAR;
    str[2*n] = 0;
    cr_ = new coroutine<vars>(n+1);
    // i, s, k, t
    CORO_CALL( 0, 0, 0, 0 );
    idx = 0;
    q = next_recursion();
}
```

The method `next()` of the `cor` class lets the offline function advance until the next result is available:

```c
int cor::next()
{
    if ( 0==q ) return 0;
    else
    {
        q = next_recursion();
        return ( q ? ++idx : 0 );
    }
}
```

Performance-wise the `coroutine`-rewritten functions are close to the original: state engines are fast and the stack operations are cheap.

The shown method can also be applied when the recursive algorithm consists of more than one function by merging the functions into one state engine.

Further, investigating the contents of the data stack can be of help in the search of a iterative solution.

The code of the converted functions is admittedly ugly but without a generic support of coroutines it seems hard to du much better. One could also use threads for the emulation of coroutines. This approach, however, adds portability problems.

An iterative algorithm for the generation of valid pairings or parenthesis is given in chapter 13 on page 317.
Part II

Combinatorial generation
Chapter 5

Conventions and considerations

We give algorithms for the generation of all combinatorial objects of certain types such as combinations, compositions, subsets, permutations, integer partitions, set partitions, restricted growth strings and necklaces. Finally, we give some constructions for Hadamard and conference matrices. Several (more esoteric) combinatorial objects that are found via searching in directed graphs are presented in chapter 19.

The routines are useful in situations where an exhaustive search over all configurations of a certain kind is needed. Combinatorial algorithms are also fundamental with many programming problems and finally they can simply be fun!

5.1 About representations and orders

For a set of $n$ elements we will take either $\{0, 1, \ldots, n - 1\}$ or $\{1, 2, \ldots, n\}$. Our convention for the set notation is to start with the smallest element. Often there is more than one useful way to represent a combinatorial object. For example the subset $\{1, 4, 6\}$ of the set $\{0, 1, 2, 3, 4, 5, 6\}$ can also be written as a delta set $[0100101]$. Some sources use the term bit string. We often write dots for zeros for readability: $[.1..1.1]$. Note that in the delta set we put the first element to the left side (array notation), this is in contrast to the usual way to print binary numbers, where the least significant bit (bit number zero) is shown on the right side.

For most objects we will give an algorithm for generation in lexicographic (or simply lex) order. In lexicographic order a string $X = [x_0, x_1, \ldots]$ precedes another string $Y = [y_0, y_1, \ldots]$ if for the smallest index $k$ where the strings differ we have $x_k < y_k$. Further, the string $X$ precedes $X.W$ (the concatenation of $X$ with $W$) for any nonempty string $W$. The ordering obtained by reversing the lexicographic order is sometimes called relex order. The co-lexicographic (or simply colex) order is obtained by the lex order of the reversed strings. The order sometimes depends on the representation that is used, for an example see figure 8.1-A on page 201.

In a minimal-change order the amount of change between successive objects is the least possible. Such an order is also called a (combinatorial) Gray code. There is in general more than one such order. Often one can impose even stricter conditions, like that (with permutations) the changes are between adjacent positions. The corresponding order is a strong minimal-change order. A very readable survey of Gray codes in given in [267], see also [235].
Chapter 5: Conventions and considerations

5.2 Ranking, unranking, and counting

For a particular ordering of combinatorial objects (say, lexicographic order for permutations) one can ask which position in the list a given object has. An algorithm for finding the position is called a ranking algorithm. An method to determine the object, given its position, is called an unranking algorithm.

Given both ranking and unranking methods one can compute the successor of a given object by computing its rank \( r \) and unranking \( r + 1 \). While this method is usually slow the idea can be used to find more efficient algorithms for computing the successor. In addition the idea often suggests interesting orderings for combinatorial objects.

We sometimes give ranking or unranking methods for numbers in special forms such as factorial representations for permutations. Ranking and unranking methods are implicit in generation algorithms based on mixed radix counting given in section 10.8 on page 259.

A simple but surprisingly powerful way for the discovery of isomorphisms (one-to-one correspondences) between combinatorial objects is counting them. If the sequences of numbers of two kinds of objects are identical chances are good to find a conversion routine between the corresponding objects. For example, there are \( 2^n \) permutations of \( n \) elements such that no element lies more than one position to the right of its original position. From this observation an algorithm for generating these permutations via binary counting can be found, see section 10.15.2 on page 288.

The representation of combinatorial objects as restricted growth strings (as shown in section 13.2 on page 319) follows from the same idea. The resulting generation methods can be very fast and flexible.

5.3 Characteristics of the algorithms

In almost all cases we produce the combinatorial objects on by one. Let \( n \) be the size of the object. The successor (with respect to the specified order) is computed from the object itself and additional data whose size is less than a constant multiple of \( n \).

Let \( B \) be the total number of combinatorial objects under consideration. Sometimes the cost of a successor computation is proportional to \( n \). Then the total cost for generating all objects is proportional to \( n \cdot B \).

If the successor computation takes a fixed number of operations (independent of the object size) then we say the algorithm is \( O(1) \). If so, there can be no loop in the implementation, we say the algorithm is loopless. The total cost for all objects then is \( c \cdot B \) for some \( c \) independent of the object size. A loopless algorithm can only exist if the amount of change between successive object is bounded by a constant that does not depend on the object size. Natural candidates for loopless algorithms are Gray codes.

In many cases the cost of computing all objects is also \( c \cdot B \) while the computation of the successor does involve a loop. As an example consider incrementing in binary using arrays: in half of the cases just the lowest bit changes, for half of the remaining cases just two bits change, and so on. The total cost is \( B \cdot (1 + \frac{1}{2}(1 + \frac{1}{2}(\cdots))) = 2 \cdot B \), independent of the number of bits used. So the total cost is as in the loopless case while the successor computation can be expensive in some cases. Algorithms with this characteristic are called constant amortized time (or CAT). Often CAT algorithms are faster than loopless algorithms, typically when their structure is more simple.

5.4 Optimization techniques

Let \( x \) be an array of \( n \) elements. The loop

```c
ulong k = 0;
while ( (k<n) && (x[k]!=0) ) ++k; // find first zero
```

[fxtbook draft of 2008-August-17]
5.5: The implementations, demo-programs, and timings

can be replaced by

```c
ulong k = 0;
while ( x[k]!=0 ) ++k; // find first zero
```

if a single sentinel element x[n]=0 is appended to the end of the array. The latter version will often be faster as less branches occur.

The test for equality as in

```c
ulong k = 0;
while ( k!=n ) { /*...*/ ++k; }
```

is more expensive than the test for equality with zero as in

```c
ulong k = n;
while ( --k!=0 ) { /*...*/ }
```

Therefore the latter version should be used when applicable.

To reduce the number of branches, replace the two tests

```c
if ( (x<0) || (x>m) ) { /*...*/ }
```

by the following single test where unsigned integers are used:

```c
if ( x>m ) { /*...*/ }
```

Use a do-while construct instead of a while-do loop whenever possible because the latter also tests the loop condition at entry. Even when the do-while version causes some unnecessary work the gain from the avoided branch may outweigh it. Note that in the C language the for-loop also tests the condition at loop entry.

When computing the next object there may be special cases where the update is trivial. If the percentage of these ‘easy cases’ is not too small an extra branch in the update routine should be created. The performance gain is very visible in most cases (section 10.4 on page 246) and can be drastic (section 10.5 on page 249).

Recursive routines can be quite elegant and versatile, see, for example, section 6.3 on page 181 and section 11.2.1 on page 293. However, expect only about half the speed of a good iterative implementation of the same algorithm. The ideas given in section 4.9 on page 165 can be used to obtain iterative versions from a recursive implementation.

Address generation can be simpler if arrays are used instead of pointers. This technique is useful for many permutation generators, see chapter 10 on page 233. Change the pointer declarations to array declarations in the corresponding class as follows:

```c
//ulong *p_; // permutation data (pointer version)
ulong p_ [32]; // permutation data (array version)
```

Here we assume that nobody would attempt to compute all permutations of 31 or more elements (31! ≈ 8.22·1033, taking about 1.3·1018 years to finish). To use arrays uncomment (in the corresponding header files) a line like

```c
#define PERM_REV2_FIXARRAYS // use arrays instead of pointers (speedup)
```

This will also disable the statements to allocate and free memory with the pointers. Whether the use of arrays tends to give a speedup is noted in the comment. The performance gain can be spectacular, see section 7.1 on page 193.

5.5 The implementations, demo-programs, and timings

Most combinatorial generators are implemented as C++ classes. The first object in the given order is created by the method `first()`. The method to compute the successor is usually `next()`. If a method for the computation of the predecessor is given then it is called `prev()`, and the method `last()` computes the last element in the list.
The current combinatorial object can be accessed through the method `data()`. In order to make all data of a class accessible the data is declared `public`. Thereby the need for various `get_something()` methods is avoided. To minimize the danger of accidental modification of class data the variable names end with an underscore. For example, the class for the generation of combinations in lexicographic order starts as

```cpp
class combination_lex
{
public:
    ulong *x_; // combination: k elements 0<=x[j]<k in increasing order
    ulong n_, k_; // Combination (n choose k)
}
```

The methods for the user of the class are `public`, the internal methods (which can leave the data in an inconsistent state) are declared `private`.

Timings for the routines are given with most demo-programs. For example, the timings for the generation of subsets in minimal-change order (as delta sets, implemented in [FXT: `class subset_gray_delta` in `comb/subset-gray-delta.h`]) are given near the end of [FXT: `comb/subset-gray-delta-demo.cc`], together with the parameters used:

```
Timing:
    time ./bin 30
    arg 1: 30 == n [Size of the set] default=5
    arg 2: 0 == rq [Whether to generate subsets in reversed order] default=0
    ./bin 30 8.07s user 0.04s system 99% cpu 8.120 total
    => 2^30/8.07 == 133,053,509 per second
    // with SUBSET_GRAY_DELTA_MAX_ARRAY_LEN defined:
    ./bin 30 7.83s user 0.04s system 99% cpu 7.875 total
    => 2^30/7.83 == 137,131,778 per second
```

For your own measurements simply uncomment the line
`
//define TIMING // uncomment to disable printing
`

near the top of the demo-program. In the text the rate of generation for a certain object is occasionally given as 123 M/s, meaning that 123 million objects are generated per second.

When a generator routine is used in an application one must do the benchmarking with the application. Choosing the optimal ordering and type of representation (for example, delta sets versus sets) for the given task is crucial for good performance. Further optimization will very likely involve the surrounding code rather than the generator alone.
Chapter 6
Combinations

The number of ways to choose \( k \) elements from a set of \( n \) elements equals the binomial coefficient \( \binom{n}{k} \):

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!} = \prod_{j=1}^{k} \frac{n-j+1}{j}
\]  

Equivalently, a set of \( n \) elements has \( \binom{n}{k} \) subsets of exactly \( k \) elements. These subsets are called the \( k \)-subsets (where \( k \) is fixed) or \( k \)-combinations of an \( n \)-set (a set with \( n \) elements).

To avoid overflow during the computation of the binomial coefficient, use the form

\[
\binom{n}{k} = \frac{(n-k+1)^k}{1^k} = \frac{n-k+1}{1} \cdot \frac{n-k+2}{2} \cdot \frac{n-k+3}{3} \cdots \frac{n}{k}
\]  

An implementation is given in [FXT: aux0/binomial.h]:

```c
inline ulong binomial(ulong n, ulong k)
{
    if ( k>n ) return 0;
    if ( (k==0) || (k==n) ) return 1;
    if ( 2*k > n ) k = n-k; // use symmetry

    ulong b = n - k + 1;
    ulong f = b;
    for (ulong j=2; j<=k; ++j)
    {
        f += j;
        b *= f;
    }
    return b;
}
```

Figure 6.0-A: The binomial coefficients \( \binom{n}{k} \) for \( 0 \leq n, k \leq 15 \).
The table of the first binomial coefficients is shown in figure 6.0-A. This table is called Pascal’s triangle, it was generated with the program \[\text{FXT: comb/binomial-demo.cc}\]. Observe that

\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}
\]  

That is, each entry is the sum of its upper and left upper neighbor. The generating function for the \(k\)-combinations of an \(n\)-set is

\[
(1 + x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k
\]

6.1 Lexicographic and co-lexicographic order

<table>
<thead>
<tr>
<th>lexicographic</th>
<th>co-lexicographic</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>delta set</td>
</tr>
<tr>
<td>1: { 0, 1, 2 }</td>
<td>111...</td>
</tr>
<tr>
<td>2: { 0, 1, 3 }</td>
<td>111.1...</td>
</tr>
<tr>
<td>3: { 0, 1, 4 }</td>
<td>111.1...</td>
</tr>
<tr>
<td>4: { 0, 1, 5 }</td>
<td>111.1...</td>
</tr>
<tr>
<td>5: { 0, 2, 3 }</td>
<td>1.11.1...</td>
</tr>
<tr>
<td>6: { 0, 2, 4 }</td>
<td>1.11.1...</td>
</tr>
<tr>
<td>7: { 0, 2, 5 }</td>
<td>111.1...</td>
</tr>
<tr>
<td>8: { 0, 3, 4 }</td>
<td>111.1...</td>
</tr>
<tr>
<td>9: { 0, 3, 5 }</td>
<td>111.1...</td>
</tr>
<tr>
<td>10: { 0, 4, 5 }</td>
<td>1...11</td>
</tr>
<tr>
<td>11: { 1, 2, 3 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>12: { 1, 2, 4 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>13: { 1, 2, 5 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>14: { 1, 3, 4 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>15: { 1, 3, 5 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>16: { 1, 4, 5 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>17: { 0, 3, 4 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>18: { 1, 2, 4 }</td>
<td>.111.1</td>
</tr>
<tr>
<td>19: { 0, 2, 5 }</td>
<td>111...</td>
</tr>
<tr>
<td>20: { 0, 2, 5 }</td>
<td>111...</td>
</tr>
</tbody>
</table>

Figure 6.1-A: All combinations \(\binom{6}{3}\) in lexicographic order (left), and co-lexicographic order (right).

The combinations of three elements out of six in lexicographic (or simply \text{lex}) order are shown in figure 6.1-A (left). The sequence is such that the sets are ordered lexicographically. Note that for the delta sets the element zero is printed first whereas with binary words (section 1.25 on page 67) the least significant bit (bit zero) is printed last. The sequence for \text{co-lexicographic} (or \text{colex}) order is such that the sets, when written reversed, are ordered lexicographically.

6.1.1 Lexicographic order

The following implementation generates the combinations in lexicographic order as sets \[\text{FXT: class combination_lex in comb/combination-lex.h}\]:

```cpp
class combination_lex
{
    ulong *x_; // combination: k elements 0<=x[j]<k in increasing order
}
```

[fxtbook draft of 2008-August-17]
ulong n_, k_; // Combination (n choose k)

public:

combination_lex(ulong n, ulong k)
{
    n_ = n; k_ = k;
    x_ = new ulong[k_];
    first();
}

~combination_lex() { delete [] x_; }

void first()
{
    for (ulong k=0; k<k_; ++k) x_[k] = k;
}

void last()
{
    for (ulong i=0; i<k_; ++i) x_[i] = n_ - k_ + i;
}

Computation of the successor and predecessor:

ulong next()
// Return smallest position that changed, return k with last combination
{
    if ( x_[0] == n_ - k_ ) // current combination is the last
    { first(); return k_; }
    ulong j = k_ - 1;
    if ( x_[j] < (n_-1) ) { ++x_[j]; return j; }

    // find highest falling edge:
    while ( 1 == (x_[j] - x_[j-1]) ) { --j; }

    // move lowest element of highest block up:
    ulong ret = j-1;
    ulong z = ++x_[j-1];

    // ... and attach rest of block:
    while ( j < k_ ) { x_[j] = ++z; ++j; }
    return ret;
}

ulong prev()
// Return smallest position that changed, return k with last combination
{
    if ( x_[k_-1] == k_-1 ) // current combination is the first
    { last(); return k_; }
    ulong j = k_ - 1;

    // find highest falling edge:
    while ( 1 == (x_[j] - x_[j-1]) ) { --j; }

    // move down edge element
    --x_[j];

    // ... and move rest of block to high end:
    while ( ++j < k_ ) x_[j] = n_ - k_ + j;
    return ret;
}

The listing in figure 6.1-A was created with the program [FXT: comb/combination-lex-demo.cc]. The routine generates the combinations $\binom{32}{20}$ at a rate of about 95 million per second. The combinations $\binom{32}{12}$ are generated at a rate of 160 million per second.
### 6.1.2 Co-lexicographic order

The combinations of three elements out of six in co-lexicographic (or `colex`) order are shown in figure 6.1-A (right). Algorithms to compute the successor and predecessor are implemented in [FXT: class `combination_colex` in `comb/combination-colex.h`]:

```cpp
class combination_colex
{
    public:
        ulong *x_; // combination: k elements 0<=x[j]<k in increasing order
        ulong n_, k_; // Combination (n choose k)

    combination_colex(ulong n, ulong k)
    {
        n_ = n; k_ = k;
        x_ = new ulong[k_+1];
        x_[k_] = n_ + 2; // sentinel
        first();
    }

    ulong next()
    // Return greatest position that changed, return k with last combination
    {
        if ( x_[0] == n_ - k_ ) // current combination is the last
            { first(); return k_; } 
        ulong j = 0;
        // until lowest rising edge: attach block at low end
        while ( 1 == (x_[j+1] - x_[j]) ) { x_[j] = j; ++j; } // can touch sentinel
        ++x_[j]; // move edge element up
        return j;
    }

    ulong prev()
    // Return greatest position that changed, return k with last combination
    {
        if ( x_[k_-1] == k_-1 ) // current combination is the first
            { last(); return k_; } 
        // find lowest falling edge:
        ulong j = 0;
        while ( j == x_[j] ) ++j; // can touch sentinel
        --x_[j]; // move edge element down
        ulong ret = j;
        // attach rest of low block:
        while ( 0!=j-- ) x_[j] = x_[j+1] - 1;
        return ret;
    }
}
```

The listing in figure 6.1-A was created with the program [FXT: `comb/combination-colex-demo.cc`]. The combinations are generated \( \binom{32}{20} \) at a rate of about 140 million objects per second, the combinations \( \binom{32}{12} \) are generated at a rate of 190 million objects per second.

As a toy application of the combinations in co-lexicographic order we compute the products of \( k \) of the \( n \) smallest primes. We maintain an array of \( k \) products shown at the right of figure 6.1-B. When the return value of the method `next()` is \( j \) then \( j + 1 \) elements have to be updated from right to left [FXT: `comb/kproducts-colex-demo.cc`]:

```cpp
combination_colex C(n, k);
const ulong *c = C.data(); // combinations as sets
ulong *tf = new ulong[n]; // table of Factors (primes)
// fill in small primes:
for (ulong j=0,f=2; j<n; ++j) { tf[j] = f; f=next_small_prime(f+1); }
ulong *tp = new ulong[k+1]; // table of Products
```
6.1: Lexicographic and co-lexicographic order

<table>
<thead>
<tr>
<th>combination</th>
<th>j</th>
<th>delta-set</th>
<th>products</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: { 0, 1, 2 }</td>
<td>2</td>
<td>111....</td>
<td>[ 30 15  5  1 ]</td>
</tr>
<tr>
<td>2: { 0, 1, 3 }</td>
<td>2</td>
<td>11.1...</td>
<td>[ 42 21  7  1 ]</td>
</tr>
<tr>
<td>3: { 0, 2, 3 }</td>
<td>1</td>
<td>1.11...</td>
<td>[ 70 35  7  1 ]</td>
</tr>
<tr>
<td>4: { 1, 2, 3 }</td>
<td>0</td>
<td>....11..</td>
<td>[105 35  7  1 ]</td>
</tr>
<tr>
<td>5: { 0, 1, 4 }</td>
<td>2</td>
<td>11...1..</td>
<td>[ 66 33 11  1 ]</td>
</tr>
<tr>
<td>6: { 0, 2, 4 }</td>
<td>1</td>
<td>1.11...</td>
<td>[110 55 11  1 ]</td>
</tr>
<tr>
<td>7: { 1, 2, 4 }</td>
<td>0</td>
<td>.111...</td>
<td>[165 55 11  1 ]</td>
</tr>
<tr>
<td>8: { 0, 3, 4 }</td>
<td>1</td>
<td>1..11..</td>
<td>[154 77 11  1 ]</td>
</tr>
<tr>
<td>9: { 1, 3, 4 }</td>
<td>0</td>
<td>.11.1..</td>
<td>[231 77 11  1 ]</td>
</tr>
<tr>
<td>10: { 2, 3, 4 }</td>
<td>0</td>
<td>...111..</td>
<td>[165 77 11  1 ]</td>
</tr>
<tr>
<td>11: { 0, 1, 5 }</td>
<td>2</td>
<td>11....1</td>
<td>[ 78 39 13  1 ]</td>
</tr>
<tr>
<td>12: { 0, 2, 5 }</td>
<td>1</td>
<td>1.1..1.</td>
<td>[130 65 13  1 ]</td>
</tr>
<tr>
<td>13: { 1, 2, 5 }</td>
<td>0</td>
<td>.11..1.</td>
<td>[195 65 13  1 ]</td>
</tr>
<tr>
<td>14: { 0, 3, 5 }</td>
<td>1</td>
<td>1..1.1.</td>
<td>[ 39 91 13  1 ]</td>
</tr>
<tr>
<td>15: { 1, 3, 5 }</td>
<td>0</td>
<td>.1.1.1.</td>
<td>[273 91 13  1 ]</td>
</tr>
<tr>
<td>16: { 2, 3, 5 }</td>
<td>0</td>
<td>..11.1.</td>
<td>[ 55 91 13  1 ]</td>
</tr>
<tr>
<td>17: { 0, 4, 5 }</td>
<td>1</td>
<td>1...1.</td>
<td>[286 143 17  1 ]</td>
</tr>
<tr>
<td>18: { 1, 4, 5 }</td>
<td>0</td>
<td>.1...1.</td>
<td>[249 143 17  1 ]</td>
</tr>
<tr>
<td>19: { 2, 4, 5 }</td>
<td>0</td>
<td>..1.1.1</td>
<td>[ 715 143 17  1 ]</td>
</tr>
<tr>
<td>20: { 3, 4, 5 }</td>
<td>0</td>
<td>...11.1</td>
<td>[1001 143 17  1 ]</td>
</tr>
<tr>
<td>21: { 0, 1, 6 }</td>
<td>2</td>
<td>11.....</td>
<td>[102 51 17  1 ]</td>
</tr>
<tr>
<td>22: { 0, 2, 6 }</td>
<td>1</td>
<td>1.1....</td>
<td>[170 85 17  1 ]</td>
</tr>
<tr>
<td>23: { 1, 2, 6 }</td>
<td>0</td>
<td>.1....1</td>
<td>[255 85 17  1 ]</td>
</tr>
<tr>
<td>24: { 0, 3, 6 }</td>
<td>1</td>
<td>1.....1</td>
<td>[238 119 17  1 ]</td>
</tr>
<tr>
<td>25: { 1, 3, 6 }</td>
<td>0</td>
<td>.1.....1</td>
<td>[256 119 17  1 ]</td>
</tr>
<tr>
<td>26: { 2, 3, 6 }</td>
<td>0</td>
<td>..1.....1</td>
<td>[ 595 119 17  1 ]</td>
</tr>
<tr>
<td>27: { 0, 4, 6 }</td>
<td>1</td>
<td>1.....1.</td>
<td>[ 374 187 17  1 ]</td>
</tr>
<tr>
<td>28: { 1, 4, 6 }</td>
<td>0</td>
<td>.1.....1.</td>
<td>[ 561 187 17  1 ]</td>
</tr>
<tr>
<td>29: { 2, 4, 6 }</td>
<td>0</td>
<td>..1.....1.</td>
<td>[ 935 187 17  1 ]</td>
</tr>
<tr>
<td>30: { 3, 4, 6 }</td>
<td>0</td>
<td>...11.1.</td>
<td>[1309 187 17  1 ]</td>
</tr>
<tr>
<td>31: { 0, 5, 6 }</td>
<td>1</td>
<td>1....11</td>
<td>[ 442 221 17  1 ]</td>
</tr>
<tr>
<td>32: { 1, 5, 6 }</td>
<td>0</td>
<td>.1....11</td>
<td>[ 663 221 17  1 ]</td>
</tr>
<tr>
<td>33: { 2, 5, 6 }</td>
<td>0</td>
<td>..1....11</td>
<td>[1105 221 17  1 ]</td>
</tr>
<tr>
<td>34: { 3, 5, 6 }</td>
<td>0</td>
<td>...1...11</td>
<td>[1547 221 17  1 ]</td>
</tr>
<tr>
<td>35: { 4, 5, 6 }</td>
<td>0</td>
<td>..1.11.1</td>
<td>[2431 221 17  1 ]</td>
</tr>
</tbody>
</table>

Figure 6.1-B: All products of \( k = 3 \) of the \( n = 7 \) smallest primes (2, 3, 5, \ldots, 17). The products are the leftmost elements of the array on the right hand side.

```c
9  tp[k] = 1; // one appended (sentinel)
10  ulong j = k-1;
11  do
12    { // update products from right:
13      ulong x = tp[j+1];
14      { ulong i = j;
15        do
16          { ulong f = tf[ c[i] ];
17            x *= f;
18            tp[i] = x;
19          } // here: final product is x == tp[0]
20        } // visit the product x here
21        j = C.next();
22    } while ( j < k );
```

The leftmost element of this array is the desired product. A sentinel element at the end of the array is used to avoid an extra branch with the loop variable. With lexicographic order the update would go from left to right.
6.2 Order by prefix shifts (cool-lex)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>2</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>3</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>4</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>5</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>6</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>7</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>8</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>9</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
<tr>
<td>10</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
</tbody>
</table>

Figure 6.2-A: Combinations \( \binom{5}{k} \), for \( k = 1, 2, 3, 4 \) in an ordering generated by prefix shifts.

Figure 6.2-B: Combinations \( \binom{9}{3} \) via prefix shifts.

An algorithm for generating combinations by prefix shifts which is given in [231]. The ordering is called cool-lex in the paper. Figure 6.2-A shows some orders for \( \binom{5}{k} \), figure 6.2-B shows the combinations \( \binom{9}{3} \). The listings were created with the program [FXT: class combination_pref in comb/combination-pref-demo.cc] which uses the implementation in [FXT: class combination pref in comb/combination-pref.h]:

```cpp
class combination_pref
{

public:
  ulong *b_; // data as delta set
  ulong s_, t_, n_; // combination (n choose k) where n=s+t, k=t.

private:
  ulong x, y; // aux

public:
  combination_pref(ulong n, ulong k)
  // Must have: n>=2, k>=1 (i.e. s!=0 and t!=0)
  {
    s_ = n - k;
    t_ = k;
    n_ = s_ + t_;
    b_ = new ulong[n_];
    first();

    --snip--

    void first()
    {
      for (ulong j=0; j<n_; ++j) b_[j] = 0;
      for (ulong j=0; j<t_; ++j) b_[j] = 1;
      x = 0; y = 0;
    }

    bool next()
    {
      if ( x==0 ) { x=1; b_[t_]=1; b_[0]=0; return true; }
      else
      {
        if ( x>=n_-1 ) return false;
        else
        { b_[x] = 0; ++x; b_[y] = 1; ++y; // X(s,t)
          if ( b_[x]==0 )
          {
            b_[x] = 1; b_[0] = 0; // Y(s,t)
          }
        }
      }
    }

    --snip--
};
```

[fxtbook draft of 2008-August-17]
The combinations are generated \( \binom{32}{20} \) at a rate of about 95 million objects per second, the combinations \( \binom{32}{12} \) are generated at a rate of 85 million objects per second.

6.3 Minimal-change order

The combinations of three elements out of six in a minimal-change order (a Gray code) are shown in figure 6.3-A (left). With each transition exactly one element changes its position. We use a recursion for the list \( C(n, k) \) of combinations \( \binom{n}{k} \) (notation as in relation 12.0-1 on page 299):

\[
C(n, k) = \frac{C(n-1, k)}{[0 \cdot C(n-1, k)]} = \frac{1 \cdot C(n-1, k) - 1}{1 \cdot C(n-1, k - 1)}
\]  

The first equality is for the set representation, the second for the delta-set representation. An implementation is given in [FXT: comb/combination-gray-rec-demo.cc]:

```
ulong *x; // elements in combination at x[1] ... x[k]
void comb_gray(ulong n, ulong k, bool z)
{
    if (k==n)
        for (ulong j=1; j<=k; ++j) x[j] = j;
        visit();
        return;
    if (z) // forward:
        comb_gray(n-1, k, !z);
        if (k>0) { x[k] = n; comb_gray(n-1, k-1, !z); }
    else // backward:
```
The recursion can be partly unfolded as follows

\[ C(n, k) = \begin{bmatrix} C(n-2, k) \\ (n-1) \cdot C^R(n-2, k-1) \\ (n) \cdot C^R(n-1, k-1) \end{bmatrix} = \begin{bmatrix} 0, 0 \cdot C(n-2, k) \\ 0, 1 \cdot C^R(n-2, k-1) \\ 1, C^R(n-1, k-1) \end{bmatrix} \] (6.3-2)

A recursion for the complemented order is

\[ C'(n, k) = \begin{bmatrix} (n) \cdot C'(n-1, k-1) \\ C'^R(n-1, k) \end{bmatrix} = \begin{bmatrix} 1, C'(n-1, k-1) \\ 0, C'^R(n-1, k) \end{bmatrix} \] (6.3-3)

A very efficient (revolving door) algorithm that generates the sets for the Gray code is given in [217]. An implementation following [174] is [FXT: class combination_revdoor in comb/combination-revdoor.h]. Usage of the class is shown in [FXT: comb/combination-revdoor-demo.cc]. The routine generates the combinations \((32, 20)\) at a rate of about 115 M/s, the combinations \((32, 12)\) are generated at a rate of 181 M/s. An implementation geared for good performance for small values of \(k\) is given in [179], a C++ adaptation is [FXT: comb/combination-lam-demo.cc]. The combinations \((32, 7)\) are generated at a rate of 190 M/s, and 250 M/s for the combinations \((64, 7)\). The routine is limited to values \(k \geq 2\).

### 6.4 The Eades-McKay strong minimal-change order

In any Gray code order for combinations just one element is moved between to successive combinations. When an element is moved across any other then there is more than one change on the set representation. If \(i\) elements are crossed then \(i + 1\) entries in the set change:

- set: \{0, 1, 2, 3\} delta set: 1111
- set: \{1, 2, 3, 4\} delta set: .1111

A strong minimal-change order is a Gray code where only one entry in the set representation is changed per step. That is, only zeros in the delta set representation are crossed, and the moves are called homogeneous. One such order is the Eades-McKay sequence described in [110]. The Eades-McKay sequence for the combinations \((6, 3)\) is shown in figure 6.4-A (left).

### 6.4.1 Recursive generation

The Eades-McKay order can be generated with the program [FXT: comb/combination-emk-rec-demo.cc]:

---

[FXTbook draft of 2008-August-17]
The Eades-McKay strong minimal-change order

<table>
<thead>
<tr>
<th>Eades-McKay</th>
<th>complemented Eades-McKay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: { 4, 5, 6 } ....111</td>
<td>1: { 4, 5, 6 } ....111</td>
</tr>
<tr>
<td>2: { 3, 5, 6 } ....1.1</td>
<td>2: { 3, 5, 6 } ....1.1</td>
</tr>
<tr>
<td>3: { 2, 5, 6 } .1.111</td>
<td>3: { 2, 5, 6 } .1.111</td>
</tr>
<tr>
<td>4: { 1, 5, 6 } .1.111</td>
<td>4: { 1, 5, 6 } .1.111</td>
</tr>
<tr>
<td>5: { 0, 5, 6 } 1..111</td>
<td>5: { 0, 5, 6 } 1..111</td>
</tr>
<tr>
<td>6: { 0, 4, 6 } 1...11</td>
<td>6: { 0, 4, 6 } 1...11</td>
</tr>
<tr>
<td>7: { 0, 2, 6 } 1.111</td>
<td>7: { 1, 4, 6 } 1.111</td>
</tr>
<tr>
<td>8: { 1, 2, 6 } .111</td>
<td>8: { 2, 4, 6 } .111</td>
</tr>
<tr>
<td>9: { 1, 3, 6 } .1.11</td>
<td>9: { 3, 4, 6 } .1.11</td>
</tr>
<tr>
<td>10: { 0, 3, 6 } 1..11</td>
<td>10: { 2, 3, 6 } 1..11</td>
</tr>
<tr>
<td>11: { 2, 3, 6 } .111</td>
<td>11: { 2, 3, 6 } .111</td>
</tr>
<tr>
<td>12: { 2, 4, 6 } .11.1</td>
<td>12: { 2, 4, 6 } .11.1</td>
</tr>
<tr>
<td>13: { 1, 4, 6 } .111</td>
<td>13: { 0, 2, 6 } 1.1.1</td>
</tr>
<tr>
<td>14: { 0, 4, 6 } 1.111</td>
<td>14: { 1, 2, 6 } 1.111</td>
</tr>
<tr>
<td>15: { 3, 4, 6 } .111</td>
<td>15: { 0, 1, 6 } 11...1</td>
</tr>
<tr>
<td>16: { 3, 4, 5 } ...111.</td>
<td>16: { 0, 1, 5 } 11...1</td>
</tr>
<tr>
<td>17: { 2, 4, 5 } .111.</td>
<td>17: { 0, 2, 5 } 11...1</td>
</tr>
<tr>
<td>18: { 1, 4, 5 } .11.1.</td>
<td>18: { 1, 3, 5 } .111.</td>
</tr>
<tr>
<td>19: { 0, 4, 5 } 1..11.</td>
<td>19: { 2, 3, 5 } ..111.</td>
</tr>
<tr>
<td>20: { 0, 1, 5 } 11..1.</td>
<td>20: { 0, 3, 5 } 1..11.</td>
</tr>
<tr>
<td>21: { 0, 2, 5 } 1.111.</td>
<td>21: { 0, 2, 5 } 1.111.</td>
</tr>
<tr>
<td>22: { 1, 2, 5 } .111.1</td>
<td>22: { 0, 5, 4 } 1..11.</td>
</tr>
<tr>
<td>23: { 1, 3, 5 } .111</td>
<td>23: { 1, 4, 5 } .111</td>
</tr>
<tr>
<td>24: { 0, 3, 5 } 1..11</td>
<td>24: { 2, 4, 5 } .111</td>
</tr>
<tr>
<td>25: { 2, 3, 5 } .111</td>
<td>25: { 3, 4, 5 } ...111</td>
</tr>
<tr>
<td>26: { 2, 3, 4 } ..111</td>
<td>26: { 2, 3, 4 } ..111</td>
</tr>
<tr>
<td>27: { 1, 3, 4 } .11.1</td>
<td>27: { 1, 3, 4 } .11.1</td>
</tr>
<tr>
<td>28: { 0, 3, 4 } 1.1.1</td>
<td>28: { 0, 3, 4 } 1.1.1</td>
</tr>
<tr>
<td>29: { 0, 1, 4 } 11.1</td>
<td>29: { 0, 1, 4 } 11.1</td>
</tr>
<tr>
<td>30: { 0, 2, 4 } 1.111</td>
<td>30: { 1, 2, 4 } .111</td>
</tr>
<tr>
<td>31: { 1, 2, 4 } .111</td>
<td>31: { 0, 1, 4 } 11.1</td>
</tr>
<tr>
<td>32: { 1, 2, 3 } .111</td>
<td>32: { 0, 1, 3 } 11.1</td>
</tr>
<tr>
<td>33: { 0, 2, 3 } 11..1</td>
<td>33: { 0, 2, 3 } 11..1</td>
</tr>
<tr>
<td>34: { 0, 1, 3 } 11.1</td>
<td>34: { 1, 2, 3 } .111</td>
</tr>
<tr>
<td>35: { 0, 1, 2 } 111....</td>
<td>35: { 0, 1, 2 } 111....</td>
</tr>
</tbody>
</table>

![Figure 6.4-A: Combinations \( \binom{32}{20} \) in the Eades-McKay order (left), and in the complemented Eades-McKay order (right).](image-url)

The combinations \( \binom{32}{20} \) are generated at a rate of about 44 million per second, the combinations \( \binom{32}{12} \) at a rate of 34 million per second.

```c
ulong *rv; // elements in combination at rv[1] ... rv[k]

void
comb_emk(ulong n, ulong k, bool z)
{
    if ( k==n )
        for (ulong j=1; j<=k; ++j) rv[j] = j;
    visit();
    return;
}

if ( (n>=2) && (k>=2) ) { rv[k] = n; rv[k-1] = n-1; comb_emk(n-2, k-2, z); }
if ( (n>=2) && (k>=1) ) { rv[k] = n; comb_emk(n-2, k-1, !z); }
if ( (n>=1) ) { comb_emk(n-1, k, z); }
else
    if ( (n>=2) && (k>=1) ) { rv[k] = n; comb_emk(n-2, k-1, !z); }
    if ( (n>=2) && (k>=2) ) { rv[k] = n; rv[k-1] = n-1; comb_emk(n-2, k-2, z); }
```
Chapter 6: Combinations

The underlying recursion for the list \( E(n, k) \) of combinations \( \binom{n}{k} \) is (notation as in relation 6.0-1 on page 176)

\[
\begin{bmatrix}
\binom{n}{k} \\
\binom{n}{k-1} \\
E(n-1, k)
\end{bmatrix} = 
\begin{bmatrix}
\binom{n-2}{k-2} \\
\binom{n-2}{k-1} \\
E(n-2, k-2)
\end{bmatrix}
\]

(6.4-1)

Again, the first equality is for the set representation, the second for the delta-set representation. Counting the elements on both sides gives the relation

\[
\binom{n}{k} = \binom{n-2}{k-2} + \binom{n-2}{k-1} + \binom{n-1}{k}
\]

(6.4-2)

A recursion for the complemented sequence (with respect to the delta sets) is

\[
\begin{bmatrix}
\binom{n}{k} \\
\binom{n}{k-1} \\
E(n-2, k)
\end{bmatrix} = 
\begin{bmatrix}
1 \\
0 \\
E(n-2, k-1)
\end{bmatrix}
\]

(6.4-3)

Counting on both sides gives

\[
\binom{n}{k} = \binom{n-2}{k-2} + \binom{n-2}{k-1} + \binom{n-1}{k}
\]

(6.4-4)

The condition for the recursion end has to be modified:

void comb_emk_compl(ulong n, ulong k, bool z)
{
if ( (k==0) || (k==n) )
{
for (ulong j=1; j<=k; ++j) rv[j] = j;
++ct;
visit();
return;
}
if ( z ) // forward:
{
if ( (n>=1) && (k>=1) ) { rv[k] = n; comb_emk_compl(n-1, k-1, z); } // 1
if ( (n>=2) && (k>=1) ) { rv[k] = n-1; comb_emk_compl(n-2, k-1, !z); } // 01
if ( (n>=2) ) { comb_emk_compl(n-2, k-0, z); } // 00
}
else // backward:
{
if ( (n>=2) ) { comb_emk_compl(n-2, k-0, z); } // 00
if ( (n>=2) && (k>=1) ) { rv[k] = n-1; comb_emk_compl(n-2, k-1, !z); } // 01
if ( (n>=1) ) { rv[k] = n; comb_emk_compl(n-1, k-1, z); } // 1
}
}

The complemented sequence is not a strong Gray code.

### 6.4.2 Iterative generation via modulo moves

An iterative algorithm for the Eades-McKay sequence is given in [FXT: class combination_emk in comb/combination-emk.h]:

class combination_emk
{
public:
ulong *x_; // combination: k elements 0<=x[j]<k in increasing order
ulong *s_; // aux: start of range for moves
ulong *a_; // aux: actual start position of moves
ulong n_, k_; // Combination (n choose k)
6.4: The Eades-McKay strong minimal-change order

```cpp
public:
  combination_emk(ulong n, ulong k)
{
    n_ = n;
    k_ = k;
    x_ = new ulong[k_+1];  // incl. high sentinel
    s_ = new ulong[k_+1];  // incl. high sentinel
    a_ = new ulong[k_];
    x_[k_] = n_;
    first();
}

void first()
{
  for (ulong j=0; j<k_; ++j) x_[j] = j;
  for (ulong j=0; j<k_; ++j) s_[j] = j;
  for (ulong j=0; j<k_; ++j) a_[j] = x_[j];
}
```

The computation of the successor uses modulo steps:

```cpp
ulong next()
// Return position where track changed, return k with last combination
{
  ulong j = k_;
  while ( j-- )  // loop over tracks
  {
    const ulong sj = s_[j];
    const ulong m = x_[j+1] - sj - 1;
    if ( 0!=m )  // unless range empty
    {
      ulong u = x_[j] - sj;
      // modulo moves:
      if ( 0==(j&1) )
      {
        ++u;
        if ( u>m ) u = 0;
      }
      else
      {
        --u;
        if ( u>m ) u = m;
      }
      u += sj;
      if ( u != a_[j] )  // next position != start position
      {
        x_[j] = u;
        s_[j+1] = u+1;
        return j;
      }
      a_[j] = x_[j];
    }
  }
  return k_;  // current combination is last
}
```

The combinations $\binom{32}{12}$ are generated at a rate of about 60 million per second, the combinations $\binom{32}{20}$ at a rate of 85 million per second [FXT: comb/combination-emk-demo.cc].

### 6.4.3 Alternative order via modulo moves

A slight modification of the successor computation gives an ordering where the first and last combination differ by a single transposition (though not a homogeneous one), see figure 6.4-B. The generator is given in [FXT: class combination_mod in [comb/combination-mod.h]](comb/combination-mod.h):
Chapter 6: Combinations

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

Figure 6.4-B: All combinations \( \binom{7}{3} \) (left) and \( \binom{7}{4} \) (right) in mod- and in EMK order.

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```

The rate of generation is identical with the EMK order [FXT: comb/combination-mod-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

```
2 {
3   [--snip--]
4   ulong next()
5 {
6   [--snip--]
7     // modulo moves:
8     // if ( 0==(j&1) ) // gives EMK
9     if ( 0!=(j&1) ) // mod
10    [--snip--]
```
for ‘Even Numbers DOwn, odd numbers up’. A routine for generating the successor in endo order with maximal value \( m \) is [FXT:comb/endo-enup.h]:

```c
inline ulong next_endo(ulong x, ulong m)
{ // Return next number in endo order
  if ( x & 1 ) // x odd
    { x += 2;
      if ( x>m ) x = m - (m&1); // == max even <= m
    }
  else // x even
    { x = ( x==0 ? 1 : x-2 );
    }
  return x;
}
```

The sequences for the first few \( m \) are shown in figure 6.5-A. The routine computes one for the input zero.

An ordering starting with the even numbers in increasing order will be called \( \text{enup} \) (for ‘Even Numbers UP, odd numbers down’). The computation of the successor can be implemented as

```c
static inline ulong next_enup(ulong x, ulong m)
{
  if ( x & 1 ) // x odd
    { x = ( x==1 ? 0 : x-2 );
    }
  else // x even
    { x += 2;
      if ( x>m ) x = m - !(m&1); // max odd <= m
    }
  return x;
}
```

The orderings are reversals of each other, so we define:

```c
static inline ulong prev_endo(ulong x, ulong m) { return next_enup(x, m); }
static inline ulong prev_enup(ulong x, ulong m) { return next_endo(x, m); }
```

A function that returns the \( x \)-th number in enup order with maximal digit \( m \) is

```c
static inline ulong enup_num(ulong x, ulong m)
{
  ulong r = 2*x;
  if ( r>m ) r = 2*m+1 - r;
  return r;
}
```

The function will only work if \( x \leq m \). For example, with \( m = 5 \):

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The inverse function is

```c
static inline ulong enup_idx(ulong x, ulong m)
{
  const ulong b = x & 1;
  x >>= 1;
  return ( b ? m-x : x );
}
```

The function to map into endo order is

```c
static inline ulong endo_num(ulong x, ulong m)
{
  // return enup_num(m-x, m);
  x = m - x;
  ulong r = 2*x;
  if ( r>m ) r = 2*m+1 - r;
  return r;
}
```

For example,
Its inverse is

```cpp
static inline ulong endo_idx(ulong x, ulong m)
{
    const ulong b = x & 1;
    x >>= 1;
    return ( b ? x : m-x );
}
```

### 6.5.2 The endo and enup orderings for combinations

Two strong minimal-change orderings for combinations can be obtained via moves in enup and endo order. Figure 6.5-B shows an ordering where the moves to the right are on even positions (enup order, left). If the moves to the right are on odd positions (endo order) then Chase’s sequence is obtained (right). Both have the property of being two-close: an element in the delta set moves by at most two positions (and the move is homogeneous, no other element is crossed). This property is a direct consequence of the fact that all moves with enup and endo order are by at most two positions. An implementation of an iterative algorithm for the computation of the combinations in enup order is [FXT: class combination_enup in comb/combination-enup.h].

```cpp
class combination_enup
{
public:
    ulong *x_; // combination: k elements 0<=x[j]<k in increasing order
    ulong *s_; // aux: start of range for enup moves
    ulong *a_; // aux: actual start position of enup moves
    ulong n_, k_; // Combination (n choose k)

public:
    combination_enup(ulong n, ulong k)
    {
        n_ = n;
        k_ = k;
        x_ = new ulong[k_+1]; // incl. padding x_[k]
        s_ = new ulong[k_+1]; // incl. padding x_[k]
        a_ = new ulong[k_];
        x_[k_] = n_;
        first();
    }

    void first()
    {
        for (ulong j=0; j<k_; ++j) x_[j] = j;
        for (ulong j=0; j<k_; ++j) s_[j] = j;
        for (ulong j=0; j<k_; ++j) a_[j] = x_[j];
    }

    ulong next()
    // Return position where track changed, return k with last combination
    {
        ulong j = k_;  
        while ( j-- ) // loop over tracks
            {
            const ulong sj = s_[j];
            const ulong m = x_[j+1] - sj - 1;
            if ( 0!=m ) // unless range empty
                {
                ulong u = x_[j] - sj;
                // move right on even positions:
                if ( 0==((sj&1)) ) u = next_enup(u, m);
```
Figure 6.5-B: Combinations $(_8^3)$ via enup moves (left), and via endo moves (Chase’s sequence, right).
Chapter 6: Combinations

The combinations \( \binom{32}{20} \) are generated at a rate of 45 million objects per second, the combinations \( \binom{32}{12} \) at a rate of 55 million per second. The only change in the implementation for computing the endo ordering is (at the obvious place in the code) [FXT: comb/combination-endo.h]:

```cpp
// move right on odd positions:
if ( 0==(sj&1) ) u = next_endo(u, m);
else u = next_enup(u, m);
```

The ordering obtained by endo moves is called *Chase’s sequence*. Figure 6.5-B was created with the programs [FXT: comb/combination-enup-demo.cc] and [FXT: comb/combination-endo-demo.cc].

The underlying recursion for the list \( U(n,k) \) of combinations \( \binom{n}{k} \) in enup order is

\[
U(n,k) = \begin{bmatrix} U(n-1, k-2) & U(n-2, k-2) \\ U(n-2, k-1) & U(n-2, k-1) \end{bmatrix}
\]

(6.5-1)

The recursion is very similar to relation 6.4-1 on page 184, therefore a recursive routine is easy to obtain. The crucial part of the routine is [FXT: comb/combination-enup-rec-demo.cc]:

```cpp
void comb_enup(ulong n, ulong k, bool z) {
    if ( k==n ) { visit(); return; }
    if ( z ) // forward:
        if ( (n>=2) && (k>=2) ) { rv[k] = n; rv[k-1] = n-1; comb_enup(n-2, k-2, z); }
        if ( (n>=2) && (k>=1) ) { rv[k] = n; comb_enup(n-2, k-1, z); }
        if ( (n>=1) ) { comb_enup(n-1, k, !z); }
    else // backward:
        if ( (n>=1) ) { comb_enup(n-1, k, !z); }
        if ( (n>=2) && (k>=1) ) { rv[k] = n; comb_enup(n-2, k-1, z); }
        if ( (n>=2) && (k>=2) ) { rv[k] = n; rv[k-1] = n-1; comb_enup(n-2, k-2, z); }
}
```

A recursion for the complemented sequence (with respect to the delta sets) is

\[
U'(n,k) = \begin{bmatrix} U'(n-1, k-1) & U'(n-2, k-1) \\ U'(n-2, k) & U'(n-2, k) \end{bmatrix}
\]

(6.5-2)

The condition for the recursion end has to be modified:

```cpp
void comb_enup_compl(ulong n, ulong k, bool z) {
    if ( (k==0) || (k==n) ) { visit(); return; }
    if ( z ) // forward:
```
6.6 Recursive generation of certain orderings

{  if ( (n>=1) && (k>=1) )  { rv[k] = n;  comb_enup_compl(n-1, k-1, !z); }  // 1
8  if ( (n>=2) && (k>=1) )  { rv[k] = n-1;  comb_enup_compl(n-2, k-1, z); }  // 01
9  if ( (n>=2) )  { comb_enup_compl(n-2, k-0, z); }  // 00
10}
11else  // backward:
12{  if ( (n>=2) )  { comb_enup_compl(n-2, k-0, z); }  // 00
13  if ( (n>=2) && (k>=1) )  { rv[k] = n-1;  comb_enup_compl(n-2, k-1, z); }  // 01
14  if ( (n>=1) && (k>=1) )  { rv[k] = n;  comb_enup_compl(n-1, k-1, !z); }  // 1
15}
16}
17}

An algorithm for Chase’s sequence that generates delta sets is described in [174], its implementation is
given in [FXT: class combination_chase in comb/combination-chase.h]. The routine generates about
64 million combinations per second for both $\binom{32}{20}$ and $\binom{32}{12}$ [FXT: comb/combination-chase-demo.cc].

6.6 Recursive generation of certain orderings

<table>
<thead>
<tr>
<th>lexicographic</th>
<th>Gray code</th>
<th>compl. enup</th>
<th>compl. Eades-McKay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>2:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>3:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>4:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>5:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>6:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>7:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>8:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>9:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>10:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>11:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>12:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>13:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>14:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>15:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>16:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>17:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>18:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>19:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>20:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>21:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>22:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>23:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>24:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>25:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>26:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>27:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>28:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>29:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>30:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>31:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>32:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>33:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>34:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>35:</td>
<td>11111111</td>
<td>11111111</td>
<td>1111111111111111</td>
</tr>
</tbody>
</table>

Figure 6.6-A: All combinations $\binom{7}{3}$ in lexicographic, minimal-change, complemented enup, and complemented Eades-McKay order (from left to right).

We give a simple recursive routine to obtain the orders shown in figure 6.6-A. The combinations are generated as sets [FXT: class comb_rec in comb/combination-rec.h]:

```
1 class comb_rec
2 {
3 public:
4     ulong n_, k_; // (n choose k)
5     ulong *rv_;  // combination: k elements 0<=x[j]<k in increasing order
6     // == Record of Visits in graph
7     ulong rq_;   // condition that determines the order:
8     // 0 => lexicographic order
9     // 1 => Gray code
```
Chapter 6: Combinations

10  // 2 ==> complemented enup order
11  // 3 ==> complemented Eades-McKay sequence
12  ulong nq_; // whether to reverse order
13  [--snip--]
14  void (*visit_)(const comb_rec &); // function to call with each combination
15  [--snip--]
16  void generate(void (*visit)(const comb_rec &), ulong rq, ulong nq=0)
17  {
18      visit_ = visit;
19      rq_ = rq;
20      nq_ = nq;
21      ct_ = 0;
22      rct_ = 0;
23      next_rec(0);
24  }

The recursion function is given in [FXT: comb/combination-rec.cc]:

1  void comb_rec::next_rec(ulong d)
2  {
3      ulong r = k_ - d; // number of elements remaining
4      if ( 0==r ) visit_(*this);
5      else
6      {
7          ulong rv1 = rv_[d-1]; // left neighbor
8          bool q;
9          switch ( rq_ )
10          {
11              case 0: q = 1; break; // 0 ==> lexicographic order
12              case 1: q = !(d&1); break; // 1 ==> Gray code
13              case 2: q = rv1&1; break; // 2 ==> complemented enup order
14              case 3: q = (d^rv1)&1; break; // 3 ==> complemented Eades-McKay sequence
15                  default: q = 1;
16              }
17          q ^= nq_; // reversed order if nq == true
18          if ( q ) // forward:
19              for (ulong x=rv1+1; x<=n_-r; ++x) { rv_[d] = x; next_rec(d+1); }
20          else // backward:
21              for (ulong x=n_-r; (long)x>=(long)rv1+1; --x) { rv_[d] = x; next_rec(d+1); }
22      }
23  }

Figure 6.6-A was created with the program [FXT: comb/combination-rec-demo.cc]. The routine generates the combinations \( \binom{32}{20} \) at a rate of about 32 million objects per second. The combinations \( \binom{12}{6} \) are generated at a rate of 45 million objects per second.
Chapter 7

Compositions

The compositions of \( n \) into at most \( k \) parts (‘\( k \)-compositions of \( n \)’) are the ordered tuples \( (x_0, x_1, \ldots, x_{k-1}) \) where \( x_0 + x_1 + \ldots + x_{k-1} = n \) and \( 0 \leq x_i \leq n \). Order matters: one 4-composition of 7 is \((0, 1, 5, 1)\), different ones are \((5, 0, 1, 1)\) and \((0, 5, 1, 1)\). To obtain the compositions of \( n \) into exactly \( k \) parts (where \( k \leq n \)) generate the compositions of \( n - k \) into \( k \) parts and add one to each position.

7.1 Co-lexicographic order

The compositions in co-lexicographic (colex) order are shown in figure 7.1-A. The generator is implemented as [FXT: class composition_colex in comb/composition-colex.h]:

```cpp
class composition_colex
{
    public:
        ulong n_, k_; // composition of n into k parts
        ulong *x_; // data (k elements)
    // [snip--]

doctor

    void first()
    {
        x_[0] = n_; // all in first position
        for (ulong k=1; k<k_; ++k) x_[k] = 0;
    }

    void last()
    {
        for (ulong k=0; k<k_; ++k) x_[k] = 0;
        x_[k_-1] = n_; // all in last position
    }
    // [snip--]
}
```

The methods to compute the successor and predecessor are:

```cpp
    ulong next()
    // Return position of rightmost change, return k with last composition.
    {
        ulong j = 0;
        while ( 0==x_[j] ) ++j; // find first nonzero
        if ( j==k_-1 ) return k_; // current composition is last
        ulong v = x_[j]; // value of first nonzero
        x_[j] = 0; // set to zero
        x_[0] = v - 1; // value-1 to first position
        ++j;
        ++x_[j]; // increment next position
        return j;
    }
```
Chapter 7: Compositions

Figure 7.1-A: The compositions of 3 into 5 parts in co-lexicographic order and positions of the rightmost change, and delta sets of the corresponding combinations (left); and the corresponding data for compositions of 7 into 3 parts (right). Dots denote zeros.

With each transition at most 3 entries are changed. The compositions of 10 into 30 parts (sparse case) are generated at a rate of about 110 million per second, the compositions of 30 into 10 parts (dense case) at about 200 million per second [FXT: comb/composition-colex-demo.cc]. With the dense case (corresponding to the right of figure 7.1-A) the computation is faster as the position to change is found earlier.

[fxtbook draft of 2008-August-17]
Optimized implementation

An implementation that is efficient also for the sparse case (that is, \(k\) much greater than \(n\)) is \[FXT: \texttt{class composition\_colex2} \text{ in comb/composition-colex2.h}\]. An additional variable \(p_0\) records the position of the first nonzero entry. The method to compute the successor is:

```cpp
class composition_colex2
{
    ulong next()
    // Return position of rightmost change, return \(k\) with last composition.
    {
        ulong j = p0_; // position of first nonzero
        if ( j==k_-1 ) return k_; // current composition is last
        ulong v = x_[j]; // value of first nonzero
        x_[j] = 0; // set to zero
        --v;
        x_[0] = v; // value-1 to first position
        ++p0_; // first nonzero one more right except ...
        if ( 0!=v ) p0_ = 0; // ... if value v was not one
        ++j;
        ++x_[j]; // increment next position
        return j;
    }
};
```

About 182 million compositions are generated per second, independent of either \(n\) and \(k\) \[FXT: \texttt{comb/composition-colex2-demo.cc}\]. With the line

```
#define COMP_COLEX2_MAX_ARRAY_LEN 128
```

just before the class definition an array is used instead of a pointer. The fixed array length limits the value of \(k\) so by default the line is commented out. Using an array gives a significant speedup, the rate is about 365 million per second (about 6 CPU cycles per update).

### 7.2 Co-lexicographic order for compositions into exactly \(k\) parts

The compositions of \(n\) into exactly \(k\) parts (where \(k \geq n\)) can be obtained from the compositions of \(n-k\) into at most \(k\) parts as shown in figure 7.2-A. The listing was created with the program \[FXT: \texttt{comb/composition-ex-colex-demo.cc}\]. The compositions can be generated in co-lexicographic order using \[FXT: \texttt{class composition\_ex\_colex} \text{ in comb/composition-ex-colex.h}\].

```cpp
class composition_ex_colex
{
    public:
        ulong n_, k_; // composition of n into exactly k parts
        ulong *x_; // data (k elements)
        ulong nk1_; // ==n-k+1
    public:
        composition_ex_colex(ulong n, ulong k)
        // Must have n>k
        {
            n_ = n;
            k_ = k;
            nk1_ = n - k + 1; // must be >= 1
            if ( (long)nk1_ < 1 ) nk1_ = 1; // avoid hang with invalid pair n,k
            x_ = new ulong[k_+1];
            x_[k] = 0; // not one
            first();
        }
};
```

The variable \(nk1\) is the maximal entry in the compositions:

[fxtbook draft of 2008-August-17]
Figure 7.2-A: The compositions of $n = 8$ into exactly $k = 5$ parts (left) are obtained from the compositions of $n - k = 3$ into at most $k = 5$ parts (right). Co-lexicographic order, dots denote zeros.

```
    exact comp. chg composition
1:  [4 1 1 1 1]  4 ]3 . . . .   [4 3 1 . . 1
2:  [3 2 1 1 1]  1 ]2 . . . .   [3 2 1 . . 1
3:  [2 3 1 1 1]  1 ]1 . . . .   [2 3 1 . . 1
4:  [1 4 1 1 1]  1 [ . 3 . . .   [1 4 1 . . .
5:  [3 1 2 1 1]  2 [2 . 1 . .   [3 1 2 . . 1
6:  [2 2 1 1 1]  1 [1 1 1 . .   [2 2 1 . . .
7:  [1 3 1 1 1]  1 [ . 2 1 . .   [1 3 1 . . 1
8:  [2 1 3 1 1]  2 [1 . 2 . .   [2 1 3 . . 1
9:  [1 2 3 1 1]  2 [ . 1 2 . .   [1 2 3 . . 1
10: [1 1 4 1 1]  3 [ . . 3 . .   [1 1 4 . . 1
11: [3 1 1 2 1]  3 [2 . . 1 .   [3 1 1 . . 1
12: [2 2 1 2 1]  1 [1 1 1 . .   [2 2 1 . . .
13: [1 3 1 2 1]  1 [ . 2 1 . .   [1 3 1 . . 1
14: [2 1 3 1 1]  2 [1 . 2 . .   [2 1 3 . . 1
15: [1 2 3 1 1]  2 [ . 1 2 . .   [1 2 3 . . 1
16: [1 1 4 1 1]  3 [ . . 3 . .   [1 1 4 . . 1
17: [2 2 1 1 2]  1 [1 1 1 . .   [2 2 1 . . .
18: [1 3 1 1 2]  1 [ . 2 1 . .   [1 3 1 . . 1
19: [2 1 2 1 2]  2 [1 . 1 1 .   [2 1 2 . . 1
20: [1 2 2 1 2]  2 [ . 1 1 1 .   [1 2 2 . . 1
21: [1 1 4 1 2]  3 [ . . 3 . .   [1 1 4 . . 1
22: [2 2 1 2 1]  1 [1 1 1 . .   [2 2 1 . . .
23: [1 3 1 2 1]  1 [ . 2 1 . .   [1 3 1 . . 1
24: [2 1 2 1 2]  2 [1 . 1 1 .   [2 1 2 . . 1
25: [1 2 2 1 2]  2 [ . 1 1 1 .   [1 2 2 . . 1
26: [1 1 4 1 2]  3 [ . . 3 . .   [1 1 4 . . 1
27: [2 2 1 2 1]  1 [1 1 1 . .   [2 2 1 . . .
28: [1 3 1 2 1]  1 [ . 2 1 . .   [1 3 1 . . 1
29: [2 1 2 1 2]  2 [1 . 1 1 .   [2 1 2 . . 1
30: [1 2 2 1 2]  2 [ . 1 1 1 .   [1 2 2 . . 1
31: [1 1 4 1 2]  3 [ . . 3 . .   [1 1 4 . . 1
32: [2 2 1 2 1]  1 [1 1 1 . .   [2 2 1 . . .
33: [1 3 1 2 1]  1 [ . 2 1 . .   [1 3 1 . . 1
34: [2 1 2 1 2]  2 [1 . 1 1 .   [2 1 2 . . 1
35: [1 2 2 1 2]  2 [ . 1 1 1 .   [1 2 2 . . 1
```

The methods for computing the successor and predecessor are adaptations from the routines from the compositions into at most $k$ parts:

```
void first() {
    x_[0] = nk1_; // all in first position
    for (ulong k=1; k<k_; ++k) x_[k] = 1;
}

void last() {
    for (ulong k=0; k<k_; ++k) x_[k] = 1;
    x_[k_-1] = nk1_; // all in last position
}

ulong next() // Return position of rightmost change, return k with last composition.
{
    ulong j = 0;
    while ( 1==x_[j] ) ++j; // find first greater than one
    if ( j==k_ ) return k_; // current composition is last
    ulong v = x_[j]; // value of first greater one
    x_[j] = 1; // set to one
    x_[0] = v - 1; // value-1 to first position
    ++j;
    ++x_[j]; // increment next position
    return j;
}
```
ulong prev()
    // Return position of rightmost change, return k with last composition.
{
    const ulong v = x_[0]; // value at first position
    if ( nk1_==v ) return k_; // current composition is first
    x_[0] = 1; // set first position to one
    ulong j = 1;
    while ( 1==x_[j] ) ++j; // find next greater than one
    --x_[j]; // decrement value
    x_[j-1] = 1 + v; // set previous position
    return j;
}

The routines are as fast as the generation into at most k parts with the corresponding parameters: the compositions of 40 into 10 parts are generated at about 200 million per second.

7.3 Compositions and combinations

<table>
<thead>
<tr>
<th>combination</th>
<th>delta set</th>
<th>composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: [ 0 1 2 ]</td>
<td>111...</td>
<td>[ 3 . . . ]</td>
</tr>
<tr>
<td>2: [ 0 2 3 ]</td>
<td>1.11..</td>
<td>[ 1 2 . . ]</td>
</tr>
<tr>
<td>3: [ 1 2 3 ]</td>
<td>.111..</td>
<td>[ . 3 . . ]</td>
</tr>
<tr>
<td>4: [ 0 1 3 ]</td>
<td>11.1..</td>
<td>[ 2 1 . . ]</td>
</tr>
<tr>
<td>5: [ 0 3 4 ]</td>
<td>.1..11</td>
<td>[ . 2 . . ]</td>
</tr>
<tr>
<td>6: [ 1 3 4 ]</td>
<td>.11.11</td>
<td>[ . 1 2 . ]</td>
</tr>
<tr>
<td>7: [ 2 3 4 ]</td>
<td>.111..</td>
<td>[ . . 3 . ]</td>
</tr>
<tr>
<td>8: [ 0 2 4 ]</td>
<td>1.1.1.</td>
<td>[ 1 1 1 . ]</td>
</tr>
<tr>
<td>9: [ 1 2 4 ]</td>
<td>.11.11</td>
<td>[ . 2 1 . ]</td>
</tr>
<tr>
<td>10: [ 0 1 4 ]</td>
<td>11.1.</td>
<td>[ 2 . 1 . ]</td>
</tr>
<tr>
<td>11: [ 0 4 5 ]</td>
<td>1...11</td>
<td>[ 1 . . 2 ]</td>
</tr>
<tr>
<td>12: [ 1 4 5 ]</td>
<td>.1..11</td>
<td>[ . 1 . 2 ]</td>
</tr>
<tr>
<td>13: [ 2 4 5 ]</td>
<td>.11.11</td>
<td>[ . . 1 2 ]</td>
</tr>
<tr>
<td>14: [ 3 4 5 ]</td>
<td>.111..</td>
<td>[ . . . 3 ]</td>
</tr>
<tr>
<td>15: [ 0 3 5 ]</td>
<td>1..11</td>
<td>[ 1 . 1 1 ]</td>
</tr>
<tr>
<td>16: [ 1 3 5 ]</td>
<td>.111..</td>
<td>[ . 1 1 1 ]</td>
</tr>
<tr>
<td>17: [ 2 3 5 ]</td>
<td>.111..</td>
<td>[ . . 2 1 ]</td>
</tr>
<tr>
<td>18: [ 0 2 5 ]</td>
<td>11..1</td>
<td>[ 1 1 . 1 ]</td>
</tr>
<tr>
<td>19: [ 1 2 5 ]</td>
<td>.11..1</td>
<td>[ . 2 . 1 ]</td>
</tr>
<tr>
<td>20: [ 0 1 5 ]</td>
<td>11111</td>
<td>[ 2 . . 1 ]</td>
</tr>
</tbody>
</table>

Figure 7.3-A: Combinations 6 choose 3 (left) and the corresponding compositions of 3 into 4 parts (right). Dots denote zeros. Note that while the sequence of combinations has a minimal-change property the corresponding sequence of compositions does not.

Figure 7.3-A shows the correspondence between compositions and combinations. The listing was generated using the program [FXT: comb/comb2comp-demo.cc]. Entries in the left column are combinations of 3 parts out of 6. The middle column is the representation of the combinations as delta sets. It also is a binary representation of a composition: A run of r consecutive ones corresponds to an entry r in the composition at the right.

Now write \( P(n, k) \) for the compositions of n into (at most) k parts and \( B(N, K) \) for the combination \( \binom{N}{K} \):

A composition of n into at most k parts corresponds to a composition of \( K = n \) parts from \( N = n + k - 1 \) elements, symbolically,

\[
P(n, k) \leftrightarrow B(N, K) = B(n + k - 1, n)
\]

A combination of \( K \) elements out of \( N \) corresponds to a composition of \( n \) into at most \( k \) parts where \( n = K \) and \( k = N - K + 1 \):

\[
B(N, K) \leftrightarrow P(n, k) = P(K, N - K + 1)
\]
Chapter 7: Compositions

Routines for the conversion between combinations and compositions are given in [FXT: comb/comp2comb.h]. The following routine converts a composition into the corresponding combination:

```c
inline void comp2comb(const ulong *p, ulong k, ulong *b)
// Convert composition P(*, k) in p[] to combination in b[]
{
    for (ulong j=0, i=0, z=0; j<k; ++j)
        {ulong pj = p[j];
            for (ulong w=0; w<pj; ++w) b[i++] = z++;
            ++z;
        }
}
```

The conversion of a combination into the corresponding composition can be implemented as

```c
inline void comb2comp(const ulong *b, ulong N, ulong K, ulong *p)
// Convert combination B(N, K) in b[] to composition P(*,k) in p[]
// Must have: K>0
{
    ulong k = N-K+1;
    for (ulong z=0; z<k; ++z) p[z] = 0;
    --k;
    ulong c1 = N;
    while (K--)
        {ulong c0 = b[K];
            ulong d = c1 - c0;
            k -= (d-1);
            ++p[k];
            c1 = c0;
        }
}
```

### 7.4 Minimal-change orders

A minimal-change order (Gray code) for compositions is such that with each transition one entry is increased by one and another is decreased by one. A recursion for the compositions $P(n,k)$ of $n$ into $k$ parts in lexicographic order is (notation as in relation 12.0-1 on page 299)

$$P(n,k) = \begin{bmatrix} 0 & P(n-0, k-1) \\ 1 & P(n-1, k-1) \\ 2 & P(n-2, k-1) \\ 3 & P(n-3, k-1) \\ 4 & P(n-4, k-1) \\ \vdots \\ n & P(0, k-1) \end{bmatrix}$$

(7.4-1)

A simple variation gives a Gray code, we change the direction if the element is even:

$$P(n,k) = \begin{bmatrix} 0 & P^R(n-0, k-1) \\ 1 & P(n-1, k-1) \\ 2 & P^R(n-2, k-1) \\ 3 & P(n-3, k-1) \\ 4 & P^R(n-4, k-1) \\ \vdots \end{bmatrix}$$

(7.4-2)
### 7.4: Minimal-change orders

**Figure 7.4-A:** Compositions of 3 into 5 parts and the corresponding combinations as delta sets and sets in two minimal-change orders: order with enup moves (left) and order with modulo moves (right). The ordering by enup moves is a two-close Gray code. Dots denote zeros.

<table>
<thead>
<tr>
<th>composition</th>
<th>combination</th>
<th>composition</th>
<th>combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: [. . . . 3]</td>
<td>...111. [ 3 4 5 ]</td>
<td>1: [ 3 . . . . ]</td>
<td>111.... [ 0 1 2 ]</td>
</tr>
<tr>
<td>2: [ . 1 . . 2 ]</td>
<td>...111. [ 1 4 5 ]</td>
<td>...111. [ 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>3: [ 1 . 3 . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>4: [ . . 1 . 2 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>5: [ . . 2 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>6: [ . . 2 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>7: [ 1 1 . . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>8: [ 2 1 . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>9: [ 1 1 . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>10: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>11: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>12: [ 1 1 . . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>13: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>14: [ 1 1 . . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>15: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>16: [ 1 1 . . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>17: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>18: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>19: [ 1 1 . . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>20: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>21: [ 1 1 . . . . ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>22: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>23: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>24: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>25: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>26: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>27: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>28: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>29: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>30: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>31: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>32: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>33: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>34: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
<tr>
<td>35: [ . 1 . 1 . 1 ]</td>
<td>...111. [ 1 2 5 ]</td>
<td>...111. [ 1 1 2 3 4 5 ]</td>
<td>...111.... [ 1 2 3 ]</td>
</tr>
</tbody>
</table>

The ordering is shown in figure 7.4-A (left), the corresponding combinations are in the (reversed) enup order from section 6.5.2 on page 188. If we change directions at the odd elements

$$P(n, k) = \begin{bmatrix} 0 & P(n - 0, k - 1) \\ 1 & P^R(n - 1, k - 1) \\ 2 & P(n - 2, k - 1) \\ 3 & P^R(n - 3, k - 1) \\ 4 & P(n - 4, k - 1) \\ \vdots \end{bmatrix}$$

then we obtain an ordering (right of figure 7.4-A) corresponding to the combinations are in the (reversed) Eades-McKay order from section 6.4 on page 182. The listings were created with the program [FXT: comb/composition-gray-rec-demo.cc].

Gray codes for combinations correspond to Gray codes for combinations where no element in the delta set crosses another. The standard Gray code for combinations does not lead to a Gray code for compositions as shown in figure 7.3-A on page 197. When the directions in the recursions are always changed then the compositions correspond to combinations that have the complemented delta sets of the standard Gray code in reversed order.

Orderings where the changes involve just one pair of adjacent entries (shown in figure 7.4-B) correspond to the complemented strong Gray codes for combinations. The amount of change is greater than one in general. The listings were created with the program [FXT: comb/combination-rec-demo.cc], see section 6.6.

[fxtbook draft of 2008-August-17]
<table>
<thead>
<tr>
<th>combination</th>
<th>composition</th>
<th>combination</th>
<th>composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: [0 5 6]</td>
<td>1....11</td>
<td>1: [0 1 2]</td>
<td>111.... [3]</td>
</tr>
<tr>
<td>2: [0 4 6]</td>
<td>1....11</td>
<td>2: [0 1 3]</td>
<td>111.... [2]</td>
</tr>
<tr>
<td>3: [0 4 5]</td>
<td>1....11</td>
<td>3: [0 1 4]</td>
<td>111.... [2]</td>
</tr>
<tr>
<td>4: [0 3 4]</td>
<td>1....11</td>
<td>4: [0 1 5]</td>
<td>111.... [2]</td>
</tr>
<tr>
<td>5: [0 3 5]</td>
<td>1....11</td>
<td>5: [0 1 6]</td>
<td>111.... [2]</td>
</tr>
<tr>
<td>7: [0 2 6]</td>
<td>1....11</td>
<td>7: [0 2 5]</td>
<td>111.... [1]</td>
</tr>
<tr>
<td>8: [0 2 5]</td>
<td>1....11</td>
<td>8: [0 2 4]</td>
<td>111.... [1]</td>
</tr>
<tr>
<td>9: [0 2 4]</td>
<td>1....11</td>
<td>9: [0 2 3]</td>
<td>111.... [1]</td>
</tr>
<tr>
<td>10: [0 2 3]</td>
<td>1....11</td>
<td>10: [0 3 4]</td>
<td>111.... [1]</td>
</tr>
</tbody>
</table>

**Figure 7.4-B:** The (reversed) complemented enup ordering (left) and Eades-McKay sequence (right) for combinations correspond to compositions where only two adjacent entries change with each transition, but by more than one in general.

on page [191]
Chapter 8

Subsets

We give algorithms to generate all subsets of a set of \( n \) elements. There are \( 2^n \) subsets, including the empty set. We further give methods to generate all subsets with \( k \) elements where \( k \) lies in given range: \( k_{\text{min}} \leq k \leq k_{\text{max}} \). The subsets with exactly \( k \) elements are treated in chapter 6 on page 175.

8.1 Lexicographic order

```
1: 1,...  {0} 1,...  {0}
2: 11... {0, 1} 11... {0, 1}
3: 111... {0, 1, 2} 111... {0, 1, 2}
4: 1111... {0, 1, 2, 3} 1111... {0, 1, 2, 3}
5: 11111... {0, 1, 2, 3, 4} 11111... {0, 1, 2, 3, 4}
6: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
7: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
8: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
9: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
10: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
11: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
12: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
13: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
14: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
15: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
16: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
17: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
18: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
19: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
20: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
21: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
22: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
23: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
24: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
25: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
26: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
27: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
28: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
29: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
30: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
31: 111.1... {0, 1, 2} 111.1... {0, 1, 2}
```

Figure 8.1-A: Non-empty subsets of a five element set in lexicographic order for the sets (left), and in lexicographic order for the delta sets (right).

The (nonempty) subsets of a set of five elements in lexicographic order are shown in figure 8.1-A. Note
that the lexicographic order is different for the set and the delta set representation.

### 8.1.1 Generation as delta sets

The listing on the right side of figure 8.1-A is with respect to the delta sets. It was created with the program [FXT: comb/subset-deltalex-demo.cc] which uses the generator [FXT: class subset_deltalex in comb/subset-deltalex.h]:

```cpp
1 class subset_deltalex
2 {
3 public:
4   ulong *d_; // subset as delta set
5   ulong n_; // subsets of the n-set \{0,1,2,\ldots,n-1\}
6    
7 public:
8   subset_deltalex(ulong n)
9   {
10      n_ = n;
11      d_ = new ulong[n+1];
12      d_[n] = 0; // sentinel
13      first();
14   }
15
16 ~subset_deltalex()
17   { delete [] d_; }
18
19 void first()
20   { for (ulong k=0; k<n_; ++k) d_[k] = 0; }
21
The algorithm for the computation of the successor is binary counting:

1 bool next()
2 {
3    ulong k = 0;
4    while ( d_[k]==1 ) { d_[k]=0; ++k; }
5    if ( k==n_ ) return false; // current subset is last
6      else
7      {
8         d_[k] = 1;
9         return true;
10     }
11  
12   const ulong * data() const { return d_; }
13}
14
About 180 million subsets per second are generated. A bit-level algorithm to compute the subsets in lexicographic order is given in section 1.27 on page 73.

### 8.1.2 Generation as sets

The lexicographic order with respect to the set representation is shown at the left side of figure 8.1-A. The routines in [FXT: class subset_lex in comb/subset-lex.h] compute the non-empty sets:

```cpp
1 class subset_lex
2 {
3 public:
4   ulong *x_; // subset of \{0,1,2,\ldots,n-1\}
5   ulong n_; // number of elements in set
6   ulong k_; // index of last element in subset
7      // Number of elements in subset == k+1
8    
9 public:
10   subset_lex(ulong n)
11   {
12      n_ = n;
13      x_ = new ulong[n_];
14      first();
15   }
16
[fxtbook draft of 2008-August-17]
8.2: Minimal-change order

```c
- subset_lex() { delete [] x_; }
ulong first()
{
  k_ = 0;
  x_[0] = 0;
  return k_ + 1;
}
ulong last()
{
  k_ = 0;
  x_[0] = n_ - 1;
  return k_ + 1;
}

ulong next()
// Generate next subset
// Return number of elements in subset
// Return zero if current == last
{
  if ( x_[k_] == n_-1 ) // last element is max ?
    { if ( k_==0 ) { first(); return 0; } --k_; // remove last element
      x_[k_]++; // increase last element
    }
  else // add next element from set:
    {
      ++k_; x_[k_] = x_[k_-1] + 1;
    }
  return k_ + 1;
}
ulong prev()
// Generate previous subset
// Return number of elements in subset
// Return zero if current == first
{
  if ( k_ == 0 ) // only one element ?
    { if ( x_[0]==0 ) { last(); return 0; }
      x_[0]--; // decr first element
      x_[++k_] = n_ - 1; // add element
    }
  else
    {
      if ( x_[k_] == x_[k_-1]+1 ) --k_; // remove last element
      else
        {
          x_[k_--]; // decr last element
          x_[++k_] = n_ - 1; // add element
        }
    }
  return k_ + 1;
}
cast ulong * data() cast { return x_; }
```

The methods `next()` and `prev()` compute the successor and predecessor, respectively:

About 270 million subsets per second are generated with `next()`, and about 155 million with `prev()` [FXT: comb/subset-lex-demo.cc]. A generalization of this order with mixed radix numbers is described in section 9.3 on page 226. A bit-level algorithm is given in section 1.27 on page 73.
Chapter 8: Subsets

0: ..... { } 0: 11111 { 0, 1, 2, 3, 4 }
1: .1.... {0} 1: .1111 { 1, 2, 3, 4 }
2: 11.... {0, 1} 2: ..111 { 0, 2, 3, 4 }
3: .11... {1, 2} 3: 1.111 { 0, 2, 3, 4 }
4: 111.. {0, 1, 2} 4: .111 { 0, 3, 4 }
5: 1111. {0, 1, 2, 3} 5: ...11 { 3, 4 }
6: 1.1.. {0, 2} 6: 1..11 { 1, 3, 4 }
7: ..11. {2} 7: 11.11 { 0, 1, 3, 4 }
8: ..111 {2, 3} 8: 11..1 { 0, 1, 4 }
9: .111. {1, 2, 3} 9: 1.1.1 { 0, 2, 4 }
10: .1111. {1, 2, 3, 4} 10: .1..1 { 1, 4 }
11: 1.11. {0, 1, 2, 3} 11: 1.11 { 0, 3, 4 }
12: 111. {0, 2, 3} 12: 1111 { 0, 1, 3, 4 }
13: 1.11 {0, 1, 2, 3, 4} 13: .11 { 1, 4 }
14: 1111. {0, 1, 2, 3, 4} 14: 1111 { 0, 1, 2 } 15: .11. {0, 2, 3} 15: 1111 { 0, 1, 3, 4 }
16: 11111 {0, 1, 2, 3, 4} 16: 1111 { 0, 1, 2 } 17: .11 {0, 3, 4} 17: 1111 { 0, 1, 2, 4 }
18: 11111 {0, 1, 2, 3, 4} 18: 1111 { 0, 1, 4 } 19: .1.1 {0, 3} 19: 1111 { 0, 1, 2, 4 }
20: 11111 {0, 1, 2, 3, 4} 20: 1111 { 0, 1, 4 } 21: .1.1 {0, 3} 21: 1111 { 0, 1, 2, 4 }
22: 11111 {0, 1, 2, 3, 4} 22: 1111 { 0, 1, 4 } 23: .1.1 {0, 3} 23: 1111 { 0, 1, 2, 4 }
24: 11111 {0, 1, 2, 3, 4} 24: 1111 { 0, 1, 4 } 25: .1.1 {0, 3} 25: 1111 { 0, 1, 2, 4 }
26: 11111 {0, 1, 2, 3, 4} 26: 1111 { 0, 1, 4 } 27: .1.1 {0, 3} 27: 1111 { 0, 1, 2, 4 }
28: 11111 {0, 1, 2, 3, 4} 28: 1111 { 0, 1, 4 } 29: .1.1 {0, 3} 29: 1111 { 0, 1, 2, 4 }
30: 1111 {0, 1, 2, 3, 4} 30: 1111 { 0, 1, 4 } 31: .1.1 {0, 3} 31: 1111 { 0, 1, 2, 4 }

Figure 8.2-A: The subsets of the set \{0, 1, 2, 3, 4\} in minimal-change order (left), and complemented minimal-change order (right). The changes are on the same places for both orders.

8.2 Minimal-change order

8.2.1 Generation as delta sets

The subsets of a set with 5 elements in minimal-change order are shown in figure 8.2-A. The implementation [FXT: class subset_gray_delta in comb/subset-gray-delta.h] uses the Gray code of binary words and updates the position corresponding to the bit that changes in the Gray code:

```cpp
class subset_gray_delta
// Subsets of the set \{0,1,2,...,n-1\} in minimal-change (Gray code) order.
{
  public:
    ulong *x_; // current subset as delta-set
    ulong n_; // number of elements in set <= BITS_PER_LONG
    ulong j_; // position of last change
    ulong ct_; // gray_code(ct_) corresponds to the current subset
    ulong mct_; // max value of ct.
    ~subset_gray_delta() { delete [] x_; }

  public:
    subset_gray_delta(ulong n)
    {
      n_ = (n ? n : 1); // not zero
      x_ = new ulong[n_];
      mct_ = (1UL<n") - 1;
      first(0);
    }

  subset_gray_delta() { delete [] x_; }
}
```

In the initializer one can choose whether the first set is the empty or the full set (left and right of
8.2: Minimal-change order

figure 8.2-A:

```cpp
void first(ulong v=0)
{
    ct_ = 0;
    j_ = n_ - 1;
    for (ulong j=0; j<n_; ++j) x_[j] = v;
}

const ulong * data() const { return x_; }
ulong pos() const { return j_; }
ulong current() const { return ct_; }
ulong next()
// Return position of change, return n with last subset
{
    if ( ct_ == mct_ ) { return n_; }
    ++ct_; 
    j_ = lowest_one_idx( ct_ );
    x_[j_] ^= 1;
    return j_; 
}
ulong prev()
// Return position of change, return n with first subset
{
    if ( ct_ == 0 ) { return n_; }
    j_ = lowest_one_idx( ct_ );
    x_[j_] ^= 1;
    --ct_;
    return j_; 
}
};
```

About 180 million subsets are generated per second [FXT: comb/subset-gray-delta-demo.cc].

8.2.2 Generation as sets

A generator for the subsets of \{1, 2, ..., n\} in set representation is [FXT: class subset_gray in comb/subset-gray.h]:

```cpp
class subset_gray
// Subsets of the set \{1,2,...,n\} in minimal-change (Gray code) order.
{
public:
    ulong *x_; // data k-subset of \{1,2,...,n\} in x[1,...,k]
    ulong n_; // subsets of n-set
    ulong k_; // number of elements in subset

public:
    subset_gray(ulong n)
    {
        n_ = n;
        x_ = new ulong[n+1];
        x_[0] = 0;
        first();
        
    ~subset_gray() { delete [] x_; }
    ulong first() { k_ = 0; return k_; }
    ulong last() { x_[1] = 1; k_ = 1; return k_; }
    const ulong * data() const { return x_*1; }
    const ulong num() const { return k_; }
};
```

The algorithm to compute the successor is described in section 1.16.3 on page 45, see also [156]:

[fxtbook draft of 2008-August-17]
ulong next_even()
{
    if ( x_[k_]==n_ ) // remove n (from end):
    {
        --k_;}
else // append n:
{
    ++k_;    
    x_[k_] = n_;}
return k_;}

ulong next_odd()
{
    if ( x_[k_]-1==x_[k_-1] ) // remove x[k]-1 (from position k-1):
    {
        x_[k_-1] = x_[k_];     
        --k_;}
else // insert x[k]-1 as second last element:
{
        x_[k_+1] = x_[k_];     
        --x_[k_];     
        ++k_;}
return k_;}

public:
ulong next()
{
    if ( 0==(k_&1 ) ) return next_even();
else return next_odd();
}

ulong prev()
{
    if ( 0==(k_&1 ) ) // k even
    {
        if ( 0==k_ ) return last();
        return next_odd();
    }else return next_even();
};

About 241 million subsets per second are generated with next(), and about 167 M/s with prev() [FXT: comb/subset-gray-demo.cc]. With arrays instead of pointers the rates are about 266 M/s and 179 M/s.

8.2.3 Computing just the positions of change

The following algorithm computes only the locations of the changes, it is given in [44]. It can also be obtained as a specialization (for radix 2) of the loopless algorithm for computing a Gray code ordering of mixed radix numbers given section 9.2 on page 222 [FXT: class ruler_func in comb/ruler-func.h].

class ruler_func
// Ruler function sequence: 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 ...   
{
public:
    ulong *f_; // focus pointer
    ulong n_;   
public:
    ruler_func(ulong n)
    {
        n_ = n;
        f_ = new ulong[n+2];
        first();
    }
    ~ruler_func() { delete [] f_; }
A curious ordering for all subsets of a given set can be generated using a binary *De Bruijn sequence* that is a cyclical sequence of zeros and ones that contains each $n$-bit word once. In figure 8.3-A the empty places of the subsets are included to make the nice feature apparent [FXT: comb/subset-debruijn-demo.cc]. The ordering has the *single track* property: each column in this (delta set) representation is a circular shift of the first column. Each subset is made from its predecessor by shifting it to the right and inserting the current element from the sequence. The underlying De Bruijn sequence is

```
1 0 0 0 1 1 0 0 1 0 0 1 1 1 0 1 0 1 1 0 1 0 1 1 1
```

The implementation [FXT: class subset_debruijn in comb/subset-debruijn.h] uses [FXT: class
## Chapter 8: Subsets

### 8.4 Shifts-order for subsets

Figure 8.4-A shows an ordering (shifts-order) of the nonempty subsets of a 6-bit binary word where all linear shifts of a word appear in succession. The generation is done by a simple recursion [FXT: comb/shift-subsets-demo.cc]:

```c++
1  ulong n; // number of bits
2  ulong N; // 2**n
3  
4  void A(ulong x)
5  {
6      if ( x>=N ) return;
7      visit(x);
8      A(2*x);
9      A(2*x+1);
10  }
```

The function visit() simply prints the binary expansion of its argument. The initial call is A(1).

---

**Table 8.3.3-B:** Subsets of a five element set in alternative order corresponding to a De Bruijn sequence. The elements 0, 2, and 4 are present exactly if they are not in figure 8.3-A.

<table>
<thead>
<tr>
<th>Index</th>
<th>Subset</th>
<th>Count</th>
<th>De Bruijn Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{ , , 2, , 4}</td>
<td>#2</td>
<td>{2, 4}</td>
</tr>
<tr>
<td>1</td>
<td>{0, 1, 2, , 4}</td>
<td>#4</td>
<td>{0, 1, 2, 4}</td>
</tr>
<tr>
<td>2</td>
<td>{0, , , 4}</td>
<td>#2</td>
<td>{0, 4}</td>
</tr>
<tr>
<td>3</td>
<td>{0, 2, 3, 4}</td>
<td>#4</td>
<td>{0, 2, 3, 4}</td>
</tr>
<tr>
<td>4</td>
<td>{ , , 2,  }</td>
<td>#1</td>
<td>{2}</td>
</tr>
<tr>
<td>5</td>
<td>{ , 1, 2, , 4}</td>
<td>#3</td>
<td>{1, 2, 4}</td>
</tr>
<tr>
<td>6</td>
<td>{0, 1, , , 4}</td>
<td>#3</td>
<td>{0, 1, 4}</td>
</tr>
<tr>
<td>7</td>
<td>{0, , 3, , 4}</td>
<td>#3</td>
<td>{0, 3, 4}</td>
</tr>
<tr>
<td>8</td>
<td>{ , , , 3, ,}</td>
<td>#2</td>
<td>{3, 4}</td>
</tr>
<tr>
<td>9</td>
<td>{0, 1, 2, , }</td>
<td>#3</td>
<td>{0, 1, 2}</td>
</tr>
<tr>
<td>10</td>
<td>{ , , , , , 4}</td>
<td>#1</td>
<td>{4}</td>
</tr>
<tr>
<td>11</td>
<td>{0, 1, 2, 3, 4}</td>
<td>#5</td>
<td>{0, 1, 2, 3, 4}</td>
</tr>
<tr>
<td>12</td>
<td>{0, , , , , }</td>
<td>#1</td>
<td>{0}</td>
</tr>
<tr>
<td>13</td>
<td>{ , , 2, 3, 4}</td>
<td>#3</td>
<td>{2, 3, 4}</td>
</tr>
<tr>
<td>14</td>
<td>{ , , 1, 2,  }</td>
<td>#2</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>15</td>
<td>{ , 1, , , 4}</td>
<td>#2</td>
<td>{1, 4}</td>
</tr>
<tr>
<td>16</td>
<td>{0, 1, , , 4}</td>
<td>#4</td>
<td>{0, 1, 3, 4}</td>
</tr>
<tr>
<td>17</td>
<td>{ , , , , , 3}</td>
<td>#1</td>
<td>{3}</td>
</tr>
<tr>
<td>18</td>
<td>{0, 1, 2, 3, }</td>
<td>#4</td>
<td>{0, 1, 2, 3}</td>
</tr>
<tr>
<td>19</td>
<td>{ , , , , , }</td>
<td>#0</td>
<td>{}</td>
</tr>
<tr>
<td>20</td>
<td>{ , 1, 2, 3, 4}</td>
<td>#4</td>
<td>{1, 2, 3, 4}</td>
</tr>
<tr>
<td>21</td>
<td>{0, , , , , }</td>
<td>#2</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>22</td>
<td>{ , , , 3, , 4}</td>
<td>#2</td>
<td>{3, 4}</td>
</tr>
<tr>
<td>23</td>
<td>{ , 1, 2, 3, }</td>
<td>#3</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>24</td>
<td>{ , , 1, , , }</td>
<td>#3</td>
<td>{1}</td>
</tr>
<tr>
<td>25</td>
<td>{ , , 1, 3, 4}</td>
<td>#3</td>
<td>{1, 3, 4}</td>
</tr>
<tr>
<td>26</td>
<td>{ , 1, , 3, }</td>
<td>#3</td>
<td>{1, 3}</td>
</tr>
<tr>
<td>27</td>
<td>{0, 1, , , 3}</td>
<td>#3</td>
<td>{0, 1, 3}</td>
</tr>
<tr>
<td>28</td>
<td>{0, , , 3, , 4}</td>
<td>#2</td>
<td>{0, 3}</td>
</tr>
<tr>
<td>29</td>
<td>{0, , 2, , 3}</td>
<td>#3</td>
<td>{0, 2, 3}</td>
</tr>
<tr>
<td>30</td>
<td>{0, , , 2, , }</td>
<td>#2</td>
<td>{0, 2}</td>
</tr>
<tr>
<td>31</td>
<td>{0, , , 2, , 4}</td>
<td>#3</td>
<td>{0, 2, 4}</td>
</tr>
</tbody>
</table>

---

debruijn in [comb/debruijn.h](http://fxtbook.de/comb/debruijn.h) (which in turn uses [FXT: class necklace in comb/necklace.h](http://fxtbook.de/comb/necklace.h)). An algorithm for the generation of binary De Bruijn sequences is given in section 39.1 on page 861.

Successive subsets differ in many elements if the sequency (see section 1.17 on page 48) is big. Using the 'sequency-complemented' subsets (see end of section 1.17) we obtain an ordering where more elements change with small sequencies as shown in figure 8.3-B. This ordering corresponds to the complement-shift sequence of section 19.2.3 on page 387.
### Figure 8.4-A: Nonempty subsets of a 6-bit binary word where all linear shifts of a word appear in succession (shifts-order). All shifts are left shifts.

| 1: .....1 1 | 17: .....1 1 | 33: .....1 1 | 49: .....1 1 |
| 2: .....1 1 | 18: .....1 1 | 34: .....1 1 | 50: .....1 1 |
| 3: .....1 1 | 19: .....1 1 | 35: .....1 1 | 51: .....1 1 |
| 4: .....1 1 | 20: .....1 1 | 36: .....1 1 | 52: .....1 1 |
| 5: .....1 1 | 21: .....1 1 | 37: .....1 1 | 53: .....1 1 |
| 6: .....1 1 | 22: .....1 1 | 38: .....1 1 | 54: .....1 1 |
| 7: .....1 1 | 23: .....1 1 | 39: .....1 1 | 55: .....1 1 |
| 8: .....1 1 | 24: .....1 1 | 40: .....1 1 | 56: .....1 1 |
| 9: .....1 1 | 25: .....1 1 | 41: .....1 1 | 57: .....1 1 |
| 10: .....1 1 | 26: .....1 1 | 42: .....1 1 | 58: .....1 1 |
| 11: .....1 1 | 27: .....1 1 | 43: .....1 1 | 59: .....1 1 |
| 12: .....1 1 | 28: .....1 1 | 44: .....1 1 | 60: .....1 1 |
| 13: .....1 1 | 29: .....1 1 | 45: .....1 1 | 61: .....1 1 |
| 14: .....1 1 | 30: .....1 1 | 46: .....1 1 | 62: .....1 1 |
| 15: .....1 1 | 31: .....1 1 | 47: .....1 1 | 63: .....1 1 |
| 16: .....1 1 | 32: .....1 1 | 48: .....1 1 | 64: .....1 1 |

### Figure 8.4-B: Nonempty subsets of a 6-bit binary word where all linear shifts of a word appear in succession and transitions that are not shifts switch just one bit (minimal-change shifts-order).

| 1: .....1 1 | 17: .....1 1 | 33: .....1 1 | 49: .....1 1 |
| 2: .....1 1 | 18: .....1 1 | 34: .....1 1 | 50: .....1 1 |
| 3: .....1 1 | 19: .....1 1 | 35: .....1 1 | 51: .....1 1 |
| 4: .....1 1 | 20: .....1 1 | 36: .....1 1 | 52: .....1 1 |
| 5: .....1 1 | 21: .....1 1 | 37: .....1 1 | 53: .....1 1 |
| 6: .....1 1 | 22: .....1 1 | 38: .....1 1 | 54: .....1 1 |
| 7: .....1 1 | 23: .....1 1 | 39: .....1 1 | 55: .....1 1 |
| 8: .....1 1 | 24: .....1 1 | 40: .....1 1 | 56: .....1 1 |
| 9: .....1 1 | 25: .....1 1 | 41: .....1 1 | 57: .....1 1 |
| 10: .....1 1 | 26: .....1 1 | 42: .....1 1 | 58: .....1 1 |
| 11: .....1 1 | 27: .....1 1 | 43: .....1 1 | 59: .....1 1 |
| 12: .....1 1 | 28: .....1 1 | 44: .....1 1 | 60: .....1 1 |
| 13: .....1 1 | 29: .....1 1 | 45: .....1 1 | 61: .....1 1 |
| 14: .....1 1 | 30: .....1 1 | 46: .....1 1 | 62: .....1 1 |
| 15: .....1 1 | 31: .....1 1 | 47: .....1 1 | 63: .....1 1 |
| 16: .....1 1 | 32: .....1 1 | 48: .....1 1 | 64: .....1 1 |

### Figure 8.4-C: Nonzero Fibonacci words in an order where all shifts appear in succession.
The transitions that are not shifts change just one bit if the following pair of functions is used for the recursion (minimal-change shifts-order shown in figure 8.4-B):

```cpp
void F(ulong x) {
  if (x >= N) return;
  visit(x);
  F(2*x);
  G(2*x+1);
}

void G(ulong x) {
  if (x >= N) return;
  F(2*x+1);
  G(2*x);
  visit(x);
}
```

The initial call is $F(1)$, the reversed order can be generated via $G(1)$.

We note that a simple variation can be used to generate the Fibonacci words in a shifts-order shown in figure 8.4-C. With transitions that are not shifts more than one bit is changed in general. The function used is [FXT: comb/shift-subsets-demo.cc]:

```cpp
void B(ulong x) {
  if (x >= N) return;
  visit(x);
  B(2*x);
  B(4*x+1);
}
```

An bit-level algorithm for combinations in shifts-order is given in section 1.25.3 on page 70.

### 8.5 $k$-subsets where $k$ lies in a given range

We give algorithms for generating all $k$-subsets of the $n$-set where $k$ lies in the range $k_{\text{min}} \leq k \leq k_{\text{max}}$. If $k_{\text{min}} = 0$ and $k_{\text{max}} = n$ we obtain all subsets, if $k_{\text{min}} = k_{\text{max}} = k$ we obtain $k$-combinations.

#### 8.5.1 Recursive algorithm

A generator for all $k$-subsets where lies in a prescribed range is [FXT: class ksubset_rec in comb/ksubset-rec.h]. The used algorithm can generate the subsets in 16 different orders. Figure 8.5-A shows the lexicographic orders, figure 8.5-B shows three Gray codes. The constructor has just one argument, the number of elements of the set whose subsets shall be generated:

```cpp
class ksubset_rec {
  // k-subsets where kmin<=k<=kmax in various orders.
  // Recursive CAT algorithm.
  public:
  long n_; // subsets of a n-element set
  long kmin_, kmax_; // k-subsets where kmin<=k<=kmax
  long *rv_; // record of visits in graph (list of elements in subset)
  ulong ct_; // count subsets
  ulong rct_; // count recursions (==work)
  ulong rct_; // condition that determines the order
  ulong pq_; // condition that determines the (printing) order
  ulong nq_; // whether to reverse order
  // function to call with each combination:
  void (*visit_)(const ksubset_rec &, long);
  public:
  ksubset_rec(ulong n) {
    n_ = n;
  }
}
```

[fxtbook draft of 2008-August-17]
8.5: \( k \)-subsets where \( k \) lies in a given range

<table>
<thead>
<tr>
<th>order #0:</th>
<th>order #8:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: { 0, 1 }</td>
<td>{ 0, 1, 2 }</td>
</tr>
<tr>
<td>1: { 0, 1, 2 }</td>
<td>{ 0, 1, 3 }</td>
</tr>
<tr>
<td>2: { 0, 1, 3 }</td>
<td>{ 0, 1, 4 }</td>
</tr>
<tr>
<td>3: { 0, 1, 4 }</td>
<td>{ 0, 1, 5 }</td>
</tr>
<tr>
<td>4: { 0, 1, 5 }</td>
<td>{ 0, 1 }</td>
</tr>
<tr>
<td>5: { 0, 2 }</td>
<td>{ 0, 2, 3 }</td>
</tr>
<tr>
<td>6: { 0, 2, 3 }</td>
<td>{ 0, 2, 4 }</td>
</tr>
<tr>
<td>7: { 0, 2, 4 }</td>
<td>{ 0, 2, 5 }</td>
</tr>
<tr>
<td>8: { 0, 2, 5 }</td>
<td>{ 0, 2 }</td>
</tr>
<tr>
<td>9: { 0, 3 }</td>
<td>{ 0, 3, 4 }</td>
</tr>
<tr>
<td>10: { 0, 3, 4 }</td>
<td>{ 0, 3, 5 }</td>
</tr>
<tr>
<td>11: { 0, 3, 5 }</td>
<td>{ 0, 3 }</td>
</tr>
<tr>
<td>12: { 0, 4 }</td>
<td>{ 0, 4, 5 }</td>
</tr>
<tr>
<td>13: { 0, 4, 5 }</td>
<td>{ 0, 4 }</td>
</tr>
<tr>
<td>14: { 0, 5 }</td>
<td>{ 0, 5 }</td>
</tr>
<tr>
<td>15: { 1, 2 }</td>
<td>{ 1, 2, 3 }</td>
</tr>
<tr>
<td>16: { 1, 2, 3 }</td>
<td>{ 1, 2, 4 }</td>
</tr>
<tr>
<td>17: { 1, 2, 4 }</td>
<td>{ 1, 2, 5 }</td>
</tr>
<tr>
<td>18: { 1, 2, 5 }</td>
<td>{ 1, 2 }</td>
</tr>
<tr>
<td>19: { 1, 3 }</td>
<td>{ 1, 3, 4 }</td>
</tr>
<tr>
<td>20: { 1, 3, 4 }</td>
<td>{ 1, 3, 5 }</td>
</tr>
<tr>
<td>21: { 1, 3, 5 }</td>
<td>{ 1, 3 }</td>
</tr>
<tr>
<td>22: { 1, 4 }</td>
<td>{ 1, 4, 5 }</td>
</tr>
<tr>
<td>23: { 1, 4, 5 }</td>
<td>{ 1, 4 }</td>
</tr>
<tr>
<td>24: { 1, 5 }</td>
<td>{ 1, 5 }</td>
</tr>
<tr>
<td>25: { 2, 3 }</td>
<td>{ 2, 3, 4 }</td>
</tr>
<tr>
<td>26: { 2, 3, 4 }</td>
<td>{ 2, 3, 5 }</td>
</tr>
<tr>
<td>27: { 2, 3, 5 }</td>
<td>{ 2, 3 }</td>
</tr>
<tr>
<td>28: { 2, 4 }</td>
<td>{ 2, 4, 5 }</td>
</tr>
<tr>
<td>29: { 2, 4, 5 }</td>
<td>{ 2, 4 }</td>
</tr>
<tr>
<td>30: { 2, 5 }</td>
<td>{ 2, 5 }</td>
</tr>
<tr>
<td>31: { 3, 4 }</td>
<td>{ 3, 4, 5 }</td>
</tr>
<tr>
<td>32: { 3, 4, 5 }</td>
<td>{ 3, 4 }</td>
</tr>
<tr>
<td>33: { 3, 5 }</td>
<td>{ 3, 5 }</td>
</tr>
<tr>
<td>34: { 4, 5 }</td>
<td>{ 4, 5 }</td>
</tr>
</tbody>
</table>

Figure 8.5-A: The \( k \)-subsets (where \( 2 \leq k \leq 3 \)) of a 6-element set. Lexicographic order for sets (left) and reversed lexicographic order for delta sets (right).

```c
rv_ = new long[n_+1];
++rv_; 
rv_[-1] = -1UL;
}

~ksubset_rec()
{
  --rv_
  delete [] rv_
}
```

One has to supply the interval for \( k \) (variables \( kmin \) and \( kmax \)) and a function that will be called with each subset. The argument \( rq \) determines which of the sixteen different orderings is chosen, the order can be reversed with nonzero \( nq \).

```c
void generate(void (*visit)(const ksubset_rec & k, long),
              long kmin, long kmax, ulong rq, ulong nq=0)
{
  ct_ = 0;
  rct_ = 0;
  kmin_ = kmin;
  kmax_ = kmax;
  if (kmax_ > kmin_ ) swap2(kmin_, kmax_);
  if (kmax_ > n_ ) kmax_ = n_;
  if (kmin_ > n_ ) kmin_ = n_;
  visit_ = visit;
  rq_ = rq % 4;
  pq_ = (rq>>2) % 4;
```
Chapter 8: Subsets

Figure 8.5-B: Three minimal-change orders of the $k$-subsets (where $2 \leq k \leq 3$) of a 6-element set.

<table>
<thead>
<tr>
<th>order #6:</th>
<th>order #7:</th>
<th>order #10:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 1...1..</td>
<td>11...1...</td>
<td>11...1...</td>
</tr>
<tr>
<td>1: 1...1..</td>
<td>11...1...</td>
<td>1...1...1</td>
</tr>
<tr>
<td>2: 1...1..</td>
<td>11...1...</td>
<td>1...1...1</td>
</tr>
<tr>
<td>3: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>4: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>5: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>6: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>7: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>8: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>9: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>10: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>11: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>12: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>13: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>14: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>15: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>16: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>17: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>18: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>19: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>20: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>21: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>22: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>23: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>24: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>25: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>26: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>27: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>28: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>29: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>30: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>31: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>32: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>33: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
<tr>
<td>34: 1...1..</td>
<td>1...1...1</td>
<td>1...1...1</td>
</tr>
</tbody>
</table>

Figure 8.5-C: With $k_{\text{min}} = 0$ and order number seven at each transition either one element is added or removed, or one element moves to an adjacent position.

[fxtbook draft of 2008-August-17]
8.5: \(k\)-subsets where \(k\) lies in a given range

The recursive routine itself is given in \[FXT: \text{comb/ks subset-rec.cc}\]:

```cpp
void
ksubset_rec::next_rec(long d)
{
   if ( d>kmax_ ) return;
   ++rct_; // measure computational work
   long rv1 = rv_[d-1]; // left neighbor
   bool q;
   switch ( rq_ % 4 )
   {
      case 0: q = 1; break;
      case 1: q = !(d&1); break;
      case 2: q = rv1&1; break;
      case 3: q = (d^rv1)&1; break;
   }
   if ( nq_ ) q = !q;
   long x0 = rv1 + 1;
   long rx = n_ - (kmin_ - d);
   long x1 = min2( n_-1, rx );
   #define PCOND(x) if ( (pq_==x) && (d>=kmin_) ) { visit_(*this, d); ++ct_; }
   PCOND(0);
   if ( q ) // forward:
      { PCOND(1);
         for (long x=x0; x<=x1; ++x) { rv_[d] = x; next_rec(d+1); }
      }
   else // backward:
      { PCOND(2);
         for (long x=x1; x>=x0; --x) { rv_[d] = x; next_rec(d+1); }
      }
   PCOND(3);
   #undef PCOND
}
```

About 50 million subsets per second are generated \[FXT: \text{comb/ks subset-rec-demo.cc}\].

8.5.2 Iterative algorithm for a minimal-change order

A generator for subsets in Gray code order is \[FXT: \text{class ksubset_gray in comb/ks subset-gray.h}\]:

```cpp
class ksubset_gray
{
   public:
      ulong n_; // \(k\)-subsets of \(\{1, 2, \ldots, n\}\)
      ulong kmin_, kmax_; // \(1\leq k \leq n\)
      ulong j_; // aux
   public:
      ksubset_gray(ulong n, ulong kmin, ulong kmax)
      {
         if ( n > 0 ) n_ = 1;
         // Must have 1\leq k \leq n
         kmin_ = kmin;
         kmax_ = kmax;
         if ( kmax_ < kmin_ ) swap2(kmin_, kmax_);
   }
}
```

[\text{fxtbook draft of 2008-August-17}]
Figure 8.5-D: The \((25)\) \(k\)-subsets where \(2 \leq k \leq 4\) of a five-element set in a minimal-change order.

```cpp
// [fxtbook draft of 2008-August-17]
```

The routines for computing the next or previous subset are adapted from a routine to compute the
successor given in [150]. It is split into two auxiliary functions:

```cpp
private:
    void prev_even()
    {
        ulong &n=n_, &kmin=kmin_, &kmax=kmax_, &j=j_; 
        if ( S_[j-1] == S_[j] ) // can touch sentinel S[0]
            { S_[j+1] = S_[j];
        if ( j > kmin )
            {
                if ( S_[kmin] == n ) { j = j-2; } else { j = j-1; }
            } else
            { 
                S_[j] = n - kmin + j;
        if ( S_[j-1]==S_[j]-1 ) { j = j-2; }
            }
        else
            { 
            }
        else
            { 
                S_[j] = S_[j] - 1;
                if ( j < kmax )
                { S_[j+1] = S_[j] + 1;
                    if ( j >= kmin-1 ) { j = j+1; } else { j = j+2; } 
                }
            }
    }
    void prev_odd()
    { 
    }
    }
```

The `next()` and `prev()` functions use these routines, note that calls cannot not be mixed.

```cpp
ulong prev()
{ 
    if ( is_first() ) { last(); return 0; }
    if ( j&1 ) prev_odd();
    else prev_even();
    if ( j<kmin_ ) { k_ = kmin_; } else { k_ = j_; };
    return k_;
}
```

```cpp
ulong next()
{ 
    if ( is_last() ) { first(); return 0; }
    if ( j&1 ) prev_even();
    else prev_odd();
    if ( j<kmin_ ) { k_ = kmin_; } else { k_ = j_; };
    return k_;
}
```
Usage of the class is shown in the program [FXT: \texttt{comb/ksubset-gray-demo.cc}], the \( k \)-subsets where \( 2 \leq k \leq 4 \) in the order generated by the algorithm are shown in figure 8.5-D. About 150 million subsets per second can be generated with the routine \texttt{next()}, and 130 million with \texttt{prev()}.

### 8.5.3 A two-close order with homogenous moves

<table>
<thead>
<tr>
<th>delta set</th>
<th>diff</th>
<th>set</th>
<th>delta set</th>
<th>diff</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: .1111</td>
<td>.....</td>
<td>{ 1, 2, 3, 4 }</td>
<td>1: ....11</td>
<td>.....</td>
<td>{ 4, 5 }</td>
</tr>
<tr>
<td>2: .1111</td>
<td>.M...</td>
<td>{ 2, 3, 4 }</td>
<td>2: ....11</td>
<td>.PM.</td>
<td>{ 3, 5 }</td>
</tr>
<tr>
<td>3: 1.111</td>
<td>P....</td>
<td>{ 0, 2, 3, 4 }</td>
<td>3: 1....1</td>
<td>.P.M.</td>
<td>{ 1, 5 }</td>
</tr>
<tr>
<td>4: 1.111</td>
<td>.PM...</td>
<td>{ 0, 1, 3, 4 }</td>
<td>4: ....11</td>
<td>.M....</td>
<td>{ 5 }</td>
</tr>
<tr>
<td>5: 1.111</td>
<td>M....</td>
<td>{ 1, 3, 4 }</td>
<td>5: 1....1</td>
<td>P....</td>
<td>{ 0, 5 }</td>
</tr>
<tr>
<td>6: ...11</td>
<td>.M...</td>
<td>{ 3, 4 }</td>
<td>6: .1....</td>
<td>M.P..</td>
<td>{ 2, 5 }</td>
</tr>
<tr>
<td>7: 1.11</td>
<td>P....</td>
<td>{ 0, 3, 4 }</td>
<td>7: .11..</td>
<td>...P.M</td>
<td>{ 2, 3 }</td>
</tr>
<tr>
<td>8: 11.1</td>
<td>.P.M...</td>
<td>{ 0, 1, 4 }</td>
<td>8: 11.1.</td>
<td>...PM..</td>
<td>{ 1, 3 }</td>
</tr>
<tr>
<td>9: 1.11</td>
<td>M....</td>
<td>{ 1, 4 }</td>
<td>9: 1....1</td>
<td>M....</td>
<td>{ 3 }</td>
</tr>
<tr>
<td>10: 1...1</td>
<td>PM...</td>
<td>{ 0, 4 }</td>
<td>10: 1....1</td>
<td>P....</td>
<td>{ 0, 3 }</td>
</tr>
<tr>
<td>11: 1....1</td>
<td>MP...</td>
<td>{ 2, 4 }</td>
<td>11: 11....</td>
<td>.P.M..</td>
<td>{ 0, 1 }</td>
</tr>
<tr>
<td>12: 1.11</td>
<td>P....</td>
<td>{ 0, 2, 4 }</td>
<td>12: .1....</td>
<td>M....</td>
<td>{ 1 }</td>
</tr>
<tr>
<td>13: 11.1</td>
<td>MP...</td>
<td>{ 1, 2, 4 }</td>
<td>13: 1....1</td>
<td>PM...</td>
<td>{ 0 }</td>
</tr>
<tr>
<td>14: 111.1</td>
<td>P....</td>
<td>{ 0, 1, 2, 4 }</td>
<td>14: .1....</td>
<td>M.P..</td>
<td>{ 2 }</td>
</tr>
<tr>
<td>15: 1111.</td>
<td>...PM</td>
<td>{ 0, 1, 2, 3 }</td>
<td>15: 11....</td>
<td>P.....</td>
<td>{ 0, 2 }</td>
</tr>
<tr>
<td>16: 1111.</td>
<td>M....</td>
<td>{ 1, 2, 3 }</td>
<td>16: 11....</td>
<td>MP...</td>
<td>{ 1, 2 }</td>
</tr>
<tr>
<td>17: 111.1</td>
<td>M....</td>
<td>{ 2, 3 }</td>
<td>17: 11....</td>
<td>MP...</td>
<td>{ 1, 4 }</td>
</tr>
<tr>
<td>18: 111.</td>
<td>P....</td>
<td>{ 0, 2, 3 }</td>
<td>18: ....11</td>
<td>M....</td>
<td>{ 4 }</td>
</tr>
<tr>
<td>19: 111.</td>
<td>.PM...</td>
<td>{ 0, 1, 3 }</td>
<td>19: 11....</td>
<td>P.....</td>
<td>{ 0, 4 }</td>
</tr>
<tr>
<td>20: 11.1</td>
<td>M....</td>
<td>{ 1, 3 }</td>
<td>20: 11....</td>
<td>M.P..</td>
<td>{ 2, 4 }</td>
</tr>
<tr>
<td>21: 11.1</td>
<td>PM...</td>
<td>{ 0, 3 }</td>
<td>21: 11....</td>
<td>..MP..</td>
<td>{ 3, 4 }</td>
</tr>
<tr>
<td>22: 11....</td>
<td>.P.M</td>
<td>{ 0, 1 }</td>
<td>22: .1..1</td>
<td>M.P...</td>
<td>{ 1 }</td>
</tr>
<tr>
<td>23: 11....</td>
<td>.MP</td>
<td>{ 0, 2 }</td>
<td>23: ..11.</td>
<td>M.P...</td>
<td>{ 2 }</td>
</tr>
<tr>
<td>24: 11....</td>
<td>MP</td>
<td>{ 1, 2 }</td>
<td>24: 11....</td>
<td>MP...</td>
<td>{ 1, 2 }</td>
</tr>
<tr>
<td>25: 111....</td>
<td>P...</td>
<td>{ 0, 1, 2 }</td>
<td>25: 111....</td>
<td>.P.M..</td>
<td>{ 0, 1 }</td>
</tr>
</tbody>
</table>

Figure 8.5-E: The \( k \)-subsets where \( 2 \leq k \leq 4 \) of 5 elements (left) and the sets where \( 1 \leq k \leq 2 \) of 6 elements (right) in two-close orders.

Orderings of the \( k \)-subsets with \( k \) in a given range that are two-close are shown in figure 8.5-E: one element is inserted or deleted or moves by at most two positions. The moves by two positions only cross a zero, the changes are homogenous. The list was produced with the program [FXT: \texttt{comb/ksubset-twoclose-demo.cc}] which uses [FXT: \texttt{class ksubset_twoclose in comb/ksubset-twoclose.h}]:

```cpp
1 class ksubset_twoclose
2 { // k-subsets (kmin<=k<=kmax) in a two-close order.
3   // Recursive algorithm.
4   public:
5     ulong *rv_; // record of visits in graph (delta set)
6     ulong n_; // subsets of the n-element set
7     // function to call with each combination:
8     void (*visit_)(const ksubset_twoclose &);
9     [--snip--]
10    void generate(void (*visit)(const ksubset_twoclose &),
11       ulong kmin, ulong kmax)
12    { 
13      visit_ = visit;
14      ulong kmax0 = n_ - kmin;
15      next_rec(n_, kmax, kmax0, 0);
16    }
17
18  }
```

The recursion is:

```cpp
1 private:
2  void next_rec(ulong d, ulong n1, ulong n0, bool q)
3  { // d: remaining depth in recursion
4    // n1: remaining ones to fill in
5    // n0: remaining zeros to fill in
6  }
```

[fxtbook draft of 2008-August-17]
8.5: \( k \)-subsets where \( k \) lies in a given range

```c
// q: direction in recursion
{
    if (0==d) { visit_(*this); return; }
    --d;
    if (q)
    {
        if (n0) { rv_[d]=0; next_rec(d, n1-0, n0-1, d&1); }
        if (n1) { rv_[d]=1; next_rec(d, n1-1, n0-0, q); }
    }
    else
    {
        if (n1) { rv_[d]=1; next_rec(d, n1-1, n0-0, q); }
        if (n0) { rv_[d]=0; next_rec(d, n1-0, n0-1, d&1); }
    }
};
```

About 75 million subsets per second can be generated. For \( k_{\text{min}} = k_{\text{max}} = k \) we obtain the enum order for combinations described in section 6.5.2 on page 188.
Chapter 9

Mixed radix numbers

The mixed radix representation $A = [a_0, a_1, a_2, \ldots, a_{n-1}]$ of a number $x$ with respect to a radix vector $M = [m_0, m_1, m_2, \ldots, m_{n-1}]$ is given by the unique expression

$$x = \sum_{k=0}^{n-1} a_k \prod_{j=0}^{k-1} m_j$$

(9.0-1)

where $0 \leq a_j < m_j$ (and $0 < x < \prod_{j=0}^{n-1} m_j$, so that $n$ digits suffice). For $M = [r, r, r, \ldots, r]$ the relation reduces to the radix-$r$ representation:

$$x = \sum_{k=0}^{n-1} a_k r^k$$

(9.0-2)

All 3-digit radix-4 numbers are shown in various orders in figure 9.0-A. Note that the least significant digit ($a_0$) is at the left side of each number (array representation).

9.1 Counting (lexicographic) order

An implementation for mixed radix counting is [FXT: class mixedradix_lex in comb/mixedradix-lex.h]:

```cpp
class mixedradix_lex
{
    public:
        ulong *a_; // digits
        ulong *m1_; // radix (minus one) for each digit
        ulong n_; // Number of digits
        ulong j_; // position of last change

    public:
        mixedradix_lex(const ulong *m, ulong n, ulong mm=0)
        {
            n_ = n;
            a_ = new ulong[n+1];
            m1_ = new ulong[n+1];
            a_[n_] = 1; // sentinel: !=0, and !=m1[n]
            m1_[n_] = 0; // sentinel
            mixedradix_init(n_, mm, m, m1_);
            first();
        }

        [--snip--]
}
```

The initialization routine mixedradix_init() is given in [FXT: comb/mixedradix-init.cc]:

[fxtbook draft of 2008-August-17]
<table>
<thead>
<tr>
<th></th>
<th>counting</th>
<th>Gray</th>
<th>modular Gray</th>
<th>gslex</th>
<th>endo</th>
<th>endo Gray</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>[ . . ]</td>
<td>[ . . ]</td>
<td>[ . . ]</td>
<td>[1 . ]</td>
<td>[ . . ]</td>
<td>[1 . ]</td>
</tr>
<tr>
<td>1:</td>
<td>[2 . ]</td>
<td>[2 . ]</td>
<td>[2 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>2:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[2 . ]</td>
<td>[2 . ]</td>
</tr>
<tr>
<td>3:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[2 . ]</td>
<td>[2 . ]</td>
</tr>
<tr>
<td>4:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[2 . ]</td>
<td>[2 . ]</td>
</tr>
<tr>
<td>5:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[2 . ]</td>
<td>[2 . ]</td>
</tr>
<tr>
<td>6:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>7:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>8:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>9:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>10:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>11:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>12:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>13:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>14:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>15:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>16:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>17:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>18:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>19:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>20:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>21:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>22:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>23:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>24:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>25:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>26:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>27:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>28:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>29:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>30:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>31:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>32:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>33:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>34:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>35:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>36:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>37:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>38:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>39:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>40:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>41:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>42:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>43:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>44:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>45:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>46:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>47:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>48:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>49:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>50:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>51:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>52:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>53:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>54:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>55:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>56:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>57:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>58:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>59:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>60:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>61:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>62:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
<tr>
<td>63:</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
<td>[3 . ]</td>
</tr>
</tbody>
</table>

Figure 9.0-A: All 3-digit, radix-4 numbers in various orders (dots denote zeros): counting-, Gray-, modular Gray-, gslex-, endo-, and endo Gray order. The least significant digit is on the left of each word (array notation).
9.1: Counting (lexicographic) order

0: [ . . . ]  [ . . . ]
1: [ 1 . . ]  [ 1 . . ]
2: [ . 1 . ]  [ 2 . . ]
3: [ 1 1 . ]  [ 3 . . ]
4: [ . 2 . ]  [ . 1 . ]
5: [ 1 2 . ]  [ 1 1 . ]
6: [ . 1 1 ]  [ 2 1 . ]
7: [ 1 1 1 ]  [ 3 . 1 ]
8: [ . 2 . ]  [ . 1 1 ]
9: [ 1 2 . ]  [ 1 1 1 ]
10: [ . 2 1 ]  [ 2 2 . ]
11: [ 1 2 1 ]  [ 3 2 . ]
12: [ . . 1 ]  [ . . 1 ]
13: [ 1 . 2 ]  [ 1 . 1 ]
14: [ . 1 2 ]  [ 2 . 1 ]
15: [ 1 1 2 ]  [ 3 1 1 ]
16: [ . 2 2 ]  [ 2 1 1 ]
17: [ 1 2 2 ]  [ 3 2 1 ]
18: [ . . 3 ]  [ 2 1 1 ]
19: [ 1 . 3 ]  [ 3 1 1 ]
20: [ . 1 3 ]  [ . 2 1 ]
21: [ 1 1 3 ]  [ 1 2 1 ]
22: [ . 2 3 ]  [ 2 2 1 ]
23: [ 1 2 3 ]  [ 3 2 1 ]

Figure 9.1-A: Mixed radix numbers in counting (lexicographic) order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (rising factorial basis, left) and $M = [4, 3, 2]$ (falling factorial basis, right). The least significant digit is on the left of each word (array notation).

```c

1 void mixedradix_init(ulong n, ulong mm, const ulong *m, ulong *m1)
2 {  // Auxiliary function used to initialize vector of nines in mixed radix classes.
3     if (m) {  // all radices given
4         for (ulong k=0; k<n; ++k) m1[k] = m[k] - 1;
5     } else {
6         if (mm>1) {  // use mm as radix for all digits:
7             for (ulong k=0; k<n; ++k) m1[k] = mm - 1;
8         } else {
9             if (mm==0) {  // falling factorial basis
10                for (ulong k=0; k<n; ++k) m1[k] = n - k;
11            } else {  // falling factorial basis
12                for (ulong k=0; k<n; ++k) m1[k] = k + 1;
13            }
14         }
15     }
16 }

Instead of the vector of radices $M = [m_0, m_1, m_2, \ldots, m_{n-1}]$ the vector of ‘nines’ ($M' = [m_0 - 1, m_1 - 1, m_2 - 1, \ldots, m_{n-1} - 1]$), variable $m_1$. This modification leads to slightly faster generation. The first $n$-digit in lexicographic order number is all-zero, the last is all-nines:

```c

```
and incrementing the next digit:

```cpp
bool next() // increment
{
    ulong j = 0;
    while (a_[j]==m1_[j]) { a_[j]=0; ++j; } // can touch sentinels
    j_ = j;
    if (j==n_) return false; // current is last
    ++a_[j];
    return true;
}
```

A number is decremented by setting all zero digits at the lower end to nine and decrementing the next digit:

```cpp
bool prev() // decrement
{
    ulong j = 0;
    while (a_[j]==0) { a_[j]=m1_[j]; ++j; } // can touch sentinels
    j_ = j;
    if (j==n_) return false; // current is first
    --a_[j];
    return true;
}
```

Figure 9.1-A shows the 3-digit mixed radix numbers for basis vector $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). The listings where created with the program [FXT: comb/mixedradix-lex-demo.cc].

The routine `next()` generates between about 166 million (radix-2 numbers, $M = [2, 2, 2, \ldots, 2]$), 257 million (radix-3), and about 370 million (radix-8) numbers per second. Note that radix-2 leads to the slowest generation as the average carries are long compared to higher radices. The number of carries with incrementing is on average:

$$
C = \frac{1}{m_0} \left( \frac{1}{m_1} \left( \frac{1}{m_2} (\ldots) \right) \right) = \sum_{k=0}^{n} \frac{1}{\prod_{j=0}^{k} m_j}
$$

The number of digits changed on average equals $C + 1$. For $M = [r, r, r, \ldots, r]$ (and $n = \infty$) we obtain $C = \frac{1}{r-1}$. For the worst case ($r = 2$) we have $C = 1$, so two digits are changed on average.

### 9.2 Gray code order

#### 9.2.1 Constant amortized time (CAT) algorithm

Figure 9.2-A shows the 3-digit mixed radix numbers for radix vectors $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right) in Gray code order. A generator for the Gray code order is [FXT: class mixedradix_gray in comb/mixedradix-gray.h]:

```cpp
class mixedradix_gray
{
    public:
        ulong *a_; // mixed radix digits
        ulong *m1_; // radices (minus one)
        ulong *i_; // direction
        ulong n_; // n digits
        ulong j_; // position of last change
        int dm_; // direction of last move
    public:
        mixedradix_gray(const ulong *m, ulong n, ulong mm=0)
        {
```
9.2: Gray code order

<table>
<thead>
<tr>
<th>M = [2, 3, 4]</th>
<th>x</th>
<th>j</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>1:</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3:</td>
<td>.</td>
<td>1</td>
<td>.</td>
</tr>
<tr>
<td>4:</td>
<td>2</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>5:</td>
<td>.</td>
<td>2</td>
<td>.</td>
</tr>
<tr>
<td>6:</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7:</td>
<td>.</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9:</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10:</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>11:</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12:</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>13:</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>14:</td>
<td>.</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15:</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>16:</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>17:</td>
<td>.</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>18:</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19:</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>20:</td>
<td>.</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>21:</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>22:</td>
<td>.</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>23:</td>
<td>.</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M = [4, 3, 2]</th>
<th>x</th>
<th>j</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>1:</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2:</td>
<td>.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3:</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4:</td>
<td>.</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5:</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6:</td>
<td>.</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8:</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10:</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11:</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12:</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13:</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14:</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15:</td>
<td>.</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>16:</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>17:</td>
<td>.</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>18:</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>19:</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>20:</td>
<td>.</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>21:</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>22:</td>
<td>.</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>23:</td>
<td>.</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 9.2-A: Mixed radix numbers in Gray code order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Columns ‘x’ give the values, columns ‘j’ and ‘d’ give the position of last change and its direction, respectively.

```c
n_ = n;
a_ = new ulong[n_+1];
a_[n_] = -1UL; // sentinel
i_ = new ulong[n_+1];
i_[n_] = 0; // sentinel
m1_ = new ulong[n_+1];
mixedradix_init(n_, mm, m, m1_);
first();
```

The array $i_[]$ contains the ‘directions’ for each digits: it contains +1 or -1 if the computation of the successor will increase or decrease the corresponding digit. It has to be filled when the first or last number is computed:

```c
void first()
{
  for (ulong k=0; k<n_; ++k) a_[k] = 0;
  for (ulong k=0; k<n_; ++k) i_[k] = +1;
  j_ = n_;
  dm_ = 0;
}

void last()
{
  // find position of last even radix:
  ulong z = 0;
  for (ulong i=0; i<n_; ++i) if (m1_[i] & 1) z = i;
  while (z<n_) // last even .. end:
    {  
a_[z] = m1_[z];
    i_[z] = +1;
    ++z;
    }
  j_ = 0;
  dm_ = -1;
}
```
A sentinel element \((i[n]=0)\) is used to optimize the computations of the successor and predecessor. The method is a constant amortized time (CAT) implementation.

```cpp
bool next()
{
    ulong j = 0;
    ulong ij;
    while ( (ij=i_[j]) ) // can touch sentinel i[n]=0
    {
        ulong dj = a_[j] + ij;
        if ( dj>m1_[j] ) // ^= if ( (dj>m1_[j]) || ((long)dj<0) )
        {
            i_[j] = -ij; // flip direction
        }
        else // can update
        {
            a_[j] = dj; // update digit
            dm_ = ij; // save for dir()
            i_ = j; // save for pos()
            return true;
        }
        ++j;
    }
    return false;
}
```

Note the if-clause, it is an optimized expression equivalent to the one given as comment. The following methods are often useful:

```cpp
ulong pos() const { return j_; } // position of last change
int dir() const { return dm_; } // direction of last change
```

The routine for the computation of the predecessor is obtained by changing the statement \(ulong dj = a_[j] + ij\) to \(ulong dj = a_[j] - ij\). The rate of generation is about 128 M/s for radix 2, and 243 M/s for radix 4, and 304 M/s for radix 8 [FXT: comb/mixedradix-gray-demo.cc].

### 9.2.2 Loopless algorithm

A loopless algorithm for the computation of the successor, taken from [174], is given in [FXT: comb/mixedradix-gray2.h]. The crucial trick to make the algorithm loopless is the use of ‘focus pointers’:

```cpp
class mixedradix_gray2
{
public:
    ulong *a_; // digits
    ulong *ml_; // radix minus one (‘nines’)  
    ulong *f_; // focus pointer
    ulong *d_; // direction
    ulong n_; // number of digits
    int dm_; // position of last change
    int j_; // direction of last move
    [--snip--]

void first()
{
    for (ulong k=0; k<n_; ++k) a_[k] = 0;
    for (ulong k=0; k<n_; ++k) d_[k] = 1;
    for (ulong k=0; k<n_; ++k) f_[k] = k;
    dm_ = 0;
    j_ = n_;
}

bool next()
{
    const ulong j = f_[0];
    f_[0] = 0;
    if ( j>=n_ ) { first(); return false; }
    const ulong dj = d_[j];
    const ulong aj = a_[j] + dj;
    [--snip--]
```
9.2.3 Modular Gray code order

Figure 9.2-B: Mixed radix numbers in modular Gray code order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). The columns ‘j’ give the position of last change.

Figure 9.2-B shows the modular Gray code order for 3-digit mixed radix numbers with radix vectors $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). The transitions are either $k \rightarrow k + 1$ or, if $k$ is maximal, $k \rightarrow 0$. The loopless implementation [FXT: class mixedradix_modular_gray in comb/mixedradix-modular-gray.h] was taken from [174]. The rate of generation is about 169 M/s with radix 2, 197 M/s with radix 4, and 256 M/s with radix 16 [FXT: comb/mixedradix-modular-gray-demo.cc].

A different algorithm is used in [FXT: class mixedradix_modular_gray2 in comb/mixedradix-modular-gray2.h]:

```cpp
1  class mixedradix_modular_gray2
2  {
3    public:
4      ulong *a_; // digits
5      ulong *m1_; // radix minus one ('nines')
6      ulong *x_; // count changes of digit
7      ulong n_; // number of digits
8      ulong j_; // position of last change
```
Chapter 9: Mixed radix numbers

10 public:
11  mixedradix_modular_gray2(ulong n, ulong mm, const ulong *m=0)
12  {
13    n_ = n;
14    a_ = new ulong[n_];
15    m1_ = new ulong[n_+1]; // incl. sentinel at m1[n]
16    x_ = new ulong[n_+1]; // incl. sentinel at x[n] (!= m1[n])
17    mixedradix_init(n_, mm, m, m1_);
18    first();
19  }
20  [--snip--]

The computation of the successor is constant amortized time

1  bool next()
2  {
3    ulong j = 0;
4    while ( x_[j] == m1_[j] ) // can touch sentinels
5      { x_[j] = 0;
6        ++j;
7      }
8    ++x_[j];
9    if ( j==n_ ) { first(); return false; } // current is last
10   j_ = j; // save position of change
11   // increment:
12   ulong aj = a_[j] + 1;
13   if ( aj>m1_[j] ) aj = 0;
14   a_[j] = aj;
15   return true;
16  }
17  [--snip--]

The rate of generation is about 151 M/s for radix 2, 254 M/s for radix 4, and 267 M/s for radix 8 [FXT: comb/mixedradix-modular-gray2-demo.cc].

9.3 gslex order

The algorithm for the generation of subsets in lexicographic order in set representation given in section 8.1.2 on page 202 can be generalized for mixed radix numbers. Figure 9.3-A shows the 3-digit mixed radix numbers for basis vector $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Note that zero is the last word in this order. For lack of a better name we call the order gslex (for generalized subset-lex) order. A generator for the gslex order is [FXT: class mixedradix gslex in comb/mixedradix-gslex.h]:

class mixedradix_gslex
{
    public:
        ulong n_;  // n-digit numbers
        ulong *a_; // digits
        ulong *m1_; // m1[k] == radix-1 at position k
    public:
        mixedradix_gslex(ulong n, ulong mm, const ulong *m=0)
        {
            n_ = n;
            a_ = new ulong[n_ + 1];
            a_[n_] = 1; // sentinel
            m1_ = new ulong[n_];
            mixedradix_init(n_, mm, m, m1_);
            first();
        }
        void first()
        {
            for (ulong k=0; k<n_; ++k) a_[k] = 0;
        }
    }
\textbf{Figure 9.3-A:} Mixed radix numbers in gslex (generalized subset lex) order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Successive words differ in at most three positions. Columns ‘x’ give the values.

<table>
<thead>
<tr>
<th>\textbf{x}</th>
<th>\textbf{x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: $[ 1 \ . \ . \ . \ 1 ]$</td>
<td>0: $[ 1 \ . \ . \ . \ 1 ]$</td>
</tr>
<tr>
<td>1: $[ 1 \ 1 \ . \ \ ]$</td>
<td>1: $[ 2 \ . \ . \ . \ 2 ]$</td>
</tr>
<tr>
<td>2: $[ . \ 1 \ . \ \ ]$</td>
<td>2: $[ 3 \ . \ . \ . \ 3 ]$</td>
</tr>
<tr>
<td>3: $[ 1 \ 2 \ . \ \ ]$</td>
<td>3: $[ 1 \ 1 \ . \ . \ 5 ]$</td>
</tr>
<tr>
<td>4: $[ 1 \ 1 \ . \ \ ]$</td>
<td>4: $[ 2 \ 1 \ . \ . \ 6 ]$</td>
</tr>
<tr>
<td>5: $[ . \ 1 \ . \ \ ]$</td>
<td>5: $[ 1 \ 2 \ . \ . \ 4 ]$</td>
</tr>
<tr>
<td>6: $[ 1 \ 2 \ . \ \ ]$</td>
<td>6: $[ 1 \ 1 \ . \ . \ 5 ]$</td>
</tr>
<tr>
<td>7: $[ 1 \ 1 \ \ ]$</td>
<td>7: $[ 1 \ 2 \ . \ . \ 9 ]$</td>
</tr>
<tr>
<td>8: $[ 1 \ 2 \ 1 \ \ ]$</td>
<td>8: $[ 2 \ 2 \ . \ . \ 10 ]$</td>
</tr>
<tr>
<td>9: $[ . \ 2 \ 1 \ \ ]$</td>
<td>9: $[ 3 \ 2 \ . \ . \ 11 ]$</td>
</tr>
<tr>
<td>10: $[ . \ 1 \ \ ]$</td>
<td>10: $[ 1 \ . \ . \ 8 ]$</td>
</tr>
<tr>
<td>11: $[ 1 \ 2 \ \ ]$</td>
<td>11: $[ 1 \ 1 \ \ ]$</td>
</tr>
<tr>
<td>12: $[ 1 \ 1 \ \ ]$</td>
<td>12: $[ 2 \ . \ . \ 14 ]$</td>
</tr>
<tr>
<td>13: $[ . \ 1 \ \ ]$</td>
<td>13: $[ 3 \ . \ . \ 15 ]$</td>
</tr>
<tr>
<td>14: $[ 1 \ 2 \ \ ]$</td>
<td>14: $[ 1 \ 1 \ \ ]$</td>
</tr>
<tr>
<td>15: $[ . \ 2 \ \ ]$</td>
<td>15: $[ 2 \ 1 \ \ ]$</td>
</tr>
<tr>
<td>16: $[ . \ 2 \ ]$</td>
<td>16: $[ 3 \ 1 \ \ ]$</td>
</tr>
<tr>
<td>17: $[ 1 \ . \ ]$</td>
<td>17: $[ 1 \ 1 \ ]$</td>
</tr>
<tr>
<td>18: $[ 1 \ 1 \ ]$</td>
<td>18: $[ 2 \ 1 \ ]$</td>
</tr>
<tr>
<td>19: $[ . \ 1 \ ]$</td>
<td>19: $[ 1 \ 2 \ ]$</td>
</tr>
<tr>
<td>20: $[ 1 \ 2 \ ]$</td>
<td>20: $[ 3 \ 2 \ ]$</td>
</tr>
<tr>
<td>21: $[ . \ 2 \ ]$</td>
<td>21: $[ . \ 2 \ ]$</td>
</tr>
<tr>
<td>22: $[ . \ 3 \ ]$</td>
<td>22: $[ . \ 1 \ ]$</td>
</tr>
<tr>
<td>23: $[ \ . \ ]$</td>
<td>23: $[ \ . \ ]$</td>
</tr>
</tbody>
</table>

The method \texttt{next()} computes the successor:

```c
bool next()
{
    ulong e = 0;
    while ( 0==a_[e] ) ++e; // can touch sentinel
    if ( e==n_ ) { first(); return false; } // current is last
    ulong ae = a_[e];
    if ( ae != m1_[e] ) // easy case: simple increment
    {
        a_[0] = 1;
        a_[e] = ae + 1;
    }
    else
    {
        a_[e] = 0;
        if ( a_[e+1]==0 ) // can touch sentinel
        {
            a_[0] = 1;
            ++a_[e+1];
        }
    }
    return true;
}
```

The predecessor is computed by the method \texttt{prev()}:

```c
bool prev()
{
    ulong e = 0;
    while ( 0==a_[e] ) ++e; // can touch sentinel
    if ( 0!=e ) // easy case: prepend nine
    {
    ```
---

228  
Chapter 9: Mixed radix numbers

8         --e;
9     a_[e] = m1_[e];
10 }
11 else
12 {
13     ulong a0 = a_[0];
14     --a0;
15     a_[0] = a0;
16     if ( 0==a0 )
17     {
18         do { ++e; } while ( 0==a_[e] ); // can touch sentinel
19         if ( e==n_ ) { last(); return false; } // current is first
20         ulong ae = a_[e];
21         --ae;
22         a_[e] = ae;
23         if ( 0==ae )
24             {             // touch sentinel
25                 a_[e] = m1_[e];
26             }
27     }
28 }
29 }
30 return true;
31 }
32

The algorithm is constant amortized time (CAT) and fast in practice. The worst performance occurs when all digits are radix 2, then about 123 million objects are created per second. With radix 4 the rate is about 198 M/s, with radix 16 about 273 M/s [FXT: comb/mixedradix-gslex-demo.cc].

Alternative gslex order

<table>
<thead>
<tr>
<th align="right">M1 = [2 3 4]</th>
<th align="right">x</th>
</tr>
</thead>
<tbody>
<tr>
<td align="right">0:</td>
<td align="right">[ . . . ] 0</td>
</tr>
<tr>
<td align="right">1:</td>
<td align="right">[ 1 . . ] 1</td>
</tr>
<tr>
<td align="right">2:</td>
<td align="right">[ 1 1 . ] 3</td>
</tr>
<tr>
<td align="right">3:</td>
<td align="right">[ 1 1 1 ] 9</td>
</tr>
<tr>
<td align="right">4:</td>
<td align="right">[ 1 1 2 ] 15</td>
</tr>
<tr>
<td align="right">5:</td>
<td align="right">[ 1 1 3 ] 21</td>
</tr>
<tr>
<td align="right">6:</td>
<td align="right">[ 1 2 . ] 5</td>
</tr>
<tr>
<td align="right">7:</td>
<td align="right">[ 1 2 1 ] 11</td>
</tr>
<tr>
<td align="right">8:</td>
<td align="right">[ 1 2 2 ] 17</td>
</tr>
<tr>
<td align="right">9:</td>
<td align="right">[ 1 2 3 ] 23</td>
</tr>
<tr>
<td align="right">10:</td>
<td align="right">[ 1 . 1 ] 7</td>
</tr>
<tr>
<td align="right">11:</td>
<td align="right">[ 1 . 2 ] 13</td>
</tr>
<tr>
<td align="right">12:</td>
<td align="right">[ 1 . 3 ] 19</td>
</tr>
<tr>
<td align="right">13:</td>
<td align="right">[ . 1 . ] 2</td>
</tr>
<tr>
<td align="right">14:</td>
<td align="right">[ . 1 1 ] 8</td>
</tr>
<tr>
<td align="right">15:</td>
<td align="right">[ . 1 2 ] 14</td>
</tr>
<tr>
<td align="right">16:</td>
<td align="right">[ . 1 3 ] 20</td>
</tr>
<tr>
<td align="right">17:</td>
<td align="right">[ . 2 . ] 4</td>
</tr>
<tr>
<td align="right">18:</td>
<td align="right">[ . 2 1 ] 10</td>
</tr>
<tr>
<td align="right">19:</td>
<td align="right">[ . 2 2 ] 16</td>
</tr>
<tr>
<td align="right">20:</td>
<td align="right">[ . 2 3 ] 22</td>
</tr>
<tr>
<td align="right">21:</td>
<td align="right">[ . . 1 ] 6</td>
</tr>
<tr>
<td align="right">22:</td>
<td align="right">[ . . 2 ] 12</td>
</tr>
<tr>
<td align="right">23:</td>
<td align="right">[ . . 3 ] 18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th align="right">M2 = [4 3 2]</th>
<th align="right">x</th>
</tr>
</thead>
<tbody>
<tr>
<td align="right">0:</td>
<td align="right">[ . . . ] 0</td>
</tr>
<tr>
<td align="right">1:</td>
<td align="right">[ 1 . . ] 1</td>
</tr>
<tr>
<td align="right">2:</td>
<td align="right">[ 1 1 . ] 5</td>
</tr>
<tr>
<td align="right">3:</td>
<td align="right">[ 1 1 1 ] 17</td>
</tr>
<tr>
<td align="right">4:</td>
<td align="right">[ 1 2 . ] 9</td>
</tr>
<tr>
<td align="right">5:</td>
<td align="right">[ 1 2 1 ] 21</td>
</tr>
<tr>
<td align="right">6:</td>
<td align="right">[ 1 . 1 ] 13</td>
</tr>
<tr>
<td align="right">7:</td>
<td align="right">[ 1 . 2 ] 2</td>
</tr>
<tr>
<td align="right">8:</td>
<td align="right">[ 1 . 3 ] 6</td>
</tr>
<tr>
<td align="right">9:</td>
<td align="right">[ 2 . 1 ] 18</td>
</tr>
<tr>
<td align="right">10:</td>
<td align="right">[ 2 . 2 ] 21</td>
</tr>
<tr>
<td align="right">11:</td>
<td align="right">[ 2 . 3 ] 23</td>
</tr>
<tr>
<td align="right">12:</td>
<td align="right">[ . 1 . ] 7</td>
</tr>
<tr>
<td align="right">13:</td>
<td align="right">[ . 1 1 ] 14</td>
</tr>
<tr>
<td align="right">14:</td>
<td align="right">[ . 1 2 ] 19</td>
</tr>
<tr>
<td align="right">15:</td>
<td align="right">[ . 1 3 ] 22</td>
</tr>
<tr>
<td align="right">16:</td>
<td align="right">[ . 2 . ] 15</td>
</tr>
<tr>
<td align="right">17:</td>
<td align="right">[ . 2 1 ] 4</td>
</tr>
<tr>
<td align="right">18:</td>
<td align="right">[ . 2 2 ] 16</td>
</tr>
<tr>
<td align="right">19:</td>
<td align="right">[ . 2 3 ] 22</td>
</tr>
<tr>
<td align="right">20:</td>
<td align="right">[ . . 1 ] 8</td>
</tr>
<tr>
<td align="right">21:</td>
<td align="right">[ . . 2 ] 20</td>
</tr>
<tr>
<td align="right">22:</td>
<td align="right">[ . . 3 ] 12</td>
</tr>
</tbody>
</table>

Figure 9.3-B: Mixed radix numbers in alternative gslex (generalized subset lex) order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Successive words differ in at most three positions. Columns ‘x’ give the values.

A variant of the gslex order is shown in figure 9.3-B. The ordering can be obtained from the gslex order by reversing the list, reversing the words, and replacing all nonzero digits $d_i$ by $r_i - d_i$ where $r_i$ is the radix at position $i$. The implementation is given in [FXT: class mixedradix_gslex_alt in comb/mixedradix-gslex-alt.h], the rate of generation is about the same as with gslex order [FXT: comb/mixedradix-gslex-alt-demo.cc].

[fxtbook draft of 2008-August-17]
9.4 endo order

Figure 9.4-A: Mixed radix numbers in endo order, dots denote zeros. The radix vector is \( M = [5, 6] \).

<table>
<thead>
<tr>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>M=</td>
<td>[ 5 6 ]</td>
</tr>
<tr>
<td>0:</td>
<td>[ . . ]</td>
</tr>
<tr>
<td>1:</td>
<td>[ 1 . ]</td>
</tr>
<tr>
<td>2:</td>
<td>[ 3 . ]</td>
</tr>
<tr>
<td>3:</td>
<td>[ 4 . ]</td>
</tr>
<tr>
<td>4:</td>
<td>[ 2 . ]</td>
</tr>
<tr>
<td>5:</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>6:</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>7:</td>
<td>[ 3 1 ]</td>
</tr>
<tr>
<td>8:</td>
<td>[ 4 1 ]</td>
</tr>
<tr>
<td>9:</td>
<td>[ 2 1 ]</td>
</tr>
<tr>
<td>10:</td>
<td>[ . 3 ]</td>
</tr>
<tr>
<td>11:</td>
<td>[ 1 3 ]</td>
</tr>
<tr>
<td>12:</td>
<td>[ 3 3 ]</td>
</tr>
<tr>
<td>13:</td>
<td>[ 4 3 ]</td>
</tr>
<tr>
<td>14:</td>
<td>[ 2 3 ]</td>
</tr>
<tr>
<td>15:</td>
<td>[ . 5 ]</td>
</tr>
<tr>
<td>16:</td>
<td>[ 1 5 ]</td>
</tr>
<tr>
<td>17:</td>
<td>[ 3 5 ]</td>
</tr>
<tr>
<td>18:</td>
<td>[ 4 5 ]</td>
</tr>
<tr>
<td>19:</td>
<td>[ 2 5 ]</td>
</tr>
<tr>
<td>20:</td>
<td>[ . 4 ]</td>
</tr>
<tr>
<td>21:</td>
<td>[ 1 4 ]</td>
</tr>
<tr>
<td>22:</td>
<td>[ 3 4 ]</td>
</tr>
<tr>
<td>23:</td>
<td>[ 4 4 ]</td>
</tr>
<tr>
<td>24:</td>
<td>[ 2 4 ]</td>
</tr>
<tr>
<td>25:</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>26:</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>27:</td>
<td>[ 3 2 ]</td>
</tr>
<tr>
<td>28:</td>
<td>[ 4 2 ]</td>
</tr>
<tr>
<td>29:</td>
<td>[ 2 2 ]</td>
</tr>
</tbody>
</table>

The computation of the successor in mixed radix endo order (see section 6.5.1 on page 186) is very similar to the counting order described section 9.1 on page 219. The implementation [FXT: class mixedradix_endo in comb/mixedradix-endo.h] uses an additional array \( le[] \) of the last nonzero elements in endo order. Its entries are 2 for \( m > 1 \), else 1:

```cpp
class mixedradix_endo
{
public:
ulong *a_; // digits, sentinel a[n]
ulong *m1_; // radix (minus one) for each digit
ulong *le_; // last positive digit in endo order, sentinel le[n]
ulong n_; // Number of digits
ulong j_; // position of last change
mixedradix_endo(const ulong *m, ulong n, ulong mm=0)
{
    n_ = n;
a_ = new ulong[n+1];
a_[n_] = 1; // sentinel: != 0
m1_ = new ulong[n_];
mixedradix_init(n_, mm, m, m1_);
le_ = new ulong[n+1];
le_[n_] = 0; // sentinel: != a[n]
for (ulong k=0; k<n_; ++k) le_[k] = 2 - (m1_[k]==1);
first();
}

void first()
{
    for (ulong k=0; k<n_; ++k) a_[k] = 0;
j_ = n_;
}

void last()
{
    for (ulong k=0; k<n_; ++k) a_[k] = le_[k];
j_ = n_;}

bool next()
{
```

In the computation of the successor the function next_endo() is used instead of a simple increment:
The function `next()` generates between about 115 million (radix 2) and 180 million (radix 16) numbers per second. The listing in figure 9.4-A was created with the program [FXT: comb/mixedradix-endo-demo.cc].

### 9.5 Gray code for endo order

A Gray code for mixed radix numbers in endo order can be obtained by a modification of the CAT algorithm for the Gray code described in section 9.2 on page 222. In the computation of the last number, the last digit have to be set to the last endo digit [FXT: class mixedradix_endo_gray in comb/mixedradix-endo-gray.h]:

```cpp
class mixedradix_endo_gray
{
public:
  ulong *a_; // mixed radix digits
  ulong *m1_; // radices (minus one)
};
```

The function `prev()` can be implemented in a similar way.

---

**Figure 9.5-A:** Mixed radix numbers in endo Gray code, dots denote zeros. The radix vector is $M = [4, 5]$. Columns ‘x’ give the values, columns ‘j’ and ‘d’ give the position of last change and its direction, respectively.
ulong *i_; // direction
ulong *le_; // last positive digit in endo order
ulong n_; // n_ digits
ulong j_; // position of last change
int dm_; // direction of last move

void first()
{
    for (ulong k=0; k<n_; ++k) a_[k] = 0;
    for (ulong k=0; k<n_; ++k) i_[k] = +1;
    j_ = n_;
    dm_ = 0;
}

void last()
{
    for (ulong k=0; k<n_; ++k) a_[k] = 0;
    for (ulong k=0; k<n_; ++k) i_[k] = -1UL;
    // find position of last even radix:
    ulong z = 0;
    for (ulong i=0; i<n_; ++i) if ( m1_[i]&1 ) z = i;
    while ( z<n_ ) // last even .. end:
    {
        a_[z] = m1_[z];
        i_[z] = +1;
        ++z;
    }
    j_ = 0;
    dm_ = -1;
}

The successor is computed as follows:

bool next()
{
    ulong j = 0;
    ulong ij;
    while ( (ij=i_[j]) ) // can touch sentinel i[n]==0
    {
        ulong dj;
        bool ovq; // overflow?
        if ( ij == 1 )
        {
            dj = next_endo(a_[j], m1_[j]);
            ovq = (dj==0);
        }
        else
        {
            ovq = (a_[j]==0);
            dj = prev_endo(a_[j], m1_[j]);
        }
        if ( ovq ) i_[j] = -ij;
        else
        {
            a_[j] = dj;
            dm_ = ij;
            j_ = j;
            return true;
        }
        ++j;
    }
    return false;
}

The routine for computation of the predecessor is obtained by changing the condition if ( ij == 1 ) to if ( ij != 1 ). About 65 million (radix 2) and 110 million (radix 16) numbers per second are generated.

The listing in figure [9.5-A] was created with the program [FXT: comb/mixedradix-endo-gray-demo.cc].
Chapter 10

Permutations

We present algorithms for the generation of all permutations in various orders such as lexicographic and minimal-change order. Some algorithms use mixed radix numbers with factorial base described in section 10.3. Algorithms for application, inversion and composition of permutations and the generation of random permutations are given in chapter 2.

10.1 Lexicographic order

<table>
<thead>
<tr>
<th>permutation</th>
<th>inv. perm.</th>
<th>compl. inv. perm.</th>
<th>reversed perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [ . 1 2 3 ]</td>
<td>[ . 1 2 3 ]</td>
<td>[ 3 2 1 . ]</td>
<td>[ 3 2 1 . ]</td>
</tr>
<tr>
<td>1: [ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ 3 2 1 . ]</td>
<td>[ 2 3 1 . ]</td>
</tr>
<tr>
<td>2: [ . 2 1 3 ]</td>
<td>[ . 3 1 2 ]</td>
<td>[ 3 . 2 1 ]</td>
<td>[ 3 1 2 . ]</td>
</tr>
<tr>
<td>3: [ . 2 3 1 ]</td>
<td>[ . 3 1 2 ]</td>
<td>[ 3 . 2 1 ]</td>
<td>[ 1 3 2 . ]</td>
</tr>
<tr>
<td>4: [ . 3 1 2 ]</td>
<td>[ . 2 3 1 ]</td>
<td>[ 3 1 . 2 ]</td>
<td>[ 2 1 3 . ]</td>
</tr>
<tr>
<td>5: [ . 3 2 1 ]</td>
<td>[ . 3 2 1 ]</td>
<td>[ 3 . 1 2 ]</td>
<td>[ 1 2 3 . ]</td>
</tr>
<tr>
<td>6: [ 1 . 2 3 ]</td>
<td>[ 1 . 2 3 ]</td>
<td>[ 2 3 . 1 ]</td>
<td>[ 3 2 . 1 ]</td>
</tr>
<tr>
<td>7: [ 1 . 3 2 ]</td>
<td>[ 1 . 3 2 ]</td>
<td>[ 2 3 . 1 ]</td>
<td>[ 2 3 . 1 ]</td>
</tr>
<tr>
<td>8: [ 1 2 . 3 ]</td>
<td>[ 2 . 1 3 ]</td>
<td>[ 1 3 . 2 ]</td>
<td>[ 3 . 1 2 ]</td>
</tr>
<tr>
<td>9: [ 1 2 3 . ]</td>
<td>[ 3 . 1 2 ]</td>
<td>[ 1 3 . 2 ]</td>
<td>[ 1 3 . 2 ]</td>
</tr>
<tr>
<td>10: [ 1 3 . 2 ]</td>
<td>[ 2 . 3 1 ]</td>
<td>[ 1 3 . 2 ]</td>
<td>[ 2 . 3 1 ]</td>
</tr>
<tr>
<td>11: [ 1 3 2 . ]</td>
<td>[ 3 . 2 1 ]</td>
<td>[ 1 3 . 2 ]</td>
<td>[ 2 . 3 1 ]</td>
</tr>
<tr>
<td>12: [ 2 . 1 3 ]</td>
<td>[ 1 2 . 3 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>13: [ 2 . 3 1 ]</td>
<td>[ 1 3 . 2 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>14: [ 2 1 . 3 ]</td>
<td>[ 2 1 . 3 ]</td>
<td>[ 1 . 3 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>15: [ 2 1 3 . ]</td>
<td>[ 3 1 . 2 ]</td>
<td>[ 1 . 3 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>16: [ 2 3 . 1 ]</td>
<td>[ 2 3 . 1 ]</td>
<td>[ 1 . 3 2 ]</td>
<td>[ 1 . 3 2 ]</td>
</tr>
<tr>
<td>17: [ 2 3 1 . ]</td>
<td>[ 3 2 . 1 ]</td>
<td>[ 1 . 3 2 ]</td>
<td>[ 1 . 3 2 ]</td>
</tr>
<tr>
<td>18: [ 3 . 1 2 ]</td>
<td>[ 1 2 . 3 ]</td>
<td>[ 2 . 3 1 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>19: [ 3 . 2 1 ]</td>
<td>[ 1 2 . 3 ]</td>
<td>[ 2 . 3 1 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>20: [ 3 1 . 2 ]</td>
<td>[ 2 1 . 3 ]</td>
<td>[ 2 . 1 3 ]</td>
<td>[ 1 . 3 2 ]</td>
</tr>
<tr>
<td>21: [ 3 1 2 . ]</td>
<td>[ 3 1 . 2 ]</td>
<td>[ 2 . 1 3 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>22: [ 3 2 . 1 ]</td>
<td>[ 2 3 . 1 ]</td>
<td>[ 1 . 2 3 ]</td>
<td>[ 1 . 2 3 ]</td>
</tr>
<tr>
<td>23: [ 3 2 1 . ]</td>
<td>[ 3 2 1 ]</td>
<td>[ 1 . 2 3 ]</td>
<td>[ 1 . 2 3 ]</td>
</tr>
</tbody>
</table>

**Figure 10.1-A:** All permutations of 4 elements in lexicographic order, their inverses, the complements of the inverses, and the reversed permutations. Dots denote zeros.

When generated in lexicographic order the permutations appear as if (read as numbers and) sorted numerically in ascending order, see figure 10.1-A. The first half of the inverse permutations are the reversed inverse permutations in the second half: the position of zero in the first half of the inverse permutations lies in the first half of each permutation, so their reversal gives the second half. Write $I$ for the operator that inverts a permutation, $C$ for the complement, and $R$ for reversal. Then we have

$$C = IRI$$

(10.1-1)
Chapter 10: Permutations

and thereby the first half of the permutations are the complements of the permutations in the second half. An implementation of an iterative algorithm is [FXT: class perm_lex in comb/perm-lex.h].

```cpp
1 class perm_lex
2 {
3  public:
4    ulong *p_; // permutation in 0, 1, ..., n-1, sentinel at [-1]
5    ulong n_; // number of elements to permute
6
7  public:
8    perm_lex(ulong n)
9    {
10       n_ = n;
11       p_ = new ulong[n+1];
12       p_[0] = 0; // sentinel
13       ++p_;         // first();
14    }
15    ~perm_lex() { --p_; delete [] p_; }
16
17    void first() { for (ulong i=0; i<n_; i++) p_[i] = i; }
18
19    const ulong *data() const { return p; }
20
21    //--snip--
22}
```

The only nontrivial part is the next()-method that computes the next permutation with each call. The routine perm_lex::next() is based on code by Glenn Rhoads.

```cpp
1 bool next()
2 {
3    // find for rightmost pair with p_[i] < p_[i+1]:
4    const ulong ni = n_ - 1;
5    ulong i = ni;
6    do { --i; } while ( p_[i] > p_[i+1] );
7    if ( (long)i<0 ) return false; // last sequence is falling seq.
8
9    // find rightmost element p[j] smaller than p[i]:
10   ulong j = ni;
11   while ( p_[i] > p_[j] ) { --j; }
12   swap2(p_[i], p_[j]);
13
14   // Here the elements p[i+1], ..., p[n-1] are a falling sequence.
15   // Reverse order to the right:
16   ulong r = ni;
17   ulong s = i + 1;
18   while ( r > s ) { swap2(p_[r], p_[s]); --r; ++s; }
19   return true;
20 }
```

Using the class is no black magic [FXT: comb/perm-lex-demo.cc]:

```cpp
ulong n = 4;
perm_lex P(n);
do{
    // visit permutation
} while ( P.next() );
```

The routine generates about 113 million permutations per second. A slightly faster algorithm is obtained by making the changes with the update operation for the co-lexicographic order (section 10.2) on the right end of the permutations [FXT: comb/perm-lex2.h]. The rate of generation is about 133 M/s when arrays are used, and about 115 M/s with pointers [FXT: comb/perm-lex2-demo.cc].

The routine for computing the successor can easily be adapted for permutations of a multiset, see section 11.2.2 on page 294.

[fxtbook draft of 2008-August-17]
10.2 Co-lexicographic order

Figure 10.2-A: The permutations of 4 elements in co-lexicographic order. Dots denote zeros.

Figure 10.2-A shows the permutations of 4 elements in co-lexicographic (colex) order. An algorithm for the generation is implemented in [FXT: class perm_colex in comb/perm-colex.h]:

```cpp
1  class perm_colex
2  {
3     public:
4         ulong *d_; // mixed radix digits with radix = [2, 3, 4, ...]
5         ulong *x_; // permutation
6         ulong n_; // permutations of n elements

7     public:
8         perm_colex(ulong n)
9             // Must have n>=2
10        {
11            n_ = n;
12            d_ = new ulong[n_];
13            d_[n-1] = 0; // sentinel
14            x_ = new ulong[n_];
15            first();
16        }
17    [--snip--]
18    }
19
20    void first()
21    {
22        for (ulong k=0; k<n_; ++k) x_[k] = n_-1-k;
23        for (ulong k=0; k<n_-1; ++k) d_[k] = 0;
24    }
25
The update process uses the rising factorial numbers described in section 10.3. Let \( j \) be the position where the digit is incremented, and \( d \) the value before the increment. The update

<table>
<thead>
<tr>
<th>permutation</th>
<th>rfact</th>
<th>inv. perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0 3 4 5 2 1 ]</td>
<td>[ 1 2 3 1 1 ]</td>
<td>[ 5 4 2 0 3 1 1 ]</td>
</tr>
<tr>
<td>[ 0 3 4 5 2 1 ]</td>
<td>[ 1 2 3 1 1 ]</td>
<td>[ 5 4 2 0 3 1 1 ]</td>
</tr>
</tbody>
</table>

is done in three steps:

<table>
<thead>
<tr>
<th>permutation</th>
<th>rfact</th>
<th>inv. perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0 3 4 5 2 1 ]</td>
<td>[ 1 2 3 1 1 ]</td>
<td>[ 5 4 2 0 3 1 1 ]</td>
</tr>
<tr>
<td>[ 0 2 4 5 3 1 ]</td>
<td>[ 1 2 3 2 1 ]</td>
<td>[ 5 4 2 0 3 1 1 ]</td>
</tr>
</tbody>
</table>

[ftxbook draft of 2008-August-17]
bool next()
{
    if ( d_[0]==0 ) // easy case
    {
        d_[0] = 1;
        swap2(x_[0], x_[1]);
        return true;
    }
    else
    {
        d_[0] = 0;
        ulong j = 1;
        ulong m1 = 2; // nine in rising factorial base
        while ( d_[j]==m1 )
        {
            d_[j] = 0;
            ++m1;
            ++j;
        }
        if ( j==n_-1 ) return false; // current permutation is last
        const ulong dj = d_[j];
        d_[j] = dj + 1;
        swap2( x_[dj], x_[j+1] ); // swap positions dj and j+1
        { // reverse range [0...j]:
            ulong a = 0, b = j;
            do
            {
                swap2(x_[a], x_[b]);
                ++a;
                --b;
            } while ( a<b );
        }
        return true;
    }
}

About 194 million permutations per second can be generated \[\text{FXT: comb/perm-colex-demo.cc}\]. With arrays instead of pointers the rate is 210 million per second.

## 10.3 Factorial representations of permutations

The factorial number system corresponds to the mixed radix bases \( M = [2, 3, 4, \ldots] \) (**rising factorial basis**) or \( M = [\ldots, 4, 3, 2] \) (**falling factorial basis**). A \((n-1)\)-digit factorial number can have \( n! \) different values.

We develop different methods to convert factorial numbers to permutations and vice versa.

### 10.3.1 The Lehmer code (inversion table)

Each permutation of \( n \) distinct elements can be converted to a unique \((n-1)\)-digit factorial number \( A = [a_0, a_1, \ldots, a_{n-2}] \) in the falling factorial base by counting, for each index \( k \), the number of elements with indices right of \( k \) that are smaller than the current element \[\text{FXT: comb/fact2perm.cc}\]:

```
void perm2ffact(const ulong *x, ulong n, ulong *fc)
// Convert permutation in x[0,...,n-1] into
// the (n-1) digit factorial representation in fc[0,...,n-2].
// One has: fc[0]<n, fc[1]<n-1, ..., fc[n-2]<2 (falling radices)
{
    for (ulong k=0; k<n-1; ++k)
    {
        ulong xk = x[k];
        ulong i = 0;
        for (ulong j=k; j<n; ++j) if ( x[j]<xk ) ++i;
```

[fxtbook draft of 2008-August-17]
The routine works as long as all elements of the permutation are distinct. The factorial representation obtained by this method is called the Lehmer code of the permutation. For example, the permutation \([3, 0, 1, 4, 2]\) has the Lehmer code \([3, 0, 0, 1]\), because three elements smaller than the first element (3) lie right to it, no element smaller than the second element (0) lies right to it, etc.

An alternative term for the Lehmer code is inversion table: An inversion of a permutation \([x_0, x_1, \ldots, x_{n-1}]\) is a pair of indices \(k\) and \(j\) where \(k < j\) and \(x_j < x_k\). Now fix \(k\) and call an inversion as above a right inversion at \(k\). The inversion table \([i_0, i_1, \ldots, i_{n-2}]\) of a permutation is obtained by setting \(i_k\) to the number of right inversions at \(k\). But this is exactly what we computed.

A routine that computes the permutation for a given Lehmer code is

```c
void ffact2perm(const ulong *fc, ulong n, ulong *x)
{
    for (ulong k=0; k<n; ++k) x[k] = k;
    for (ulong k=0; k<n-1; ++k)
        {
            ulong fa = fc[k];
            if ( fa ) rotate_right1(x+k, fa+1);
        }
}
```

A routine to compute the inverse permutation from the Lehmer code is

```c
void ffact2invperm(const ulong *fc, ulong n, ulong *x)
{
    for (ulong k=0; k<n; ++k) x[k] = n-1; // "empty"
    for (ulong k=0; k<n-1; ++k)
        {
            ulong fa = fc[k];
            for (ulong j=0; ; ++j)
                {
                    if ( x[j]==n-1 ) // if empty
                        {
                            if ( 0==fa ) { x[j] = k; break; }
                            --fa;
                        }
                    }
        }
}
```

A similar method can compute a representation in the rising factorial base. We count the number of elements to the left of \(k\) that are greater than the element at \(k\) (the number of left inversions at \(k\)):

```c
void perm2rfact(const ulong *x, ulong n, ulong *fc)
{
    for (ulong k=1; k<n; ++k)
        {
            ulong zk = x[k];
            ulong i = 0;
            for (ulong j=0; j<k; ++j) if ( x[j]>zk ) ++i;
            fc[k-1] = i;
        }
}
```

The inverse routine is

```c
void rfact2perm(const ulong *fc, ulong n, ulong *x)
{
    //
}
```
Chapter 10: Permutations

**Figure 10.3-A:** Numbers in falling factorial basis and permutations so that the number is the Lehmer code of it (left columns). Dots denote zeros. The rising factorial representation of the reversed and complemented permutation equals the reversed Lehmer code (right columns).

<table>
<thead>
<tr>
<th>ffact</th>
<th>permutation</th>
<th>rev.compl.perm.</th>
<th>rfact</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>[. . .]</td>
<td>[. 1 2 3]</td>
<td>[. 1 2 3]</td>
</tr>
<tr>
<td>1:</td>
<td>[. 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 1 . 2 3]</td>
</tr>
<tr>
<td>2:</td>
<td>[. 2 .]</td>
<td>[. 2 . 1 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>3:</td>
<td>[. 3 .]</td>
<td>[. 3 . 1 2]</td>
<td>[. 3 2 .]</td>
</tr>
<tr>
<td>4:</td>
<td>[. . 1]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>5:</td>
<td>[. 1 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>6:</td>
<td>[. 2 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>7:</td>
<td>[. 3 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>8:</td>
<td>[. . 2]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>9:</td>
<td>[. 2 . 1]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>10:</td>
<td>[. . 3]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>11:</td>
<td>[. 1 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>12:</td>
<td>[. 2 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>13:</td>
<td>[. 3 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 2 . 3 1]</td>
</tr>
<tr>
<td>14:</td>
<td>[. . 1]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>15:</td>
<td>[. 1 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>16:</td>
<td>[. 2 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>17:</td>
<td>[. 3 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>18:</td>
<td>[. . 2]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>19:</td>
<td>[. 1 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>20:</td>
<td>[. 2 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>21:</td>
<td>[. 3 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>22:</td>
<td>[. . 3]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
<tr>
<td>23:</td>
<td>[. 1 1 .]</td>
<td>[. 1 . 2 3]</td>
<td>[. 3 1 .]</td>
</tr>
</tbody>
</table>

**Figure 10.3-B:** Numbers in rising factorial basis and permutations so that the number is the Lehmer code of it (left columns). The reversed and complemented permutations and their falling factorial representations are shown in the right columns. They appear in lexicographic order.
10.3: Factorial representations of permutations

```cpp
for (ulong k=0; k<n; ++k) x[k] = k;
ulong *y = x+n;
for (ulong k=n-1; k!=0; --k, --y)
{
    ulong fa = fc[k-1];
    if ( fa )
    {
        ++fa;
        rotate_left1(y-fa, fa);
    }
}
```

A routine for the inverse permutation is

```cpp
void rfact2invperm(const ulong *fc, ulong n, ulong *x)
// Convert the (n-1) digit factorial representation in fc[0,...,n-2].
// into permutation in x[0,...,n-1] such that
// the permutation is the inverse of the one computed via rfact2perm().
{
    for (ulong k=0; k<n; ++k) x[k] = 0; // "empty"
    for (ulong k=n-2; (long)k>=0; --k)
    {
        ulong fa = fc[k];
        for (ulong j=0; ; ++j)
        {
            if ( x[j]==0 ) // if empty
            {
                if ( 0==fa ) { x[j] = k+1; break; }
                --fa;
            }
            --fa;
        }
    }
    for (ulong k=0, r=n-1; k<r; ++k, --r) swap2(x[k], x[r]);
}
```

The permutations corresponding to the Lehmer codes (in counting order) are shown in figure 10.3-A (left columns). The permutation whose rising factorial representation is the digit-reversed Lehmer code is obtained by reversing and complementing (replacing each element \( x \) by \( n - 1 - x \)) the original permutation:

<table>
<thead>
<tr>
<th>Lehmer code</th>
<th>permutation</th>
<th>rev.perm</th>
<th>compl.rev.perm</th>
<th>rising fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3,0,0,1]</td>
<td>[3,0,1,4,2]</td>
<td>[2,4,1,0,3]</td>
<td>[2,0,3,4,1]</td>
<td>[1,0,0,3]</td>
</tr>
</tbody>
</table>

The permutations obtained from counting in the rising factorial base are shown in figure 10.3-B.

10.3.2 Computation with large arrays

Using the left-right array described in section 4.7 on page 160 the conversion to and from the Lehmer code can be done in \( O(n \log(n)) \) operations [FXT: comb/big-fact2perm.cc]:

```cpp
void perm2ffact(const ulong *x, ulong n, ulong *fc, left_right_array &LR)
{
    LR.set_all();
    for (ulong k=0; k<n-1; ++k)
    {
        // i := number of Free positions Left of i, Excluding i.
        ulong i = LR.num_SLE( x[k] );
        LR.get_set_idx_chg( i );
        fc[k] = i;
    }
}
```

The LR-array passed as extra argument has to be of size \( n \). Conversion of an array of, say, 10 million entries is a matter of seconds if this routine is used [FXT: comb/big-fact2perm-demo.cc]. Merging the methods num_set_left() and get_set_idx_chg() would double the performance.

```cpp
void ffact2perm(const ulong *fc, ulong n, ulong *x, left_right_array &LR)
{
    LR.free_all();
    for (ulong k=0; k<n-1; ++k)
    {
```
The routines for rising factorials are:

```plaintext
void perm2rfact(const ulong *x, ulong n, ulong *fc, left_right_array &LR)
{
    LR.set_all();
    for (ulong k=0, r=n-1; k<n-1; ++k, --r) // r == n-1-k;
    { // i := number of Free positions Left of i, Excluding i.
        ulong i = LR.num_SLE( x[r] );
        LR.get_set_idx_chg( i );
        fc[r-1] = r - i;
    }
}
```

The conversion of the routines that compute permutations from factorial numbers into routines that compute the inverse permutations is especially easy, just change as follows:

```plaintext
x[a] = b; =---> x[b] = a;
```

Thereby we obtain the routines

```plaintext
void rfact2invperm(const ulong *fc, ulong n, ulong *x, left_right_array &LR)
{
    LR.free_all();
    for (ulong k=0; k<n-1; ++k)
    { // i := number of Free positions Right of i, Excluding i.
        ulong i = LR.num_SLE( x[n-1-k] );
        LR.get_set_idx_chg( i );
        x[n-1-k] = n-1-i;
    }
    ulong i = LR.get_free_idx_chg( 0 );
    x[0] = n-1-i;
}
```

and

```plaintext
void rfact2invperm(const ulong *fc, ulong n, ulong *x, left_right_array &LR)
{
    LR.free_all();
    for (ulong k=0; k<n-1; ++k)
    { // i := number of Free positions Right of i, Excluding i.
        ulong i = LR.num_SLE( x[n-1-k] );
        LR.get_set_idx_chg( i );
        x[n-1-k] = n-1-i;
    }
    ulong i = LR.get_free_idx_chg( 0 );
    x[n-1-i] = 0;
}
```

### 10.3.3 An representation via reversals

Replacing the rotations in the computation of a permutation from its Lehmer code by reversals one obtains a different one-to-one relation between factorial numbers and permutations. The routine for the falling factorial basis is [FXT: comb/fact2perm-rev.cc]:

[fxtbook draft of 2008-August-17]
### 10.3: Factorial representations of permutations

**Figure 10.3-C:** Numbers in falling factorial basis and permutations so that the number is the alternative (reversal-) code of it (left columns). The inverse permutations and their rising factorial representations are shown in the right columns.

<table>
<thead>
<tr>
<th>n</th>
<th>ffact</th>
<th>permutation</th>
<th>inv.perm.</th>
<th>ffact</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[ .   ]</td>
<td>[ . 1 2 3 ]</td>
<td>[ . 1 2 3 ]</td>
<td>[ .   ]</td>
</tr>
<tr>
<td>1</td>
<td>[ 1   ]</td>
<td>[ 1 1 2 3 ]</td>
<td>[ 1 1 2 3 ]</td>
<td>[ 1   ]</td>
</tr>
<tr>
<td>2</td>
<td>[ . 2 ]</td>
<td>[ 1 2 1 3 ]</td>
<td>[ 1 2 1 3 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>3</td>
<td>[ . 1 ]</td>
<td>[ 2 1 2 3 ]</td>
<td>[ 2 1 2 3 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>4</td>
<td>[ 1 1 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>5</td>
<td>[ 1 1 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>6</td>
<td>[ 1 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>7</td>
<td>[ 1 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>8</td>
<td>[ 1 1 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>9</td>
<td>[ 1 1 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>10</td>
<td>[ 2 2 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 2 2 ]</td>
</tr>
</tbody>
</table>

**Figure 10.3-D:** Numbers in rising factorial basis and permutations so that the number is the alternative (reversal-) code of it (left columns). The inverse permutations and their falling factorial representations are shown in the right columns.

<table>
<thead>
<tr>
<th>n</th>
<th>rfact</th>
<th>permutation</th>
<th>inv.perm.</th>
<th>rfact</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[ 1   ]</td>
<td>[ . 1 2 3 ]</td>
<td>[ . 1 2 3 ]</td>
<td>[ 1   ]</td>
</tr>
<tr>
<td>1</td>
<td>[ 1   ]</td>
<td>[ 1 1 2 3 ]</td>
<td>[ 1 1 2 3 ]</td>
<td>[ 1   ]</td>
</tr>
<tr>
<td>2</td>
<td>[ . 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>3</td>
<td>[ . 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>4</td>
<td>[ 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 2 1 ]</td>
</tr>
<tr>
<td>5</td>
<td>[ 3 1 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 3 1 ]</td>
</tr>
<tr>
<td>6</td>
<td>[ 2 2 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 1 2 3 2 ]</td>
<td>[ 2 2 ]</td>
</tr>
<tr>
<td>7</td>
<td>[ . 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>8</td>
<td>[ 1 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>9</td>
<td>[ 1 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>10</td>
<td>[ 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 1 3 2 1 ]</td>
<td>[ 2 1 ]</td>
</tr>
<tr>
<td>11</td>
<td>[ 2 2 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 2 2 ]</td>
</tr>
<tr>
<td>12</td>
<td>[ 1 1 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>13</td>
<td>[ 1 1 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>14</td>
<td>[ 1 2 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>15</td>
<td>[ 1 1 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>16</td>
<td>[ 1 1 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 2 1 3 2 ]</td>
<td>[ 1 1 ]</td>
</tr>
<tr>
<td>17</td>
<td>[ 1 2 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 2 3 1 ]</td>
<td>[ 1 2 ]</td>
</tr>
</tbody>
</table>

[fxtbook draft of 2008-August-17]
void perm2ffact_rev(const ulong *x, ulong n, ulong *fc) {
  ALLOCA(ulong, ti, n); // inverse permutation
  for (ulong k=0; k<n; ++k) ti[x[k]] = k;
  for (ulong k=0; k<n-1; ++k) {
    ulong j; // find element k
    for (j=k; j<n; ++j) if ( ti[j] == k ) break;
    j -= k;
    fc[k] = j;
    reverse(ti+k, j+1);
  }
}

The routine is the inverse of

void ffact2perm_rev(const ulong *fc, ulong n, ulong *x) {
  for (ulong k=0; k<n; ++k) x[k] = k;
  for (ulong k=0; k<n-1; ++k) {
    ulong fa = fc[k];
    // Lehmer: rotate_right1(x+k, fa+1);
    if ( fa ) reverse(x+k, fa+1);
  }
}

Figure 10.3-C shows the permutations of 4 elements and their falling factorial representations, it was created with the program [FXT: comb/fact2perm-rev-demo.cc]. The routines for the rising factorial basis (see figure 10.3-D) are

void perm2rfact_rev(const ulong *x, ulong n, ulong *fc) {
  ALLOCA(ulong, ti, n); // inverse permutation
  for (ulong k=0; k<n; ++k) ti[x[k]] = k;
  for (ulong k=n-1; k!=0; --k) {
    ulong j; // find element k
    for (j=0; j<=k; ++j) if ( ti[j] == k ) break;
    j = k - j;
    fc[k-1] = j;
    reverse(ti+k-j, j+1);
  }
}

and

void rfact2perm_rev(const ulong *fc, ulong n, ulong *x) {
  for (ulong k=0; k<n; ++k) x[k] = k;
  ulong *y = x+n;
  for (ulong k=n-1; k!=0; --k, --y) {
    ulong fa = fc[k-1];
    if ( fa ) {
      ++fa;
      // Lehmer: rotate_left1(y-fa, fa);
      reverse(y-fa, fa);
  }
}

10.3.4 A representation via swaps

The routines for the conversion from permutations to factorial representations shown so far have complexity $n^2$. The following routines compute different factorial representations with complexity $n$ [FXT: comb/fact2perm-swp.cc]:

void perm2ffact_swp(const ulong *x, ulong n, ulong *fc) {
  // Convert permutation in x[0,...,n-1] into
### 10.3: Factorial representations of permutations

#### Figure 10.3-E:
Numbers in falling factorial basis and permutations so that the number is the alternative (swaps-) code of it (left columns). The inverse permutations and their rising factorial representations are shown in the right columns.

<table>
<thead>
<tr>
<th>ffact.</th>
<th>permutation</th>
<th>inv.perm.</th>
<th>rfact.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ... ]</td>
<td>[. . 1 2 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. ... ]</td>
</tr>
<tr>
<td>[ 1 ... ]</td>
<td>[1 2 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 ... ]</td>
</tr>
<tr>
<td>[ 2 ... ]</td>
<td>[2 1 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 3 ... ]</td>
</tr>
<tr>
<td>[ 3 ... ]</td>
<td>[3 1 2 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 1 1 ... ]</td>
<td>[1 2 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 2 1 ... ]</td>
<td>[2 1 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 3 1 ... ]</td>
<td>[3 1 2 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 1 2 ... ]</td>
<td>[1 3 2 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 2 2 ... ]</td>
<td>[2 3 1 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 3 2 ... ]</td>
<td>[3 2 1 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
</tbody>
</table>

#### Figure 10.3-F:
Numbers in rising factorial basis and permutations so that the number is the alternative (swaps-) code of it (left columns). The inverse permutations and their falling factorial representations are shown in the right columns.

<table>
<thead>
<tr>
<th>rfact.</th>
<th>permutation</th>
<th>inv.perm.</th>
<th>ffact.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ... ]</td>
<td>[. . 1 2 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. ... ]</td>
</tr>
<tr>
<td>[ 1 ... ]</td>
<td>[1 2 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 ... ]</td>
</tr>
<tr>
<td>[ 2 ... ]</td>
<td>[2 1 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 3 ... ]</td>
<td>[3 1 2 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 1 1 ... ]</td>
<td>[1 2 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 2 1 ... ]</td>
<td>[2 1 3 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 3 1 ... ]</td>
<td>[3 1 2 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 1 2 ... ]</td>
<td>[1 3 2 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 2 2 ... ]</td>
<td>[2 3 1 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
<tr>
<td>[ 3 2 ... ]</td>
<td>[3 2 1 ]</td>
<td>[. 1 2 3 ]</td>
<td>[. 1 1 ]</td>
</tr>
</tbody>
</table>

[fxtbook draft of 2008-August-17]
Chapter 10: Permutations

The (n-1) digit (swaps-) factorial representation in fc[0,...,n-2].
One has: fc[0]<n, fc[1]<n-1, ..., fc[n-2]<2 (falling radices)

\{ 
  ALLOCA(ulong, t, n);
  for (ulong k=0; k<n; ++k) t[k] = x[k];
  ALLOCA(ulong, ti, n); // inverse permutation
  for (ulong k=0; k<n; ++k) ti[t[k]] = k;
  for (ulong k=0; k<n-1; ++k) 
    { 
      ulong tk = t[k]; // >= k 
      fc[k] = tk - k;
      ulong j = ti[k]; // location of element k 
      ti[tk] = ti[t[j]];
      t[j] = tk;
    }
\}

void perm2rfact_swp(const ulong *x, ulong n, ulong *fc)
// Convert permutation in x[0,...,n-1] into 
// the (n-1) digit (swaps-) factorial representation in fc[0,...,n-2].
// One has: fc[0]<2, fc[1]<3, ... , fc[n-2]<n (rising radices)
{ 
  ALLOCA(ulong, t, n);
  for (ulong k=0; k<n; ++k) t[k] = x[k];
  ALLOCA(ulong, ti, n); // inverse permutation
  for (ulong k=0; k<n; ++k) ti[t[k]] = k;
  for (ulong k=0; k<n-1; ++k) 
    { 
      ulong j = ti[k]; // location of element k, j>=k 
      fc[n-2-k] = j - k;
      ulong tk = t[k];
      ti[tk] = ti[t[j]];
      t[j] = tk;
    }
}

Their inverses also have complexity n. The routine for falling base is

void ffact2perm_swp(const ulong *fc, ulong n, ulong *x)
// Inverse of perm2ffact_swp().
{ 
  for (ulong k=0; k<n; ++k) x[k] = k;
  for (ulong k=0; k<n-1; ++k) 
    { 
      ulong fa = fc[k];
      swap2( x[k], x[k+fa] );
    }
}

The routine for the rising base is

void rfact2perm_swp(const ulong *fc, ulong n, ulong *x)
// Inverse of perm2rfact_swp().
{ 
  for (ulong k=0; k<n; ++k) x[k] = k;
  for (ulong k=0, j=n-2; k<n-1; ++k,--j) 
    { 
      ulong fa = fc[k];
      swap2( x[j], x[j+fa] );
    }
}

The permutations corresponding to the alternative codes for the falling basis are shown in figure 10.3-E (left columns). The inverse permutation has the rising factorial representation that is digit-reversed representation (right columns). The permutations corresponding to the alternative codes for rising basis are shown in figure 10.3-F. The listings were created with the program [FXT: comb/fact2perm-swp-demo.cc]. The routines can serve as a means to find interesting orders of permutations. Indeed, the permutation generator shown in section 10.4 on page 246 was found this way. A recursive algorithm for the (inverse) permutations shown in figure 10.3-F is given in section 10.13 on page 279.

[fxtbook draft of 2008-August-17]
10.3.5 Cyclic permutations from factorial numbers

<table>
<thead>
<tr>
<th>falling fact.</th>
<th>permutation</th>
<th>cycle</th>
<th>inv.perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0   1   2   3   4 ]</td>
<td>(0, 1, 2, 3, 4)</td>
<td>[ 0 1 2 3 4 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   2   4   3 ]</td>
<td>(0, 1, 4, 3, 2)</td>
<td>[ 2 4 1 3 0 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   3   2   4 ]</td>
<td>(0, 3, 2, 1, 4)</td>
<td>[ 3 2 0 4 1 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   3   4   2 ]</td>
<td>(0, 3, 1, 2, 4)</td>
<td>[ 2 3 4 1 0 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   4   2   3 ]</td>
<td>(0, 1, 2, 4, 3)</td>
<td>[ 1 2 4 0 3 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   4   3   2 ]</td>
<td>(0, 1, 3, 4, 2)</td>
<td>[ 1 4 3 0 2 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   2   3   4 ]</td>
<td>(0, 2, 3, 4, 1)</td>
<td>[ 3 2 4 1 0 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   2   4   3 ]</td>
<td>(0, 2, 4, 3, 1)</td>
<td>[ 3 4 0 2 1 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   3   2   4 ]</td>
<td>(0, 3, 2, 1, 4)</td>
<td>[ 2 3 1 4 0 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   3   4   2 ]</td>
<td>(0, 3, 1, 2, 4)</td>
<td>[ 2 1 3 4 0 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   4   2   3 ]</td>
<td>(0, 1, 2, 4, 3)</td>
<td>[ 1 2 4 0 3 ]</td>
<td></td>
</tr>
<tr>
<td>[0   1   4   3   2 ]</td>
<td>(0, 1, 3, 4, 2)</td>
<td>[ 1 4 3 0 2 ]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.3-G: Numbers in falling factorial basis and the corresponding cyclic permutations.

The cyclic permutations of \( n \) elements (permutations that consists of one cycle of size \( n \), see section 2.10.3 on page 112) can be obtained from length-(\( n - 2 \)) factorial numbers. We give routines for both falling and rising base [FXT: comb/fact2cyclic.cc]:

```cpp
1 void ffact2cyclic(const ulong *fc, ulong n, ulong x[])
2 // Generate cyclic permutation (standard representation) in x[]
3 // from the (n-2) digit factorial number in fc[0,...,n-3].
4 // Falling radices: [n-1, ..., 3, 2]
5 {
6     for (ulong k=0; k<n; ++k) x[k] = k;
7     for (ulong k=n-1; k>1; --k)
8         {ulong z = n-1-k; // 0, ..., n-3
9             ulong i = fc[z]; // 0, ..., n-3
10             swap2(x[k], x[i]);
11         }
12     if ( n>1 ) swap2(x[0], x[1]);
13 }
14
15 void rfact2cyclic(const ulong *fc, ulong n, ulong x[])
16 // Generate cyclic permutation (standard representation) in x[]
17 // from the (n-2) digit factorial number in fc[0,...,n-3].
18 // Rising radices: [2, 3, ..., n-1]
19 {
20     for (ulong k=0; k<n; ++k) x[k] = k;
21     for (ulong k=n-1; k>1; --k)
22         {ulong i = fc[k-2]; // k-2 == n-3, ..., 0
23             swap2(x[k], x[i]);
24         }
25     if ( n>1 ) swap2(x[0], x[1]);
26 }
```

The cyclic permutations of 5 elements are shown in figures 10.3-G (falling base) and 10.3-H (rising base). The listings were created with the program [FXT: comb/fact2cyclic-demo.cc]. Note that the cycle
Chapter 10: Permutations

rising fact.  permutation  cycle  inv.perm.
[ . . . ]  [ 1 2 3 4 0 ]  ( 0, 1, 2, 3, 4, )  [ 4 0 1 2 3 ]
[ 1 . . ]  [ 2 3 1 4 0 ]  ( 0, 2, 1, 3, 4, )  [ 4 2 0 1 3 ]
[ . 1 . ]  [ 3 2 4 1 0 ]  ( 0, 3, 1, 2, 4, )  [ 4 3 1 0 2 ]
[ 1 1 . ]  [ 2 4 3 1 0 ]  ( 0, 2, 3, 1, 4, )  [ 4 3 0 2 1 ]
[ . 2 . ]  [ 1 3 4 2 0 ]  ( 0, 1, 3, 2, 4, )  [ 4 0 3 1 2 ]
[ 1 2 . ]  [ 3 4 1 2 0 ]  ( 0, 3, 2, 1, 4, )  [ 4 2 3 0 1 ]
[ . 1 1 ]  [ 4 2 3 0 1 ]  ( 0, 2, 4, 1, 3, )  [ 3 4 1 2 0 ]
[ 1 1 1 ]  [ 2 3 4 0 1 ]  ( 0, 3, 4, 1, 2, )  [ 3 4 0 1 2 ]
[ . 2 1 ]  [ 4 3 0 1 2 ]  ( 0, 4, 1, 3, 2, )  [ 2 4 3 1 0 ]
[ 1 2 1 ]  [ 3 0 4 1 2 ]  ( 0, 3, 2, 4, 1, )  [ 1 4 0 2 3 ]
[ . 2 2 ]  [ 1 4 3 0 2 ]  ( 0, 1, 4, 2, 3, )  [ 3 0 4 2 1 ]
[ 1 2 2 ]  [ 4 3 1 0 2 ]  ( 0, 4, 2, 1, 3, )  [ 3 2 4 1 0 ]
[ . 1 2 ]  [ 3 4 0 1 2 ]  ( 0, 3, 1, 4, 2, )  [ 2 3 4 1 0 ]
[ 1 1 2 ]  [ 4 0 3 1 2 ]  ( 0, 4, 2, 3, 1, )  [ 1 3 4 2 0 ]
[ . 2 2 ]  [ 1 3 0 4 2 ]  ( 0, 1, 3, 4, 2, )  [ 2 0 4 1 3 ]
[ 1 2 2 ]  [ 3 0 1 4 2 ]  ( 0, 3, 4, 2, 1, )  [ 1 2 4 0 3 ]
[ . 3 2 ]  [ 1 2 4 0 3 ]  ( 0, 1, 2, 4, 3, )  [ 3 0 1 4 2 ]
[ 1 2 2 ]  [ 2 4 1 0 3 ]  ( 0, 2, 1, 4, 3, )  [ 3 2 0 4 1 ]
[ . 1 3 ]  [ 4 2 0 1 3 ]  ( 0, 4, 3, 1, 2, )  [ 2 3 1 4 0 ]
[ 1 2 3 ]  [ 4 2 0 1 3 ]  ( 0, 2, 4, 3, 1, )  [ 1 3 0 4 2 ]
[ . 2 3 ]  [ 1 4 0 2 3 ]  ( 0, 1, 4, 3, 2, )  [ 2 0 3 4 1 ]
[ 1 2 3 ]  [ 4 0 1 2 3 ]  ( 0, 4, 3, 2, 1, )  [ 1 2 3 4 0 ]

Figure 10.3-H: Numbers in rising factorial basis and corresponding cyclic permutations.

representation could be obtained by applying the transformations to (all) permutations to all but the
first element. That is, one can generate all cyclic permutations in cycle form by permuting all elements
but the first with any permutation algorithm.

10.4 An order from reversing prefixes

A surprisingly simple algorithm for the generation of all permutations is obtained by mixed radix counting
with the radices \([2, 3, 4, \ldots]\) (column digits in figure 10.4-A). Whenever the first \(j\) digits change with
an increment then the permutation is updated by reversing the first \(j+1\) elements (the method is given
in [285]).

As with lex order the first half of the permutations are the complements of the permutations in the second
half, now rewrite relation 10.1-1 on page 233 as

\[
R = ICI
\]

(10.4-1)
to see that the first half of the inverse permutations are the reversed inverse permutations in the second
half. This can (for \(n\) even) also be observed from the positions of the largest element in the inverse
permutations. A generator is [FXT: class perm_rev in comb/perm-rev.h]:

```
1  class perm_rev
2  {
3    public:
4      ulong *d_;  // mixed radix digits with radix = [2, 3, 4, \ldots, n-1, (sentinel=-1)]
5      ulong *p_;  // permutation
6      ulong n_;  // permutations of n elements
7    }
8    public:
9      perm_rev(ulong n)
10     {
11        n_ = n;
12        p_ = new ulong[n_];
13        d_ = new ulong[n_];
14        d_[n-1] = -1UL;  // sentinel
15        first();
```
An order from reversing prefixes

Figure 10.4-A: All permutations of 4 elements in an order where the first \( j + 1 \) elements are reversed when the first \( j \) digits change in the mixed radix counting sequence with radices \([2, 3, 4, \ldots]\).

```cpp
16 } 17 -perm_rev()
19 { 20    delete [] p_;
21    delete [] d_;
22 }
23 void first()
25 { 26    for (ulong k=0; k<n_-1; ++k) d_[k] = 0;
27    for (ulong k=0; k<n_; ++k) p_[k] = k;
28 }
29 void last()
31 { 32    for (ulong k=0; k<n_-1; ++k) d_[k] = k+1;
33    for (ulong k=0; k<n_; ++k) p_[k] = n_-1-k;
34 }

The update routines are quite concise:

```
Chapter 10: Permutations

Note that the routines work for arbitrary (distinct) entries of the array $p_\[\]$.

An upper bound for the average number of elements that are moved in the transitions when generating all $N=n!$ permutations is $e \approx 2.7182818$ so the algorithm is CAT. The implementation is actually fast, it generates more than 110 million permutations per second [FXT: comb/perm-rev-demo.cc]. Usage of the class is as simple as:

```c++
ulong n = 4; // Number of elements to permute
perm_rev P(n);
P.first();
do {
  // Use permutation here
} while ( P.next() );
```

We note that the inverse permutations have the single-track property, see section 10.10 on page 271.

### 10.4.1 Optimizing the update routine

We optimize the update routine by observing that 5 out of six updates are the swaps

$(0,1) \ (0,2) \ (0,1) \ (0,2) \ (0,1)$

We use a counter $ct_\_$ and modify the methods first() and next() accordingly [FXT: class perm_rev2 in comb/perm-rev2.h]:

```c++
class perm_rev2
{
    [--snip--]
    void first()
    {
        for (ulong k=0; k<n_-1; ++k) d_[k] = 0;
        for (ulong k=0; k<n_; ++k) p_[k] = k;
        ct_ = 5;
    }

    bool next()
    {
        if ( ct_!=0 ) // easy case(s)
        {
            --ct_;
            swap2(p_[0], p_[1 + (ct_ & 1)]);
            return true;
        }
        else // increment mixed radix number:
        {
            ct_ = 5; // reset counter
            ulong j = 2; // note: start with 2
            while ( d_[j]==j+1 ) { d_[j]=0; ++j; }
            // j==n-1 for last permutation
            if ( j!=n_-1 ) // only if no overflow
            {
                ++d_[j];
                reverse(p_, j+2); // update permutation
                return true;
            }
            else return false;
        }
    }
};
```

[fxtbook draft of 2008-August-17]
10.5 Minimal-change order (Heap’s algorithm)

The speedup is remarkable, about 186 million permutations per second are generated (about 11.8 cycles per update). If arrays are used instead of pointers, the rate is about 200 million per second (10.8 cycles per update). The routine can be used for permutations of at least three elements [FXT: comb/permrev2-demo.cc].

10.4.2 Method for unranking

Conversion of a rising factorial number into the corresponding permutation proceeds as exemplified for the 16-th permutation \((15 = 1 \cdot 1 + 1 \cdot 2 + 2 \cdot 6, \text{ so } d=[1,1,2])\):

1: \(p=[0, 1, 2, 3]\) \(d=[0, 0, 0]\) // start

13: \(p=[2, 3, 0, 1]\) \(d=[0, 0, 2]\) // right rotate all elements twice

15: \(p=[0, 2, 3, 1]\) \(d=[0, 1, 2]\) // right rotate first three elements

16: \(p=[2, 0, 3, 1]\) \(d=[1, 1, 2]\) // right rotate first two elements

The idea can be implemented as

```c
void goto_rfact(const ulong *d)
// Goto permutation corresponding to d[] (i.e. unrank d[]).
// d[] must be a valid (rising) factorial mixed radix string:
// d[]==[d(0), d(1), d(2), ..., d(n-2)] (n-1 elements) where 0<=d(j)<=j+1
{
    for (ulong k=0; k<n_; ++k) p_[k] = k;
    for (ulong k=0; k<n_-1; ++k) d_[k] = d[k];
    for (long j=n_-2; j>=0; --j) rotate_right(p_, j+2, d_[j]);
}
```

10.5 Minimal-change order (Heap’s algorithm)

<table>
<thead>
<tr>
<th>permutation</th>
<th>swap</th>
<th>digits</th>
<th>rfact(perm)</th>
<th>inv. perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [ . 1 2 3 ]</td>
<td>(0, 0)</td>
<td>[ . . . ]</td>
<td>[ . . . ]</td>
<td>[ . 1 2 3 ]</td>
</tr>
<tr>
<td>1: [ 1 1 2 3 ]</td>
<td>(1, 0)</td>
<td>[ 1 . . ]</td>
<td>[ 1 . . ]</td>
<td>[ 1 2 3 ]</td>
</tr>
<tr>
<td>2: [ 2 1 3 ]</td>
<td>(2, 0)</td>
<td>[ . 1 . ]</td>
<td>[ 1 1 . ]</td>
<td>[ 1 2 3 ]</td>
</tr>
<tr>
<td>3: [ 1 2 3 ]</td>
<td>(1, 0)</td>
<td>[ 1 1 . ]</td>
<td>[ . 1 . ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>4: [ 2 1 3 ]</td>
<td>(2, 0)</td>
<td>[ . 2 . ]</td>
<td>[ . 2 . ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>5: [ 2 1 3 ]</td>
<td>(1, 0)</td>
<td>[ 2 1 . ]</td>
<td>[ 2 1 . ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>6: [ 3 1 2 ]</td>
<td>(3, 0)</td>
<td>[ . 1 1 ]</td>
<td>[ 1 2 1 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>7: [ 1 3 2 ]</td>
<td>(1, 0)</td>
<td>[ 1 1 1 ]</td>
<td>[ . 2 1 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>8: [ 3 1 2 ]</td>
<td>(2, 0)</td>
<td>[ 1 1 1 ]</td>
<td>[ 1 1 1 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>9: [ 3 1 2 ]</td>
<td>(1, 0)</td>
<td>[ 1 1 1 ]</td>
<td>[ 1 1 1 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>10: [ 1 3 2 ]</td>
<td>(2, 0)</td>
<td>[ 1 2 1 ]</td>
<td>[ . 1 1 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>11: [ 1 3 2 ]</td>
<td>(1, 0)</td>
<td>[ . 1 2 ]</td>
<td>[ . 1 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>12: [ 2 3 1 ]</td>
<td>(3, 1)</td>
<td>[ 1 1 2 ]</td>
<td>[ . 2 1 ]</td>
<td>[ . 3 1 2 ]</td>
</tr>
<tr>
<td>13: [ 2 3 1 ]</td>
<td>(1, 0)</td>
<td>[ 1 1 2 ]</td>
<td>[ 1 2 1 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>14: [ 3 2 1 ]</td>
<td>(2, 0)</td>
<td>[ . 1 2 ]</td>
<td>[ 1 1 2 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>15: [ 3 2 1 ]</td>
<td>(1, 0)</td>
<td>[ 1 1 2 ]</td>
<td>[ 1 2 1 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>16: [ 3 2 1 ]</td>
<td>(2, 0)</td>
<td>[ . 2 2 ]</td>
<td>[ . 1 2 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>17: [ 3 2 1 ]</td>
<td>(1, 0)</td>
<td>[ 1 2 2 ]</td>
<td>[ 1 2 2 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>18: [ 3 2 1 ]</td>
<td>(3, 2)</td>
<td>[ . 3 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 3 2 1 ]</td>
</tr>
<tr>
<td>19: [ 3 2 1 ]</td>
<td>(1, 0)</td>
<td>[ 1 3 3 ]</td>
<td>[ . 2 3 ]</td>
<td>[ . 3 2 1 ]</td>
</tr>
<tr>
<td>20: [ 3 2 1 ]</td>
<td>(2, 0)</td>
<td>[ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 3 2 1 ]</td>
</tr>
<tr>
<td>21: [ 3 1 2 ]</td>
<td>(1, 0)</td>
<td>[ 1 1 3 ]</td>
<td>[ 1 1 3 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>22: [ 2 1 3 ]</td>
<td>(2, 0)</td>
<td>[ . 2 3 ]</td>
<td>[ 1 3 3 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>23: [ 1 2 3 ]</td>
<td>(1, 0)</td>
<td>[ 1 2 3 ]</td>
<td>[ . 3 3 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
</tbody>
</table>

Figure 10.5-A: The permutations of 4 elements in a minimal-change order. Dots denote zeros.
Chapter 10: Permutations

The mixed radix string $R$. Then the swap is $(j+1,x)$ where $x = 0$ if $j$ is odd, and $x = R_j - 1$ if $j$ is even. The sequence of values $j+1$ starts

$$1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, \ldots$$

The $n$-th value (starting with $n = 1$) is the largest $z$ such that $z!$ divides $n$ (entry A055881 of [245]). The column labeled $\text{rfact}(\text{perm})$ of the figure shows the rising factorial representations of the permutation, see section 10.8 on page 259. The column is a Gray code only for permutations of up to four elements.

An implementation of the algorithm (given in [144]) is [FXT: class perm_heap in comb/perm-heap.h]:

```cpp
class perm_heap {
public:
    ulong *d_; // mixed radix digits with radix = [2, 3, 4, ..., n-1, (sentinel=-1)]
    ulong *p_; // permutation
    ulong n_; // permutations of n elements
    ulong sw1_, sw2_; // indices of swapped elements
    [--snip--]

    bool next() {
        // increment mixed radix number:
        ulong j = 0;
        while ( d_[j]==j+1 ) { d_[j]=0; ++j; }
        // j==n-1 for last permutation
        if ( j!=n_-1 ) // only if no overflow
            { // increment mixed radix number:
                ulong k = j+1;
                ulong x = ( (k&1) ? d_[j] : 0);
                swap2(p_[k], p_[x]);
                sw1_ = k; sw2_ = x;
                ++d_[j];
                return true;
            }
        else return false;
    [--snip--]
}
```

About 115 million permutations are generated per second. Often one will only use the indices of the swapped elements to update the visited configurations:

```cpp
void get_swap(ulong &s1, ulong &s2) const { s1=sw1_; s2=sw2_; }
```

Then the statement `swap2(p_[k], p_[x]);` in the update routine can be omitted which leads to a rate of 165 million permutations per second. Figure 10.5-A shows the permutations of 4 elements, it was created with the program [FXT: comb/perm-heap-demo.cc].

### 10.5.1 Optimized implementation

The algorithm can be optimized by treating 5 out of 6 cases separately, those where the first or second digit in the mixed radix number changes. We use a counter $\text{ct}_-$ that is decremented [FXT: class perm_heap2 in comb/perm-heap2.h]:

```cpp
class perm_heap2 {
public:
    ulong *d_; // mixed radix digits with radix = [2, 3, 4, 5, ..., n-1, (sentinel=-1)]
    ulong *p_; // permutation
    ulong n_; // permutations of n elements
    ulong sw1_, sw2_; // indices of swapped elements
    ulong ct_; // count 5,4,3,2,1,(0); nonzero => easy cases
    [--snip--]

    The counter is set to 5 in the method first(). The update routine is
```
10.6: Lipski’s Minimal-change orders

10.6.1 Variants of Heap’s algorithm

Various algorithms similar to Heap’s method are given in Lipski’s paper [191], we take three of those
and add a similar one. The four orderings obtained for the permutations of five elements are shown

July 15, 2008 17:30
Page 251

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

Note that the routine only works for permutations of at least three elements. Usage of the class is shown
in [FXT: comb/perm-heap2-demo.cc]. The rate is about 190 million updates per second.

10.5.2 Computing just the swaps

If only the swaps are of interest we can simply omit all statements involving the permutation array p_.
The implementation is [FXT: class perm_heap2_swaps in comb/perm-heap2-swaps.h], usage of the class
is shown in [FXT: comb/perm-heap2-swaps-demo.cc]:

The update routine works at a rate about 381 million per second. Using arrays instead of pointers results
in a rate of about 387 million per second (5.7 cycles per update).

Heap’s algorithm and the optimization idea was taken from the excellent survey [240] which gives several
permutation algorithms and implementations in pseudo code.

10.6 Lipski’s Minimal-change orders
Figure 10.6-A: First half and last permutations of five elements as obtained by variants of Heap’s method. Next to the permutations the swaps are shown as \((x, y)\), a swap \((x, 0)\) is given as \(x\).
in figure 10.6-A. The leftmost order is Heap’s order. The implementation is given in [FXT: class perm_gray_lipski in comb/perm-gray-lipski.h], the variable \( r \) determines the order that is generated:

```cpp
class perm_gray_lipski
{
    ulong r_; // order (0<=r<4):

    bool next()
    {
        // increment mixed radix number:
        ulong j = 0;
        while ( d_[j]==j+1 ) { d_[j]=0; ++j; }
        if ( j<n_-1 ) // only if no overflow
        {
            const ulong d = d_[j];
            ulong x;
            switch ( r_ )
            {
                case 0: x = (j&1 ? 0 : d); break; // Lipski(9) == Heap
                case 1: x = (j&1 ? 0 : j-d); break; // Lipski(16)
                case 2: x = (j&1 ? j-1 : d); break; // Lipski(10)
                default: x = (j&1 ? j-1 : j-d); break; // not in Lipski’s paper
            }
            const ulong k = j+1;
            swap2(p_[k], p_[x]);
            d_[j] = d + 1;
            return true;
        }
        else return false; // j=n-1 for last permutation
    }
}
```

The top lines in figure 10.6-A repeat the statements in the switch-block. For three or less elements all orderings coincide, with \( n=4 \) elements the orderings for \( r=0 \) and \( r=2 \), and the orderings for \( r=1 \) and \( r=3 \) coincide. About 110 million permutations per second are generated [FXT: comb/perm-gray-lipski-demo.cc]. Optimizations similar to those for Heaps method should be obvious.

### 10.6.2 Variants of Wells’ algorithm

A Gray code for permutations given by Wells [274] is shown in the left of figure 10.6-B. The following implementation, following Lipski’s paper [191], includes two variants of the algorithm. We just give the crucial assignments in the method that computes the successor [FXT: class perm_gray_wells in comb/perm-gray-wells.h]:

```cpp
class perm_gray_wells
{
    switch ( r_ )
    {
        case 1: x = ( (j&1) || (d==0) ? 0 : d-1); break; // Lipski(14)
        case 2: x = ( (j&1) || (d==0) ? j : d-1); break; // Lipski(15)
        default: x = ( (j&1) || (d<=1) ? j : j-d); break; // Wells’ order == Lipski(8)
    }
}
```

Both expressions \( (d==0) \) can be changed to \( (d<=1) \) without changing the algorithm. More than 90 million permutations per second are generated [FXT: comb/perm-gray-wells-demo.cc].
Chapter 10: Permutations

10.7 Strong minimal-change order (Trotter’s algorithm)

10.7.1 Variant where smallest element moves most often

Figure 10.6-B shows the permutations of 4 elements in a strong minimal-change order: just two elements are swapped with each update and these are adjacent. Note that, in the sequence of the inverse permutations the swapped pair always consists of elements $x$ and $x + 1$. Also the first and last permutation differ by an adjacent transposition (of the last two elements). The ordering can be obtained by an interleaving process shown in figure 10.7-B. The first half of the permutations in this order are the reversals of the second half: the relative order of the two smallest elements is changed only with the transition just after the first half and reversal changes the order of these two elements. Mutual permutations lie $n! / 2$ positions apart.

A computer program to obtain all permutations in the shown order was given 1962 by H. F. Trotter [261] (see also [157] and [113]). However, the order was already known long before in connection with bell ringing, under the name Plain Changes, see [174].

We compute both the permutation and its inverse [FXT: class perm_trotter in comb/perm-trotter.h]:

```cpp
class perm_trotter
{
  public:
    ulong n_; // number of elements to permute
    ulong *x_; // permutation of {0, 1, ..., n-1}
    ulong *xi_; // inverse permutation
    ulong *d_; // auxiliary: directions
    ulong sw1_, sw2_; // indices of elements swapped most recently

  public:
    void perm_trotter(ulong n)
    {
      n_ = n;
      x_ = new ulong[n + 2];
      xi_ = new ulong[n];
      d_ = new ulong[n];
    }
};
```

Figure 10.6-B: Wells’ order for the permutations of four elements (left), and an order where most swaps are with the first position (right). Dots denote the element zero.
### Figure 10.7-A: The permutations of 4 elements in a strong minimal-change order (smallest element moves most often). Dots denote zeros.

<table>
<thead>
<tr>
<th>permutation</th>
<th>swap</th>
<th>inverse p.</th>
<th>direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [ . 1 2 3 ]</td>
<td>(3, 2)</td>
<td>[ . 1 2 3 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>1: [ 1 . 2 3 ]</td>
<td>(0, 1)</td>
<td>[ 1 . 2 3 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>2: [ 1 2 . 3 ]</td>
<td>(1, 2)</td>
<td>[ 2 . 1 3 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>3: [ 1 2 3 . ]</td>
<td>(2, 3)</td>
<td>[ 3 . 1 2 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>4: [ 2 1 . 3 ]</td>
<td>(0, 1)</td>
<td>[ 3 1 . 2 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>5: [ 2 . 1 . 3 ]</td>
<td>(3, 2)</td>
<td>[ 2 1 . 3 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>6: [ 2 . 1 3 . ]</td>
<td>(2, 1)</td>
<td>[ 2 . 1 3 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>7: [ 2 1 . 3 ]</td>
<td>(1, 0)</td>
<td>[ 2 1 3 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>8: [ 2 3 1 . ]</td>
<td>(0, 1)</td>
<td>[ 3 1 2 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>9: [ 2 . 3 1 ]</td>
<td>(3, 2)</td>
<td>[ 2 . 3 1 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>10: [ . 2 1 . 3 ]</td>
<td>(2, 1)</td>
<td>[ . 2 3 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>11: [ . 2 3 1 ]</td>
<td>(1, 0)</td>
<td>[ . 2 3 1 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>12: [ . 3 2 1 ]</td>
<td>(0, 1)</td>
<td>[ . 3 2 1 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>13: [ . 3 2 1 ]</td>
<td>(3, 2)</td>
<td>[ . 3 2 1 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>14: [ 3 . 2 1 ]</td>
<td>(2, 3)</td>
<td>[ 1 2 3 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>15: [ 3 2 1 . ]</td>
<td>(1, 0)</td>
<td>[ 3 2 1 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>16: [ 3 . 1 2 ]</td>
<td>(0, 1)</td>
<td>[ 3 . 1 2 ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>17: [ 3 1 2 . ]</td>
<td>(2, 0)</td>
<td>[ 3 1 2 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>18: [ 3 1 2 ]</td>
<td>(2, 3)</td>
<td>[ 1 3 2 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>19: [ 3 2 1 ]</td>
<td>(1, 0)</td>
<td>[ 3 2 1 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>20: [ 1 3 2 ]</td>
<td>(3, 2)</td>
<td>[ 3 2 1 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>21: [ 1 3 2 ]</td>
<td>(2, 0)</td>
<td>[ 1 3 2 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>22: [ 1 3 2 ]</td>
<td>(1, 0)</td>
<td>[ 1 3 2 . ]</td>
<td>+ + + +</td>
</tr>
<tr>
<td>23: [ . 1 3 2 ]</td>
<td>(0, 1)</td>
<td>[ . 1 3 2 ]</td>
<td>+ + + +</td>
</tr>
</tbody>
</table>

### Figure 10.7-B: Trotter’s construction as an interleaving process.

---

perm(4)==

|------------------ | perm(4)== |
|------------------ | [0, 1, 2, 3] |
| --> [0, 1, 2, 3] | [1, 0, 2, 3] |
| --> [1, 0, 2, 3] | [1, 2, 0, 3] |
| --> [1, 2, 0, 3] | [1, 2, 3, 0] |
| --> [1, 2, 3, 0] | [2, 1, 3, 0] |
| --> [2, 1, 3, 0] | [2, 0, 3, 0] |
| --> [2, 0, 3, 0] | [2, 0, 1, 3] |
| --> [2, 0, 1, 3] | [2, 0, 3, 1] |
| --> [2, 0, 3, 1] | [2, 0, 1, 3] |
| --> [2, 0, 1, 3] | [0, 2, 1, 3] |
| --> [0, 2, 1, 3] | [0, 2, 3, 1] |
| --> [0, 2, 3, 1] | [0, 2, 1, 3] |
| --> [0, 2, 1, 3] | [0, 1, 2, 3] |
| --> [0, 1, 2, 3] | [0, 1, 3, 2] |
| --> [0, 1, 3, 2] | [0, 1, 3, 2] |
| --> [0, 1, 3, 2] | [0, 1, 3, 2] |
| --> [0, 1, 3, 2] | [0, 1, 3, 2] |
| --> [0, 1, 3, 2] | [0, 1, 3, 2] |

[fxtbook draft of 2008-August-17]
Chapter 10: Permutations

```cpp
ulong sen = 0; // sentinel value minimal
x_[0] = x_[n_+1] = sen;
++x_; 
first();
}
[--snip--]
Note that sentinel elements are at the lower and higher end of the array for the permutation. For each element we store a direction-flag = ±1 in an array d_. Initially all are set to +1:

```cpp
void first()
{
  for (ulong i=0; i<n_; i++) xi_[i] = i;
  for (ulong i=0; i<n_; i++) x_[i] = i;
  for (ulong i=0; i<n_; i++) d_[i] = 1;
  sw1_ = n_ - 1; sw2_ = n_ - 2; // relative to last permutation
}
[--snip--]
To compute the successor, find the smallest element e1 whose neighbor e2 (left or right neighbor, according to the direction) is greater than e1. Swap the elements e1 with e2 and change the direction of all elements that could not be moved. The location of the elements, i1 and i2 are found with the inverse permutation, which has to be updated accordingly:

```cpp
bool next()
{
  // e1 is the element we try to move
  ulong i1 = xi_[e1]; // position of element e1
  ulong d = d_[e1]; // direction to move e1
  ulong i2 = i1 + d; // position to swap with
  ulong e2 = x_[i2]; // element to swap with
  if ( e1 < e2 ) // can we swap?
  {
    xi_[e1] = i2;
    xi_[e2] = i1;
    x_[i1] = e2;
    x_[i2] = e1;
    sw1_ = i1; sw2_ = i2;
    while ( e1-- ) d_[e1] = -d_[e1];
    return true;
  }
  first();
  return false;
}
```

The locations of the swap can be obtained with the method

```cpp
void get_swap(ulong &s1, ulong &s2) const
{
  s1=sw1_; s2=sw2_; }
```

The last permutation can be obtained via

```cpp
void last()
{
  for (ulong i=0; i<n_; i++) xi_[i] = i;
  for (ulong i=0; i<n_; i++) x_[i] = i;
  for (ulong i=0; i<n_; i++) d_[i] = -1UL;
  sw1_ = n_ - 1; sw2_ = n_ - 2; // relative to first permutation
  d_[sw1_] = +1;
  d_[sw2_] = +1;
  swap2(x_[sw1_], x_[sw2_]);
  swap2(xi_[sw1_], xi_[sw2_]);
}
```

The routine for the predecessor is obtained by adding one character to the routine next(), it's a minus:

```cpp
bool prev()
{
[--snip--]
ulong d = -d_[e1]; // direction to move e1 (NOTE: negated)
```
```
10.7: Strong minimal-change order (Trotter’s algorithm)

Well, we also changed the call \texttt{first()} to \texttt{last()}.

The routines \texttt{next()} and \texttt{prev()} generate about 137 million permutations per second. Figure 10.7-A was created with the program [FXT: \texttt{comb/perm-trotter-demo.cc}):

\begin{verbatim}
ulong n = 4;
perm_trotter P(n);
do {
  // visit permutation
} while ( P.next() );
\end{verbatim}

\subsection*{10.7.2 Optimized routines}

The element zero is moved most often, so we can treat that case separately [FXT: \texttt{comb/perm-trotter.h}]:

\begin{verbatim}
bool next()
{
  // most frequent case: e1 == 0
  ulong i1 = x_[0]; // position of element e1
  ulong d = d_[0]; // direction to move e1
  ulong i2 = i1 + d; // position to swap with
  ulong e2 = x_[i2]; // element to swap with
  if ( 0 < e2 ) // can we swap?
  {
    x_[0] = i2;
    x_[e2] = i1;
    x_[i1] = e2;
    x_[i2] = 0;
    sw1_ = i1; sw2_ = i2;
    return true;
  }
  for (ulong e1=1; e1<n_; ++e1) // note: start at e1=1
  [--snip--]
\end{verbatim}

The very same modification can be applied to the method \texttt{prev()}, only the minus has to be added:

\begin{verbatim}
ulong d = -d_[0]; // direction to move e1 (NOTE: negated)
\end{verbatim}

Now both methods compute about 174 million permutations per second, corresponding to less than 12.6 CPU cycles per update.

We can also treat the second most frequent case separately by adding the block:

\begin{verbatim}
bool next()
{
  // most frequent case: e1 == 0
  [--snip--]
  \}[--snip--]
  \}
  // second most frequent case: e1 == 1
  ulong i1 = x_[1]; // position of element e1
  ulong d = d_[1]; // direction to move e1
  ulong i2 = i1 + d; // position to swap with
  ulong e2 = x_[i2]; // element to swap with
  if ( 1 < e2 ) // can we swap?
  {
    x_[1] = i2;
    x_[e2] = i1;
    x_[i1] = e2;
    x_[i2] = 1;
    d_[0] = -d_[0]; // negate
    sw1_ = i1; sw2_ = i2;
    return true;
  }
}\end{verbatim}
22
23
24 for (ulong e1=2; e1<n_; ++e1) // note: start at e1=2
25 [--snip--]

With this modification the rate increases to about 179 million per second (12.3 cycles per update).

10.7.3 Variant where largest element moves most often

<table>
<thead>
<tr>
<th>permutation</th>
<th>swap</th>
<th>inverse p.</th>
<th>direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [ . 1 2 3 ]</td>
<td>(0, 1)</td>
<td>[. 1 2 3 ]</td>
<td>- - - -</td>
</tr>
<tr>
<td>1: [ . 1 3 2 ]</td>
<td>(3, 2)</td>
<td>[. 1 3 2 ]</td>
<td>- - - -</td>
</tr>
<tr>
<td>2: [ . 3 1 2 ]</td>
<td>(2, 1)</td>
<td>[. 2 3 1 ]</td>
<td>- - - -</td>
</tr>
<tr>
<td>3: [ . 3 1 2 ]</td>
<td>(1, 0)</td>
<td>[ 1 3 2 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>4: [ 3 . 2 1 ]</td>
<td>(3, 2)</td>
<td>[ 3 1 2 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>5: [ . 3 2 1 ]</td>
<td>(0, 1)</td>
<td>[. 3 2 1 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>6: [ . 2 3 1 ]</td>
<td>(1, 2)</td>
<td>[. 2 3 1 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>7: [ 2 . 1 3 ]</td>
<td>(2, 3)</td>
<td>[ . 2 1 3 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>8: [ 2 . 1 3 ]</td>
<td>(1, 0)</td>
<td>[ 1 2 . 3 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>9: [ 2 . 3 1 ]</td>
<td>(3, 2)</td>
<td>[ 1 3 . 2 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>10: [ 2 3 . 1 ]</td>
<td>(2, 1)</td>
<td>[ 2 3 . 1 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>11: [ 3 2 . 1 ]</td>
<td>(1, 0)</td>
<td>[ 2 3 1 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>12: [ 3 2 1 . ]</td>
<td>(3, 2)</td>
<td>[ 3 2 1 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>13: [ 3 2 1 . ]</td>
<td>(0, 1)</td>
<td>[ 3 2 . 1 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>14: [ 2 1 3 . ]</td>
<td>(1, 2)</td>
<td>[ 3 1 . 2 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>15: [ 2 1 3 . ]</td>
<td>(2, 3)</td>
<td>[ 2 1 3 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>16: [ 2 . 1 3 ]</td>
<td>(0, 1)</td>
<td>[ 2 . 1 3 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>17: [ 1 2 . 3 ]</td>
<td>(3, 2)</td>
<td>[ 1 2 3 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>18: [ 1 3 . 2 ]</td>
<td>(2, 1)</td>
<td>[ 1 3 2 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>19: [ 3 1 . 2 ]</td>
<td>(1, 0)</td>
<td>[ 3 1 2 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>20: [ 3 1 . 2 ]</td>
<td>(2, 3)</td>
<td>[ 3 1 2 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>21: [ 1 3 . 2 ]</td>
<td>(0, 1)</td>
<td>[ 1 3 . 2 ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>22: [ 1 3 2 . ]</td>
<td>(1, 2)</td>
<td>[ 1 3 2 . ]</td>
<td>- - - +</td>
</tr>
<tr>
<td>23: [ 1 2 3 . ]</td>
<td>(2, 3)</td>
<td>[ 1 2 3 . ]</td>
<td>- - - +</td>
</tr>
</tbody>
</table>

Figure 10.7-C: The permutations of 4 elements in a strong minimal-change order (largest element moves most often). Dots denote zeros.

A variant of the algorithm moves the largest element most often as shown in figure 10.7-C (created with [FXT: comb/perm-trotter-lg-demo.cc]). Only a few modifications have to be made to the code [FXT: class perm_trotter_lg in comb/perm-trotter-lg.h]. The sentinel needs to be greater than all elements of the permutations, and the directions start with minus one:

```cpp
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
17 18
19 20
21
```

```cpp
1 class perm_trotter_lg
2 {
3 [--snip--]
4 public:
5     perm_trotter_lg(ulong n)
6     {
7         ulong sen = n_; // sentinel value maximal
8         x_[0] = x_[-n_+1] = sen;
9         ++x_;
10         first();
11     }
12     }
13     }
14     void first()
15     {
16         for (ulong i=0; i<n_; i++) x_[i] = i;
17         for (ulong i=0; i<n_; i++) x_[i] = i;
18         for (ulong i=0; i<n_; i++) d_[i] = -1UL;
19         sw1_ = 0; sw2_ = 1; // relative to last permutation
20     }
21     }
```

In the update routine we look for the largest element whose neighbor is smaller than itself:
bool next()
{
    ulong e1 = n_; // e1 is the element we try to move
    while ( e1-- )
    {
        ulong i1 = xi_[e1]; // position of element e1
        ulong d = d_[e1]; // direction to move e1
        ulong i2 = i1 + d; // position to swap with
        ulong e2 = x_[i2]; // element to swap with
        if ( e1 > e2 ) // can we swap?
        {
            xi_[e1] = i2;
            xi_[e2] = i1;
            x_[i1] = e2;
            x_[i2] = e1;
            sw1_ = i1; sw2_ = i2;
            while ( ++e1<n_ ) d_[e1] = -d_[e1];
            return true;
        }
    }
    first();
    return false;
}

The last permutation can be obtained via

void last()
{
    for (ulong i=0; i<n_; i++) xi_[i] = i;
    for (ulong i=0; i<n_; i++) x_[i] = i;
    for (ulong i=0; i<n_; i++) d_[i] = +1;
    sw1_ = 0; sw2_ = 1; // relative to first permutation
    d_[sw1_] = -1UL;
    d_[sw2_] = -1UL;
    swap2(x_[sw1_], x_[sw2_]);
    swap2(xi_[sw1_], xi_[sw2_]);
}

The method to compute the predecessor is obtained as before. The routines next() and prev() generate about 126 million permutations per second.

10.8 Minimal-change orders from factorial numbers

10.8.1 Permutations with falling factorial numbers

The Gray code for the mixed radix numbers with falling factorial base allows the computation of the permutations in minimal-change order in an elegant way. See figure [10.8-A] which was created with the program [FXT: comb/perm-gray-ffact2-demo.cc]. The algorithm is implemented in [FXT: class perm_gray_ffact2 in comb/perm-gray-ffact2.h]:

class perm_gray_ffact2
{
public:
    mixedradix_gray2 *mrg_; // loopless routine
    ulong n_; // number of elements to permute
    ulong *x_; // current permutation (of {0, 1, ..., n-1})
    ulong *ix_; // inverse permutation
    ulong sw1_, sw2_; // indices of elements swapped most recently

public:
    perm_gray_ffact2(ulong n)
    : n_(n)
    {
        x_ = new ulong[n_];
        ix_ = new ulong[n_];
        mrg_ = new mixedradix_gray2(n_-1, 0); // falling factorial base
Chapter 10: Permutations

Figure 10.8-A: Permutations in minimal-change order (left) and Gray code for mixed radix numbers with falling factorial base. The two rightmost columns give the place of change with the mixed radix numbers and its direction. Whenever digit \( p \) (\( = \text{pos} \)) changes by \( d = \pm 1 \) (\( = \text{dir} \)) in the mixed radix sequence the element \( p \) of the permutation is swapped with its right \( (d = +1) \) or left \( (d = -1) \) neighbor.

```cpp
17 void first() {
18   mrg_->first();
19   for (ulong k=0; k<n_; ++k) x_[k] = ix_[k] = k;
20   sw1_=n_-1; sw2_=n_-2;
21 }
22
23 bool next() {
24   // Compute next mixed radix number in Gray code order:
25   if ( false == mrg_->next() ) { first(); return false; }
26   const ulong j = mrg_->pos(); // position of changed digit
27   const int d = mrg_-dir(); // direction of change
28   // swap:
29   const ulong x1 = j; // element j
30   const ulong i1 = ix_[x1]; // position of j
31   const ulong x2 = i1 + d; // neighbor
32   // swap2(x_[i1], x_[i2]); // position of neighbor
33   const ulong i1 = x2; // swap2(x_[i1], x_[i2]);
34   const ulong i2 = i1; // swap2(ix_[x1], ix_[x2]);
35   sw1_=i1; sw2_=i2;
36   return true;
37 }
```

The crucial part is the computation of the successor:

The class uses the loopless algorithm for the computation of the mixed radix Gray code, so it is loopless itself. An alternative (CAT) algorithm is implemented in [FXT: class perm_gray_ffact in comb/perm-gray-ffact.h], we give just the routine for the successor:

```cpp
1 private:
2   void swap(ulong j, ulong im) // used with next() and prev()
3   {
4     const ulong x1 = j; // element j
```
10.8: Minimal-change orders from factorial numbers

5  const ulong i1 = ix_[x1]; // position of j
6  const ulong i2 = i1 + im; // position of neighbor
7  x_[i1] = x2; x_[i2] = x1; // swap2(x_[i1], x_[i2]);
8  ix_[x1] = i2; ix_[x2] = i1; // swap2(ix_[x1], ix_[x2]);
9  sw1_=i1; sw2_=i2;
10  }
11
12 public:
13  bool next()
14  {
15    ulong j = 0;
16    ulong m1 = n_ - 1; // nine in falling factorial base
17    ulong ij;
18    while ( (ij=i_[j]) )
19    {
20      ulong im = i_[j];
21      ulong dj = d_[j] + im;
22      if ( dj>m1 ) // ^= if ( (dj>m1) || ((long)dj<0) )
23      {
24        i_[j] = -ij;
25      }
26      else
27      {
28        d_[j] = dj;
29        swap(j, im);
30        return true;
31      }
32    --m1;
33    ++j;
34  }
35  return false;
36  }

The routine for the predecessor (prev()) is obtained by replacing the statement ulong im = i_[j];
by ulong im = -i_[j];. The loopless routine computes about 80 million permutations per second, the
CAT version about 110 million per second [FXT: comb/perm-gray-ffact-demo.cc]. Both are slower than
the implementations given in section 10.7 on page 254.

10.8.2 Permutations with rising factorial numbers

Figure 10.8-B shows a Gray code for permutations based on the Gray code for numbers in rising factorial
base. The ordering coincides with Heap’s algorithm (see section 10.5 on page 249) for up to four ele-
ments. A recursive construction for the order is shown in figure 10.8-C. The figure was created with the
program [FXT: comb/perm-gray-rfact-demo.cc] (see also [FXT: comb/fact2perm-demo.cc]). A constant
amortized time (CAT) algorithm for generating the permutations is [FXT: class perm_gray_rfact in
comb/perm-gray-rfact.h]:

class perm_gray_rfact
{
public:
  mixedradix_gray *M_; // loopless routine
  ulong n_; // number of elements to permute
  ulong *x_; // current permutation (of {0, 1, ..., n-1})
  ulong *ix_; // inverse permutation
  ulong sw1_, sw2_; // indices of elements swapped most recently

public:
  perm_gray_rfact(ulong n)
  : n_(n)
  {
    x_ = new ulong[n_];
    ix_ = new ulong[n_];
    M_ = new mixedradix_gray(n_-1, 1); // rising factorial base
    first();
  }

  [--snip--]
  void first()
  {

}
Figure 10.8-B: Permutations in minimal-change order (left) and Gray code for mixed radix numbers with rising factorial base. For even \( n \), the first and last permutations are cyclic shifts by one of each other.

Figure 10.8-C: Recursive construction of the permutations.
Let \( j \geq 0 \) be the position of the digit changed with incrementing the mixed radix number, and \( d = \pm 1 \) the increment or decrement of that digit. The swap to obtain the successor permutation swaps the element \( x_1 \) at position \( j + 1 \) with the element \( x_2 \) where \( x_2 \) is lying to the left of \( x_1 \) and it is the greatest element smaller than \( x_1 \) for \( d > 0 \), and the smallest element greater than \( x_1 \) for \( d < 0 \):

```java
bool next()
{
    // Compute next mixed radix number in Gray code order:
    if ( false == M_->next() ) { first(); return false; }
    ulong j = M_->pos(); // position of changed digit

    if ( j<1 ) // easy cases: swap == (0,j+1)
    {
        const ulong i2 = j+1; // i1 == 0
        const ulong x1 = x_[0], x2 = x_[i2];
        x_[0] = x2; x_[i2] = x1; // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2; ix_[x2] = 0; // swap2(ix_[x1], ix_[x2]);
        sw1_=0; sw2_=i2;
        return true;
    }
    else
    {
        ulong i1 = j+1, i2 = i1;
        ulong x1 = x_[i1], x2;
        int d = M_->dir(); // direction of change
        if ( d>0 ) // in the inverse permutation search first smaller element left:
        {
            ulong xt = x_[t];
            if ( (xt < x1) && (xt >= x2) ) { i2=t; x2=xt; }
        }
        else // in the inverse permutation search first larger element:
        {
            for (ulong t=0; t<i1; ++t) // search minimal larger element
            {
                ulong xt = x_[t];
                if ( (xt > x1) && (xt <= x2) ) { i2=t; x2=xt; }
            }
        }
        x_[i1] = x2; x_[i2] = x1; // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2; ix_[x2] = i1; // swap2(ix_[x1], ix_[x2]);
        sw1_=i2; sw2_=i1;
        return true;
    }
}
```

There is a slightly more efficient algorithm to compute the successor using the inverse permutations:
The method is chosen by defining SUCC_BY_INV in the file [FXT: comb/perm-gray-rfact.h]. About 68 million permutations per second are generated, about 58 million with the first method.

### 10.8.3 Permutations with permuted factorial numbers

<table>
<thead>
<tr>
<th>permutation</th>
<th>swap</th>
<th>xfact</th>
<th>pos dir</th>
<th>inv.perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [1 2 3 4]</td>
<td>[1 . . .]</td>
<td>[ . . .]</td>
<td>0 +1</td>
<td>[1 2 3 4]</td>
</tr>
<tr>
<td>1: [1 2 3 4]</td>
<td>(0, 1)</td>
<td>[ . 1 .]</td>
<td>1 +1</td>
<td>[1 2 3 4]</td>
</tr>
<tr>
<td>2: [2 1 3 4]</td>
<td>(0, 2)</td>
<td>[1 . .]</td>
<td>0 -1</td>
<td>[1 2 3 4]</td>
</tr>
<tr>
<td>3: [1 2 3 4]</td>
<td>(0, 2)</td>
<td>[ . 2 .]</td>
<td>1 +1</td>
<td>[1 2 3 4]</td>
</tr>
<tr>
<td>4: [2 1 3 4]</td>
<td>(0, 1)</td>
<td>[ . . 1]</td>
<td>0 +1</td>
<td>[2 1 3 4]</td>
</tr>
<tr>
<td>5: [2 1 3 4]</td>
<td>(3, 4)</td>
<td>[1 2 1 .]</td>
<td>2 +1</td>
<td>[2 1 3 4]</td>
</tr>
<tr>
<td>6: [2 1 3 4]</td>
<td>(0, 1)</td>
<td>[1 2 .]</td>
<td>0 -1</td>
<td>[2 1 3 4]</td>
</tr>
<tr>
<td>7: [1 2 3 4]</td>
<td>(0, 1)</td>
<td>[ . 1 2]</td>
<td>0 -1</td>
<td>[2 1 4 3]</td>
</tr>
</tbody>
</table>

[---snip---]

91: [3 4 2 1 .] | (0, 1) | [ . 2 4 3] | 0 -1 | [4 3 2 1]
92: [2 4 3 1 .] | (0, 2) | [ . 1 4 3] | 1 -1 | [4 3 2 1]
93: [4 2 3 1 .] | (0, 1) | [1 4 . 3] | 0 +1 | [4 3 1 2]
94: [3 2 4 1 .] | (0, 2) | [ . 4 3 1] | 1 -1 | [4 3 1 2]
95: [2 3 4 1 .] | (0, 1) | [ . 4 3] | 0 -1 | [4 3 1 2]
96: [2 3 4 . 1] | (3, 4) | [ . 3 3] | 2 -1 | [3 4 1 2]
97: [3 2 4 . 1] | (0, 1) | [1 3 3] | 0 +1 | [3 4 1 2]

[---snip---]

106: [3 1 4 . 2] | (0, 2) | [1 . 2 3] | 1 -1 | [3 1 4 2]
107: [1 3 4 . 2] | (0, 1) | [ . 2 3] | 0 -1 | [3 4 . 1 2]
108: [1 2 4 . 3] | (1, 4) | [1 1 . 3] | 2 -1 | [3 4 1 2]
109: [2 1 4 . 3] | (0, 1) | [1 1 3] | 0 +1 | [3 4 1 2]
110: [4 1 2 . 3] | (0, 2) | [ . 1 1 3] | 1 +1 | [3 1 2 4]
111: [1 4 2 . 3] | (0, 1) | [1 1 . 3] | 0 -1 | [3 2 4 1]
112: [2 4 1 . 3] | (0, 2) | [ . 2 1 3] | 1 +1 | [3 2 4 1]
113: [4 2 1 . 3] | (0, 1) | [1 2 1 3] | 0 +1 | [3 1 2 4]
114: [3 2 1 . 4] | (0, 4) | [1 2 . 3] | 2 -1 | [3 2 1 4]
115: [2 3 1 . 4] | (0, 1) | [ . 2 3] | 0 -1 | [3 2 1 4]
116: [1 3 2 . 4] | (0, 2) | [1 1 . 3] | 1 -1 | [3 2 1 4]
117: [3 1 2 . 4] | (0, 1) | [1 1 . 3] | 0 +1 | [3 2 1 4]
118: [2 1 3 . 4] | (0, 2) | [ . 3 . 1] | 1 -1 | [3 1 2 4]
119: [1 2 3 . 4] | (0, 1) | [ . . 3] | 0 -1 | [3 1 2 4]

Figure 10.8-D: Permutations with mixed radix numbers with radix vector [2, 3, 5, 4].

The rising and falling factorial numbers are special cases of factorial numbers with permuted digits. We give a method to compute the Gray code for permutations from the Gray code for permuted (falling) factorial numbers. A permutation of the radices determines how often a digit at any position is changed: the leftmost changes most often, the rightmost least often. The permutations corresponding to the mixed radix numbers with radix vector [2, 3, 5, 4], the falling factorial last two radices swapped, is shown in figure 10.8-D [FXT: comb/perm-gray-rot1-demo.cc]. The desired property of this ordering is that the last permutation is as close to a cyclic shift by one of the first as possible. With even \( n \) the Gray code with the falling factorial basis the last permutation is a shift by one. With odd \( n \) no such Gray code exists: the total number of transpositions with any Gray code is odd for all \( n > 1 \), but the cyclic rotation by one corresponds to an even number of transpositions. The best we can get is that the first \( e \) elements where \( e \leq n \) is the greatest possible even number. For example,

\[
\begin{align*}
\text{first} & : [0 \ 1 \ 2 \ 3 \ 4 \ 5] \\
\text{last} & : [1 \ 2 \ 3 \ 4 \ 5 \ 0]
\end{align*}
\]

\( n=6 \):
We use this ordering to show the general method [FXT: class perm_gray_rot1 in comb/perm-gray-rot1.h]:

```cpp
class perm_gray_rot1
{
public:
   mixedradix_gray *M_; // Gray code for factorial numbers
}```
10.9 Orders where the smallest element always moves right

An ordering for the permutations where the first element always moves right can be obtained by the
interleaving process shown in figure 10.9-A. The process is the same as that for Trotter’s order shown in
figure 10.7-B on page 255, but without changing the directions. The second half of the permutations is the
reversed list of the reversed permutations in the first half. The permutations are shown in figure 10.9-B.

---snip---
Chapter 10: Permutations

Figure 10.9-A: Interleaving process to obtain all permutations by right moves.

Figure 10.9-B: All permutations of 4 elements and falling factorial numbers used to update the permutations. Dots denote zeros.
they are the inverses of the permutations corresponding to the falling factorial numbers, see figure \[\text{10.3-A}\] on page 238. An implementation is \[\text{FXT: class perm\_mv0 in comb/perm-mv0.h}].

```cpp
class perm\_mv0{
  public:
    ulong *d_; // mixed radix digits with radix = [n-1, n-2, n-3, ..., 2]
    ulong *x_; // permutation
    ulong ect_; // counter for easy case
    ulong n_; // permutations of n elements
  public:
    perm\_mv0(ulong n)
    // Must have n>=2
    {
      n_ = n;
      d_ = new ulong[n_];
      d_[n-1] = 1; // sentinel (must be nonzero)
      x_ = new ulong[n_];
      first();
    }
    void first()
    {
      for (ulong k=0; k<n_; ++k) x_[k] = k;
      for (ulong k=0; k<n_-1; ++k) d_[k] = 0;
      ect_ = 0;
    }

    bool next()
    {
      if ( ++ect_ < n_ ) // easy case
      {
        swap2(x_[ect_], x_[ect_-1]);
        return true;
      }
      else
      {
        ect_ = 0;
        ulong j = 1;
        ulong m1 = n_ - 2; // nine in falling factorial base
        while ( d_[j]==m1 ) // find digit to increment
        {
          d_[j] = 0;
          --m1;
          ++j;
        }
        if ( j==n_-1 ) return false; // current permutation is last
        const ulong dj = d_[j];
        d_[j] = dj + 1;
        // element at d[j] moves one position to the right:
        swap2( x_[dj], x_[dj+1] );
        { // move n-j elements to end:
          ulong s = n_-j, d = n_;
Chapter 10: Permutations

The routine generates about 160 million permutations per second [FXT: comb/perm-mv0-demo.cc].

10.9.2 Ives’ algorithm

<table>
<thead>
<tr>
<th>permutation</th>
<th>inv. perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: [ . 1 2 3 ]</td>
<td>[ . 1 2 3 ]</td>
</tr>
<tr>
<td>2: [ 1 . 2 3 ]</td>
<td>[ 1 . 2 3 ]</td>
</tr>
<tr>
<td>3: [ 1 2 . 3 ]</td>
<td>[ 2 . 1 3 ]</td>
</tr>
<tr>
<td>4: [ 1 2 3 . ]</td>
<td>[ 3 . 1 2 ]</td>
</tr>
<tr>
<td>5: [ . 2 3 1 ]</td>
<td>[ . 3 1 2 ]</td>
</tr>
<tr>
<td>6: [ 2 . 3 1 ]</td>
<td>[ 1 3 . 2 ]</td>
</tr>
<tr>
<td>7: [ 2 3 . 1 ]</td>
<td>[ 2 3 . 1 ]</td>
</tr>
<tr>
<td>8: [ 2 3 1 . ]</td>
<td>[ 3 2 . 1 ]</td>
</tr>
<tr>
<td>9: [ 3 1 2 . ]</td>
<td>[ 2 3 1 . ]</td>
</tr>
<tr>
<td>10: [ 3 1 2 ]</td>
<td>[ 1 2 3 ]</td>
</tr>
<tr>
<td>11: [ 3 1 . 2 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>12: [ 3 1 2 . ]</td>
<td>[ 3 1 2 . ] &lt;&lt; only update with more</td>
</tr>
<tr>
<td>13: [ . 2 1 3 ]</td>
<td>[ . 2 1 3 ] &lt;&lt; than one transposition</td>
</tr>
<tr>
<td>14: [ 2 . 1 3 ]</td>
<td>[ 1 2 . 3 ]</td>
</tr>
<tr>
<td>15: [ 2 1 . 3 ]</td>
<td>[ 2 1 . 3 ]</td>
</tr>
<tr>
<td>16: [ 2 1 3 . ]</td>
<td>[ 3 1 . 2 ]</td>
</tr>
<tr>
<td>17: [ 1 . 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
</tr>
<tr>
<td>18: [ 1 3 . 2 ]</td>
<td>[ 1 . 3 2 ]</td>
</tr>
<tr>
<td>19: [ 1 3 2 . ]</td>
<td>[ 2 . 3 1 ]</td>
</tr>
<tr>
<td>20: [ 1 3 2 . ]</td>
<td>[ 3 . 2 1 ]</td>
</tr>
<tr>
<td>21: [ . 3 2 1 ]</td>
<td>[ 1 3 2 . ]</td>
</tr>
<tr>
<td>22: [ 3 2 1 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>23: [ 3 2 . 1 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>24: [ 3 2 1 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
</tbody>
</table>

Figure 10.9-C: All permutations of 4 elements in an order by Ives.

An ordering where most of the moves are a move by one to the right of the smallest element is shown in figure 10.9-C. With \( n \) elements only one in \( n(n-1) \) moves is more than a transposition (only the update from 12 to 13 in figure 10.9-C). The second half of the list of permutations is the reversed list of the reversed permutations in the first half. The algorithm, given by Ives [154], is implemented in [FXT: class perm_ives in comb/perm-ives.h]:

```cpp
1 class perm_ives
2 {
3 public:
4   ulong *p_;  // permutation
5   ulong *ip_; // inverse permutation
6   ulong n_;  // permutations of n elements
7 }
8 public:
9   perm_ives(ulong n)
10  // Must have: n >= 2
11 {
12     n_ = n;
13     p_ = new ulong[n_];
14     ip_ = new ulong[n_];
```
The computation of the successor is

```c
bool next()
{
    ulong e1 = 0, u = n_ - 1;
    do
    {
        const ulong i1 = ip_[e1];
        const ulong i2 = (i1==u ? e1 : i1+1);
        const ulong e2 = p_[i2];
        p_[i1] = e2;  p_[i2] = e1;
        ip_[e1] = i2; ip_[e2] = i1;
        if ( (p_[e1]!=e1) || (p_[u]!=u) ) return true;
        ++e1;
        --u;
    } while ( u > e1 );
    return false;
}
```

The rate of generation is about 180 M/s [FXT: comb/perm-ives-demo.cc]. Using arrays instead of pointers increases the rate to about 190 M/s.

As the easy case with the update (when just the first element is moved) occurs so often it is natural to create an extra branch for it. When the define for `PERM_IVES_OPT` is made before the class definition then a counter is created:

```c
class perm_ives
{
    ulong ctm_; // aux: counter for easy case
    ulong ctm0_; // aux: start value of ctm == n*(n-1)-1

    bool next()
    {
        if ( ctm__-- ) // easy case
        {
            const ulong i1 = ip_[0]; // e1 == 0
            const ulong i2 = (i1==n_-1 ? 0 : i1+1);
            const ulong e2 = p_[i2];
            p_[i1] = e2;  p_[i2] = 0;
            ip_[0] = i2; ip_[e2] = i1;
            return true;
        }
        ctm_ = ctm0_;
    } // rest as before
```

When arrays are used a minimal speedup is obtained (rate 192 M/s), when pointers are used, the effect is a notable slowdown (rate 163 M/s).

The greatest speedup is obtained by a modified condition in the loop:

```c
if ( (p_[e1]!e1) || (p_[u]!u) ) return true;
// same as: if ( (p_[e1]!=e1) || (p_[u]!=u) ) return true;
```

The rate is increased to almost 194 M/s. This optimization is activated by defining `PERM_IVES_OPT2`. 

[fxtbook draft of 2008-August-17]
Figure 10.10-A: Permutations of 4 elements in single track order. Dots denote zeros.

Figure 10.10-B: Construction of the single track order for permutations of 4 elements.
10.10 Single track orders

Figure [10.10-A] shows a single track order for the permutations of four elements. Each column in the list of permutations is a cyclic shift of the first column. A recursive construction for the ordering is shown in figure [10.10-B]. The figure was created with the program [FXT: comb/perm-st-demo.cc] which uses [FXT: class perm_st in comb/perm-st.h]:

```cpp
class perm_st {
public:
  ulong *d_; // mixed radix digits with radix = [2, 3, 4, ..., n-1, (sentinel=-1)]
  ulong *p_; // permutation
  ulong *pi_; // inverse permutation
  ulong n_; // permutations of n elements

public:
  perm_st(ulong n) {
    n_ = n;
    d_ = new ulong[n_];
    p_ = new ulong[n_];
    pi_ = new ulong[n_];
    d_[n-1] = -1UL; // sentinel
    first();
  }

  const ulong *data() const { return p_; }
  const ulong *invdata() const { return pi_; }

  void first() {
    for (ulong k=0; k<n_-1; ++k) d_[k] = 0;
    for (ulong k=0, e=0; k<n_; ++k) {
      p_[k] = e;
      pi_[e] = k;
      e = next_enup(e, n_-1);
    }
  }

  bool next() {
    // increment mixed radix number:
    ulong j = 0;
    while ( (d_[j]==j+1) ) { d_[j]=0; ++j; }
    if ( j==n_-1 ) return false; // current permutation is last
    ++d_[j];
    for (ulong e1=n_-2-j, e2=e1+1; e2<n_; e1+=2, e2+=2) {
      const ulong i1 = pi_[e1]; // position of element e1
      const ulong i2 = pi_[e2]; // position of element e2
      pi_[e1] = i2;
      pi_[e2] = i1;
    }
  }
};
```

The first permutation is in enup order (see section 6.5.1 on page 186):

```
const ulong *data() const { return p_; }
const ulong *invdata() const { return pi_; }

void first() {
  for (ulong k=0; k<n_-1; ++k) d_[k] = 0;
  for (ulong k=0, e=0; k<n_; ++k) {
    p_[k] = e;
    pi_[e] = k;
    e = next_enup(e, n_-1);
  }
}
```

The swap with the inverse permutations are determined by the rightmost position \( j \) changing with mixed radix counting with rising factorial base. We write \(-1\) for the last element, \(-2\) for the second last, and so on:

\[
\begin{align*}
  j & \quad \text{swaps} \\
  0: & (-2,-1) \\
  1: & (-3,-2) \\
  2: & (-4,-3) (-2,-1) \\
  3: & (-5,-4) (-3,-2) \\
  4: & (-6,-5) (-4,-3) (-2,-1) \\
  5: & (-7,-6) (-5,-4) (-3,-2) \\
  \vdots & \vdots \\
  j: & (-j-2, -j-1) \ldots (-2-(j\%1), -1-(j\%1))
\end{align*}
\]

The computation of the successor is CAT:

```cpp
bool next() {
    // increment mixed radix number:
    ulong j = 0;
    while ( (d_[j]==j+1) ) { d_[j]=0; ++j; }
    if ( j==n_-1 ) return false; // current permutation is last
    ++d_[j];
    for (ulong e1=n_-2-j, e2=e1+1; e2<n_; e1+=2, e2+=2) {
      const ulong i1 = pi_[e1]; // position of element e1
      const ulong i2 = pi_[e2]; // position of element e2
      pi_[e1] = i2;
      pi_[e2] = i1;
    }
}
```

[fxtbook draft of 2008-August-17]
Chapter 10: Permutations

p_[i1] = e2;
p_[i2] = e1;
}
return true;
}

All swaps with the inverse permutations are of adjacent pairs. The reversals of the first half of all permutations lie in the second half, the reversal of the $k$-the permutation lies at position $n! - 1 - k$

<table>
<thead>
<tr>
<th>permutation</th>
<th>inv. perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [ . 1 2 3 ]</td>
<td>[ . . ]</td>
</tr>
<tr>
<td>1: [ . 1 3 2 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>2: [ . 2 3 1 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>3: [ . 3 2 1 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>4: [ . 3 1 2 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>5: [ . 2 1 3 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>6: [ . 1 3 2 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>7: [ . 1 2 3 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>8: [ . 1 3 2 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>9: [ . 2 3 1 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>10: [ . 2 1 3 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>11: [ . 3 2 1 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>12: [ . 3 1 2 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>13: [ . 1 3 2 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>14: [ . 1 3 2 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>15: [ . 3 1 2 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>16: [ . 3 2 1 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>17: [ . 3 1 2 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>18: [ . 3 1 2 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>19: [ . 2 3 1 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>20: [ . 2 3 1 ]</td>
<td>[ . 1 ]</td>
</tr>
<tr>
<td>21: [ . 1 2 3 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>22: [ . 1 2 3 ]</td>
<td>[ . 2 ]</td>
</tr>
<tr>
<td>23: [ . 1 2 3 ]</td>
<td>[ . 2 ]</td>
</tr>
</tbody>
</table>

Figure 10.10-C: Permutations of 4 elements in single track order starting with the identical permutation.

The single track property is independent of the first permutation, one can start with the trivial permutation using

```c
void first_id() // start with identical permutation
{
    for (ulong k=0; k<n_-1; ++k) d_[k] = 0;
    for (ulong k=0; k<n_; ++k) p_[k] = pi_[k] = k;
}
```

The ordering obtained is shown in figure 10.10-C. The reversal of the $k$-the permutation lies at position $(n!/2 + k$. About 85 million permutations per second can be generated.

10.10.1 Construction of all single track orders

```
112233
231312 <-- permutations of 3 elements in lex order (columns)
323121
00000 112233 231312 323121 <-- concatenate rows and prepend new element
00000 112233 231312 323121 <-- shift 0
323121 00000 112233 231312 <-- shift 6
231312 323121 00000 112233 <-- shift 12
112233 231312 323121 00000 <-- shift 18
```

Figure 10.10-D: Construction of a single track order for permutations of 4 elements from an arbitrary ordering of the permutations of 3 elements.

A construction for a single track order of $n+1$ elements from an arbitrary ordering of $n$ elements is shown in figure 10.10-D (for $n = 3$ and lexicographic order). Thereby we obtain as many single track orders
10.10: Single track orders

<table>
<thead>
<tr>
<th>single track ordering</th>
<th>modified single track ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>112233 321312 323121</td>
<td>21.113 1.333 212.1 332.22</td>
</tr>
<tr>
<td>323121 231312 231312</td>
<td>1.333 212.1 332.22 21.113</td>
</tr>
<tr>
<td>231312 323121 112233</td>
<td>.212.1 332.22 21.113 1.333</td>
</tr>
<tr>
<td>112233 231312 323121</td>
<td>...... 210321 &lt;= cyclic shifts</td>
</tr>
</tbody>
</table>

Figure 10.10-E: In each of the first $(n-1)!$ permutations in a single track ordering (first block left) an arbitrary rotation can be applied (first block right), leading to a different single track ordering.

for the permutations of $n$ elements as there are orders of the permutations of $n-1$ elements, namely $(n-1)!$. One can apply cyclic shifts in each blocks as shown in figure 10.10-E. The shifts in the first $(n-1)!$ positions (first blocks in the figure) determine the shifts for the remaining permutations. Now there are $n$ different cyclic shifts in each position. Indeed all single track orderings are of this form, so their number is

$$N_s(n) = ((n-1)!) n^{(n-1)!} \quad (10.10-1)$$

The number of single track orders that start with the identical permutation, and where the $k$-th run of $(n-1)!$ elements starts with $k$ (and thereby all shifts between successive tracks are left shifts by $(n-1)!$ positions) is

$$N_s(n)/n! = ((n-1)! - 1)! n^{(n-1)! - 1} \quad (10.10-2)$$

10.10.2 A single track Gray code

A Gray code for permutations that has the single track property can be constructed by using a Gray code for the permutations of $n-1$ elements if the first and last permutation are cyclic shifts by one position of each other. Such Gray codes exist for even lengths only. Figure 10.10-F shows a single track Gray code for $n = 5$. For even $n$ we use a Gray code where all but the last element are cyclically shifted between the first and last permutation. Such a Gray code is given in section 10.8.3 on page 264. The resulting single track order is as close to a Gray code as possible, just $n-1$ extra transpositions occur for all permutation of $n$ elements, see figure 10.10-G. The listings were created with the program [FXT: comb/perm-st-gray-demo.cc] which uses [FXT: class perm_gray in comb/perm-st-gray.h].
Figure 10.10-G: The single track ordering for odd $n$ with the least number of transpositions contains $n - 1$ extra transpositions. Here the non-Gray transitions are cycles between the elements $0$, $4$, and $5$.

We define two auxiliary routine to swap elements by their value and by their positions:

```cpp
class perm_st_gray
{
    public:
        perm_gray_rot1 *G; // underlying permutations
        ulong *x_; // permutation
        ulong *ix_; // inverse permutation
        ulong n_; // number of elements
        ulong act_; // count cyclic shifts
    }

    public:
        perm_st_gray(ulong n)
        // Must have n>=2
        {
            n_ = (n>=2 ? n : 2);
            G = new perm_gray_rot1(n-1);
            x_ = new ulong[n_];
            ix_ = new ulong[n_];
            first();
        }

    void first()
    {
        G->first();
        for (ulong j=0; j<n_; ++j) ix_[j] = x_[j] = j;
        act_ = n_;
    }

    void swap_elements(ulong x1, ulong x2)
    {
        const ulong i1 = ix_[x1], i2 = ix_[x2];
        x_[i1] = x2; x_[i2] = x1; // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2; ix_[x2] = i1; // swap2(ix_[x1], ix_[x2]);
    }

    void swap_positions(ulong i1, ulong i2)
    {
        ...
    }
}
### 10.11 Star-transposition order

The permutations can be ordered so that successive permutations differ by a swap of the element at the first position with some other element (star transposition), figure 10.11-A. An algorithm for the generation of such an ordering is given in [174]. The implementation of the algorithm, ascribed to Gideon Ehrlich, is given in [FXT: class perm-star in comb/perm-star.h]. The generation of the inverse permutations is an option that is activated by the \#define \texttt{PERM\_STAR\_WITH\_INVERSE}. If the successive permutations differ by a transposition

\[
\text{swap2}(a_\[0\], a_\[\text{swp}\]);
\]

then the inverse can be updated as

\[
\text{swap2}(ia_\[a_\[0]\], ia_\[a_\[\text{swp}\]\]);
\]

Note that in the sequence of the inverse permutations the zero is always moved. In the list of the inverse permutations the reversed permutations of the first half are in the second half.

The listing shown in figure 10.11-A can be obtained with [FXT: comb/perm-star-demo.cc]. About 77 million permutations per second are generated and about 115 million when the inverse permutation is not computed. If the only the swaps are of interest then use [FXT: class perm-star-swaps in comb/perm-star-swaps.h] whose update routine works at a rate about 158 million per second [FXT: comb/perm-star-swaps-demo.cc].
## Chapter 10: Permutations

### Permutation, Swap, Inverse

<table>
<thead>
<tr>
<th></th>
<th>permutation</th>
<th>swap</th>
<th>inverse p.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>[. 1 2 3]</td>
<td></td>
<td>[. 1 2 3]</td>
</tr>
<tr>
<td>1:</td>
<td>[ 1 . 2 3]</td>
<td>(0, 1)</td>
<td>[ 1 . 2 3]</td>
</tr>
<tr>
<td>2:</td>
<td>[ 2 . 1 3]</td>
<td>(0, 2)</td>
<td>[ 1 2 . 3]</td>
</tr>
<tr>
<td>3:</td>
<td>[ 2 1 . 3]</td>
<td>(0, 1)</td>
<td>[ . 2 1 3]</td>
</tr>
<tr>
<td>4:</td>
<td>[ 1 2 . 3]</td>
<td>(0, 2)</td>
<td>[ 2 . 1 3]</td>
</tr>
<tr>
<td>5:</td>
<td>[ 2 1 . 3]</td>
<td>(0, 1)</td>
<td>[ 2 1 . 3]</td>
</tr>
<tr>
<td>6:</td>
<td>[ 3 1 . 2]</td>
<td>(0, 3)</td>
<td>[ 2 1 3 .]</td>
</tr>
<tr>
<td>7:</td>
<td>[ . 1 3 2]</td>
<td>(0, 2)</td>
<td>[ . 1 3 2]</td>
</tr>
<tr>
<td>8:</td>
<td>[ 1 1 . 3 2]</td>
<td>(0, 1)</td>
<td>[ 1 . 3 2]</td>
</tr>
<tr>
<td>9:</td>
<td>[ . 3 1 2]</td>
<td>(0, 2)</td>
<td>[ . 1 3 2]</td>
</tr>
<tr>
<td>10:</td>
<td>[ 3 1 . 2]</td>
<td>(0, 1)</td>
<td>[ 3 1 . 2]</td>
</tr>
<tr>
<td>11:</td>
<td>[ 1 3 . 2]</td>
<td>(0, 2)</td>
<td>[ . 2 3 1]</td>
</tr>
<tr>
<td>12:</td>
<td>[ 3 1 . 2]</td>
<td>(0, 3)</td>
<td>[ 2 3 . 1]</td>
</tr>
<tr>
<td>13:</td>
<td>[ 3 2 . 1]</td>
<td>(0, 1)</td>
<td>[ 3 2 1 .]</td>
</tr>
<tr>
<td>14:</td>
<td>[ . 2 3 1]</td>
<td>(0, 2)</td>
<td>[ . 3 1 2]</td>
</tr>
<tr>
<td>15:</td>
<td>[ 2 3 . 1]</td>
<td>(0, 1)</td>
<td>[ 1 3 . 2]</td>
</tr>
<tr>
<td>16:</td>
<td>[ 3 . 2 1]</td>
<td>(0, 2)</td>
<td>[ 1 3 2 .]</td>
</tr>
<tr>
<td>17:</td>
<td>[ . 3 2 1]</td>
<td>(0, 1)</td>
<td>[ . 3 2 1]</td>
</tr>
<tr>
<td>18:</td>
<td>[ 1 3 2 .]</td>
<td>(0, 3)</td>
<td>[ 3 . 2 1]</td>
</tr>
<tr>
<td>19:</td>
<td>[ 3 2 1 .]</td>
<td>(0, 2)</td>
<td>[ 2 3 . 1]</td>
</tr>
<tr>
<td>20:</td>
<td>[ 3 2 1 .]</td>
<td>(0, 1)</td>
<td>[ 3 2 1 .]</td>
</tr>
<tr>
<td>21:</td>
<td>[ 1 2 3 .]</td>
<td>(0, 2)</td>
<td>[ 3 . 1 2]</td>
</tr>
<tr>
<td>22:</td>
<td>[ 2 1 3 .]</td>
<td>(0, 1)</td>
<td>[ 3 1 . 2]</td>
</tr>
<tr>
<td>23:</td>
<td>[ 3 1 2 .]</td>
<td>(0, 2)</td>
<td>[ 3 1 2 .]</td>
</tr>
</tbody>
</table>

**Figure 10.11-A:** The permutations of 4 elements in star-transposition order. Dots denote zeros.

<table>
<thead>
<tr>
<th></th>
<th>permutation</th>
<th>inverse perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>[. 1 2 3]</td>
<td>[. 1 2 3]</td>
</tr>
<tr>
<td>1:</td>
<td>[ 3 . 1 2]</td>
<td>[ 1 2 3 .]</td>
</tr>
<tr>
<td>2:</td>
<td>[ 1 . 2 3]</td>
<td>[ 3 1 2 .]</td>
</tr>
<tr>
<td>3:</td>
<td>[ 2 3 . 1]</td>
<td>[ 2 3 1 .]</td>
</tr>
<tr>
<td>4:</td>
<td>[ 1 . 2 3]</td>
<td>[ 1 . 2 3]</td>
</tr>
<tr>
<td>5:</td>
<td>[ 3 1 . 2]</td>
<td>[ 2 1 3 .]</td>
</tr>
<tr>
<td>6:</td>
<td>[ 2 3 1 .]</td>
<td>[ 3 2 1 .]</td>
</tr>
<tr>
<td>7:</td>
<td>[ 1 . 2 3]</td>
<td>[ 2 1 3 .]</td>
</tr>
<tr>
<td>8:</td>
<td>[ . 1 2 3]</td>
<td>[ 2 1 3 .]</td>
</tr>
<tr>
<td>9:</td>
<td>[ 2 3 1 .]</td>
<td>[ 3 2 1 .]</td>
</tr>
<tr>
<td>10:</td>
<td>[ . 2 3 1]</td>
<td>[ 3 1 2 .]</td>
</tr>
<tr>
<td>11:</td>
<td>[ 2 . 3 1]</td>
<td>[ 1 3 . 2]</td>
</tr>
<tr>
<td>12:</td>
<td>[ 3 . 2 1]</td>
<td>[ 2 3 1 .]</td>
</tr>
<tr>
<td>13:</td>
<td>[ 2 . 3 1]</td>
<td>[ 1 3 2 .]</td>
</tr>
<tr>
<td>14:</td>
<td>[ . 2 3 1]</td>
<td>[ 2 3 1 .]</td>
</tr>
<tr>
<td>15:</td>
<td>[ 3 2 1 .]</td>
<td>[ 3 2 1 .]</td>
</tr>
<tr>
<td>16:</td>
<td>[ 2 . 1 3]</td>
<td>[ 1 2 3 .]</td>
</tr>
<tr>
<td>17:</td>
<td>[ 3 . 2 1]</td>
<td>[ 2 3 1 .]</td>
</tr>
<tr>
<td>18:</td>
<td>[ 2 . 1 3]</td>
<td>[ 1 2 3 .]</td>
</tr>
<tr>
<td>19:</td>
<td>[ 3 . 2 1]</td>
<td>[ 1 3 2 .]</td>
</tr>
<tr>
<td>20:</td>
<td>[ 2 1 3 .]</td>
<td>[ 2 1 3 .]</td>
</tr>
<tr>
<td>21:</td>
<td>[ 3 . 2 1]</td>
<td>[ 1 3 2 .]</td>
</tr>
<tr>
<td>22:</td>
<td>[ 2 1 3 .]</td>
<td>[ 3 1 . 2]</td>
</tr>
<tr>
<td>23:</td>
<td>[ 1 3 . 2]</td>
<td>[ 2 . 3 1]</td>
</tr>
</tbody>
</table>

**Figure 10.12-A:** The permutations of 4 elements in derangement order.
10.12 Derangement order

The derangement order for permutations is characterized by the fact that two successive permutations have no element at the same position, as shown in figure 10.12-A. The listing was created with the program [FXT: comb/perm-derange-demo.cc]. There is no derangement order for \( n = 3 \). An implementation of the underlying algorithm (given in [235, p.611]) is [FXT: class perm_derange in comb/perm-derange.h]:

```cpp
class perm_derange
{
public:
    ulong n_; // number of elements
    ulong *x_; // current permutation
    ulong ctm_; // counter modulo n
    perm_trotter* T_;

public:
    perm_derange(ulong n)
    // Must have: n>=4
    // n=2: trivial, n=3: no solution exists, n>=4: ok
    {
        n_ = n;
        x_ = new ulong[n_];
        T_ = new perm_trotter(n_-1);
        first();
    }

    bool next()
    {
        ++ctm_; // every n steps: need next perm_trotter
        if ( ctm_==n_ ) // current permutation is last
            return false;
        if ( ! T_->next() ) return false;
        for (ulong k=0; k<n_-1; ++k) x_[k] = t[k];
        x_[n_-1] = n_-1; // last element
        else // rotate
            if ( ctm_==n_-1 ) rotate_left1(x_, n_);
            else // last two elements swapped
                rotate_right1(x_, n_);
        return true;
    }
};
```

The routines `rotate_right1()` and `rotate_last()` rotate the array `x_[]` by one position [FXT: perm/rotate.h]. These rotations are the performance bottleneck, the cost of one update of a length-\( n \) permutation is proportional to \( n \). Still, about 35 million permutations per second are generated for \( n = 12 \).

Gray codes have the minimal number of changes between successive permutations while derangement orders have the maximum. An algorithm for generating all permutations of \( n \) objects with \( k \) transitions (where \( 2 \leq k \leq n \) and \( k \neq 3 \)) is given in [234].

Derangement order for even \( n \)

An algorithm for the generation of permutations via cyclic shifts suggested in [181] generates a derangement order if the number \( n \) of elements is even, see figure 10.12-B. An implementation of the algorithm, following [174], is [FXT: class perm_rot in comb/perm-rot.h]. For odd \( n \) the number of times that
## Chapter 10: Permutations

### Figure 10.12-B: Permutations generated via cyclic shifts. The order is a derangement order for even $n$ (left), but not for odd $n$ (right). Dots denote zeros.

<table>
<thead>
<tr>
<th>permutation</th>
<th>inv. perm.</th>
<th>permutation</th>
<th>inv. perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>0: [ . 1 2 ]</td>
<td>[ . 1 2 ]</td>
</tr>
<tr>
<td>1: [ 2 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>1: [ 1 2 ]</td>
<td>[ 2 1 ]</td>
</tr>
<tr>
<td>2: [ 2 3 ]</td>
<td>[ 2 3 1 ]</td>
<td>2: [ 2 1 ]</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>3: [ 1 2 ]</td>
<td>[ 1 2 3 ]</td>
<td>3: [ 1 2 ]</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>4: [ 1 2 ]</td>
<td>[ 2 3 1 ]</td>
<td>4: [ 2 1 ]</td>
<td>[ 2 1 ]</td>
</tr>
<tr>
<td>5: [ 2 3 ]</td>
<td>[ 3 2 1 ]</td>
<td>5: [ 2 1 ]</td>
<td>[ 2 1 ]</td>
</tr>
<tr>
<td>6: [ 3 2 ]</td>
<td>[ 3 1 2 ]</td>
<td>6: [ . 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>7: [ . 3 1 2 ]</td>
<td>[ . 3 1 2 ]</td>
<td>7: [ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>8: [ 2 1 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>8: [ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
</tr>
<tr>
<td>9: [ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
<td>9: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
</tr>
<tr>
<td>10: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
<td>10: [ . 3 2 1 ]</td>
<td>[ . 3 2 1 ]</td>
</tr>
<tr>
<td>11: [ 2 3 1 ]</td>
<td>[ 2 3 1 ]</td>
<td>11: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
</tr>
<tr>
<td>12: [ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
<td>12: [ . 3 1 2 ]</td>
<td>[ . 3 1 2 ]</td>
</tr>
<tr>
<td>13: [ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>13: [ 2 3 1 ]</td>
<td>[ 2 3 1 ]</td>
</tr>
<tr>
<td>14: [ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
<td>14: [ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>15: [ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>15: [ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>16: [ . 3 2 1 ]</td>
<td>[ . 3 2 1 ]</td>
<td>16: [ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>17: [ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
<td>17: [ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
</tr>
<tr>
<td>18: [ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>18: [ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>19: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
<td>19: [ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>20: [ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
<td>20: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
</tr>
<tr>
<td>21: [ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
<td>21: [ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>22: [ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>22: [ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>23: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
<td>23: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
</tr>
</tbody>
</table>

### Figure 10.12-C: Alternative ordering for permutations generated via cyclic shifts. The order is a derangement order for even $n$ (left), but not for odd $n$ (right).

<table>
<thead>
<tr>
<th>ffact</th>
<th>permutation</th>
<th>inv. perm.</th>
<th>ffact</th>
<th>perm.</th>
<th>inv. perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [ . . . ]</td>
<td>[ . 1 2 3 ]</td>
<td>[ . 1 2 3 ]</td>
<td>0: [ . . ]</td>
<td>[ . 1 2 ]</td>
<td>[ . 1 2 ]</td>
</tr>
<tr>
<td>1: [ 1 . . ]</td>
<td>[ 3 1 2 ]</td>
<td>[ 1 2 3 ]</td>
<td>1: [ 1 . ]</td>
<td>[ 2 1 ]</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>2: [ 3 . 1 ]</td>
<td>[ 2 3 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>2: [ 2 . ]</td>
<td>[ 1 2 ]</td>
<td>[ 2 1 ]</td>
</tr>
<tr>
<td>3: [ 1 2 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 3 1 2 ]</td>
<td>3: [ 1 . ]</td>
<td>[ 1 2 ]</td>
<td>[ 1 2 ]</td>
</tr>
<tr>
<td>4: [ . 1 . ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
<td>4: [ . 1 1 ]</td>
<td>[ . 1 2 ]</td>
<td>[ . 1 2 ]</td>
</tr>
<tr>
<td>5: [ 1 1 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 1 3 2 ]</td>
<td>5: [ 1 1 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
</tr>
<tr>
<td>6: [ 2 1 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 2 1 3 ]</td>
<td>6: [ 2 1 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
</tr>
<tr>
<td>7: [ 3 1 ]</td>
<td>[ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
<td>7: [ 3 1 1 ]</td>
<td>[ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>8: [ . 2 1 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>8: [ . 2 1 ]</td>
<td>[ 1 2 3 ]</td>
<td>[ 1 2 3 ]</td>
</tr>
<tr>
<td>9: [ 1 2 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>9: [ 1 2 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>10: [ 2 2 ]</td>
<td>[ 3 1 2 ]</td>
<td>[ 2 1 3 ]</td>
<td>10: [ 2 2 ]</td>
<td>[ 3 1 2 ]</td>
<td>[ 3 1 2 ]</td>
</tr>
<tr>
<td>11: [ 3 1 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>11: [ 3 1 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>12: [ . 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>12: [ . 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>13: [ 2 1 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 1 2 3 ]</td>
<td>13: [ 2 1 ]</td>
<td>[ 2 1 3 ]</td>
<td>[ 2 1 3 ]</td>
</tr>
<tr>
<td>14: [ 3 . 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>14: [ 3 . 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>15: [ 1 3 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>15: [ 1 3 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>16: [ . 1 1 ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
<td>16: [ . 1 1 ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
</tr>
<tr>
<td>17: [ 1 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>17: [ 1 1 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>18: [ . 1 1 ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
<td>18: [ . 1 1 ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
</tr>
<tr>
<td>19: [ 1 1 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>19: [ 1 1 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>20: [ . 1 ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
<td>20: [ . 1 ]</td>
<td>[ . 1 3 2 ]</td>
<td>[ . 1 3 2 ]</td>
</tr>
<tr>
<td>21: [ 1 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>21: [ 1 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>22: [ 2 . 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
<td>22: [ 2 . 1 ]</td>
<td>[ 1 3 2 ]</td>
<td>[ 1 3 2 ]</td>
</tr>
<tr>
<td>23: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
<td>23: [ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
<td>[ 3 2 1 ]</td>
</tr>
</tbody>
</table>
the successor is not a derangement of the predecessor equals \((n + 1)/2\)! − 1. The program [FXT: comb/perm-rot-demo.cc] generates the permutations and counts those transitions.

An alternative ordering with the same number of transitions that are no derangements is obtained via mixed radix counting in falling factorial basis and the routine [FXT: comb/perm-rot-unrank-demo.cc]

```c
1 void ffact2perm_rot(const ulong *fc, ulong n, ulong *x)
2 // Convert falling factorial number fc[0, ..., n-2] into
3 // permutation of x[0, ..., n-1].
4 {
5     for (ulong k=0; k<n; ++k) x[k] = k;
6     for (ulong k=n-1, j=2; k!=0; --k, ++j) rotate_right(x+k-1, j, fc[k-1]);
7 }
```

Figure 10.12-C shows the generated ordering for \(n = 4\) and \(n = 3\). The observation that the permutations in second ordering are the complemented reversals of the first leads to the unranking routine

```c
1 class perm_rot
2 {
3     ulong *a_; // permutation of n elements
4     ulong n_;// number of elements
5     [--snip--]
6     void goto_ffact(const ulong *d)
7     // Goto permutation corresponding to d[] (i.e. unrank d[]).
8     // d[] must be a valid (falling) factorial mixed radix string.
9     {
10         for (ulong k=0; k<n_; ++k) a_[k] = k;
11         for (ulong k=n_-1, j=2; k!=0; --k, ++j) rotate_right(a_+k-1, j, d[k-1]);
12         reverse(a_, n_);
13         make_complement(a_, a_, n_);
14     }
15     [--snip--]
16 }
```

Compare to the unranking for permutations by prefix reversals shown in section 10.4.2 on page 249.

### 10.13 Recursive algorithm for cyclic permutations

A simple recursive algorithm for the generation of all permutations of \(n\) elements can be described as follows: Put each of the \(n\) element of the array to the first position and generate all permutations of \(n-1\) elements. If \(n\) equals one, print the permutation.

The order obtained is shown in figure 10.13-A it corresponds to the alternative (swaps-) factorial representation with falling basis, given in section 10.3.4 on page 242.

The algorithm is implemented in [FXT: class perm_rec in comb/perm-rec.h]:

```c
1 class perm_rec
2 {
3     public:
4         ulong *x_; // permutation
5         ulong n_; // number of elements
6         void (*visit_)(const perm_lex_rec &); // function to call with each permutation
7     
8     public:
9         perm_rec(ulong n)
10         {
11             n_ = n;
12             x_ = new ulong[n_];
13         }
14         ~perm_rec()
15         {
16             delete [] x_;
17         }
18         void init()
19         {
20             for (ulong k=0; k<n_; ++k) x_[k] = k;
21         }
22     
23     public:
24         void perm()
25         {
26             for (ulong k=0; k<n_; ++k) permutation(k);
27         }
28     
29     public:
30         void perm_1(ulong j)
31         {
32             for (ulong k=0; k<n_; ++k) x_[k] = k;
33             for (ulong k=n_-1; k!=j; --k) x_[k] = x_[k+1];
34             x_[j] = x_[j+1];
35             x_[j+1] = x_[j];
36         }
37         void perm_2(ulong j)
38         {
39             for (ulong k=0; k<n_; ++k) x_[k] = k;
40             for (ulong k=n_-1; k!=j; --k) x_[k] = x_[k+1];
41             x_[j] = x_[j+1];
42             x_[j+1] = x_[j];
43         }
44         void perm_3(ulong j)
45         {
46             for (ulong k=0; k<n_; ++k) x_[k] = k;
47             for (ulong k=n_-1; k!=j; --k) x_[k] = x_[k+1];
48             x_[j] = x_[j+1];
49             x_[j+1] = x_[j];
50         }
51         void perm_4(ulong j)
52         {
53             for (ulong k=0; k<n_; ++k) x_[k] = k;
54             for (ulong k=n_-1; k!=j; --k) x_[k] = x_[k+1];
55             x_[j] = x_[j+1];
56             x_[j+1] = x_[j];
57         }
58         void perm_5(ulong j)
59         {
60             for (ulong k=0; k<n_; ++k) x_[k] = k;
61             for (ulong k=n_-1; k!=j; --k) x_[k] = x_[k+1];
62             x_[j] = x_[j+1];
63             x_[j+1] = x_[j];
64         }
65     
66     public:
67         void perm_lex_rec()
68         {
69             // ...
The recursive function next_rec() is

```c
    void next_rec(ulong d)
    {
        if ( d==n_-1 ) visit_(*this);
        else
        {
            const ulong pd = x_[d];
            for (ulong k=d; k<n_; ++k)
            {
                ulong px = x_[k];
                x_[k] = pd; x_[d] = px; // ^= swap2(x_[d], x_[k]);
                next_rec(d+1);
                x_[k] = px; x_[d] = pd; // ^= swap2(x_[d], x_[k]);
            }
        }
    }
```

The algorithm works because at each recursive call the elements \(x[d], \ldots, x[n-1]\) are in a different order and when the function returns the elements are in the same order as they were initially. With the 'for'-statement changed to

```c
    for (ulong x=n_-1; (long)x>=(long)d; --x)
```

the permutations would appear in reversed order. Changing the loop in the function next_rec() to

```c
    for (ulong k=d; k<n_; ++k)
    {
        swap2(x_[d], x_[k]);
        next_rec(d+1, qq);
    }
    rotate_left1(x_+d, n_-d);
```

produces lexicographic order.
A modified function generates the cyclic permutations (permutations consisting of exactly one cycle of full length, see section 2.10.3 on page 112). The only change is to skip the case $x = d$ in the loop:

```c
for (ulong k=d+1; k<n_; ++k) // omit k==d
```

The cyclic permutations of five elements are shown in figure 10.13-B. The program [FXT: comb/perm-rec-demo.cc] was used to create the figures in this section.

```c
void visit(const perm_rec &P) // function to call with each permutation
{
  // Print the permutation
}
```

```c
int main(int argc, char **argv)
{
  ulong n = 5; // Number of elements to permute
  bool cq = 1; // Whether to generate only cyclic permutations
  perm_rec P(n);
  if ( cq ) P.generate_cyclic(visit);
  else P.generate(visit);
  return 0;
}
```

The routines generate about 57 million permutations and about 37 million cyclic permutations per second.

### 10.14 Minimal-change order for cyclic permutations

All cyclic permutations can be generated from a mixed radix Gray code with falling factorial base (see section 9.2 on page 222). Two successive permutations differ at three positions as shown in figure 10.14-A. An constant amortized time (CAT) implementation is [FXT: class cyclic_perm in comb/cyclic-perm.h]:

```c
1  class cyclic_perm
2  {
3    public:
4      mixedradix_gray *M_;
```
Chapter 10: Permutations

- permutation fact.num. cycle
  
  0: [ 4 0 1 2 3 ] [ . . ] ( 4, 3, 2, 1, 0, )
  1: [ 3 4 1 2 0 ] [ 1 . ] ( 4, 0, 3, 2, 1, )
  2: [ 3 0 4 2 1 ] [ 2 . ] ( 4, 1, 0, 3, 2, )
  3: [ 3 0 1 4 2 ] [ 3 . ] ( 4, 2, 1, 0, 3, )
  4: [ 2 3 1 4 0 ] [ 3 1 . ] ( 4, 0, 2, 1, 3, )
  5: [ 2 3 4 0 1 ] [ 2 1 . ] ( 4, 1, 3, 0, 2, )
  6: [ 2 4 1 0 3 ] [ 1 1 . ] ( 4, 3, 0, 2, 1, )
  7: [ 4 3 1 0 2 ] [ . 1 . ] ( 4, 2, 1, 3, 0, )
  8: [ 4 0 3 1 2 ] [ . 2 . ] ( 4, 2, 3, 1, 0, )
  9: [ 2 4 3 0 1 ] [ 1 2 . ] ( 4, 0, 2, 3, 1, )
 10: [ 2 0 4 1 3 ] [ 2 2 . ] ( 4, 3, 1, 0, 2, )
 11: [ 2 0 3 4 1 ] [ 3 2 . ] ( 4, 1, 0, 2, 3, )
 12: [ 1 2 3 4 0 ] [ 3 2 1 ] ( 4, 0, 1, 2, 3, )
 13: [ 1 2 4 0 3 ] [ 2 2 1 ] ( 4, 3, 0, 1, 2, )
 14: [ 1 4 3 0 2 ] [ 1 2 1 ] ( 4, 2, 3, 0, 1, )
 15: [ 4 2 3 0 1 ] [ 1 2 1 ] ( 4, 1, 2, 3, 0, )
 16: [ 4 3 0 2 1 ] [ 1 1 . ] ( 4, 1, 3, 2, 0, )
 17: [ 1 4 0 2 3 ] [ 1 1 1 ] ( 4, 3, 2, 0, 1, )
 18: [ 1 3 4 2 0 ] [ 2 1 1 ] ( 4, 0, 1, 3, 2, )
 19: [ 1 3 0 4 2 ] [ 3 1 1 ] ( 4, 2, 0, 1, 3, )
 20: [ 3 2 0 4 1 ] [ 3 . 1 ] ( 4, 1, 2, 0, 3, )
 21: [ 3 2 4 1 0 ] [ 2 . 1 ] ( 4, 0, 3, 1, 2, )
 22: [ 3 4 0 1 2 ] [ 1 1 . ] ( 4, 2, 0, 3, 1, )
 23: [ 4 2 0 1 3 ] [ . 1 . ] ( 4, 3, 1, 2, 0, )

Figure 10.14-A: All cyclic permutations of 5 elements in a minimal-change order.

```
ulong n_; // number of elements to permute
ulong *ix_; // current permutation (of {0, 1, ..., n-1})
ulong *x_; // inverse permutation

public:
cyclic_perm(ulong n)
: n_(n)
{
    ix_ = new ulong[n_];
    x_ = new ulong[n_];
    M_ = new mixedradix_gray(n_-2, 0); // falling factorial base
    first();
}

--snip--

private:
void setup()
{
    const ulong *fc = M_->data();
    for (ulong k=0; k<n_; ++k) ix_[k] = k;
    for (ulong k=n_-1; k>1; --k)
    {
        ulong z = n_-3-(k-2); // 0, ..., n-3
        ulong i = fc[z];
        swap2(ix_[k], ix_[i]);
    }
    if ( n_>1 ) swap2(ix_[0], ix_[1]);
    make_inverse(ix_, x_, n_);
}

public:
void first()
{
    M_->first();
    setup();
}

bool next()
{
```
The order so that the permutation is the same as if one would compute it via the function [FXT: ffact2cyclic()] in comb/fact2cyclic.cc which is given in section 10.3.5 on page 245. The listing in figure 10.14-A was created with the program [FXT: comb/cyclic-perm-demo.cc]. About 40 million permutations per second are generated.

10.15 Permutations with special properties

10.15.1 The number of certain permutations

We discuss permutations with special properties, such as involutions, derangements, and permutations with prescribed cycle types.

10.15.1.1 Permutations with \( m \) cycles: Stirling cycle numbers

\[
\begin{array}{cccccccccc}
 n & \text{total} & m=1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
 1 & 1 & & & & & & & & \\
 2 & 1 & 1 & & & & & & & \\
 3 & 6 & 2 & 3 & & & & & & \\
 4 & 24 & 6 & 11 & 6 & & & & & \\
 5 & 120 & 24 & 50 & 35 & 10 & & & & \\
 6 & 720 & 120 & 274 & 225 & 85 & 15 & & & \\
 7 & 5040 & 720 & 1764 & 1624 & 735 & 175 & 21 & & \\
 8 & 40320 & 5040 & 13068 & 13132 & 6769 & 1960 & 322 & 28 & \\
 9 & 362880 & 40320 & 109584 & 118124 & 67284 & 22449 & 4536 & 546 & 36 \\
\end{array}
\]

Figure 10.15-A: Stirling numbers of the first kind \( s(n, m) \) (Stirling cycle numbers).

The number of permutations of \( n \) elements into \( m \) cycles is given by the (unsigned) Stirling numbers of the first kind (or Stirling cycle numbers) \( s(n, m) \). The first few are shown in figure 10.15-A which was created with the program [FXT: comb/stirling1-demo.cc]. One has \( s(1, 1) = 1 \) and

\[
s(n, m) = s(n-1, m-1) + (n-1) s(n-1, m) \tag{10.15-1}
\]

\[
\sum_{m=0}^{n} s(n, m) e^m = \prod_{m=0}^{n} e + m = e^n \tag{10.15-2}
\]

A generating function is given as relation [35.2-75a] on page 707, see also entry [A008275] of [245]. The Stirling numbers of the second kind (Stirling set numbers) are treated in section 15.2 on page 345. Many
identities involving the Stirling numbers are given in [134, pp.243-253]. We note just a few, writing \( S(n, k) \) for the Stirling set numbers:

\[
x^n = \sum_{k=0}^{n} S(n, k) x^k = \sum_{k=0}^{n} S(n, k) (-1)^{n-k} x^k \] (10.15-3a)

where \( x^k = (x-1)(x-2) \cdots (x-k+1) \) and \( x^k = x(x+1)(x+2) \cdots (x+k-1) \).

\[
x^k = \sum_{k=0}^{n} s(n, k) (-1)^{n-k} x^k \] (10.15-3b)

\[
x^k = \sum_{k=0}^{n} s(n, k) x^k \] (10.15-3c)

Further [134, p.296], with \( D := \frac{d}{dz} \) and \( \vartheta = z \frac{d}{dz} \), we have the operator identities

\[
\vartheta^n = \sum_{k=0}^{n} S(n, k) z^k D^k \] (10.15-4a)

\[
z^n D^n = \sum_{k=0}^{n} s(n, k) (-1)^{n-k} \vartheta^k \] (10.15-4b)

### 10.15.1.2 Permutations with prescribed cycle type

A permutation of \( n \) elements is of type \( C = [c_1, c_2, c_3, \ldots, c_n] \) if it has \( c_1 \) fixed points, \( c_2 \) cycles of length 2, \( c_3 \) cycles of length 3, and so on. The number \( Z_{n,C} \) of permutations of \( n \) elements with type \( C \) equals [53, p.80]

\[
Z_{n,C} = \frac{n!}{c_1! c_2! c_3! \cdots c_n!} \left( 1^{c_1} 2^{c_2} 3^{c_3} \cdots n^{c_n} \right) \] (10.15-5)

We necessarily have \( n = c_1 + 2c_2 + \ldots + nc_n \), that is, the \( c_j \) correspond to a integer partition of \( n \).

The exponential generating function \( \exp(L(z)) \) where

\[
L(z) = \sum_{k=1}^{\infty} \frac{c_k z^k}{k} \] (10.15-6)

gives detailed information about the all cycle types:

\[
\exp(L(z)) = \sum_{n=0}^{\infty} \left( \sum_{C} \left( Z_{n,C} \prod t_k^{c_k} \right) \right) \frac{z^n}{n!} \] (10.15-7)

That is, the exponent of \( t_k \) indicates how many cycles of length \( k \) are present in the given cycle type.

See section 15.2.2 on page 346 for a similar construction for the EGF for set partitions of given type.
10.15: Permutations with special properties

10.15.1.3 Prefix conditions

Some types of permutations can be generated efficiently by a routine that produces the lexicographically ordered list of permutations subject to conditions for all prefixes. The implementation (following [174]) is [FXT: class perm_restrpref in comb/perm-restrpref.h]. The condition (as a function pointer) has to be supplied upon creation of an instance of the class. The program [FXT: comb/perm-restrpref-deno.cc] demonstrates the usage, it can be used to generate all involutions, up-down, indecomposable, or derangement permutations, see figure 10.15-B.

10.15.1.4 Involutions

The sequence $I(n)$ of the number of involutions (self-inverse permutations) starts as ($n \geq 1$)

$$I(1) = 1, \quad I(2) = 2, \quad I(3) = 4, \quad I(4) = 10, \quad I(5) = 26, \quad I(6) = 76, \quad I(7) = 232, \quad I(8) = 764, \quad I(9) = 2620, \quad I(10) = 9496, \quad I(11) = 35696, \quad I(12) = 140152, \quad I(13) = 568504, \ldots$$

This is sequence A000085 of [245]. Compute $I(n)$ using the relation

$$I(n) = I(n - 1) + (n - 1) I(n - 2) \quad (10.15-8)$$

Let $h_n(x)$ be the polynomial such that the coefficient of $x^k$ gives the number of involutions of $n$ elements with $k$ fixed points. The polynomials can be computed recursively via $h_{n+1} = h_n + x h_n$ (starting with $h_0 = 1$). We have $h_n(1) = I(n)$:

```python
N=20; v=vector(N);
v[1]=1; v[2]=2;
for(n=3,N,v[n]=v[n-1]+(n-1)*v[n-2]); v
[1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152, 568504, 2390480, ...]
```

The exponential generating function (EGF) is

$$\sum_{k=0}^{\infty} \frac{I(k) x^k}{k!} = \exp \left( x + \frac{x^2}{2} \right) \quad (10.15-9)$$

We further have

$$\sum_{k=0}^{\infty} \frac{h_k(t) x^k}{k!} = \exp \left( t x + \frac{t x^2}{2} \right) \quad (10.15-10)$$

![Figure 10.15-B: Examples of permutations subject to conditions on the prefixes. From left to right: involutions, up-down permutations, indecomposable permutations and derangements.](image)
The EGF for the number permutations whose $m$-th power is identity is given in [281, p.85]:

$$\exp \left( \sum_{d|m} \frac{x^d}{d} \right)$$

The special case $m = 2$ gives relation [10.15-9]

The condition function for involutions is

```c
bool cond_inv(const ulong *a, ulong k)
{
    ulong ak = a[k];
    if ( (ak<=k) && (a[ak]!=k) ) return false;
    return true;
}
```

10.15.1.5 Alternating permutations

The alternating permutations (or up-down permutations) satisfy $a_0 < a_1 > a_2 < a_3 > \ldots$. The condition function is

```c
bool cond_updown(const ulong *a, ulong k)
// up-down condition: a1 < a2 > a3 < a4 > 
{
    if ( k<2 ) return true;
    if ( (k%2) ) return ( a[k]<a[k-1] );
    else return ( a[k]>a[k-1] );
}
```

Note that the routine is for the permutations of the elements 1,2,...,n in a one-based array.

The sequence $A(n)$ of the number of alternating permutations starts as $(n \geq 1)$

$$1, 1, 2, 5, 16, 61, 272, 1385, 7936, 50521, 353792, 2702765, 22368256, \ldots$$

It is sequence A000111 of [245], the sequence of the Euler numbers. The list can be computed using the relation

$$A(n) = \frac{1}{2} \sum_{k=0}^{n-1} \binom{n-1}{k} A(k) A(n-1-k)$$

An exponential generating function is

$$\frac{1 + \sin(z)}{\cos(z)} = \sum_{k=0}^{\infty} \frac{A(k)}{k!} z^k$$

for

$$1 + z + z^2 + 2z^3 + 5z^4 + 16z^5 + 61z^6 + 272z^7 + 1385z^8 + 7936z^9 + \ldots$$

10.15.1.6 Indecomposable permutations

The indecomposable (or connected) permutations satisfy, for $k = 0, 1, \ldots, n-2$, the inequality of sets

$$\{a_0, a_1, \ldots, a_k\} \neq \{0, 1, \ldots, k\}$$

That is, there is no prefix of length $< n$ that is a permutation of itself. The condition function is
ulong N; // set to n in main()
bool cond_indecomp(const ulong *a, ulong k)
  // indecomposable condition: \{a_1, \ldots, a_k\} \neq \{1, \ldots, k\} for all k<n
  {
    if ( k==N ) return true;
    for (ulong i=1; i<=k; ++i) if ( a[i]>k ) return true;
    return false;
  }

The sequence $C(n)$ of the number of indecomposable permutations starts as ($n \geq 1$)
1, 1, 3, 13, 71, 461, 3447, 29093, 273343, 2829325, 31998903, 392743957, ...
This is sequence A003319 of [245]. Compute $C(n)$ using
\[ C(n) = n! - \sum_{k=1}^{n-1} k! C(n-k) \quad (10.15-15) \]

N=20; v=vector(N);
for(n=1,N,v[n]=n!-sum(k=1,n-1,k!*v[n-k])); v
[1, 1, 3, 13, 71, 461, 3447, ...]

The ordinary generating function can be given as
\[ \sum_{n=1}^{\infty} C(n) z^n = 1 - \frac{1}{\sum_{k=0}^{\infty} k! z^k} = z + z^2 + 3 z^3 + 13 z^4 + 71 z^5 + \ldots \quad (10.15-16) \]
The logarithmic generating function is
\[ \sum_{n=1}^{\infty} \frac{C(n)}{n} z^n = z \left[ 1 + \log \left( \sum_{k=0}^{\infty} k! z^k \right) \right] = z + z^2 + \frac{3}{2} z^3 + \frac{13}{3} z^4 + \frac{71}{4} z^5 + \ldots \quad (10.15-17) \]

10.15.1.7 Derangements
A permutation is a derangement if $a_k \neq k$ for all $k$:
bool cond_derange(const ulong *a, ulong k)
  // derangement condition: f[k]!=k for all k
  {
    return ( a[k]!=k );
  }

The sequence $D(n)$ of the number of derangements starts as ($n \geq 1$)
0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961, 14684570, 176214841, ...
This is sequence A000166 of [245], the subfactorial numbers. Compute $D(n)$ using either of
\[ D(n) = (n-1) \left[ D(n-1) + D(n-2) \right] \quad (10.15-18a) \]
\[ = n D(n-1) + (-1)^n \quad (10.15-18b) \]
\[ = \sum_{k=0}^{n} (-1)^{n-k} \frac{n!}{(n-k)!} \quad (10.15-18c) \]
\[ = \left\lfloor \frac{(n! + 1/2)}{e} \right\rfloor \quad (10.15-18d) \]
where $e = \exp(1)$. We use the recursion 10.15-18a.

N=20; v=vector(N); v[1]=0; v[2]=1;
for(n=3,N,v[n]=(n-1)*(v[n-1]+v[n-2])); v
[0, 1, 2, 9, 44, 265, 1854, 14833, ...]

The exponential generating function can be obtained by setting $t_1 = 0$ and $t_k = 1$ for $k \neq 1$ in relation 10.15-6 we have $L(z) = \log \left( 1/(1-z) \right) - z$ and
\[ \sum_{k=0}^{\infty} \frac{D(n) z^n}{n!} = \exp L(z) = \frac{\exp(-z)}{1-z} \quad (10.15-19) \]
10.15.2 Permutations with distance restrictions

We present constructions for Gray codes for permutations with certain restrictions. These are computed from Gray codes of mixed radix numbers with factorial basis. We write \( p(k) \) for the position of the element \( k \) in a given permutation.

10.15.2.1 Permutations where \( p(k) \leq k + 1 \)

<table>
<thead>
<tr>
<th>( k )</th>
<th>( \text{ffact perm} )</th>
<th>( \text{inv. perm} )</th>
<th>( \text{ffact(inv)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>. 3 ..</td>
<td>[ 0 4 1 2 3 ]</td>
<td>[ 0 2 3 4 1 ]</td>
</tr>
<tr>
<td>2</td>
<td>. 2 . .</td>
<td>[ 0 3 1 2 4 ]</td>
<td>[ 0 2 3 1 4 ]</td>
</tr>
<tr>
<td>3</td>
<td>. 1 . .</td>
<td>[ 0 2 1 3 4 ]</td>
<td>[ 0 2 1 3 4 ]</td>
</tr>
<tr>
<td>4</td>
<td>. 1 . 1</td>
<td>[ 0 2 1 4 3 ]</td>
<td>[ 0 2 1 4 3 ]</td>
</tr>
<tr>
<td>5</td>
<td>. . . 1</td>
<td>[ 0 1 2 4 3 ]</td>
<td>[ 0 1 2 4 3 ]</td>
</tr>
<tr>
<td>6</td>
<td>. . . .</td>
<td>[ 0 1 2 3 4 ]</td>
<td>[ 0 1 2 3 4 ]</td>
</tr>
<tr>
<td>7</td>
<td>. . 1 .</td>
<td>[ 0 1 3 2 4 ]</td>
<td>[ 0 1 3 2 4 ]</td>
</tr>
<tr>
<td>8</td>
<td>. . 2 .</td>
<td>[ 0 1 4 2 3 ]</td>
<td>[ 0 1 3 4 2 ]</td>
</tr>
<tr>
<td>9</td>
<td>1 . 2 .</td>
<td>[ 1 0 4 2 3 ]</td>
<td>[ 1 0 3 4 2 ]</td>
</tr>
<tr>
<td>10</td>
<td>1 . 1 .</td>
<td>[ 1 0 3 2 4 ]</td>
<td>[ 1 0 3 2 4 ]</td>
</tr>
<tr>
<td>11</td>
<td>1 . . 1</td>
<td>[ 1 0 2 4 3 ]</td>
<td>[ 1 0 2 4 3 ]</td>
</tr>
<tr>
<td>12</td>
<td>1 . . .</td>
<td>[ 1 0 2 3 4 ]</td>
<td>[ 1 0 2 3 4 ]</td>
</tr>
<tr>
<td>13</td>
<td>2 . . 1</td>
<td>[ 2 0 1 4 3 ]</td>
<td>[ 1 2 0 4 3 ]</td>
</tr>
<tr>
<td>14</td>
<td>2 . . .</td>
<td>[ 2 0 1 3 4 ]</td>
<td>[ 1 2 0 3 4 ]</td>
</tr>
<tr>
<td>15</td>
<td>3 . . .</td>
<td>[ 3 0 1 2 4 ]</td>
<td>[ 1 2 3 0 4 ]</td>
</tr>
<tr>
<td>16</td>
<td>4 . . .</td>
<td>[ 4 0 1 2 3 ]</td>
<td>[ 1 2 3 4 0 ]</td>
</tr>
</tbody>
</table>

Figure 10.15-C: Gray code for the permutations of 5 elements where no element lies more than one place to the right of its position in the identical permutation.

Let \( M(n) \) the number of permutations of \( n \) elements where no element can move more than one place to the right. We have \( M(n) = 2^{n-1} \). A Gray code for these permutation is shown in figure 10.15-C which was created with the program [FXT: comb/perm-right1-gray-demo.cc]. \( M(n) \) also counts the permutations that start as a rising sequence (ending in the maximal element) and end as a falling sequence. The recursion for the array in the leftmost column of figure 10.15-C can be obtained by the recursion

```c
1 void Y_rec(ulong d, bool z)
2 {
3     if ( d==n ) visit();
4     else {
5         if ( z ) // forward:
6             // words 0, 10, 200, 3000, 40000, ...
7             ulong k = 0;
8             do {
9                 ff[d] = k;
10                    Y_rec(d+k+1, !z);
11             } while ( ++k <= (n-d) );
12     } else // backward:
13         // words ..., 40000, 3000, 200, 10, 0
14         ulong k = n-d+1;
15         do {
16             --k;
17             ff[d] = k;
18             Y_rec(d+k+1, !z);
19         } while ( k != 0 );
20     }
```

The array \( \text{ff} \) (of length \( n \)) must be initialized with zeros and the initial call is \( \text{Y_rec}(0, \text{true}) \). About 85 million words per second are generated. In the inverse permutations (where no element is more than
one place left of its original position) the swaps are adjacent and their position is determined by the ruler function. Thereby the inverse permutations can be generated using [FXT: class ruler_func in comb/ruler-func.h] which described in section 8.2.3 on page 206.

10.15.2.2 Permutations where \(k-1 \leq p(k) \leq k+1\)

\[
\begin{array}{ll}
1: & 1 . 1 . . \left[ 1 0 2 4 3 5 6 \right] \\
2: & 1 2 . . . . \left[ 1 0 2 4 3 6 5 \right] \\
3: & 1 . . . . . \left[ 1 0 2 3 4 6 5 \right] \\
4: & 1 . . . . . \left[ 1 0 2 3 4 5 6 \right] \\
5: & 1 . . . . . \left[ 1 0 2 3 5 4 6 \right] \\
6: & 1 1 . . . . \left[ 1 0 2 3 5 6 4 \right] \\
7: & 1 . . . . . \left[ 1 0 2 4 3 5 6 \right] \\
8: & 1 . 1 . . . \left[ 1 0 2 4 3 6 5 \right] \\
9: & . . . . 1 . \left[ 1 0 2 3 4 6 5 \right] \\
10: & . . . . . . \left[ 1 0 2 3 4 5 6 \right] \\
11: & . . . . . . \left[ 1 0 2 3 5 4 6 \right] \\
12: & . . . . . . \left[ 1 0 2 3 5 6 4 \right] \\
13: & . . . . . . \left[ 1 0 2 3 4 5 6 \right]
\end{array}
\]

Figure 10.15-D: Gray code for the permutations of 7 elements where no element lies more than one place away from its position in the identical permutation. The permutations are self-inverse.

Let \(F(n)\) the number of permutations of \(n\) elements where no element can move more than one place to the left. Then \(F(n)\) is the \((n+1)\)-st Fibonacci number. A Gray code for these permutation is shown in figure 10.15-D which was created with the program [FXT: comb/perm-dist1-gray-demo.cc].

10.15.2.3 Permutations where \(k-1 \leq p(k) \leq k+d\)

\[
\begin{array}{ll}
1: & 1 1 . 1 . . \left[ 1 2 0 3 5 4 \right] \\
2: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
3: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
4: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
5: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
6: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
7: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
8: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
9: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
10: & 1 . . . . . \left[ 1 2 0 3 4 5 \right] \\
11: & . . . . . . \left[ 1 2 0 3 4 5 \right] \\
12: & . . . . . . \left[ 1 2 0 3 4 5 \right] \\
13: & . . . . . . \left[ 1 2 0 3 4 5 \right]
\end{array}
\]

Figure 10.15-E: Gray code for the permutations of 6 elements where no element lies more than one place away from the left or two places to the right of its original position. The permutations are self-inverse.

A Gray code for the permutations where no element lies more than one place to the left or \(d\) places to the right of its original position can be obtained via the Gray codes for binary words with at most \(d\) successive ones given in section 12.2 on page 302. Figure 10.15-E shows the permutations of 6 elements.
with $d = 2$, it was created with the program [FXT: comb/perm-l1r2-gray-demo.cc]. The array shown leftmost in figure 10.15-E can be generated via the recursion

```cpp
void Y_rec(ulong d, bool z)
{
    if ( d>=n ) visit();
    else
    {
        const ulong w = n-d;
        if ( z )
        {
            if ( w>1 ) { ff[d]=1; ff[d+1]=1; ff[d+2]=0; Y_rec(d+3, !z); }
            ff[d]=1; ff[d+1]=0; Y_rec(d+2, !z);
            ff[d]=0; Y_rec(d+1, !z);
        }
        else
        {
            ff[d]=0; Y_rec(d+1, !z);
            ff[d]=1; ff[d+1]=0; Y_rec(d+2, !z);
            if ( w>1 ) { ff[d]=1; ff[d+1]=1; ff[d+2]=0; Y_rec(d+3, !z); }
        }
    }
}
```

If the two lines starting `if ( w>1 )` are omitted then the Fibonacci words are obtained. About 100 million words per second are generated.
Chapter 11

Subsets and permutations of a multiset

11.1 Subsets of a multiset

A multiset (or bag) is a collection of elements where elements can be repeated, and order does not matter. A subset of a set of $n$ elements can be identified with the bits of all $n$-bit binary words. The subsets of a multiset can be obtained as mixed radix numbers: if the $j$-th element is repeated $r_j$ times then the radix of digit $j$ has to be $r_j + 1$. Thereby all methods of chapter 9 on page 219 can be applied.

As an example, all divisors of a number $x$ whose factorization $x = p_0^{e_0} \cdot p_1^{e_1} \cdots p_{n-1}^{e_{n-1}}$ is known can be obtained via the length-$n$ mixed radix numbers with radices $[e_0 + 1, e_1 + 1, \ldots, e_{n-1} + 1]$. The implementation [FXT: class divisors in mod/divisors.h] generates the subsets of the exponent-multiset.
in counting order (figure 11.1-A shows the data for \( x = 630 \)). An auxiliary array \( T \) of products is updated with each step: if the changed digit (at position \( j \)) became 1 then set \( t := T_{j+1} \cdot p_j \), else set \( t := T_j \cdot p_j \). Set \( T_i = t \) for all \( 0 \leq i \leq j \). A sentinel element \( T_n = 1 \) avoids unnecessary code. Figure 11.1-A was created with the program [FXT: mod/divisors-demo.cc]. The computation of all products of \( k \) out of \( n \) given factors is described in section 6.1.2 on page 178.

### 11.2 Permutations of a multiset

<table>
<thead>
<tr>
<th>((2, 2, 1))</th>
<th>((6, 2))</th>
<th>((1, 1, 1, 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: [  ]</td>
<td>1: [ 1 1 ]</td>
<td>1: [ 1 1 1 ]</td>
</tr>
<tr>
<td>1: [ 1 1 ]</td>
<td>1: [  ]</td>
<td>2: [ 1 1 1 ]</td>
</tr>
<tr>
<td>1: [ 2 1 1 ]</td>
<td>1: [  ]</td>
<td>3: [  ]</td>
</tr>
<tr>
<td>2: [ 1 1 1 ]</td>
<td>2: [  ]</td>
<td>4: [ 1 1 1 ]</td>
</tr>
<tr>
<td>3: [ 1 1 1 ]</td>
<td>3: [  ]</td>
<td>5: [  ]</td>
</tr>
<tr>
<td>4: [ 2 1 1 ]</td>
<td>4: [  ]</td>
<td>6: [  ]</td>
</tr>
<tr>
<td>5: [ 2 1 1 ]</td>
<td>5: [  ]</td>
<td>7: [  ]</td>
</tr>
<tr>
<td>6: [ 2 1 1 ]</td>
<td>6: [  ]</td>
<td>8: [  ]</td>
</tr>
<tr>
<td>7: [ 2 1 1 ]</td>
<td>7: [  ]</td>
<td>9: [  ]</td>
</tr>
<tr>
<td>8: [ 2 1 1 ]</td>
<td>8: [  ]</td>
<td>10: [  ]</td>
</tr>
<tr>
<td>9: [ 2 1 1 ]</td>
<td>9: [  ]</td>
<td>11: [  ]</td>
</tr>
<tr>
<td>10: [ 2 1 1 ]</td>
<td>10: [  ]</td>
<td>12: [  ]</td>
</tr>
<tr>
<td>11: [ 2 1 1 ]</td>
<td>11: [  ]</td>
<td>13: [ 1 1 1 ]</td>
</tr>
<tr>
<td>12: [ 2 1 1 ]</td>
<td>12: [  ]</td>
<td>14: [ 1 1 1 ]</td>
</tr>
<tr>
<td>13: [ 2 1 1 ]</td>
<td>13: [  ]</td>
<td>15: [ 1 1 1 ]</td>
</tr>
<tr>
<td>14: [ 2 1 1 ]</td>
<td>14: [  ]</td>
<td>16: [ 1 1 1 ]</td>
</tr>
<tr>
<td>15: [ 2 1 1 ]</td>
<td>15: [  ]</td>
<td>17: [ 1 1 1 ]</td>
</tr>
<tr>
<td>16: [ 2 1 1 ]</td>
<td>16: [  ]</td>
<td>18: [ 1 1 1 ]</td>
</tr>
<tr>
<td>17: [ 2 1 1 ]</td>
<td>17: [  ]</td>
<td>19: [ 1 1 1 ]</td>
</tr>
<tr>
<td>18: [ 2 1 1 ]</td>
<td>18: [  ]</td>
<td>20: [ 1 1 1 ]</td>
</tr>
<tr>
<td>19: [ 2 1 1 ]</td>
<td>19: [  ]</td>
<td>21: [ 1 1 1 ]</td>
</tr>
<tr>
<td>20: [ 2 1 1 ]</td>
<td>20: [  ]</td>
<td>22: [ 1 1 1 ]</td>
</tr>
<tr>
<td>21: [ 2 1 1 ]</td>
<td>21: [  ]</td>
<td>23: [ 1 1 1 ]</td>
</tr>
<tr>
<td>22: [ 2 1 1 ]</td>
<td>22: [  ]</td>
<td>24: [ 1 1 1 ]</td>
</tr>
<tr>
<td>23: [ 2 1 1 ]</td>
<td>23: [  ]</td>
<td>25: [ 1 1 1 ]</td>
</tr>
<tr>
<td>24: [ 2 1 1 ]</td>
<td>24: [  ]</td>
<td>26: [ 1 1 1 ]</td>
</tr>
<tr>
<td>25: [ 2 1 1 ]</td>
<td>25: [  ]</td>
<td>27: [ 1 1 1 ]</td>
</tr>
<tr>
<td>26: [ 2 1 1 ]</td>
<td>26: [  ]</td>
<td>28: [ 1 1 1 ]</td>
</tr>
<tr>
<td>27: [ 2 1 1 ]</td>
<td>27: [  ]</td>
<td>29: [ 1 1 1 ]</td>
</tr>
<tr>
<td>28: [ 2 1 1 ]</td>
<td>28: [  ]</td>
<td>30: [ 1 1 1 ]</td>
</tr>
<tr>
<td>29: [ 2 1 1 ]</td>
<td>29: [  ]</td>
<td>30: [ 1 1 1 ]</td>
</tr>
</tbody>
</table>

Figure 11.2-A: Permutations of multisets in lexicographic order: the multiset \((2, 2, 1)\) (left), \((6, 2)\) (combinations \(\binom{9+2}{2}\), middle), and \((1, 1, 1, 1)\) (permutations of four elements, right). Dots denote zeros.

We write \((r_0, r_1, \ldots, r_{k-1})\) for a multiset with \( r_0 \) elements of the first sort, \( r_1 \) of the second sort, \ldots, and \( r_{k-1} \) elements of the \( k \)-th sort. The total number of elements is \( n = \sum_{j=0}^{k-1} r_k \). For the elements of the \( j \)-th sort we always use the number \( j \). The number of permutations \( P(r_0, r_1, \ldots, r_{k-1}) \) of the multiset \((r_0, r_1, \ldots, r_{k-1})\) is a multinomial coefficient:

\[
P(r_0, r_1, \ldots, r_{k-1}) = \frac{n!}{r_0! r_1! r_2! \cdots r_{k-1}!} \quad (11.2-1a)
\]

\[
= \binom{n}{r_0} \binom{n-r_0}{r_1} \binom{n-r_0-r_1}{r_2} \cdots \binom{n-r_0-r_1-r_2-\cdots-r_{k-3}}{r_{k-3}} \binom{r_{k-2}+r_{k-1}}{r_{k-2}} \binom{r_{k-1}}{r_{k-1}} \quad (11.2-1b)
\]

\[
= \binom{r_0}{r_0} \binom{r_0+r_1}{r_1} \binom{r_0+r_1+r_2}{r_2} \binom{r_0+r_1+r_2+r_3}{r_3} \cdots \binom{n}{r_{k-1}} \quad (11.2-1c)
\]

Relation 11.2-1a is obtained by observing that among the \( n! \) ways to arrange all \( n \) elements \( r_0! \) permutations of the first sort of elements, \( r_1! \) of the second, and so on, lead to identical permutations.
11.2: Permutations of a multiset

11.2.1 Recursive generation

Let \([r_0, r_1, r_2, \ldots, r_{k-1}]\) denote the list of all permutations of the multiset \((r_0, r_1, r_2, \ldots, r_{k-1})\). The recursion

\[
[r_0, r_1, r_2, \ldots, r_{k-1}] = \begin{cases} 
    r_0 \cdot [r_0 - 1, r_1, r_2, \ldots, r_{k-1}] & r_0 > 0 \\
    r_1 \cdot [r_0, r_1 - 1, r_2, \ldots, r_{k-1}] & r_1 > 0 \\
    \vdots & \\
    r_{k-1} \cdot [r_0, r_1, r_2, \ldots, r_{k-1} - 1] & r_{k-1} > 0
\end{cases}
\]

(11.2-2)

is used in the following procedure [FXT: comb/mset-perm-lex-rec-demo.cc]:

```c
1 ulong n; // number of objects
2 ulong *ms; // multiset data in ms[0], ..., ms[n-1]
3 ulong k; // number of different sorts of objects
4 ulong *r; // number of elements '0' in r[0], '1' in r[1], ..., 'k-1' in r[k-1]

With the recursion
1 void mset_perm_rec(ulong d)
2 {
3     if ( d>=n ) visit();
4     else {
5         for (ulong j=0; j<k; ++j) // for all buckets
6             ++wct;
7             if ( r[j] ) // bucket has elements left
8                 { // bucket has elements left
9                     ++rct;
10                     --r[j]; // take element from bucket
11                     ms[d] = j; // put element in place
12                     mset_perm_rec(d+1); // recursion
13                     ++r[j]; // put element back
14                 }
15         }
16     }
17 }
18
and the initial call mset_perm_rec(0) we generate all multiset permutations in lexicographic order. As given the routine is inefficient when used with (many) small numbers \(r_j\). An extreme case is \(r_j = 1\) for all \(j\), corresponding to the (regular) permutations: we have \(n = k\) and for the \(n!\) permutations the work is proportional to \(n^n\). The method can be made efficient by maintaining a list of pointers to the next nonzero 'bucket' \(n_k[]\) [FXT: class mset_perm_lex_rec in comb/mset-perm-lex-rec.h]:

```c
1 class mset_perm_lex_rec
2 {
3    public:
4        ulong k_; // number of different sorts of objects
5        ulong *r_; // number of elements '0' in r[0], '1' in r[1], ..., 'k-1' in r[k-1]
6        ulong n_; // number of objects
7        ulong *ms_; // multiset data in ms[0], ..., ms[n-1]
8        ulong *nn_; // position of next nonempty bucket
9        void (*visit_)(const mset_perm_lex_rec &); // function to call with each permutation
10       ulong ct_; // count objects
11       ulong rct_; // count recursions (==work)
12     [--snip--]
13 }
14
The initializer takes as arguments an array of multiplicities and its length:

```c
1 public:
2     mset_perm_lex_rec(ulong *r, ulong k)
3     {
4         k_ = k;
5         r_ = new ulong[k];
6         for (ulong j=0; j<k_; ++j) r_[j] = r[j]; // get buckets
7         n_ = 0;
8         for (ulong j=0; j<k_; ++j) n_ += r_[j];
9         ms_ = new ulong[n_];
10     }
11 }
12
[fxtbook draft of 2008-August-17]
Chapter 11: Subsets and permutations of a multiset

```
11
12    nn_ = new ulong[k_+1]; // incl sentinel
13    for (ulong j=0; j<k_; ++j) nn_[j] = j+1;
14    nn_[k_] = 0; // pointer to first nonempty bucket
15 }
16 {[--snip--]

The method to generate all permutations takes a ‘visit’ function as argument:
```
1
void generate(void (*visit)(const mset_perm_lex_rec &))
2 {
3     visit_ = visit;
4     ct_ = 0;
5     rct_ = 0;
6     mset_perm_rec(0);
7 }
8
private:
9    void mset_perm_rec(ulong d);
10 };

The recursion itself is [FXT: comb/mset-perm-lex-rec.cc]:
```
1
void mset_perm_lex_rec::mset_perm_rec(ulong d)
2 {
3     if ( d>=n_ )
4     {
5         ++ct_; // work == number of recursions
6         ms_[d] = j; // put element in place
7         if ( r_[j]==0 ) // bucket now empty?
8         {
9             ulong f = nn_[jf]; // where we come from
10            nn_[jf] = nn_[j]; // let recursions skip over j
11            mset_perm_rec(d+1); // recursion
12            nn_[jf] = f; // remove skip
13            } else mset_perm_rec(d+1); // recursion
14        ++r_[j]; // put element back
15     }
16     else for (ulong jf=k_, j=nn_[jf]; j<k_; jf=j, j=nn_[j]) // for all nonempty buckets
17     {
18         --rct_; // work == number of recursions
19         --r_[j]; // take element from bucket
20         if ( d==n_ )
21         {
22             ++ct_; // work == number of recursions
23             visit_( *this );
24         }
25     }
26 }

Note that the test whether the current bucket is nonempty is omitted as empty buckets are skipped. Now the work involved with (regular) permutations is (less than) $e = 2.71828 \ldots$ times the number of the generated permutations. Usage of the class is shown in [FXT: comb/mset-perm-lex-rec2-demo.cc]. The permutations of 12 elements are generated at a rate of about 25 million per second, the combinations $\binom{15}{9}$ at about 40 million per second, and the permutations of $(2, 2, 2, 3, 3, 3)$ at about 20 million per second.

11.2.2 Iterative generation

The algorithm to generate the next permutation in lexicographic order given in section 10.1 on page 233 can be adapted for an iterative method for multiset permutations [FXT: class mset_perm_lex in comb/mset-perm-lex.h]:
```
1 class mset_perm_lex
2 {
3     public:
4         ulong k_; // number of different sorts of objects
5         ulong *r_; // number of elements '0' in r[0], '1' in r[1], ..., 'k-1' in r[k-1]
6         ulong n_; // number of objects
7         ulong *ms_; // multiset data in ms[0], ..., ms[n-1], sentinel at [-1]
691
```
11.2: Permutations of a multiset

public:
    mset_perm_lex(const ulong *r, ulong k)
{
    k_ = k;
    r_ = new ulong[k];
    for (ulong j=0; j<k_; ++j) r_[j] = r[j]; // get buckets
    n_ = 0;
    for (ulong j=0; j<k_; ++j) n_ += r_[j];
    ms_ = new ulong[n_+1];
    ms_[0] = 0; // sentinel
    ++ms_; // nota bene

    first();
}

void first()
{
    for (ulong j=0, i=0; j<k_; ++j)
        for (ulong h=r_[j]; h!=0; --h, ++i) ms_[i] = j;
}

bool next()
{
    // find for rightmost pair with p[i] < p[i+1]:
    const ulong n1 = n_ - 1;
    ulong i = n1;
    do { --i; } while ( ms_[i] >= ms_[i+1] ); // can touch sentinel
    if ( (long)i<0 ) return false; // last sequence is falling seq.

    // find rightmost element p[j] smaller than p[i]:
    ulong j = n1;
    while ( ms_[i] >= ms_[j] ) { --j; }
    swap2(ms_[i], ms_[j]);

    // Here the elements p[i+1], ..., p[n-1] are a falling sequence.
    // Reverse order to the right:
    ulong r = n1;
    ulong s = i + 1;
    while ( r > s ) { swap2(ms_[r], ms_[s]); --r; ++s; }
    return true;
}

Usage of the class is shown in [FXT: comb/mset-perm-lex-demo.cc]:

ulong ct = 0;
do {
    // visit
}while ( P.next() );

The permutations of 12 elements are generated at a rate of about 110 million per second, the combinations \( \binom{30}{15} \) at about 60 million per second, and the permutations of \( (2, 2, 2, 3, 3, 3) \) at about 82 million per second.

11.2.3 Minimal-change order

An algorithm for the generation of a Gray code for the permutations of a multiset given by Fred Lunnon [priv.comm.], figure 11.2-B shows examples of the ordering. It is a generalization of Trotter’s order for permutations described in section 10.7 on page 254. The implementation is [FXT: class mset_perm_gray in comb/mset-perm-gray.h]:

class mset_perm_gray
{
public:
    ulong *ms_; // permuted elements (Lunnon’s R_[])
}
Chapter 11: Subsets and permutations of a multiset

Figure 11.2-B: Gray code for permutations of multisets: the multiset (2, 2, 1) (left, with swaps), (6, 2) (combinations $\binom{6+2}{2}$), middle), and (1, 1, 1, 1) (permutations of four elements, right). Dots denote ones.

```c
5 ulong *P_; // permutation
6 ulong *Q_; // inverse permutation
7 ulong *D_; // direction
8 ulong k_; // number of different sorts of objects
9 ulong n_; // number of objects
10 ulong sw1_, sw2_; // positions swapped with last update
11 ulong *r_; // number of elements '1' in r[0], '2' in r[1], ..., 'k' in r[k-1]
12 public:
13     mset_perm_gray(const ulong *r, ulong k)
14     {
15         k_ = k;
16         r_ = new ulong[k_];
17         for (ulong j=0; j<k_; ++j) r_[j] = r[j];
18         n_ = 0;
19         for (ulong j=0; j<k_; ++j) n_ += r_[j];
20         ms_ = new ulong[n_+4];
21         P_ = new ulong[n_+4];
22         Q_ = new ulong[n_+4];
23         D_ = new ulong[n_+4];
24         first();
25     }
26     ~mset_perm_gray() // destructor
27     {
28         delete[] ms_; delete[] P_; delete[] Q_; delete[] D_;
29     }
30
31     const ulong * data() const { return ms_+1; }
32     void get_swaps(ulong &sw1, ulong &sw2) const { sw1=sw1_; sw2=sw2_; }
33
34         The arrays have four extra elements that are used as sentinels:
35         void first()
36         {
37             sw1_ = sw2_ = 0;
38         }
```
11.2: Permutations of a multiset

for (ulong j=0, i=1; j<k_; ++j)
  for (ulong h=r_[j]; h!=0; --h, ++i) ms_[i] = j + 1;

const ulong n = n_;
for (ulong j=1; j<n; ++j) { P_[j] = j; Q_[j] = j; D_[j] = +1UL; }

// sentinels:
ms_[0] = 0; P_[0] = 0; Q_[0] = 0; D_[0] = 0;
ulong j;
j = n+1; ms_[j] = 0; P_[j] = 0; Q_[j] = n+2; D_[j] = 0;
j = n+2; ms_[j] = k+1; P_[j] = n+1; Q_[j] = n+3; D_[j] = +1;
j = n+3; ms_[j] = k+2; P_[j] = n+2; Q_[j] = 0; D_[j] = +1;
}

To compute the successor we find the first run of identical elements that can be moved:

bool next() {
  // locate earliest unblocked element at j, starting at blocked element 0
  ulong j = 0, i = 0, d = 0, l = 0; // init of l not needed
  while ( ms_[j] >= ms_[i] )
    D_[j] = -d; // blocked at j; reverse drift d pre-emptively
  // next element at j, neighbor at i:
  j = Q_[P_[j]+1];
  d = D_[j];
  i = j+d;
  if ( ms_[j-1] != ms_[j] ) l = j; // save left end of run in l
  else
    if ( (long)d < 0) i = l-1;
  }

  if ( j > n_ ) return false; // current permutation is last
  // restore left end at head of run
  // shift run of equal rank from i-d,i-2d,...,l to i,i-d,...,l+d
  ulong e = D_[i], p = P_[i]; // save neighbor drift e and identifier p
  for (ulong k=i; k!=l; k-=d)
    P_[k] = P_[k-d];
  Q_[P_[k]] = k;
  D_[k] = -1UL; // reset drifts of run tail elements
}

sw1_ = i - 1; sw2_ = 1 - 1; // save positions swapped
swap2(ms_[i], ms_[l]);
D_[1] = e; D_[i] = d; // restore drifts of head and neighbor
P_[1] = p; Q_[p] = 1; // wrap neighbor around to other end
return true;
}

The rate of generation is roughly 40 M/s [FXT: comb/mset-perm-gray-demo.cc].
Chapter 12

Gray codes for strings with restrictions

We give constructions for Gray codes for strings with certain restrictions, such as forbidding two successive zeros or nonzero digits. The constraints considered are such that the number of strings of a given type satisfies a linear recursion with constant coefficients.

\[
\begin{align*}
\text{W}(n) &= \text{[120 W(n-3)]} + \text{rev([10 W(n-2)])} + \text{[00 W(n-2)]} \\
&= \begin{cases} 
0 & \text{if } n = 0 \\
10 & \text{if } n = 1 \\
120 & \text{if } n = 2 \\
\text{[120 . W(n-3)]} & \text{if } n > 2 \end{cases}
\end{align*}
\]

The algorithms are given as list recursions. For example, write \(W(n)\) for the list of \(n\)-digit words (of a certain type), write \(W^R(n)\) for the reversed list, and \([x . W(n)]\) for the list with the word \(x\) prepended at each word. The recursion for a Gray code is

\[
W(n) = \begin{bmatrix} 0 & 0 & W(n - 2) \\ 1 & 0 & W^R(n - 2) \\ 1 & 2 & 0 & W(n - 3) \end{bmatrix}
\]

A relation like this always implies a backward version which is obtained by reversing the order of the sublists on the right hand side and additionally reversing each sublist

\[
W^R(n) = \begin{bmatrix} 1 & 2 & 0 & W^R(n - 3) \\ 1 & 0 & W(n - 2) \\ 0 & 0 & W^R(n - 2) \end{bmatrix}
\]

The construction is illustrated in figure 12.0-A. An implementation of the algorithm is [FXT: comb/fib-alt-gray-demo.cc].
Chapter 12: Gray codes for strings with restrictions

void X_rec(ulong d, bool z)
{
  if ( d>=n )
    if ( d<=n+1 ) // avoid duplicates
    {
      visit();
    }
  else
    {
      if ( z )
        {
          rv[d]=0; rv[d+1]=0; X_rec(d+2, z);
          rv[d]=1; rv[d+1]=0; X_rec(d+2, !z);
          rv[d]=1; rv[d+1]=2; rv[d+2]=0; X_rec(d+3, z);
        }
      else
        {
          rv[d]=1; rv[d+2]=0; X_rec(d+3, z);
          rv[d]=1; rv[d+1]=0; X_rec(d+2, !z);
          rv[d]=0; rv[d+1]=0; X_rec(d+2, z);
        }
    }
}

The initial call is X_rec(0, 0). The parameter z determines whether the list is generated in forward or backward order. No optimizations are made as these tend to obscure the idea. Here we could omit one statement rv[d]=1; in both branches, replace the arguments z and !z in the recursive calls by constants, or create an iterative version.

The number $w(n)$ of words $W(n)$ is determined by (some initial values and) a recursion and that can be obtained by counting the size of the lists on both sides of the recursion relation. With relation 12.0-1 we obtain the recursion

$$w(n) = 2w(n-2) + w(n-3) \quad \text{(12.0-3)}$$

One can typically set $w(0) = 1$, there is one empty list and it satisfies all conditions. The numbers $w(n)$ are in fact the Fibonacci numbers.

12.1 Fibonacci words

A recursive routine to generate the Fibonacci words (binary words not containing two consecutive ones) can be given as follows:

```c
ulong n; // number of bits in words
ulong *rv; // bits of the word

void fib_rec(ulong d)
{
  if ( d>n ) visit();
  else
    {
      if ( d<n )
        {
          rv[d]=0; fib_rec(d+1);
          rv[d]=1; rv[d+1]=0; fib_rec(d+2);
        }
      else
        {
          rv[d]=0; rv[d+1]=0; fib_rec(d+2);
        }
    }
}
```

We allocate one extra element (a sentinel) to reduce the number of if-statements in the code:

```c
int main()
{
  n = 7;
  rv = new ulong[n+1]; // incl. sentinel rv[n]
  fib_rec(0);
  return 0;
}
```

The output (assuming visit() simply prints the array) is given in the left of figure 12.1-A. Note that with the $n$-bit Fibonacci Gray code the number of ones in the first and last, second and second-last, etc. tracks
are equal. Thereby the sequence of reversed words is also a Fibonacci Gray code. A simple modification of the routine generates a Gray code through the Fibonacci words [FXT: comb/fibgray-rec-demo.cc]:

```c
void fib_rec(ulong d, bool z)
{
    if ( d>=n ) visit();
    else
    {
        z = !z; // change direction for Gray code
        if ( z )
        {
            rv[d]=0; fib_rec(d+1, z);
            rv[d]=1; rv[d+1]=0; fib_rec(d+2, z);
        }
        else
        {
            rv[d]=1; rv[d+1]=0; fib_rec(d+2, z);
            rv[d]=0; fib_rec(d+1, z);
        }
    }
}
```

The variable \( z \) controls the direction in the recursion, it is changed unconditionally with each step. The \textbf{if-else} blocks can be merged into

```c
rv[d]=!z; rv[d+1]= z; fib_rec(d+1+!z, z);
rv[d]= z; rv[d+1]=!z; fib_rec(d+1+ z, z);
```

The algorithm is CAT (constant amortized time) and about 70 million objects are generated per second. A bit-level algorithm is given in section 1.29.2 on page 81.

The algorithm for the list of the length-\( n \) Fibonacci words \( F(n) \) can be given as a recursion:

\[
F(n) = \begin{bmatrix} 1 & 0 \cdot F^R(n-2) \\ 0 & F^R(n-1) \end{bmatrix} \]  

(12.1.1)

The generation could be sped up by merging two steps:

\[
F(n) = \begin{bmatrix} 100 \cdot F(n-3) \\ 1010 \cdot F(n-4) \\ 00 \cdot F(n-2) \\ 010 \cdot F(n-3) \end{bmatrix} \]  

(12.1.2)
12.2 Generalized Fibonacci words

![Figure 12.2-A: The 7-bit binary words with maximal 2 successive ones in lexicographic (top) and minimal-change (bottom) order. Dots denote zeros.](image)

![Figure 12.2-B: Recursive structure for the 7-bit binary words with maximal 2 successive ones.](image)

We generalize the Fibonacci words by allowing a fixed maximum value \( r \) of successive ones in a binary word. The Fibonacci words correspond to \( r = 1 \). Figure 12.2-A shows the 7-bit words with \( r = 2 \). The method to generate a Gray code for these words is a generalization of the recursion for the Fibonacci words. Write \( L_r(n) \) for the list of \( n \)-bit words with at most \( r \) successive ones, then the recursive structure for the Gray code is

\[
L_r(n) = \begin{bmatrix}
0 & L_r(n-1) \\
10 & L_r(n-2) \\
110 & L_r(n-3) \\
\vdots & \vdots \\
1^{r-2}0 & L_r(n-1-r+2) \\
1^{r-1}0 & L_r(n-1-r+1) \\
1^r0 & L_r(n-1-r)
\end{bmatrix}
\]  

(12.2-1)

Figure 12.2-B shows the structure for \( L_2(7) \), corresponding to the three lowest sublists on the right side of the equation. An implementation is [FXT: comb/maxrep-gray-demo.cc]:

```c
1 ulong n; // number of bits in words
2 ulong *rv; // bits of the word
3 long mr; // maximum number of successive ones
1
2
3
4
5
6
7
8
9
10
11
12

void maxrep_rec(ulong d, bool z)
{
    if ( d==n ) visit();
    else
        { 
            z = !z;
            long km = mr;
            if ( d+km > n ) km = n - d;
            if ( z )
                { 
                    // words: 0, 10, 110, 1110, ...
                    for (long k=0; k<km; ++k)
                    } 
            
```
Let \( w_r(n) \) be the number of \( n \)-bit words \( W_r(n) \) with \( \leq r \) successive ones. Taking the length of the lists on both sides of relation (12.2-1) we obtain the recursion

\[
w_r(n) = \sum_{j=0}^{r} w_r(n - 1 - j)
\]

(12.2-2)

where we set \( w_r(n) = 2^k \) for \( 0 \leq n \leq r \). The sequences for \( r \leq 5 \) start as

\[
\begin{array}{cccccccccccccccccccc}
\text{n:} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\text{r=1:} & 1 & 2 & 3 & 4 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 & 233 & 377 & 610 & 987 & 1597 \\
\text{r=2:} & 1 & 2 & 4 & 7 & 11 & 18 & 29 & 47 & 76 & 123 & 199 & 322 & 521 & 843 & 1364 & 2207 & 3571 \\
\text{r=3:} & 1 & 2 & 4 & 7 & 13 & 24 & 44 & 78 & 128 & 208 & 336 & 544 & 880 & 1424 & 2304 & 3728 & 6032 \\
\text{r=4:} & 1 & 2 & 4 & 7 & 13 & 24 & 44 & 78 & 128 & 208 & 336 & 544 & 880 & 1424 & 2304 & 3728 & 6032 \\
\text{r=5:} & 1 & 2 & 4 & 7 & 13 & 24 & 44 & 78 & 128 & 208 & 336 & 544 & 880 & 1424 & 2304 & 3728 & 6032 \\
\end{array}
\]
Chapter 12: Gray codes for strings with restrictions

For $r = 1$ we obtain the Fibonacci numbers, entry \[ A000045 \] of [245]. For $r = 2$ the tribonacci numbers, entry \[ A000073 \]; for $r = 3$ the tetranacci numbers, entry \[ A000078 \] for $r = 4$ the pentanacci numbers, entry \[ A001591 \]; for $r = 5$ the hexanacci numbers, entry \[ A001592 \]. The variant of the Fibonacci sequence where each number is the sum of its $k$ predecessors is also called Fibonacci $k$-step sequence. The generating function for $w_r(n)$ is

$$\sum_{n=0}^{\infty} w_r(n) x^n = \frac{\sum_{k=0}^{r-1} x^k}{1 - \sum_{k=1}^{r+1} x^k} \quad (12.2-3)$$

Alternative Gray code for words without substrings 111 ($r = 2$)

![Figure 12.2-D: The 7-bit binary words with maximal 2 successive ones in a minimal-change order.](image)

The list recursion for the Gray code for binary words without substrings 111 is the special case $r = 2$ of relation $12.2-1$ on page 302:

$$L_2(n) = \begin{bmatrix} 110 & . & L_R^R(n-3) \\ 10 & . & L_R^R(n-2) \\ 0 & . & L_R^R(n-1) \end{bmatrix} \quad (12.2-4)$$

A different Gray code is obtained via

$$L'_2(n) = \begin{bmatrix} 10 & . & L'_2(n-2) \\ 110 & . & L'_2(n-3) \\ 0 & . & L'_2(n-1) \end{bmatrix} \quad (12.2-5)$$

The ordering is shown in figure [12.2-D] it was created with the program \[ FXT: comb/no111-gray-demo.cc \].

Alternative Gray code for words without substrings 1111 ($r = 3$)

A list recursion for an alternative Gray code for binary words without substrings 1111 ($r = 3$) is

$$L'_3(n) = \begin{bmatrix} 110 & . & L'_3^R(n-3) \\ 0 & . & L'_3^R(n-1) \\ 1110 & . & L'_3^R(n-4) \\ 10 & . & L'_3^R(n-2) \end{bmatrix} \quad (12.2-6)$$

The ordering is shown in figure [12.2-E] it was created with the program \[ FXT: comb/no1111-gray-demo.cc \]. For all odd $r \geq 3$ a Gray code can be obtained by a list recursion where the prefixes with an even number of ones are followed by those with an odd number of ones. For example, with $r = 5$ the recursion is

$$L'_5(n) = \begin{bmatrix} 11110 & . & L'_5^R(n-7) \\ 110 & . & L'_5^R(n-3) \\ 0 & . & L'_5^R(n-1) \\ 111110 & . & L'_5^R(n-6) \\ 1110 & . & L'_5^R(n-4) \\ 10 & . & L'_5^R(n-2) \end{bmatrix} \quad (12.2-7)$$

[fxtbook draft of 2008-August-17]
12.3 Digit $x$ followed by at least $x$ zeros

A implementation is \[\text{FXT}: \text{comb/gexz-gray-demo.cc}\]:

```
ulong n; // number of digits in words
ulong *rv; // digits of the word
ulong mr; // radix== mr+1

void gexz_rec(ulong d, bool z)
{
    if ( d>=n ) visit();
    else
    {
        if ( z )
        {
            // words 0, 10, 200, 3000, 40000, ...
            ulong k = 0;
            do
            {
                rv[d]=k;
                for (ulong j=1; j<k; ++j) rv[d+j] = 0;
                gexz_rec(d+k+1, !z);
            } while ( k++ < 1000000000L );
        }
    }
}
```

Figure 12.2-E: The 7-bit binary words with maximal 3 successive ones in a minimal-change order.

Figure 12.3-A: Gray code for the length-5 words with maximal digit 3 where a digit $x$ is followed by at least $x$ zeros. Dots denote zeros.

Figure 12.3-A shows a Gray code for the length-5 words with maximal digit 3 where a digit $x$ is followed by at least $x$ zeros. For the Gray code list $Z_r(n)$ of the length-$n$ words with maximal digit $r$ we have

$$Z_r(n) = \begin{bmatrix} 0 \cdot Z^R_r(n-1) \\ 10 \cdot Z^R_r(n-2) \\ 200 \cdot Z^R_r(n-3) \\ 3000 \cdot Z^R_r(n-4) \\ \vdots \\ r0^r \cdot Z^R_r(n-r-1) \end{bmatrix}$$

(12.3-1)
Chapter 12: Gray codes for strings with restrictions

Let $z_r(n)$ be the number of $n$-bit words $Z_r(n)$, then

$$z_r(n) = \sum_{j=1}^{r+1} z_r(n-j) \quad (12.3-2)$$

where we set $z_r(n) = 1$ for $n \leq 0$. The sequences for $r \leq 5$ start as

<table>
<thead>
<tr>
<th>r=1</th>
<th>1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597</th>
</tr>
</thead>
<tbody>
<tr>
<td>r=2</td>
<td>1 3 5 9 17 31 61 123 247 495 989 1977 3957 7915</td>
</tr>
<tr>
<td>r=3</td>
<td>1 4 7 11 20 31 51 92 163 295 557 1054 1951</td>
</tr>
<tr>
<td>r=4</td>
<td>1 5 9 17 33 65 129 253 497 994 1977 3955</td>
</tr>
<tr>
<td>r=5</td>
<td>1 6 11 21 43 85 169 339 681 1372 2740</td>
</tr>
</tbody>
</table>

For $r = 1$ we obtain the Fibonacci numbers, entry A000045 of [245]; For $r = 2$ the tribonacci numbers, entry A000213; for $r = 3$ the tetranacci numbers, entry A000288; for $r = 4$ the pentanacci numbers, entry A000322; for $r = 5$ the hexanacci numbers, entry A000383. Note that the sequences for $r \geq 2$ are different from those obtained via relation 12.2-2 on page 303 as the starting values differ.

12.4 Generalized Pell words

12.4.1 Gray code for Pell words

A Gray code of the Pell words (ternary words without the substrings "21" and "22") can be computed as follows:

```c
ulong n; // number of digits in words
ulong *rv; // digits of the word
bool zq; // order: 0==>Lex, 1==>Gray
```

Figure 12.4-A: Start and end of the lists of five digit Pell words in counting order (top) and Gray code order (bottom). The lowest row is the least significant digit, dots denote zeros.
12.4: Generalized Pell words

void pell_rec(ulong d, bool z)
{
  if ( d==n ) visit();
  else
    {
      if ( 0==z )
        {
          rv[d]=0; pell_rec(d+1, z);
          rv[d]=1; pell_rec(d+1, zq^z);
          rv[d]=2; rv[d+1]=0; pell_rec(d+2, z);
        }
      else
        {
          rv[d]=2; rv[d+1]=0; pell_rec(d+2, z);
          rv[d]=1; pell_rec(d+1, zq^z);
          rv[d]=0; pell_rec(d+1, z);
        }
    }
}

The global boolean variable $zq$ controls whether the counting order or the Gray code is generated. The code is given in [FXT: comb/pellgray-rec-demo.cc]. Both orderings are shown in figure 12.4-A. About 110 million words per second are generated. The computation of a function whose power series coefficients are related to the Pell Gray code is described in section 36.12.3 on page 756.

12.4.2 Gray code for generalized Pell words

\[ P_r(n) = \begin{bmatrix}
0 & P_r(n-1) & P_r^R(n-1) \\
1 & P_r^R(n-1) \\
2 & P_r(n-1) \\
\vdots & \\
(r-1) & P_r^R(n-1) \\
(r) & 0 & P_r(n-2)
\end{bmatrix} \]  (12.4-1a)

if $r$ is even, and by the recursion

\[ P_r(n) = \begin{bmatrix}
0 & P_r^R(n-1) \\
1 & P_r(n-1) \\
2 & P_r^R(n-1) \\
\vdots & \\
(r-1) & P_r(n-1) \\
(r) & 0 & P_r^R(n-2)
\end{bmatrix} \]  (12.4-1b)

if $r$ is odd. Figure 12.4-B shows a Gray code for the 4-digit strings with $r = 3$. An implementation of the algorithm is [FXT: comb/pellgen-gray-demo.cc]:

![Figure 12.4-B: Gray code for 4-digit radix-4 strings with no substring 3x with $x \neq 0$.](image)

A generalization of the Pell words are the radix-$(r+1)$ strings where the substring $rx$ with $x \neq 0$ is forbidden (that is, a nine can only be followed by a zero). Let $P_r(n)$ the length-$n$ words, a Gray code for these strings can be generated by the recursion

The global boolean variable $zq$ controls whether the counting order or the Gray code is generated. The code is given in [FXT: comb/pellgray-rec-demo.cc]. Both orderings are shown in figure 12.4-A. About 110 million words per second are generated. The computation of a function whose power series coefficients are related to the Pell Gray code is described in section 36.12.3 on page 756.

12.4.2 Gray code for generalized Pell words

A generalization of the Pell words are the radix-$(r+1)$ strings where the substring $rx$ with $x \neq 0$ is forbidden (that is, a nine can only be followed by a zero). Let $P_r(n)$ the length-$n$ words, a Gray code for these strings can be generated by the recursion

\[ P_r(n) = \begin{bmatrix}
0 & P_r(n-1) & P_r^R(n-1) \\
1 & P_r^R(n-1) \\
2 & P_r(n-1) \\
\vdots & \\
(r-1) & P_r^R(n-1) \\
(r) & 0 & P_r(n-2)
\end{bmatrix} \]  (12.4-1a)

if $r$ is even, and by the recursion

\[ P_r(n) = \begin{bmatrix}
0 & P_r^R(n-1) \\
1 & P_r(n-1) \\
2 & P_r^R(n-1) \\
\vdots & \\
(r-1) & P_r(n-1) \\
(r) & 0 & P_r^R(n-2)
\end{bmatrix} \]  (12.4-1b)

if $r$ is odd. Figure 12.4-B shows a Gray code for the 4-digit strings with $r = 3$. An implementation of the algorithm is [FXT: comb/pellgen-gray-demo.cc]:

![Figure 12.4-B: Gray code for 4-digit radix-4 strings with no substring 3x with $x \neq 0$.](image)
Chapter 12: Gray codes for strings with restrictions

ulong n; // number of digits in words
ulong *rv; // digits of the word (radix r+1)
long r; // Forbidden substrings are [r, x] where x! = 0

void pellgen_rec(ulong d, bool z)
{
  if (d >= n) visit();
  else
  {
    const bool p = r & 1; // parity of r
    rv[d] = 0;
    if (z)
    {
      for (long k = 0; k < r; ++k) { rv[d] = k; pellgen_rec(d + 1, z ^ p ^ (k & 1)); }
      { rv[d] = r; rv[d+1] = 0; pellgen_rec(d+2, p ^ z); }
    }
    else
    {
      { rv[d] = r; rv[d+1] = 0; pellgen_rec(d+2, p ^ z); }
      for (long k = r - 1; k >= 0; --k) { rv[d] = k; pellgen_rec(d+1, z ^ p ^ (k & 1)); }
    }
  }
}

With \( r = 1 \) we again obtain the Gray code for Fibonacci words. Taking the number \( p_r(n) \) of words \( P_r(n) \) on both sides of relations 12.4-1a and 12.4-1b we find

\[
p_r(n) = r p_r(n) + p_r(n-2) \tag{12.4-2}\]

where \( p_r(0) = 1 \) and \( p_r(1) = r + 1 \). For \( r \leq 5 \) the sequences start as

\[
\begin{array}{ccccccccccccccc}
n: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
r=1: & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 & 233 \\
r=2: & 1 & 3 & 7 & 17 & 41 & 99 & 239 & 577 & 1393 & 3363 & 8119 & 19601 \\
r=3: & 1 & 4 & 13 & 44 & 155 & 544 & 1897 & 6859 & 24274 & 89203 & 321931 & 1136681 \\
r=4: & 1 & 5 & 21 & 89 & 377 & 1597 & 6765 & 28657 & 121393 & 514229 & 2178309 & 9227465 \\
r=5: & 1 & 6 & 31 & 161 & 836 & 4341 & 22541 & 117046 & 607771 & 3155901 & 16387276 & 85092281 \\
\end{array}
\]

The sequences are the following entries of [245]: \( r = 1: \) A000045 \( r = 2: \) A001333 \( r = 3: \) A003688 \( r = 4: \) A015448 \( r = 5: \) A015449  \( r \leq 5 \) The generating function for \( p_r(n) \) is

\[
\sum_{n=0}^{\infty} p_r(n) x^n = \frac{1 + x}{1 - r x - x^2} \tag{12.4-3}\]

### 12.5 Sparse signed binary words

Figure 12.5-A shows a minimal-change order (Gray code) for the sparse signed binary words (nonadjacent form (NAF), see section 1.24 on page 66). Note that we allow a digit to switch between +1 and −1. If all words with any positive digit (‘P’) are omitted then we obtain the Gray code for Fibonacci words given in section 12.1 on page 300.

A recursive routine for the generation of the Gray code is given in [FXT: comb/naf-gray-rec-demo.cc]:

```
ulong n; // number of digits of the string
int *rv; // the string
```
12.5: Sparse signed binary words

```c
void sb_rec(ulong d, bool z)
{
    if ( d==n ) visit();
    else
    {
        if ( 0==z )
            { rv[d]=0; sb_rec(d+1, 1);
              rv[d]=-1; rv[d+1]=0; sb_rec(d+2, 1);
              rv[d]=+1; rv[d+1]=0; sb_rec(d+2, 0);
            }
        else
        {
            rv[d]=+1; rv[d+1]=0; sb_rec(d+2, 1);
            rv[d]=-1; rv[d+1]=0; sb_rec(d+2, 0);
            rv[d]=0; sb_rec(d+1, 0);
        }
    }
}
```

About 120 million words per second are generated.

Let $S(n)$ be the number of $n$-digit sparse signed binary numbers (of both signs), and $P(n)$ be the number of positive $n$-digit sparse signed binary numbers, then

<table>
<thead>
<tr>
<th>$n$</th>
<th>$S(n)$</th>
<th>$P(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>43</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>85</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>171</td>
<td>86</td>
</tr>
<tr>
<td>8</td>
<td>341</td>
<td>171</td>
</tr>
<tr>
<td>9</td>
<td>683</td>
<td>342</td>
</tr>
<tr>
<td>10</td>
<td>1365</td>
<td>683</td>
</tr>
<tr>
<td>11</td>
<td>2731</td>
<td>1366</td>
</tr>
<tr>
<td>12</td>
<td>5461</td>
<td>2731</td>
</tr>
<tr>
<td>13</td>
<td>10923</td>
<td>5462</td>
</tr>
<tr>
<td>14</td>
<td>21845</td>
<td>10923</td>
</tr>
<tr>
<td>15</td>
<td>43691</td>
<td>21846</td>
</tr>
<tr>
<td>16</td>
<td>87381</td>
<td>43691</td>
</tr>
</tbody>
</table>

The sequence $S(n)$ is entry A001045 of [245], the sequence $P(n)$ is entry A005578. We have (with $e := n \mod 2$)

$$S(n) = \frac{2^{n+2} - 1 + 2e}{3} = 2S(n-1) - 1 + 2e \quad \text{(12.5-1a)}$$
$$P(n) = \frac{2^{n+1} + 1 + e}{3} = 2P(n-1) - 1 - e = S(n-1) + e \quad \text{(12.5-1b)}$$

Almost Gray code for positive words *

```
<<< >>>
```

If we start with the following routine that calls `sb_rec()` only after a one has been inserted, we obtain an ordering of the positive numbers:

1. void pos_rec(ulong d, bool z)
2. {
3.     if ( d==n ) visit();
4.     else
5.         {
6.             if ( 0==z )
7.                 { rv[d]=0; pos_rec(d+1, 1);
8.                 rv[d]=+1; rv[d+1]=0; sb_rec(d+2, 1);
9.             }
10.         }
11.     }
12. }

Figure 12.5-B: An ordering of the 86 sparse 7-bit positive signed binary words that is almost a Gray code. The transitions that are not minimal are marked with ‘$>$’. Dots denote zeros.
The ordering obtained with \( n \)-digit words is a Gray code, except for \( n - 4 \) transitions. An ordering with only about \( n/2 \) non-Gray transitions is obtained by the more complicated recursion \cite{FXT:comb/naf-pos-rec-demo.cc}:

```c
void pos_AAA(ulong d, bool z)
{
  if ( d==n ) visit();
  else
    { 
      if ( 0==z )
        { 
          rv[d]=+1; rv[d+1]=0; sb_rec(d+2, 0); // 0
          rv[d]=0; pos_AAA(d+1, 1); // 1
        }
      else
        { 
          rv[d]=0; pos_BBB(d+1, 0); // 0
          rv[d]=+1; rv[d+1]=0; sb_rec(d+2, 1); // 1
        }
    }
}
void pos_BBB(ulong d, bool z)
{
  if ( d==n ) visit();
  else
    { 
      if ( 0==z )
        { 
          rv[d]=+1; rv[d+1]=0; sb_rec(d+2, 1); // 1
          rv[d]=0; pos_BBB(d+1, 1); // 1
        }
      else
        { 
          rv[d]=0; pos_AAA(d+1, 0); // 0
          rv[d]=+1; rv[d+1]=0; sb_rec(d+2, 0); // 0
        }
    }
}
```

The initial call is `pos_AAA(0,0)`. The result for \( n = 7 \) is shown in figure \ref{fig:12.5-B}. We list the number \( N \) of non-Gray transitions and the number of digit changes \( X \) in excess of a Gray code for \( n \leq 30 \):

\[
\begin{array}{cccccccccccccccccccccccccccc}
\text{N:} & 0 & 0 & 0 & 0 & 1 & 2 & 2 & 2 & 3 & 4 & 4 & 4 & 5 & 6 & 6 & 6 & 7 & 8 & 8 & 8 & 8 & 9 & 9 & 10 & 10 & 11 & 12 & 12 & 12 & 13 & 14 \\
\end{array}
\]

### 12.6 Strings with no two successive nonzero digits

A Gray code for the length-\( n \) strings with radix \( (r+1) \) and no two successive nonzero digits is obtained by the following recursion for the list \( D_r(n) \):

\[
D_r(n) = \begin{bmatrix}
0 & D_r^R(n-1) \\
10 & D_r^R(n-1) \\
20 & D_r(n-1) \\
30 & D_r^R(n-1) \\
40 & D_r(n-1) \\
50 & D_r^R(n-1) \\
\vdots & \vdots \\
\end{bmatrix}
\]

An implementation is \cite{FXT:comb/ntnz-gray-demo.cc}:
Strings with no two successive nonzero digits

Figure 12.6-A: Gray code for the length-4 radix-4 strings with no two successive nonzero digits.

```
ulong n; // length of strings
ulong *rv; // digits of strings
ulong mr; // max digit

void ntnz_rec(ulong d, bool z)
{
    if ( d>=n ) visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=0; ntnz_rec(d+1, 1);
            for (ulong t=1; t<=mr; ++t) { rv[d]=t; rv[d+1]=0; ntnz_rec(d+2, t&1); }
        }
        else
        {
            for (ulong t=mr; t>0; --t) { rv[d]=t; rv[d+1]=0; ntnz_rec(d+2, !(t&1)); }
            rv[d]=0; ntnz_rec(d+1, 0);
        }
    }
}
```

Figure 12.6-A shows the Gray code for length-4, radix-4 \((r=3)\) strings. With \(r=2\) and upon replacing \(1\) with \(-1\) and \(2\) with \(+1\) we obtain the Gray code for the sparse binary words (figure 12.5-A on page 308). With \(r=1\) we again obtain the Gray code for the Fibonacci words.

Counting the elements on both sides of relation 12.6-1 we find that for the number \(d_r(n)\) of strings in the list \(D_r(n)\) we have

\[
d_r(n) = d_r(n-1) + r d_r(n-2) \tag{12.6-2}
\]

where \(d_r(0) = 1\) and \(d_r(1) = r + 1\). The sequences of these numbers start as

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>r=1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>233</td>
<td>377</td>
<td>610</td>
</tr>
<tr>
<td>r=2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>21</td>
<td>36</td>
<td>67</td>
<td>130</td>
<td>257</td>
<td>505</td>
<td>982</td>
<td>1942</td>
<td>3883</td>
<td>7765</td>
</tr>
<tr>
<td>r=3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>29</td>
<td>61</td>
<td>123</td>
<td>255</td>
<td>511</td>
<td>1023</td>
<td>2047</td>
<td>4095</td>
<td>8191</td>
<td>16385</td>
</tr>
<tr>
<td>r=4</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>20</td>
<td>40</td>
<td>81</td>
<td>162</td>
<td>325</td>
<td>650</td>
<td>1301</td>
<td>2602</td>
<td>5203</td>
<td>10406</td>
<td>20813</td>
</tr>
</tbody>
</table>

These are the following entries of [245]: \(r=1: \text{A000045}\), \(r=2: \text{A001045}\), \(r=3: \text{A006130}\), \(r=4: \text{A006131}\), \(r=5: \text{A015440}\), \(r=6: \text{A015441}\), \(r=7: \text{A015442}\), \(r=8: \text{A015443}\). The generating function for \(d_r(n)\) is

\[
\sum_{n=0}^{\infty} d_r(n) x^n = \frac{1 + r x}{1 - x - r x^2} \tag{12.6-3}
\]
12.7 Strings with no two successive zeros

A Gray code for the length-\(n\) strings with radix \((r+1)\) and no two successive zeros (see figure 12.7-A) is obtained by the recursion for the list \(Z_r(n)\) as follows:

\[
Z_r(n) = \begin{cases} 
[0 \ 1 \ Z_r(n-2)] & \text{for } r \text{ even,} \\
[0 \ 2 \ Z_r(n-2)] & \\
[0 \ 3 \ Z_r(n-2)] & \\
[0 \ 4 \ Z_r(n-2)] & \\
[0 \ 5 \ Z_r(n-2)] & \\
[0 \ 6 \ Z_r(n-2)] & \\
[0 \ 7 \ Z_r(n-2)] & \\
[0 \ 8 \ Z_r(n-2)] & \\
[0 \ 9 \ Z_r(n-2)] & \\
[1 \ 0 \ Z_r(n-2)] & \\
[1 \ 1 \ Z_r(n-2)] & \\
[1 \ 2 \ Z_r(n-2)] & \\
[1 \ 3 \ Z_r(n-2)] & \\
[1 \ 4 \ Z_r(n-2)] & \\
[1 \ 5 \ Z_r(n-2)] & \\
[1 \ 6 \ Z_r(n-2)] & \\
[1 \ 7 \ Z_r(n-2)] & \\
[1 \ 8 \ Z_r(n-2)] & \\
[1 \ 9 \ Z_r(n-2)] & \\
[2 \ 0 \ Z_r(n-2)] & \\
[2 \ 1 \ Z_r(n-2)] & \\
[2 \ 2 \ Z_r(n-2)] & \\
[2 \ 3 \ Z_r(n-2)] & \\
[2 \ 4 \ Z_r(n-2)] & \\
[2 \ 5 \ Z_r(n-2)] & \\
[2 \ 6 \ Z_r(n-2)] & \\
[2 \ 7 \ Z_r(n-2)] & \\
[2 \ 8 \ Z_r(n-2)] & \\
[2 \ 9 \ Z_r(n-2)] & \\
[3 \ 0 \ Z_r(n-2)] & \\
[3 \ 1 \ Z_r(n-2)] & \\
[3 \ 2 \ Z_r(n-2)] & \\
[3 \ 3 \ Z_r(n-2)] & \\
[3 \ 4 \ Z_r(n-2)] & \\
[3 \ 5 \ Z_r(n-2)] & \\
[3 \ 6 \ Z_r(n-2)] & \\
[3 \ 7 \ Z_r(n-2)] & \\
[3 \ 8 \ Z_r(n-2)] & \\
[3 \ 9 \ Z_r(n-2)] & \\
[4 \ 0 \ Z_r(n-2)] & \\
[4 \ 1 \ Z_r(n-2)] & \\
[4 \ 2 \ Z_r(n-2)] & \\
[4 \ 3 \ Z_r(n-2)] & \\
[4 \ 4 \ Z_r(n-2)] & \\
[4 \ 5 \ Z_r(n-2)] & \\
[4 \ 6 \ Z_r(n-2)] & \\
[4 \ 7 \ Z_r(n-2)] & \\
[4 \ 8 \ Z_r(n-2)] & \\
[4 \ 9 \ Z_r(n-2)] & \\
[5 \ 0 \ Z_r(n-2)] & \\
[5 \ 1 \ Z_r(n-2)] & \\
[5 \ 2 \ Z_r(n-2)] & \\
[5 \ 3 \ Z_r(n-2)] & \\
[5 \ 4 \ Z_r(n-2)] & \\
[5 \ 5 \ Z_r(n-2)] & \\
[5 \ 6 \ Z_r(n-2)] & \\
[5 \ 7 \ Z_r(n-2)] & \\
[5 \ 8 \ Z_r(n-2)] & \\
[5 \ 9 \ Z_r(n-2)] & \\
[6 \ 0 \ Z_r(n-2)] & \\
[6 \ 1 \ Z_r(n-2)] & \\
[6 \ 2 \ Z_r(n-2)] & \\
[6 \ 3 \ Z_r(n-2)] & \\
[6 \ 4 \ Z_r(n-2)] & \\
[6 \ 5 \ Z_r(n-2)] & \\
[6 \ 6 \ Z_r(n-2)] & \\
[6 \ 7 \ Z_r(n-2)] & \\
[6 \ 8 \ Z_r(n-2)] & \\
[6 \ 9 \ Z_r(n-2)] & \\
[7 \ 0 \ Z_r(n-2)] & \\
[7 \ 1 \ Z_r(n-2)] & \\
[7 \ 2 \ Z_r(n-2)] & \\
[7 \ 3 \ Z_r(n-2)] & \\
[7 \ 4 \ Z_r(n-2)] & \\
[7 \ 5 \ Z_r(n-2)] & \\
[7 \ 6 \ Z_r(n-2)] & \\
[7 \ 7 \ Z_r(n-2)] & \\
[7 \ 8 \ Z_r(n-2)] & \\
[7 \ 9 \ Z_r(n-2)] & \\
[8 \ 0 \ Z_r(n-2)] & \\
[8 \ 1 \ Z_r(n-2)] & \\
[8 \ 2 \ Z_r(n-2)] & \\
[8 \ 3 \ Z_r(n-2)] & \\
[8 \ 4 \ Z_r(n-2)] & \\
[8 \ 5 \ Z_r(n-2)] & \\
[8 \ 6 \ Z_r(n-2)] & \\
[8 \ 7 \ Z_r(n-2)] & \\
[8 \ 8 \ Z_r(n-2)] & \\
[8 \ 9 \ Z_r(n-2)] & \\
[9 \ 0 \ Z_r(n-2)] & \\
[9 \ 1 \ Z_r(n-2)] & \\
[9 \ 2 \ Z_r(n-2)] & \\
[9 \ 3 \ Z_r(n-2)] & \\
[9 \ 4 \ Z_r(n-2)] & \\
[9 \ 5 \ Z_r(n-2)] & \\
[9 \ 6 \ Z_r(n-2)] & \\
[9 \ 7 \ Z_r(n-2)] & \\
[9 \ 8 \ Z_r(n-2)] & \\
[9 \ 9 \ Z_r(n-2)] & \\
\end{cases}
\]

Figure 12.7-A: Gray codes for strings with no two successive zeros: length-3 radix-4 (left), and length-4 radix-3 (right). Dots denote zeros.

An implementation is given in [FXT: comb/ntz-gray-demo.cc]:

```c
ulong n; // number of digits in words
ulong *rv; // digits of the word (radix r+1)
long r; // Forbidden substrings are [r, x] where x!=0

void ntz_rec(ulong d, bool z)
{
    if ( d==n ) visit();
    else
    {
        bool w = 0; // r-parity: w depends on z ...
        if ( r&1 ) w = !z; // ... if r odd
        if ( z )
        {
            // words 0X:
            rv[d] = 0;
            if ( d+2<n )
            {
                for (long k=1; k<r; ++k, w=!w) { rv[d+1]=k; ntz_rec(d+2, w); }
            }
        }
    }
}
```

[fxtbook draft of 2008-August-17]
12.8 Binary strings without substrings 1x1 or 1xy1

With \( r = 1 \) we obtain the complement of the minimal-change list of Fibonacci words. Let \( z_r(n) \) be the number of words \( W_r(n) \), we find

\[
z_r(n) = r z_r(n-1) + r z_r(n-1)
\]

where \( z_r(0) = 1 \) and \( z_r(1) = r + 1 \). The sequences for \( r \leq 5 \) start

<table>
<thead>
<tr>
<th>( r )</th>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>233</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>15</td>
<td>30</td>
<td>60</td>
<td>120</td>
<td>240</td>
<td>480</td>
<td>960</td>
<td>1920</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>17</td>
<td>34</td>
<td>70</td>
<td>141</td>
<td>282</td>
<td>564</td>
<td>1128</td>
<td>2256</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
</tr>
</tbody>
</table>

These (for \( r \leq 4 \)) are the following entries of [245]: \( r = 1: A000045; r = 2: A028859; r = 3: A25145; r = 4: A086347 \). The generating function for \( z_r(n) \) is

\[
\sum_{n=0}^{\infty} z_r(n) x^n = \frac{1 + x}{1 - r x - r x^2}
\]

12.8 Binary strings without substrings 1x1 or 1xy1

12.8.1 No substrings 1x1

A Gray code for binary strings with no substring 1x1 is shown in figure 12.8-A. The recursive structure for the list \( V(n) \) of the \( n \)-bit words is

\[
V(n) = \begin{bmatrix} 100 & V(n-3) \\ 1100 & V^R(n-4) \\ 0 & V(n-1) \end{bmatrix}
\]

The implied algorithm can be implemented as [FXT: comb/no1x1-gray-demo.cc]:

```c++
ulong n; // number of bits in words
ulong *rv; // bits of the word
```

[fxtbook draft of 2008-August-17]
void no1x1_rec(ulong d, bool z)
{
    if ( d==n ) { if ( d<=n+2 ) visit(); }
    else
    {
        if ( z )
        {
            rv[d]=1; rv[d+1]=0; rv[d+2]=0; no1x1_rec(d+3, z);
            rv[d]=1; rv[d+1]=1; rv[d+2]=0; rv[d+3]=0; no1x1_rec(d+4, !z);
            rv[d]=0; no1x1_rec(d+1, z);
        }
        else
        {
            rv[d]=0; no1x1_rec(d+1, z);
            rv[d]=1; rv[d+1]=1; rv[d+2]=0; rv[d+3]=0; no1x1_rec(d+4, !z);
            rv[d]=1; rv[d+1]=0; rv[d+2]=0; no1x1_rec(d+3, z);
        }
    }
}

The sequence of the numbers \( v(n) \) of length-\( n \) strings starts as

\[
\begin{array}{cccccccccccccc}
 n: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
v(n): & 1 & 2 & 4 & 6 & 9 & 15 & 25 & 40 & 64 & 104 & 169 & 273 & 441 & 714 & 1156 & 1870 & 3025 & 4895 \\
\end{array}
\]

This is entry \textbf{A006498} of [245]. The recurrence relation is

\[
v(n) = v(n-1) + v(n-3) + v(n-4) \tag{12.8-2}\]

The generating function is

\[
\sum_{n=0}^{\infty} v(n) x^n = \frac{1 + x + 2x^2 + x^3}{1 - x - x^3 - x^4} \tag{12.8-3}\]

### 12.8.2 No substrings 1xy1

Figure [12.8-B] shows a Gray code for binary words with no substring 1xy1. The recursion for the list of \( n \)-bit words \( Y(n) \) is

\[
Y(n) = \begin{bmatrix}
1000 & Y(n-4) \\
101000 & Y^R(n-6) \\
110000 & Y(n-6) \\
110000 & Y^R(n-5) \\
0 & Y(n-1)
\end{bmatrix} \tag{12.8-4}
\]

An implementation is given in \texttt{comb/no1xy1-gray-demo.cc}:
12.8: Binary strings without substrings 1x1 or 1yx1

Figure 12.8-B: The length-10 binary strings with no substring 1xy1 (where x and y are either 0 or 1) in minimal-change order. Dots denote zeros.

```c
void Y_rec(long p1, long p2, bool z)
{
    if ( p1 > p2 ) { visit(); return; }
    #define S1(a) rv[p1+0]=a
    #define S2(a,b) S1(a); rv[p1+1]=b;
    #define S3(a,b,c) S2(a,b); rv[p1+2]=c;
    #define S4(a,b,c,d) S3(a,b,c); rv[p1+3]=d;
    #define S5(a,b,c,d,e) S4(a,b,c,d); rv[p1+4]=e;
    #define S6(a,b,c,d,e,f) S5(a,b,c,d,e); rv[p1+5]=f;
    long d = p2 - p1;
    if ( z ) {
        if ( d >= 0 ) { S4(1,0,0,0); Y_rec(p1+4, p2, z); } // 1 0 0 0
        if ( d >= 2 ) { S6(1,0,1,0,0,0); Y_rec(p1+6, p2, !z); } // 1 0 1 0 0 0
        if ( d >= 2 ) { S6(1,1,1,0,0,0); Y_rec(p1+6, p2, z); } // 1 1 1 0 0 0
        if ( d >= 1 ) { S5(1,1,0,0,0); Y_rec(p1+5, p2, !z); } // 1 1 0 0 0
        if ( d >= 0 ) { S1(0); Y_rec(p1+1, p2, z); } // 0
    }
    else {
        if ( d >= 0 ) { S1(0); Y_rec(p1+1, p2, z); } // 0
        if ( d >= 1 ) { S5(1,1,0,0,0); Y_rec(p1+5, p2, !z); } // 1 1 0 0 0
        if ( d >= 2 ) { S6(1,0,1,0,0,0); Y_rec(p1+6, p2, !z); } // 1 0 1 0 0 0
        if ( d >= 2 ) { S6(1,1,1,0,0,0); Y_rec(p1+6, p2, z); } // 1 1 1 0 0 0
        if ( d >= 0 ) { S4(1,0,0,0); Y_rec(p1+4, p2, z); } // 1 0 0 0
    }
}
```

Note the conditions if ( d >= ? ) that make sure that no string appears repeated. The initial call is Y_rec(0, n-1, 0). The sequence of the numbers y(n) of length-n strings starts as

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>y(n)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>17</td>
<td>25</td>
<td>41</td>
<td>69</td>
<td>114</td>
<td>180</td>
<td>280</td>
<td>440</td>
<td>705</td>
<td>1137</td>
<td>1825</td>
<td>2905</td>
<td>4610</td>
</tr>
</tbody>
</table>

The generating function is

\[
\sum_{n=0}^{\infty} y(n) x^n = \frac{1 + x + 2x^2 + 4x^3 + 3x^4 + 2x^5}{1 - x - x^4 - x^5 - 2x^6} \quad (12.8-5)
\]
Chapter 12: Gray codes for strings with restrictions

12.8.3 Neither substrings 1x1 nor substrings 1xy1

A recursion for a Gray code of the of \( n \)-bit binary words \( Z(n) \) with no substrings 1x1 or 1xy1 (shown in figure 12.8-C) is

\[
Z(n) = \begin{bmatrix}
1 & 0 & 0 & 0 & Z(n-4) \\
1 & 1 & 0 & 0 & ZR(n-5)
\end{bmatrix}
\]

The sequence of the numbers \( z(n) \) of length-\( n \) strings starts as

\[
\begin{array}{cccccccccccccccc}
n: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
z(n): & 1 & 2 & 4 & 6 & 8 & 11 & 17 & 27 & 41 & 60 & 88 & 132 & 200 & 301 & 449 & 669 & 1001 & 1502
\end{array}
\]

The sequence is (apart from three leading ones) entry \[A079972\] of \[245\] where two combinatorial interpretations are given:

Number of permutations satisfying \(-k \leq p(i)-i \leq r\) and \(p(i)-i\) not in \(I\), \(i=1..n\), with \(k=1\), \(r=4\), \(I=\{1,2\}\).

Number of compositions (ordered partitions) of \(n\) into elements of the set \{1,4,5\}.

The generating function is

\[
\sum_{n=0}^{\infty} z(n) x^n = \frac{1 + x + 2 x^2 + 2 x^3 + x^4}{1 - x - x^4 - x^5} \quad (12.8-7)
\]
Chapter 13

Parenthesis strings

13.1 Co-lexicographic order

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Co-lexicographic order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>2: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>3: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>4: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>5: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>6: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>7: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>8: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>9: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>10: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>11: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>12: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>13: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>14: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>15: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>16: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>17: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>18: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>19: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>20: ((((()))))</td>
<td>11111....</td>
</tr>
<tr>
<td>21: ((((()))))</td>
<td>11111....</td>
</tr>
</tbody>
</table>

Figure 13.1-A: All (42) valid strings of 5 pairs of parenthesis in colex order.

An iterative scheme to generate all valid ways to group parenthesis can be obtained from a modified version of the combinations in co-lexicographic order (see section 6.1.2 on page 178). For $n = 5$ pairs the possible combinations are shown in figure 13.1-A. This is the output of [FXT: comb/paren-demo.cc].

Consider the sequences to the right of the paren strings as binary words. If the leftmost block has more than a single one then its rightmost one is moved one position to the right. Else (the leftmost block consists of a single one) the ones of the longest run of the repeated pattern ‘1.’ at the left are gathered at the left end and the rightmost one in next block of ones (which contains at least two ones) is moved by one position to the right and the rest of the block is gathered at the left end (see the transitions from #14 to #15 or #37 to #38).

The generator is [FXT: class paren in comb/paren.h]:

```c++
1  class paren
2  {
3      ulong k_; // Number of paren pairs
4      ulong n_; // ==2*k
5      ulong *x_; // Positions where a opening paren occurs
```
Chapter 13: Parenthesis strings

```cpp
7 char *str_; // String representation, e.g. "((())())()"

9 public:
10   paren(ulong k)
11   {
12     k_ = (k>1 ? k : 2); // not zero (empty) or one (trivial: "()")
13     n_ = 2 * k_
14     x_ = new ulong[n_ + 1];
15     x_[k_] = 999; // sentinel
16     str_ = new char[n_ + 1];
17     str_[n_] = 0;
18     first();
19   }
20
21-~paren()
22   {
23     delete [] x_;  
24     delete [] str_;
25   }
26
27   void first() { for (ulong i=0; i<k_; ++i) x_[i] = i; }
28
29   void last() { for (ulong i=0; i<k_; ++i) x_[i] = 2*i; }
30
31--snip--
```

The code for the computation of the successor and predecessor is quite concise. A sentinel x[k] is used to save one branch with the generation of the next string.

```cpp
1 ulong next() // return zero if current paren is the last
2 {
3   // if ( k==1 ) return 0; // uncomment to make algorithm work for k==1
4   ulong j = 0;
5   if ( x_[1] == 2 )
6     {
7       // scan for low end == 010101:
8         j = 1;
9         while ( x_[j]==2 ) ++j; // can touch sentinel
10        if ( j==k_ ) { first(); return 0; }
11   }
12   
13   // scan block:
14   while ( i == (x_[j+1] - x_[j]) ) { ++j; }
15   ++x_[j]; // move edge element up
16   for (ulong i=0; i<j; ++i) x_[i] = i; // attach block at low end
17   return 1;
18
19-~snip--
```

```cpp
20 ulong prev() // return zero if current paren is the first
21 {
22   // if ( k==1 ) return 0; // uncomment to make algorithm work for k==1
23   ulong j = 0;
24   // scan for first gap:
25   while ( x_[j]==j ) {++j; }
26   if ( j==k_ ) { last(); return 0; }
27   if ( x_[j]-x_[j-1] == 2 ) --x_[j]; // gap of length one
28   else
29     {
30       ulong i = --x_[j];
31       --j;
32       --1;
33       // j items to go, distribute as 1.1.1.11111
34       for ( ; 2*i>j; --i,--j) x_[j] = i;
35       for ( ; i; --i) x_[i] = 2*i;
36       x_[0] = 0;
37     }
38   
39   return 1;
40
41-~snip--
```

```cpp
42 const ulong * data() { return x_; }
43
44--snip--
```

[fxtbook draft of 2008-August-17]
The strings are set up on demand only:

```cpp
const char * string() // generate on demand
{
    for (ulong j=0; j<n_; ++j) str_[j] = ')';
    for (ulong j=0; j<k_; ++j) str_[x_[j]] = '('.
    return str_;}
}
```

The 477,638,700 paren words for \( n = 18 \) are generated at a rate of about 67 million objects per second. Section 13.30 on page 83 gives a bit-level algorithm for the generation of the paren words in colex order.

### 13.2 Gray code via restricted growth strings

The valid paren strings can be represented by sequences \( a_0, a_1, \ldots, a_n \) where \( a_0 = 0 \) and \( a_k \leq a_{k-1} + 1 \). These sequences are examples of so-called restricted growth strings (RGS). Some sources use the term restricted growth functions. The RGSs for \( n = 4 \) are shown in figure 13.2-A. A RGS can be incremented by incrementing the highest (rightmost in figure 13.2-A) digit \( a_j \) where \( a_j \leq a_{j-1} \) and setting \( a_i = 0 \) for all \( i > j \). A decrement is obtained by decrementing the highest digit \( a_j \neq 0 \) and setting \( a_i = a_i - 1 + 1 \) for all \( i > j \).

The RGSs for a given \( n \) can be generated as follows [FXT: class catalan in comb/catalan.h]:

```cpp
class catalan
// Catalan restricted growth strings (RGS)
// By default in near-perfect minimal-change order, i.e.
// exactly two symbols in paren string change with each step
{
    int *as_; // digits of the RGS: as[k] <= as[k-1] + 1
    int *d_; // direction with recursion (+1 or -1)
    ulong n_; // Number of digits (paren pairs)
    char *str_; // parent string
    bool xdr_; // whether to change direction in recursion (== minimal-change order)
    int dr0_; // dr0: starting direction in each recursive step:
    // dr0=+1 ==> start with as[]=0,0, ..., 0 == ()()()...()
    // dr0=-1 ==> start with as[]=0,1,2,...,n-1 == ((( ... )))

    public:

    catalan(ulong n, bool xdr=true, int dr0=+1)
    {
        n_ = n;
        as_ = new int[n_];
        d_ = new int[n_];
        str_ = new char[2*n_+1]; str_[2*n_] = 0;
        init(xdr, dr0);
    }
}
```

Figure 13.2-A: Length-4 restricted growth strings in lexicographic order (left), and the corresponding paren strings (middle) and delta sets (right).
Chapter 13: Parenthesis strings

| 1: [0 1 2 3 4] | [---] | (())()() | 11111... |
| 2: [0 1 2 3 3] | [---] | (())()() | 11111... ((XA))) |
| 3: [0 1 2 3 2] | [---] | (())()() | 11111... (XA)) |
| 4: [0 1 2 3 1] | [---] | (())()() | 11111... (X(A)) |
| 5: [0 1 2 3 0] | [---] | (())()() | 11111... (XA)) |
| 6: [0 1 2 2 0] | [--+] | (())()() | 11111... (X(A)) |
| 7: [0 1 2 2 1] | [---] | (())()() | 11111... (X(A)) |
| 8: [0 1 2 2 2] | [---] | (())()() | 11111... (X(A)) |
| 9: [0 1 2 2 3] | [---] | (())()() | 11111... (X(A)) |
| 10: [0 1 2 1 2] | [---] | (())()() | 11111... (X(A)) |
| 11: [0 1 2 1 1] | [---] | (())()() | 11111... (X(A)) |
| 12: [0 1 2 1 0] | [---] | (())()() | 11111... (X(A)) |
| 13: [0 1 2 0 0] | [---] | (())()() | 11111... (X(A)) |
| 14: [0 1 2 0 1] | [---] | (())()() | 11111... (X(A)) |
| 15: [0 1 2 0 2] | [---] | (())()() | 11111... (X(A)) |
| 16: [0 1 2 0 3] | [---] | (())()() | 11111... (X(A)) |
| 17: [0 1 1 1 0] | [---] | (())()() | 11111... (X(A)) |
| 18: [0 1 1 1 1] | [---] | (())()() | 11111... (X(A)) |
| 19: [0 1 1 1 2] | [---] | (())()() | 11111... (X(A)) |
| 20: [0 1 1 1 3] | [---] | (())()() | 11111... (X(A)) |
| 21: [0 1 1 1 4] | [---] | (())()() | 11111... (X(A)) |
| 22: [0 1 1 1 5] | [---] | (())()() | 11111... (X(A)) |
| 23: [0 1 1 1 6] | [---] | (())()() | 11111... (X(A)) |
| 24: [0 1 1 1 7] | [---] | (())()() | 11111... (X(A)) |
| 25: [0 1 1 1 8] | [---] | (())()() | 11111... (X(A)) |
| 26: [0 1 1 1 9] | [---] | (())()() | 11111... (X(A)) |
| 27: [0 1 1 1 10] | [---] | (())()() | 11111... (X(A)) |
| 28: [0 1 1 1 11] | [---] | (())()() | 11111... (X(A)) |
| 29: [0 1 1 1 12] | [---] | (())()() | 11111... (X(A)) |
| 30: [0 1 1 1 13] | [---] | (())()() | 11111... (X(A)) |
| 31: [0 1 1 1 14] | [---] | (())()() | 11111... (X(A)) |
| 32: [0 1 1 1 15] | [---] | (())()() | 11111... (X(A)) |
| 33: [0 1 1 1 16] | [---] | (())()() | 11111... (X(A)) |
| 34: [0 1 1 1 17] | [---] | (())()() | 11111... (X(A)) |
| 35: [0 1 1 1 18] | [---] | (())()() | 11111... (X(A)) |
| 36: [0 1 1 1 19] | [---] | (())()() | 11111... (X(A)) |
| 37: [0 1 1 1 20] | [---] | (())()() | 11111... (X(A)) |
| 38: [0 1 1 1 21] | [---] | (())()() | 11111... (X(A)) |
| 39: [0 1 1 1 22] | [---] | (())()() | 11111... (X(A)) |
| 40: [0 1 1 1 23] | [---] | (())()() | 11111... (X(A)) |
| 41: [0 1 1 1 24] | [---] | (())()() | 11111... (X(A)) |
| 42: [0 1 1 1 25] | [---] | (())()() | 11111... (X(A)) |

Figure 13.2-B: Minimal-change order for the paren strings of 5 pairs. From left to right: restricted growth strings, arrays of directions, paren strings, delta sets, and difference strings. If the changes are not adjacent then the distance of changed positions is given at the right. The order corresponds to dr0=-1.
Figure 13.2-C: Minimal-change order for the paren strings of 5 pairs. From left to right: restricted growth strings, arrays of directions, paren strings, delta sets, and difference strings. If the changes are not adjacent then the distance of changed positions is given at the right. The order corresponds to $dr_{0}\equiv+1$. 

<table>
<thead>
<tr>
<th>Pair</th>
<th>Restricted Growth Strings</th>
<th>Arrays of Directions</th>
<th>Paren Strings</th>
<th>Delta Sets</th>
<th>Difference Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[0 0 0 0 0]</td>
<td>++ + + + +</td>
<td>() () () () ()</td>
<td>1.1.1.1.1</td>
<td>() () () (AX)</td>
</tr>
<tr>
<td>2</td>
<td>[0 0 0 1 0]</td>
<td>++ + + + -</td>
<td>() () () ()</td>
<td>1.1.1.1.1</td>
<td>() () () (AX)</td>
</tr>
<tr>
<td>3</td>
<td>[0 0 1 0 0]</td>
<td>++ + + - +</td>
<td>() () () ()</td>
<td>1.11.11...</td>
<td>() () () AX</td>
</tr>
<tr>
<td>4</td>
<td>[0 0 1 0 1]</td>
<td>++ + + - -</td>
<td>() () () ()</td>
<td>1.11.11...</td>
<td>() () () AX</td>
</tr>
<tr>
<td>5</td>
<td>[0 0 1 1 0]</td>
<td>++ + - + -</td>
<td>() () () ()</td>
<td>1.11.11...</td>
<td>() () () AX</td>
</tr>
<tr>
<td>6</td>
<td>[0 0 1 1 1]</td>
<td>++ + - - -</td>
<td>() () () ()</td>
<td>1.11.11...</td>
<td>() () () AX</td>
</tr>
</tbody>
</table>

Note: The order corresponds to $dr_{0}\equiv+1$. 

[ftxtbook draft of 2008-August-17]
Chapter 13: Parenthesis strings

```cpp
void init(bool xdr, int dr0)
{
    dr0_ = ( (dr0>=0) ? +1 : -1 );
    xdr_ = xdr;
    ulong n = n_;
    if ( dr0_>0 ) for (ulong k=0; k<n; ++k) as_[k] = 0;
    else for (ulong k=0; k<n; ++k) as_[k] = k;
    for (ulong k=0; k<n; ++k) d_[k] = dr0_; 
}

bool next() { return next_rec(n_-1); }

const int *get() const { return as_; }

const char* str() { make_str(); return (const char*)str_; }
```

The minimal-change order is obtained by changing the ‘direction’ in the recursion, an essentially identical mechanism (for the generation of set partitions) is shown in chapter 15 on page 341. The function is given in [FXT: comb/catalan.cc]:

```cpp
bool catalan::next_rec(ulong k)
{
    if ( k<1 ) return false; // current is last
    int d = d_[k];
    int as = as_[k] + d;
    bool ovq = ( (d>0) ? (as>as_[k-1]+1) : (as<0) );
    if ( ovq ) // have to recurse
    {
        ulong ns1 = next_rec(k-1);
        if ( 0==ns1 ) return false;
        d = ( xdr_ ? -d : dr0_ );
        d_[k] = d;
        as = ( (d>0) ? 0 : as_[k-1]+1 );
    }
    as_[k] = as;
    return true;
}
```

The program [FXT: comb/catalan-demo.cc] demonstrates the usage:

```cpp
ulong n = 4;
bool xdr = true;
int dr0 = -1;
catalan C(n, xdr, dr0);
do
{
    // visit
} while ( C.next() );
```

About 69 million strings per second are generated. Figure 13.2-B shows the minimal-change order for \( n = 5 \) and \( dr0=-1 \), and figure 13.2-C for \( dr0=+1 \).

More minimal-change orders

The Gray code order shown in figure 13.2-D can be generated via a simple recursion:
13.2: Gray code via restricted growth strings

The initial call is `next_rec(0, 0);`. About 81 million strings per second are generated [FXT: `comb/paren-gray-rec-demo.cc`].

An loopless algorithm (that does not use RGS) given in [254] is implemented in [FXT: `class paren_gray` in `comb/paren-gray.h`]. The generated order for five paren pairs is shown in figure 13.2-F. About 54 million strings per second are generated [FXT: `comb/paren-gray-demo.cc`]. Still more algorithms for the parentheses strings in minimal-change order are given in [75], [263], and [284].

For even values of `n` it is possible to generate paren strings in `strong minimal-change order` where changes occur only in adjacent positions. Figure 13.2-F shows an example for four pairs of parens. The listing was generated with [FXT: `graph/graph-parengray-demo.cc` that uses directed graphs and the search algorithms described in chapter 19 on page 381].
Chapter 13: Parenthesis strings

Figure 13.2-E: Strings of 5 pairs of parenthesis in Gray code order as generated by a loopless algorithm.

Figure 13.2-F: A strong minimal-change order for the paren strings of 4 pairs.

13.3 The number of parenthesis strings: Catalan numbers

The number of valid combinations of $n$ parenthesis pairs is

$$C_n = \frac{\binom{2n}{n}}{n+1} = \frac{\binom{2n+1}{n}}{2n+1} = \frac{\binom{2n-2}{n-1}}{n} = \binom{2n}{n} - \binom{2n}{n-1}$$  \hfill (13.3-1)

as nicely explained in [134, p.343-346]. These are the Catalan numbers, sequence [A000108 of [245]:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$C_n$</th>
<th>$n$</th>
<th>$C_n$</th>
<th>$n$</th>
<th>$C_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>58786</td>
<td>21</td>
<td>24466267020</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>12</td>
<td>208012</td>
<td>22</td>
<td>91482563640</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>13</td>
<td>742900</td>
<td>23</td>
<td>343059613650</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>14</td>
<td>2674440</td>
<td>24</td>
<td>12899044174324</td>
</tr>
<tr>
<td>5</td>
<td>42</td>
<td>15</td>
<td>9694845</td>
<td>25</td>
<td>4861946401452</td>
</tr>
<tr>
<td>6</td>
<td>132</td>
<td>16</td>
<td>35357670</td>
<td>26</td>
<td>18367353072152</td>
</tr>
<tr>
<td>7</td>
<td>429</td>
<td>17</td>
<td>129644790</td>
<td>27</td>
<td>69533550916004</td>
</tr>
<tr>
<td>8</td>
<td>1430</td>
<td>18</td>
<td>477638700</td>
<td>28</td>
<td>26337451750360</td>
</tr>
<tr>
<td>9</td>
<td>4862</td>
<td>19</td>
<td>1767263190</td>
<td>29</td>
<td>1002242216651368</td>
</tr>
<tr>
<td>10</td>
<td>16796</td>
<td>20</td>
<td>6564120420</td>
<td>30</td>
<td>3814986502092304</td>
</tr>
</tbody>
</table>

[fxtbook draft of 2008-August-17]
13.4: Increment-\(i\) RGS and \(k\)-ary trees

The Catalan numbers are generated most easily with the relation

\[
C_{n+1} = \frac{2(2n+1)}{n+2} \, C_n
\]  

(13.3-2)

The generating function is

\[
C(x) = \frac{1 - \sqrt{1-4x}}{2x} = \sum_{n=0}^\infty C_n \, x^n = 1 + x + 2 \, x^2 + 5 \, x^3 + 14 \, x^4 + 42 \, x^5 + \ldots
\]  

(13.3-3)

The function \(C(x)\) satisfies the equation \([x \, C(x)] = x + [x \, C(x)]^2\) which is equivalent to the following convolution property for the Catalan numbers:

\[
C_n = \sum_{k=0}^{n-1} C_k \, C_{n-k-1}
\]  

(13.3-4)

### 13.4  Increment-\(i\) RGS and \(k\)-ary trees

#### 13.4.1 Generation in lexicographic order

We now allow an increment of \(i\) in the restricted growth strings (\(i = 1\) corresponds to the paren RGS of section 13.2). Figure 13.4-A shows the increment-2 restricted growths strings of length 4. The strings can be generated in lexicographic order via [FXT: class rgs_binomial in comb/rgs-binomial.h](https://www.fxtbook.com/src/rgs-binomial.h).

```cpp
class rgs_binomial
{ // Restricted growth strings (RGS) s[0,...,n-1] so that s[k] <= s[k-1]+i

    ulong *s_; // restricted growth string
    ulong n_; // Length of strings
    ulong i_; // s[k] <= s[k-1]+i

    [--snip--]

    ulong next() // Return index of first changed element in s[],
    ulong zero() // Return zero if current string is the last

    [--snip--]

    start:
```

Figure 13.4-A: The 55 increment-2 restricted growths strings of length 4.
--k;
if ( k==0 ) return 0;
ulong sk = s_[k] + 1;
ulong mp = s_[k-1] + i_;
if ( sk > mp ) // "carry"
{
    s_[k] = 0;
goto start;
}

s_[k] = sk;
return k;

The rate of generation is about 129 M/s for \( i = 1 \) (corresponding to paren strings), 143 M/s for \( i = 2 \), and 156 M/s with \( i = 3 \) [FXT: comb/rgs-binomial-demo.cc].

### 13.4.2 The number of increment-\( i \) RGS

The number \( C_{n,i} \) of length-\( n \) increment-\( i \) strings equals

\[
C_{n,i} = \frac{(i+1)n}{n+1} \quad (13.4-1)
\]

A recursion generalizing relation [13.3-2] is

\[
C_{n+1,i} = (i+1) \frac{\prod_{k=1}^{i} (i + 1 + k)}{\prod_{k=1}^{i} (i + k + 1)} C_{n,i} \quad (13.4-2)
\]

The sequences of numbers of length-\( n \) strings for \( i = 1, 2, 3, 4 \) start

\[
\begin{align*}
\text{n:} & & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{i=1:} & & 1 & 2 & 5 & 14 & 42 & 132 & 429 & 1430 & 4862 & 16796 & 58786 \\
\text{i=2:} & & 1 & 3 & 12 & 55 & 273 & 1428 & 7752 & 43263 & 246675 & 1430715 & 8148640 \\
\text{i=3:} & & 1 & 4 & 22 & 140 & 969 & 7084 & 53820 & 420732 & 3362260 & 27343888 & 225568798 \\
\text{i=4:} & & 1 & 5 & 25 & 285 & 2530 & 23751 & 231880 & 2330445 & 23950355 & 250543370 & 2655968130
\end{align*}
\]

These are respectively the entries [A000108], [A001764], [A002293], [A002294] of [245] where combinatorial interpretations are given. We note that for the generating function \( C_i(x) \) we have the following expression as a hypergeometric function (see section 35.2 on page 696):

\[
C_i(x) = \sum_{n=0}^{\infty} C_{n,i} x^n = F \left( \begin{array}{c} \frac{1}{i+1}, \frac{2}{i+1}, \frac{3}{i+1}, \ldots, \frac{(i+1)(i+1)}{i} \\ \frac{2/i, 3/i, \ldots, i/i, (i+1)/i} \end{array} \right| \frac{x^{(i+1)}}{i^i} \right) \quad (13.4-3a)
\]

Note that the last upper and second last lower parameter cancel. Now let \( f_i(x) := x C_i(x') \), then

\[
f_i(x) - f_i(x)^{i+1} = x \quad (13.4-4)
\]

That is, \( f_i(x) \) can be obtained as the series reversion of \( x^{i+1} - x \). We choose \( i = 2 \) for an example:

? t1=serreverse(x-x^3+0(x^-17))
? x + x^3 + 3*x^5 + 12*x^7 + 55*x^9 + 273*x^11 + 1428*x^13 + 7752*x^15 + O(x^-17)
? t2=hypergeom([1/3,2/3,3/3],[2/2,3/2,4/2],[x^3]/2^2)*0(x^-17)
1 + x + 3*x^2 + 12*x^3 + 55*x^4 + 273*x^5 + 1428*x^6 + 7752*x^7 + \ldots + O(x^-17)
? f=x*subst(t2,x,x^2);
\text{f is actually the series reversion of } x-x^3
\]

[fxtbook draft of 2008-August-17]
We further have the following convolution property, generalizing relation [13.3-4]

\[ C_{n,i} = \sum_{j_1+j_2+\ldots+j_{i+1}=n-1} C_{j_1,i} C_{j_2,i} C_{j_3,i} \cdots C_{j_{i+1},i} \]  

(13.4-5)

An explicit expression for the function \( C_i(x) \) is

\[ C_i(x) = \exp \left( \frac{1}{i+1} \sum_{n=1}^{\infty} \frac{(i+1)n}{n} x^n \right) \]  

(13.4-6)

The expression generalizes a relation given in [154, rel.6] (set \( i = 1 \) and take logarithm)

\[ \sum_{n=1}^{\infty} \frac{1}{n} \frac{2n}{n} x^n = 2 \log \left( \frac{1 - \sqrt{1 - 4x}}{2x} \right) \]  

(13.4-7)

13.4.3 Gray code for \( k \)-ary trees

The length-\( n \) increment-\( i \) RGS correspond to \( k \)-ary trees with \( n \) internal nodes and \( k = i+1 \). An loopless algorithm for the generation of a Gray code for \( k \)-ary tree with only homogeneous changes is given in [31]. The RGS used in the algorithm gives the positions (one-based) of the ones in the delta sets, see figure [13.4-B]. An implementation is [FXT: class tree_gray in comb/tree-gray.h]:

```cpp
class tree_gray
{
  // all arrays are one-based
  ulong *sq_; // sequence of bit positions (seq[]) elements \in \{1,2,\ldots,n\}
  ulong *dr_; // aux: direction (dir[])
  ulong *np_; // aux: next position (nextPos[])
  ulong *mx_; // aux: max position (max[])
  ulong n_; // n (internal) nodes
  ulong k_; // k-ary tree

  tree_gray(ulong n, ulong k)
  {
    n_ = n;
    k_ = k;
    // all arrays are one-based
    sq_ = new ulong[n+1];
    dr_ = new ulong[n+1];
    np_ = new ulong[n+2]; // one pad element right
    mx_ = new ulong[n+1]; // unchanged in next()
    first();
  }

  void first(ulong k=0)
  {
    if ( k ) k_ = k;
    for (ulong j=1, e=1; j<=n_; ++j, e+=k_) sq_[j] = mx_[j] = e;
    for (ulong j=0; j<=n_; ++j) dr_[j] = 1; // "right"
    for (ulong j=0; j<n_; ++j) np_[j] = j - 1;
  }

  ulong next()
  {
    ulong i = np_[n+1];
    if ( i==0 ) return 0; // current string is last
    if ( dr_[i]==1 ) // direction == "right"
      {
        if ( sq_[i] == mx_[i] ) sq_[i] = sq_[i-1] + 1;
        else sq_[i] += 1;
      }
  }
};
```

The computation of the successor is a variant of the method given in [44]:
<table>
<thead>
<tr>
<th>positions</th>
<th>direction</th>
<th>delta set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: [ 1 4 7 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..1..1..</td>
</tr>
<tr>
<td>2: [ 1 4 7 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>3: [ 1 4 7 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>4: [ 1 4 5 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>5: [ 1 4 5 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>6: [ 1 4 5 7 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>7: [ 1 4 5 6 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>8: [ 1 4 5 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>9: [ 1 4 6 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>10: [ 1 4 6 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>11: [ 1 4 6 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>12: [ 1 4 6 7 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>13: [ 1 4 6 5 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>14: [ 1 4 6 4 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>15: [ 1 4 6 3 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>16: [ 1 4 6 2 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>17: [ 1 4 6 1 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>18: [ 1 4 6 0 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>19: [ 1 4 5 7 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>20: [ 1 4 5 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>21: [ 1 4 5 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>22: [ 1 4 5 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>23: [ 1 4 6 7 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>24: [ 1 4 6 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>25: [ 1 4 6 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>26: [ 1 4 6 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>27: [ 1 4 7 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>28: [ 1 4 7 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>29: [ 1 4 7 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>30: [ 1 4 8 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>31: [ 1 4 9 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>32: [ 1 5 8 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>33: [ 1 5 9 A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>34: [ 1 6 A A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>35: [ 1 6 A 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>36: [ 1 6 A 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>37: [ 1 6 A 7 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>38: [ 1 6 A 6 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>39: [ 1 6 A 5 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>40: [ 1 6 A 4 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>41: [ 1 6 A 3 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>42: [ 1 6 A 2 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>43: [ 1 6 A 1 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>44: [ 1 6 A 0 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>45: [ 1 7 A A ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>46: [ 1 7 A 9 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>47: [ 1 7 A 8 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>48: [ 1 7 A 7 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>49: [ 1 7 A 6 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>50: [ 1 7 A 5 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>51: [ 1 7 A 4 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>52: [ 1 7 A 3 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>53: [ 1 7 A 2 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>54: [ 1 7 A 1 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
<tr>
<td>55: [ 1 7 A 0 ]</td>
<td>[ + + + + ]</td>
<td>1..1..11...</td>
</tr>
</tbody>
</table>

Figure 13.4-B: Gray code for 3-ary trees with 4 internal nodes with all changes being homogeneous. The left column shows the vectors of (one-based) positions, the symbol ‘A’ is used for the number 10.
if ( sq[i] == mx[i] - 1 )
    np[i+1] = np[i]; // can access element n+1
    np[i] = i - 1;
    dr[i] = -1UL; // "left"
else
    if ( sq[i] == sq[i-1] + 1 )
        sq[i] = mx[i];
        dr[i] = 1; // "right"
        np[i+1] = np[i]; // can access element n+1
        np[i] = i - 1;
    
    else sq[i] -- 1;
else sq[i] -- 1;
if ( i<n_ ) np[n_+1] = n_;
Chapter 14

Integer partitions

<table>
<thead>
<tr>
<th></th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4 + 2</td>
</tr>
<tr>
<td>3</td>
<td>3 + 3</td>
</tr>
<tr>
<td>4</td>
<td>2 + 3</td>
</tr>
<tr>
<td>5</td>
<td>1 + 5</td>
</tr>
<tr>
<td>6</td>
<td>0 + 6</td>
</tr>
</tbody>
</table>

An integer \( x \) is the sum of the positive integers less or equal to itself in various ways. The decompositions into sums of integers are called the integer partitions of the number \( x \). Figure 14.0-A shows all integer partitions of \( x = 6 \).

14.1 Solution of a generalized problem

We can solve a slightly more general problem and find all partitions of a number \( x \) with respect to a set \( V = \{v_0, v_1, \ldots, v_{n-1}\} \) where \( v_i > 0 \), that is all decompositions of the form \( x = \sum_{k=0}^{n-1} c_k \cdot v_k \) where \( c_i \geq 0 \). The integer partitions are the special case \( V = \{1, 2, 3, \ldots, n\} \).

To generate the partitions assign to the first bucket \( r_0 \) an integer multiple of the first element \( v_0 \): \( r_0 = c \cdot v_0 \). This has to be done for all \( c \geq 0 \) for which \( r_0 \leq x \). Now set \( c_0 = c \). If \( r_0 = x \) we already found a partition (consisting of \( c_0 \) only), else (if \( r_0 < x \)) solve the remaining problem where \( x' := x - c_0 \cdot v_0 \) and \( V' := \{v_1, v_2, \ldots, v_{n-1}\} \).

A C++ class for the generation of all partitions is [FXT: class partition_rec in comb/partition-rec.h]:

```cpp
class partition_rec {
public:
    ulong ct_; // Number of partitions found so far
    ulong n_; // Number of values
    ulong i_; // level in iterative search
    long *pv_; // values into which to partition
    ulong *pc_; // multipliers for values
    ulong pci_; // temporary for pc_[i_]
    long *r_; // rest
}
```
Chapter 14: Integer partitions

14 long ri_;  // temporary for r_[i_]
15 long x_;   // value to partition
16
17 public:
18 partition_rec(ulong x, ulong n=0, const ulong *pv=0)
19 {
20     if ( 0==n ) n = x;
21     n_ = n;
22     pv_ = new long[n_+1];
23     if ( pv ) for (ulong j=0; j<n_; ++j) pv_[j] = pv[j];
24     else for (ulong j=0; j<n_; ++j) pv_[j] = j + 1;
25     pc_ = new ulong[n_+1];
26     r_ = new long[n_+1];
27     init(x);
28 }
29
30 void init(ulong x)
31 {
32     x_ = x;
33     ct_ = 0;
34     for (ulong k=0; k<n_; ++k) pc_[k] = 0;
35     for (ulong k=0; k<n_; ++k) r_[k] = 0;
36     r_[n_-1] = x_;
37     r_[n_] = x_;
38     i_ = n_ - 1;
39     pci_ = 0;
40     ri_ = x_; 
41 }
42 ~partition_rec()
43 {
44     delete [] pv_; 
45     delete [] pc_; 
46     delete [] r_; 
47 }
48
49 ulong next(); // generate next partition 
50 ulong next_func(ulong i); // aux
51 [--snip--]
52
53 The routine to compute the next partition is given in [FXT: comb/partition-rec.cc]:
54
55 ulong
56 partition_rec::next()
57 {
58     if ( i_>=n_ ) return n_;
59     r_[i_] = ri_;  
60     pc_[i_] = pci_;  
61     i_ = next_func(i_);  
62     for (ulong j=0; j<i_; ++j) pc_[j] = r_[j] = 0; 
63     ++i_; 
64     ri_ = r_[i_] - pv_[i_]; 
65     pci_ = pc_[i_] + 1; 
66     return i_ - 1; // >=0 
67 }
68
69 ulong 
70 partition_rec::next_func(ulong i)
71 {
72     start:
73         if ( 0!=i )
74         {
75             while ( r_[i]>0 )
76             {
77                 pc_[i-1] = 0; 
78                 r_[i-1] = r_[i]; 
79                 --i; goto start; // iteration
80             }
81         }
82         else // iteration end
83         {
84             ;
85         }}
14.2: Iterative algorithm

An iterative implementation for the generation of the integer partitions is given in [FXT: class partition in comb/partition.h]:

```cpp
class partition
{
  public:
    ulong *c_; // partition: c[1]* 1 + c[2]* 2 + ... + c[n]* n == n
    ulong *s_; // cumulative sums: s[j+1] = c[1]* 1 + c[2]* 2 + ... + c[j]* j
    ulong n_; // partitions of n

  public:
    partition(ulong n)
    {
      n_ = n;
      c_ = new ulong[n+1];
      s_ = new ulong[n+1];
      s_[0] = 0; // unused
      c_[0] = 0; // unused
      first();
    }

    ~partition()
    {
      delete [] c_;
      delete [] s_;
    }

    void first()
    {
      c_[1] = n_;
      for (ulong i=2; i<=n_; i++) { c_[i] = 0; }
      s_[1] = 0;
      for (ulong i=2; i<=n_; i++) { s_[i] = n_; }
    }

    void last()
    {
      for (ulong i=1; i<n_; i++) { c_[i] = 0; }
      c_[n_] = 1;
      for (ulong i=1; i<n_; i++) { s_[i] = 0; }
      // s_[n+1] = n_; // unused (and out of bounds)
    }
};
```

The routines can easily be adapted to the generation of partitions satisfying certain restrictions, for example, partitions into unequal parts (that is, \( c_i \leq 1 \)). The listing shown in figure 14.0-A can be generated with [FXT: comb/partition-rec-demo.cc]. The 190,569,292 partitions of 100 are generated at a rate of about 18 M/s.
To obtain the next partition, find the smallest index \( i \geq 2 \) so that \([c_1,c_2,\ldots,c_{i-1},c_i]\) can be replaced by \([z,0,\ldots,0,c_i+1]\) where \( z \geq 0 \). The index \( i \) is determined using cumulative sums. The partitions are generated in the same order as shown in figure 14.0-A. The algorithm was given (2006) by Torsten Finke [priv.comm.].

```cpp
bool next()
{
  if ( c_[n_]!=0 ) return false; // last == 1* n (c[n]=1)
  // Find first coefficient c[i], i>=2 that can be increased:
  ulong i = 2;
  while ( s_[i]<i ) ++i;
  ++c_[i];
  s_[i] -= i;
  ulong z = s_[i];
  // Now set c[1], c[2], ..., c[i-1] to the first partition
  // of z into i-1 parts, i.e. set to z, 0, 0, ..., 0:
  while ( --i > 1 )
  {
    s_[i] = z;
    c_[i] = 0;
  }
  c_[1] = z; // z* 1 == z
  // s_[1] unused
  return true;
}
```

The preceding partition can be computed as follows:

```cpp
bool prev()
{
  if ( c_[1]==n_ ) return false; // first == n* 1 (c[1]=n)
  // Find first nonzero coefficient c[i] where i>=2:
  ulong i = 2;
  while ( c_[i]==0 ) ++i;
  --c_[i];
  s_[i] += i;
  ulong z = s_[i];
  // Now set c[1], c[2], ..., c[i-1] to the last partition
  // of z into i-1 parts:
  while ( --i > 1 )
  {
    ulong q = (z>=i ? z/i : 0); // == z/i;
    c_[i] = q;
    s_[i+1] = z;
    z -= q*i;
  }
  c_[1] = z;
  s_[2] = z;
  // s_[1] unused
  return true;
}
```

Note that divisions which result in \( q = 0 \) are avoided, leading to a small speedup. The program [FXT: comb/partition-demo.cc] demonstrates the usage of the class. More than 140 million partitions per second are generated, about 66 million when going backward.

### 14.3 Partitions into \( m \) parts

An algorithm for the generation of all partitions of \( n \) into \( m \) parts is given in [104, p.106] (method ascribed to Hindenburg):
14.3: Partitions into $m$ parts

The initial partition contains $m-1$ units and the element $n-m+1$. To obtain a new partition from a given one, pass over the elements of the latter from right to left, stopping at the first element $f$ which is less, by at least two units, than the final element [...]. Without altering any element at the left of $f$, write $f+1$ in place of $f$ and every element to the right of $f$ with the exception of the final element, in whose place is written the number which when added to all the other new elements gives the sum $n$. The process to obtain partitions stops when we reach one in which no part is less than the final part by at least two units.

Figure 14.3-A shows the partitions of 19 into 11 parts. The data was generated with the program [FXT: comb/mpartition-demo.cc]. The implementation used is [FXT: class mpartition in comb/mpartition.h]:

```cpp
class mpartition
// Integer partitions of n into m parts
{
public:
ulong *x_; // partition: x[1]+x[2]+...+x[m] = n
ulong *s_; // aux: cumulative sums of x[] (s[0]=0)
ulong n_; // integer partitions of n (must have n>0)
ulong m_; // ... into m parts (must have 0<m<=n)
public:
mpartition(ulong n, ulong m)
: n_(n), m_(m)
{
x_ = new ulong [m_+1];
s_ = new ulong [m_+1];
init();
}
~mpartition()
{
delete [] x_;
delete [] s_;}
const ulong *data() const { return x_+1; }

void init()
{
x_[0] = 0;
for (ulong k=1; k<m_; ++k) x_[k] = 1;
x_[m_] = n_ - m_ + 1;
ulong s = 0;
for (ulong k=0; k<m_; ++k) { s+=x_[k]; s_ [k]=s; }
}

bool next()
{
ulong u = x_[m_]; // last element
ulong k = m_;
while ( --k ) { if ( x_[k]+2<=u ) break; }
if ( k==0 ) return false;
ulong f = x_[k] + 1;
}
}
```

The successor is computed as follows:
Chapter 14: Integer partitions

The auxiliary array of cumulative sums allows the recalculation of the final element without rescanning more than the elements just changed. About 105 million partitions per second can be generated.

A (complicated) construction for a Gray code for integer partitions is given in [225].

14.4 The number of integer partitions

14.4.1 Unrestricted partitions and partitions into \( m \) parts

The number of integer partitions of \( n \) is sequence A000041 of [245], the values for 1 \( \leq x \leq 50 \) are shown in figure 14.4-A. If we denote the number of partitions of \( n \) into exactly \( m \) parts by \( P(n,m) \) then

\[
P(n,m) = P(n-1,m-1) + P(n-m,m)
\]

(14.4-1)
were we set $P(0,0) = 1$. We obviously have $P_n = \sum_{m=1}^{n} P(n,m)$. Figure 14.4-B shows $P(n,m)$ for $n \leq 16$, it was created with the program [FXT: comb/num-partitions-demo.cc]. We note that the number of partitions into $m$ parts equals the number of partitions with maximal part equal to $m$. This can easily be seen by drawing a diagram and its transposed as follows (for the partition $5 + 2 + 2 + 1$ of 10):

Thereby any partition with maximal part $m$ (here 5) corresponds to a partition into exactly $m$ parts. The generating function for the partitions into exactly $m$ parts is

$$\sum_{n=1}^{\infty} P(n,m) x^n = \frac{x^m}{\prod_{k=1}^{m} (1 - x^k)}$$ (14.4-2)

For example, the row for $m = 3$ in figure 14.4-B corresponds to the power series

? m=3; (x^m/prod(k=1,m,1-x^k)+O(x^17))

x^3 + x^4 + 2*x^5 + 3*x^6 + 4*x^7 + 5*x^8 + 7*x^9 + 8*x^10 + \ 
10*x^11 + 12*x^12 + 14*x^13 + 16*x^14 + 19*x^15 + 21*x^16 + O(x^17)

The generating function for the number $P_n$ of integer partitions of $n$ can be given as [116, p.357]

$$\sum_{n=0}^{\infty} P_n x^n = \frac{1}{\prod_{n=1}^{\infty} (1 - x^n)} =: \frac{1}{\eta(x)}$$ (14.4-3)

Summing over $m$ in relation 14.4-2 we find that

$$\frac{1}{\eta(x)} = \sum_{n=0}^{\infty} \frac{x^n}{\prod_{k=1}^{n} (1 - x^k)}$$ (14.4-4)

This relation can also be obtained as the special case $a_n = x^n$ (and $N \to \infty$) of [204, p.83]

$$\frac{1}{\prod_{n=1}^{N} (1 - a_n)} = \sum_{n=0}^{N} a_n \prod_{k=1}^{n} (1 - a_k)$$ (14.4-5)

We also have

$$\frac{1}{\eta(x)} = \sum_{n=0}^{\infty} \frac{x^n^2}{(\prod_{k=1}^{n} (1 - x^k))^2}$$ (14.4-6)

The expression can be obtained by observing that a partition can be decomposed into a square and two partitions whose maximal part does not exceed the length of the side of the square:

The relation is also the special case $q = x$ of an identity given in [202]:

$$\frac{1}{\prod_{n=0}^{\infty} (1 - x q^n)} = \sum_{n=0}^{\infty} \frac{x^n q^{n(n-1)}}{\prod_{k=0}^{n-1} (1 - q q^n) \prod_{k=0}^{n-1} (1 - x q^n)}$$ (14.4-7)

The *pentagonal number theorem*, due to Euler, see [35], is:

$$\eta(x) = \sum_{n=-\infty}^{+\infty} (-1)^n \left(x^n(3n-1)/2\right) = \sum_{n=0}^{\infty} (-1)^n x^{n(3n-1)/2} (1 + x^n)$$ (14.4-8)

[fxtbook draft of 2008-August-17]
Further expressions for $\eta$ are

$$\eta(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{n(n+1)/2}}{\prod_{k=1}^{n+1} (1 - x^k)} = \sum_{n=0}^{\infty} \frac{x^{2n^2+n} (1 - 2x^{2n+1})}{\prod_{k=1}^{2n+1} (1 - x^k)}$$  \hspace{1cm} (14.4-9)$$

Write $\eta(x) = \prod_{j=0}^{\infty} J(x^{2j+1})$ where $J$ is defined by relation 36.1-2a on page 723. Then a division-free expression for $1/\eta$ is obtained via relation 36.1-11d on page 725:

$$\frac{1}{\eta(x)} = \prod_{k=0}^{\infty} \prod_{j=0}^{\infty} \left(1 + x^{(2j+1)2^k}\right)^{k+1}$$  \hspace{1cm} (14.4-10)$$

### 14.4.2 Partitions into distinct parts

The generating function for the number $D_n$ of partitions of $n$ into distinct parts is

$$\eta_+(x) := \prod_{n=1}^{\infty} (1 + x^n) = \sum_{n=0}^{\infty} D_n x^n$$  \hspace{1cm} (14.4-11)$$

The number of partitions into distinct parts equals the number of partitions into odd parts:

$$\eta_+(x) = \frac{\eta(x^2)}{\eta(x)} = \prod_{k=1}^{\infty} \frac{1}{1 - x^{2k-1}}$$  \hspace{1cm} (14.4-12)$$

The sequence of coefficients $D_n$ is entry A000009 of [243]:

$$[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18, 22, 27, 32, 38, 46, 54, 64, 76, 89, 104, 122, 142, 165, 192, 222, 256, \ldots]$$

The generating function for $D(n, m)$, the number of partitions of $n$ into exactly $m$ distinct parts is (see [15, p.559])

$$\sum_{n=0}^{\infty} D(n, m) x^n = \frac{x^{m(m+1)/2}}{\prod_{k=1}^{m} (1 - x^k)}$$  \hspace{1cm} (14.4-13)$$

Summing over $m$ we obtain

$$\eta_+(x) = \sum_{n=0}^{\infty} \frac{x^{n(n+1)/2}}{\prod_{k=1}^{n+1} (1 - x^k)}$$  \hspace{1cm} (14.4-14)$$

The relation can also be obtained by setting $q = x$ in

$$\prod_{n=0}^{\infty} (1 + x^n q^n) = \sum_{n=0}^{\infty} \frac{q^{n(n-1)/2}}{\prod_{k=1}^{n} (1 - q^k)} x^n$$  \hspace{1cm} (14.4-15)$$

An alternative expression is

$$\eta_+(x) = \sum_{n=0}^{\infty} \frac{x^{2n^2+n}}{\prod_{k=1}^{2n+1} (1 - x^k)}$$  \hspace{1cm} (14.4-16)$$

A generating function for the number of partitions into distinct odd parts is

$$\prod_{n=0}^{\infty} (1 + x^{2n+1}) = \sum_{n=0}^{\infty} \frac{x^{n^2}}{\prod_{k=1}^{n} (1 - x^{2k})} = \frac{1}{\eta_+(-x)}$$  \hspace{1cm} (14.4-17)$$

The first equality is given in [204] p.27. The sequence of such numbers is entry A000700 of [245].
14.4: The number of integer partitions

1, 1, 0, 1, 1, 1, 2, 2, 2, 3, 3, 4, 5, 5, 6, 7, 8, 8, 9, 11, 12, 12, 14, 16, 17, 18, 20, 23, 25, 26, 29, 33, 35, 37, 41, 46, 49, 52, 57, ...

For the number of partitions into distinct even parts we have:

$$\prod_{n=1}^{\infty} (1 + x^{2n}) = \sum_{n=0}^{\infty} \frac{x^{n^2+n}}{\prod_{k=1}^{n} (1 - x^{2k})} = \eta_+(x^2) = \eta_+(-x) \eta_+(+x) \quad (14.4-18)$$

The sequence of such numbers is entry A035457 of [245]:

1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 5, 0, 6, 0, 8, 0, 10, 0, 12, 0, 15, 0, 18, 0, 22, 0, 27, 0, 32, 0, 38, 0, 46, 0, 54, 0, 64, 0, 76, 0, 89, 0, 104, 0, ...

The number of partitions where at most \( r \) elements of each partition are equal has the generating function

$$\prod_{n=1}^{\infty} (1 + x^{n^2+n} + x^{2n^2} + \ldots + x^{rn}) = \frac{\eta(x^{r+1})}{\eta(x)} = \frac{1}{\prod_{k \neq 0 \mod r+1} (1 - x^k)} \quad (14.4-19a)$$

The second equality tells us that the number of such partitions equals the number of partitions into parts not divisible by \( r+1 \), equivalently, partitions into \( m \) parts where \( m \) is not divisible by \( r+1 \).

A generating function for the number of partitions into an even number of distinct parts is

$$\eta_+(x) + \eta(x) = \sum_{n=0}^{\infty} \frac{x^{2n^2+n}}{\prod_{k=1}^{n} (1 - x^k)} \quad (14.4-20)$$

For the partitions into an odd number of distinct parts we have the generating function

$$\frac{\eta_+(x) - \eta(x)}{2} = \sum_{n=0}^{\infty} \frac{x^{2n^2+3n+1}}{\prod_{k=1}^{n+1} (1 - x^k)} \quad (14.4-21)$$

The sequences are entries A067661 and A067659 of [245], respectively:

1, , 0, 0, 1, 1, 2, 3, 4, 6, 9, 12, 16, 21, 28, 36, 47, 60, 76, 96, 120, 150, ...

1, 0, 0, 0, 1, 0, 0, 0, 2, 1, 0, 0, 3, 1, 0, 0, 5, 2, 2, 0, 7, 3, 2, 0, 11, 6, 4, 3, 15, 8, 6, 3, 22, 13, 11, 6, 34, 18, 15, 9, 46, 27, 3, 2, 0, 11, 6, 4, 3, ...

14.4.3 Partitions into square-free parts *

We give relations for the ordinary generating functions for partitions into square-free parts. The Möbius function \( \mu \) is defined in section 35.1.2 on page 689. We give the sequence of power series coefficients at the end of each relation.

Partitions into square-free parts (entry A073576 of [245]):

$$\prod_{n=1}^{\infty} \frac{1}{1 - \mu(n)^2 x^n} = \prod_{n=1}^{\infty} \eta(x^n)^{-\mu(n)} \quad (14.4-22)$$

1, 1, 2, 3, 4, 6, 9, 12, 16, 21, 28, 36, 47, 60, 76, 96, 120, 150, ...

Partitions into parts that are not square-free, note the lower index at the right hand side, (entry A114374):

$$\prod_{n=1}^{\infty} \frac{1}{1 - (1 - \mu(n))^2 x^n} = \prod_{n=2}^{\infty} \eta(x^n)^{+\mu(n)} \quad (14.4-23)$$

1, 0, 0, 0, 1, 0, 0, 0, 2, 1, 0, 0, 3, 1, 0, 0, 5, 2, 2, 0, 7, 3, 2, 0, 11, 6, 4, 3, 15,...
Partitions into distinct square-free parts (entry A087188):

\[
\prod_{n=1}^{\infty} 1 + \mu(n)^2 x^n = \prod_{n=1}^{\infty} \eta_+ \left( x^{n^2} \right)^{+\mu(n)} \tag{14.4-24}
\]

\[
1, 1, 1, 2, 1, 2, 3, 3, 4, 4, 5, 6, 6, 8, 9, 10, 13, 14, 16, 18, 20, \ldots
\]

Partitions into odd square-free parts, also partitions into parts \( m \) such that \( 2m \) is square-free:

\[
\prod_{n=1}^{\infty} \frac{1}{1 - \mu(2n - 1)^2 x^{2n-1}} = \prod_{n=1}^{\infty} \frac{1}{1 - \mu(2n)^2 x^n} = \prod_{n=1}^{\infty} \eta_+ \left( x^{(2n-1)^2} \right)^{-\mu(2n-1)} \tag{14.4-25a}
\]

\[
\prod_{n=1}^{\infty} \left[ \frac{\eta_+ \left( x^{(2n-1)^2} \right)}{\eta_+ \left( x^{2(2n-1)^2} \right)} \right]^{-\mu(2n-1)} = \prod_{n=1}^{\infty} \eta_+ \left( x^{(2n-1)^2} \right)^{+\mu(2n-1)} \tag{14.4-25b}
\]

\[
1, 1, 1, 2, 2, 3, 4, 5, 6, 7, 9, 11, 13, 16, 19, 23, 27, 32, 38, 44, \ldots
\]

Partitions into distinct odd square-free parts, also partitions into distinct parts \( m \) such that \( 2m \) is square-free:

\[
\prod_{n=1}^{\infty} 1 + \mu(2n - 1)^2 x^{2n-1} = \prod_{n=1}^{\infty} 1 + \mu(2n)^2 x^n = \prod_{n=1}^{\infty} \eta_+ \left( x^{(2n-1)^2} \right)^{+\mu(2n-1)} \tag{14.4-26a}
\]

\[
\prod_{n=1}^{\infty} \left[ \frac{\eta_+ \left( x^{(2n-1)^2} \right)}{\eta_+ \left( x^{2(2n-1)^2} \right)} \right]^{+\mu(2n-1)} = \prod_{n=1}^{\infty} \left[ \frac{\eta \left( x^{(2n-1)^2} \right) \eta \left( x^{4(2n-1)^2} \right)}{\eta \left( x^{2(2n-1)^2} \right)^2} \right]^{-\mu(2n-1)} \tag{14.4-26b}
\]

\[
1, 1, 0, 1, 1, 1, 1, 2, 1, 1, 2, 2, 2, 3, 4, 3, 4, 5, 5, 6, 6, 7, \ldots
\]

Partitions into square-free parts \( m \neq 0 \) mod \( p \) where \( p \) is prime:

\[
\prod_{n=1}^{\infty} \frac{1}{1 - \mu(p n)^2 x^n} = \prod_{n=1}^{p-1} \prod_{r=1}^{p-1} \left[ \frac{\eta \left( x^{(p n-r)^2} \right)}{\eta \left( x^{p(p n-r)^2} \right)} \right]^{-\mu(p n-r)} \tag{14.4-27}
\]

Partitions into distinct square-free parts \( m \neq 0 \) mod \( p \) where \( p \) is prime:

\[
\prod_{n=1}^{\infty} 1 + \mu(p n)^2 x^n = \prod_{n=1}^{p-1} \prod_{r=1}^{p-1} \left[ \frac{\eta_+ \left( x^{(p n-r)^2} \right)}{\eta_+ \left( x^{p(p n-r)^2} \right)} \right]^{+\mu(p n-r)} \tag{14.4-28}
\]
Chapter 15
Set partitions

For a set of \( n \) elements, say \( S_n := \{1, 2, \ldots, n\} \), a set partition is a set \( P = \{s_1, s_2, \ldots, s_k\} \) of non-empty subsets \( s_i \) of \( S_n \) whose intersection is empty and whose union equals \( S \).

For example, there are 5 set partitions of the set \( S_3 = \{1, 2, 3\} \):

1: \( \{1, 2, 3\} \)
2: \( \{1, 2\}, \{3\} \)
3: \( \{1, 3\}, \{2\} \)
4: \( \{1\}, \{2, 3\} \)
5: \( \{1\}, \{2\}, \{3\} \)

The following sets are not set partitions of \( S_3 \):

\( \{ \{1, 2, 3\}, \{1\} \} \) // intersection not empty
\( \{ \{1\}, \{3\} \} \) // union does not contain 2

As the order of elements in a set does not matter we sort them in ascending order. For a set of sets we order the sets in ascending order of the first elements.

The number of set partitions of the \( n \)-set is the Bell number \( B_n \), see section 15.2 on page 345.

15.1 Recursive generation

We write \( Z_n \) for the list of all set partitions of the \( n \)-element set \( S_n \). In order to generate \( Z_n \) we observe that with a complete list \( Z_{n-1} \) of partitions of the set \( S_{n-1} \) we can generate the elements of \( Z_n \) in the following way: For each element (set partition) \( P \in Z_{n-1} \), create set partitions of \( S_n \) by appending the element \( n \) to the first, second, \ldots, last subset and one more by appending the set \( n \) as the last subset.

For example, the partition \( \{\{1, 2\}, \{3, 4\}\} \in Z_4 \) leads to 3 partitions of \( S_5 \):

\[ P = \{\{1, 2\}, \{3, 4\}\} \]
\[ \rightarrow \{\{1, 2, 5\}, \{3, 4\}\} \]
\[ \rightarrow \{\{1, 2\}, \{3, 4, 5\}\} \]
\[ \rightarrow \{\{1, 2\}, \{3, 4\}, \{5\}\} \]

Now we start with the only partition of the one-element set, \( \{\{1\}\} \), and apply the described step \( n-1 \) times. The construction is shown in the left column of figure 15.1-A the right column shows all set partitions for \( n = 5 \).

A modified version of the recursive construction generates the set partitions in a minimal-change order. We can generate the ‘incremented’ partitions in two ways, forward (left to right)

\[ P = \{\{1, 2\}, \{3, 4\}\} \]
\[ \rightarrow \{\{1, 2, 5\}, \{3, 4\}\} \]
\[ \rightarrow \{\{1, 2\}, \{3, 4, 5\}\} \]
\[ \rightarrow \{\{1, 2\}, \{3, 4\}, \{5\}\} \]
Chapter 15: Set partitions

---

setpart(4) ==
1: {1, 2, 3, 4}
2: {1, 2, 3}, {4}
3: {1, 2, 4}, {3}
4: {1, 2}, {3, 4}
5: {1, 2}, {3}, {4}
6: {1, 3, 4}, {2}
7: {1, 3}, {2, 4}
8: {1, 3}, {2}, {4}
9: {1, 4}, {2, 3}
10: {1}, {2, 3, 4}
11: {1}, {2, 3}, {4}
12: {1}, {2, 4}, {3}
13: {1}, {2, 4}, {3}
14: {1}, {2}, {3}, {4}
15: {1}, {2}, {3}, {4}

---

Figure 15.1-A: Recursive construction of the set partitions of the 4-element set $S_4 = \{1, 2, 3, 4\}$ (left), and the resulting list of all set partitions of 4 elements (right).

---

Figure 15.1-B: Construction of a Gray code for set partitions as an interleaving process.
15.1: Recursive generation

<table>
<thead>
<tr>
<th>Recursive generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: {1, 2, 3, 4}</td>
</tr>
<tr>
<td>2: {1, 2, 3}</td>
</tr>
<tr>
<td>3: {1, 2}</td>
</tr>
<tr>
<td>4: {1, 2}</td>
</tr>
<tr>
<td>5: {1, 2}</td>
</tr>
<tr>
<td>6: {1}</td>
</tr>
<tr>
<td>7: {1}</td>
</tr>
<tr>
<td>8: {1}</td>
</tr>
<tr>
<td>9: {1}</td>
</tr>
<tr>
<td>10: {1}</td>
</tr>
<tr>
<td>11: {1}</td>
</tr>
<tr>
<td>12: {1}</td>
</tr>
<tr>
<td>13: {1}</td>
</tr>
<tr>
<td>14: {1}</td>
</tr>
<tr>
<td>15: {1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Minimal-change orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>2: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>3: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>4: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>5: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>6: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>7: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>8: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>9: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>10: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>11: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>12: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>13: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>14: {1}, {2}, {3}, {4}</td>
</tr>
<tr>
<td>15: {1}, {2}, {3}, {4}</td>
</tr>
</tbody>
</table>

Figure 15.1-C: Set partitions of \(S_4 = \{1, 2, 3, 4\}\) in two different minimal-change orders.

The resulting process of interleaving elements is shown in figure [15.1-B] The method is similar to Trotter’s construction for permutations, see figure [10.7-B] on page [255]. If we change the direction with every subset that is to be incremented, we obtain the minimal-change order shown in figure [15.1-C] for \(n = 4\). The left column is obtained by starting with the forward direction in each step of the recursion, the right by starting with the backward direction. The lists can be computed with [FXT: comb/setpart-demo.cc].

The C++ class [FXT: class setpart in comb/setpart.h] stores the list in an array of signed characters. The stored value is negated if the element is the last in the subset. The work involved with the creation of \(Z_n\) is proportional to \(\sum_{k=1}^{n} k B_k\) where \(B_k\) is the \(k\)-th Bell number.

The parameter xdr of the constructor determines the order in which the partitions are being created:

```cpp
class setpart
// Set partitions of the set {1,2,3,...,n}
// By default in minimal-change order
{
      public:
        ulong n_; // Number of elements of set (set = \{1,2,3,...,n\})
        int *p_; // p[] contains set partitions of length 1,2,3,...,n
        int **pp_; // pp[k] points to start of set partition k
        int *ns_; // ns[k] Number of Sets in set partition k
        int *as_; // element k attached At Set (0<=as[k]<=k) of set(k-1)
        int *d_; // direction with recursion (+1 or -1)
        int *x_; // current set partition (==pp[n])
        bool xdr_; // whether to change direction in recursion (== minimal-change order)
        int dr0_; // dr0: starting direction in each recursive step:
                  // dr0=+1 ==> start with partition \{1,2,3,...,n\}
                  // dr0=-1 ==> start with partition \{1\},\{2\},\{3\},...,\{n\}

      public:
        setpart(ulong n, bool xdr=true, int dr0=+1)
        
        {
          n_ = n;
          ulong np = (n_*(n_+1))/2; // == \sum_{k=1}^{n}\{k\}
          p_ = new int[np];
          pp_ = new int *[n+1];
          pp_[0] = 0; // unused
          pp_[1] = p_;
          for (ulong k=2; k<=n_; ++k) pp_[k] = pp_[k-1] + (k-1);
          ns_ = new int[n+1];
          as_ = new int[n+1];
```
The actual work is done by the methods \texttt{next\_rec()} and \texttt{cp\_append()} [FXT: \texttt{comb/setpart.cc}):

```cpp
1 int setpart::cp_append(const int *src, int *dst, ulong k, ulong a)
2 // Copy partition in src[0,...,k-2] to dst[0,...,k-1]
3 // Append element k at subset a (a>=0)
4 // Return number of sets in created partition.
5 {
6  ulong ct = 0;
7  for (ulong j=0; j<k-1; ++j)
8  {
9     int e = src[j];
10     if (e > 0) dst[j] = e;
11     else
12     {
13       if (a==ct) {dst[j]=-e;++dst; dst[j]=-k;}
14       else dst[j] = e;
15       ++ct;
16     }
17   }
18   if (a>ct) {dst[k-1] = -k;++ct;}
19   return ct;
20 }
1 int setpart::next\_rec(ulong k)
2 // Update partition in level k from partition in level k-1 (k<=n)
3 // Return number of sets in created partition
4 {
5  if (k<=1) return 0; // current is last
6  int d = d_[k];
7  int as = as_[k] + d;
8  bool ovq = ( (d>0) ? (as>ns_\[k-1\]) : (as<0) );
9  if (ovq) // have to recurse
10  {
11    ulong ns1 = next\_rec(k-1);
12    if (0==ns1) return 0;
13    d = ( xdr_ ? ~d : dr0_ );
14    d_[k] = d;
15    as = ( (d>0) ? 0 : ns_\[k-1\] );
16  }
17  as_[k] = as;
18  ulong ns = cp\_append(pp_\[k-1\], pp_\[k\], k, as);
19  ns_\[k\] = ns;
20  return ns;
21 }
```

The partitions are stored as an array of signed integers that are greater than zero and smaller or equal to \( n \). A negative value indicates that it is the last of the subset. For example, the set partitions of \( S_4 \)
15.2: The number of set partitions: Stirling set numbers and Bell numbers

The number of set partitions: Stirling set numbers and Bell numbers

1: as[ 0 0 0 0 ] x[ +1 +2 +3 -4 ] {1, 2, 3, 4}
2: as[ 0 0 0 1 ] x[ +1 +2 -3 -4 ] {1, 2, 3}, {4}
3: as[ 0 0 1 0 ] x[ +1 +2 -4 -3 ] {1, 2}, {3, 4}
4: as[ 0 0 1 1 ] x[ +1 -2 +3 -4 ] {1, 2}, {3}, {4}
5: as[ 0 1 0 0 ] x[ +1 +3 -4 -2 ] {1, 3, 4}, {2}
6: as[ 0 1 0 1 ] x[ +1 -3 +2 -4 ] {1, 3}, {2, 4}
7: as[ 0 1 0 2 ] x[ +1 -3 -2 -4 ] {1, 3}, {2}, {4}
8: as[ 0 1 1 0 ] x[ +1 -4 +2 -3 ] {1, 4}, {2, 3}
9: as[ 0 1 1 1 ] x[ +1 -4 -2 -3 ] {1, 4}, {2}, {3}
10: as[ 0 1 1 2 ] x[ +1 -4 -2 -3 ] {1, 4}, {2}, {3}
11: as[ 0 1 2 0 ] x[ +1 -4 -2 -3 ] {1, 4}, {2}, {3}
12: as[ 0 1 2 1 ] x[ +1 -4 -2 -3 ] {1, 4}, {2}, {3}
13: as[ 0 1 2 2 ] x[ +1 -4 -2 -3 ] {1, 4}, {2}, {3}
14: as[ 0 1 2 3 ] x[ +1 -4 -2 -3 ] {1, 4}, {2}, {3}
15: as[ 0 1 2 4 ] x[ +1 -4 -2 -3 ] {1, 4}, {2}, {3}

Figure 15.1-D: The partitions of the set \( S_4 = \{1, 2, 3, 4\} \) together with the internal representations: the 'signed value' array \( x[] \) and the 'attachment' array \( as[] \).

together with their 'signed value' representations are shown in figure [15.1-D] The array \( as[] \) contains a restricted growth string (RGS) with the condition \( a_j \leq 1 + \max_{i<j}(a_i) \). A different sort of RGS is described in section [13.2] on page [319].

The copying is the performance bottleneck of the algorithm. Therefore only about 11 million partitions are generated per second. An \( O(1) \) algorithm for the Gray code starting with all elements in one set is given in [161].

15.2 The number of set partitions: Stirling set numbers and Bell numbers

<table>
<thead>
<tr>
<th>n</th>
<th>b= B(n)</th>
<th>k: 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b= 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b= 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b= 5</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b= 15</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>b= 52</td>
<td>1</td>
<td>15</td>
<td>25</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>b= 2033</td>
<td>1</td>
<td>51</td>
<td>90</td>
<td>65</td>
<td>15</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>b= 877</td>
<td>1</td>
<td>63</td>
<td>301</td>
<td>350</td>
<td>140</td>
<td>21</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b= 4140</td>
<td>1</td>
<td>127</td>
<td>966</td>
<td>1701</td>
<td>1050</td>
<td>266</td>
<td>28</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b= 7447</td>
<td>1</td>
<td>265</td>
<td>3025</td>
<td>7770</td>
<td>6851</td>
<td>2646</td>
<td>462</td>
<td>36</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>b= 115975</td>
<td>1</td>
<td>511</td>
<td>9330</td>
<td>34105</td>
<td>42525</td>
<td>22827</td>
<td>5880</td>
<td>750</td>
<td>45</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 15.2-A: Stirling numbers of the second kind and Bell numbers.

The sequence of numbers of set partitions of \( S_n \) for \( n \geq 1 \) is \( 1, 2, 5, 15, 52, 203, 877, \ldots \). These are the Bell numbers, sequence [A000110] of [245]. The Bell numbers can be computed using the observation that in every step of the computation a partition of \( S_{x-1} \) into \( k \) subsets leads to \( k+1 \) partitions of \( S_x \). Of these partitions \( k \) contain \( k \) subsets and one contains \( k+1 \) subsets.

If we write the number of partitions of \( S_n \) into \( k \) subsets as a triangular array, then the entry can be computed as the sum of its upper left neighbor plus \( k \) times its upper neighbor. We start with a single one in the row for \( n = 1 \). The scheme is shown in figure [15.2-A]. The numbers are the Stirling numbers of the second kind (or Stirling set numbers), see sequence [A008277] of [245].

The sum over all elements of row \( n \) gives the \( n \)-th Bell number. Another way of computing the Bell numbers is given in section [3.8.2] on page [137]. The array shown can be generated with the program [FXT: comb/stirling2-demo.cc]. For the Bell numbers we also have the recursion

\[
B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k
\]

[15.2-1]
Chapter 15: Set partitions

As pari/gp code:

```gp
? N=11; v=vector(N); v[1]=1;
? for (j=2, N, v[j]=sum(k=1, j-1, binomial(j-2,k-1)*v[k])); v
\[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975\]
```

15.2.1 Generating functions

The ordinary generating function for the Bell numbers can be given as

$$\sum_{n=0}^{\infty} B_n x^n = \sum_{k=0}^{\infty} \frac{x^k}{\prod_{j=1}^{k} (1-jx)} = 1 + x + 2x^2 + 5x^3 + 15x^4 + 52x^5 + \ldots \quad (15.2-2)$$

The exponential generating function (EGF) is

$$\exp[\exp(x) - 1] = \sum_{n=0}^{\infty} \frac{B_n}{n!} x^n \quad (15.2-3)$$

Dobinski’s formula for the Bell numbers is

$$B_n = \frac{1}{e} \sum_{k=1}^{\infty} \frac{n^k}{k!} \quad (15.2-4)$$

The array of Stirling numbers shown in figure 15.2-A can also be computed in polynomial form by setting $B_0(x) = 1$ and

$$B_{n+1}(x) = x \left[ B_n(x) + B_0(x) \right] \quad (15.2-5)$$

The coefficients of $B_n(x)$ are the Stirling numbers and $B_n(1) = B_n$:

```gp
? B=1; for(k=1,6, B=x*(deriv(B)+B); print(subst(B,x,1),": ",B))
1: x
2: x^2 + x
5: x^3 + 3x^2 + x
15: x^4 + 6x^3 + 7x^2 + x
52: x^5 + 10x^4 + 25x^3 + 15x^2 + x
203: x^6 + 15x^5 + 65x^4 + 90x^3 + 31x^2 + x
```

The polynomials are called Bell polynomials, see [273].

15.2.2 Set partitions of a given type

We say a set partition of the $n$-element set is of type $C = [c_1, c_2, c_3, \ldots, c_n]$ if it has $c_1$ one-element sets, $c_2$ two-element sets, $c_3$ three-element sets, and so on. Define

$$L(z) = \sum_{k=1}^{\infty} \frac{c_k z^k}{k!} \quad (15.2-6)$$

then

$$\exp(L(z)) = \sum_{n=0}^{\infty} \left[ \sum_{C} \left( Z_{n,C} \prod t_k^C \right) \right] \frac{z^n}{n!} \quad (15.2-7)$$

where $Z_{n,C}$ is the number of set partitions of the $n$-element set with type $C$.  

[fatbook draft of 2008-August-17]
15.3: Restricted growth strings

\[ n=8; R=O(z^{n+1}); \]
\[ L=\sum_{k=1}^8 \text{eval}(\text{Str}("t"^k)*z^k/k!)+R \]
\[ t_1*z + \frac{1}{2}t_2*z^2 + \frac{1}{6}t_3*z^3 + \frac{1}{24}t_4*z^4 + \ldots + \frac{1}{40320}t_8*z^8 + O(z^9) \]
\[ \text{serlaplace}(\exp(L)) \]
\[ 1 + t_1*z + (t_1^2 + t_2)*z^2 + (t_1^3 + 3*t_2*t_1 + t_3)*z^3 + (t_1^4 + 6*t_2*t_1^2 + 4*t_3*t_1 + 3*t_2^2 + t_4)*z^4 + \]
\[ (t_1^5 + 10*t_2*t_1^3 + 10*t_3*t_1^2 + 15*t_1*t_2^2 + 5*t_1*t_4 + 10*t_3*t_2 + t_5)*z^5 \]
\[ (t_1^6 + 15*t_2*t_1^4 + 20*t_3*t_1^3 + \ldots + 15*t_2^3 + 15*t_4*t_2 + 10*t_3^2 + t_6)*z^6 \]
\[ (t_1^7 + 21*t_2*t_1^5 + 35*t_3*t_1^4 + \ldots + 105*t_3*t_2^2 + 21*t_5*t_2 + 35*t_4*t_3 + t_7)*z^7 \]
\[ (t_1^8 + 28*t_2*t_1^6 + 56*t_3*t_1^5 + \ldots + 28*t_6*t_2 + 56*t_5*t_3 + 35*t_4^2 + t_8)*z^8 + O(z^9) \]

Specializations give generating functions for set partitions with certain restrictions. For example, the EGF for the partitions without sets of size one is (set \( t_1 = 0 \) and \( t_k = 1 \) for \( k \neq 1 \)) \( \exp(\exp(z) - 1 - z) \), see entry A000296 of [245]. Section 10.15.1.2 on page 284 gives a similar construction for the EGF for permutations of prescribed cycle type.

15.3 Restricted growth strings

For some applications the restricted growth strings (RGS) may suffice. We give algorithms for their generation, and describe classes of generalized RGS that contain the RGS for set partitions as a special case.

15.3.1 RGS for set partitions in lexicographic order

The C++ implementation [FXT: class setpart_rgs_lex in comb/setpart-rgs-lex.h] generates the RGS for set partitions in lexicographic order:

```cpp
class setpart_rgs_lex
// Set partitions of the n-set as restricted growth strings (RGS).
// Lexicographic order.
{
  public:
    ulong n_; // Number of elements of set (set = \{1,2,3,...,n\})
    ulong *m_; // m[k+1] = max(s[0], s[1],..., s[k]) + 1
    ulong *s_; // RGS

  public:
    setpart_rgs_lex(ulong n)
    {
      n_ = n;
      m_ = new ulong[n+1];
      m_[0] = ~0UL; // sentinel m[0] = infinity
      s_ = new ulong[n];
      first();
    }

    void last()
    {
      for (ulong k=0; k<s_; ++k) s_[k] = k;
    }
}
```

The method to compute the successor predecessor resembles the one used with mixed radix counting (see section 9.1 on page 219): find the first digit that can be incremented and increment it, then set all skipped digits to zero and adjust the array of maxima accordingly.

[fxtbook draft of 2008-August-17]
Chapter 15: Set partitions

bool next()
{
    if ( m_[n_] == n_ ) return false;

    ulong k = n_; 
    do { --k; } while ( (s_[k] + 1) > m_[k] ); 

    s_[k] += 1UL; 
    ulong mm = m_[k]; 
    mm += (s_[k]>=mm); 
    m_[k+1] = mm; // == max2(m_[k], s_[k]+1)

    while ( ++k<n_ )
    {
        s_[k] = 0;
        m_[k+1] = mm;
    } 

    return true;
}

The method for the predecessor is

bool prev()
{
    if ( m_[n_] == 1 ) return false;

    ulong k = n_; 
    do { --k; } while ( s_[k]==0 ); 

    s_[k] -= 1;
    ulong mm = m_[k+1] = max2(m_[k], s_[k]+1);

    while ( ++k<n_ )
    {
        s_[k] = mm; // == m[k]
        ++mm;
        m_[k+1] = mm;
    } 

    return true;
}

The rate of generation is about 157 M/s with next(), and 190 M/s with prev() [FXT: comb/setpart-rgs-lex-demo.cc].

15.3.2 RGS for set partitions into \(p\) parts (lex order)

The RGS corresponding to set partitions into \(p\) parts can be generated with [FXT: class setpart_p_rgs_lex in comb/setpart-p-rgs-lex.h]:

class setpart_p_rgs_lex
{
    public:
        ulong n_; // Number of elements of set (set = \{1,2,3,...,n\})
        ulong p_; // Exactly \(p\) subsets
        ulong *m_; // \(m[k+1] = \max(s[0], s[1],..., s[k]) + 1\)
        ulong *s_; // RGS

    public:
        setpart_p_rgs_lex(ulong n, ulong p)
        {
            n_ = n;
            m_ = new ulong[n+1];
            m_[0] = ~0UL; // sentinel m[0] = infinity
            s_ = new ulong[n_];
            first(p);
        }

        ~setpart_p_rgs_lex() // destructor
        {
            delete [] m_; 
            delete [] s_; 
        }

        void first(ulong p)
        // Must have 2<=p<=n
        {
            [fxtbook draft of 2008-August-17]
array of minimal values for m[] is [1 1 1 2 3]

1: s[ . . . 1 2 ] m[ 1 1 1 2 3 ] {1, 2, 3}, {4}, {5}
2: s[ . . . 1 . 2 ] m[ 1 1 1 2 3 ] {1, 2, 4}, {3}, {5}
3: s[ . . . 1 1 2 ] m[ 1 1 1 2 3 ] {1, 2}, {3, 4}, {5}
4: s[ . . . 1 1 . 2 ] m[ 1 1 2 3 3 ] {1, 2, 5}, {3}, {4}
5: s[ . . . 1 1 2 1 ] m[ 1 1 2 3 3 ] {1, 2}, {3, 5}, {4}
6: s[ . . . 1 1 2 2 ] m[ 1 1 2 3 3 ] {1, 2}, {3}, {4, 5}
7: s[ . . 1 . . 2 ] m[ 1 1 2 2 3 ] {1, 3, 4}, {2}, {5}
8: s[ . . 1 1 . 2 ] m[ 1 1 2 2 3 ] {1, 3}, {2, 4}, {5}
9: s[ . . 1 1 2 . ] m[ 1 1 2 2 3 ] {1, 3, 5}, {2}, {4}
10: s[ . . 1 1 2 1 ] m[ 1 1 2 2 3 ] {1, 3}, {2, 5}, {4}
11: s[ . . 1 1 2 2 ] m[ 1 1 2 2 3 ] {1, 3}, {2}, {4, 5}
12: s[ . . 1 1 . 2 1 ] m[ 1 1 2 2 3 ] {1, 4}, {2, 3}, {5}
13: s[ . . 1 1 1 2 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 4}, {5}
14: s[ . . 1 1 1 2 1 ] m[ 1 1 2 2 3 ] {1}, {2, 3}, {4, 5}
15: s[ . . 1 1 1 2 2 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 5}, {4}
16: s[ . . 1 1 2 2 ] m[ 1 1 2 2 3 ] {1}, {2, 3}, {4, 5}
17: s[ . . 1 . . 2 1 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 4}, {5}
18: s[ . . 1 1 . 2 1 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 4, 5}
19: s[ . . 1 1 1 . 2 ] m[ 1 1 2 2 3 ] {1}, {2, 3}, {4, 5}
20: s[ . . 1 1 1 2 1 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 4}, {5}
21: s[ . . 1 1 1 2 2 ] m[ 1 1 2 2 3 ] {1}, {2, 3}, {4, 5}
22: s[ . . 1 1 2 2 ] m[ 1 1 2 2 3 ] {1}, {2, 3}, {4, 5}
23: s[ . . 1 2 . . 1 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 4}, {5}
24: s[ . . 1 2 . 1 1 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 5}, {4}
25: s[ . . 1 2 1 . 2 ] m[ 1 1 2 2 3 ] {1}, {2, 3, 4, 5}

Figure 15.3-A: Restricted growth strings (left, dots for zeros) and array of prefix-maxima (middle) for
the set partitions of the 5-set into 3 parts (right).

```c
for (ulong k=0; k<n_; ++k) s_[k] = 0;
for (ulong k=n_-p+1, j=1; k<n_; ++k, ++j) s_[k] = j;
for (ulong k=1; k<=n_; ++k) m_[k] = s_[k-1]+1;
p_ = p;
}
bool next()
{
    // if ( 1==p_ ) return false; // make things work with p==1
    ulong k = n_; 
    bool q;
    do
    {
        --k;
        const ulong sk1 = s_[k] + 1;
        q = (sk1 > m_[k]); // greater max
        q |= (sk1 >= p_); // more than p parts
    } while ( q );
    if ( k == 0 ) return false;
    s_[k] += 1UL;
    ulong mm = m_[k];
    mm += (s_[k]>mm);
    m_[k+1] = mm; // == max2(m_[k], s_[k]+1);
    while ( ++k<n_ )
    {
        s_[k] = 0;
        m_[k+1] = mm;
    }
    ulong p = p;
    if ( mm<p ) // repair tail
```
Chapter 15: Set partitions

As given the computation will fail for $p = 1$, the line commented out removes this limitation. The rate of generation is about 108 M/s [FXT: `comb/setpart-p-rgs-lex-demo.cc`]. Figure 15.3-A shows all set partitions of the 5-set into 3 parts, together with their RGSs.

### 15.3.3 RGS for set partitions in minimal-change order

For the Gray code we need an additional array of directions, see section 9.2 on page 222 for the equivalent routines with mixed radix numbers. The implementation [FXT: `class setpart_rgs_gray in comb/setpart-rgs-gray.h`] allows to start either with the partition into one set or the partition into $n$ sets:

```cpp
1 class setpart_rgs_gray
2 {
3     public:
4         ulong n_; // Number of elements of set (set = {1,2,3,...,n})
5         ulong *m_; // m[k+1] = max(s[0], s[1],..., s[k]) + 1
6         ulong *s_; // RGS
7         ulong *d_; // direction with recursion (+1 or -1)
8         
9     public:
10         setpart_rgs_gray(ulong n, int dr0=+1)
11             // dr0=+1 ==> start with partition {{1,2,3,...,n}}
12             // dr0=-1 ==> start with partition {{1},{2},{3},...,{n}}
13             {
14             n_ = n;
15             m_ = new ulong[n+1];
16             m_[0] = ~0UL; // sentinel m[0] = infinity
17             s_ = new ulong[n_];
18             d_ = new ulong[n_];
19             first(dr0);
20         }
21     }
22 }
```

The method to compute the successor is

```cpp
1 bool next()
2 {
3     ulong k = n_;
4     do { --k; } while ( (s_[k] + d_[k]) > m_[k] ); // <0 or >max
5     if ( k == 0 ) return false;
6     s_[k] += d_[k];
7     m_[k+1] = max2(m_[k], s_[k]+1);
8     while ( ++k<n_ )
9         {
10         
11         
12         }
```
15.3: Restricted growth strings

const ulong d = d[k] = -d[k];
const ulong mk = m[k];

m[k+1] = mk + (d!=1UL); // == max2(mk, s[k]+1)

return true;

It must be noted that while the corresponding set partitions are in minimal-change order (see figure 15.1 on page 343) the RGS occasionally changes in more than one digit. The rate of generation is about 154 M/s [FXT: comb/setpart-rgs-gray-demo.cc].

15.3.4 Max-increment RGS *

The generation of RGSs \( s = [s_0, s_1, \ldots, s_{n-1}] \) where \( s_k \leq i + \max_{j<k}(s_j) \) is a generalization of the RGSs for set partitions (where \( i = 1 \)). Figure 15.3-B show RGSs in lexicographic order for \( i = 2 \) (left) and \( i = 1 \) (right). The strings can be generated in lexicographic order using [FXT: class rgs_maxincr in comb/rgs-maxincr.h]:

```cpp
class rgs_maxincr{
public:
ulong *s_; // restricted growth string
ulong *m_; // m[k-1] == max possible value for s[k]
ulong n_; // Length of strings
ulong i_; // s[k] <= max_{j<k}(s[j]+i)
// i==1 ==> RGS for set partitions

public:
rgs_maxincr(ulong n, ulong i=1)
{
    n_ = n;
    m_ = new ulong[n_];
    s_ = new ulong[n_];
    i_ = i;
    first();
}

~rgs_maxincr(){
    delete [] m_;
    delete [] s_;
}

void first()
{
    ulong n = n_;
    for (ulong k=0; k<n; ++k) s_[k] = 0;
    for (ulong k=0; k<n; ++k) m_[k] = i_;
}
}
```

The computation if the successor returns the index of first (leftmost) changed element in the string. Zero is returned if the current string is the last:

```cpp
ulong next()
{
    ulong k = n_;
    start:-k;
    if ( k==0 ) return 0;
    ulong sk = s_[k] + 1;
    ulong ml = m_[k-1];
    if ( sk > ml+i_ ) // "carry"
    {
        s_[k] = 0;
        goto start;
    }
    s_[k] = sk;
}
<table>
<thead>
<tr>
<th></th>
<th>RGS(4,2)</th>
<th>max(4,2)</th>
<th></th>
<th>RGS(5,1)</th>
<th>max(5,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>[. . . . ]</td>
<td>[. . . . ]</td>
<td>1:</td>
<td>[. . . . ]</td>
<td>[. . . . ]</td>
</tr>
<tr>
<td>2:</td>
<td>[. . . 1 ]</td>
<td>[. . . 1 ]</td>
<td>2:</td>
<td>[. . . . 1 ]</td>
<td>[. . . . 1 ]</td>
</tr>
<tr>
<td>3:</td>
<td>[. . . 2 ]</td>
<td>[. . . 2 ]</td>
<td>3:</td>
<td>[. . . 1 ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>4:</td>
<td>[. . . 3 ]</td>
<td>[. . . 3 ]</td>
<td>4:</td>
<td>[. . . 1 ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>6:</td>
<td>[. . 1 1 ]</td>
<td>[. . 1 1 ]</td>
<td>6:</td>
<td>[. . 1 . ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>7:</td>
<td>[. . 1 2 ]</td>
<td>[. . 1 2 ]</td>
<td>7:</td>
<td>[. . 1 . ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>8:</td>
<td>[. . 1 3 ]</td>
<td>[. . 1 3 ]</td>
<td>8:</td>
<td>[. . 1 . ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>9:</td>
<td>[. . 2 . ]</td>
<td>[. . 2 . ]</td>
<td>9:</td>
<td>[. . 1 . ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>10:</td>
<td>[. . 2 1 ]</td>
<td>[. . 2 1 ]</td>
<td>10:</td>
<td>[. . 1 . ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>11:</td>
<td>[. . 2 2 ]</td>
<td>[. . 2 2 ]</td>
<td>11:</td>
<td>[. . 1 . ]</td>
<td>[. . . 1 ]</td>
</tr>
<tr>
<td>14:</td>
<td>[. . 1 1 ]</td>
<td>[. . 1 1 ]</td>
<td>14:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>20:</td>
<td>[. . 2 1 ]</td>
<td>[. . 2 1 ]</td>
<td>20:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>29:</td>
<td>[. . 1 8 ]</td>
<td>[. . 1 8 ]</td>
<td>29:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>31:</td>
<td>[. . 2 0 ]</td>
<td>[. . 2 0 ]</td>
<td>31:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>32:</td>
<td>[. . 2 1 ]</td>
<td>[. . 2 1 ]</td>
<td>32:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>33:</td>
<td>[. . 2 2 ]</td>
<td>[. . 2 2 ]</td>
<td>33:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>34:</td>
<td>[. . 2 3 ]</td>
<td>[. . 2 3 ]</td>
<td>34:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>39:</td>
<td>[. . 2 8 ]</td>
<td>[. . 2 8 ]</td>
<td>39:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>41:</td>
<td>[. . 3 0 ]</td>
<td>[. . 3 0 ]</td>
<td>41:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>42:</td>
<td>[. . 3 1 ]</td>
<td>[. . 3 1 ]</td>
<td>42:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>43:</td>
<td>[. . 3 2 ]</td>
<td>[. . 3 2 ]</td>
<td>43:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>44:</td>
<td>[. . 3 3 ]</td>
<td>[. . 3 3 ]</td>
<td>44:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>47:</td>
<td>[. . 3 6 ]</td>
<td>[. . 3 6 ]</td>
<td>47:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>49:</td>
<td>[. . 3 8 ]</td>
<td>[. . 3 8 ]</td>
<td>49:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>50:</td>
<td>[. . 3 9 ]</td>
<td>[. . 3 9 ]</td>
<td>50:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>51:</td>
<td>[. . 4 0 ]</td>
<td>[. . 4 0 ]</td>
<td>51:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>52:</td>
<td>[. . 4 1 ]</td>
<td>[. . 4 1 ]</td>
<td>52:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>53:</td>
<td>[. . 4 2 ]</td>
<td>[. . 4 2 ]</td>
<td>53:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>54:</td>
<td>[. . 4 3 ]</td>
<td>[. . 4 3 ]</td>
<td>54:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>56:</td>
<td>[. . 4 5 ]</td>
<td>[. . 4 5 ]</td>
<td>56:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>58:</td>
<td>[. . 4 7 ]</td>
<td>[. . 4 7 ]</td>
<td>58:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
<tr>
<td>59:</td>
<td>[. . 4 8 ]</td>
<td>[. . 4 8 ]</td>
<td>59:</td>
<td>[. . 1 ]</td>
<td>[. . 1 ]</td>
</tr>
</tbody>
</table>

Figure 15.3-B: Length-4 max-increment RGS with maximal increment 2 and the corresponding array of maxima (left), and length-5 RGSs with maximal increment 1 (right). Dots denote zeros.
15.3: Restricted growth strings

if ( sk>m1 ) m1 = sk;
for (ulong j=k; j<n_; ++j ) m_[j] = m1;
return k;
}

About 115 million RGSs per second are generated with the routine. Figure 15.3-B was created with the program [FXT: comb/rgs-maxincr-demo.cc]. The sequence of numbers of Max-increment RGSs with increment $i$ =1, 2, 3, and 4, start

<table>
<thead>
<tr>
<th>n: 0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1: 1 1 2 5 15 52 203 877 4140 21147 115975</td>
</tr>
<tr>
<td>i=2: 1 1 3 12 59 339 2210 16035 127643 10269643</td>
</tr>
<tr>
<td>i=3: 1 1 4 22 150 1200 10922 110844 1236326 14990380 195895202</td>
</tr>
<tr>
<td>i=4: 1 1 5 35 305 3125 36479 475295 6811205 106170245 1784531879</td>
</tr>
</tbody>
</table>

The sequence for $i=2$ is entry A080337 of [245], it has the exponential generating function (EGF)

$$
\sum_{n=0}^{\infty} B_{n+1,2} \frac{x^n}{n!} = \exp \left[ x + \exp(x) + \frac{\exp(2x)}{2} - \frac{3}{2} \right]
$$

(15.3-1)

The sequence of numbers of increment-3 RGSs has the EGF

$$
\sum_{n=0}^{\infty} B_{n+1,3} \frac{x^n}{n!} = \exp \left[ x + \exp(x) + \frac{\exp(2x)}{2} + \frac{\exp(3x)}{3} - \frac{11}{6} \right]
$$

(15.3-2)

Omitting the empty set, we restate the EGF for the Bell numbers (relation 15.2-3 on page 346) as

$$
\sum_{n=0}^{\infty} B_{n+1,1} \frac{x^n}{n!} = \exp [x + \exp(x) - 1] = \frac{1}{0!} + \frac{2}{1!} x + \frac{5}{2!} x^2 + \frac{15}{3!} x^3 + \frac{52}{4!} x^4 + \ldots
$$

(15.3-3)

The EGF for the increment-$i$ RGS is

$$
\sum_{n=0}^{\infty} B_{n+1,i} \frac{x^n}{n!} = \exp \left[ x + \sum_{j=1}^{i} \frac{\exp(jx) - 1}{j} \right]
$$

(15.3-4)

15.3.5 F-increment RGS *

To obtain a different generalization of the RGS for set partitions we rewrite the condition $s_k \leq i + \max_{j<k}(s_j)$ for the RGS considered in the previous section:

$$
s_k \leq M(k) + i \text{ where } M(0) = 0 \text{ and } M(k+1) = \begin{cases} 
  s_{k+1} & \text{if } s_{k+1} - s_k > 0 \\
  M(k) & \text{else}
\end{cases}
$$

(15.3-5a)

(15.3-5b)

The function $M(k)$ is max$_{j<k}(s_j)$ in notational disguise. We define F-increment RGSs with respect to a function $F$ as follows:

$$
s_k \leq F(k) + i \text{ where } F(0) = 0 \text{ and } F(k+1) = \begin{cases} 
  s_{k+1} & \text{if } s_{k+1} - s_k = i \\
  F(k) & \text{else}
\end{cases}
$$

(15.3-6a)

(15.3-6b)

The function $F(k)$ is a ‘maximum’ that is increased only if the last increment ($s_k - s_{k-1}$) was maximal. For $i=1$ we obtain the RGSs for set partitions. Figure 15.3-C shows all length-4 F-increment RGSs for $i=2$ (left), and all length-3 RGSs for $i=5$ (right), together with the arrays of F-values. The listings were created with the program [FXT: comb/rgs-fincr-demo.cc] which uses the implementation [FXT: class rgs_fincr in comb/rgs-fincr.h]:

[fxtbook draft of 2008-August-17]
**Figure 15.3-C:** Length-4 F-increment restricted growth strings with maximal increment 2 and the corresponding array of values of $F$ (left), and length-3 RGSs with maximal increment 5 (right). Dots denote zeros.
15.3: Restricted growth strings

```cpp
class rgs_fincr
{
    public:
        ulong *s_; // restricted growth string
        ulong *f_; // values F(k)
        ulong n_; // Length of strings
        ulong i_; // s[k] <= f[k]+i

    ulong next() // Return index of first changed element in s[],
        // Return zero if current string is the last
    {
        ulong k = n_; // snip--
        start:
            if ( k==0 ) return 0;
            ulong sk = s_[k] + 1;
            ulong m1 = f_[k-1];
            ulong mp = m1 + i_;
            if ( sk > mp ) // "carry"
            {
                s_[k] = 0;
                goto start;
            }
            s_[k] = sk;
            if ( sk==mp ) m1 += i_;
            for (ulong j=k; j<n_; ++j ) f_[j] = m1;
        return k;
    }

The sequences of numbers of F-increment RGSs with increments \( i = 1, 2, 3, \) and \( 4 \), start

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>i=2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>15</td>
<td>52</td>
<td>203</td>
<td>877</td>
<td>4140</td>
<td>21147</td>
<td>115975</td>
</tr>
<tr>
<td>i=3</td>
<td>1</td>
<td>4</td>
<td>19</td>
<td>109</td>
<td>742</td>
<td>5815</td>
<td>51193</td>
<td>498118</td>
<td>5296321</td>
<td>60987817</td>
</tr>
<tr>
<td>i=4</td>
<td>1</td>
<td>5</td>
<td>29</td>
<td>201</td>
<td>1657</td>
<td>1539</td>
<td>10299</td>
<td>75905</td>
<td>609441</td>
<td>5284451</td>
</tr>
</tbody>
</table>

These are respectively entries \[ A000110 \) (Bell numbers), \[ A004211 \), \[ A004212 \), \[ A004213 \) of \[ 245 \. In general, the number \( F_{n,i} \) of F-increment RGSs (length \( n \), with increment \( i \)) is

\[
F_{n,i} = \sum_{k=0}^{n} i^{n-k} S(n,k) \tag{15.3-7}
\]

where \( S(n,k) \) are the Stirling numbers of the second kind. The exponential generating functions are

\[
\sum_{n=0}^{\infty} F_{n,i} \frac{x^n}{n!} = \exp \left[ \frac{\exp(ix) - 1}{i} \right] \tag{15.3-8}
\]

The ordinary generating functions are

\[
\sum_{n=0}^{\infty} F_{n,i} x^n = \sum_{n=0}^{\infty} \frac{x^n}{\prod_{k=1}^{n} (1 - ikx)} \tag{15.3-9}
\]

15.3.6 K-increment RGS *

We mention yet another type of restricted growth strings, the \( K \)-increment RGS, which satisfy

\[
s_k \leq s_{k-1} + k \tag{15.3-10}
\]

An implementation for their generation in lexicographic order is given in \[ FXT: \texttt{comb/rgs-kincr.h} \].
Chapter 15: Set partitions

Figure 15.3-D: The 37 K-increment RGS of length 4 in lexicographic order.

```cpp
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

The sequence of the numbers of K-increment RGS of length $n$ is entry A107877 of [245]:

```
n: 1 2 3 4 5 6 7 8 9 10 11 12
```

The strings of length 4 are shown in figure 15.3-D. They can be generated with the program [FXT: comb/rgs-kincr-demo.cc].
Chapter 16

A string substitution engine

---

Number of symbols = 3
Start: x
Rules:
  x --&gt; A
  A --&gt; Bx
  B --&gt; Bx

-------------
0: (#=1)
x 1: (#=1)
A 2: (#=2)
Bx 3: (#=3)
BxA 4: (#=5)
BxABx 5: (#=8)
BxABxBxA 6: (#=13)
BxABxBxABxABx 7: (#=21)
BxABxBxABxABxBxABxBxA 8: (#=34)
BxABxBxABxBxABxBxABxBxABxBxABxBx 9: (#=55)
BxABxBxABxBxABxBxABxBxABxBxABxBxABxBxABxBxABxBxABxBxA

Figure 16.0-A: Rules, axiom, and first steps of the evolution for a specific substitution engine.

String substitution: A finite set of symbols ('alphabet'), a set of substitution rules that map symbols to words (strings of symbols) and a start value ('axiom'). The rules have to be applied in parallel. An example, let \{x, A, B\} be the symbols and \{x \mapsto A, A \mapsto Bx, B \mapsto Bx\} the substitution rules. Start with a single x. The observed evolution is shown in figure 16.0-A. It is trivial to implement such a specific system, for the above:

```cpp
void fx(ulong n)
{
    if ( 0==n ) { cout << "x"; return; }
    fA(n-1);
}

void fA(ulong n)
{
    if ( 0==n ) { cout << "A"; return; }
    fB(n-1); fx(n-1);
}

void fB(ulong n)
{
    if ( 0==n ) { cout << "B"; return; }
    fx(n-1); fA(n-1);
}
```

[fxtbook draft of 2008-August-17]
Chapter 16: A string substitution engine

A utility class to create string substitution engines at run time is given in [FXT: class string_subst in comb/stringsubst.h]:

```cpp
class string_subst
{
public:
  // example values generate rabbit sequence:
  ulong nsym_; // # of symbols
  // == 2
  char *symbol_; // alphabet
  // == { '0', '1' };
  char **symrule_; // symrule_[i] string to replace i-th symbol with
  // == { "0", "1", "1", "10" }; for 0 |-> 1, 1 |-> 10
  ulong xlate_[256]; // translate char --> rule
  // 'a' --> symrule[xlate_['a']]
  ulong cmax_; // max string length
  char *cc_; // string to hold result
  ulong ctc_; // count chars of result actually produced

public:
  string_subst(int cmax, int nsym, char*const* symrule);

  ulong subst(ulong maxn, const char *start)
  // maxn:=number of generations, start:=axiom
  {
    ctc_ = 0;
    do_subst(maxn, start);
    cc_[ctc_] = 0; // terminate string
    return ctc_;
  }

private:
  void do_subst(ulong n, const char *rule)
  {
    if ( 0==n ) // add symbols to string:
      {
        for (ulong i=0; ctc_<cmax_; ++i)
          {
            char c = rule[i];
            if ( 0==c ) break;
            cc_[ctc_] = c;
            ++ctc_;}
    else // recurse:
      {
        for (ulong i=0; 0!=rule[i]; ++i)
          {
            ulong r = (ulong)rule[i];
            ulong x = xlate_[r];
            do_subst(n-1, symrule_[x]); // recursion
          }
      }
  }
};
```

Rules and symbols have to be supplied on construction. The member function subst takes an axiom and the number of generations as arguments and returns the numbers of produced characters. It calls the recursive function do_subst:

```cpp
int main()
{
  ulong n = 10; // max depth
  for (ulong k=0; k<=n; ++k)
    {
      fx(k);
      cout << endl;
    }
  return 0;
}
```

A utility class to create string substitution engines at run time is given in [FXT: class string_subst in comb/stringsubst.h]:

```cpp
class string_subst
{
public:
  // example values generate rabbit sequence:
  ulong nsym_; // # of symbols
  // == 2
  char *symbol_; // alphabet
  // == { '0', '1' };
  char **symrule_; // symrule_[i] string to replace i-th symbol with
  // == { "0", "1", "1", "10" }; for 0 |-> 1, 1 |-> 10
  ulong xlate_[256]; // translate char --> rule
  // 'a' --> symrule[xlate_['a']]
  ulong cmax_; // max string length
  char *cc_; // string to hold result
  ulong ctc_; // count chars of result actually produced

public:
  string_subst(int cmax, int nsym, char*const* symrule);

  ulong subst(ulong maxn, const char *start)
  // maxn:=number of generations, start:=axiom
  {
    ctc_ = 0;
    do_subst(maxn, start);
    cc_[ctc_] = 0; // terminate string
    return ctc_;
  }

private:
  void do_subst(ulong n, const char *rule)
  {
    if ( 0==n ) // add symbols to string:
      {
        for (ulong i=0; ctc_<cmax_; ++i)
          {
            char c = rule[i];
            if ( 0==c ) break;
            cc_[ctc_] = c;
            ++ctc_;}
    else // recurse:
      {
        for (ulong i=0; 0!=rule[i]; ++i)
          {
            ulong r = (ulong)rule[i];
            ulong x = xlate_[r];
            do_subst(n-1, symrule_[x]); // recursion
          }
      }
  }
};
```

A utility class to create string substitution engines at run time is given in [FXT: class string_subst in comb/stringsubst.h]:

```cpp
class string_subst
{
public:
  // example values generate rabbit sequence:
  ulong nsym_; // # of symbols
  // == 2
  char *symbol_; // alphabet
  // == { '0', '1' };
  char **symrule_; // symrule_[i] string to replace i-th symbol with
  // == { "0", "1", "1", "10" }; for 0 |-> 1, 1 |-> 10
  ulong xlate_[256]; // translate char --> rule
  // 'a' --> symrule[xlate_['a']]
  ulong cmax_; // max string length
  char *cc_; // string to hold result
  ulong ctc_; // count chars of result actually produced

public:
  string_subst(int cmax, int nsym, char*const* symrule);

  ulong subst(ulong maxn, const char *start)
  // maxn:=number of generations, start:=axiom
  {
    ctc_ = 0;
    do_subst(maxn, start);
    cc_[ctc_] = 0; // terminate string
    return ctc_;
  }
```

Rules and symbols have to be supplied on construction. The member function subst takes an axiom and the number of generations as arguments and returns the numbers of produced characters. It calls the recursive function do_subst:

```cpp
int main()
{
  ulong n = 10; // max depth
  for (ulong k=0; k<=n; ++k)
    {
      fx(k);
      cout << endl;
    }
  return 0;
}
```
The resulting string is stored in an array of characters that can be accessed via the member function string(). The characters that do not fit into the array are silently discarded. We use an array of one million characters, and rules and axiom as follows:

- **Number of symbols = 5**
- **Start:** a
- **Rules:**
  - a → -bF+aFa+Fb-
  - b → +aF-bFb-Fa+
  - F → -

We obtain (displaying only the string lengths from the fourth generation on):

<table>
<thead>
<tr>
<th>Generation</th>
<th>String Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(#=1) a</td>
</tr>
<tr>
<td>1</td>
<td>(#=11) -bF+aFa+Fb-</td>
</tr>
<tr>
<td>2</td>
<td>(#=51) -aF-bFb-Fa++bF+aFa+Fb-F-bF+aFa+Fb+-F-aF-bFb-Fa+-</td>
</tr>
<tr>
<td>3</td>
<td>(#=211) ++bF+aFa+Fb-F+aF-bFb-Fa++F-bF+aFa+Fb+F+-aF-bFb- \ Fa+F-bF+aFa+Fb-F-bFb-Fa+++F-bF+aFa+Fb-F++bF+aFa+Fb-F+aF-bFb-Fa+Fa+</td>
</tr>
<tr>
<td>4</td>
<td>(#=851) -++aF-bFb-Fa++F-bF+aFa+Fb-F+aF-bFb-Fa+F+-aF-bFb-Fa+-F-bF+aFa+Fb-+F+aF-bFb-Fa+-F-aF-bFb-Fa+-</td>
</tr>
<tr>
<td>5</td>
<td>(#=3411)</td>
</tr>
<tr>
<td>6</td>
<td>(#=13651)</td>
</tr>
<tr>
<td>7</td>
<td>(#=54611)</td>
</tr>
<tr>
<td>8</td>
<td>(#=218461)</td>
</tr>
<tr>
<td>9</td>
<td>(#=873811)</td>
</tr>
<tr>
<td>10</td>
<td>(#=1000000) // &lt;-- truncation</td>
</tr>
<tr>
<td>11</td>
<td>(#=1000000)</td>
</tr>
</tbody>
</table>

The computation takes a split second only. The strings generated by the shown rule correspond to the Hilbert curves in successively finer resolutions if one initializes:

- **position:** (x, y) := (0,0)
- **direction:** (dx, dy) := (1, 0)

and identifies:

- `'-'` == "turn left": { t= dx; dx=-dy; dy=t; }
- `'+'` == "turn right": { t=-dx; dx= dy; dy=t; }
- `'F'` == "advance": { line(x, y, x+dx, y+dy); x+=dx; y+=dy; }
- either 'a' or 'b' == (ignore)

The program [FXT: ds/stringsubst-demo.cc](#) prints evolutions for several rules. A 3-dimensional Hilbert (space-filling) curve is generated by the rule a → "<aF"<aF-a-F">aFvF+++aF-a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F">a-F>..."  

The number of symbols that occur in each generation can be computed by a simple matrix power. With i different symbols let \( n_j(k) \) be the number of occurrences of the j-th symbol at generation k. Then \( n_j(0) \) is determined by the axiom and

\[
\vec{N}(k) := [n_1(k), n_2(k), \ldots, n_i(k)]^T
\]  

(16.0-1)

\[
\vec{N}(k) = M^k \vec{N}(0)
\]  

(16.0-2)

where \( M \) is a matrix with columns according to the occurrences of the symbols in the transfer rules. For example, with the rules and axiom

- **Number of symbols = 3**
- **Start:** x
- **Rules:**
  - x → A 1 → [0, 1, 0]
  - A → Bx 2 → [1, 0, 1] =: transpose(M)
  - B → Bx 3 → [1, 0, 1]
we obtain

\[
M = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad M^k = \begin{bmatrix} F_{k-1} & F_k & F_k \\ F_{k-2} & F_{k-1} & F_{k-1} \\ F_{k-1} & F_k & F_k \end{bmatrix} \quad k \geq 2
\]

(16.0-3)

where \(F_k\) is the \(k\)-th Fibonacci number. For the tenth generation, \(k = 10\):

\[
M^{10} [1,0,0]^T = \begin{bmatrix} 34 & 55 & 55 \\ 21 & 34 & 34 \\ 34 & 55 & 55 \end{bmatrix} [1,0,1]^T = [34, 21, 34]^T
\]

(16.0-4)

So we have 34 ‘x’, 21 ‘A’ and 34 ‘B’ or a total of 89 symbols.

<table>
<thead>
<tr>
<th>Number of symbols</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start: a</td>
<td></td>
</tr>
<tr>
<td>Rules:</td>
<td></td>
</tr>
<tr>
<td>a --&gt; -bF+aFa+Fb-</td>
<td>1 --&gt; [2, 2, 2, 2, 3]</td>
</tr>
<tr>
<td>b --&gt; +aF-bFb-Fa+</td>
<td>2 --&gt; [2, 2, 2, 2, 3]</td>
</tr>
<tr>
<td>+ --&gt; +</td>
<td>3 --&gt; [0, 0, 1, 0, 0]</td>
</tr>
<tr>
<td>- --&gt; -</td>
<td>4 --&gt; [0, 0, 0, 1, 0]</td>
</tr>
<tr>
<td>F --&gt; F</td>
<td>5 --&gt; [0, 0, 0, 0, 1]</td>
</tr>
</tbody>
</table>

Table: Rules for a string substitution engine for a 3-dimensional Hilbert curve (top), and number of occurrences of the symbols after \(k\) iterations (bottom).

<table>
<thead>
<tr>
<th>k</th>
<th>total [ n(a), n(b), n(+), n(-), n(F) ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[ 1, 0, 0, 0, 0 ]</td>
</tr>
<tr>
<td>1</td>
<td>[ 2, 2, 2, 2, 3 ]</td>
</tr>
<tr>
<td>2</td>
<td>[ 8, 8, 10, 10, 15 ]</td>
</tr>
<tr>
<td>3</td>
<td>[ 32, 32, 42, 42, 63 ]</td>
</tr>
<tr>
<td>4</td>
<td>[ 128, 128, 170, 170, 255 ]</td>
</tr>
<tr>
<td>5</td>
<td>[ 512, 512, 682, 682, 1023 ]</td>
</tr>
<tr>
<td>6</td>
<td>[ 2048, 2048, 2730, 2730, 4095 ]</td>
</tr>
<tr>
<td>7</td>
<td>[ 8192, 8192, 10922, 10922, 16383 ]</td>
</tr>
<tr>
<td>8</td>
<td>[ 32768, 32768, 43690, 43690, 65535 ]</td>
</tr>
<tr>
<td>9</td>
<td>[ 131072, 131072, 174762, 174762, 262143 ]</td>
</tr>
<tr>
<td>10</td>
<td>[ 524288, 524288, 699050, 699050, 1048575 ]</td>
</tr>
</tbody>
</table>

Figure 16.0-B: Rules for a string substitution engine for a 3-dimensional Hilbert curve (top), and number of occurrences of the symbols after \(k\) iterations (bottom).

The rules for the engine corresponding to a 3D Hilbert curve are given in figure 16.0-B (top), the number of symbols after the \(k\)-th iteration are given at the bottom of the figure. In chapter 36 on page 723 the string substitution idea is used to construct fast iterations for various functions whose power series are determined by the limiting string.

An efficient algorithm for finding the \(n\)-th letter of the string produced by \(k\) iterations of a substitution rule is described in [242]. The problem of the identification of a finite automaton (substitution engine) that generates a given infinite string is discussed in [182]. The mathematical properties of sequences generated by finite automata are studied in [10].
Chapter 17

Necklaces and Lyndon words

A sequence that is minimal among all its cyclic rotations is called a *necklace* (see section 3.8.1 on page 134 for the definition in terms of equivalence classes). When there are $k$ possible values for each element one talks about an $n$-bead, $k$-color (or $k$-ary length-$n$) necklaces. We restrict our attention to the case were only two sorts of beads are allowed and represent them by 0 and 1.

<table>
<thead>
<tr>
<th>$n=1$: #=2</th>
<th>$n=2$: #=3</th>
<th>$n=3$: #=4</th>
<th>$n=4$: #=6</th>
<th>$n=5$: #=8</th>
<th>$n=6$: #=14</th>
<th>$n=7$: #=20</th>
<th>$n=8$: #=36</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: i 1</td>
<td>0: ........1 6</td>
<td>0: ........1 8</td>
<td>0: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: 11</td>
<td>1: ........1 8</td>
<td>1: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: 11</td>
<td>2: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: 111</td>
<td>3: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: 1111</td>
<td>4: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: 11111</td>
<td>5: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: 111111</td>
<td>6: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: 1111111</td>
<td>7: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: 11111111</td>
<td>8: ........1 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 17.0-A:** All binary necklaces of lengths up to 8 and their periods. Dots represent zeros.

Scanning all binary words of length $n$ as to whether they are necklaces can easily be achieved by testing whether the word $x$ is equal to the return value of the function `bit_cyclic_min(x,n)` shown in section 1.13 on page 30. For $n$ up to 8 one obtains the sequences of binary necklaces shown in figure 17.0-A. As $2^n$ words have to be tested this approach gets inefficient for large $n$. Luckily there is both a much better algorithm for generating all necklaces and a formula for their number.

Not all necklaces are created equal. Each necklace can be assigned a period that is a divisor of the length.
That period is the smallest (nonzero) cyclic shift that transforms the word into itself. The periods are given directly right to each necklace in figure 17.0-A. For $n$ prime the only periodic necklaces are those two that contain all ones or zeros. Aperiodic (or equivalently, period equals length) necklaces are called 
Lyndon words.

For a length-$n$ binary word $x$ the function $\text{bit}_\text{cyclic}_\text{period}(x, n)$ from section 1.13 on page 30 returns the period of the word.

## 17.1 Generating all necklaces

We give several methods to generate all necklaces of a given size. An efficient algorithm for the generation of bracelets (see section 3.8.1.4 on page 135) is given in [236].

### 17.1.1 The FKM algorithm

The following algorithm for generation of all necklaces actually produces pre-necklaces a subset of which are the necklaces. A pre-necklace is a string that is the prefix of some necklace. The FKM algorithm (for Fredericksen, Kessler, Maiorana) to generate all $k$-ary length-$n$ pre-necklaces proceeds as follows:

Figure 17.1-A: Ternary length-4 (left) and binary length-6 (right) pre-necklaces as generated by the FKM algorithm. Dots are used for zeros, necklaces are marked with 'N', Lyndon words with 'L'.

The following algorithm for generation of all necklaces actually produces pre-necklaces a subset of which are the necklaces. A pre-necklace is a string that is the prefix of some necklace. The FKM algorithm (for Fredericksen, Kessler, Maiorana) to generate all $k$-ary length-$n$ pre-necklaces proceeds as follows:
17.1: Generating all necklaces

1. Initialize the word \( F = [f_1, f_2, \ldots, f_n] \) to all zeros. Set \( j = 1 \).

2. (Visit pre-necklace \( F \). If \( j \) divides \( n \) then \( F \) is a necklace. If \( j \) equals \( n \) then \( F \) is a Lyndon word.)

3. Find the largest index \( j \) so that \( f_j < k-1 \). If there is no such index (then \( F = [k-1, k-1, \ldots, k-1] \), the last necklace), then terminate.

4. Increment \( f_j \). Fill the suffix starting at \( f_{j+1} \) with copies of \( [f_1, \ldots, f_j] \). Goto step 2.

The crucial steps are \[\text{FXT: comb/necklace-fkm-demo.cc}\]:

```plaintext
1 for (ulong i=1; i<=n; ++i) f[i] = 0; // Initialize to zero
2 bool nq = 1; // whether pre-necklace is a necklace
3 bool lq = 0; // whether pre-necklace is a Lyndon word
4 ulong j = 1;
5 while ( 1 )
6 {
7     // Print necklace:
8     cout << setw(4) << pct << "":";
9     print_vec( "", f+1, n, true);
10    cout << " j=" << j;
11    if ( nq ) cout << " N";
12    if ( lq ) cout << " L";
13    cout << endl;
14
15    // Find largest index where we can increment:
16    j = n;
17    while ( f[j]==k-1 ) { --j; }
18
19    if ( j==0 ) break;
20    ++f[j];
21
22    // Copy periodically:
23    for (ulong i=1,t=j+1; t<=n; ++i,++t) f[t] = f[i];
24    nq = ( (n%j)==0 ); // necklace if j divides n
25    lq = ( j==n ); // Lyndon word if j equals n
26 }
```

Two example runs are shown in figure \[\text{17.1-A}\]. An efficient implementation of the algorithm is \[\text{FXT: class necklace in comb/necklace.h}\]:

```plaintext
class necklace
{
public:
    ulong *a_; // the string
    ulong *dv_; // delta sequence of divisors of n
    ulong n_; // length of strings
    ulong m1_; // m-ary strings, m1=m-1
    ulong j_; // period of the word (if necklaces)

public:
    necklace(ulong m, ulong n)
    {
        n_ = ( n ? n : 1 ); // at least one
        m1_ = ( m>1 ? m-1 : 1); // at least two
        a_ = new ulong[n+m1];
        dv_ = new ulong[n+m1];
        for (ulong j=1; j<=n; ++j) dv_[j] = ( 0==(n%j ) ); // divisors
        first();
    }

    void first()
    {
        for (ulong j=0; j<n_; ++j) a_[j] = 0;
        j_ = 1;
    }

    ulong next_pre() // next pre-necklace
    {
        // return j (zero when finished)
    }
```

[fxtbook draft of 2008-August-17]
Chapter 17: Necklaces and Lyndon words

```cpp
{ // Find rightmost digit that can be incremented:
ulong j = n_;
while ( a_[j] == m1_ ) { --j; }
// Increment:
// if ( 0==j_ ) return 0; // last
++a_[j];
// Copy periodically:
for (ulong k=j+1; k<n_; ++k) a_[k] = a_[k-j];
j_ = j;
return j;
}
```

Note the commented out return with the last word, this gives a speedup (and no harm is done with the following copying). The array dv allows determination whether the current pre-necklace is also a necklace (or Lyndon word) via simple lookups:

```cpp
bool is_necklace() const
{
  return ( 0!=dv_[j_] ); // whether j divides n
}

bool is_lyn() const
{
  return ( j_==n_ ); // whether j equals n
}
```

Thereby we have methods for the computation of the next necklace or Lyndon word:

```cpp
ulong next() // next necklace
{
  do
  {
    next_pre();
    if ( 0==j_ ) return 0;
  } while ( 0==dv_[j_] ); // until j divides n
  return j_;
}

ulong next_lyn() // next Lyndon word
{
  do
  {
    next_pre();
    if ( 0==j_ ) return 0;
  } while ( j_==n_ ); // until j equals n
  return j_; // == n
}
```

The rate of generation for pre-necklaces is about 98 M/s for base 2, 140 M/s for base 3, and 180 M/s for base 4 [FXT: comb/necklace-demo.cc]. An specialization of the algorithm for binary necklaces is [FXT: class binary_necklace in comb/binary-necklace.h]. The rate of generation for pre-necklaces is about 128 M/s [FXT: comb/binary-necklace-demo.cc]. A version of the algorithm that produces the binary necklaces as bits of a word is given in section 1.13.3 on page 32.

The binary necklaces of length \( n \) can be used as cycle leaders in the length-2\(^n\) zip permutation (and its inverse) that is discussed in section 2.5 on page 99. An algorithm for the generation of all irreducible binary polynomials via Lyndon words is described in section 38.10 on page 852.
17.1: Generating all necklaces

17.1.2 Binary Lyndon words with length a Mersenne exponent

The length-$n$ binary Lyndon words for $n$ an exponent of a Mersenne prime $M_n = 2^n - 1$ can be generated efficiently as binary expansions of the powers of a primitive root $r$ of $m$ until the second word with just one bit is reached. With $n = 7$, $m = 127$ and the primitive root $r = 3$ we get the sequence shown in figure 17.1-B. The sequence of minimal primitive roots $r_0$ of the first Mersenne primes $M_n = 2^n - 1$ is entry A096393 of [245]:

$$
\begin{array}{c c}
2 & 2 \\
3 & 3 \\
5 & 3 \\
7 & 3 \\
13 & 17 \\
\end{array}
$$

The routine generates about 71 million binary pre-necklaces per second. Ternary and 5-ary pre-necklaces are generated at a rate of about 100 and 113 million per second, respectively.

17.1.3 A constant amortized time (CAT) algorithm

A constant amortized time (CAT) algorithm to generate all $k$-ary length-$n$ (pre-)necklaces is given in [80]. The crucial part of a recursive algorithm [FXT: comb/necklace-cat-demo.cc] is the function

```cpp
ulong K, N; // K-ary pre-necklaces of length N
ulong f[N];
void crsms_gen(ulong n, ulong j)
{
    if ( n > N ) visit(j); // pre-necklace in f[1,...,N]
    else
        { f[n] = f[n-j];
          crsms_gen(n+1, j);
          for (ulong i=f[n-j]+1; i<K; ++i)
              { f[n] = i;
                crsms_gen(n+1, n);
              }
        }
}
```

After initializing the array with zeros the function must be called with both arguments equal to one. The routine generates about 71 million binary pre-necklaces per second. Ternary and 5-ary pre-necklaces are generated at a rate of about 100 and 113 million per second, respectively.
Chapter 17: Necklaces and Lyndon words

17.1.4 An order with fewer transitions

<table>
<thead>
<tr>
<th>n</th>
<th>X_n</th>
<th>n</th>
<th>X_n</th>
<th>n</th>
<th>X_n</th>
<th>n</th>
<th>X_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>13</td>
<td>95</td>
<td>19</td>
<td>2598</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>14</td>
<td>163</td>
<td>20</td>
<td>4546</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>290</td>
<td>21</td>
<td>8135</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>479</td>
<td>22</td>
<td>14427</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>11</td>
<td>34</td>
<td>17</td>
<td>859</td>
<td>23</td>
<td>26212</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>12</td>
<td>54</td>
<td>18</td>
<td>1450</td>
<td>24</td>
<td>46957</td>
</tr>
</tbody>
</table>

Figure 17.1-C: The 30 binary 8-bit Lyndon words in an order with few changes between successive words. Transitions where more than one bit changes are marked with a ‘<<’.

The following routine generates the binary pre-necklaces words in the order that would be obtained by selecting valid words from the binary Gray code:

```c
void xgen(ulong n, ulong j, int x=+1)
{
    if ( n > N ) visit(j);
    else
    {
        if ( -1==x )
        {
            if ( 0==f[n-j] ) { f[n] = 1; xgen(n+1, n, -x); }
            f[n] = f[n-j]; xgen(n+1, j, +x);
        }
        else
        {
            if ( 0==f[n-j] ) { f[n] = 1; xgen(n+1, j, +x); }
        }
    }
}
```

The program [FXT: comb/necklace-gray-demo.cc] computes the binary Lyndon words with the given routine. The ordering obtained has fewer transitions between successive elements but is in general not a Gray code (for up to 6-bit words a Gray code is generated). Figure 17.1-C shows the output with 8-bit Lyndon words. The first 2^((n/2) – 1) Lyndon words of length n are in Gray code order. The number X_n of additional transitions of the length-n Lyndon words is, for n ≤ 30, shown in figure 17.1-D.

17.1.5 An order with at most three changes per transition

An algorithm to generate necklaces in an order such that at most 3 elements change with each update is given in [275]. The recursion can be given as (corrected and shortened) [FXT: comb/necklace-gray3-demo.cc]:

```c
long *f; // data in f[1..m], f[0] = 0
long N; // word length
int k; // k-ary necklaces, k==sigma in the paper
void gen3(int z, int t, int j)
{
```
17.1: Generating all necklaces

Figure 17.1-E: The 30 binary 8-bit necklaces in an order with at most 3 changes per transition. Transitions where more than one bit changes are marked with ‘<<’.

<table>
<thead>
<tr>
<th></th>
<th>X_n</th>
<th></th>
<th>X_n</th>
<th></th>
<th>X_n</th>
<th></th>
<th>X_n</th>
<th></th>
<th>X_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:</td>
<td>1</td>
<td>8:</td>
<td>12</td>
<td>14:</td>
<td>360</td>
<td>20:</td>
<td>11722</td>
<td>26:</td>
<td>441370</td>
</tr>
</tbody>
</table>

Figure 17.1-F: Excess (with respect to Gray code) of number of bits changed.

```
7  if ( t > N ) { visit(j); }
8  else
9  {
10   if ( (z&1) == 0 ) // z (number of elements == (k-1)) is even?
11     {
12       for (int i=f[t-j]; i<=k-1; ++i)
13         {
14           f[t] = i;
15           gen3( z+(i!=k-1), t+1, (i!=f[t-j]?t:j) );
16         }
17     }
18   else
19     {
20       for (int i=k-1; i>=f[t-j]; --i)
21         {
22           f[t] = i;
23           gen3( z+(i!=k-1), t+1, (i!=f[t-j]?t:j) );
24         }
25     }
26 }
27 }
```

The variable z counts the number of maximal elements. The output with length-8 binary necklaces is shown in figure 17.1-E. Selecting the necklaces from the reversed list of complemented Gray codes of the n-bit binary words produces the same list.

17.1.6 Binary necklaces of length $2^n$ via Gray-cycle leaders *

The algorithm for the generation of cycle leaders for the Gray code permutation given section 2.8.1 on page 104 and relation 1.19-10c on page 56 written as

$$S_k Y x = Y g^k x$$

($Y$ is the yellow code, or bit-wise Reed-Muller transform) allow us to generate the necklaces of length $2^n$. The cyclic shifts of $Y x$ are equal to $Y g^k x$ for $k = 0, \ldots, l - 1$ where $l$ is the cycle length. Figure 17.1-G shows the correspondence between cycles of the Gray code permutation and cyclic shifts, it was generated with the program [FXT: comb/necklaces-via-gray-leaders-demo.cc].
Chapter 17: Necklaces and Lyndon words

### 17.1.7 Binary necklaces via cyclic shifts and complements *

A recursive algorithm to generate all nonzero binary necklaces via cyclic shifts and complements of the lowest bit is described in [228]. An implementation of the method is given in [FXT: comb/necklace-sigma-tau-demo.cc]:

```plaintext
1 inline ulong sigma(ulong x) { return bit_rotate_left(x, 1, n); }
2 inline ulong tau(ulong x) { return x ^ 1; }
3 void search(ulong y)
4 {
5     visit(y);
6     ulong t = y;
7     for (n = 3; n <= 8; n++)
8     {
9         search(sigma(t));
10         search(tau(t));
11     }
```

---

If no better algorithm for the cyclic leaders of the Gray code permutation was known we could generate them as $Y^{-1}(N) = Y(N)$ where $N$ are the necklaces of length $2^n$. The same idea, and relation 1.19-11b on page 56, give the relation

$$S_k B x = B e^{-k} x$$

(17.1-2)

where $B$ is the blue code and $e$ the reversed Gray code.

---

### Figure 17.1-G:

Left: the cycle leaders (minima) $L$ of the Gray code permutation where is highest bit has index 7 and their bit-wise Reed-Muller transforms $Y(L)$. Right: the last two cycles and the transforms of their elements.

---

### Figure 17.1-H:

Nonzero binary necklaces of lengths $n = 3, 4, \ldots, 8$ as generated by the shift and complement algorithm.

---

[kx] [fxtbook draft of 2008-August-17]
The initial call is \texttt{search(1)}. The generated ordering for lengths $n = 3, 4, \ldots, 8$ is shown in figure 17.1-H.

### 17.2 The number of binary necklaces

The number of binary necklaces of length $n$ equals

$$N_n = \frac{1}{n} \sum_{d|n} \varphi(d) 2^{n/d} = \frac{1}{n} \sum_{j=1}^{n} 2^{\gcd(j,n)}$$  \hspace{1cm} (17.2-1)$$

Replace 2 by $k$ to get the number for $k$ different sorts of beads (possible values at each digit). The values for $n \leq 40$ are shown in figure 17.2-A. The sequence is entry A000031 of [245].

The number of Lyndon words (aperiodic necklaces) equals

$$L_n = \frac{1}{n} \sum_{d|n} \mu(d) 2^{n/d} = \frac{1}{n} \sum_{d|n} \mu(n/d) 2^d$$  \hspace{1cm} (17.2-2)$$
Chapter 17: Necklaces and Lyndon words

The Möbius function $\mu$ is defined in relation 35.1-6 on page 689. The values for $n \leq 40$ are given in figure 17.2-B. The sequence is entry A001037 of [245]. For prime $n = p$ we have $L_p = N_p - 2$ and

$$L_p = \frac{2p - 2}{p} = \frac{1}{p} \sum_{k=1}^{p-1} \binom{p}{k}$$

(17.2-3)

The latter form transpires that there are exactly $\binom{p}{k}/p$ Lyndon words with $k$ ones for $1 \leq k \leq p - 1$. The difference of two is due to the necklaces that consist of all zeros or ones. The number of irreducible binary polynomials (see section 38.6 on page 840) of degree $n$ also equals $L_n$. For the equivalence between necklaces and irreducible polynomials see section 38.10 on page 852.

Let $d$ be a divisor of $n$. There are $2^n$ binary words of length $n$, each having some period $d$ that divides $n$. There are $d$ different shifts of the corresponding word, thereby

$$2^n = \sum_{d|n} d L_d$$

(17.2-4)

Möbius inversion gives relation 17.2-2. The necklaces of length $n$ and period $d$ are obtained by concatenation of $n/d$ Lyndon words of length $d$, so

$$N_n = \sum_{d|n} L_d$$

(17.2-5)

We note the relations (see section 35.1.4 on page 692)

$$\prod_{k=1}^{\infty} \left(1 - y^k\right)^{L_k} = \sum_{k=1}^{\infty} \frac{-\mu(k)}{k} \log \left(1 - 2y^k\right)$$

(17.2-6a)

$$\sum_{k=1}^{\infty} L_k y^k = \sum_{k=1}^{\infty} \frac{-\mu(k)}{k} \log \left(1 - 2y^k\right)$$

(17.2-6b)

Defining

$$\eta_B(x) := \prod_{k=1}^{\infty} \left(1 - By^k\right)$$

(17.2-7a)

we have

$$\eta_2(x) = \prod_{k=1}^{\infty} \left(1 - y^k\right)^{N_k}$$

(17.2-7b)

$$\eta_2(x) = \prod_{k=1}^{\infty} \eta_1(y^k)^{L_k}$$

(17.2-7c)

17.3 The number of binary necklaces with fixed content

Let $N_{(n,n_0)}$ be the number of binary length-$n$ necklaces with exactly $n_0$ zeros (and $n_1 = n - n_0$ ones) the necklaces with fixed density. One has

$$N_{(n,n_0)} = \frac{1}{n} \sum_{j|g} \varphi(j) \binom{n/j}{d/j}$$

(17.3-1)

where $g = \gcd(n,n_0)$. Bit-wise complementing gives the symmetry relation $N_{(n,n_0)} = N_{(n,n-n_0)} = N_{(n,n_1)}$. A table of small values is given in figure 17.3-A.
### 17.3: The number of binary necklaces with fixed content

<table>
<thead>
<tr>
<th>( n )</th>
<th>( N_n )</th>
<th>( N_{(n,0)} )</th>
<th>( N_{(n,1)} )</th>
<th>( N_{(n,2)} )</th>
<th>( N_{(n,3)} )</th>
<th>( N_{(n,4)} )</th>
<th>( N_{(n,5)} )</th>
<th>( N_{(n,6)} )</th>
<th>( N_{(n,7)} )</th>
<th>( N_{(n,8)} )</th>
<th>( N_{(n,9)} )</th>
<th>( N_{(n,10)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>36</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>60</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>14</td>
<td>14</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>108</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>22</td>
<td>26</td>
<td>22</td>
<td>12</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>188</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>15</td>
<td>30</td>
<td>42</td>
<td>42</td>
<td>30</td>
<td>15</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>352</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>19</td>
<td>43</td>
<td>66</td>
<td>80</td>
<td>66</td>
<td>43</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>632</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>22</td>
<td>55</td>
<td>99</td>
<td>132</td>
<td>132</td>
<td>99</td>
<td>55</td>
<td>22</td>
</tr>
<tr>
<td>14</td>
<td>1182</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>26</td>
<td>73</td>
<td>143</td>
<td>217</td>
<td>246</td>
<td>217</td>
<td>143</td>
<td>73</td>
</tr>
<tr>
<td>15</td>
<td>2192</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>31</td>
<td>91</td>
<td>201</td>
<td>335</td>
<td>429</td>
<td>429</td>
<td>335</td>
<td>201</td>
</tr>
<tr>
<td>16</td>
<td>4116</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>35</td>
<td>116</td>
<td>273</td>
<td>504</td>
<td>715</td>
<td>810</td>
<td>715</td>
<td>504</td>
</tr>
<tr>
<td>17</td>
<td>7712</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>40</td>
<td>140</td>
<td>364</td>
<td>728</td>
<td>1144</td>
<td>1430</td>
<td>1430</td>
<td>1144</td>
</tr>
<tr>
<td>18</td>
<td>14602</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>46</td>
<td>172</td>
<td>476</td>
<td>1038</td>
<td>1768</td>
<td>2438</td>
<td>2704</td>
<td>2438</td>
</tr>
<tr>
<td>19</td>
<td>27596</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>51</td>
<td>204</td>
<td>612</td>
<td>1428</td>
<td>2652</td>
<td>3978</td>
<td>4862</td>
<td>4862</td>
</tr>
<tr>
<td>20</td>
<td>52488</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>57</td>
<td>245</td>
<td>776</td>
<td>1944</td>
<td>3876</td>
<td>6310</td>
<td>8398</td>
<td>9252</td>
</tr>
</tbody>
</table>

**Figure 17.3-A:** The number \( N_{(n,z)} \) of binary necklaces of length \( n \) with \( z \) zeros.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( L_n )</th>
<th>( L_{(n,0)} )</th>
<th>( L_{(n,1)} )</th>
<th>( L_{(n,2)} )</th>
<th>( L_{(n,3)} )</th>
<th>( L_{(n,4)} )</th>
<th>( L_{(n,5)} )</th>
<th>( L_{(n,6)} )</th>
<th>( L_{(n,7)} )</th>
<th>( L_{(n,8)} )</th>
<th>( L_{(n,9)} )</th>
<th>( L_{(n,10)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>56</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>14</td>
<td>14</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>99</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>20</td>
<td>25</td>
<td>20</td>
<td>12</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>186</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>15</td>
<td>30</td>
<td>42</td>
<td>42</td>
<td>30</td>
<td>15</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>335</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>18</td>
<td>40</td>
<td>66</td>
<td>75</td>
<td>66</td>
<td>40</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>630</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>22</td>
<td>55</td>
<td>99</td>
<td>132</td>
<td>132</td>
<td>99</td>
<td>55</td>
<td>22</td>
</tr>
<tr>
<td>14</td>
<td>1161</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>26</td>
<td>70</td>
<td>143</td>
<td>212</td>
<td>245</td>
<td>212</td>
<td>143</td>
<td>70</td>
</tr>
<tr>
<td>15</td>
<td>2182</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>30</td>
<td>91</td>
<td>200</td>
<td>333</td>
<td>429</td>
<td>429</td>
<td>333</td>
<td>200</td>
</tr>
<tr>
<td>16</td>
<td>4080</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>35</td>
<td>112</td>
<td>273</td>
<td>497</td>
<td>715</td>
<td>800</td>
<td>715</td>
<td>497</td>
</tr>
<tr>
<td>17</td>
<td>7710</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>40</td>
<td>140</td>
<td>364</td>
<td>728</td>
<td>1144</td>
<td>1430</td>
<td>1430</td>
<td>1144</td>
</tr>
<tr>
<td>18</td>
<td>14532</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>45</td>
<td>168</td>
<td>476</td>
<td>1026</td>
<td>1768</td>
<td>2424</td>
<td>2700</td>
<td>2424</td>
</tr>
<tr>
<td>19</td>
<td>27594</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>51</td>
<td>204</td>
<td>612</td>
<td>1428</td>
<td>2652</td>
<td>3978</td>
<td>4862</td>
<td>4862</td>
</tr>
<tr>
<td>20</td>
<td>52377</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>57</td>
<td>240</td>
<td>775</td>
<td>1932</td>
<td>3876</td>
<td>6288</td>
<td>8398</td>
<td>9225</td>
</tr>
</tbody>
</table>

**Figure 17.3-B:** The number \( L_{(n,z)} \) of binary Lyndon words of length \( n \) with \( z \) zeros.
Let $L_{(n,n_0)}$ be the number of binary length-$n$ Lyndon words with exactly $n_0$ zeros (Lyndon words with fixed density), then

$$L_{(n,n_0)} = \frac{1}{n} \sum_{j \in g} \mu(j) \binom{n/j}{d/j}$$

(17.3-2)

where $g = \gcd(n, n_0)$. The symmetry relation is the same as for $N_{(n,n_0)}$. A table of small values is given in figure 17.3-B.

Let $N_{(n_0,n_1,\ldots,n_k-1)}$ be the number of $k$-symbol length-$n$ necklaces with $n_j$ occurrences of symbol $j$, the number of such necklaces with fixed content, we have ($n = \sum_{j<s} n_j$ and):

$$N_{(n_0,n_1,\ldots,n_k-1)} = \frac{1}{n} \sum_{d \mid g} \varphi(d) \frac{(n/d)!}{(n_0/d)!\cdots(n_k-1/d)!}$$

(17.3-3)

where $g = \gcd(n_0, n_1, \ldots, n_k-1)$. The equivalent formula for the Lyndon words with fixed content is

$$L_{(n_0,n_1,\ldots,n_k-1)} = \frac{1}{n} \sum_{d \mid g} \mu(d) \frac{(n/d)!}{(n_0/d)!\cdots(n_k-1/d)!}$$

(17.3-4)

where $g = \gcd(n_0, n_1, \ldots, n_k-1)$. The relations were taken from [229] and [237] which also give efficient algorithms for the generation of necklaces and Lyndon words with fixed density and content, respectively. The number of strings with fixed content is a multinomial coefficient, see relation 11.2-1a on page 202.

A method for the generation of all necklaces with forbidden substrings is given in [230].
Chapter 18

Hadamard and conference matrices

The $2^k \times 2^k$-matrices corresponding to the Walsh transforms (see chapter 22 on page 457) are special cases of so-called Hadamard matrices. Such matrices also exist for certain sizes $n \times n$ for $n$ not a power of two. We give construction schemes for Hadamard matrices that come from the theory of finite fields.

If we denote the transform matrix for an $N$-point Walsh transform by $H$, then

$$HH^T = N \text{id} \quad (18.0-1)$$

where id is the unit matrix. The matrix $H$ is orthogonal (up to normalization) and its determinant equals

$$\det(H) = \det(HH^T)^{1/2} = N^{N/2} \quad (18.0-2)$$

Further, all entries are either $+1$ or $-1$. An orthogonal matrix with these properties is called a Hadamard matrix. We know that for $N = 2^n$ we always can find such a matrix. For $N = 2$ we have

$$H_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \quad (18.0-3)$$

and we can use the Kronecker product (see section 22.3 on page 461) to construct $H_{2N}$ from $H_N$ via

$$H_n = \begin{bmatrix} +H_{N/2} & +H_{N/2} \\ +H_{N/2} & -H_{N/2} \end{bmatrix} = H_2 \otimes H_{N/2} \quad (18.0-4)$$

The problem of determining Hadamard matrices (especially for $N$ not a power of two) comes from combinatorics. Hadamard matrices of size $N \times N$ can only exist if $N$ equals 1, 2, or $4k$.

18.1 Hadamard matrices via LFSR

We start with a construction for certain Hadamard matrices for $N$ a power of two that uses m-sequences that are created by shift registers (see section 39.1 on page 861 and section 39.5 on page 870). Figure 18.1-A shows three Hadamard matrices that were constructed as follows:

1. Choose $N = 2^n$ and create a maximum length binary shift register sequence $S$ of length $N - 1$.
2. Make $S$ signed, that is, replace all ones by $-1$ and all zeros by $+1$.
3. The $N \times N$ matrix $H$ is obtained by filling the first row and the first column with ones and filling the remaining entries with cyclical copies of $s$: for $r = 1, 2, \ldots N - 1$ and $c = 1, 2, \ldots N - 1$ set $H_{r,c} := S_{c-r+1 \mod N-1}$.
For given $n$ we can obtain as many different Hadamard matrices as there are m-sequences.

The matrices in figure 18.1-A where produced with the program [FXT: comb/hadamard-srs-demo.cc].

```cpp
#include "bpol/lfsr.h" // class lfsr
#include "aux1/copy.h" // copy_cyclic()
#include "matrix/matrix.h" // class matrix
typedef matrix<int> Smat; // matrix with integer entries

ulong n = 5;
ulong N = 1UL << n;

// --- create signed SRS:
int vec[N-1];
lfsr S(n);
for (ulong k=0; k<N-1; ++k) {
    ulong x = 1UL & S.get_a();
    vec[k] = ( x ? -1 : +1 );
    S.next();
}

// --- create Hadamard matrix:
Smat H(N,N);
for (c=0; c<N; ++c) H.set(0, c, +1); // first row = [1,1,1,...,1]
for (ulong r=1; r<N; ++r)
    copy_cyclic(vec, H.rowp_[r]+1, N-1, N-r);
```

The function copy_cyclic() is defined in [FXT: aux1/copy.h].

```cpp
template <typename Type>
inline void copy_cyclic(const Type *src, Type *dst, ulong n, ulong s)
// Copy array src[] to dst[] // starting from position s in src[]
// wrap around end of src[] (src[n-1])
// src[] is assumed to be of length n
// dst[] must be length n at least
// Equivalent to: { copy(src, dst, n); rotate_right(dst, n, s)}
{
    ulong k = 0;
    while ( s<n ) dst[k++] = src[s++];
    s = 0;
}
```
16     while ( k<n ) dst[k++] = src[s++];
17  }

If we define the matrix $X$ to be the $(N - 1) \times (N - 1)$ block of $H$ obtained by deleting the first row and column then

$$
X X^T = \begin{bmatrix}
N - 1 & -1 & -1 & \cdots & -1 \\
-1 & N - 1 & -1 & \cdots & -1 \\
-1 & -1 & N - 1 & \cdots & -1 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
-1 & -1 & -1 & \cdots & N - 1
\end{bmatrix}
$$

(18.1-1)

Equivalently, for the (cyclic) auto correlation of $S$ (see section 39.6 on page 872):

$$
\sum_{k=0}^{L-1} S_k S_{k+\tau \mod L} = \begin{cases} 
+L & \text{if } \tau = 0 \\
-1 & \text{else}
\end{cases}
$$

(18.1-2)

where $L = N - 1$ is the length of the sequence.

A alternative way to obtain Hadamard matrices of dimension $2^n$ is to use the signs in the multiplication table for hypercomplex numbers described in section 37.14 on page 814.

### 18.2 Hadamard matrices via conference matrices

<table>
<thead>
<tr>
<th>Quadratic characters modulo 13:</th>
<th>Quadratic characters modulo 11:</th>
</tr>
</thead>
<tbody>
<tr>
<td>14x14 conference matrix C:</td>
<td>12x12 conference matrix C:</td>
</tr>
<tr>
<td>0 + + + + + + + + + + + + + +</td>
<td>0 + + + + + + + + + + + + + +</td>
</tr>
<tr>
<td>+ 0 + + + + + + + + + + + + +</td>
<td>- 0 + + + + + + + + + + + + +</td>
</tr>
<tr>
<td>+ + 0 + + + + + + + + + + + +</td>
<td>- + 0 + + + + + + + + + + + +</td>
</tr>
<tr>
<td>+ + + 0 + + + + + + + + + + +</td>
<td>- + + 0 + + + + + + + + + + +</td>
</tr>
<tr>
<td>+ + + + 0 + + + + + + + + + +</td>
<td>- + + + 0 + + + + + + + + + +</td>
</tr>
<tr>
<td>+ + + + + 0 + + + + + + + + +</td>
<td>- + + + + 0 + + + + + + + + +</td>
</tr>
<tr>
<td>+ + + + + + 0 + + + + + + + +</td>
<td>- + + + + + 0 + + + + + + + +</td>
</tr>
<tr>
<td>+ + + + + + + 0 + + + + + + +</td>
<td>- + + + + + + 0 + + + + + + +</td>
</tr>
<tr>
<td>+ + + + + + + + 0 + + + + + +</td>
<td>- + + + + + + + 0 + + + + + +</td>
</tr>
<tr>
<td>+ + + + + + + + + 0 + + + + +</td>
<td>- + + + + + + + + 0 + + + + +</td>
</tr>
<tr>
<td>+ + + + + + + + + + 0 + + + +</td>
<td>- + + + + + + + + + 0 + + + +</td>
</tr>
<tr>
<td>+ + + + + + + + + + + 0 + + +</td>
<td>- + + + + + + + + + + 0 + + +</td>
</tr>
<tr>
<td>+ + + + + + + + + + + + 0 + +</td>
<td>- + + + + + + + + + + + 0 + +</td>
</tr>
<tr>
<td>+ + + + + + + + + + + + + 0 +</td>
<td>- + + + + + + + + + + + + 0 +</td>
</tr>
<tr>
<td>+ + + + + + + + + + + + + + 0</td>
<td>- + + + + + + + + + + + + + 0</td>
</tr>
</tbody>
</table>

Figure 18.2-A: Two Conference matrices, the entries not on the diagonal are ±1 and only the sign is given. The left is a symmetric $14 \times 14$ matrix ($13 \equiv 1 \mod 4$), the right is a antisymmetric $12 \times 12$ matrix ($11 \equiv 3 \mod 4$). Replacing all diagonal elements of the right matrix with +1 gives a $12 \times 12$ Hadamard matrix.

A conference matrix $C_Q$ is a $Q \times Q$ matrix with zero diagonal and all other entries ±1 so that

$$
C C^T = (Q - 1) \text{id}
$$

(18.2-1)

An algorithm for the construction of $C_q$ for $Q = q + 1$ where $q$ is an odd prime:

1. Create a length-$q$ array $S$ with entries $S_k \in \{-1, 0, +1\}$ as follows: set $S_0 = 0$ and, for $1 \leq k < q$ set $S_k = +1$ if $k$ is a square modulo $q$, $S_k = -1$ else.

2. Set $y = 1$ if $q \equiv 1 \mod 4$, else $y = -1$ (then $q \equiv 3 \mod 4$).

3. Create a $Q \times Q$ matrix $C$ as follows: set $C_{0,0} = 0$ and $C_{0,k} = +1$ for $1 \leq k < Q$ (first row). Set $C_{k,0} = y$ for $1 \leq k < Q$ (first column). Fill the remaining entries with cyclical copies of $S$: for $1 \leq r < q$ and $1 \leq c < q$ set $C_{r,c} = S_{c-r+1 \mod q}$.
The quantity $y$ tells us whether $C$ is symmetric ($y = +1$) or antisymmetric ($y = -1$).

If $C_Q$ is antisymmetric then $H_Q = C_Q + \text{id}$ is a Hadamard matrix. For example, replacing all zeros in the $12 \times 12$ matrix in figure 18.2-A by $+1$ gives a $12 \times 12$ Hadamard matrix.

If $C_Q$ is symmetric then a $2Q \times 2Q$ Hadamard matrix is given by

\[
H_{2Q} := \begin{bmatrix}
+ \text{id} + C & - \text{id} + C \\
- \text{id} + C & - \text{id} - C
\end{bmatrix}
\] (18.2-2)

Figure 18.2-B shows a $12 \times 12$ Hadamard matrix that was created using this formula.

The program [FXT: comb/conference-quadres-demo.cc] outputs for a given $q$ the $Q \times Q$ conference matrix and the corresponding Hadamard matrix:

```cpp
#include "mod/numtheory.h" // kronecker()  
#include "matrix/matrix.h" // class matrix  
#include "aux1/copy.h" // copy_cyclic()  

--snip--

int y = ( 1==q%4 ? +1 : -1 );  
ulong Q = q+1;  

--snip--

// --- create table of quadratic characters modulo q:  
int vec[q]; fill<int>(vec, q, -1); vec[0] = 0;  
for (ulong k=1; k<(q+1)/2; ++k) vec[(k*k)%q] = +1;  

--snip--

// --- create Q x Q conference matrix:  
Smat C(Q,Q);  
C.set(0,0, 0);  
for (ulong c=1; c<Q; ++c) C.set(0, c, +1); // first row = [1,1,1,...,1]  
for (ulong r=1; r<Q; ++r)  
{  
    C.set(r, 0, y); // first column = +-[1,1,1,...,1]^T  
    copy_cyclic(vec, C.rowp_[r]+1, q, Q-r);  
}

--snip--

// --- create a N x N Hadamard matrix:  
ulong N = ( y<0 ? Q : 2*Q );  
Smat H(N,N);  
if ( N==Q )  
{  
    copy(C, H);  
    H.diag_add_val(1);  
}  
else  
{  
    Smat K2(2,2); K2.fill(+1); K2.set(1,1, -1); // K2 = [+1,+1; +1,-1]  
    H.kronecker(K2, C); // Kronecker product of matrices  
    for (ulong k=0; k<Q; ++k) // adjust diagonal of sub-matrices  
    {  
        ulong r, c;  
        r=k; c=k; H.set(r,c, H.get(r,c)+1);  
        r=k; c=k+Q; H.set(r,c, H.get(r,c)-1);  
        r=k+Q; c=k; H.set(r,c, H.get(r,c)-1);  
    }
}
```

[fxtbook draft of 2008-August-17]
If both $H_a$ and $H_b$ are Hadamard matrices (of dimensions $a$ and $b$, respectively) then their Kronecker product $H_{ab} = H_a \otimes H_b$ is again a Hadamard matrix:

$$H_{ab} H_{ab}^T = (H_a \otimes H_b) (H_a \otimes H_b)^T = (H_a \otimes H_b) H_a^T \otimes H_b^T \equiv \star$$

(18.2-3a)

(the starred equality uses relation 22.3-11a on page 402)

$$= (H_a^T \otimes H_b^T) (H_a \otimes H_b) = (a \text{id}) \otimes (b \text{id}) = ab \text{id} \quad (18.2-3b)$$

(the starred equality uses relation 22.3-10a on page 402).

### 18.3 Conference matrices via finite fields

The algorithm for odd primes $q$ can be modified to work also for powers of odd primes. Then one has to work with the finite fields $GF(q^n)$. The entries $C_{r+1,c+1}$ for $r = 0, 1, \ldots, q^n-1$ and $c = 0, 1, \ldots, q^n-1$ have to be the quadratic character of $z_r - z_c$ where $z_0, z_1, \ldots, z_{q^n-1}$ are the elements in $GF(q^n)$ in some (fixed) order.

We implement the algorithm in pari/gp. Firstly, we give two simple routines that map the elements $z_i \in GF(q^n)$ (represented as polynomials modulo $q$) to the numbers $0, 1, \ldots, q^n-1$. The polynomial $p(x) = c_0 + c_1 x + \ldots + c_{n-1} x^{n-1}$ is mapped to $N = c_0 + c_1 q + \ldots + c_{n-1} q^{n-1}$.

```plaintext
pol2num(p,q)=
{ local(t, n); n=0;
  for (k=0, poldegree(p),
    t = polcoeff(p, k);
    t *= Mod(1,q);
    t = component(t, 2);
    t *= q^k;
    n += t;
  );
  return( n );
}
```

The inverse routine is

```plaintext
num2pol(n,q)=
{ local(p, mq, k);
  p = Pol(0,'x);
  k = 0;
  while ( 0!=n,
    mq = n \% q;
    p += mq * ('x)^k;
    n = mq;
    n \= q;
    k++;
  );
  return( p );
}
```

The quadratic character of an element $z$ can be determined by computing $z^{(q^n-1)/2}$ modulo the field polynomial. The result will be zero for $z = 0$, else $\pm 1$. The following routine determines the character of the difference of two elements as required for the computation of conference matrices:

```plaintext
getquadchar_n(n1, n2, q, fp, n)=
{ local(p, mq, k);
  p = Pol(0,'x);
  k = 0;
  while ( 0!=n,
    mq = n \% q;
    p += mq * ('x)^k;
    n = mq;
    n \= q;
    k++;
  );
  return( p );
}
```

[fxtbook draft of 2008-August-17]
Chapter 18: Hadamard and conference matrices

```plaintext
local(p1, p2, d, nd, sc);
if ( n1==n2, return(0) );
p1 = num2pol(n1, q);
p2 = num2pol(n2, q);
d = Mod(1,q) * (p2-p1);
d = Mod(d,fp) \((q^n-1)/2\);
d = component(d, 2);
if ( Mod(1,q)==d, sc=+1, sc=-1 );
return( sc );
}
The input are two numbers that are mapped to the corresponding field elements.

In order to reduce the computational work we create a table of the quadratic characters for later lookup:

```plaintext
quadcharvec(fp, q)=
\ Return a table of quadratic characters in GF(q^n)
\ fp is the field polynomial.
{
local(n, qn, sv, pl);
n=poldegree(fp);
qn=q^n-1;
sv=vector(qn+1, j, -1);
sv[1] = 0;
for (k=1, qn,
pl = num2pol(k,q);
pl = Mod(Mod(1,q)*pl, fp);
sq = pl * pl;
sq = component(sq, 2);
i = pol2num( sq, q );
sv[i+1] = +1;
);
return( sv );
}
```
Using this table we can compute the quadratic characters of the difference of two elements in a more efficient manner:

```plaintext
getquadchar_v(n1, n2, q, fp, sv)=
\ Return the quadratic character of (n2-n1) in GF(q^n)
\ Table lookup method
{
local(p1, p2, d, nd, sc);
if ( n1==n2, return(0) );
p1 = num2pol(n1, q);
p2 = num2pol(n2, q);
d = (p2-p1) % fp;
nd = pol2num(d, q);
sc = sv[nd+1];
return( sc );
}
```
Now we can compute conference matrices:

```plaintext
matconference(q, fp, sv)=
\ Return a QxQ conference matrix.
\ q an odd prime.
\ sv table of quadratic characters in GF(q^n)
\ where n is the degree of fp.
{
local(y, Q, C, n);
n = poldegree(fp);
Q=q^n+1;
C = matrix(Q,Q);
for (k=2, Q, C[1,k]=+1); \ first row
for (k=2, Q, C[k,1]=+1); \ first column
for (r=2, Q,
for (c=2, Q,
sc = getquadchar_n(r-2, c-2, q, fp, n);
sc = getquadchar_v(r-2, c-2, q, fp, sv); \ same result
C[r,c] = sc;
);
);
```
```
To compute a $Q \times Q$ conference matrix where $Q = q^n$ we need to find an polynomial of degree $n$ that is irreducible modulo $q$. With $q = 3$ and the field polynomial $f = x^2 + 1$ we obtain the $10 \times 10$ conference matrix shown in figure [18.3-A]. A conference matrix for $q = 3$ and $f = x^3 - x + 1$ is given in figure [18.3-B]. Hadamard matrices can be created in the same manner as before, the symmetry criterion being whether $q^n \equiv \pm 1 \pmod{4}$.

The construction of Hadamard matrices via conference matrices is due to Raymond Paley. The conference matrices obtained are of size $c = q^n + 1$ where $q$ is an odd prime. The values $c \leq 100$ are (see sequence A061344 of [245]):

$$4, 6, 8, 10, 12, 14, 16, 18, 20, 24, 26, 28, 30, 32, 38, 42, 44, 48, 50, 54, 60, 62, 68, 72, 74, 80, 82, 84, 90, 96, 98$$

We do not obtain conference matrices for any odd $c$ and these even values $c \leq 100$:

$$2, 16, 22, 34, 36, 40, 46, 52, 56, 58, 64, 66, 70, 76, 78, 86, 88, 92, 94, 96, 100$$

For example, $c = 16 = 15 + 1 = 3 \cdot 5 + 1$ has not the required form.
If a conference matrix of size $c$ exists then we can create Hadamard matrices of sizes $N = c$ whenever $q^n \equiv 3 \mod 4$, and $N = 2c$ whenever $q^n \equiv 1 \mod 4$. Further, if Hadamard matrices of sizes $N$ and $M$ exist then a $(N \cdot M) \times (N \cdot M)$ Hadamard matrix can be obtained via the Kronecker product.

The values of $N = 4k \leq 2000$ such that this construction does not give a $N \times N$ Hadamard matrix are:


This is sequence [A046116](OEIS) of [245](https://oeis.org/A046116). It can be obtained by starting with a list of all numbers of the form $4k$ and deleting all values $k = 2^a (q + 1)$ where $q$ is a power of an odd prime. Constructions for Hadamard matrices for numbers of certain forms are known, see [190](https://oeis.org/A046116). Whether Hadamard matrices exist for all values $N = 4k$ is an open problem. A readable source about constructions for Hadamard matrices is [248](https://oeis.org/A046116).
Chapter 19

Searching paths in directed graphs

We describe how certain combinatorial structures can be represented as paths or cycles in a directed graph. As an example consider Gray codes of $n$-bit binary words: we are looking for sequences of all $2^n$ binary words such that only one bit changes between two successive words. A convenient representation of the search space is that of a graph. The nodes are the binary words and an edge is drawn between two nodes if the node’s values differ by exactly one bit. Every path that visits all nodes of that graph corresponds to a Gray code. If the path is a cycle then a Gray cycle was found.

In general we can, depending on the size of the problem,

1. try to find at least one object
2. generate all objects
3. show that no such object exists

The method used is usually called backtracking. We will see how to reduce the search space if additional constraints are imposed on the paths. Finally, we show how careful optimization can lead to surprising algorithms for objects of a size where one would hardly expect to obtain a result at all. In fact, Gray cycles through the $n$-bit binary Lyndon words for all odd $n \leq 37$ are determined.

Terminology and conventions

We will use the terms node (instead of vertex) and edge (sometimes called arc). We restrict our attention to directed graphs (or digraphs) as undirected graphs are just the special case of these: an edge in an undirected graph corresponds to two antiparallel edges (think: ‘arrows’) in a directed graph.

A length-$k$ path is a sequence of nodes where an edge leads from each node to its successor. A path is called simple if the nodes are pairwise distinct. We restrict our attention to simple paths of length $N$ where $N$ is the number of nodes of in the graph. We use the term full path for a simple path of length $N$.

If in a simple path there is an edge from the last node of the path to the starting node the path is a cycle (or circuit). A full path that is a cycle is called a Hamiltonian cycle, a graph containing such a cycle is called Hamiltonian.

We allow for loops (edges that start and point to the same node). Graphs that contain loops are called pseudo graphs. The algorithms used will effectively ignore loops. We disallow multigraphs (where multiple edges can start and end at the same two nodes), as these would lead to repeated output of identical objects.

The neighbors of a node are those nodes where outgoing edges point to. Neighbors can be reached with one step. The neighbors of a node a called adjacent to the node. The adjacency matrix of a graph with $N$ nodes is a $N \times N$ matrix $A$ where $A_{i,j} = 1$ if there is an edge from node $i$ to node $j$, else $A_{i,j} = 0$. 
While trivial to implement (and later modify) we will not use this kind of representation as the memory requirement would be prohibitive for large graphs.

19.1 Representation of digraphs

For our purposes a static implementation of the graph as arrays of nodes and (outgoing) edges will suffice. The container class `digraph` merely allocates memory for the nodes and edges. The correct initialization is left to the user [FXT: class digraph in graph/digraph.h]:

```cpp
class digraph
{
public:
    ulong ng_; // number of Nodes of Graph
    ulong *ep_; // e[ep[k]], ..., e[ep[k+1]-1]: outgoing connections of node k
    ulong *e_; // outgoing connections (Edges)
    ulong *vn_; // optional: sorted values for nodes
    // if vn is used, then node k must correspond to vn[k]
    public:
    digraph(ulong ng, ulong ne, ulong *&ep, ulong *&e, bool vnq=false)
    : ng_(0), ep_(0), e_(0), vn_(0)
    {
        ng_ = ng;
        ep_ = new ulong[ng_+1];
        e_ = new ulong[ne];
        ep = ep_;
        e = e_;
        if ( vnq ) vn_ = new ulong[ng_];
    }
    ~digraph()
    {
        delete [] ep_;
        delete [] e_;
        if ( vn_ ) delete [] vn_;
    }
    --snip--
    void get_edge_idx(ulong p, ulong &fe, ulong &en) const
    // Setup fe and en so that the nodes reachable from p are
    // e[fe], e[fe+1], ..., e[en-1].
    // Must have: 0<=p<ng
    {
        fe = ep_[p]; // (index of) First Edge
        en = ep_[p+1]; // (index of) first Edge of Next node
    }
    --snip--
    void print(const char *bla=0) const;
};
```

The nodes reachable from node `p` could be listed using

```cpp
// ulong p; // == position
cout << "The nodes reachable from node " << p << " are:" << endl;
ulong fe, en;
g_.get_edge_idx(p, fe, en);
for (ulong ep=fe; ep<en; ++ep) cout << e_[ep] << endl;
```

Using our representation there is no cheap method to find the incoming edges. We will not need this information for our purposes. If the graph is known to be undirected, the same routine obviously lists the incoming edges.

Initialization routines for certain digraphs are declared in [FXT: graph/mk-special-digraphs.h]. A simple example is [FXT: make_complete_digraph() in graph/mk-complete-digraph.cc]:

```cpp
digraph
make_complete_digraph(ulong n)
// Initialization for the complete graph.
```

[fxtbook draft of 2008-August-17]
19.2: Searching full paths

\begin{verbatim}
{  ulong ng = n, ne = n*(n-1);
  ulong *ep, *e;
  digraph dg(ng, ne, ep, e);
  ulong j = 0;
  for (ulong k=0; k<ng; ++k) // for all nodes
  {
    ep[k] = j;
    for (ulong i=0; i<n; ++i) // connect to all nodes
    {
      if ( k==i ) continue; // skip loops
      e[j++] = i;
    }
  }
  ep[ng] = j;
  return dg;
}
\end{verbatim}

We initialize the complete graph (the undirected graph that has edges between any two of its nodes) for \( n = 5 \) and print it [FXT: \texttt{graph/graph-perm-demo.cc}]:

\begin{verbatim}
digraph dg = make_complete_digraph(5);
dg.print("Graph =");
\end{verbatim}

The output is

\begin{verbatim}
Graph =
Node: Edge0 Edge1 ... 
 0: 1 2 3 41: 0 2 3 42: 0 1 3 43: 0 1 2 44: 0 1 2 3
#nodes=5 #edges=20
\end{verbatim}

For many purposes it suffices to implicitly represent the nodes as values \( p \) with \( 0 \leq p < N \) where \( N \) is the number of nodes. If not, the values of the nodes have to be stored in the array \( vn[] \). One such example is a graph where the value of node \( p \) is the \( p \)-th (cyclically minimal) Lyndon word that we will meet at the end of this chapter. To make the search for a node by value reasonably fast the array \( vn[] \) should be sorted so that binary search can be used.

### 19.2 Searching full paths

In order to search full paths starting from some position \( p_0 \) we need two additional arrays for the bookkeeping: A record \( rv[] \) of the path so far, its \( k \)-th entry shall be \( p_k \), the node visited at step \( k \). Further a tag array \( qq[] \) that shall contain a zero for nodes not visited so far, else one. The crucial parts of the implementation are [FXT: \texttt{class digraph_paths in graph/digraph-paths.h}]:

\begin{verbatim}
class digraph_paths
  // Find all full paths in a directed graph.
  { public:
    digraph &g_; // the graph
    ulong *rv_; // Record of Visits: rv[k] == node visited at step k
    ulong *qq_; // qq[k] == whether node k has been visited yet
    [--snip--]
    // function to call with each path found with all_paths():
    ulong (*pfunc_)(digraph_paths &);
    [--snip--]
    // function to impose condition with all_cond_paths():
    bool (*cfunc_)(digraph_paths &, ulong ns);
  }

public:
  // graph/digraph.cc:
  digraph_paths(digraph &g);
  ~digraph_paths();
  [--snip--]
  bool path_is_cycle() const;
\end{verbatim}

[fxtbook draft of 2008-August-17]
Chapter 19: Searching paths in directed graphs

```cpp
Chapter 19: Searching paths in directed graphs

21 // graph/digraphpaths-search.cc:
22 ulong all_paths(ulong (*pfunc)(digraph_paths &),
23 ulong ns=0, ulong p=0, ulong maxnp=0);
24
25 // pfunc: function to visit (process) paths
26 // ns: start at node index ns (for fixing start of path)
27 // p: start at node value p (for fixing start of path)
28 // maxnp: stop if maxnp paths were found
29 { pfdone_ = 0;
30  pfunc_ = pfunc;
31  maxnp_ = maxnp;
32  next_path(ns, p);
33  return pfct_; // Number of paths where pfunc() returned true
34 }
```

We could have used a bit-array for the tag values \(qq_[]\). It turns out that some additional information can be saved there as we will see in a moment.

To keep matters simple a recursive algorithm is used to search for (full) paths. The search is started via call to `all_paths()` [FXT: graph/digraph-paths.cc]:

```cpp
ulong
digraph_paths::all_paths(ulong (*pfunc)(const digraph_paths &),
ulong ns/*=0*/, ulong p/*=0*/, ulong maxnp/*=0*/) // pfunc: function to visit (process) paths
// ns: start at node index ns (for fixing start of path)
// p: start at node value p (for fixing start of path)
// maxnp: stop if maxnp paths were found
{ pct_ = 0;
cct_ = 0;
pfct_ = 0;
pfunc_ = pfunc;
pdone_ = 0;
maxnp_ = maxnp;
next_path(ns, p);
return pfct_; // Number of paths where pfunc() returned true
}
```

The search is done by the function `next_path()`:

```cpp
void
digraph_paths::next_path(ulong ns, ulong p)
// ns+1 == how many nodes seen
// p == position (node we are on)
{ if ( pfdone_ ) return;
  rv_[ns] = p; // record position
  ++ns;
  if ( ns==ng_ ) // all nodes seen ?
  { pfunc_(*this);
    return;
  }
  else
  { qq_[p] = 1; // mark position as seen (else loops lead to errors)
    ulong fe, en;
    g_.get_edge_idx(p, fe, en);
    ulong fct = 0; // count free reachable nodes // FCT
    for (ulong ep=fe; ep<en; ++ep)
    { ulong t = g_.e_[ep]; // next node
      if ( 0==qq_[t] ) // node free?
      { ++fct;
        qq_[p] = fct; // mark position as seen: record turns // FCT
        next_path(ns, t);
      }
    }
    // if ( 0==fct ) { "dead end: this is a U-turn"; } // FCT
    qq_[p] = 0; // unmark position
  }
}
```

[fxtbook draft of 2008-August-17]
The lines that are commented with // FCT record which among the free nodes is visited. The algorithm still works if these lines are commented out.

### 19.2.1 Paths in the complete graph: permutations

```
    #nodes=5  #edges=20
```

#### Figure 19.2-A: Edges of the complete graph with 5 nodes (left), and full paths starting at node 0 (right). The paths (where 0 is omitted) correspond to the permutations of 4 elements in lexicographic order.

The program [FXT: graph/graph-perm-demo.cc](#) shows the paths in the complete graph from section 19.1 on page 382. We give slightly simplified version:

```
ulong pfunc_perm(digraph_paths &dp)
#else Function to be called with each path:
    const ulong *rv = dp.rv_; 
    ulong ng = dp.ng_; 
    cout << setw(4) << dp.pfct_ <<": ";
    for (ulong k=1; k<ng; ++k) cout << " " << rv[k];
    cout << endl;
#else return 1;
```

The output, shown in figure 19.2-A, is a listing of the permutations of the numbers 1, 2, 3, 4 in lexicographic order (see section 10.1 on page 233).

### 19.2.2 Paths in the De Bruijn graph: De Bruijn sequences

The graph with $2^n$ nodes and two outgoing edges from node $k$ to $2k$ mod $2^n$ and $2k + 1$ mod $2^n$ is called a (binary) De Bruijn graph. For $n = 8$ the graph is (printed horizontally):

```
    Node:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    Edge 0:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    Edge 1:  1  3  5  7  9 11 13 15  1  3  5  7  9 11 13 15
```

The graph has two loops at the first and the last node. All paths in the De Bruijn graph are cycles, the graph is Hamiltonian.
Chapter 19: Searching paths in directed graphs

Graph =
  Node: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  Edge 0: 0 2 4 6 8 10 12 14 0 2 4 6 8 10 12 14
  Edge 1: 1 3 5 7 9 11 13 15 1 3 5 7 9 11 13 15

Paths            DBSs
0 1 2 4 9 3 6 13 10 5 11 7 15 14 12 8 .1...11.1.1111...0 1 2 4 9 3 6 13 10 5 11 7 15 14 12 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...
0 1 2 4 9 3 7 15 14 13 10 4 9 3 6 13 10 4 8 8 .1...11.1.1111...

n = 8 (ng=16)  #cycles = 16

Figure 19.2-B: Edges of the De Bruijn graph (top), and all paths starting at node 0 together with the corresponding the De Bruijn sequences (bottom).

With \( n \) a power of two the paths correspond to the De Bruijn sequences (DBS) of length \( 2n \). The graph has as many full paths as there are DBSs and the zeros/ones in the DBS correspond to even/odd values of the nodes, respectively. This is demonstrated in [FXT: graph/graph-debruijn-demo.cc] (shortened):

```c
ulong pq = 1; // whether and what to print with each cycle
ulong pfunc_db(digraph_paths &dp)
  // Function to be called with each cycle.
  {
    switch ( pq )
    {
      case 0: break; // just count
      case 1: // print lowest bits (De Bruijn sequence)
      {
        ulong *rv = dp.rv_, ng = dp.ng_;
        for (ulong k=0; k<ng; ++k) cout << (rv[k]&1UL ? '1' : '.');
        cout << endl;
        break;
      }
    } [--snip--]
  }
  return 1;
}
int main(int argc, char **argv)
{
  ulong n = 8;
  NXARG(pq, "what to do in pfunc()");
  ulong maxnp = 0;
  NXARG(maxnp, "stop after maxnp paths (0: never stop)"),
  ulong p0 = 0;
  NXARG(p0, "start position <2*n"),
  digraph dg = make_debruijn_digraph(n);
  digraph_paths dp(dg);
  dg.print_horiz("Graph =");
  // call pfunc() with each cycle:
  dp.all_paths(pfunc_db, 0, p0, maxnp);
  cout << "n = " << n;
  cout << " (ng=" << dg.ng_ << ")";
  cout << ": " #cycles = " << dp.cct_;
  cout << endl;
  return 0;
}
```

[fxtbook draft of 2008-August-17]
The macro \texttt{NXARG()} reads one argument, it is defined in [FXT: \texttt{nextarg.h}]. Figure \textbf{19.2-B} was created with the shown program.

The algorithm is a very effective way for generating all DBSs of a given length, the 67,108,864 DBSs of length 64 are generated in 140 seconds when printing is disabled (set argument \texttt{pq} to zero), corresponding to a rate of more than 450,000 DBSs per second.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig19.2-b.png}
\caption{A path in the De Bruijn graph with 64 nodes. Each binary word is printed vertically, the symbols ‘#’ and ‘-’ stand for one and zero, respectively.}
\end{figure}

Setting the argument \texttt{pq} to 4 prints the binary values of the successive nodes in the path horizontally, see figure \textbf{19.2-C}. The graph is constructed in a way that each word is the predecessor shifted by one with either zero or one inserted at position zero (top row of figure \textbf{19.2-C}).

The number of cycles in the De Bruijn graph equals the number of degree-\(n\) normal binary polynomials. An efficient procedure to compute it is given in section \textbf{40.6.3.5}. A closed form for the special case \(n = 2^k\) is given in section \textbf{39.5} on page \textbf{870}.

\subsection*{19.2.3 A modified De Bruijn graph: complement-shift sequences}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig19.2-d.png}
\caption{A path in the modified De Bruijn graph with 64 nodes. Each binary word is printed vertically, the symbols ‘#’ and ‘-’ stand for one and zero, respectively.}
\end{figure}

A modification of the De Bruijn graph forces the nodes to be the complement of its predecessor shifted by one (again with either zero or one inserted at position zero). The routine to set up the graph is [FXT: \texttt{make_complement_shift_digraph()} in \texttt{graph/mk-debruijn-digraph.cc}:}

\begin{verbatim}
digraph
make_complement_shift_digraph(ulong n)
{
ulong ng = 2*n, ne = 2*ng;
ulong *ep, *e;
digraph dg(ng, ne, ep, e);
ulong j = 0;
for (ulong k=0; k<ng; ++k) // for all nodes
{
ep[k] = j;
ulong r = (2*k) % ng;
e[j+1] = r; // connect node k to node (2*k) mod ng
r = (2*k+1) % ng;
e[j+1] = r; // connect node k to node (2*k+1) mod ng
}
ep[ng] = j;
// Here we have a De Bruijn graph.
for (ulong k=0, j=ng-1; k<j; ++k, --j) swap2(e[ep[k]], e[ep[j]]); // end with ones
for (ulong k=0, j=ng-1; k<j; ++k, --j) swap2(e[ep[k]+1], e[ep[j]+1]);
}
\end{verbatim}
Chapter 19: Searching paths in directed graphs

return dg;

The output of the program [FXT: graph/graph-complementshift-demo.cc] is shown in figure 19.2-D.

For \( n \) a power of two the sequence of binary words has the interesting property that the changes between successive words depend on their sequency: words with higher sequency change in less positions. Further, if two adjacent bits are set in some word then the next word never contains both bits again. Out of a run of \( k > 0 \) consecutive set bits in a word only one is contained in the next word.

See section 8.3 on page 207 for the connection with De Bruijn sequences.

### 19.3 Conditional search

Sometimes one wants to find paths that are subject to certain restrictions. Testing for each path found whether it has the desired property and discarding it if not is the most simple way. However, this will in many cases be extremely ineffective. An upper bound for the number of recursive calls of the search function `next_path()` with an graph with \( N \) nodes and an maximal number of \( v \) outgoing edges at each node is \( u = N^v \).

For example, the graph corresponding to Gray codes of \( n \)-bit binary words has \( N = 2^n \) nodes and (exactly) \( c = n \) outgoing edges at each node. The graph is the \( n \)-dimensional hypercube.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( N )</th>
<th>( u = N^v = N^n = 2^{n \cdot n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>65,536</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>33,554,432</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>68,719,476,736</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>562,949,953,421,312</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>18,446,744,073,709,551,616</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>2,417,851,639,229,258,349,412,352</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>1,267,650,600,228,229,401,496,703,205,376</td>
</tr>
</tbody>
</table>

To reduce the search space we use a function that rejects branches that would lead to a path not satisfying the imposed restrictions. A conditional search can be started via `all_cond_paths()` that has an additional function pointer `cfunc()` as argument that shall implement the condition. The corresponding method is declared as [FXT: graph/digraph-paths.h]:

```cpp
bool (*cfunc_)(digraph_paths &, ulong ns);
```

Besides the data from the digraph-class it needs the number of nodes seen so far (\( ns \)) as an argument. A slight modification of the search routine does what we want [FXT: graph/search-digraph-cond.cc]:

```cpp
void digraph_paths::next_cond_path(ulong ns, ulong p)
{
    [--snip--] // same as next_path()
    if ( ns==ng_ ) // all nodes seen ?
        [--snip--] // same as next_path()
    else
    {
        qq_[p] = 1; // mark position as seen (else loops lead to errors)
        ulong fe, en;
        g_.get_edge_idx(p, fe, en);
        ulong fct = 0; // count free reachable nodes
        for (ulong ep=fe; ep<en; ++ep)
        {
            ulong t = g_.e_[ep]; // next node
            if ( 0==qq_[t] ) // node free?
            {
```
rv_[ns] = t; // for cfunc()
if ( cfunc_(*this, ns) )
{
  ++fct;
  qq_[p] = fct; // mark position as seen: record turns
  next_cond_path(ns, t);
}
qq_[p] = 0; // unmark position

The free node under consideration is written to the end of the record of visited nodes so cfunc() does not need it as an explicit argument.

## 19.3.1 Modular adjacent changes (MAC) Gray codes

We search for Gray codes that have the modular adjacent changes (MAC) property: the values of successive elements of the delta sequence shall change by $\pm 1$ modulo $n$. Two examples are shown in figure 19.3-A. The sequence on the right side even has the stated property if the term ‘modular’ is omitted: It has the adjacent changes (AC) property.

As bit-wise cyclic shifts and reflections of MAC Gray codes are again MAC Gray codes we consider paths starting $0 \rightarrow 1 \rightarrow 2$ as canonical paths.

In the demo [FXT: graph/graph-macgray-demo.cc] the search is done as follows (shortened):

```c
int main(int argc, char **argv)
{
    ulong n = 5;
    NXARG(n, "size in bits");
    cf_nb = n;
    digraph dg = make_gray_digraph(n, 0);
    digraph_paths dp(dg);
    ulong ns = 0, p = 0;
    // MAC: canonical paths start as 0-->1-->3
    { dp.mark(0, ns);
      dp.mark(1, ns);
      p = 3;
    }
    dp.all_cond_paths(pfunc, cfunc_mac, ns, p, maxnp);
    return 0;
}
```

The function used to impose the MAC condition is:

```c
ulong cf_nb; // number of bits, set in main()
```

<table>
<thead>
<tr>
<th>0:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 19.3-A:** Two four-bit modular adjacent changes (MAC) Gray codes. Both are cycles.
Chapter 19: Searching paths in directed graphs

```cpp
// Condition: difference of successive delta values (modulo n) == +1
{
  // path initialized, we have ns>=2
  ulong p  = dp.rv_[ns], p1 = dp.rv_[ns-1], p2 = dp.rv_[ns-2];
  ulong c  = p ^ p1, c1 = p1 ^ p2;
  if ( c & bit_rotate_left(c1,1,cf_nb) ) return true;
  if ( c1 & bit_rotate_left(c,1,cf_nb) ) return true;
  return false;
}
```

One finds paths for \( n \leq 7 \) (\( n = 7 \) takes about 15 minutes). Whether MAC Gray codes exist for \( n \geq 8 \) is unknown (none is found with a 40 hour search).

### 19.3.2 Adjacent changes (AC) Gray codes

For AC paths one can only discard track-reflected solutions, the canonical paths are those where the delta sequence starts with a values that is \( \leq \lceil n/2 \rceil \). A function to impose the AC condition is

```cpp
ulong cf_mt; // mid track < cf_mt, set in main()
bool cfunc_ac(digraph_paths &dp, ulong ns)
// Condition: difference of successive delta values == +1
{
  if ( ns<2 ) return (dp.rv_[1] < cf_mt); // avoid track-reflected solutions
  ulong p  = dp.rv_[ns], p1 = dp.rv_[ns-1], p2 = dp.rv_[ns-2];
  ulong c  = p ^ p1, c1 = p1 ^ p2;
  if ( c & (c1<<1) ) return true;
  if ( c1 & (c<<1) ) return true;
  return false;
}
```

![Figure 19.3-B: Two five-bit adjacent changes (AC) Gray codes that are cycles.](image)

The program [FXT: graph/graph-acgray-demo.cc](#) allows searches for AC Gray codes. Two cycles for \( n = 5 \) are shown in figure 19.3-B. It turns out that such paths exist for \( n \leq 6 \) (the only path for \( n = 6 \) is shown in figure 19.3-C but there is no AC Gray code for \( n = 7 \):

```bash
time ./bin 7
arg 1: 7 == n [size in bits] default=5
arg 2: 0 == maxnp [ stop after maxnp paths (0: never stop)] default=0
```

[fxtbook draft of 2008-August-17]
1.3 Conditional search

Figure 19.3-C: The (essentially unique) AC Gray code for \( n = 6 \). While the path is a cycle in the graph the AC condition does not hold for the transition from the last to the first word.

% n = 7  #pftc = 0  
#paths = 0  #cycles = 0  
./bin 7  20.77  s user 0.11 s system 98% cpu 21.232 total  

Nothing is known about the case \( n \geq 8 \). For \( n = 8 \) no path is found within 15 days.

Inspection of the AC Gray codes for different values of \( n \) result in a hand-woven algorithm. The function [FXT: ac_gray_delta() in comb/acgray.cc] computes the delta sequence for an AC Gray codes for \( n \leq 6 \):

```c
void ac_gray_delta(uchar *d, ulong ldn)
// Generate a delta sequence for an adjacent-changes (AC) Gray code
// of length n=2**ldn where ldn<=6.
{
  if ( ldn<=2 ) // standard Gray code
    { 
      d[0] = 0;
      if ( ldn==2 ) { d[1] = 1; d[2] = 0; }
      return;
    }
  ac_gray_delta(d, ldn-1); // recursion
  ulong n = 1UL<<ldn;
  ulong nh = n/2;
  if ( 0==(ldn&1) )
    { 
      if ( ldn>=6 )
        { reverse(d, nh-1);
          for (ulong k=0; k<nh; ++k) d[k] = (ldn-2) - d[k];
        }
      for (ulong k=2; k<nh; ++k) d[k] = d[k-2];
      d[nh-1] = ldn - 1;
    }
  else
    { 
      reverse(d, nh-1);
      for (ulong k=0; k<nh; ++k) d[k] = (ldn-2) - d[k];
      d[nh-1] = ldn - 1;
    }
}
```
The Gray code is computed via

```c
void ac_gray(ulong *g, ulong ldn)
{
    ulong n = 1UL<<ldn;
    ALLOCA(uchar, d, n);
    ac_gray_delta(d, ldn);
    delta2gray(d, ldn, g);
}
```

where the routine `delta2gray()` is given in [FXT: comb/delta2gray.cc]:

```c
void delta2gray(const unsigned char *d, ulong ldn, ulong *g, ulong g0/*=0*/)
{
    g[0] = g0;
    ulong n = 1UL << ldn;
    for (ulong k=0; k<n-1; ++k) g[k+1] = g[k] ^ (1UL << d[k]);
}
```

The program [FXT: comb/acgray-demo.cc] can be used to create AC Gray codes for \(n \leq 6\). For \(n \geq 7\) the algorithm produces near-AC Gray codes, where the number of non-AC transitions equals \(2^n - 5 - 1\) for odd values of \(n\) and \(2^n - 5 - 2\) for \(n\) even:

<table>
<thead>
<tr>
<th># non-AC transitions:</th>
<th>(n = \ldots)</th>
<th>#non-ac</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n = 0, 6)</td>
<td>(= 0)</td>
<td>(= 0)</td>
</tr>
<tr>
<td>(n = 7)</td>
<td>(= 3)</td>
<td>(= 3)</td>
</tr>
<tr>
<td>(n = 8)</td>
<td>(= 6)</td>
<td>(= 6)</td>
</tr>
<tr>
<td>(n = 9)</td>
<td>(= 15)</td>
<td>(= 15)</td>
</tr>
<tr>
<td>(n = 10)</td>
<td>(= 30)</td>
<td>(= 30)</td>
</tr>
<tr>
<td>(n = 11)</td>
<td>(= 63)</td>
<td>(= 63)</td>
</tr>
<tr>
<td>(n = 12)</td>
<td>(= 126)</td>
<td>(= 126)</td>
</tr>
</tbody>
</table>

It seems likely that near-AC Gray codes with fewer non-AC transitions exist.

## 19.4 Edge sorting and lucky paths

The order of the nodes in the representation of the graph does not matter with finding paths as the algorithm does at no point refer to it. The order of the outgoing edges, however, does matter.

### 19.4.1 Edge sorting

Consider a large graph that has only a few paths. The calling tree of the recursive function `next_path()` obviously depends on the edge order. Thereby the first path can appear earlier or later in the search. ‘Later’ may well mean that the path is not found within any reasonable amount of time.

With a bit of luck one might find an ordering of the edges of the graph that will shorten the time span until the first path is found. The program [FXT: graph/graph-monotonicgray-demo.cc] searches for monotonic Gray codes and optionally sorts the edges of the graph. The method [FXT: digraph::sort_edges() in graph/digraph.cc] sorts the outgoing edges of each node according to a supplied comparison function.

The comparison function actually used imposes the lexicographic order shown in section 1.27 on page 73:

```c
int my_cmp(const ulong &a, const ulong &b)
{
    if ( a==b ) return 0;
#define CODE(x) lexrev2negidx(x);
    ulong ca = CODE(a);
    ulong cb = CODE(b);
    return (ca<cb ? +1 : -1);
}
```
The choice was inspired by the observation that the bit-wise difference of successive elements in bit-lex order is either one or three. We search until the first path for 8-bit words is found: for the unsorted graph this task takes 1.14 seconds, for the sorted it takes 0.03 seconds.

19.4.2 Lucky paths

<table>
<thead>
<tr>
<th>Step</th>
<th>Node</th>
<th>Next</th>
<th>Neighbors</th>
<th>Node</th>
<th>Next</th>
<th>Neighbors</th>
<th>Node</th>
<th>Next</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>15</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>111</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>12</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>13</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>13</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>14</td>
<td>9</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>3</td>
<td>[1.11 -]</td>
<td>15</td>
<td>9</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Path: #non-first-free turns = 2

Figure 19.4-A: A Gray code in the hypercube graph with randomized edge order (left) and the path description (right, see text).

The first Gray code found in the hypercube graph with randomized edge order is shown in figure 19.4-A (left). The corresponding path (as reported by [FXT: digraph_paths::print_turns() in graph/digraph_paths.cc]) is described in the right column. Here nn is the number of neighbors of node, xe is the index of the neighbor (next) in the list of edges of node. Finally xf is the index among the free nodes in the list. The latter corresponds to the value fct-1 in the function next_path() given in section 19.2 on page 383.

If xf equals zero at some step then the first free neighbor was visited next. If xf is nonzero then a dead end was reached in the course of the search and there was at least one U-turn. If the path is not the first found then the U-turn might well correspond to a previous path.

If there was no U-turn then the number of non-first-free turns is zero (the number is given as the last line of the report). If it is zero we call the path found a lucky path. For each given ordering of the edges and each starting position of the search there is at most one lucky path and if there is, it is the first path found.

When the first path is a lucky path then the search effectively ‘falls through’: the number of operations is a constant times the number of edges. That is, if a lucky path exists it is found almost immediately even for huge graphs.
19.5 Gray codes for Lyndon words

We search Gray codes for $n$-bit binary Lyndon words where $n$ is a prime. Here is a Gray code for the 5-bit Lyndon words that is a cycle:

```
. . . . .
. . . . .
. . . . .
```

An important application of such Gray codes is the construction of so-called single track Gray codes which can be obtained by appending rotated versions of the block. The following is a single track Gray code based on the block given. At each stage, the block is rotated by two positions (horizontal format):

```
######## --###- -####- ------ ---###-####- ------ ---###
-####- ------ ---###-####- ------ ---###-####- ------ ---###
-####- ------ ---###-####- ------ ---###-####- ------ ---###
-####- ------ ---###-####- ------ ---###-####- ------ ---###
```

The transition count (the number of zero-one transitions) is by construction the same for each track. The all-zero and the all-one words are missing in the Gray code, its length equals $2^n - 2$.

19.5.1 Graph search with edge sorting

Gray codes for the 7-bit binary Lyndon words like those shown in figure 19.5-A can easily be found by a graph search. In fact, all of them can be generated in short time: for $n = 7$ there are 395 Gray codes (starting with the word 0000...001) of which 112 are cycles.

The search for such a path for the next prime, $n = 11$, does not seem to give a result in reasonable time. If we do not insist on a Gray code through the cyclic minima but arbitrary rotations of the Lyndon words then more Gray codes exist. For that purpose nodes are declared adjacent if there is any cyclic rotation of the second node’s value that differs in exactly one bit to the first node’s value. The cyclic rotations can be recovered easily after a path is found. This is done in [FXT: graph/lyndon-gray-demo.cc] whose output is shown in figure 19.5-B. Still, already for $n = 11$ we do not get a result. As the corresponding graph has 186 nodes and 1954 edges, this is not a surprise.

Now we try edge sorting, we sort the edges according to the comparison function [FXT: graph/lyndon-cmp.cc]

```c
int lyndon_cmp0(const ulong &a, const ulong &b) {
    int bc = bit_count_cmp(a, b);
    if ( bc ) return -bc; // more bits first
    else
        if ( a==b ) return 0;
```
19.5: Gray codes for Lyndon words

<table>
<thead>
<tr>
<th>k</th>
<th>node</th>
<th>lyn_dec</th>
<th>lyn_bin</th>
<th>#rot</th>
<th>rot(lyn)</th>
<th>diff</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0]</td>
<td>1 ......1 0 ......1 ......1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[1]</td>
<td>3 ......11 0 ......11 ......1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[3]</td>
<td>7 ......111 0 ......111 ......1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[7]</td>
<td>15 ......1111 0 ......1111 ......1 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[13]</td>
<td>31 ......11111 0 ......11111 ......1 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>[17]</td>
<td>63 ......111111 0 ......111111 ......1 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[15]</td>
<td>47 ......111111 0 ......111111 ......1 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[10]</td>
<td>23 ......11111 0 ......11111 ......1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[16]</td>
<td>55 ......11111 0 ......11111 ......1 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>[11]</td>
<td>27 ......1111 0 ......1111 ......1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[5]</td>
<td>11 ......11 0 ......11 ......1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>[14]</td>
<td>43 ......111 0 ......111 ......1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>[6]</td>
<td>13 ......11 0 ......11 ......1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>[12]</td>
<td>29 ......111 0 ......111 ......1 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>[8]</td>
<td>19 ......11 0 ......11 ......1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>[4]</td>
<td>9 ......1 0 ......1 ......1 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>[9]</td>
<td>21 ......1 0 ......1 ......1 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>[2]</td>
<td>5 ......1 0 ......1 ......1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 19.5-B: A Gray code through the length-7 binary Lyndon words found by searching through all 7-bit binary words.

```cpp
8 return (a>b ? +1 : -1); // bigger numbers last
9 }
10 }
```

where `bit_count_cmp()` is defined in [FXT: bits/bitcount.h]:

```cpp
1 static inline int bit_count_cmp(const ulong &a, const ulong &b)
2 {
3 ulong ca = bit_count(a);
4 ulong cb = bit_count(b);
5 return (ca==cb ? 0 : (ca>cb ? +1 : -1));
6 }
```

We find a Gray code (which also is a cycle) for \(n=11\) immediately. Same for \(n=13\), again a cycle. The graph for \(n=13\) has 630 nodes and 8,056 edges, so finding a path is quite unexpected. The cycle found starts and ends as shown in figure 19.5-C.

For next candidate \((n=17)\) we do not find a Gray code within many hours of search. No surprise for a graph with 7,710 nodes and 130,828 edges.

We try another edge sorting scheme, an ordering based on the binary Gray code [FXT: graph/lyndon-cmp.cc]:

```cpp
1 int lyndon_cmp2(const ulong &a, const ulong &b)
2 {
3 if (a==b) return 0;
4 #define CODE(x) gray_code(x)
5 ulong ta = CODE(a), tb = CODE(b);
6 return (ta<tb ? +1 : -1);
7 }
```

There we go, we find a cycle for \(n=17\) and all smaller primes. All are cycles and all paths are lucky paths. The following edge sorting scheme also leads to Gray codes for all prime \(n\) where \(3 \leq n \leq 17\):

```cpp
1 int lyndon_cmp3(const ulong &a, const ulong &b)
2 {
3 if (a==b) return 0;
4 #define CODE(x) inverse_gray_code(x)
5 ulong ta = CODE(a), tb = CODE(b);
6 return (ta<tb ? +1 : -1);
7 }
```

Same for \(n=19\), the graph has 27,594 nodes and 523,978 edges. Indeed the sorting scheme leads to cycles for all odd \(n \leq 27!\). All these paths are lucky paths, a fact that we can exploit for an optimized search.
### 19.5.2 An optimized algorithm

With edge sorting functions that lead to a lucky path we can discard most of the data used with graph searching. We only need to keep track of whether a node has been visited so far. A tag-array ([FXT: ds/bitarray.h](https://fxtbook.de/ds/bitarray.h), see section 4.6 on page 158) suffices.

With \( n \)-bit Lyndon words the amount of tag-bits needed is \( 2^n \). Find an implementation of the algorithm as [FXT: class lyndon_gray in graph/lyndon-gray.h](https://fxtbook.de/graph/lyndon-gray.h).

If only the cyclical minima of the values are tagged then only \( 2^n/2 \) bits are needed if the access to the single necklace consisting of all ones is treated separately. This variant of the algorithm is activated by uncommenting the line `#define ALT_ALGORITHM`. Noting that the lowest bit in a necklace is always one we need only \( 2^n/4 \) bits: simply shift the words right by one before testing or writing to the tag array. This can be achieved by additionally uncommenting the line `#define ALTALT` in the file.

When a node is visited the algorithm creates a table of neighbors and selects the minimum among the free nodes with respect to the edge sorting function used. The table of neighbors is discarded then in order to minimize memory usage.

When no neighbor is found the number of nodes visited so far is returned. If this number equals the number of \( n \)-bit Lyndon words a lucky path was found. With composite \( n \) a Gray code for \( n \)-bit necklaces with the exception of the all-ones and the all-zeros word will be searched.

Four flavors of the algorithm have been found so far, corresponding to edge sorting with the 3rd, 5th,
19.5: Gray codes for Lyndon words

<table>
<thead>
<tr>
<th>n</th>
<th>number of nodes</th>
<th>tag-size</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>364,722</td>
<td>0.25 MB</td>
<td>1 sec</td>
</tr>
<tr>
<td>25</td>
<td>1,342,182</td>
<td>1 MB</td>
<td>3 sec</td>
</tr>
<tr>
<td>27</td>
<td>4,971,066</td>
<td>4 MB</td>
<td>12 sec</td>
</tr>
<tr>
<td>29</td>
<td>18,512,790</td>
<td>16 MB</td>
<td>1 min</td>
</tr>
<tr>
<td>31</td>
<td>69,273,666</td>
<td>64 MB</td>
<td>4 min</td>
</tr>
<tr>
<td>33</td>
<td>260,301,174</td>
<td>256 MB</td>
<td>16 min</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>number of nodes</th>
<th>tag-size</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>981,796,830</td>
<td>1 GB</td>
<td>1 h</td>
</tr>
<tr>
<td>37</td>
<td>3,714,566,310</td>
<td>4 GB</td>
<td>7 h</td>
</tr>
<tr>
<td>39</td>
<td>14,096,303,342</td>
<td>16 GB</td>
<td>2 d</td>
</tr>
<tr>
<td>41</td>
<td>53,634,713,550</td>
<td>64 GB</td>
<td>10 d</td>
</tr>
<tr>
<td>43</td>
<td>204,560,302,842</td>
<td>256 GB</td>
<td>&gt;40 d</td>
</tr>
<tr>
<td>45</td>
<td>781,874,934,568</td>
<td>1 TB</td>
<td>&gt;160 d</td>
</tr>
</tbody>
</table>

| Figure 19.5-D: Memory and (approximate) time needed for computing Gray codes with $n$-bit Lyndon words. The number of nodes equals the number of length-$n$ necklaces minus two. The size of the tag array equals $2^n/4$ bits or $2^n/32$ bytes.

21th, and 29th power of the Gray code. We refer to these functions as comparison functions 0, 1, 2, and 3, respectively. All of these lead to cycles for all primes $n \leq 31$. The resources needed with larger values of $n$ are shown in figure 19.5-D.

Using a 64-bit machine equipped with more than 4 Gigabyte of RAM it can be verified that three of the edge sorting functions lead to a Gray cycles also for $n = 37$, the 3rd power version fails. One of the sorting functions may lead to a Gray code for $n = 41$.

A program to compute the Gray codes is [FXT: graph/lyndon-gray-demo.cc], four arguments can be given:

```plaintext
arg 1: 13 == n  [ a prime < BITS_PER_LONG ]  default=17
arg 2: 1 == wh  [printing: 0==none, 1==>delta seq., 2==>full output]  default=1
arg 3: 3 == ncmp [use comparison function (0,1,2,3)]  default=2
arg 4: 0 == testall [special: test all odd values <= value]  default=0
```

An example with full output is given in figure 19.5-E. A 64-bit CRC (see section 39.3 on page 865) is computed from the delta sequence (rightmost column) and printed with the last word.

For larger $n$ one might want to print only the delta sequence, this is shown in figure 19.5-F. The CRC allows us to conveniently determine whether two delta sequences are different. Different sequences sometimes start identically.

For still larger values of $n$ even the delta sequence tends to get huge (for example, with $n = 37$ the sequence would be approximately 3.7 GB). One can suppress all output except for a progress indication,
Chapter 19: Searching paths in directed graphs

% ./bin 13 1 2 # 13 bits, delta seq. output, comparison function 2
n = 13 # lyn = 630
06B5748356465962546436734A76844A106C0145120825747A45247AC5B54567018A765467484A756454657CA1ACBC1C 856BA946489745654864565964521982515315BC82BC75BA023926253642674264275A4AC7B9761566C37412583758CA5624 B9C6AC6A87AC2CB4A8564534065A405822975697651563160204229078A3C1485C6505200109456B984A9B1B9AC0 A9A9B98A564856885765487458695670536054127531105458764574A56745470379A58680076985767697675 9A76576656657654657665851032A0AOA0BC8A0BC5234382385653252475763433253237235465764357373624636442 45235974234325365323432362352356252423258252644236823632463623584235283232323232323232323232
523242323323423642324235323523423
last = ............11 crc=568ab04b55aa2f
n = 13 # lyn = 630 # = 630

% ./bin 13 1 3 # 13 bits, delta seq. output, comparison function 3
n = 13 # lyn = 630
06B5748356465962546436734A76844A106C0145120825747A45247AC5B54567018A765467484A756454657CA1ACBC1C 856BA946489745654864565964521982515315BC82BC75BA023926253642674264275A4AC7B9761566C37412583758CA5624 B9C6AC6A87AC2CB4A8564534065A405822975697651563160204229078A3C1485C6505200109456B984A9B1B9AC0 A9A9B98A564856885765487458695670536054127531105458764574A56745470379A58680076985767697675 9A76576656657654657665851032A0AOA0BC8A0BC5234382385653252475763433253237235465764357373624636442 45235974234325365323432362352356252423258252644236823632463623584235283232323232323232323232
523242323323423642324235323523423
last = ............11 crc=745def277bfbed0
n = 13 # lyn = 630 # = 630

Figure 19.5-F: Delta sequences for two different Gray codes for 13-bit Lyndon words.

% ./bin 29 0 0 # 29 bits, output=progress, comparison function 0
n = 29 # lyn = 18512790
................. 1048576 ( 5.6606 % ) crc=ceabc5f2056b6e99
................. 2097152 ( 11.3281 % ) crc=7691c4f1a554b5d0
................. 3145728 ( 16.9922 % ) crc=6b39957f141f4fd
................. 4194304 ( 22.6563 % ) crc=5341f91f155d5c0
................. 5242880 ( 28.3203 % ) crc=45d45b193f8e666
................. 6291456 ( 33.9846 % ) crc=95a2c924f56e966
................. 7340032 ( 39.6484 % ) crc=000e1f5f0248e34
................. 8388608 ( 45.3125 % ) crc=23c74d3ea0c587
................. 9437184 ( 50.9766 % ) crc=896fd0c87dd04d3
................. 10485760 ( 56.4604 % ) crc=b00d8c8990f0c7f1
................. 11534336 ( 62.0304 % ) crc=d145f1b95b23eaab
................. 12582912 ( 67.9688 % ) crc=82971e2ed480350
................. 13631488 ( 73.6328 % ) crc=f249ad5b4fed252d
................. 14680064 ( 79.2969 % ) crc=909821d20c7246a98
................. 15728640 ( 84.9609 % ) crc=1c5d68e385b3ca
................. 16777216 ( 90.625 % ) crc=0e64f82c6fc79e1f
................. 17825792 ( 96.2991 % ) crc=62c17b9f3c644396
last = ............11 crc=5736fc9365da927e
n = 29 # lyn = 18512790 # = 18512790

Figure 19.5-G: Computation of a Gray code through the 29-bit Lyndon words. Most output is suppressed, only the CRC is printed at certain checkpoints.
as shown in figure 19.5-G. Here the CRC checksum is updated only with every (cyclically unadjusted) $2^{16}$-th Lyndon word.

Sometimes a Gray code through the necklaces (except for the all-zeros and all-ones words) is also found for composite $n$. Comparison functions 0, 1, and 2 lead to Gray codes (which are cycles) for all odd $n \leq 33$. Gray cycles are also found with comparison function 3, except for $n = 21, 27, \text{ and } 33$. All functions give Gray cycles also for $n = 4$ and $n = 6$. The values of $n$ for which no Gray code was found are the even values $\geq 8$.

### 19.5.3 No Gray codes for even $n \geq 8$

As the parity of the words in a Gray code sequence alternates between one and zero the difference between the numbers words of odd and even weight must be zero or one. If it is ones, no Gray cycle can exist because the parity of the first and last word is identical. We use the relations from section 17.3 on page 370.

For Lyndon words of odd length there are the same number of words for odd and even weight by symmetry, see figure 17.3-B on page 371. So a Gray code (and also a Gray cycle) can exist.

For even lengths the sequence of numbers of Lyndon words of odd and even weights start as:

- **n:** 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, ...
- **odd:** 1, 2, 5, 16, 51, 170, 585, 2048, 7280, 26214, 95325, 349520, ...
- **even:** 0, 1, 4, 14, 48, 165, 576, 2032, 7252, 26163, 95232, 349350, ...
- **diff:** 1, 1, 1, 1, 2, 3, 5, 9, 16, 28, 51, 93, 170, ...

The last row gives the differences, entry A000048 of [245]. All entries for $n \geq 8$ are greater than one, so no Gray code does exist.

For the number of necklaces we obtain, for $n = 2, 4, 6, \ldots$

- **n:** 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, ...
- **odd:** 1, 2, 6, 16, 52, 172, 586, 2048, 7286, 26216, 95326, 349536, ...
- **even:** 2, 4, 8, 20, 56, 180, 596, 2068, 7316, 26272, 95420, ...
- **diff:** 1, 2, 4, 4, 8, 10, 16, 20, 30, 56, 94, 180, ...

The (absolute) difference of both sequences is entry A000013 of [245]. We see that for $n \geq 4$ the numbers are greater than one, so no Gray code exists.

If we exclude the all-ones and all-zeros words, then the differences are

- **n:** 2, 4, 6, 8, 10, 12, 14, 16, 20, 22, 24, 26, ...
- **diff:** 1, 0, 2, 2, 6, 8, 18, 28, 54, 92, 178, 314, ...

And again, no Gray code exists for $n \geq 8$. That is, we have found Gray codes, and even cycles, for all computational feasible sizes were they can exist.
Chapter 19: Searching paths in directed graphs
Part III

Fast orthogonal transforms
Chapter 20
The Fourier transform

We introduce the discrete Fourier transform and give algorithms for its fast computation. Implementations and optimization considerations for complex and real valued transforms are given. The fast Fourier transforms (FFTs) are the basis of the algorithms for fast convolution (given in chapter 21) which are in turn the core of the fast high precision multiplication routines described in chapter 27. The number theoretic Fourier transforms (NTTs) are treated in chapter 25. Algorithms for Fourier transforms based on fast convolution are given in chapter 21, the transform for arbitrary length (Bluestein’s algorithm) in section 21.5, and prime-length transforms (Rader’s algorithm) in section 21.6.

20.1 The discrete Fourier transform

The discrete Fourier transform (DFT or simply FT) of a complex sequence \( a = [a_0, a_1, \ldots, a_{n-1}] \) of length \( n \) is the complex sequence \( c = [c_0, c_1, \ldots, c_{n-1}] \) defined by

\[
\begin{align*}
    c &= \mathcal{F}[a] \\
    c_k &= \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x z^{xk} \quad \text{where} \quad z = e^{2\pi i/n}
\end{align*}
\]

where \( z \) is a primitive \( n \)-th root of unity: \( z^n = 1 \) and \( z^j \neq 1 \) for \( 0 < j < n \).

Back-transform (or inverse discrete Fourier transform, IDFT or simply IFT) is then

\[
\begin{align*}
    a &= \mathcal{F}^{-1}[c] \\
    a_x &= \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k z^{-xk}
\end{align*}
\]

To see this, consider element \( y \) of the IFT of the FT of \( a \):

\[
\mathcal{F}^{-1}[\mathcal{F}[a]]_y = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} (a_x z^{xk}) z^{-yk} = \frac{1}{n} \sum_{x} a_x \sum_{k} (z^{x-y})^k
\]

Now \( \sum_{k} (z^{x-y})^k = n \) for \( x = y \) and zero else. This is because \( z \) is an \( n \)-th primitive root of unity: with \( x = y \) the sum consists of \( n \) times \( z^0 = 1 \), with \( x \neq y \) the summands lie on the unit circle (on the vertices
of an equilateral polygon with center zero) and add up to zero. Therefore the whole expression is equal to

\[
\frac{1}{n} \sum_x a_x \delta_{x,y} = a_y \text{ where } \delta_{x,y} := \begin{cases} 1 & (x = y) \\ 0 & (x \neq y) \end{cases}
\] (20.1-4a)

Here we will call the FT with the plus in the exponent the forward transform. The choice is actually arbitrary, engineers seem to prefer the minus for the forward transform, mathematicians the plus. The sign in the exponent is called the sign of the transform.

The FT is a linear transform. That is, for \( \alpha, \beta \in \mathbb{C} \)

\[
\mathcal{F} [ \alpha a + \beta b ] = \alpha \mathcal{F} [ a ] + \beta \mathcal{F} [ b ]
\] (20.1-5)

Further Parseval’s equation holds, the sum of squares of the absolute values is identical for a sequence and its Fourier transform:

\[
\sum_{x=0}^{n-1} |a_x|^2 = \sum_{k=0}^{n-1} |c_k|^2
\] (20.1-6)

A straightforward implementation of the discrete Fourier transform, that is, the computation of \( n \) sums each of length \( n \), requires \( \sim n^2 \) operations.

```c
void slow_ft(Complex *f, long n, int is)
{
    Complex h[n];
    const double ph0 = is*2.0*M_PI/n;
    for (long w=0; w<n; ++w)
    {
        Complex t = 0.0;
        for (long k=0; k<n; ++k)
        {
            t += f[k] * SinCos(ph0*k*w);
        }  
        h[w] = t;
    }
    copy(h, f, n);
}
```

This is [FXT: slow_ft() in fft/slowft.cc]. The variable `is` = \( \pm 1 \) is the sign of the transform, the function `SinCos(x)` returns a `Complex(cos(x), sin(x))`.

Note that the normalization factor \( \frac{1}{\sqrt{n}} \) in front of the FT sums has been left out. The inverse of the transform with one sign is the transform with the opposite sign followed by a multiplication of each element by \( \frac{1}{n} \). One has to keep in mind if that the sum of squares of the original sequence and its transform are equal up to a factor \( \frac{1}{\sqrt{n}} \).

A fast Fourier transform (FFT) algorithm is an algorithm that improves the operation count to proportional \( n \sum_{k=1}^{m} (p_k - 1) \), where \( n = p_1 p_2 \cdots p_m \) is a factorization of \( n \). In case of a power \( n = p^m \) the value computes to \( n (p - 1) \log_p (n) \). In the special case \( p = 2 \) even \( n/2 \log_2(n) \) (complex) multiplications suffice. There are several different FFT algorithms with many variants.

### 20.2 Summary of definitions of Fourier transforms *

We summarize the definitions of the continuous, semi-continuous and the discrete Fourier transform.
The continuous Fourier transform

The (continuous) Fourier transform (FT) of a function \( f : \mathbb{C}^n \to \mathbb{C}^n, \quad \vec{x} \mapsto f(\vec{x}) \) is defined by

\[
F(\vec{\omega}) := \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{C}^n} f(\vec{x}) e^{+\sigma i \vec{\omega} \cdot \vec{x}} d^n x
\]  

(20.2-1)

where \( \sigma = \pm 1 \) is the sign of the transform. The FT is a unitary transform. Its inverse is the complex conjugate transform

\[
f(\vec{x}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{C}^n} F(\vec{\omega}) e^{-\sigma i \vec{\omega} \cdot \vec{x}} d^n \omega
\]  

(20.2-2)

For the 1-dimensional case one has

\[
F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x) e^{+\sigma i \omega x} dx
\]  

(20.2-3)

\[
f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} F(\omega) e^{-\sigma i \omega x} d\omega
\]  

(20.2-4)

The ‘frequency-form’ is

\[
F(\nu) = \int_{-\infty}^{+\infty} f(x) e^{+2\pi \sigma \nu x} dx
\]  

(20.2-5)

\[
f(x) = \int_{-\infty}^{+\infty} F(\nu) e^{-2\pi \sigma \nu x} d\nu
\]  

(20.2-6)

The semi-continuous Fourier transform

For periodic functions defined on a interval \( L \in \mathbb{R}, f : L \to \mathbb{R}, x \mapsto f(x) \) one defines the semi-continuous Fourier transform as:

\[
c_k := \frac{1}{\sqrt{L}} \int_L f(x) e^{+\sigma 2\pi i k x/L} dx
\]  

(20.2-7)

The inverse transform is an infinite sum with the property

\[
\frac{1}{\sqrt{L}} \sum_{k=-\infty}^{k=+\infty} c_k e^{-\sigma 2\pi i k x/L} = \begin{cases} 
\frac{f(x)}{2} & \text{if } f \text{ continuous at } x \\
\frac{f(x+0)+f(x-0)}{2} & \text{else}
\end{cases}
\]  

(20.2-8)

An equivalent form of the semi-continuous Fourier transform is given by

\[
a_k := \frac{1}{\sqrt{L}} \int_L f(x) \cos \frac{2\pi k x}{L} dx, \quad k = 0, 1, 2, \ldots
\]  

(20.2-9a)

\[
b_k := \frac{1}{\sqrt{L}} \int_L f(x) \sin \frac{2\pi k x}{L} dx, \quad k = 1, 2, \ldots
\]  

(20.2-9b)

\[
f(x) = \frac{1}{\sqrt{L}} \left[ a_0 + \sum_{k=1}^{\infty} \left( a_k \cos \frac{2\pi k x}{L} + b_k \sin \frac{2\pi k x}{L} \right) \right]
\]  

(20.2-9c)

The connection between the two forms is given by:

\[
c_k = \begin{cases} 
a_0 & (k = 0) \\
\frac{1}{2}(a_k - ib_k) & (k > 0) \\
\frac{1}{2}(a_k + ib_k) & (k < 0)
\end{cases}
\]  

(20.2-10)
The discrete Fourier transform

The discrete Fourier transform (DFT) of a sequence \( f \) of length \( n \) with elements \( f_x \) is defined by

\[
c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} f_x e^{\sigma \frac{2\pi i k}{n}}
\]

(20.2-11)

The inverse transform is

\[
f_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k e^{-\sigma \frac{2\pi i k}{n}}
\]

(20.2-12)

Some sources define \( c_k := \sum \ldots \) and \( f_x = \frac{1}{n} \sum \ldots \). With this definition the transform does not preserve norms but there is a computational saving when the normalization does not matter. When a transform and its inverse are applied in succession (as with convolution algorithms) then only one normalization (factors \( \frac{1}{n} \)) is needed instead of two (factors \( \frac{1}{\sqrt{n}} \)). In the implementations of the transform we will in general omit the normalization altogether and leave it to the user to apply it where needed.

### 20.3 Radix-2 FFT algorithms

#### A little bit of notation

In what follows let \( a \) be a length-\( n \) sequence with \( n \) a power of two.

- Let \( a^{(\text{even})} \) and \( a^{(\text{odd})} \) denote the length-\( n/2 \) subsequences of those elements of \( a \) that have even and odd indices, respectively. That is, \( a^{(\text{even})} = [a_0, a_2, a_4, a_6, \ldots, a_{n-2}] \) and \( a^{(\text{odd})} = [a_1, a_3, \ldots, a_{n-1}] \).
- Let \( a^{(\text{left})} \) and \( a^{(\text{right})} \) denote the left and right subsequences, respectively. That is, \( a^{(\text{left})} = [a_0, a_1, \ldots, a_{n/2-1}] \) and \( a^{(\text{right})} = [a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}] \).
- Let \( c = S^k a \) denote the sequence with elements \( c_x = a_x e^{\sigma \frac{2\pi i x}{n}} \) where \( \sigma = \pm 1 \) is the sign of the transform. The symbol \( S \) shall suggest a shift operator. With radix-2 FFT algorithms only \( S^{1/2} \) is needed.
- In relations between sequences we sometimes emphasize the length of the sequences on both sides as in \( a^{(\text{even})} n/2 b^{(\text{odd})} + c^{(\text{odd})} \). In these relations the operators plus and minus are to be understood as element-wise.

#### 20.3.1 Decimation in time (DIT) FFT

The following observation is the key to the (radix-2) decimation in time (DIT) FFT algorithm, also called Cooley-Tukey FFT algorithm: For even values of \( n \) the \( k \)-th element of the Fourier transform is

\[
\mathcal{F}[a]_k = \sum_{x=0}^{n-1} a_x z^{xk} = \sum_{x=0}^{n/2-1} a_{2x} z^{2xk} + \sum_{x=0}^{n/2-1} a_{2x+1} z^{(2x+1)k}
\]

(20.3-1a)

\[
= \sum_{x=0}^{n/2-1} a_{2x} z^{2xk} + z^k \sum_{x=0}^{n/2-1} a_{2x+1} z^{2xk}
\]

(20.3-1b)

where \( z = e^{\sigma \frac{i 2\pi}{n}} \), \( \sigma = \pm 1 \) is the sign of the transform, and \( k \in \{0, 1, \ldots, n-1\} \).

The identity tells us how to compute the \( k \)-th element of the length-\( n \) Fourier transform from the length-\( n/2 \) Fourier transforms of the even and odd indexed subsequences.
20.3: Radix-2 FFT algorithms

To actually rewrite the length-$n$ FT in terms of length-$n/2$ FTs one has to distinguish whether $0 \leq k < n/2$ or $n/2 \leq k < n$. In the expressions we rewrite $k \in \{0, 1, 2, \ldots, n-1\}$ as $k = j + \delta \frac{n}{2}$ where $j \in \{0, 1, 2, \ldots, n/2 - 1\}$ and $\delta \in \{0, 1\}$:

\[
\sum_{x=0}^{n-1} a_x z^x (j + \delta \frac{n}{2}) = \sum_{x=0}^{n/2-1} a_x^{(\text{even})} z^x j + \delta \frac{n}{2} + \sum_{x=0}^{n/2-1} a_x^{(\text{odd})} z^x j (j + \delta \frac{n}{2}) \quad (20.3-2a)
\]

\[
= \begin{cases} 
\sum_{x=0}^{n/2-1} a_x^{(\text{even})} z^x j + \sum_{x=0}^{n/2-1} a_x^{(\text{odd})} z^x j \quad & \text{for } \delta = 0 \\
\sum_{x=0}^{n/2-1} a_x^{(\text{even})} z^x j - \sum_{x=0}^{n/2-1} a_x^{(\text{odd})} z^x j \quad & \text{for } \delta = 1
\end{cases} \quad (20.3-2b)
\]

The minus sign in the relation for $\delta = 1$ is due to the fact that $z^{j+1} = z^j z^{n/2} = -z^j$.

Observing that $z^2$ is just the root of unity that appears in a length-$n/2$ transform one can rewrite the last two equations to obtain the radix-2 DIT FFT step:

\[
\mathcal{F}[a]^{(\text{left})} \underset{n/2}{=} \mathcal{F}[a^{(\text{even})}] + S^{1/2} \mathcal{F}[a^{(\text{odd})}] \quad (20.3-3a)
\]

\[
\mathcal{F}[a]^{(\text{right})} \underset{n/2}{=} \mathcal{F}[a^{(\text{even})}] - S^{1/2} \mathcal{F}[a^{(\text{odd})}] \quad (20.3-3b)
\]

The length-$n$ transform has been replaced by two transforms of length $n/2$. If $n$ is a power of 2 this scheme can be applied recursively until length-one transforms are reached which are identity (‘do nothing’) operations.

Thereby the operation count is improved to proportional $n \cdot \log_2(n)$: there are $\log_2(n)$ splitting steps, the work in each step is proportional to $n$.

Note that the operator $S$ depends on the sign of the transform.

20.3.1.1 Recursive implementation

A recursive implementation of radix-2 DIT FFT given as pseudo code (C++ version in [FXT: fft/recfft2.cc]) is

```cpp
procedure rec_fft_dit2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1] // workspace
    complex s[0..n/2-1], t[0..n/2-1] // workspace

    if n == 1 then // end of recursion
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1 // copy to workspace
        { s[k] := a[2*k] // even indexed elements
          t[k] := a[2*k+1] // odd indexed elements
        }

    // recursion: call two half-length FFTs:
    rec_fft_dit2(s[], nh, b[], is)
    rec_fft_dit2(t[], nh, c[], is)

    fourier_shift(c[], nh, is*1/2)
}
```

[fxtbook draft of 2008-August-17]
The parameter \( i \sigma = \sigma = \pm 1 \) is the sign of the transform. The data length \( n \) must be a power of 2. The result is returned in the array \( x[] \). Note that normalization (multiplication of each element of \( x[] \) by \( 1/\sqrt{n} \)) is not included here.

The procedure uses the subroutine \texttt{fourier\_shift()} which modifies the array \( c[] \) according to the operation \( S^v \): each element \( c[k] \) is multiplied by \( e^{v \cdot 2\pi i k/n} \). It is called with \( v = \pm 1/2 \) for the Fourier transform. The pseudo code (C++ equivalent in [FXT: fft/fouriershift.cc]) is

\begin{verbatim}
procedure fourier_shift(c[], n, v)
  for k:=0 to n-1 // copy back from workspace
    \{ x[k] := b[k] + c[k]  \\
    x[k+nh] := b[k] - c[k]  \\
    \}
end

end
\end{verbatim}

This version of a non-recursive FFT procedure already avoids the calling overhead and it works in-place. But it is a bit wasteful. The (expensive) computation \( e := \exp(i \sigma \cdot 2\pi \cdot 1 \cdot j/n) \) is done \( n/2 \cdot \log_2(n) \) times.
20.3.1.3 Saving trigonometric computations

To reduce the number of sine and cosine computations, one can simply swap the two inner loops, leading to the first ‘real world’ FFT procedure presented here. Pseudo code for a non-recursive procedure of the radix-2 DIT algorithm (C++ version in [FXT: fft/fftdit2.cc]):

```c
procedure fft_dit2(a[], ldn, is)
    // complex a[0..2**ldn-1] input, result
    n := 2**ldn
    revbin_permute(a[], n)
    for ldm:=1 to ldn // log_2(n) iterations
    {
        m := 2**ldm
        mh := m/2
        for j:=0 to mh-1 // m/2 iterations
        {
            e := exp(is*2*PI*I*j/m) // 1 + 2 + ... + n/8 + n/4 + n/2 == n-1 computations
            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh] * e
                a[r+j] := u + v
                a[r+j+mh] := u - v
            }
        }
    }
```

Swapping the two inner loops reduces the number of trigonometric computations to n but leads to a feature that many FFT implementations share: memory access is highly nonlocal. For each recursion stage (value of ldm) the array is traversed mh times with n/m accesses in strides of mh. As mh is a power of two this can (on computers that use memory cache) have a very negative performance impact for large values of n. With computers where the memory access is very slow compared to the CPU the naive version can actually be faster.

It is a good idea to extract the ldm==1 stage of the outermost loop, this avoids complex multiplications with the trivial factors 1 + 0 i (and the computations of these quantities as trigonometric functions). Replace the line

for ldm:=1 to ldn

by

```c
for r:=0 to n-1 step 2
{
    {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
}
```

for ldm:=2 to ldn

20.3.2 Decimation in frequency (DIF) FFT

Splitting of the Fourier sum into a left and right half leads to the decimation in frequency (DIF) FFT algorithm, also called Sande-Tukey FFT algorithm. For even values of n the k-th element of the Fourier
Chapter 20: The Fourier transform

The Fourier transform is

\[
F[a]_k = \sum_{x=0}^{n-1} a_x z^{xk} = \sum_{x=0}^{n/2-1} a_x z^{xk} + \sum_{x=n/2}^{n-1} a_x z^{xk} = \sum_{x=0}^{n/2-1} a_x z^{xk} + \sum_{x=0}^{n/2-1} a_{x+n/2} z^{(x+n/2)k} = \sum_{x=0}^{n/2-1} (a_x^{left} + z^{kn/2} a_x^{right}) z^{xk}
\]  

(20.3-4a)

where \( z = e^{\sigma i 2\pi/n} \), \( \sigma = \pm 1 \) is the sign of the transform, and \( k \in \{0, 1, \ldots, n - 1\} \).

Here one has to distinguish whether \( k \) is even or odd. Therefore we rewrite \( k \in \{0, 1, 2, \ldots, n - 1\} \) as \( k = 2j + \delta \) where \( j \in \{0, 1, 2, \ldots, n/2 - 1\} \) and \( \delta \in \{0, 1\} \):

\[
\sum_{x=0}^{n-1} a_x z^{(2j+\delta)k} = \sum_{x=0}^{n/2-1} (a_x^{left} + z^{(2j+\delta)n/2} a_x^{right}) z^{(2j+\delta)k} = \begin{cases} 
\sum_{x=0}^{n/2-1} (a_x^{left} + a_x^{right}) z^{2xj} & \text{for } \delta = 0 \\
\sum_{x=0}^{n/2-1} z^x (a_x^{left} - a_x^{right}) z^{2xj} & \text{for } \delta = 1 
\end{cases}
\]  

(20.3-5a)

\( z^{(2j+\delta)n/2} = e^{\pm \pi i \delta} \) is equal to plus or minus one for \( \delta = 0 \) or \( \delta = 1 \) corresponding to \( k \) even or odd. The last two equations are, more compactly written, the key to the radix-2 DIF FFT step:

\[
F[a]^{(even)} \overset{n/2}{=} F[a^{left}] + a^{right} \\
F[a]^{(odd)} \overset{n/2}{=} F[S^{1/2} \left( a^{left} - a^{right} \right)]
\]  

(20.3-6a)

A recursive implementation of radix-2 DIF FFT given as pseudo code (C++ version given in [FXT: fft/recfft2.cc]) is

```c++
1 procedure rec_fft_dif2(a[], n, x[], is)
2 // complex a[0..n-1] input
3 // complex x[0..n-1] result
4 {
5   complex b[0..n/2-1], c[0..n/2-1] // workspace
6   complex s[0..n/2-1], t[0..n/2-1] // workspace
7   if n == 1 then
8     x[0] := a[0]
9     return
10 }  
11 nh := n/2
12 for k:=0 to nh-1
13   {  
14     s[k] := a[k] // 'left' elements
15     t[k] := a[k+nh] // 'right' elements
16   }
17 for k:=0 to nh-1
18   {  
19     {s[k], t[k]} := {(s[k]+t[k]), (s[k]-t[k])}
20   }
21 fourier_shift(t[], nh, is*0.5)
22 rec_fft_dif2(s[], nh, b[], is)
23 rec_fft_dif2(t[], nh, c[], is)
```
The parameter $\text{is} = \sigma = \pm 1$ is the sign of the transform. The data length $n$ must be a power of 2. The result is returned in the array $x[]$. Again, the routine does no normalization.

Pseudo code for a non-recursive procedure (a C++ implementation is given in [FXT: fft/ftdif2.cc]):

```plaintext
procedure fft_dif2(a[], ldn, is)
  // complex a[0..2**ldn-1] input, result
  {
    n := 2**ldn
    for ldm:=ldn to 1 step -1
    {
      m := 2**ldm
      mh := m/2
      for j:=0 to mh-1
      {
        e := exp(is*2*PI*I*j/m)
        for r:=0 to n-m step m
        {
          u := a[r+j]
          v := a[r+j+mh]
          a[r+j] := (u + v)
          a[r+j+mh] := (u - v) * e
        }
      }
    }
    revbin_permute(a[], n)
  }
```

In DIF FFTs the `revbin_permute()`-procedure is called after the main loop, in the DIT code it is called before the main loop. As in the procedure for the DIT FFT (section 20.3.1.3 on page 409) the inner loops were swapped to save trigonometric computations.

Extracting the $ldm==1$ stage of the outermost loop is again a good idea. Replace the line

```
for ldm:=ldn to 1 step -1
```

by

```
for ldm:=ldn to 2 step -1
```

and insert

```
for r:=0 to n-1 step 2
{
  {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
}
```

before the call of `revbin_permute(a[], n)`.

## 20.4 Saving trigonometric computations

The sine and cosine computations are an expensive part of any FFT. There are two apparent ways for saving the involved CPU cycles, the use of lookup-tables and recursive methods. The CORDIC algorithms for sine and cosine given in section 33.2.1 on page 660 can be useful when implementing FFTs in hardware.
20.4.1 Using lookup tables

The idea is to precompute all necessary values, store them in an array, and later looking up the values needed. This is a good idea if one wants to compute many FFTs of the same (small) length. For FFTs of large sequences one gets large lookup tables that can introduce a high cache-miss rate. Thereby one is likely experiencing little or no speed gain, even a notable slowdown is possible. However, for a length-n FFT one does not need to store all the \((n\) complex or \(2n\) real) sine/cosine values \(\exp(2\pi i k/n) = \cos(2\pi k/n) + i \sin(2\pi k/n)\) where \(k = 0, 1, 2, 3, \ldots, n-1\). For the lookups one can use the symmetry relations

\[
\begin{align*}
\cos(\pi + x) & = -\cos(x) \\
\sin(\pi + x) & = -\sin(x)
\end{align*}
\] (20.4-1a)

To reduce the interval from \(0 \ldots 2\pi\) to \(0 \ldots \pi\). Exploiting the relations

\[
\begin{align*}
\cos(\pi/2 + x) & = -\sin(x) \\
\sin(\pi/2 + x) & = +\cos(x)
\end{align*}
\] (20.4-2a)

further reduces the interval to \(0 \ldots \pi/2\). Finally, the relation

\[
\sin(x) = \cos(\pi/2 - x)
\] (20.4-3)

shows that only the table of cosines is needed. That is, already a table of the \(n/4\) real values \(\cos(2\pi i k/n)\) for \(k = 0, 1, 2, 3, \ldots, n/4 - 1\) suffices for a length-\(n\) FFT computation. The size of the table is thereby cut by a factor of 8. Possible cache problems can sometimes be mitigated by simply storing the trigonometric values in reversed order which can avoid many equidistant memory accesses.

20.4.2 Recursive generation

In FFT computations one typically needs the values

\[
\begin{bmatrix}
\exp(i \varphi 0) = 1, & \exp(i \varphi \gamma), & \exp(i \varphi 2\gamma), & \exp(i \varphi 3\gamma), & \ldots
\end{bmatrix}
\]

where \(\varphi \in \mathbb{R}\) in sequence. The naive idea for a recursive computation of these values is to precompute \(d = \exp(i \varphi \gamma)\) and then compute the following value using the identity \(\exp (i \varphi k \gamma) = d \cdot \exp (i \varphi (k - 1) \gamma)\). This method, however, is of no practical value because the numerical error grows exponentially in the process.

A stable version of a trigonometric recursion for the computation of the sequence can be stated as follows. Precompute

\[
\begin{align*}
c & = \cos \varphi, \\
s & = \sin \varphi, \\
\alpha & = 1 - \cos \gamma \quad \text{[Cancellation!]} \\
& = 2 \left(\sin \frac{\gamma}{2}\right)^2 \quad \text{[OK.]} \\
\beta & = \sin \gamma
\end{align*}
\] (20.4-4a-d)

Then compute the next pair \((c_+, s_+)\) from the previous one \((c, s)\) via

\[
\begin{align*}
c_+ & = c - (\alpha c + \beta s); \\
s_+ & = s - (\alpha s - \beta c); \\
\end{align*}
\] (20.4-5a-b)

The underlying idea is to use the relation \(E(\varphi + \gamma) = E(\varphi) - E(\varphi) \cdot z\) where \(E(x) := \exp(2\pi i x)\). This leads to \(z = 1 - \cos \gamma - i \sin \gamma = 2 \left(\sin \frac{\gamma}{2}\right)^2 - i \sin \gamma.\)
Do not expect to get all the precision you would get with calls of the sine and cosine functions, but even for very long FFTs less than 3 bits of precision are lost. When working with (C-type) doubles it might be a good idea to use the type long double with the trigonometric recursion: the generated values will then always be accurate within the double-precision, provided long doubles are actually more precise than doubles. With high precision multiplication routines (that is, with exact integer convolution) this can be mandatory.

A real-world example from [FXT: hftdif_core() in hft/hftdif.cc]:

```c
--snip--

double tt = M_PI_4/kh; // the angle increment
double s1 = 0.0, c1 = 1.0; // start at angle zero
double al = sin(0.5*tt);
al *= (2.0*al);
double be = sin(tt);

for (ulong i=1; i<kh; i++)
{
    double t1 = c1;
    c1 -= (al*t1+be*s1);
    s1 -= (al*s1-be*t1);
    // here c1 = cos(tt*i) and s1 = sin(tt*i)
}

--snip--
```

The variable tt equals γ in relations 20.4-4d and 20.4-4e.

### 20.5 Higher radix FFT algorithms

Higher radix FFT algorithms save trigonometric computations. The radix-4 FFT algorithms presented in what follows replace all multiplications with complex factors (0, ±i) by the obvious simpler operations. Radix-8 algorithms also simplify the special cases where the sines and cosines equal ±√1/2.

Further the bookkeeping overhead is reduced due to the more unrolled structure. Moreover, the number of loads and stores is reduced.

**More notation**

Let a be a length-n sequence where n is a multiple of m.

- Let \( a^{(r\%m)} \) denote the subsequence of the elements with index \( x \) where \( x \equiv r \mod m \). For example, \( a^{(0\%2)} = a^{(\text{even})} \) and \( a^{(3\%4)} = [a_3, a_7, a_{11}, a_{15}, \ldots] \). The length of \( a^{(r\%m)} \) is \( n/m \).

- Let \( a^{(r/m)} \) denote the subsequence of elements with indices \( \lfloor \frac{r}{m} \rfloor \cdot \frac{n}{m}, \ldots, \lfloor \frac{r+1}{m} \rfloor \cdot \frac{n}{m} - 1 \). For example \( a^{(1/2)} = a^{(\text{right})} \) and \( a^{(2/3)} \) is the last third of \( a \). The length of \( a^{(r/m)} \) is also \( n/m \).

#### 20.5.1 Decimation in time algorithms

First rewrite the radix-2 DIT step (relations 20.3-3a and 20.3-3b on page 407) in the new notation:

\[
\mathcal{F}[a]^{(0/2)} \overset{n/2}{=} \mathcal{S}^{0/2} \mathcal{F}[a^{(0\%2)}] + \mathcal{S}^{1/2} \mathcal{F}[a^{(1\%2)}] \tag{20.5-1a}
\]

\[
\mathcal{F}[a]^{(1/2)} \overset{n/2}{=} \mathcal{S}^{0/2} \mathcal{F}[a^{(0\%2)}] - \mathcal{S}^{1/2} \mathcal{F}[a^{(1\%2)}] \tag{20.5-1b}
\]

The operator \( \mathcal{S} \) is defined on section 20.3 on page 406, note that \( \mathcal{S}^{0/2} = \mathcal{S}^0 \) is the identity operator.

The derivation of the radix-4 step is analogous to the radix-2 step, it just involves more writing and does not give additional insights. So we just state the radix-4 DIT FFT step which can be applied when \( n \) is
Chapter 20: The Fourier transform

divisible by 4:

\[
F[a](0/4) = \frac{n}{4} + S0/4F[a(0/4)] + S1/4F[a(1/4)] + S2/4F[a(2/4)] + S3/4F[a(3/4)] \quad (20.5-2a)
\]

\[
F[a](1/4) = \frac{n}{4} + S0/4F[a(0/4)] + i\sigma S1/4F[a(1/4)] - S2/4F[a(2/4)] - i\sigma S3/4F[a(3/4)] \quad (20.5-2b)
\]

\[
F[a](2/4) = \frac{n}{4} + S0/4F[a(0/4)] - S1/4F[a(1/4)] + S2/4F[a(2/4)] - S3/4F[a(3/4)] \quad (20.5-2c)
\]

\[
F[a](3/4) = \frac{n}{4} + S0/4F[a(0/4)] - i\sigma S1/4F[a(1/4)] - S2/4F[a(2/4)] + i\sigma S3/4F[a(3/4)] \quad (20.5-2d)
\]

In contrast to the radix-2 step that happens to be identical for forward and backward transform the sign of the transform \(\sigma = \pm 1\) appears explicitly. The relations, written more compactly:

\[
F[a](j/4) = \frac{n}{4} + e^{\sigma 2\pi i j/4}S0/4F[a(0/4)] + e^{\sigma 2\pi i j/4}S1/4F[a(1/4)] + e^{\sigma 2\pi i j/4}S2/4F[a(2/4)] + e^{\sigma 2\pi i j/4}S3/4F[a(3/4)] \quad (20.5-3)
\]

where \(j \in \{0, 1, 2, 3\}\) and \(n\) is a multiple of 4. Still more compactly:

\[
F[a](j/4) = \frac{n}{4} \sum_{k=0}^{3} e^{\sigma 2\pi i k j/4} S^{k/4}F[a(k/4)] \quad j \in \{0, 1, 2, 3\} \quad (20.5-4)
\]

where the summation symbol denotes \(element-wise\) summation of the sequences. The dot indicates multiplication of all elements of the sequence by the exponential.

The general \(radix-r\) \(DIT\) \(FFT\) step, applicable when \(n\) is a multiple of \(r\), is:

\[
F[a](j/r) = \frac{n}{r} \sum_{k=0}^{r-1} e^{\sigma 2\pi i k j/r} S^{k/r}F[a(k/r)] \quad j = 0, 1, 2, \ldots, r - 1 \quad (20.5-5)
\]

Our notation turned out to be useful indeed.

20.5.2 Decimation in frequency algorithms

The radix-2 DIF step (relations \textcolor{red}{20.3-6a} and \textcolor{red}{20.3-6b} on page 410), in the new notation:

\[
F[a](0/2) = \frac{n}{2} \quad F[S^{0/2}(a(0/2) + a(1/2))] \quad (20.5-6a)
\]

\[
F[a](1/2) = \frac{n}{2} \quad F[S^{1/2}(a(0/2) - a(1/2))] \quad (20.5-6b)
\]

The radix-4 DIF \(FFT\) step, applicable for \(n\) divisible by 4, is

\[
F[a](0/4) = \frac{n}{4} \quad F[S^{0/4}(a(0/4) + a(1/4) + a(2/4) + a(3/4))] \quad (20.5-7a)
\]

\[
F[a](1/4) = \frac{n}{4} \quad F[S^{1/4}(a(0/4) + i\sigma a(1/4) - a(2/4) - i\sigma a(3/4))] \quad (20.5-7b)
\]

\[
F[a](2/4) = \frac{n}{4} \quad F[S^{2/4}(a(0/4) - a(1/4) + a(2/4) - a(3/4))] \quad (20.5-7c)
\]

\[
F[a](3/4) = \frac{n}{4} \quad F[S^{3/4}(a(0/4) - i\sigma a(1/4) - a(2/4) + i\sigma a(3/4))] \quad (20.5-7d)
\]

Again, \(\sigma = \pm 1\) is the sign of the transform. More compactly:

\[
F[a](j/4) = \frac{n}{4} \quad F[S^{j/4}(\sum_{k=0}^{3} e^{\sigma 2\pi i k j/4} a(k/4))] \quad j \in \{0, 1, 2, 3\} \quad (20.5-8)
\]

The general \(radix-r\) \(DIF\) \(FFT\) step is

\[
F[a](j/r) = \frac{n}{r} \quad F[S^{j/r}(\sum_{k=0}^{r-1} e^{\sigma 2\pi i k j/r} a(k/r))] \quad j \in \{0, 1, 2, \ldots, r - 1\} \quad (20.5-9)
\]
20.5.3 Implementation of radix-$r$ FFTs

For the implementation of a radix-$r$ FFT with $r \neq 2$ the `revbin_permute` routine has to be replaced by its radix-$r$ version `radix_permute`. The reordering now swaps elements $a_x$ with $a_{\bar{x}}$ where $\bar{x}$ is obtained from $x$ by reversing its radix-$r$ expansion (see section 2.2 on page 95). In most practical cases one considers $r = p^x$ where $p$ is prime. Pseudo code for a radix $r = p^x$ DIT FFT:

```plaintext
procedure fftdit_r(a[], n, is)
  // complex a[0..n-1] input, result
  // p (hard-coded)
  // r == power of p (hard-coded)
  // n == power of p (not necessarily a power of r)
  {
    radix_permute(a[], n, p)
    lx := log(r) / log(p) // r == p ** lx
    ln := log(n) / log(p)
    ldm := (log(n)/log(p)) % lx
    if ( ldm != 0 ) // n is not a power of p
        xx := p**lx
        for z:=0 to n-xx step xx
            fft_dit_xx(a[z..z+xx-1], is) // inlined length-xx dit fft
    for ldm:=ldm+lx to ln step lx
        m := p**ldm
        mr := m/r
        for j := 0 to mr-1
            e := exp(is*2*PI*I*j/m)
            for k:=0 to n-m step m
                u[z] := a[k+j+mr*z]
            radix_permute(u[], r, p)
            for z:=1 to r-1 // e**0 == 1
                u[z] := u[z] * e**z
            r_point_fft(u[], is)
            for z:=0 to r-1
                a[k+j+mr*z] := u[z]
  }
Of course the loops that use the variable z have to be unrolled, the (length-$p^x$) scratch space u[] has to be replaced by explicit variables (for example, u0, u1, ...), and the r_point_fft(u[],is) shall be an inlined $p^x$-point FFT.

There is one pitfall: if one uses the radix-$p$ permutation instead of a radix-$p^x$ permutation (for example, the radix-2 `revbin_permute` for a radix-4 FFT), then some additional reordering is necessary in the innermost loop. In the given pseudo code this is indicated by the `radix_permute(u[],p)` just before the
```
20.5.4 Radix-4 DIT FFT

C++ code for a radix-4 DIT FFT is given in [FXT: fft/fftdit4l.cc]:

```cpp
static const ulong RX = 4; // == r
static const ulong LX = 2; // == log(r)/log(p) == log_2(r)
void fft_dit4l(Complex *f, ulong ldn, int is)
// Decimation in time radix-4 FFT.
{
    double s2pi = ( is>0 ? 2.0*M_PI : -2.0*M_PI );
    const ulong n = (1UL<<ldn);
    revbin_permute(f, n);
    ulong ldm = (ldn&1);
    if ( ldm!=0 ) // n is not a power of 4, need a radix-2 step
        for (ulong r=0; r<n; r+=2)
            { 
               Complex a0 = f[r];
               Complex a1 = f[r+1];
               f[r] = a0 + a1;
               f[r+1] = a0 - a1;
            }
    ldm += LX;
    for ( ; ldm<=ldn ; ldm+=LX)
        { 
            ulong m = (1UL<<ldm);
            ulong m4 = (m>>LX);
            double ph0 = s2pi/m;
            for (ulong j=0; j<m4; j++)
                { 
                    double phi = j*ph0;
                    Complex e = SinCos(phi);
                    Complex e2 = SinCos(2.0*phi);
                    Complex e3 = SinCos(3.0*phi);
                    for (ulong r=0; r<n; r+=m)
                        { 
                            ulong i0 = j + r;
                            ulong i1 = i0 + m4;
                            ulong i2 = i1 + m4;
                            ulong i3 = i2 + m4;
                            Complex a0 = f[i0];
                            Complex a1 = f[i2]; // (!)
                            Complex a2 = f[i1]; // (!)
                            Complex a3 = f[i3];
                            a1 *= e;
                            a2 *= e2;
                            a3 *= e3;
                            Complex t0 = (a0+a2) + (a1+a3);
                            Complex t2 = (a0+a2) - (a1+a3);
                            Complex t1 = (a0-a2) + Complex(0,is) * (a1-a3);
                            Complex t3 = (a0-a2) - Complex(0,is) * (a1-a3);
                            f[i0] = t0;
                            f[i1] = t1;
                            f[i2] = t2;
                            f[i3] = t3;
                        }
        }
```
Higher radix FFT algorithms

For reasonable performance the call to the procedure `radix_permute(u[], p)` of the pseudo code has been replaced by changing indices in the loops where the `a[z]` are read. The respective lines are marked with the comment `// (!)`.

In order not to restrict the possible array sizes to powers of \( p^x = 4 \) but only to powers of \( p = 2 \) an additional radix-2 step has been prepended that is used when \( n \) is an odd power of two.

The routine \([\text{FXT: fft\_dit4\_core\_p1()} \text{ in fft/fftdit4.cc}]\) is a reasonably optimized radix-4 DIT FFT implementation. It starts with an radix-2 or radix-8 step for the initial pass with trivial exp()-values. The core routine is hard-coded for \( \sigma = +1 \) and called with swapped real and imaginary part for the inverse transform as explained in section 20.8 on page 425.

The routine, however, uses separate arrays for real and imaginary parts which is very problematic with large transforms: the memory access pattern in skips that are a power of two \( \text{will lead to cache problems} \).

A routine that uses the C++ type `complex` is given in \([\text{FXT: fft/cfftdit4.cc}]\). The core routine is hard-coded for \( \sigma = -1 \) (therefore the name suffix \_m1):
The sumdiff() function is defined in [FXT: aux0/sumdiff.h]:

```cpp
template <typename Type>
static inline void sumdiff(Type &a, Type &b)
// {a, b} <--| {a+b, a-b}
{ Type t=a-b; a+=b; b=t; }
```

The routine `fft8_dit_core_m1()` is an unrolled size-8 DIT FFT (hard-coded for $\sigma = -1$) given in [FXT: fft/fft8ditcore.cc]. We further need a version of the routine for the positive sign. It uses a routine `fft8_dit_core_p1()` for the computation of length-8 DIT FFTs (with $\sigma = -1$). The following changes need to be made in the core routine [FXT: fft/cfftdit4.cc]:

```cpp
void
fft_dit4_core_p1(Complex *f, ulong ldn)
// Fixed isign = +1
{
    [--snip--]
    for (ulong i0=0; i0<n; i0+=8) fft8_dit_core_p1(f+i0); // isign
    [--snip--]
    v *= Complex(0, +1); // isign
    [--snip--]
    const double ph0 = +2.0*M_PI/m; // isign
    [--snip--]
    v *= Complex(0, +1); // isign
    [--snip--]
}
```

The routine to be called by the user is

```cpp
void
fft_dit4(Complex *f, ulong ldn, int is)
// Fast Fourier Transform
// ldn := base-2 logarithm of the array length
// is := sign of the transform (+1 or -1) 
// Radix-4 decimation in time algorithm
{ 
    revbin_permute(f, 1UL<<ld); 
    if ( is>0 ) fft_dit4_core_p1(f, ldn); 
    else fft_dit4_core_m1(f, ldn); 
}
```

A version that uses the separate arrays for real and imaginary part is given in [FXT: fft/fft4dit.cc]. The type `complex` version, should be preferred for large transforms.

### 20.5.5 Radix-4 DIF FFT

Pseudo code for a radix-4 DIF FFT:

```plaintext
\texttt{fftbook draft of 2008-August-17}
```
20.5: Higher radix FFT algorithms

procedure fftdif4(a[], ldn, is)
#define complex a[0..2**ldn-1] input, result
n := 2**ldn
for ldm := ldn to 2 step -2
{
m := 2**ldm
mr := m/4
for j := 0 to mr-1
{
e := exp(is*2*PI*I*j/m)
e2 := e * e
e3 := e2 * e
for r := 0 to n-m step m
{
u0 := a[r+j]
u1 := a[r+j+mr]
u2 := a[r+j+mr*2]
u3 := a[r+j+mr*3]
x := u0 + u2
y := u1 + u3
t0 := x + y // == (u0+u2) + (u1+u3)
t2 := x - y // == (u0+u2) - (u1+u3)
x := u0 - u2
y := (u1 - u3)*I*is
t1 := x + y // == (u0-u2) + (u1-u3)*I*is
t3 := x - y // == (u0-u2) - (u1-u3)*I*is
t1 := t1 * e
t2 := t2 * e2
t3 := t3 * e3
a[r+j] := t0
a[r+j+mr] := t2 // (!)
a[r+j+mr*2] := t1 // (!)
a[r+j+mr*3] := t3
}
}
if is_odd(ldn) then // n not a power of 4
{
for r := 0 to n-2 step 2
{
{a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
}
}
revbin_permute(a[], n)

The C++ equivalent is [FXT: fft_dif4l() in [fft/fftdif4l.cc]]. A reasonably optimized implementation is given in [FXT: fft/cfftdif4.cc], it is hard-coded for \( \sigma = +1 \) (therefore the name suffix _p1):

```c++
1 static const ulong RX = 4;
2 static const ulong LX = 2;
3
4 void fftdif4_core_p1(Complex *f, ulong ldn)
5 // Auxiliary routine for fftdif4().
6 // Radix-4 decimation in frequency FFT.
7 // Output data is in revbin_permuted order.
8 // ldn := base-2 logarithm of the array length.
9 // Fixed isign = +1
10 {
11 const ulong n = (1UL<<ldn);
12 if ( n<=2 )
13 {
14 if ( n==2 ) sumdiff(f[0], f[1]);
15 return;
16 }
17 for (ulong ldm=ldn; ldm>(LX<<1); ldm--) {
```
The routine for \( \sigma = -1 \) needs changes where the comment `isign` appears [FXT: `fft/cfftdif4.cc`]:

```c
void  
fft_dif4_core_m1(Complex *f, ulong ldn)  
// Fixed isign = -1  
{  
  --snip--  
  const double ph0 = -2.0*M_PI/m; // isign  
  --snip--  
  v *= Complex(0, -1); // isign  
  --snip--  
  for (ulong i0=0; i0<n; i0+=8) fft8_dif_core_p1(f+i0); // isign  
  --snip--  
  v *= Complex(0, -1); // isign  
  --snip--  
}
```

The routine to be called by the user is
void fft_dif4(Complex *f, ulong ldn, int is)
// Fast Fourier Transform
// ldn := base-2 logarithm of the array length
// is := sign of the transform (+1 or -1)
// radix-4 decimation in frequency algorithm
{
    if ( is>0 ) fft_dif4_core_pl(f, ldn);
    else fft_dif4_core_ml(f, ldn);
    revbin_permute(f, 1UL<<ldn);
}

A version that uses the separate arrays for real and imaginary part is given in [FXT: fft/fftdif4.cc]. Again, the type complex version, should be preferred for large transforms.

### 20.6 Split-radix Fourier transforms

The idea underlying the split-radix FFT algorithm is to use both radix-2 and radix-4 decompositions at the same time. We use one relation from the radix-2 (DIF) decomposition (relation 20.3-6a on page 410, the one for the even indices), and for the odd indices we use the radix-4 splitting (relations 20.5-7b and 20.5-7d on page 414) in a slightly reordered form. The radix-4 decimation in frequency (DIF) step for the split-radix FFT is

\[
\mathcal{F}[a](0\%2) \frac{n}{2} = \mathcal{F}\left(a(0/2) + a(1/2)\right) \tag{20.6-1a}
\]

\[
\mathcal{F}[a](1\%4) \frac{n}{4} = \mathcal{F}\left(S^{1/4}\left((a(0/4) - a(2/4)) + i\sigma(a(1/4) - a(3/4))\right)\right) \tag{20.6-1b}
\]

\[
\mathcal{F}[a](3\%4) \frac{n}{4} = \mathcal{F}\left(S^{3/4}\left((a(0/4) - a(2/4)) - i\sigma(a(1/4) - a(3/4))\right)\right) \tag{20.6-1c}
\]

Now we have expressed the length-\(N = 2^n\) FFT as one length-\(N/2\) and two length-\(N/4\) FFTs. A nice feature is that the operation count of the split-radix FFT is actually lower than that of the radix-4 FFT. Using the introduced notation it is almost trivial to write down the DIT version of the algorithm: The radix-4 decimation in time (DIT) step for the split-radix FFT is

\[
\mathcal{F}[a](0\%2) \frac{n}{2} = \left(\mathcal{F}\left(a(0/2)\right) + S^{1/2}\mathcal{F}\left(a(1/2)\right)\right) \tag{20.6-2a}
\]

\[
\mathcal{F}[a](1/4) \frac{n}{4} = \left(\mathcal{F}\left(a(0/4)\right) - S^{1/4}\mathcal{F}\left(a(2/4)\right)\right) + i\sigma S^{1/4}\left(\mathcal{F}\left[a(1/4)\right] - S^{2/4}\mathcal{F}\left[a(3/4)\right]\right) \tag{20.6-2b}
\]

\[
\mathcal{F}[a](3/4) \frac{n}{4} = \left(\mathcal{F}\left[a(0/4)\right] - S^{1/4}\mathcal{F}\left[a(2/4)\right]\right) - i\sigma S^{1/4}\left(\mathcal{F}\left[a(1/4)\right] - S^{2/4}\mathcal{F}\left[a(3/4)\right]\right) \tag{20.6-2c}
\]

Pseudo code for the split-radix DIF algorithm:

```plaintext
procedure fft_splitradix_dif(x[], y[], ldn, is)
{
    n := 2**ldn
    if n<=1 return
    n2 := 2*n
    for k:=1 to ldn
    {
        n2 := n2 / 2
        n4 := n2 / 4
        e := 2 * PI / n2
        for j:=0 to n4-1
        {
            a := j * e
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a) // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a) // == 4*ss1*(0.75-ss1*ss1)
        }
    }
}
```
Chapter 20: The Fourier transform

```cpp
ix := j
id := 2*n2
while ix<n-1 {
  i0 := ix
  while i0 < n {
    i1 := i0 + n4
    i2 := i1 + n4
    {x[i0], r1} := {x[i0] + x[i2], x[i0] - x[i2]}
    {x[i1], r2} := {x[i1] + x[i3], x[i1] - x[i3]}
    {y[i0], s1} := {y[i0] + y[i2], y[i0] - y[i2]}
    {y[i1], s2} := {y[i1] + y[i3], y[i1] - y[i3]}
    {r1, s3} := {r1+s2, r1-s2}
    {r2, s2} := {r2+s1, r2-s1}
    // complex mult: (x[i2],y[i2]) := -(s2,r1) * (ss1,cc1)
    x[i2] := r1*cc1 - s2*ss1
    y[i2] := -s2*cc1 - r1*ss1
  }
  i0 := i0 + id
}
ix := 2 * id - n2 + j
id := 4 * id
}

revbin_permute(x[],n)
revbin_permute(y[],n)

if is>0 {
  for j:=1 to n/2-1
  { swap(x[j], x[n-j])
  }
}
```

The C++ implementation given in [FXT: fft/fftsplitradix.cc] uses a DIF core as above which was given in [106]. The C++ type complex version of the split-radix FFT given in [FXT: fft/cfftsplitradix.cc] uses a DIF or DIT core, depending on the sign of the transform. Here we just give the DIF version:

```cpp
void split_radix_dif_fft_core(Complex *f, ulong ldn)
// Split-radix decimation in frequency (DIF) FFT.
// ldn := base-2 logarithm of the array length.
// Fixed isign = +1
// Output data is in revbin_permuted order.
```

```c
7  {  
8    if ( ldnn==0 ) return;
9    const ulong n = (1UL<<ldn);
10    double s2pi = 2.0*M_PI; // pi*2*isign
11    ulong n2 = 2*n; 
12    for (ulong k=1; k<ldn; k++)
13      {  
14        n2 >>= 1; // == n>>(k-1) == n, n/2, n/4, ..., 4
15        const ulong n4 = n2 >> 2; // == n/4, n/8, ..., 1
16        const double e = s2pi / n2;
17        for (ulong j=0; j<n4; j++)
18          {  
19            double a = j * e;
20            double cc1, ss1, cc3, ss3;
21            SinCos(a, &ss1, &cc1);
22            SinCos(3.0*a, &ss3, &cc3);
23            ulong ix = j;
24            ulong id = (n2<<1);
25            while ( ix<n )
26              {  
27                ulong i1 = i0 + n4;
28                ulong i2 = i1 + n4;
29                ulong i3 = i2 + n4;
30                Complex t0, t1;
31                sumdiff3(f[i0], f[i2], t0);
32                sumdiff3(f[i1], f[i3], t1);
33                // t1 *= Complex(0, 1); // +isign
34                t1 = Complex(-t1.imag(), t1.real());
35                sumdiff(t0, t1);
36                f[i2] = t0; // * Complex(cc1, ss1);
37                f[i3] = t1; // * Complex(cc3, ss3);
38                ix = (id<<1) - n2 + j;
39                id <<= 2;
40              }
41          }
42    }
43    for (ulong j=1; j<n4; j++)
44      {  
45        double a = j * e;
46        double cc1, ss1, cc3, ss3;
47        SinCos(a, &ss1, &cc1);
48        SinCos(3.0*a, &ss3, &cc3);
49        ulong ix = j;
50        ulong id = (n2<<1);
51        while ( ix<n )
52          {  
53            ulong i1 = i0 + n4;
54            ulong i2 = i1 + n4;
55            ulong i3 = i2 + n4;
56            Complex t0, t1;
57            sumdiff3(f[i0], f[i2], t0);
58            sumdiff3(f[i1], f[i3], t1);
59            t1 = Complex(-t1.imag(), t1.real());
60            sumdiff(t0, t1);
61            f[i2] = t0 * Complex(cc1, ss1);
62            f[i3] = t1 * Complex(cc3, ss3);
63            ix = (id<<1) - n2 + j;
64            id <<= 2;
65          }
66      }
67    for (ulong ix=0, id=4; ix<n; id+=4)
68  }
```
The function \( \text{sumdiff3}() \) is defined in [FXT: aux0/sumdiff.h]:

```c
template <typename Type>
static inline void sumdiff3(Type &a, Type b, Type &d)
// {a, b, d} <--| {a+b, b, a-b} (used in split-radix fft)
{ d=a-b; a+=b; }
```

### 20.7 Symmetries of the Fourier transform

A bit of notation again. Let \( \overline{a} \) be the length-\( n \) sequence \( a \) reversed around the element with index 0:

\[
\overline{a}_0 := a_0 \\
\overline{a}_{n/2} := a_{n/2} \quad \text{if } n \text{ even} \\
\overline{a}_k := a_{n-k} = a_{-k} \quad (20.7-1c)
\]

That is, we consider the indices modulo \( n \) and \( \overline{a} \) is the sequence \( a \) with negated indices. Element zero stays in its place and for even \( n \) there is also an element with index \( n/2 \) that stays in place.

Example one, length-4: \( a := [0, 1, 2, 3] \) then \( \overline{a} = [0, 3, 2, 1] \) (zero and two stay).

Example two, length-5: \( a := [0, 1, 2, 3, 4] \) then \( \overline{a} = [0, 4, 3, 2, 1] \) (only zero stays).

Let \( a_S \) and \( a_A \) denote the symmetric and antisymmetric part of the sequence \( a \), respectively:

\[
a_S := \frac{1}{2} (a + \overline{a}) \quad (20.7-2a) \\
a_A := \frac{1}{2} (a - \overline{a}) \quad (20.7-2b)
\]

The elements with index 0 (and \( n/2 \) for even \( n \) \( a_A \) are zero. One has

\[
a = a_S + a_A \\
\overline{a} = a_S - a_A \quad (20.7-3a)
\]

Let \( c + i d \) be the FT of the sequence \( a + i b \). Then

\[
\mathcal{F}[ (a_S + a_A) + i (b_S + b_A) ] = (c_S + c_A) + i (d_S + d_A) \quad \text{where} \quad (20.7-4a)
\]

\[
\mathcal{F}[a_S] = c_S \in \mathbb{R} \quad (20.7-4b) \\
\mathcal{F}[a_A] = i d_A \in i \mathbb{R} \quad (20.7-4c) \\
\mathcal{F}[i b_S] = i d_S \in i \mathbb{R} \quad (20.7-4d) \\
\mathcal{F}[i b_A] = c_A \in \mathbb{R} \quad (20.7-4e)
\]

Where we write \( a \in \mathbb{R} \) as a short form for a purely real sequence \( a \). Equivalently, write \( a \in i \mathbb{R} \) for purely imaginary sequences. Thereby the FT of a complex symmetric or antisymmetric sequence is symmetric or antisymmetric, respectively:

\[
\mathcal{F}[a_S + i b_S] = c_S + i d_S \quad (20.7-5a) \\
\mathcal{F}[a_A + i b_A] = c_A + i d_A \quad (20.7-5b)
\]

The real and imaginary part of the transform of a symmetric sequence correspond to the real and imaginary part of the original sequence. With an antisymmetric sequence the transform of the real
and imaginary part correspond to the imaginary and real part of the original sequence.

\[
\mathcal{F}
\begin{bmatrix}
a_S + a_A \\
i(b_S + b_A)
\end{bmatrix}
= c_S + i d_A
\]  
(20.7-6a)

\[
\mathcal{F}
\begin{bmatrix}
(a_S + a_A) \\
i(b_S + b_A)
\end{bmatrix}
= c_A + i d_S
\]  
(20.7-6b)

Now let the sequence \(a\) be purely real. Then

\[
\mathcal{F}
\begin{bmatrix}
a_S \\
a_A
\end{bmatrix}
= \pm \mathcal{F}
\begin{bmatrix}
a_S \\
a_A
\end{bmatrix}
\in \mathbb{R}
\]  
(20.7-7a)

\[
\mathcal{F}
\begin{bmatrix}
a_S \\
a_A
\end{bmatrix}
= -\mathcal{F}
\begin{bmatrix}
a_S \\
a_A
\end{bmatrix}
\in i \mathbb{R}
\]  
(20.7-7b)

That is, the FT of a real symmetric sequence is real and symmetric and the FT of a real antisymmetric sequence is purely imaginary and antisymmetric. Thereby the FT of a general real sequence is the complex conjugate of its reversed:

\[
\mathcal{F}
\begin{bmatrix}
a
\end{bmatrix}
= \overline{\mathcal{F}
\begin{bmatrix}
a
\end{bmatrix}}
\text{ for } a \in \mathbb{R}
\]  
(20.7-8)

Similarly, for a purely imaginary sequence \(b \in i \mathbb{R}\):

\[
\mathcal{F}
\begin{bmatrix}
b_S \\
b_A
\end{bmatrix}
= \pm \mathcal{F}
\begin{bmatrix}
b_S \\
b_A
\end{bmatrix}
\in i \mathbb{R}
\]  
(20.7-9a)

\[
\mathcal{F}
\begin{bmatrix}
b_S \\
b_A
\end{bmatrix}
= -\mathcal{F}
\begin{bmatrix}
b_S \\
b_A
\end{bmatrix}
\in \mathbb{R}
\]  
(20.7-9b)

We compare the results of the Fourier transform and its inverse (the transform with negated sign \(\sigma\)) by symbolically writing the transforms as a complex multiplication with the trigonometric term (using \(C\) for cosine, \(S\) for sine):

\[
\mathcal{F}
\begin{bmatrix}
a + ib
\end{bmatrix}
: (a + ib) (C + i S) = (a C - b S) + i (b C + a S)
\]  
(20.7-10a)

\[
\mathcal{F}^{-1}
\begin{bmatrix}
a + ib
\end{bmatrix}
: (a + ib) (C - i S) = (a C + b S) + i (b C - a S)
\]  
(20.7-10b)

The terms on the right side can be identified with those in relation 20.7-4a. We see that changing the sign of the transform leads to a result where the components due to the antisymmetric components of the input are negated.

Now write \(\mathcal{F}\) for the Fourier operator, and \(\mathcal{R}\) for the reversal. We have \(\mathcal{F}^4 = \text{id}\), \(\mathcal{F}^3 = \mathcal{F}^{-1}\), and \(\mathcal{F}^2 = \mathcal{R}\). Thereby the inverse transform can be computed as either

\[
\mathcal{F}^{-1} = \mathcal{R} \mathcal{F} = \mathcal{F} \mathcal{R}
\]  
(20.7-11)

20.8 Inverse FFT for free

Some FFT implementations are hard-coded for a fixed sign of the transform. If one cannot easily modify the implementation into the transform with the other sign (the inverse transform), then how can one compute the inverse FFT?

If the implementation uses separate arrays for real and imaginary part of the complex sequences to be transformed, as in

```plaintext
procedure my_fft(ar[], ai[], ldn) // only for is==+1 !
// real ar[0..2**ldn-1] input, result, real part
// real ai[0..2**ldn-1] input, result, imaginary part
{
// incredibly complicated code
// that you cannot see how to modify
// for is==-1
}
```

then do as follows: with the forward transform being
my_fft(ar[], ai[], ldn) // forward FFT
compute the inverse transform as
my_fft(ai[], ar[], ldn) // backward FFT
Note the swapped real and imaginary parts! The same trick works for a procedure coded for fixed is=−1.

To see why this works, we first note that
\[
\mathcal{F}[a + ib] = \mathcal{F}[a_S] + i\sigma \mathcal{F}[a_A] + i \mathcal{F}[b_S] + \sigma \mathcal{F}[b_A] \tag{20.8-1a}
\]
and the computation with swapped real- and imaginary parts gives
\[
\mathcal{F}[b + ia] = \mathcal{F}[b_S] + i \mathcal{F}[a_S] + i\sigma (\mathcal{F}[a_A] - i \mathcal{F}[b_A]) \tag{20.8-2a}
\]
... but the real and imaginary parts are implicitly swapped at the end of the computation, giving
\[
\mathcal{F}[a_S] + i \mathcal{F}[b_S] - i\sigma (\mathcal{F}[a_A] - i \mathcal{F}[b_A]) = \mathcal{F}^{-1}[a + ib] \tag{20.8-2b}
\]
When a complex type is used then the best way to achieve the inverse transform may be to reverse the sequence according to the symmetry of the FT according to relation 20.7-11: the transform with negated sign can be computed by reversing the order of the result (use \texttt{FXT: reverse}() in \texttt{perm/reverse.h}). The reversal can also happen with the input data before the transform, which is advantageous if the data has to be copied anyway (use \texttt{FXT: copy} \texttt{reverse}() in \texttt{aux1/copy.h}). While not really ‘free’ the additional work will usually not matter.

A mechanical way to obtain a routine for the inverse FFT from a given FFT routine for length \(n\) is to replace all reads and writes at nonzero array indices \(i\) by the operations at indices \(n - i\) (where \(n\) is identified with zero).

### 20.9 Real valued Fourier transforms

The Fourier transform of a purely real sequence \(c = \mathcal{F}[a]\) where \(a \in \mathbb{R}\) has a symmetric real part (\(\Re c = \Re \mathcal{F}[a]\), relation 20.7-8) and an antisymmetric imaginary part (\(\Im c = -\Im \mathcal{F}[a]\)). The symmetric and antisymmetric part of the original sequence correspond to the symmetric (and purely real) and antisymmetric (and purely imaginary) part of the transform, respectively:
\[
\mathcal{F}[a] = \mathcal{F}[a_S] + i\sigma \mathcal{F}[a_A] \tag{20.9-1}
\]
Simply using a complex FFT for real input is basically a waste by a factor 2 of memory and CPU cycles. There are several ways out:

- wrapper routines for complex FFTs (section 20.9.3 on page 428)
- usage of the fast Hartley transform (section 24.5 on page 519)
- special versions of the split-radix algorithm (section 20.9.4 on page 429)

All techniques have in common that they store only half of the complex result to avoid the redundancy due to the symmetries of a complex FT of purely real input. The result of a real to (half-) complex FT (abbreviated R2CFT) contains the purely real components \(c_0\) (the ‘DC-part’ of the input signal) and, in case \(n\) is even, \(c_{n/2}\) (the Nyquist frequency part). The inverse procedure, the (half-) complex to real transform (abbreviated C2RFT) must be compatible to the ordering of the R2CFT.
20.9: Real valued Fourier transforms

20.9.1 Sign of the transforms

The sign of the transform can be chosen arbitrarily to be either +1 or −1. Note that transform with the ‘other sign’ is not the inverse transform. The R2CFT and its inverse C2RFT must use the same sign.

Some R2CFT and C2RFT implementations are hard-coded for a fixed sign. In order to obtain the R2CFT with the other sign the trick (in the spirit of section 20.8) is to negate the imaginary part after the transform. This of course is not much of a trick at all. In case one has to copy the data anyway before the transform one can exploit the relation

\[ F[a] = F[a_S] - i \sigma F[a_A] \]  

(20.9-2)

That is, copy the real data in reversed order to get the transform with the other sign. This technique does not involve an extra pass and should be virtually for free.

For the complex-to-real FTs (C2RFT) one has to negate the imaginary part before the transform in order to obtain the inverse transform for the other sign.

20.9.2 Data ordering

Let c be the Fourier transform of the purely real sequence, it is stored in the array a[]. The procedures presented in what follows use one of the following schemes for storing the transformed sequence.

A scheme that interleaves real and imaginary parts (‘complex ordering’) is

\[
\begin{align*}
a[0] &= \Re c_0 \\
a[1] &= \Re c_{n/2} \\
a[2] &= \Re c_1 \\
a[3] &= \Im c_1 \\
a[4] &= \Re c_2 \\
a[5] &= \Im c_2 \\
& \vdots \\
a[n-2] &= \Re c_{n/2-1} \\
a[n-1] &= \Im c_{n/2-1}
\end{align*}
\]

Note the absence of the elements \( \Im c_0 \) and \( \Im c_{n/2} \) which are always zero.

Some routines store the real parts in the lower half, and imaginary parts in upper half. The data in the lower half will always be ordered as follows:

\[
\begin{align*}
a[0] &= \Re c_0 \\
a[1] &= \Re c_1 \\
a[2] &= \Re c_2 \\
& \vdots \\
a[n/2] &= \Re c_{n/2}
\end{align*}
\]

For the imaginary part of the result there are two schemes:
Scheme 1 (‘parallel ordering’) is
\[
\begin{align*}
a[n/2 + 1] & = \Im c_1 \\
a[n/2 + 2] & = \Im c_2 \\
a[n/2 + 3] & = \Im c_3 \\
& \vdots \\
a[n - 1] & = \Im c_{n/2 - 1}
\end{align*}
\] (20.9-5)

Scheme 2 (‘antiparallel ordering’) is
\[
\begin{align*}
a[n/2 + 1] & = \Im c_{n/2 - 1} \\
a[n/2 + 2] & = \Im c_{n/2 - 2} \\
a[n/2 + 3] & = \Im c_{n/2 - 3} \\
& \vdots \\
a[n - 1] & = \Im c_1
\end{align*}
\] (20.9-6)

20.9.3 Real valued FT via wrapper routines

A simple way to use a complex length-$n/2$ FFT for a real length-$n$ FFT ($n$ even) is to use some post-and preprocessing routines. For a real sequence $a$ one feeds the (half length) complex sequence $f = a^{(\text{even})} + ia^{(\text{odd})}$ into a complex FFT. Some post-processing is necessary. This is not the most elegant real FFT available, but it is directly usable to turn complex FFTs of any (even) length into a real-valued FFT.

A C++ implementation of the real to complex FFT (R2CFT) is given in \[FXT: \text{realfft/realfftwrap.cc}\], the sign of the transform is hard-coded to $\sigma = +1$:

```cpp
void wrap_real_complex_fft(double *f, ulong ldn)
// Real to complex FFT (R2CFT)
{
if ( ldn==0 ) return;

fht_fft((Complex *)f, ldn-1, +1); // cast

const ulong n = 1UL<<ldn;
const ulong nh = n/2, n4 = n/4;
const double phi0 = M_PI / nh;
for(ulong i=1; i<n4; i++)
{
    ulong i1 = 2 * i; // re low [2, 4, ..., n/2-2]
    ulong i2 = i1 + 1; // im low [3, 5, ..., n/2-1]
    ulong i3 = n - i1; // re hi [n-2, n-4, ..., n/2+2]
    ulong i4 = i3 + 1; // im hi [n-1, n-3, ..., n/2+3]

double fir, f2i;
sumdiff05(f[i3], f[i1], fir, f2i);

double f2r, f1i;
sumdiff05(f[i2], f[i4], f2r, f1i);

double c, s;
SinCos(phi, &s, &c);
double tr, ti;
cmult(c, s, f2r, f2i, tr, ti);
// f[i1] = fir + tr; // re low
// f[i3] = fir - tr; // re hi
// ==
sumdiff(fir, tr, f[i1], f[i3]);
}
```

[fxtbook draft of 2008-August-17]
20.9.4 Real valued split-radix Fourier transforms

We give pseudo code for the split-radix real to complex FFT and its inverse. The C++ implementations are given in [FXT: realfft/realfftsplitradix.cc]. The code given here follows [107], see also [250] (erratum for page 859 of [250]: at the start of the D0 32 loop replace assignments by \( CC1=\cos(A) \), \( SS1=\sin(A) \), \( CC3=\cos(A3) \), \( SS3=\sin(A3) \)).
Recall that the pseudo code
\[
\{ a, b \} := \{ c, d \}
\]
is a parallel assignment. For example,
\[
\{x_0, x_1\} := \{x_0+x_1, x_0-x_1\}
\]
would translate to the C code
\[
\text{double } s = x_0 + x_1, \ d = x_0 - x_1;\\
x_0 = s; \ x_1 = d;
\]

Use the function \[\text{FXT: sumdiff()}\] in \[\text{aux0/sumdiff.h}\] for the translation of the parallel assignments.

### 20.9.5 Real to complex split-radix FFT

Pseudo code for the split-radix R2CFT algorithm, the sign of the transform is hard-coded to \(\sigma = -1\).

```
procedure r2cft_splitradix_dit(x[], ldn)
{
    n := 2**ldn
    revbin_permute(x[], n);
    ix := 1;
    id := 4;
    do
    { i0 := ix-1
        while i0<n
        { i1 := i0 + 1
            \{x[i0], x[i1]} := \{x[i0]+x[i1], x[i0]-x[i1]\}
            i0 := i0 + id
        }
        ix := 2*id-1
        id := 4 + id
    }
    while ix<n
    n2 := 2
    nn := n/4
    while nn!=0
    { ix := 0
        n2 := 2*n2
        id := 2*n2
        n4 := n2/4
        n8 := n2/8
        do // ix loop
        { i0 := ix
            while i0<n
            { i1 := i0
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4
            \{t1, x[i4]} := \{x[i4]+x[i3], x[i4]-x[i3]\}
                \{x[i1], x[i3]} := \{x[i1]+t1, x[i1]-t1\}
            if n4!=1
            { i1 := i1 + n8
                i2 := i2 + n8
                i3 := i3 + n8
                i4 := i4 + n8
                t1 := (x[i3]+x[i4]) * sqrt(1/2)
                t2 := (x[i3]-x[i4]) * sqrt(1/2)
                \{x[i4], x[i3]} := \{x[i2]-t1, -x[i2]-t1\}
                \{x[i1], x[i2]} := \{x[i1]+t2, x[i1]-t2\}
            }
            i0 := i0 + id
        }
        ix := 2*id - n2
        id := 2*id
    }
    while ix<n
```
20.9: Real valued Fourier transforms

The ordering of the output is given as relations 20.9-4 on page 427 for the real part, and relation 20.9-6 for the imaginary part.

20.9.6 Complex to real split-radix FFT

The following routine is the inverse of r2cft_splitradix_dit(). The imaginary part of the input data must be ordered according to relation 20.9-6 on page 428. We give pseudo code for the split-radix C2RFT algorithm, the sign of the transform is hard-coded to $\sigma = -1$:

```
procedure c2rft_splitradix_dif(x[], ldn)
{
    n := 2**ldn
    n2 := n/2
    nn := n/4
    while nn!=0
    {
        ix := 2*id - n2
        id := 2*id
    }
}
```
id := n2
n2 := n2/2
n4 := n2/4
n8 := n2/8

do // ix loop
{
    i0 := ix
    while i0<n
    {
        i1 := i0
        i3 := i1 + n4
        i4 := i3 + n4
        {x[i1], t1} := {x[i1]+x[i3], x[i1]-x[i3]}
        x[i2] := 2*x[i2]
        x[i4] := 2*x[i4]
        {x[i3], x[i4]} := {t1+x[i4], t1-x[i4]}

        if n4!=1
        {
            i1 := i1 + n8
            i3 := i3 + n8
            i4 := i4 + n8
            {x[i1], t1} := {x[i2]+x[i1], x[i2]-x[i1]}
            {t2, x[i2]} := {x[i4]+x[i3], x[i4]-x[i3]}
            x[i3] := -sqrt(2)*(t2+t1)
            x[i4] := sqrt(2)*(t1-t2)
        }

        i0 := i0 + id
    }

    ix := 2*id - n2
    id := 2*id
}
while ix<n

e := 2.0*PI/n2
a := e
for j:=2 to n8
{
    cc1 := cos(a)
    ss1 := sin(a)
    cc3 := cos(3*a) // == 4*cc1*(cc1*cc1-0.75)
    ss3 := sin(3*a) // == 4*ss1*(0.75-ss1*ss1)
    a := j*e

    ix := 0
    id := 2*n2
    do // ix-loop
    {
        i0 := ix
        while i0<n
        {
            i1 := i0 + j - 1
            i2 := i1 + n6
            i3 := i2 + n6
            i4 := i3 + n6 - j + 1
            i5 := i4 + n6
            i6 := i5 + n6
            i7 := i6 + n6
            {x[i1], t1} := {x[i1]+x[i6], x[i1]-x[i6]}
            {x[i5], x[i2]} := {x[i5]+x[i12], x[i5]-x[i12]}
            {t3, x[i5]} := {x[i18]+x[i13], x[i18]-x[i13]}
            {t4, x[i2]} := {x[i14]+x[i17], x[i14]-x[i17]}
            {t1, t5} := {t1+t4, t1-t4}
            {t2, t4} := {t2+t3, t2-t3}

            // complex mult: (x[i7],x[i3]) := (t5,t4) * (ss1,cc1)
            x[i3] := t5*cc1 + t4*ss1
            x[i7] := -t4*cc1 + t5*ss1

            // complex mult: (x[i4],x[i8]) := (t1,t2) * (cc3,ss3)
            x[i4] := t1*cc3 - t2*ss3
            x[i8] := t2*cc3 + t1*ss3

            i0 := i0 + id
        }
        ix := 2*id - n2
    }
}
20.10: Multidimensional Fourier transforms

20.10.1 Definition

Let $a_{x,y}$ ($x = 0, 1, 2, \ldots, C - 1$ and $y = 0, 1, 2, \ldots, R - 1$) be a 2-dimensional array. That is, a $R \times C$ 'matrix' of $R$ rows (of length $C$) and $C$ columns (of length $R$). Its 2-dimensional Fourier transform is defined by:

$$c = \mathcal{F}[a]$$

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} \sum_{y=0}^{R-1} a_{x,y} z^{+(xk/C+yh/R)}$$

where $z = e^{\sigma 2 \pi i}$

where $k \in \{0, 1, 2, \ldots, C - 1\}$, $h \in \{0, 1, 2, \ldots, R - 1\}$, and $n = R \cdot C$. The inverse transform is

$$a = \mathcal{F}^{-1}[c]$$

$$a_{x,y} = \frac{1}{\sqrt{n}} \sum_{k=0}^{C-1} \sum_{h=0}^{R-1} c_{k,h} z^{-+(xk/C+yh/R)}$$

For a $m$-dimensional array $a_{\vec{x}}$ (where $\vec{x} = (x_1, x_2, x_3, \ldots, x_m)$ and $x_i \in \{0, 1, 2, \ldots, S_i\}$) the $m$-dimensional Fourier transform $c_{\vec{k}}$ (where $\vec{k} = (k_1, k_2, k_3, \ldots, k_m)$ and $k_i \in \{0, 1, 2, \ldots, S_i\}$) is defined as

$$c_{\vec{k}} := \frac{1}{\sqrt{n}} \sum_{x_1=0}^{S_1-1} \sum_{x_2=0}^{S_2-1} \ldots \sum_{x_m=0}^{S_m-1} a_{\vec{x}} z^{x_1 k_1/S_1 + x_2 k_2/S_2 + \ldots + x_m k_m/S_m}$$

The inverse transform is, like in the 1-dimensional case, the complex conjugate transform.

20.10.2 The row-column algorithm

The equation of the definition of the two dimensional FT (relation \[20.10-1a\]) can be recast as

$$c_{k,h} = \frac{1}{\sqrt{n}} \sum_{y=0}^{R-1} \sum_{x=0}^{C-1} a_{x,y} \exp(yh/R) \sum_{x=0}^{C-1} a_{x,y} \exp(xk/C)$$

20.10 Multidimensional Fourier transforms

20.10.1 Definition

Let $a_{x,y}$ ($x = 0, 1, 2, \ldots, C - 1$ and $y = 0, 1, 2, \ldots, R - 1$) be a 2-dimensional array. That is, a $R \times C$ ‘matrix’ of $R$ rows (of length $C$) and $C$ columns (of length $R$). Its 2-dimensional Fourier transform is defined by:

$$c = \mathcal{F}[a]$$

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} \sum_{y=0}^{R-1} a_{x,y} z^{+(xk/C+yh/R)}$$

where $z = e^{\sigma 2 \pi i}$

where $k \in \{0, 1, 2, \ldots, C - 1\}$, $h \in \{0, 1, 2, \ldots, R - 1\}$, and $n = R \cdot C$. The inverse transform is

$$a = \mathcal{F}^{-1}[c]$$

$$a_{x,y} = \frac{1}{\sqrt{n}} \sum_{k=0}^{C-1} \sum_{h=0}^{R-1} c_{k,h} z^{-+(xk/C+yh/R)}$$

For a $m$-dimensional array $a_{\vec{x}}$ (where $\vec{x} = (x_1, x_2, x_3, \ldots, x_m)$ and $x_i \in \{0, 1, 2, \ldots, S_i\}$) the $m$-dimensional Fourier transform $c_{\vec{k}}$ (where $\vec{k} = (k_1, k_2, k_3, \ldots, k_m)$ and $k_i \in \{0, 1, 2, \ldots, S_i\}$) is defined as

$$c_{\vec{k}} := \frac{1}{\sqrt{n}} \sum_{x_1=0}^{S_1-1} \sum_{x_2=0}^{S_2-1} \ldots \sum_{x_m=0}^{S_m-1} a_{\vec{x}} z^{x_1 k_1/S_1 + x_2 k_2/S_2 + \ldots + x_m k_m/S_m}$$

The inverse transform is, like in the 1-dimensional case, the complex conjugate transform.

20.10.2 The row-column algorithm

The equation of the definition of the two dimensional FT (relation \[20.10-1a\]) can be recast as

$$c_{k,h} = \frac{1}{\sqrt{n}} \sum_{y=0}^{R-1} \sum_{x=0}^{C-1} a_{x,y} \exp(yh/R) \sum_{x=0}^{C-1} a_{x,y} \exp(xk/C)$$
which shows that the 2-dimensional FT can be obtained by first applying 1-dimensional transforms on
the rows and then applying 1-dimensional transforms on the columns. The same result is obtained when
the columns are transformed first and then the rows.

This leads us directly to the row-column algorithm for 2-dimensional FFTs. Pseudo code to compute the
two dimensional FT of \(a[][]\) using the row-column method:

```plaintext
1 procedure rowcol_ft(a[][], R, C, is)
2 {
3 complex a[R][C] // R (length-C) rows, C (length-R) columns
4 for r:=0 to R-1 // FFT rows
5 { fft(a[r][]), C, is)
6 }
7 complex t[R] // scratch array for columns
8 for c:=0 to C-1 // FFT columns
9 { copy a[0,1,...,R-1][c] to t[] // get column
10 fft(t[]), R, is)
11 copy t[] to a[0,1,...,R-1][c] // write back column
12 }
13 }
```

Here it is assumed that the rows lie in contiguous memory (as in the C language). The equivalent C++
code is given in [FFT: ft/twodimfft.cc].

Transposing the array before the column pass avoids the notorious problem with memory-cache and will
improve performance in most cases:

```plaintext
1 procedure rowcol_fft2d(a[][], R, C, is)
2 {
3 complex a[R][C] // R (length-C) rows, C (length-R) columns
4 for r:=0 to R-1 // FFT rows
5 { fft(a[r][]), C, is)
6 }
7 transpose( a[R][C] ) // in-place
8 for c:=0 to C-1 // FFT columns (which are rows now)
9 { fft(a[c][]), R, is)
10 }
11 transpose( a[C][R] ) // transpose back (note swapped R,C)
12 }
```

The transposing back at the end of the routine can be avoided if a back-transform will follow immediately
as typical for convolution. The back-transform must then be called with \(R\) and \(C\) swapped.

The generalization to higher dimensions is straightforward, C++ code is given in [FFT: fft/ndimfft.cc].

### 20.11 The matrix Fourier algorithm (MFA)

The matrix Fourier algorithm (MFA) is an algorithm for 1-dimensional FFTs that works for data lengths
\(n = RC\). It is quite similar to the row-column algorithm (relation \(20.10-4\)) for 2-dimensional FFTs. The
only differences are \(n\) multiplications with trigonometric factors and a final matrix transposition.

Consider the input array as a \(R \times C\)-matrix (\(R\) rows, \(C\) columns), the rows shall be contiguous in memory.
Then the matrix Fourier algorithm (MFA) can be stated as follows:

1. Apply a (length \(R\)) FFT on each column.
2. Multiply each matrix element (index \(r,c\)) by \(\exp(\sigma 2 \pi i r c/n)\)
3. Apply a (length \(C\)) FFT on each row.
4. Transpose the matrix.

Note the elegance! A variant of the MFA is called four step FFT in [23]. A trivial modification is obtained if the steps are executed in reversed order. The transposed matrix Fourier algorithm (TMFA) for the FFT:

1. Transpose the matrix.
2. Apply a (length $C$) FFT on each row of the matrix.
3. Multiply each matrix element (index $r,c$) by $\exp(\sigma 2 \pi i r c/n)$.
4. Apply a (length $R$) FFT on each column of the matrix.

A variant of the MFA that, apart from the transpositions, accesses the memory only in consecutive address ranges can be stated as

1. Transpose the matrix.
2. Apply a (length $C$) FFT on each row of the transposed matrix.
3. Multiply each matrix element (index $r,c$) by $\exp(\sigma 2 \pi i r c/n)$.
4. Transpose the matrix back.
5. Apply a (length $R$) FFT on each row of the matrix.
6. Transpose the matrix (if the order of the transformed data matters).

The ‘transposed’ version of this algorithm is identical. The performance will depend critically on the performance of the transposition routine.

It is usually a good idea to use factors of the data length $n$ that are close to $\sqrt{n}$. Of course one can apply the same algorithm for the row (or column) FFTs again: it can be an improvement to split $n$ into 3 factors (as close to $n^{1/3}$ as possible) if a length-$n^{1/3}$ FFT fits completely into the cache. Especially for systems where CPU clock speed is much higher than memory clock speed the performance may increase drastically, a speedup by a factor of three (even when compared to otherwise very well optimized FFTs) can sometimes be observed. Another algorithm that is efficient with large arrays is the localized transform as described in section 24.9 on page 525 for the Hartley transform.
Chapter 21

Algorithms for fast convolution

This chapter gives several FFT based algorithms for fast convolution. These are in practice the most important applications of the FFT. An efficient algorithm for the convolution of arrays that do not fit into the main memory (mass storage convolution) is given for both complex and real data. Further, weighted convolutions and their algorithms are introduced.

We describe how fast convolution can be used for computing the $z$-transform of sequences of arbitrary length. Another convolution based algorithm for the Fourier transform of arrays of prime length, Rader’s algorithm, is described at the end of the chapter.

Convolution algorithms based on the fast Hartley transform are described in section 24.7. The dyadic convolution, which is computed via the Walsh transform is treated in section 22.7.

21.1 Convolution

The cyclic convolution (or circular convolution) of two length-$n$ sequences $a = [a_0, a_1, \ldots, a_{n-1}]$ and $b = [b_0, b_1, \ldots, b_{n-1}]$ is defined as the length-$n$ sequence $h$ with elements $h_\tau$ as:

\[
h = a \circledast b
\]

\[
h_\tau := \sum_{x+y \equiv \tau \pmod{n}} a_x b_y
\]

The last equation may be rewritten as

\[
h_\tau := \sum_{x=0}^{n-1} a_x b_{(\tau-x) \pmod{n}}
\]

That is, indices $\tau - x$ wrap around, it is a cyclic convolution. A convenient way to illustrate the cyclic convolution of two sequences is shown in figure 21.1-A.

21.1.1 Direct computation

A C++ implementation of the computation by definition is [FXT: slow_convolution() in convolution/slowcnvl.h]:

```
1 template <typename Type>
2 void slow_convolution(const Type *f, const Type *g, Type *h, ulong n)
3     // (cyclic) convolution: h[] := f[] (*) g[]
```

[fxtbook draft of 2008-August-17]
Chapter 21: Algorithms for fast convolution

Figure 21.1-A: Semi-symbolic table of the cyclic convolution of two sequences (top). The entries denote where in the convolution the products of the input elements can be found (bottom).

```plaintext
++-- 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0
2: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0
3: 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2
4: 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3
5: 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4
6: 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5
7: 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6
++-- 0 1 2 3 (a)
0: 0 1 2 4 ...1: 1 3 5 <--= h[5] contains a[2]*b[1]
3: ...(b):
```

\[
\text{Figure 21.1-A: Semi-symbolic table of the cyclic convolution of two sequences (top). The entries denote where in the convolution the products of the input elements can be found (bottom).}
\]

```plaintext
// n := array length
for (ulong tau=0; tau<n; ++tau)
{
    Type s = 0.0;
    for (ulong k=0; k<n; ++k)
    {
        ulong k2 = tau - k;
        if ((long)k2<0) k2 += n; // modulo n
        s += (f[k]*g[k2]);
    }
    h[tau] = s;
}
```

The following version avoids the if statement in the inner loop:

```plaintext
for (ulong tau=0; tau<n; ++tau)
{
    Type s = 0.0;
    ulong k = 0;
    for (ulong k2=tau; k<=tau; ++k, --k2) s += (f[k]*g[k2]);
    for (ulong k2=n-1; k<n; ++k, --k2) s += (f[k]*g[k2]); // wrapped around
    h[tau] = s;
}
```

For length-\( n \) sequences this procedure involves proportional \( n^2 \) operations, therefore it is slow for large values of \( n \). For short lengths the algorithm is just fine. Unrolled routines will offer good performance, especially for convolutions of fixed length. For medium length convolutions the splitting schemes given in section 27.2 on page 552 and section 38.2 on page 824 are applicable.

### 21.1.2 Computation via FFT

The Fourier transform provides us with an efficient way to compute convolutions that only uses proportional \( n \log(n) \) operations. The convolution property of the Fourier transform is

\[
\mathcal{F}[a \circledast b] = \mathcal{F}[a] \mathcal{F}[b]
\] (21.1-3)
That is, convolution in original space is element-wise multiplication in Fourier space. The statement can be motivated as follows:

\[
\mathcal{F}[a]_k \mathcal{F}[b]_k = \sum_x a_x z^{kx} \sum_y b_y z^{ky} = \mathcal{F}[a \ast b]_k
\]  
(21.1-4a)

\[
= \sum_x a_x z^{kx} \sum_{\tau-x} b_{\tau-x} z^{k(\tau-x)} \quad \text{where} \quad y = \tau - x
\]  
(21.1-4b)

\[
= \sum_x \sum_{\tau-x} a_x z^{kx} b_{\tau-x} z^{k(\tau-x)} = \sum_{\tau} \left( \sum_x a_x b_{\tau-x} \right) z^{k\tau}
\]  
(21.1-4c)

\[
= \left( \mathcal{F} \left[ \sum_x a_x b_{\tau-x} \right] \right)_k = (\mathcal{F} [a \ast b])_k
\]  
(21.1-4d)

Rewriting relation 21.1-3 as

\[
a \ast b = \mathcal{F}^{-1} \left[ \mathcal{F}[a] \mathcal{F}[b] \right]
\]  
(21.1-5)

tells us how to proceed. We give pseudo code for the cyclic convolution of two complex valued sequences \(x[\] and \(y[\), result is returned in \(y[\):

```plaintext
1 procedure fft_cyclic_convolution(x[], y[], n)
2 {
3    complex x[0..n-1], y[0..n-1]
4    // transform data:
5    fft(x[], n, +1)
6    fft(y[], n, +1)
7    // convolution in transformed domain:
8    for i:=0 to n-1
9    {
10       y[i] := y[i] * x[i]
11    }
12    // transform back:
13    fft(y[], n, -1)
14    // normalize:
15    n1 := 1 / n
16    for i:=0 to n-1
17    {
18       y[i] := y[i] * n1
19    }
20 }
```

It is assumed that the procedure \(\text{fft}()\) does no normalization. For the normalization loop we precompute \(1/n\) and multiply as divisions are usually much slower than multiplications.

Relation 21.1-3 also holds for the more general \(z\)-transform (see section 21.5 on page 450). However, there is no (efficient) algorithm for the back-transform, so we cannot turn the relation

\[
\mathcal{Z}[a \ast b] = \mathcal{Z}[a] \mathcal{Z}[b]
\]  
(21.1-6)

into a practical algorithm for convolution.

### 21.1.3 Avoiding the revbin permutations

One can save the revbin permutations by observing that any DIF FFT is of the form

\[
\text{DIF_FFT_CORE}(f, n);
\text{revbin_permute}(f, n);
\]

and any DIT FFT is of the form

\[
\text{revbin_permute}(f, n);
\text{DIT_FFT_CORE}(f, n);
\]
Chapter 21: Algorithms for fast convolution

Thereby a convolution routine that uses DIF FFTs for the forward transform and DIT FFTs as backward transform can omit the revbin permutations as demonstrated in the C++ implementation for the cyclic convolution of complex sequences [FXT: fft_complex_convolution() in convolution/fftcocl.cc]:

```c
#define DIT_FFT_CORE fft_dit4_core_m1 // isign = -1
#define DIF_FFT_CORE fft_dif4_core_p1 // isign = +1

void
fft_complex_convolution(Complex * restrict f, Complex * restrict g,
ulong ldn, double v/*=0.0*/)
// (complex, cyclic) convolution: g[] := f[] (*) g[]
// (use zero padded data for usual convolution)
// ldn := base-2 logarithm of the array length
// Supply a value for v for a normalization factor != 1/n
{
const ulong n = (1UL<<ldn);
DIF_FFT_CORE(f, ldn);
DIF_FFT_CORE(g, ldn);
if ( v==0.0 ) v = 1.0/n;
for (ulong i=0; i<n; ++i)
{
Complex t = g[i] * f[i];
g[i] = t * v;
}
DIT_FFT_CORE(g, ldn);
}
```

The signs of the two FFTs must be different but are else immaterial.

The so-called auto convolution (or self convolution) of a sequence is defined as the convolution of a sequence with itself: \( h = a \ast a \). The corresponding procedure needs only two instead of three FFTs.

### 21.1.4 Linear (acyclic) convolution

In the definition of the cyclic convolution (relations 21.1-1a and 21.1-1b) one can distinguish between those summands where the \( x + y \) ‘wrapped around’ (i.e. \( x + y = n + \tau \)) and those where simply \( x + y = \tau \) holds. These are (following the notation in [96]) denoted by \( h^{(1)} \) and \( h^{(0)} \) respectively. We have

\[
\begin{align*}
h &= h^{(0)} + h^{(1)} \\
h^{(0)} &= \sum_{x\leq\tau} a_x b_{\tau-x} \\
h^{(1)} &= \sum_{x>\tau} a_x b_{n+\tau-x}
\end{align*}
\]

There is a simple way to separate \( h^{(0)} \) and \( h^{(1)} \) as the left and right half of a length-2 \( n \) sequence. This is just what the linear convolution (or acyclic convolution) does: Acyclic convolution of two (length-\( n \)) sequences \( a \) and \( b \) can be defined as that length-2 \( n \) sequence \( h \) which is the cyclic convolution of the zero padded sequences \( A \) and \( B \):

\[
A := [a_0, a_1, a_2, \ldots, a_{n-1}, 0, 0, \ldots, 0]
\]

Same for \( B \). Then the linear convolution is defined as

\[
\begin{align*}
h &= a \ast_{\text{lin}} b \\
h_\tau &= \sum_{x=0}^{2n-1} A_x B_{\tau-x} \quad \tau = 0, 1, 2, \ldots, 2n - 1
\end{align*}
\]

As an illustration consider the convolution of the sequence \([1, 1, 1, 1]\) with itself: its linear self convolution is the length-8 sequence \([h_0][h_1] = [1, 2, 3, 4, 3, 2, 1, 0]\), its cyclic self convolution is \([h_0 + h_1] = [4, 4, 4, 4]\).
Figure 21.1-B: Semi-symbolic table for the (length-31) linear convolution of two length-16 sequences.

The semi-symbolic table for the acyclic convolution is given in figure 21.1-B. The elements in the lower right triangle do not ‘wrap around’ anymore, they go to extra buckets. Note there are 31 buckets labeled 0...30.

Linear convolution is *polynomial multiplication*: let \( A = a_0 + a_1 x + a_2 x^2 + \ldots \), \( B = b_0 + b_1 x + b_2 x^2 + \ldots \) and \( C = AB = c_0 + c_1 x + c_2 x^2 + \ldots \) then

\[
c_k = \sum_{i+j=k} a_i b_j \tag{21.1-10}
\]

Chapter 27 on page 551 explains how fast convolution algorithms can be used for fast multiplication of multiprecision numbers.

The direct (slow) algorithm can be modified to compute just \( h^{(0)} \) or \( h^{(1)} \) [FXT: convolution/slowcnvlhalf.h]:

```cpp
1 template <typename Type>
2 void slow_half_convolution(const Type *f, const Type *g, Type *h, ulong n, int h01)
3 // Half cyclic convolution.
4 // Part determined by h01 which must be 0 or 1.
5 // n := array length
6 {
7    if ( 0==h01 ) // compute h0:
8    {
9        for (ulong tau=0; tau<n; ++tau)
10        {
11            Type s0 = 0.0;
12            for (ulong k=0, k2=tau; k<=tau; ++k, --k2) s0 += (f[k]*g[k2]);
13            h[tau] = s0;
14        }
15    } //...}
16    else // compute h1 (wrapped part):
17    {
18        for (ulong tau=0; tau<n; ++tau)
19        {
20            Type s1 = 0.0;
21            for (ulong k2=n-1, k=tau+1; k<n; ++k, --k2) s1 += (f[k]*g[k2]);
22            h[tau] = s1;
23        }
24    }
25 }
```

The cost is half of what it is for the linear convolution. With the FFT-based computation of the convolution the parts \( h^{(0)} \) and \( h^{(1)} \) can be isolated using weighted convolution algorithms that are given in section 21.3 on page 445. The cost is that of a linear convolution. No algorithm to compute just one of \( h^{(0)} \) or \( h^{(1)} \) that is significantly cheaper is known.
21.2 Correlation

The cyclic correlation (or circular correlation) of two real length-\(n\) sequences \(a = [a_0, a_1, \ldots, a_{n-1}]\) and \(b = [b_0, b_1, \ldots, b_{n-1}]\) is defined as the length-\(n\) sequence \(h\) with elements \(h_\tau\) as:

\[
h_\tau := \sum_{x - y \equiv \tau \mod n} a_x b_y \quad (21.2-1)
\]

The relation can be recast as

\[
h_\tau = \sum_{x=0}^{n-1} a_x b_{(\tau + x) \mod n} \quad (21.2-2)
\]

The semi-symbolic table for the (cyclic) correlation is shown in figure 21.2-A. For the computation of the linear (or acyclic) correlation the sequences have to be zero-padded as in the algorithm for the linear convolution. The semi-symbolic table is shown in figure 21.2-B.

The auto correlation (or self-correlation) is the correlation of a sequence with itself, the correlation of two distinct sequences is also called cross correlation. The term auto correlation function (ACF) is often used for the auto correlation sequence.
21.2: Correlation

21.2.1 Direct computation

A C++ implementation of the computation by the definition is [FXT: correlation/slowcorr.h]:

```cpp
1 template <typename Type>
2 void slow_correlation(const Type *f, const Type *g, Type * restrict h, ulong n)
3 // Cyclic correlation of f[], g[], both real-valued sequences.
4 // n := array length
5 {
6     for (ulong tau=0; tau<n; ++tau)
7         {
8             Type s = 0.0;
9             for (ulong k=0; k<n; ++k)
10                 {
11                     ulong k2 = k + tau;
12                     if ( k2>=n ) k2 -= n;
13                     s += (g[k]*f[k2]);
14                 }
15             h[tau] = s;
16         }
17 }
```

The if statement in the inner loop is avoided by the following version:

```cpp
1 for (ulong tau=0; tau<n; ++tau)
2 {
3     Type s = 0.0;
4     ulong k = 0;
5     for (ulong k2=tau; k2<n; ++k, ++k2) s += (g[k]*f[k2]);
6     for (ulong k2=0; k<n; ++k, ++k2) s += (g[k]*f[k2]);
7     h[tau] = s;
8 }
```

For the linear correlation one can avoid zero products:

```cpp
1 template <typename Type>
2 void slow_correlation0(const Type *f, const Type *g, Type * restrict h, ulong n)
3 // Linear correlation of f[], g[], both real-valued sequences.
4 // n := array length
5 // Version for zero padded data:
6 // f[k],g[k] == 0 for k=n/2 ... n-1
7 // n must be >=2
8 {
9     const ulong nh = n/2;
10     for (ulong tau=0; tau<nh; ++tau) // k2 == tau + k
11         {
12             Type s = 0;
13             for (ulong k=0, k2=tau; k2<nh; ++k, ++k2) s += (f[k]*g[k2]);
14             h[tau] = s;
15         }
16     for (ulong tau=nh; tau<n; ++tau) // k2 == tau + k - n
17         {
18             Type s = 0;
19             for (ulong k=n-tau, k2=0; k<nh; ++k, ++k2) s += (f[k]*g[k2]);
20             h[tau] = s;
21         }
22 }
```

The algorithm involves proportional \(n^2\) operations and is therefore slow with very long arrays.

21.2.2 Computation via FFT

A simple algorithm for fast correlation follows from the relation

\[
h_\tau = \mathcal{F}^{-1}[\mathcal{F}[a] \mathcal{F}[b]]
\]  

(21.2-3)

That is, use a convolution algorithm with one of the input sequences reversed (indices negated modulo \(n\)). For purely real sequences the relation is equivalent to complex conjugation of one of the inner transforms:

\[
h_\tau = \mathcal{F}^{-1}[\mathcal{F}[a]^{*} \mathcal{F}[b]]
\]  

(21.2-4)

[fxtbook draft of 2008-August-17]
For the computation of self-correlation the latter relation is the only reasonable way to go: first transform the input sequence, then multiply each element by its complex conjugate and finally transform back. A C++ implementation is [FXT: correlation/fftcorr.cc]:

```c
void
fft_correlation(double *f, double *g, ulong ldn)
// Cyclic correlation of f[], g[], both real-valued sequences.
// Result is written to g[].
// ldn := base-2 logarithm of the array length
{
    const ulong n=(1UL<<ldn);
    const ulong nh=(n>>1);
    fht_real_complex_fft(f, ldn); // real, imag part in lower, upper half
    fht_real_complex_fft(g, ldn);
    const double v = 1.0/n;
    g[0] *= f[0] * v;
    g[nh] *= f[nh] * v;
    for (ulong i=1,j=n-1; i<nh; ++i,--j) // real at index i, imag at index j
    {
        cmult_n(f[i], -f[j], g[i], g[j], v);
    }
    fht_complex_real_fft(g, ldn);
}
```

The function `cmult_n()` is defined in [FXT: aux0/cmult.h]:

```c
static inline void
cmult_n(double c, double s, double &u, double &v, double dn)
// {u,v} <--| {dn*(u*c-v*s), dn*(u*s+v*c)}
{
    double t = u*s+v*c;
    u *= c; u -= v*s; u *= dn; v = t*dn;
}
```

We note that relation 21.2.4 also holds for complex sequences.

### 21.2.3 Correlation and difference sets *

The linear auto-correlation of a sequence that contains zeros and ones only (a delta sequence) is the set of mutual differences of the positions of the ones, including multiplicity. An example:

| [1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0] <-- delta array R |
| [4, 2, 1, 2, 1, 0, 0, 1, 2, 1, 2] <-- linear auto correlation ACF |
| 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ... 26, <-- index |

Element zero of the ACF tells us that there are four elements in R (each element has difference zero to just itself). Element one tells us that there are two pairs of consecutive elements, it is identical to the last element (element minus one). There is just one pair of elements in R whose indices differ by two (elements two and minus two of the ACF), and so on. Note that the ACF does not tell us where the elements with a certain difference are.

The delta array with ones at the 7 positions 0, 3, 4, 12, 18, 23, and 25 has the ACF

| [7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, (+symm.)] |
| 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ... 26, <-- index |

That is, a ruler of length 26 with marks only at the 7 given positions can be used to measure most of the distances up to 26 (the smallest missing distance is 10). Further, no distance appears more than once. Sequences with this property are called Golomb rulers and they are very hard to find.

If we allow for two rulers then the set of mutual differences in positions is the cross correlation. For this setting analogues of Golomb rulers (that do not have any missing differences) can be found. We use dots for zeros:

| 11.11................................. <-- R1 |
| ............................11.11............. <-- cross correlation |

The rulers are binary representations of the evaluations $F(1/2)$ and $F(1/4)$ of a curious function given in section 36.10.1 on page 747.
21.3 Weighted Fourier transforms and convolutions

21.3.1 The weighted Fourier transform

We define a new kind of transform by slightly modifying the definition of the FT (formula 20.1-1a on page 403):

\[ c = \mathcal{W}_v[a] \]  \hspace{1cm} (21.3-1a)

\[ c_k := \sum_{x=0}^{n-1} v_x a_x z^k \quad v_x \neq 0 \quad \forall x \]  \hspace{1cm} (21.3-1b)

where \( z := e^{\sigma 2 \pi i/n} \). The sequence \( c \) shall be called (discrete) weighted transform of the sequence \( a \) with the weight (sequence) \( v \). Note the \( v_x \) that entered: the weighted transform with \( v_x = \frac{1}{\sqrt{n}} \forall x \) is just the usual Fourier transform. The inverse transform is

\[ a = \mathcal{W}_v^{-1}[c] \]  \hspace{1cm} (21.3-2a)

\[ a_x = \frac{1}{n v_x} \sum_{k=0}^{n-1} c_k z^{-x k} \]  \hspace{1cm} (21.3-2b)

This can be easily seen:

\[ \mathcal{W}_v^{-1} [\mathcal{W}_v[a]]_y = \frac{1}{n v_y} \sum_{k=0}^{n-1} \sum_{x=0}^{n-1} v_x a_x z^k z^{-y k} \]  \hspace{1cm} (21.3-3a)

\[ = \frac{1}{n} \sum_{k=0}^{n-1} \sum_{x=0}^{n-1} v_x \frac{1}{v_y} a_x z^k z^{-y k} \]  \hspace{1cm} (21.3-3b)

\[ = \frac{1}{n} \sum_{x=0}^{n-1} v_x \frac{1}{v_y} a_x \delta_{x,y} n = a_y \]  \hspace{1cm} (21.3-3c)

Obviously all \( v_x \) have to be invertible. That \( \mathcal{W}_v [\mathcal{W}_v^{-1} [a]] \) is also identity is apparent from the definitions.

Given an FFT routine it is trivial to set up a weighted Fourier transform. Pseudo code for the discrete weighted Fourier transform:

```
1 procedure weighted_ft(a[], v[], n, is)
2 {
3    for x:=0 to n-1
4        a[x] := a[x] * v[x]
5    }
6    fft(a[], n, is)
7 }
```

The inverse is essentially identical. Pseudo code for the inverse discrete weighted Fourier transform:

```
1 procedure inverse_weighted_ft(a[], v[], n, is)
2 {
3    fft(a[], n, -is)
4    for x:=0 to n-1
5        a[x] := a[x] / v[x]
6    }
```

The C++ implementations are given in [FXT: fft/weightedft.cc].
21.3.2 Weighted convolution

Define the weighted (cyclic) convolution \( h_v \) by

\[
    h_v = a \odot_v b \quad (21.3-4a)
\]

\[
    = W_v^{-1} [W_v [a] \, W_v [b]] \quad (21.3-4b)
\]

Then, for the special case \( v_x = V^x \), one has

\[
    h_v = h^{(0)} + V^n h^{(1)} \quad (21.3-5)
\]

Here \( h^{(0)} \) and \( h^{(1)} \) are defined as in relation 21.1-7a on page 440. It is not hard to see why this is: up to the final division by the weight sequence, the weighted convolution is just the cyclic convolution of the two weighted sequences, which is for the element with index \( \tau \) equal to

\[
    \sum_{x+y \equiv \tau \mod n} (a_x V^x) (b_y V^y) = \sum_{x \leq \tau} a_x b_{\tau-x} V^\tau + \sum_{x > \tau} a_x b_{n+\tau-x} V^{n+\tau} \quad (21.3-6)
\]

Final division of this element (by \( V^\tau \)) gives \( h^{(0)} + V^n h^{(1)} \) as stated.

The cases when \( V^n \) is some root of unity are particularly interesting: For \( V^n = \pm i = \pm \sqrt{-1} \) one obtains the so-called right-angle convolution:

\[
    h_v = h^{(0)} \mp i h^{(1)} \quad (21.3-7)
\]

This gives a nice possibility to directly use complex FFTs for the computation of a linear (acyclic) convolution of two real sequences: For length-\( n \) sequences the elements of the linear convolution with indices \( 0, 1, \ldots, n-1 \) are the real part of the result, the elements \( n, n+1, \ldots, 2n-1 \) are the imaginary part. Choosing \( V^n = -1 \) leads to the negacyclic convolution (or skew circular convolution):

\[
    h_v = h^{(0)} - h^{(1)} \quad (21.3-8)
\]

Cyclic, negacyclic and right-angle convolution can be understood as polynomial products modulo the polynomials \( z^n - 1, z^n + 1 \) and \( z^n \pm i \), respectively (see [212]).

C++ implementations of the weighted-, negacyclic- and right-angle (self) convolution are given in [FXT: convolution/weightedconv.cc].

<table>
<thead>
<tr>
<th>++-</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>1:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
</tr>
<tr>
<td>2:</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
</tr>
<tr>
<td>3:</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
</tr>
<tr>
<td>4:</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
</tr>
<tr>
<td>5:</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
</tr>
<tr>
<td>6:</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
</tr>
<tr>
<td>7:</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
</tr>
<tr>
<td>8:</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
<td>8-</td>
</tr>
<tr>
<td>9:</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
<td>8-</td>
<td>9-</td>
</tr>
<tr>
<td>10:</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
<td>8-</td>
<td>9-</td>
<td>10-</td>
</tr>
<tr>
<td>11:</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
<td>8-</td>
<td>9-</td>
<td>10-</td>
<td>11-</td>
</tr>
<tr>
<td>12:</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
<td>8-</td>
<td>9-</td>
<td>10-</td>
<td>11-</td>
<td>12-</td>
</tr>
<tr>
<td>13:</td>
<td>14</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
<td>8-</td>
<td>9-</td>
<td>10-</td>
<td>11-</td>
<td>12-</td>
<td>13-</td>
</tr>
<tr>
<td>14:</td>
<td>15</td>
<td>0-</td>
<td>1-</td>
<td>2-</td>
<td>3-</td>
<td>4-</td>
<td>5-</td>
<td>6-</td>
<td>7-</td>
<td>8-</td>
<td>9-</td>
<td>10-</td>
<td>11-</td>
<td>12-</td>
<td>13-</td>
<td>14-</td>
</tr>
</tbody>
</table>

**Figure 21.3-A:** Semi-symbolic table for the negacyclic convolution. The products that enter with negative sign are indicated with a postfix minus at the corresponding entry.

The semi-symbolic table for the negacyclic convolution is shown in figure 21.3-A. With right-angle convolution the minuses have to be replaced by \( i = \sqrt{-1} \) which means the wrap-around (i.e. \( h^{(1)} \)) elements...
go to the imaginary part. With real input one thereby effectively separates \( h^{(0)} \) and \( h^{(1)} \). Thereby the linear convolution of real sequences can be computed using the complex right-angle convolution.

With routines for both cyclic and negacyclic convolution the parts \( h^{(0)} \) and \( h^{(1)} \) can be computed as sum and difference, respectively. Thereby all expressions of the form \( \alpha h^{(0)} + \beta h^{(1)} \) where \( \alpha, \beta \in \mathbb{C} \) can be computed.

The direct (slow, proportional \( n^2 \)) computation can be obtained by a minimal modification of the non-weighted convolution algorithm [FXT: convolution/slowweightedcnvl.h]:

```cpp
template <typename Type>
void slow_weighted_convolution(const Type *f, const Type *g, Type *h, ulong n, Type w)
{
    // weighted (cyclic) convolution: \( h[] := f[] (*)_w g[] \)
    // n := array length
    for (ulong tau=0; tau<n; ++tau)
    {
        ulong k = 0;
        Type s0 = 0.0;
        for (ulong k2=tau; k<=tau; ++k, --k2) s0 += (f[k]*g[k2]);
        Type s1 = 0.0;
        for (ulong k2=n-1; k<n; ++k, --k2) s1 += (f[k]*g[k2]); // wrapped around
        h[tau] = s0 + s1*w;
    }
}
```

21.4 Convolution using the MFA

We give an algorithm for convolution that use the matrix Fourier algorithm (MFA, see section 20.11 on page 434). The MFA is used for the forward transform and the transposed algorithm (TMFA) for the backward transform. The elements of each row are assumed to lie contiguous in memory. For the sake of simplicity auto convolution is considered. The matrix FFT convolution algorithm:

1. Apply a (length \( R \)) FFT on each column. 
   (memory access with \( C \)-skips)
2. Multiply each matrix element (index \( r,c \)) by \( \exp(+\sigma 2 \pi i r c/n) \).
3. Apply a (length \( C \)) FFT on each row. 
   (memory access without skips)
5. Apply a (length \( C \)) FFT on each row (of the transposed matrix). 
   (memroy access is without skips)
6. Multiply each matrix element (index \( r,c \)) by \( \exp(-\sigma 2 \pi i r c/n) \).
7. Apply a (length \( R \)) FFT on each column (of the transposed matrix). 
   (memory access with \( C \)-skips)

Note that steps 3, 4, and 5 constitute a length-\( C \) convolution on each row.

C++ implementations of the cyclic and linear convolution using the described algorithm are given in [FXT: convolution/matrixfftcnvl.cc]. The code for self-convolution can be found in [FXT: convolution/matrixfftcnvla.cc].

With the weighted convolutions in mind we reformulate the matrix (self-) convolution algorithm

1. Apply a FFT on each column.

2. On each row apply the weighted convolution with \( V^C = e^{2 \pi i r / R} = 1^{r/R} \) where \( R \) is the total number of rows, \( r = 0..R-1 \) the index of the row, \( C \) the length of each row (equivalently, the total number columns)
3. Apply a FFT on each column (of the transposed matrix).

We first consider the special cases of two and three rows and then formulate an MFA-based algorithm for the convolution of real sequences.

21.4.1 The case \( R = 2 \)

Define \( s \) and \( d \) as the sums and differences of the lower and higher halves of a given sequence \( x \):

\[
\begin{align*}
\text{s} & := x^{(0/2)} + x^{(1/2)} \quad (21.4-1a) \\
\text{d} & := x^{(0/2)} - x^{(1/2)} \quad (21.4-1b)
\end{align*}
\]

Then the cyclic auto convolution of the sequence \( x \) can be obtained by two half-length convolutions of \( s \) and \( d \) as

\[
x \star x = \frac{1}{2} \left[ \text{s} \star \text{s} + \text{d} \star \text{d}, \quad \text{s} \star \text{d} \right] (21.4-2)
\]

where the symbols \( \star \) and \( \star_- \) stand for cyclic and negacyclic convolution, respectively (see section 21.3 on page 445). The equivalent formula for the cyclic convolution of two sequences \( x \) and \( y \) is

\[
x \star y = \frac{1}{2} \left[ \text{s}_x \star \text{s}_y + \text{d}_x \star_- \text{d}_y, \quad \text{s}_x \star \text{d}_y - \text{d}_x \star_- \text{d}_y \right] (21.4-3)
\]

where

\[
\begin{align*}
\text{s}_x & := x^{(0/2)} + x^{(1/2)} \quad (21.4-4a) \\
\text{d}_x & := x^{(0/2)} - x^{(1/2)} \quad (21.4-4b) \\
\text{s}_y & := y^{(0/2)} + y^{(1/2)} \quad (21.4-4c) \\
\text{d}_y & := y^{(0/2)} - y^{(1/2)} \quad (21.4-4d)
\end{align*}
\]

Now use the fact that an linear convolution is computed by a cyclic convolution of zero-padded sequences whose upper halves are simply zero, so \( s_x = d_x = x \) and \( s_y = d_y = y \). Then relation 21.4-3 reads:

\[
x \star_{\text{lin}} y = \frac{1}{2} \left[ x \star y + x \star_- y, \quad x \star y - x \star_- y \right] (21.4-5)
\]

And for the acyclic auto convolution:

\[
x \star_{\text{lin}} x = \frac{1}{2} \left[ x \star x + x \star_- x, \quad x \star x - x \star_- x \right] (21.4-6)
\]

The lower and upper halves of the linear convolution can be obtained from the sum and difference of the cyclic and the negacyclic convolution.

21.4.2 The case \( R = 3 \)

Let \( \omega = \frac{1}{2} (1 + i \sqrt{3}) \) and define

\[
\begin{align*}
\text{A} & := x^{(0/3)} + x^{(1/3)} + x^{(2/3)} \quad (21.4-7a) \\
\text{B} & := x^{(0/3)} + \omega x^{(1/3)} + \omega^2 x^{(2/3)} \quad (21.4-7b) \\
\text{C} & := x^{(0/3)} + \omega^2 x^{(1/3)} + \omega x^{(2/3)} \quad (21.4-7c)
\end{align*}
\]

Then, if \( h := x \star x \), one has

\[
\begin{align*}
h^{(0/3)} & = \text{A} \star A + \text{B} \star (\omega) \text{B} + \text{C} \star (\omega^2) \text{C} \quad (21.4-8a) \\
h^{(1/3)} & = \text{A} \star A + \omega^2 \text{B} \star (\omega) \text{B} + \omega \text{C} \star (\omega^2) \text{C} \quad (21.4-8b) \\
h^{(2/3)} & = \text{A} \star A + \omega \text{B} \star (\omega) \text{B} + \omega^2 \text{C} \star (\omega^2) \text{C} \quad (21.4-8c)
\end{align*}
\]
For real valued data $C$ is the complex conjugate (cc.) of $B$ and (with $\omega^2 = \text{cc.}\omega$) $B \odot (\omega^2) \odot C$ and therefore every $B \odot (\omega)$-term is the cc. of the $C \odot (\omega)$-term in the same line. Is there a nice and general scheme for real valued convolutions based on the MFA? Read on for the positive answer.

### 21.4.3 Convolution of real valued data using the MFA

Consider the MFA-algorithm for the cyclic convolution as given in section 21.4 on page 447 but with real input data: for row 0 which is real after the column FFTs one needs to compute the usual cyclic convolution; for row $R/2$ which is also purely real after the column FFTs a negacyclic convolution is needed (for odd values of $R$ there is no such row and no negacyclic convolution is needed), the code for negacyclic convolution is given in section 24.8 on page 525.

All other weighted convolutions involve complex computations, but it is easy to see how to reduce the work by 50 percent: as the result must be real the data in row number $R - r$ must, because of the symmetries of the real and imaginary part of the (inverse) Fourier transform of real data, be the complex conjugate of the data in row $r$. Therefore one can use real FFTs (R2CFTs) for all column-transforms for step 1 and half-complex to real FFTs (C2RFTs) for step 3.

Let the computational cost of a cyclic (real) convolution be $q$, then

- For even values of $R$ one must perform one cyclic (row 0), one negacyclic (row $R/2$) and $R/2 - 2$ complex weighted convolutions (rows $1, 2, \ldots, R/2 - 1$)

- For $R$ odd one must perform 1 cyclic (row 0) and $(R - 1)/2$ complex weighted convolutions (rows $1, 2, \ldots, (R - 1)/2$)

Now assume, slightly simplifying, that the cyclic and the negacyclic real convolution involve the same number of computations and that the cost of a weighted complex convolution is twice that. Then in both cases above the total work is exactly half of that for the complex case, which is what one expects from a real world real valued convolution algorithm.

For the computation of the linear convolution one can use the right angle convolution (and complex FFTs in the column passes), see section 21.3 on page 445.

### 21.4.4 Mass storage convolution using the MFA

Algorithms on data sets that do not fit into physical RAM are sometimes called external or out of core algorithms. Simply using the virtual memory mechanism of the operating system is not an option, eternal hard disk activity would be the consequence in most cases. We give a method for the mass storage convolution. It is based on the matrix FFT convolution algorithm given in section 21.4 on page 447. The number of disk seeks has to be kept minimal because these are slow operations which degrade performance unacceptably if they occur too often.

The crucial modification of the use of the MFA is to not choose $R$ and $C$ as close as possible to $\sqrt{n}$ as is usually done. Instead one chooses $R$ to be minimal so that the row length $C$ corresponds to the biggest data set that fits into the available RAM. We now analyze how the number of seeks depends on the choice of $R$ and $C$: In what follows it is assumed that the data lies in memory as row 0, row 1, \ldots, row $R - 1$. In other words, the elements of each rows lie contiguous in memory. Further let $\alpha \geq 2$ be the number of times the data set exceeds the available RAM size.

In step 1 and 3 of the convolution algorithm given in section 21.4 one reads from disk (row by row, involving $R$ seeks) the number of columns that just fit into RAM, does the (many, short) column-FFTs, writes back (again $R$ seeks), and proceeds to the next block; this happens for $\alpha$ of these blocks, giving a total of $4\alpha R$ seeks for steps 1 and 3.
In step 2 one reads ($\alpha$ times) blocks of one or more rows, which lie in contiguous portions of the disk, perform the FFT on the rows, and write back to disk. This gives a total of $2\alpha$ seeks for step 2.

Thereby there are $2\alpha + 4\alpha R$ seeks during the whole computation, which is minimized by the choice of maximal $C$. This means that one chooses a shape of the matrix so that the rows are as big as possible subject to the constraint that they have to fit into main memory, which in turn means there are $R = \alpha$ rows, leading to an optimal seek count of $K = 2\alpha + 4\alpha^2$.

Let $S_D$ be the seek time of the hard disk in seconds, $W_D$ be the disk transfer rate in megabyte per second. Further let $A$ be the amount of available RAM in megabytes. Then the total time spent for disk operations is the sums of the time spent in seeks and the time for reading and writing:

$$T = K S_D + 6 \frac{\alpha A}{W_D}$$

(21.4-9a)

$$= (2\alpha + 4\alpha^2) S_D + 6 \frac{\alpha A}{R_D}$$

(21.4-9b)

We give two examples for an FFT whose size that exceeds the available RAM by a factor of $\alpha = 16$ (so the number of seeks is $k = 1056$).

With a machine where the disk seek takes 10 milliseconds ($S_D = 0.010$) about 10 seconds are needed for the seeks. Further assume we have $A = 64$ Megabytes of RAM available so the transform size is 1 GB corresponding to 128 million (double precision) floats. The disk transfer rate shall be $W_D = 10$ MB/sec. Then the overhead for the read and write would amount to $6 \cdot 1024 MB/(10 MB/sec) \approx 615 sec$ or approximately 10 minutes. So the total disk activity takes approximately 10.5 minutes.

With a workstation machine with $S_D = 0.005$ sec (5 milliseconds), $W_d = 50$ MB/sec, and 512 MB of RAM: the transform size is 8 GB (1 G doubles) and we obtain $T = 5.28 + 983.04 = 988.32$ or approximately 16.5 minutes.

In both cases the CPU will be busy for several minutes which is in the same order as the time for the disk activity. Therefore, when multi-threading is available, one may want to use a double buffer variant: Choose the row length so that it fits twice into the RAM workspace; then let always one (CPU-intensive) thread do the FFTs in one of the scratch spaces and the other (hard disk intensive) thread write back the data from the other scratch space and read the next data to be processed. With not too small main memory (and not too slow hard disk) and some fine tuning double buffering can to keep the CPU busy during much of the hard disk operations. Thereby most of the disk time can be ‘hidden’ as the CPU activity does not stall waiting for data.

The mass storage convolution as described was used for the calculation of the number

$$9^{99} \approx \ 0.4281247 \cdot 10^{369.693.100}$$

(21.4-10)

on a 32-bit machine in 1999. The computation used two files of size 2 Gigabytes each and took less than eight hours on a system with an AMD K6/2 CPU at 366 MHz with 66 MHz memory. The log-file of the computation is [hfloat: examples/run1-pow999.txt]. The computation did not use the double-buffer technique.

### 21.5 The z-transform (ZT)

In this section we will learn a technique to compute the Fourier transform by a cyclic convolution. In fact, the transform computed is the z-transform, a more general transform that in a special case is identical to the Fourier transform.
The discrete z-transform (ZT) of a length-$n$ sequence $a$ is a length-$n$ sequence $c$ defined by

$$c = \mathcal{Z}[a]$$

$$c_k := \sum_{x=0}^{n-1} a_x z^{kx}$$

The z-transform is a linear transformation. It is not an orthogonal transformation unless $z$ is a root of unity. For $z = e^{\pm 2\pi i/n}$ the z-transform specializes to the discrete Fourier transform. An important property is the convolution property:

$$\mathcal{Z}[a \ast b] = \mathcal{Z}[a] \mathcal{Z}[b]$$

Convolution in original space corresponds to ordinary (element-wise) multiplication in z-space. This can be turned into an efficient convolution algorithm for the special case of the Fourier transform but not in general because no efficient algorithm for the inverse transform is known.

### 21.5.1 Computation via convolution (Bluestein’s algorithm)

Noting that

$$x_k = \frac{1}{2} \left( x^2 + k^2 - (k - x)^2 \right)$$

we find, for element $c_k$ of the Fourier transform of the sequence $a$,

$$c_k = \sum_{x=0}^{n-1} a_x z^{xk} = z^{k^2/2} \left[ \sum_{x=0}^{n-1} (a_x z^{x^2/2}) z^{-(k-x)^2/2} \right]$$

The expression in brackets is a cyclic convolution of the sequence $a_x z^{x^2/2}$ with the sequence $z^{-x^2/2}$.

This leads to the algorithm for the *chirp z-transform*:

1. Multiply the sequence $a$ element-wise with $z^{x^2/2}$.
2. Convolve the resulting sequence with the sequence $z^{-x^2/2}$.
3. Multiply element-wise with the sequence $z^{k^2/2}$.

The above algorithm constitutes a fast algorithm for the ZT because fast convolution is possible via FFT. The idea is due to Bluestein [47], a detailed description of the algorithm for computing the Bluestein FFT is given in [255].

### 21.5.2 Arbitrary length FFT by ZT

The length $n$ of the input sequence $a$ for the fast z-transform is not limited to highly composite values: For values of $n$ where a FFT is not feasible pad the sequence with zeros up to a length $L$ with $L \geq 2n$ such that a length-$L$ FFT can be computed (highly composite $L$, for example a power of two).

As the Fourier transform is the special case $z = e^{\pm 2\pi i/n}$ of the ZT the chirp-ZT algorithm constitutes an FFT algorithm for sequences of arbitrary length.

The transform takes a few times more than an direct FFT. The worst case (if only FFTs for $n$ a power of 2 are available) is $n = 2^n + 1$: one must perform three FFTs of length $L = 2^{n+1} \approx 2n$ for the computation of the convolution. So the total work amounts to about 6 times the work a FFT of length $n$. It is possible to lower this ‘worst case factor’ to 3 by using highly composite $L$ slightly greater than $n$. 

[fxtbook draft of 2008-August-17]
For multiple computations of $z$-transforms of the same length one may want to store the Fourier transform of the sequence $z^{k^2/2}$ as it does not change. Thereby the worst case is reduced to a factor 2 with highly composite FFTs and 4 if FFTs are available for powers of two only.

A C++ implementation of the Fourier transform for sequences of arbitrary length is given in [FXT: chirpzt/fftarblen.cc]:

```c++
void fft_arblen(Complex *x, ulong n, int is)
{
    const ulong ldnn = 1 + ld( (n << 1) - 1 );
    const ulong nn = (1UL<<ldnn); // smallest power of 2 >= 2*n
    Complex *f = new Complex[nn];
    copy(x, f, n);
    null(f+n, nn-n);
    Complex *w = new Complex[nn];
    make_fft_chirp(w, n, nn, is);
    multiply(f, n, w);
    double *dw = (double *)w;
    for (ulong k=1; k<2*n; k+=2) dw[k] = -dw[k]; // =*= make_fft_chirp(w, n, nn, -is);
    fft_complex_convolution(w, f, ldnn);
    if ( n & 1 ) subtract(f, n, f+n); // odd n: negacyclic convolution
    else add(f, n, f+n); // even n: cyclic convolution
    make_fft_chirp(w, n, nn, is);
    multiply(w, n, f);
    copy(w, x, n);
    delete [] w;
    delete [] f;
}
```

The auxiliary routine `make_fft_chirp()` is

```c++
static inline void make_fft_chirp(Complex *w, ulong n, ulong nn, int is)
{
    double phi = 1.0*is*M_PI/n; // == (i*2*Pi/n)/2
    // where i = sqrt(-1)
    // For k=0..n-1: w[k] := exp( is * k*k * (i*2*PI/n)/2 )
    // For k=n..nn-1: w[k] = 0
    double phi = 1.0*is*M_PI/n; // == (i*2*Pi/n)/2
    ulong k2 = 0, n2 = 2*n;
    for (ulong k=0; k<n; ++k)
    {
        w[k] = SinCos(phi*k2);
        k2 += (2*k+1);
        if ( k2>n2 ) k2 = n2;
        // here: k2 == (k*k) mod 2*n;
    }
    null(w+n, nn-n);
}
```

### 21.5.3 Fractional Fourier transform by ZT

The $z$-transform with $z = e^{α^2 \pi i/n}$ is called the fractional Fourier transform in [20]. We note that the term is usually used for the fractional order transform given as relation 24.14-6 on page 532, see also [220] ch.13.

For $α = ±1$ one again obtains the usual Fourier transform. The fractional Fourier transform can be used for the computation of the Fourier transform of sequences with only few nonzero elements and for the exact detection of frequencies that are not integer multiples of the lowest frequency of the DFT.

A C++ implementation of the fractional Fourier transform for sequences of arbitrary length is given in [FXT: chirpzt/fftfract.cc]:

[fxtbook draft of 2008-August-17]
void fft_fract(Complex *x, ulong n, double v)
  // Fractional (fast) Fourier transform.
{
  const ulong ldnn = 1 + ld((n << 1) - 1);
  const ulong nn = (1UL<<ldnn); // smallest power of 2 >= 2*n
  Complex *f = new Complex[nn];
  copy(x, f, n);
  null(f+n, nn-n);
  Complex *w = new Complex[nn];
  make_fft_fract_chirp(w, v, n, nn);
  for (ulong j=0; j<n; ++j) f[j] *= w[j];
  for (ulong j=0; j<nn; ++j) w[j] = conj(w[j]);
  fft_complex_convolution(w, f, ldnn);
  make_fft_fract_chirp(w, v, n, nn);
  for (ulong j=0; j<n; ++j) w[j] *= f[j];
  copy(w+n, x, n);
  delete [] w;
  delete [] f;
}

The auxiliary routine make_fft_fract_chirp() is

static inline void make_fft_fract_chirp(Complex *w, double v, ulong n, ulong nn)
  // For k=0..nn-1: w[k] == exp(v*sqrt(-1)*k*k*2*pi*/n/2)
{
  const double phi = v*2.0*M_PI/n/2;
  ulong n2 = 2*n;
  ulong np=0;
  for (ulong k=0; k<nn; ++k)
  {
    w[k] = SinCos(phi*np);
    np += ((k<<1)+1); // np == (k*k)%n2
    if ( np>=n2 ) np -= n2;
  }
}

21.6 Prime length FFTs

For the computation of FFTs for sequences whose length is prime we can exploit the existence of primitive roots. We will be able to express the transform of all but the first element as a cyclic convolution of two sequences whose length is reduced by one.

Let \( p \) be prime, then an element \( g \) exists so that the least positive exponent \( e \) so that \( g^e \equiv 1 \pmod{p} \) is \( e = p - 1 \). The element \( g \) is called a generator (or primitive root) modulo \( p \) (see section 37.5 on page 772). Every nonzero element modulo \( p \) can uniquely be expressed as a power \( g^e \) where \( 0 \leq e < p - 1 \). For example, a generator modulo \( p = 11 \) is \( g = 2 \), its powers are:

\[
\begin{align*}
1 & \equiv 1 \\
2 & \equiv 2 \\
4 & \equiv 8 \\
8 & \equiv 5 \\
10 & \equiv -1 \\
6 & \equiv 9 \\
7 & \equiv 7 \\
9 & \equiv 3 \\
3 & \equiv 6 \\
5 & \equiv g^{p-1} \equiv 1
\end{align*}
\]

Likewise, we can express any nonzero element as a negative power of \( g \). Let \( h = g^{-1} \), then with our example, \( h \equiv 6 \) and

\[
\begin{align*}
0 & \equiv 1 \\
1 & \equiv 6 \\
2 & \equiv 3 \\
3 & \equiv 7 \\
4 & \equiv 9 \\
5 & \equiv 10 \equiv -1 \\
6 & \equiv 5 \\
7 & \equiv 8 \\
8 & \equiv 4 \\
9 & \equiv 2 \\
10 & \equiv g^{-1} \equiv 1
\end{align*}
\]

This is just the reversed sequence of values. Let the \( C \) be the Fourier transform of length-\( p \) sequence \( A \):

\[
C_k = \sum_{x=0}^{p-1} A_x W^{\sigma x k} \tag{21.6-1}
\]

[fxtbook draft of 2008-August-17]
where $W = \exp (2i\pi/p)$ and $\sigma = \pm 1$ is the sign of the transform. We split the computation of the Fourier transform in two parts, we compute the first element of the transform as

$$C_0 = \sum_{x=0}^{p-1} A_x$$  \hspace{1cm} (21.6-2)

Now it remains to compute, for $1 \leq k \leq p-1$,

$$C_k = A_0 + \sum_{x=1}^{p-1} A_x W^{\sigma x k}$$  \hspace{1cm} (21.6-3)

Note the lower index of the sum. We write $k \equiv g^e$ and $x \equiv g^{-f}$ (modulo $p$), so

$$C(g^e) - A_0 = \sum_{f=0}^{p-2} A(g^{-f}) W^{\sigma (g^{-f}) (g^e)} = \sum_{f=0}^{p-2} A(g^{-f}) W^{\sigma (g^{-f})}$$  \hspace{1cm} (21.6-4)

The sum is a cyclic convolution of the sequences $W(g^w)$ and $A(g^{-w})$ where $0 \leq w \leq p-2$. The main algorithm (ignoring the constant terms $A_0$ and $C_0$) can be outlined as follows:

1. Compute $A^*$ and $W^*$ by permuting the sequences $A$ and $W$.
2. Compute $C^*$ the cyclic convolution of $A^*$ and $W^*$.
3. Compute $W$ by permuting $W^*$.

The method is given in [223], it is called Rader's algorithm. We implement it in pari/gp:

```plaintext
1 ft_rader(a, is=+1)=
2 \Fourier transform for prime lengths (Rader's algorithm)
3 {
4 local(n, a0, c0, g, w);
5 local(c, ixp, ixm, pa, pw, t);
6 n = length(a);
7 a0 = a[1]; c0 = sum(j=1, n, a[j]); \ constant terms
8 \ prepare permutations:
9 g = znprimroot(n); ixp = vector(n, j, component(g^(j-1), 2) );
10 g = g^(-1); ixm = vector(n, j, component(g^(j-1), 2) );
11 \ permute sequence W:
12 w = is*2*I*Pi/n; pw = vector(n-1, j, exp(w*ixp[j]) );
13 \ permute sequence A:
14 pa = vector(n-1); for (j=1, n-1, pa[j]=a[1+ixp[1+n-j]] );
15 \ cyclic convolution of permuted sequences:
16 t = cconv(pa, pw); \ cyclic convolution
17 \ set C_0, and add A_0 to each C_k:
18 c = vector(n); c[1] = c0; for (k=1, n-1, c[1+k]=t[k]+a0);
19 \ permute to obtain result:
20 t = vector(n); t[1] = c[1]; for (k=2, n, t[1+ixp[k-1]]=c[k]);
21 return( t );
22 }
```

With a (slow) implementation of the cyclic convolution and DFT we can check whether the method works by comparing the results:

```plaintext
1 cconv(a, b)=
2 /* Cyclic convolution (direct computation, n^2 operations) */
3 /* Example: cconv([a,b],[c,d]) ==> [b*d + c*a, a*d + c*b] */
4 {
5 local(n, f, s, k, k2);
6 n = length(a);
7 f = vector(n);
```

[txtbook draft of 2008-August-17]
In order to turn the algorithm into a fast Fourier transform we need to compute the convolution via fast transforms of length \((p - 1)\). This is trivially possible when \(p - 1 = 2^\ell\), for example when \(p = 5\) or \(p = 17\). As \(p - 1\) is always divisible by two, we can split at least once. For \(p = 11\) we have \((p - 1)/2 = 5\) so we can again use Rader’s algorithm and length-4 transforms.

The method can be used to generate code for short (prime) length FFTs. One should precompute the permuted and transformed sequence of the powers of the primitive root \(W\). Thereby only two FFTs of length \((p - 1)\) will be needed for a length-\(p\) transform.
Chapter 22

The Walsh transform and its relatives

We describe several variants of the Walsh transform, sometimes called Walsh-Hadamard transform or just Hadamard transform. The Walsh transform has the same complexity as the Fourier transform but does not involve any multiplications. In fact, one can obtain a Walsh transform routine by removing all multiplications (with sines and cosines) in a given FFT routine.

We also give related transforms like the slant transform and the Reed-Muller transform. The dyadic convolution that can be computed efficiently by the Walsh transform is introduced.

22.1 The Walsh transform: Walsh-Kronecker basis

How to make a Walsh transform out of your FFT:
‘Replace exp(anything) by 1, done.’

Removing all exp(anything) from the radix-2, decimation in time Fourier transform we obtain

```c
void slow_walsh_wak_dit2(double *f, ulong ldn)
{// (this routine has a problem)
 ulong n = (1UL<<ldn);
 for (ulong ld=1; ld<=ldn; ++ld)
  {
   const ulong m = (1<<ld);
   const ulong mh = (m>>1);
   for (ulong j=0; j<mh; ++j)
    {
      for (ulong r=0; r<n; r+=m)
       {
        const ulong t1 = r+j;
        const ulong t2 = t1+mh;
        double u = f[t1];
        double v = f[t2];
        f[t1] = u+v;
        f[t2] = u-v;
       }
  }
}
```

The transform involves proportional \( n \log_2(n) \) additions (and subtractions) and no multiplication at all. The transform, as given, is its own inverse up to a factor \( 1/n \). The Walsh transform of integer input is integral.
Figure 22.1-A: Basis functions for the Walsh transform (Walsh-Kronecker basis). Asterisks denote the value +1, blank entries denote −1.

As the slow in the name shall suggest, the implementation has a problem as given. The memory access pattern is highly non-local. Let’s make a slight improvement: here we just took the radix-2 DIT FFT code from section 20.3.1.3 on page 409 and threw away all trigonometric computations (and multiplications). But the swapping of the inner loops, that we did for the FFT in order to save trigonometric computations is now of no advantage anymore. So we try the following [FXT: `walsh_wak_dit2()` in `walsh/walshwak2.h`]:

```cpp
template <typename Type>
void walsh_wak_dit2(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-2 decimation in frequency (DIF) algorithm.
{
    ulong n = (1UL<<ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=mh)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0; j<mh; ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The performance impact is quite drastic. For \(n = 2^{21}\) (and type double, 16 MByte of memory) it gives a

[fxtbook draft of 2008-August-17]
speedup by a factor of about eight. For smaller lengths the ratio approaches one.

The data flow diagram (butterfly diagram) for the radix-2 decimation in time (DIT) algorithm is shown in figure 22.1-B. The figure was created with the program [FXT: fft/butterfly-texpic-demo.cc]. The diagram for the decimation in frequency (DIF) algorithm is obtained by reversing the order of the steps. In the code, only the outermost loop has to be changed [FXT: walsh_wak_dif2() in walsh/walshwak2.h]:

```c
template<typename Type>
void walsh_wak_dif2(Type *f, ulong ldn)
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0; j<mh; ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

A function that computes the $k$-th base function of the transform is [FXT: walsh_wak_basefunc() in walsh/walshbasefunc.h]:

```c
template<typename Type>
void walsh_wak_basefunc(Type *f, ulong n, ulong k)
{
    for (ulong i=0; i<n; ++i)
    {
```
ulong x = i & k;
x = parity(x);
}
f[i] = ( 0==x ? +1 : -1 );
}

The basis functions are shown in figure 22.1-A. Note that the lowest row is (the signed version of) the Thue-Morse sequence, see section 1.16.1 on page 44.

Multi-dimensional Walsh transform

If one applies the row-column algorithm (see section 20.10.2 on page 433) to compute a two-dimensional \( n \times m \) Walsh transform then the result is exactly the same as with a 1-dimensional \( n \cdot m \) transform. That is, algorithmically nothing needs to be done for multidimensional Walsh transforms: a \( k \)-dimensional \( n_1 \times n_2 \times \ldots \times n_k \)-transform is identical to a 1-dimensional \( n_1 \cdot n_2 \cdot \ldots \cdot n_k \)-transform. The length-2\(^n\) Walsh transform is identical to a \( n \)-dimensional length-2 Fourier transform.

22.2 Eigenvectors of the Walsh transform *

\[
\begin{align*}
0: & \quad [ +5 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 ]
1: & \quad [ +1 +3 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 ]
2: & \quad [ +1 +1 +3 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 ]
3: & \quad [ +1 -1 -1 +5 +1 -1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 ]
4: & \quad [ +1 +1 +1 +1 +3 -1 -1 -1 +1 +1 +1 +1 -1 -1 -1 -1 ]
5: & \quad [ +1 -1 +1 -1 -1 +5 -1 +1 +1 -1 +1 -1 +1 -1 +1 +1 ]
6: & \quad [ +1 +1 -1 -1 +1 -1 +1 -1 -1 +1 -1 +1 -1 +1 +1 -1 ]
7: & \quad [ +1 -1 -1 +1 +1 -1 +1 -1 +1 -1 +1 +1 -1 +1 -1 +1 ]
8: & \quad [ +1 +1 +1 +1 +1 +1 +1 +1 +1 -5 -1 -1 -1 -1 -1 -1 -1 -1 ]
9: & \quad [ +1 -1 +1 -1 +1 -1 +1 -1 -3 -1 +1 -1 +1 +1 +1 +1 ]
10: & \quad [ +1 +1 -1 -1 +1 +1 -1 -1 -3 +1 +1 -1 -1 -1 -1 -1 ]
11: & \quad [ +1 -1 -1 +1 +1 -1 -1 -1 +1 +1 +1 +1 +1 +1 +1 +1 ]
12: & \quad [ +1 +1 +1 +1 +1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ]
13: & \quad [ +1 -1 -1 -1 +1 -1 -1 -1 +1 +1 +1 +1 +1 +1 +1 +1 ]
14: & \quad [ +1 +1 -1 -1 -1 -1 +1 -1 +1 -1 +1 +1 +1 +1 +1 +1 ]
15: & \quad [ +1 -1 -1 +1 +1 -1 +1 -1 -1 +1 +1 +1 +1 +1 +1 +1 ]
\end{align*}
\]

Figure 22.2-A: Eigenvectors of the length-16 Walsh transform (Walsh-Kronecker basis) as row vectors. The eigenvalues are +1 for the vectors 0...7 and −1 for the vectors 8...16. Linear combinations of vectors with the same eigenvalue \( e \) are again eigenvectors with eigenvalue \( e \).

The Walsh transforms are self-inverse, so their eigenvalues can only be plus or minus one. Let \( a \) be a sequence and let \( W(a) \) denote the Walsh transform of \( a \). Set
\[
u_+ := W(a) + a \quad (22.2-1)
\]
Then
\[
W(u_+) = W(W(a)) + W(a) = a + W(a) = +1 \cdot u_+ \quad (22.2-2)
\]
That is, \( u_+ \) is an eigenvector of \( W \) with eigenvalue +1. Equivalently, \( u_- := W(a) - a \) is an eigenvector with eigenvalue −1. Thereby, two eigenvectors can be obtained from an arbitrary nonzero sequence.
We are interested in a simple routine that for a Walsh transform of length \( n \) gives a set of \( n \) eigenvectors that span the \( n \)-dimensional space. With a routine that computes the \( k \)-th basis function of the transform we can obtain an eigenvector efficiently by simply adding a delta peak at position \( k \) to the basis function. The delta peak has to be scaled according to whether a positive or negative eigenvalue is desired and according to the normalization of the transform.

A suitable routine for the Walsh-Kronecker basis (whose basis functions are given in figure 22.1-A on page 458) is

```plaintext
void walsh_wak_eigen(double *v, ulong ldn, ulong k) // Eigenvectors of the Walsh transform (walsh_wak).
// Eigenvalues are +1 if k<n/2, else -1
{
    ulong n = 1UL << ldn;
    walsh_wak_basefunc(v, n, k);
    double d = sqrt(n);
    v[k] += (k<n/2 ? +d : -d);
}
```

This routine is given in [FXT: walsh/walsheigen.cc]. Figure 22.2-A was created with the program [FXT: fft/walsh-eigenvec-demo.cc].

Note that with the unnormalized transforms the eigenvalues are \( \pm \sqrt{n} \).

### 22.3 The Kronecker product

The length-2 Walsh transform is equivalent to the multiplication of a 2-component vector by the matrix

\[
W_2 = \begin{bmatrix}
+1 & +1 \\
+1 & -1
\end{bmatrix}
\]  

(22.3-1)

The length-4 Walsh transform corresponds to

\[
W_4 = \begin{bmatrix}
+1 & +1 & +1 & +1 \\
+1 & -1 & +1 & -1 \\
+1 & -1 & -1 & -1 \\
+1 & -1 & -1 & +1
\end{bmatrix}
\]  

(22.3-2)

One might be tempted to write

\[
W_4 = \begin{bmatrix}
W_2 & +W_2 & +W_2 & -W_2
\end{bmatrix}
\]  

(22.3-3)

This idea can indeed be turned into a well-defined notation which is quite powerful when dealing with orthogonal transforms and their fast algorithms. Let \( A \) be an \( m \times n \) matrix

\[
A = \begin{bmatrix}
a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\
a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{bmatrix}
\]  

(22.3-4)

then the (right) Kronecker product (or tensor product) with a matrix \( B \) is

\[
A \otimes B := \begin{bmatrix}
a_{0,0}B & a_{0,1}B & \cdots & a_{0,n-1}B \\
a_{1,0}B & a_{1,1}B & \cdots & a_{1,n-1}B \\
\vdots & \vdots & \ddots & \vdots \\
a_{m-1,0}B & a_{m-1,1}B & \cdots & a_{m-1,n-1}B
\end{bmatrix}
\]  

(22.3-5)
For a scalar factor \( \alpha \) the following relations are immediate:

\[
\alpha \mathbf{A} \otimes \mathbf{B} = \alpha (\mathbf{A} \otimes \mathbf{B}) \quad (22.3-6a)
\]

\[
\mathbf{A} \otimes (\alpha \mathbf{B}) = \alpha (\mathbf{A} \otimes \mathbf{B}) \quad (22.3-6b)
\]

The next relations are the same as for the ordinary matrix product. Distributivity (the matrices on both sides of a plus sign must be of the same dimensions):

\[
(\mathbf{A} + \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes \mathbf{C} + \mathbf{B} \otimes \mathbf{C} \quad (22.3-7a)
\]

\[
\mathbf{A} \otimes (\mathbf{B} + \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} + \mathbf{A} \otimes \mathbf{C} \quad (22.3-7b)
\]

Associativity:

\[
\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) = (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} \quad (22.3-8)
\]

The matrix product (indicated by a dot) of Kronecker products can be rewritten as

\[
(\mathbf{A} \otimes \mathbf{B}) \cdot (\mathbf{C} \otimes \mathbf{D}) = (\mathbf{A} \otimes \mathbf{C}) \otimes (\mathbf{B} \otimes \mathbf{D}) \quad (22.3-9a)
\]

\[
(\mathbf{L}_1 \otimes \mathbf{R}_1) \cdot (\mathbf{L}_2 \otimes \mathbf{R}_2) \cdot \ldots \cdot (\mathbf{L}_n \otimes \mathbf{R}_n) = (\mathbf{L}_1 \cdot \mathbf{L}_2 \cdot \ldots \cdot \mathbf{L}_n) \otimes (\mathbf{R}_1 \cdot \mathbf{R}_2 \cdot \ldots \cdot \mathbf{R}_n) \quad (22.3-9b)
\]

Set \( \mathbf{L}_1 = \mathbf{L}_2 = \ldots = \mathbf{L}_n = \mathbf{L} \) and \( \mathbf{R}_1 = \mathbf{R}_2 = \ldots = \mathbf{R}_n = \mathbf{R} \) in the latter relation to obtain

\[
(\mathbf{L} \otimes \mathbf{R})^n = \mathbf{L}^n \otimes \mathbf{R}^n \quad (22.3-9c)
\]

The Kronecker product of matrix products can be rewritten as

\[
(\mathbf{A} \cdot \mathbf{B}) \otimes (\mathbf{C} \cdot \mathbf{D}) = (\mathbf{A} \otimes \mathbf{C}) \cdot (\mathbf{B} \otimes \mathbf{D}) \quad (22.3-10a)
\]

\[
(\mathbf{L}_1 \cdot \mathbf{R}_1) \otimes (\mathbf{L}_2 \cdot \mathbf{R}_2) \otimes \ldots \otimes (\mathbf{L}_n \cdot \mathbf{R}_n) = (\mathbf{L}_1 \otimes \mathbf{L}_2 \otimes \ldots \otimes \mathbf{L}_n) \cdot (\mathbf{R}_1 \otimes \mathbf{R}_2 \otimes \ldots \otimes \mathbf{R}_n) \quad (22.3-10b)
\]

Here the matrices left and right from a dot must be compatible for ordinary matrix multiplication.

One has

\[
(\mathbf{A} \otimes \mathbf{B})^T = \mathbf{A}^T \otimes \mathbf{B}^T \quad (22.3-11a)
\]

\[
(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \quad (22.3-11b)
\]

If \( \mathbf{A} \) and \( \mathbf{B} \) are respectively \( m \times n \) and \( r \times s \) matrices then

\[
\mathbf{A} \otimes \mathbf{B} = (\mathbf{I}_m \otimes \mathbf{B}) \cdot (\mathbf{A} \otimes \mathbf{I}_n) \quad (22.3-12a)
\]

\[
= (\mathbf{A} \otimes \mathbf{I}_r) \cdot (\mathbf{I}_n \otimes \mathbf{B}) \quad (22.3-12b)
\]

where \( \mathbf{I}_n \) is the \( n \times n \) identity matrix. If \( \mathbf{A} \) is \( n \times n \) and \( \mathbf{B} \) is \( t \times t \) then

\[
\det (\mathbf{A} \otimes \mathbf{B}) = \det(\mathbf{A})^t \det(\mathbf{B})^n \quad (22.3-13)
\]

Back to the Walsh transform, we have \( \mathbf{W}_1 = [1] \) and for \( n = 2^k, n > 1 \):

\[
\mathbf{W}_n = \begin{bmatrix} +\mathbf{W}_{n/2} & +\mathbf{W}_{n/2} \\ +\mathbf{W}_{n/2} & -\mathbf{W}_{n/2} \end{bmatrix} = \mathbf{W}_2 \otimes \mathbf{W}_{n/2} \quad (22.3-14)
\]

In order to see that this relation is the statement of a fast algorithm split the (to be transformed) vector \( x \) into halves

\[
x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \quad (22.3-15)
\]
22.4: A variant of the Walsh transform *

and write out the matrix-vector product

\[
W_n x = \begin{bmatrix}
W_{n/2} x_0 + W_{n/2} x_1 \\
W_{n/2} x_0 - W_{n/2} x_1
\end{bmatrix} = \begin{bmatrix}
W_{n/2} (x_0 + x_1) \\
W_{n/2} (x_0 - x_1)
\end{bmatrix}
\]  

(22.3-16)

That is, a length-\(n\) transform can be computed by two length-\(n/2\) transforms of the sum and difference of the first and second half of \(x\).

We define a notation equivalent to the product sign,

\[
\bigotimes_{k=1}^n M_k := M_1 \otimes M_2 \otimes M_3 \otimes \ldots \otimes M_n
\]  

(22.3-17)

where the empty product equals a 1 \(\times\) 1 matrix with entry 1. When \(A = B\) in relation \(22.3-11b\) we have \((A \otimes A)^{-1} = A^{-1} \otimes A^{-1}\), \((A \otimes A \otimes A)^{-1} = A^{-1} \otimes A^{-1} \otimes A^{-1}\) and so on. That is,

\[
\left( \bigotimes_{k=1}^n A \right)^{-1} = \bigotimes_{k=1}^n A^{-1}
\]  

(22.3-18)

For the Walsh transform we have

\[
W_n = \bigotimes_{k=1}^{\log_2(n)} W_2
\]  

(22.3-19)

and

\[
W_n^{-1} = \bigotimes_{k=1}^{\log_2(n)} W_2^{-1}
\]  

(22.3-20)

The latter relation isn’t that exciting as \(W_2^{-1} = W_2\) for the Walsh transform. However, it also holds when the inverse transform is different from the forward transform. Thereby, given a fast algorithm for some transform in form of a Kronecker product, the fast algorithm for the backward transform is immediate.

The **direct sum** of two matrices is defined as

\[
A \oplus B := \begin{bmatrix}
A & 0 \\
0 & B
\end{bmatrix}
\]  

(22.3-21)

In general \(A \oplus B \neq B \oplus A\). As an analogue to the sum sign we have

\[
\bigoplus_{k=1}^n A := I_n \otimes A
\]  

(22.3-22)

where \(I_n\) is the \(n \times n\) identity matrix. The matrix \(I_n \otimes A\) consists of \(n\) copies of \(A\) that lie on the diagonal. The Kronecker product can be used to derive properties of unitary transforms, see \[226\].

22.4 A variant of the Walsh transform *

All operations necessary for the Walsh transform are cheap: loads, stores, additions and subtractions. The memory access pattern is a major concern with direct mapped cache, as we have verified comparing the first two implementations in this chapter. Even the one found to be superior due to its more localized access is guaranteed to have a performance problem as soon as the array is long enough: all accesses are separated by a power-of-two distance and cache misses will occur beyond a certain limit. Rather bizarre attempts like inserting ‘pad data’ have been reported in order to mitigate the problem. The Gray code permutation described in section 2.8 on page 103 suggests an interesting solution where the sub-arrays are always accessed in mutually reversed order [FXT: walsh/walshgray.h].
Chapter 22: The Walsh transform and its relatives

1 template<typename Type>
2 void walsh_gray(Type *f, ulong ldn)
3 // Gray variant of the Walsh transform.
4 // Radix-2 decimation in frequency (DIF) algorithm
5 {
6     const ulong n = (1UL<<ldn);
7     for (ulong ldm=ldn; ldm>0; --ldm) // dif
8     {
9         const ulong m = (1UL<<ldm);
10        for (ulong r=0; r<n; r+=m)
11        {
12            ulong t1 = r;
13            ulong t2 = r + m - 1;
14            for ( ; t1<t2; ++t1,--t2)
15            {
16                Type u = f[t1];
17                Type v = f[t2];
18                f[t1] = u + v;
19                f[t2] = u - v;
20            }
21        }
22     }
23 }
24 }

The transform is not self-inverse, however, the inverse transform can be implemented easily be reversing the steps:

1 template<typename Type>
2 void inverse_walsh_gray(Type *f, ulong ldn)
3 // Inverse of walsh_gray().
4 // Radix-2 decimation in time (DIT) algorithm.
5 {
6     const ulong n = (1UL<<ldn);
7     for (ulong ldm=1; ldm<=ldn; ++ldm) // dit
8     {
9         const ulong m = (1UL<<ldm);
10        for (ulong r=0; r<n; r+=m)
11        {
12            ulong t1 = r;
13            ulong t2 = r + m - 1;
14            for ( ; t1<t2; ++t1,--t2)
15            {
16                Type u = f[t1];
17                Type v = f[t2];
18                f[t1] = u + v;
19                f[t2] = u - v;
20            }
21        }
22     }
23 }
24 }

Using $Q$ for the grs_negate() routine (described below), $W_k$ for walsh_wak(), $W_g$ for walsh_gray() and $G$ for gray_permute() then

\[ W_k = Q W_g G^{-1} = G W_g^{-1} Q \]  \hspace{1cm} (22.4-1)

That is, either of the following two sequences of statements

1 { inverse_gray_permute(f, n); walsh_gray(f, ldn); grs_negate(f, n); }
2 { grs_negate(f, n); inverse_walsh_gray(f, ldn); gray_permute(f, n); }

is equivalent to the call walsh_wak(f, ldn). The function \texttt{FXT:grs_negate()} in aux1/grsnegate.h changes signs for certain elements:

1 template<typename Type>
2 void grs_negate(Type *f, ulong n)
3 // Negate elements at indices where the Golay-Rudin-Shapiro is negative.
4 {
5     for (ulong k=0; k<n; ++k)
6     {
7         if ( grs_negative_q(k) ) f[k] = -f[k];
8     }

[fxtbook draft of 2008-August-17]
The function \texttt{grs\_negative\_q()} is described in section 1.16.5 on page 46. It turns out that the Gray-variant only wins on machines where the memory clock speed is significantly lower than the CPU. While a call to \texttt{walsh\_gray()} alone is never slower than the \texttt{walsh\_wak()} routine the additional steps often cause too much overhead.

22.5 Higher radix Walsh transforms

A generator for short-length Walsh (wak) transforms is given as \texttt{fft/gen-walsh-demo.cc}. It can create code for DIF and DIT transforms. For example, the code for the 4-point DIF transforms is

```cpp
template <typename Type>
inline void
short_walsh_wak_dif_4(Type *f)
{
    Type t0, t1, t2, t3;
    t0 = f[0];
    t1 = f[1];
    t2 = f[2];
    t3 = f[3];
    sumdiff( t0, t2 );
    sumdiff( t1, t3 );
    sumdiff( t0, t1 );
    sumdiff( t2, t3 );
    f[0] = t0;
    f[1] = t1;
    f[2] = t2;
    f[3] = t3;
}
```

To make the code more readable we use function \texttt{sumdiff()} in \texttt{aux0/sumdiff.h}:

```cpp
template <typename Type>
static inline void sumdiff(Type &a, Type &b)
{// \{a, b\} \leftarrow \{a+b, a-b\}
    Type t=a-b; a+=b; b=t;
}
```

We further need a variant that transforms elements which are not contiguous but lie apart by a distance \(s\):

```cpp
template <typename Type>
inline void
short_walsh_wak_dif_4(Type *f, ulong s)
{
    Type t0, t1, t2, t3;
    
    ulong x = 0;
    t0 = f[x]; x += s;
    t1 = f[x]; x += s;
    t2 = f[x]; x += s;
    t3 = f[x];
    sumdiff( t0, t2 );
    sumdiff( t1, t3 );
    sumdiff( t0, t1 );
    sumdiff( t2, t3 );
    
    ulong x = 0;
    f[x] = t0; x += s;
    f[x] = t1; x += s;
    f[x] = t2; x += s;
    f[x] = t3;
}
```

The short DIF transforms are given in \texttt{walsh/shortwalshwakdif.h}, DIT variants in \texttt{walsh/shortwalshwakdit.h}. A radix-4 DIF transform using these ingredients is \texttt{walsh/walshwak4.h}:
# Chapter 22: The Walsh transform and its relatives

```c
void walsh_wak_dif4(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-4 decimation in frequency (DIF) algorithm.
// Self-inverse.
{
    const ulong n = (1UL<<ldn);
    if ( n<=2 )
        {
            if ( n==2 ) short_walsh_wak_dif_2(f);
            return;
        }
    for (ulong ldm=ldn; ldm>3; ldm-=2)
        {
            ulong m = (1UL<<ldm);
            ulong m4 = (m>>2);
            for (ulong r=0; r<n; r+=m)
                {
                    for (ulong j=0; j<m4; j++) short_walsh_wak_dif_4(f+j+r, m4);
                }
        }
    if ( ldn & 1 ) // n is not a power of 4, need a radix-8 step
        {
            for (ulong i0=0; i0<n; i0+=8) short_walsh_wak_dif_8(f+i0);
        }
    else
        {
            for (ulong i0=0; i0<n; i0+=4) short_walsh_wak_dif_4(f+i0);
        }
}
```

With the implementation radix-8 DIF transform some care must be taken to choose the correct final step size [FXT: walsh/walshwak8.h]:

```c
template <typename Type>
void walsh_wak_dif8(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-8 decimation in frequency (DIF) algorithm.
// Self-inverse.
{
    const ulong n = (1UL<<ldn);
    if ( n<=4 )
        {
            switch (n )
            {
            case 4: short_walsh_wak_dif_4(f); break;
            case 2: short_walsh_wak_dif_2(f); break;
            }
            return;
        }
    const ulong xx = 4;
    ulong ldm;
    for (ldm=ldn; ldm>xx; ldm-=3)
        {
            ulong m = (1UL<<ldm);
            ulong m8 = (m>>3);
            for (ulong r=0; r<n; r+=m)
                {
                    for (ulong j=0; j<m8; j++) short_walsh_wak_dif_8(f+j+r, m8);
                }
        }
    switch ( ldm )
    {
    case 4:
        for (ulong i0=0; i0<n; i0+=16) short_walsh_wak_dif_16(f+i0);
        break;
    case 3:
        for (ulong i0=0; i0<n; i0+=8) short_walsh_wak_dif_8(f+i0);
        break;
    case 2:
        for (ulong i0=0; i0<n; i0+=4) short_walsh_wak_dif_4(f+i0);
        break;
    }
```

[fxtbook draft of 2008-August-17]
22.5: Higher radix Walsh transforms

Performance

For the performance comparison we include a matrix variant of the Walsh transform [FXT: walsh/walshwakmatrix.h]:

```cpp
template <typename Type>
void walsh_wak_matrix(Type *f, ulong ldn)
{
  ulong ldc = (ldn>>1);
  ulong ldr = ldn-ldc; // ldr>=ldc
  ulong nc = (1UL<<ldc);
  ulong nr = (1UL<<ldr); // nrow >= ncol
  for (ulong r=0; r<nr; ++r) walsh_wak_dif4(f+r*nc, ldc);
  transpose2(f, nr, nc);
  for (ulong c=0; c<nc; ++c) walsh_wak_dif4(f+c*nr, ldr);
  transpose2(f, nc, nr);
}
```

The transposition routine is given in [FXT: aux2/transpose2.h]. We only use even powers of two so the transposition is that of a square matrix.

As for dyadic convolutions we do not need the data in a particular order so we also include a version of the matrix algorithm that omits the final transposition:

```cpp
template <typename Type>
void walsh_wak_matrix_1(Type *f, ulong ldn, int is)
{
  ulong ldc = (ldn>>1);
  ulong ldr = ldn-ldc; // ldr>=ldc
  if ( is<0 ) swap2(ldr, ldc); // inverse
  ulong nc = (1UL<<ldc);
  ulong nr = (1UL<<ldr); // nrow >= ncol
  for (ulong r=0; r<nr; ++r) walsh_wak_dif4(f+r*nc, ldc);
  transpose2(f, nr, nc);
  for (ulong c=0; c<nc; ++c) walsh_wak_dif4(f+c*nr, ldr);
}
```

The following calls give (up to normalization) the mutually inverse transforms:

```cpp
walsh_wak_matrix_1(f, ldn, +1);
walsh_wak_matrix_1(f, ldn, -1);
```

We do not consider the range of transform lengths \(n < 128\) where unrolled routines and the radix-4 algorithm consistently win. Figure 22.5-A shows a comparison of the routines given so far. There are clearly two regions to distinguish: firstly, the region were the transforms fit into the first-level data cache (which is 64 kilobyte, corresponding to \(ldn = 13\)). Secondly, the region where \(ldn > 13\) and the performance becomes more and more memory bound.

In the first region the radix-4 routine is the fastest. The radix-8 routine comes close but, somewhat surprisingly, never wins.

In the second region the matrix version is the best. However, for very large sizes its performance could be better. Note that with odd \(ldn\) (not shown) its performance drops significantly due to the more expensive transposition operation. The transposition is clearly the bottleneck. One can use (machine specific) optimizations for the transposition to further improve the performance.

In the next section we give an algorithm that avoids the transposition completely and consistently outperforms the matrix algorithm.
Figure 22.5-A: Relative speed of different implementations of the Walsh (wak) transform. The transforms were run \( \text{rep} \) times for each measurement, the quantity \( \text{dt} \) gives the elapsed time for \( \text{rep} \) transforms of the given type. The quantity ‘MB/s’ gives the memory transfer rate as if a radix-2 algorithm was used, it equals ‘Memsize’ times ‘ldn’ divided by the time elapsed for a single transform. The ‘rel’ gives the performance relative to the radix-2 version, smaller values mean better performance.
22.6 Localized Walsh transforms

A decimation in time (DIT) algorithm combines the two halves of the array, then the halves of the two halves, the halves of each quarter, and so on. With each step the whole array is accessed which leads to the performance drop as soon as the array does not fit into the cache.

22.6.1 The method of localization

One can reorganize the algorithm as follows: combine the two halves of the array and postpone further processing of the upper half, then combine the halves of the lowers half and again postpone processing of its upper half. Repeat until size two is reached. Then use the algorithm at the postponed parts, starting with the smallest (last postponed).

The scheme can be sketched for size 16, as follows:

```
    //hhhhhhhhhhhhhhhh
    //hhhhhhhh44444444
    //hhhh333344444444
    //hh22333344444444
```

The letters ‘h’ denote places processed before any recursive call. The blocks of twos, threes and fours denote postponed blocks. The Walsh transform is thereby decomposed into a sequence of Haar transforms (see figure 23.6-A on page 504). The algorithm described is most easily implemented via recursion:

```
template <typename Type>
void walsh_wak_loc_dit2(Type *f, ulong ldn)
{
    if ( ldn<1 ) return;
    // Recursion:
    for (ulong ldm=1; ldm<ldn; ++ldm) walsh_wak_loc_dit2(f+(1UL<<ldm), ldm);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh; t1<mh; ++t1, ++t2) sumdiff(f[t1], f[t2]);
    }
}
```

22.6.2 Optimizing the routine

Optimizations are obtained by avoiding recursions for small sizes. We use a radix-4 algorithm as soon as the transform size is smaller or equal to the cache size and we avoid recursion for tiny transforms [FXT: walsh/walshwakloc2.h]:

```
template <typename Type>
void walsh_wak_loc_dit2(Type *f, ulong ldn)
{
    if ( ldn<=13 ) // parameter: (2**13)*sizeof(Type) <= L1-cache
    {
        walsh_wak_dif4(f,ldn); // note: DIF version, result is the same
        return;
    }
    // Recursion:
    short_walsh_wak_dit_2(f+2); // ldm==1
    short_walsh_wak_dit_4(f+4); // ldm==2
    short_walsh_wak_dit_8(f+8); // ldm==3
    short_walsh_wak_dit_16(f+16); // ldm==4
    for (ulong ldm=5; ldm<ldn; ++ldm) walsh_wak_loc_dit2(f+(1UL<<ldm), ldm);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        //...
    }
}
```
Chapter 22: The Walsh transform and its relatives

A decimation in frequency (DIF) version is obtained by reversion of the steps:

```cpp
template <typename Type>
void walsh_wak_loc_dif2(Type *f, ulong ldn)
{
  if ( ldn<=13 ) // parameter: (2**13)*sizeof(Type) <= L1-cache
    { walsh_wak_dif4(f,ldn);
      return;
    }
  for (ulong ldm=ldn; ldm>=1; --ldm)
    { const ulong m = (1UL<<ldm);
      const ulong mh = (m>>1);
      for (ulong t1=0, t2=mh; t1<mh; ++t1, ++t2)
        { Type u = f[t1];
          Type v = f[t2];
          f[t1] = u + v;
          f[t2] = u - v;
        }
      // Recursion:
      short_walsh_wak_dif_2(f+2); // ldm==1
      short_walsh_wak_dif_4(f+4); // ldm==2
      short_walsh_wak_dif_8(f+8); // ldm==3
      short_walsh_wak_dif_16(f+16); // ldm==4
      for (ulong ldm=5; ldm<ldn; ++ldm) walsh_wak_loc_dif2(f+(1UL<<ldm), ldm);
    }
}
```

The double loop in the algorithm is a reversed Haar transform, see chapter 23 on page 493. The double loop in the DIF algorithm is a transposed reversed Haar transform. The (generated) short-length transforms are given in the files [FXT: walsh/shortwalshwakdif.h] and [FXT: walsh/shortwalshwakdit.h]. For example, the length-8, decimation in frequency routine is

```cpp
template <typename Type>
inline void short_walsh_wak_dif_8(Type *f)
{
  Type t0, t1, t2, t3, t4, t5, t6, t7;
  t0 = f[0]; t1 = f[1]; t2 = f[2]; t3 = f[3];
  t4 = f[4]; t5 = f[5]; t6 = f[6]; t7 = f[7];
  sumdiff( t0, t4 ); sumdiff( t1, t5 ); sumdiff( t2, t6 ); sumdiff( t3, t7 );
  sumdiff( t0, t2 ); sumdiff( t1, t3 ); sumdiff( t4, t6 ); sumdiff( t5, t7 );
  sumdiff( t0, t1 ); sumdiff( t2, t3 ); sumdiff( t4, t5 ); sumdiff( t6, t7 );
  f[0] = t0; f[1] = t1; f[2] = t2; f[3] = t3;
}
```

The used strategy leads to a very favorable memory access pattern that results in excellent performance for large transforms. Figure 22.6-A shows a comparison between the localized transforms and the matrix algorithm. Small sizes are omitted because the localized algorithm has the very same speed as the radix-4 algorithm it falls back to. The localized algorithms are the clear winners, even against the matrix algorithm with only one transposition. For very large transforms the decimation in frequency version is very slightly faster. This is due to the fact that it starts with smaller chunks of data so more of the data is in cache when the larger sub-arrays are accessed.

The localized algorithm can easily be implemented for transforms where a radix-2 step is known. Section 24.9 on page 525 gives the fast Hartley transform variant of the localized algorithm.

One can develop similar routines with higher radix. However, a radix-4 version was found to be slower than the given routines. A certain speedup can be achieved by unrolling and prefetching. We use the C-type `double` whose size is 8 bytes. Substitute the double loop in the DIF version (that is, the Haar transform) by
### 22.6: Localized Walsh transforms

<table>
<thead>
<tr>
<th>ldn</th>
<th>MemSize (kB)</th>
<th>MemSize (doubles)</th>
<th>rep</th>
<th>walsh_wak_matrix(f,ldn); dt=</th>
<th>MB/s=</th>
<th>rel=</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.672327 MB/s= 5674 rel= 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>512</td>
<td>65536</td>
<td>1</td>
<td>0.556581 MB/s= 6683 rel= 0.826756</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>256</td>
<td>1</td>
<td>0.533746 MB/s= 7148 rel= 0.793878</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>1024</td>
<td>1</td>
<td>0.919579 MB/s= 4150 rel= 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>32</td>
<td>4096</td>
<td>1</td>
<td>0.938006 MB/s= 4068 rel= 0.422225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.927804 MB/s= 4113 rel= 0.419611 *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ldn</th>
<th>MemSize (kB)</th>
<th>MemSize (doubles)</th>
<th>rep</th>
<th>walsh_wak_matrix_1(f,ldn,+1); dt=</th>
<th>MB/s=</th>
<th>rel=</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.498558 MB/s= 7652 rel= 0.745141 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>512</td>
<td>65536</td>
<td>1</td>
<td>0.692488 MB/s= 5511 rel= 0.753049</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>256</td>
<td>1</td>
<td>0.653256 MB/s= 5942 rel= 0.710386 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>1024</td>
<td>1</td>
<td>0.938006 MB/s= 4068 rel= 0.422225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>32</td>
<td>4096</td>
<td>1</td>
<td>0.927804 MB/s= 4113 rel= 0.419611 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.927804 MB/s= 4113 rel= 0.419611 *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ldn</th>
<th>MemSize (kB)</th>
<th>MemSize (doubles)</th>
<th>rep</th>
<th>walsh_wak_loc_dit2(f,ldn); dt=</th>
<th>MB/s=</th>
<th>rel=</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.533746 MB/s= 7148 rel= 0.793878</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>512</td>
<td>65536</td>
<td>1</td>
<td>0.692488 MB/s= 5511 rel= 0.753049</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>256</td>
<td>1</td>
<td>0.653256 MB/s= 5942 rel= 0.710386 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>1024</td>
<td>1</td>
<td>0.938006 MB/s= 4068 rel= 0.422225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>32</td>
<td>4096</td>
<td>1</td>
<td>0.927804 MB/s= 4113 rel= 0.419611 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.927804 MB/s= 4113 rel= 0.419611 *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ldn</th>
<th>MemSize (kB)</th>
<th>MemSize (doubles)</th>
<th>rep</th>
<th>walsh_wak_loc_dif2(f,ldn); dt=</th>
<th>MB/s=</th>
<th>rel=</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.533746 MB/s= 7148 rel= 0.793878</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>512</td>
<td>65536</td>
<td>1</td>
<td>0.692488 MB/s= 5511 rel= 0.753049</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>256</td>
<td>1</td>
<td>0.653256 MB/s= 5942 rel= 0.710386 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>1024</td>
<td>1</td>
<td>0.938006 MB/s= 4068 rel= 0.422225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>32</td>
<td>4096</td>
<td>1</td>
<td>0.927804 MB/s= 4113 rel= 0.419611 *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>128</td>
<td>16384</td>
<td>1</td>
<td>0.927804 MB/s= 4113 rel= 0.419611 *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### 2.26.2: Speed comparison between localized and matrix algorithms for the Walsh transform.

```c
// machine specific prefetch instruction:
#define PREF(p,o) asm volatile ("prefetchw " #o " (%0) " : : "r" (p) )

ulong ldm;
for (ldm=ldn; ldm>=6; --ldm)
{
    const ulong m = (1UL<<ldm);
    const ulong mh = (m>>1);
    PREF(f, 0); PREF(f+mh, 0);
    PREF(f, 64); PREF(f+mh, 64);
    PREF(f, 128); PREF(f+mh, 128);
    PREF(f, 192); PREF(f+mh, 192);
    for (ulong t1=0, t2=mh; t1<mh; t1+=8, t2+=8)
    {
        double *p1 = f + t1, *p2 = f + t2;
        PREF(p1, 256); PREF(p2, 256);
        double u0 = f[t1+0], v0 = f[t2+0];
        double u1 = f[t1+1], v1 = f[t2+1];
        double u2 = f[t1+2], v2 = f[t2+2];
        double u3 = f[t1+3], v3 = f[t2+3];
        sumdiff(u0, v0); f[t1+0] = u0; f[t2+0] = v0;
        sumdiff(u1, v1); f[t1+1] = u1; f[t2+1] = v1;
        sumdiff(u2, v2); f[t1+2] = u2; f[t2+2] = v2;
        sumdiff(u3, v3); f[t1+3] = u3; f[t2+3] = v3;
        double u4 = f[t1+4], v4 = f[t2+4];
        double u5 = f[t1+5], v5 = f[t2+5];
        double u6 = f[t1+6], v6 = f[t2+6];
        double u7 = f[t1+7], v7 = f[t2+7];
        sumdiff(u4, v4); f[t1+4] = u4; f[t2+4] = v4;
        sumdiff(u5, v5); f[t1+5] = u5; f[t2+5] = v5;
        sumdiff(u6, v6); f[t1+6] = u6; f[t2+6] = v6;
        sumdiff(u7, v7); f[t1+7] = u7; f[t2+7] = v7;
    }
}
for ( ; ldm>=1; --ldm)
```

[ftxbook draft of 2008-August-17]
{  const ulong m = (1UL<<ldm);
  const ulong mh = (m>>1);
  for (ulong t1=0, t2=mh; t1<mh; ++t1, ++t2) sumdiff(f[t1], f[t2]);
}

The following list gives the speed ratio between the optimized and the unoptimized DIF routine:

<table>
<thead>
<tr>
<th>Size</th>
<th>MemSize</th>
<th>Ratio</th>
</tr>
</thead>
</table>
| 14 d
| 128 kB | 1.24252|
| 16 d
| 512 kB | 1.43568|
| 18 d
| 2 MB   | 1.23875|
| 20 d
| 8 MB   | 1.21012|
| 22 d
| 32 MB  | 1.19939|
| 24 d
| 128 MB | 1.18245|

For sizes that are out of (level-2) cache most of the speedup is due to the memory prefetch.

### 22.6.3 Iterative versions of the algorithms

![Figure 22.6-B: Binary values of the start index and length of the Haar transforms in the iterative version of the localized DIF (left) and DIT (right) transform. Dots are used for zeros.](fxtbook draft of 2008-August-17)

In the DIF algorithm the Haar transforms are executed at positions $f+2, f+4, f+6, \ldots$ and the length of the transform at position $f+s$ is determined by the lowest set bit in $s$. Additionally, a full-length Haar transform has to be done at the beginning. As C++ code:

```c++
template <typename Type>
inline void haar_dif2(Type *f, ulong n)
{
  for (ulong m=n; m>=2; m>>=1)
    {
      const ulong mh = (m>>1);
      for (ulong t1=0, t2=mh; t1<mh; ++t1, ++t2) sumdiff(f[t1], f[t2]);
    }
}

void loc_dif2(Type *f, ulong n)
{
  haar_dif2(f, n);
  for (ulong z=2; z<n; z+=2) haar_dif2(f+z, (z&-z));
}
```

Note that the routines now take the length of the transform as second argument, not its base-2 logarithm.

With the DIT algorithm matters are slightly more complicated. A pattern can be observed by printing the binary expansions of the starting position and length of the transforms shown in figure 22.6-B (created with [FXT: fft/locrec-demo.cc]). The lengths are again determined by the lowest bit of the start position. And we have also seen the pattern in the left column: the reversed binary words in reversed (subset-) lexicographic order, see figure 1.27-A on page 73. The implementation is quite concise:

```c++
template <typename Type>
inline void haar_dit2(Type *f, ulong n)
{
  for (ulong z=2; z<n; z+=2) haar_dif2(f+z, (z&-z));
}
```

Note that the routines now take the length of the transform as second argument, not its base-2 logarithm.
22.7: Dyadic (XOR) convolution

Dyadic convolution has XOR where the usual one has plus.

The dyadic convolution of the sequences $a$ and $b$ is the sequence $h$ defined by

$$ h_\tau := \sum_{i \oplus j = \tau} a_i b_j $$ \hspace{1cm} (22.7-1a)

$$ = \sum_i a_i b_{i \oplus \tau} $$ \hspace{1cm} (22.7-1b)

where the symbol ‘$\oplus$’ stands for bit-wise XOR operator. All three sequence must be of the same length that is a power of 2. The dyadic convolution could rightfully be called *XOR-convolution*.

The semi-symbolic scheme of the convolution is shown in figure 22.7-A. The table is equivalent to the one (for cyclic convolution) given in figure 21.1-A on page 438. The dyadic convolution can be used for the multiplication of hypercomplex numbers as shown in section 37.14 on page 814.

A fast algorithm for the computation of the dyadic convolution uses the Walsh transform [FXT: dyadic_convolution() in walsh/dyadiccnvl.h]:

```cpp
template <typename Type>
void dyadic_convolution(Type * restrict f, Type * restrict g, ulong ldn)
// Dyadic convolution (XOR-convolution): h[] of f[] and g[]:
// h[k] = sum( i XOR j = k, f[i]*g[j] )
// Result is written to g[].
// ldn := base-2 logarithm of the array length
{
    walsh_wak(f, ldn);
    walsh_wak(g, ldn);
    const ulong n = (1UL<<ldn);
    for (ulong k=0; k<n; ++k) g[k] *= f[k];
    walsh_wak(g, ldn);
}
```
Figure 22.7-A: Semi-symbolic scheme for the dyadic convolution of two length-16 sequences.

Figure 22.7-B: Semi-symbolic scheme for the dyadic equivalent of the negacyclic convolution. Negative contributions to a bucket have a minus appended.

An scheme similar to that of the negacyclic convolution is shown in figure 22.7-B. It can be obtained via

```c
walsh_wal_dif2_core(f, ldn);
walsh_wal_dif2_core(g, ldn);
ulong n = (1UL<<ldn);
for (ulong i=0; i<n; i++) fht_mul(f[i], f[i], g[i], g[i], 0.5);
walsh_wal_dit2_core(g, ldn);
```

where `fht_mul()` is the operation used for the convolution with fast Hartley transforms [FXT: convolution/fhtmul32.h].

```c
template <typename Type>
static inline void
fht_mul(Type xi, Type xj, Type &yi, Type &yj, double v)
// yi <-- v*( 2*xi*xj + xi*xi - xj*xj )
// yj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type h1p = xi, h1m = xj;
    Type s1 = h1p + h1m, d1 = h1p - h1m;
    Type h2p = yi, h2m = yj;
    yi = (h2p * s1 + h2m * d1) * v;
yj = (h2m * s1 - h2p * d1) * v;
}
```

[fxtbook draft of 2008-August-17]
22.8 The Walsh transform: Walsh-Paley basis

The basis functions are shown in figure 22.8-A. Actually one can also apply the revbin permutation before the transform. That is,

$$W_p = W_k R = RW_k$$  \hspace{1cm} (22.8-1)$$

One has for $W_p$

$$W_p = GW_p G = G^{-1} W_p G^{-1} \hspace{1cm} (22.8-2)$$

$$= ZW_p Z = Z^{-1} W_p Z^{-1} \hspace{1cm} (22.8-3)$$

where $Z$ denotes the zip permutation (see section 2.5 on page 99) and $G$ denotes the Gray permutation (see section 2.8 on page 103).
A function that computes the $k$-th base function of the transform is [FXT: `walsh_pal_basefunc()` in `walsh/walshbasefunc.h`]:

```cpp
1 template <typename Type>
2 void walsh_pal_basefunc(Type *f, ulong n, ulong k)
3 {
4     k = revbin(k, ld(n));
5     for (ulong i=0; i<n; ++i)
6     {
7         ulong x = i & k;
8         x = parity(x);
9         f[i] = (0==x ? +1 : -1);
10    }
11 }
```

### 22.9 Sequency ordered Walsh transforms

![Basis functions for the sequency-ordered Walsh transform (Walsh-Kacmarz basis). Asterisks denote the value +1, blank entries denote −1.](image)

The term corresponding to the frequency of the Fourier basis functions is the so-called *sequency* of the Walsh functions, the number of the changes of sign of the individual functions. Note that the sequency of a signal with frequency $f$ usually is $2f$.

If the basis functions shall be ordered by their sequency one can use

```cpp
1 const ulong n = (1UL<<ldn);
2 walsh_wak(f, ldn);
3 revbin_permute(f, n);
4 inverse_gray_permute(f, n);
```
That is
\[
W_w = G^{-1} R W_k = W_k R G
\] (22.9-1)

A function that computes the \( k \)-th base function of the transform is [FXT: walsh/walsh_basefunc()] in [walsh/walshbasefunc.h]:

```cpp
template<typename Type>
void walsh_wal_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n)+1);
    k = gray_code(k);
    // k = revbin(k, ld(n));
    // k = rev_gray_code(k);
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

A version of the transform that avoids the Gray permutation is based on [FXT: walsh/walshwal.h]:

```cpp
template<typename Type>
void walsh_wal_dif2_core(Type *f, ulong ldn)
// Core routine for sequency ordered Walsh transform.
// Radix-2 decimation in frequency (DIF) algorithm.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=2; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r+j;
            ulong t2 = t1+mh;
            double u = f[t1];
            double v = f[t2];
            f[t1] = u + v;
            f[t2] = v - u; // reversed
        }
    }
    if ( ldn )
    {
        // ulong ldm=1;
        const ulong m = 2; //(1UL<<ldm);
        const ulong mh = 1; //(m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r+j;
            ulong t2 = t1+mh;
            double u = f[t1];
            double v = f[t2];
            f[t1] = u + v;
            f[t2] = v - u; // reversed
        }
    }
}
```

[fxtbook draft of 2008-August-17]
double u = f[t1];
double v = f[t2];
f[t1] = u + v;
f[t2] = u - v;
}
}

The transform still needs the revbin permutation:

```cpp
template <typename Type>
inline void walsh_wal(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dif2_core(f, ldn);
    // =^=
    // walsh_wal_dit2_core(f, ldn);
    // revbin_permute(f, n);
}
```

A decimation in time (DIT) version of the core-routine is also given in [FXT: walsh/walshwal.h]. The procedure gray_permute() is given in section 2.8 on page 103.

The sequence of statements

\[
\text{walsh_gray}(f, \text{ldn}); \; \text{grs_negate}(f, \text{n}); \; \text{revbin_permute}(f, \text{n});
\]

is equivalent to \text{walsh_wal}(f, \text{ldn}) and might be faster for large arrays. We have

\[
W_w = RQ W_g = W_g^{-1} R Q
\]  

### 22.9.1 Even/odd ordering of sequencies

An alternative ordering of the base functions (first even sequencies ascending then odd sequencies descending [FXT: walsh/walshwak.h] in [walsh/walshwalrev.h]) can be obtained by either of the following sequences of statements (with \( n=1UL<<\text{ldn} \)):

```cpp
{ revbin_permute(f, n); gray_permute(f, n); walsh_wak(f, ldn); }
{ walsh_wak(f, ldn); inverse_gray_permute(f, n); revbin_permute(f, n); }
{ zip_rev(f, n); walsh_wal(f, ldn); }
{ walsh_wal(f, ldn); unzip_rev(f, n); }
{ walsh_wak(f, ldn); inverse_gray_permute(f, n); revbin_permute(f, n); }
{ revbin_permute(f, n); walsh_gray(f, ldn); grs_negate(f, n); }
```

That is,

\[
\overline{W}_w = W_k G R = R G^{-1} W_k
\]

\[
= W_w Z = Z^{-1} W_w
\]

\[
= Q W_g R
\]  

However, an implementation that is more efficient uses the core-routines that have the Gray permutation ‘absorbed’ [FXT: walsh_wal_rev() in [walsh/walshwalrev.h]]:

```cpp
template <typename Type>
inline void walsh_wal_rev(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dit2_core(f, ldn);
    // =^=
    // walsh_wal_dif2_core(f, ldn);
    // revbin_permute(f, n);
}
This implementation uses the fact that

\[ \tilde{W}_w = R W_w R \]  

(22.9-6)

The following sequence of statements computes the same transform:

```c
{ revbin_permute(f, n); walsh_gray(f, ldn); grs_negate(f, n); }
```

Similar relations as for the transform with Walsh-Paley basis (22.8-2 and 22.8-3 on page 475) hold for \( W_w \):

\[
W_w = G W_w G = G^{-1} W_w G^{-1} \]  

(22.9-7)

\[
= Z W_w Z = Z^{-1} W_w Z^{-1} \]  

(22.9-8)

The \( k \)-th base function of the transform can be computed as [FXT: `walsh_wal_rev_basefunc()` in `walsh/walshbasefunc.h`]

```c
1 template <typename Type>
2 void walsh_wal_rev_basefunc(Type *f, ulong n, ulong k)
3 {
4   k = revbin(k, ld(n));
5   k = gray_code(k);
6   // k = rev_gray_code(k);
7   // k = revbin(k, ld(n));
8   for (ulong i=0; i<n; ++i)
9     { 
10       ulong x = i & k;
11       x = parity(x);
12       f[i] = ( 0==x ? +1 : -1 );
13     }
14 }
```
Figure 22.9-C: Basis functions for a self-inverse Walsh transform that has sequencies $n/2$ and $n/2 - 1$ only. Asterisks denote the value $+1$, blank entries denote $-1$.

### 22.9.2 Transforms with sequencies $n/2$ or $n/2 - 1$

The next variant of the Walsh transform has the interesting feature that the basis functions for a length-$n$ transform have only sequencies $n/2$ and $n/2 - 1$ at the even and odd indices, respectively. The transform is self-inverse (the basis is shown in figure 22.9-C) and can be obtained via [FXT: walsh_q1() in walsh/walshq.h](#).

```cpp
template <typename Type>
void walsh_q1(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    grs_negate(f, n);
    walsh_gray(f, ldn);
    revbin_permute(f, n);
}
```

A different transform with sequency $n/2$ for the first half of the basis, sequency $n/2 - 1$ for the second half ([FXT: walsh_q2() in walsh/walshq.h](#)), basis shown in figure 22.9-D, is computed by

```cpp
template <typename Type>
void walsh_q2(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    revbin_permute(f, n);
    grs_negate(f, n);
    walsh_gray(f, ldn);
}
```

[fxtbook draft of 2008-August-17]
22.9: Sequency ordered Walsh transforms

Figure 22.9-D: Basis functions for a self-inverse Walsh transform (second form) that has sequencies \( n/2 \) and \( n/2 - 1 \) only. Asterisks denote the value +1, blank entries denote −1.

One has:

\[
W_{q2} = RW_{q1} R
\]  

(22.9-9)

The base functions of the transforms can be computed as [FXT: walsh/walshbasefunc.h]

```c
1  template <typename Type>
2  void walsh_q1_basefunc(Type *f, ulong n, ulong k)
3  {
4      ulong qk = (grs_negative_q(k) ? 1 : 0);
5      k = gray_code(k);
6      k = revbin(k, ld(n));
7      for (ulong i=0; i<n; ++i)
8      {
9          ulong x = i & k;
10         x = parity(x);
11         ulong qi = (grs_negative_q(i) ? 1 : 0);
12         x ^= (qk ^ qi);
13         f[i] = ( 0==x ? +1 : -1 );
14      }
15  }
```

and

```c
1  template <typename Type>
2  void walsh_q2_basefunc(Type *f, ulong n, ulong k)
3  {
4      ulong qk = (grs_negative_q(k) ? 1 : 0);
5      k = revbin(k, ld(n));
```
\begin{verbatim}
6     k = gray_code(k);
7     for (ulong i=0; i<n; ++i)
8         { 
9             ulong x = i & k;
10             x = parity(x);
11             ulong qi = (grs_negative_q(i) ? 1 : 0);
12             x ^= (qk ^ qi);
13             f[i] = ( 0==x ? +1 : -1 );
14         }

The function \texttt{grs_negative_q()} is described in section \ref{sec:grs} on page \pageref{sec:grs}.
\end{verbatim}

\section{Slant transform}

The \textit{slant transform} can be implemented using a Walsh Transform and just a little pre/post-processing \cite{fxt:walsh/slant.cc}:

\begin{verbatim}
1 void slant(double *f, ulong ldn)
2 {
3     walsh_wak(f, ldn);
4     ulong n = 1UL<<ldn;
5     for (ulong ldm=0; ldm<ldn-1; ++ldm)
6         {
7             ulong m = 1UL<<ldm;  // m = 1, 2, 4, 8, ..., n/4
8             double N = m*2, N2 = N*N;
9             double a = sqrt(3.0*N2/(4.0*N2-1.0));
10            double b = sqrt(1.0-a*a);  // == sqrt((N2-1)/(4*N2-1));
11            for (ulong j=m; j<n-1; j+=4*m)
12                { 
13                    ulong t1 = j;
14                    ulong t2 = j + m;
15                    double f1 = f[t1], f2 = f[t2];
16                    f[t1] = b * f2 + a * f1;
17                    f[t2] = a * f2 - b * f1;
18                }
19         }
20 }

Apart from the Walsh transform only an amount of work linear with the array size has to be done: the inner loop accesses the elements in strides of 4, 8, 16, ..., \(2^{n-1}\).

The inverse transform is:

\begin{verbatim}
1 void inverse_slant(double *f, ulong ldn)
2 {
3     ulong n = 1UL<<ldn;
4     ulong ldm=ldn-2;
5     do
6         {
7             ulong m = 1UL<<ldm;  // m = n/4, n/2, ..., 4, 2, 1
8             double N = m*2, N2 = N*N;
9             double a = sqrt(3.0*N2/(4.0*N2-1.0));
10            double b = sqrt(1.0-a*a);  // == sqrt((N2-1)/(4*N2-1));
11            for (ulong jm=j; jm<n-1; jm+=4*m)
12                { 
13                    ulong t1 = jm;
14                    ulong t2 = t1 + m;
15                    double f1 = f[t1], f2 = f[t2];
16                    f[t1] = a * f2 + b * f1;
17                    f[t2] = b * f2 - a * f1;
18                }
19         }
20     while ( ldm-- );
21     walsh_wak(f, ldn);
22 }
\end{verbatim}

A sequency ordered version of the transform can be implemented as follows:

\begin{verbatim}
1 void slant_seq(double *f, ulong ldn)
\end{verbatim}
22.11 Arithmetic transform

This implementation can be optimized by combining the involved permutations, see [269].

The inverse is obtained by calling the inverse operations in reversed order:

```c
void inverse_slant_seq(double *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    zip_rev(f, n);
    gray_permute(f, n);
    inverse_slant(f, ldn);
}
```

22.11 Arithmetic transform

How to make a arithmetic transform out of a Walsh transform:

'Forward: replace \((u+v)\) by \((u)\), and \((u-v)\) by \((v-u)\).
Backward: replace \((u+v)\) by \((u)\), and \((u-v)\) by \((u+v)\).' 

```
0: [+ - + - + - + - - + - - + - + + - + + - + - + - + - - + - ]
1: [ + - - + + - + - - + + - - + - + + - + + - + - + - - + - ]
2: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
3: [ + - - + + - + - - + + - - + - + + - + + - + - + - - + - ]
4: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
5: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
6: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
7: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
8: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
9: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
10: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
11: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
12: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
13: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
14: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
15: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
16: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
17: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
18: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
19: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
20: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
21: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
22: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
23: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
24: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
25: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
26: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
27: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
28: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
29: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
30: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
31: [ + - - + - + + - - + - - + - + + - + + - + - + - - + - ]
```

Figure 22.11-A: Basis functions for the Arithmetic transform \((A^-)\), the one with the minus sign). The values are ±1, blank entries denote 0.
Chapter 22: The Walsh transform and its relatives

Figure 22.11-B: Basis functions for the inverse Arithmetic transform ($A^+$, the one without minus sign). The values are $+1$, blank entries denote $0$.

On to the code [FXT: walsh/arithtransform.h]:

```cpp
template <typename Type>
void arith_transform_plus(Type *f, ulong ldn)
// Arithmetic Transform (positive sign).
// Radix-2 decimation In Frequency (DIF) algorithm.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0; j<mh; ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u;
                f[t2] = u + v;
            }
        }
    }
}
```

and

```cpp
template <typename Type>
void arith_transform_minus(Type *f, ulong ldn)
// Arithmetic Transform (negative sign).
```
22.11: Arithmetic transform

// Radix-2 decimation In Frequency (DIF) algorithm.
// Inverse of arith_transform_plus().
{
    f[t1] = u;
    f[t2] = v - u;
}

The length-2 transforms can be written as

\[
A^+_2 v = \begin{bmatrix} +1 & 0 \\ +1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \quad (22.11-1) \\
A^-_2 v = \begin{bmatrix} +1 & 0 \\ -1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b-a \end{bmatrix} \quad (22.11-2)
\]

That the transform with the minus is called the forward transform is tradition. Similar to the Fourier transform we avoid the forward- backward- naming scheme and put:

```c
template<typename Type>
inline void arith_transform(Type *f, ulong ldn, int is)
{
    if ( is>0 ) arith_transform_plus(f, ldn);
    else arith_transform_minus(f, ldn);
}
```

In Kronecker product notation (see section 22.3 on page 461) the arithmetic transform and its inverse can be written as

\[
A^+_2 = \bigotimes_{k=1}^{\log_2(n)} A^+_2 \\
A^-_2 = \bigotimes_{k=1}^{\log_2(n)} A^-_2 
\]

The \( k \)-th element of the arithmetic \( A^+ \) transform is

\[
(A^+)[a]_k = \sum_{i \subset k} a_i \quad (22.11-4a)
\]

where \( i \subset k \) means that the bits of \( i \) are a subset of the bits of \( k \): \( i \subset k \iff (i \land k) = i \). For the transform \( A^- \) we have

\[
(A^-)[a]_k = (-1)^{p(k)} \sum_{i \subset k} (-1)^{p(i)} a_i \quad (22.11-4b)
\]

where \( p(x) \) is the parity of \( x \).

22.11.1 Transposed arithmetic transform

We define a transposed arithmetic transforms \( B^+ \) and \( B^- \) via

\[
B^+_2 = \begin{bmatrix} +1 & +1 \\ 0 & +1 \end{bmatrix} \quad B^+_n = \bigotimes_{k=1}^{\log_2(n)} B^+_2 \\
B^-_2 = \begin{bmatrix} +1 & -1 \\ 0 & +1 \end{bmatrix} \quad B^-_n = \bigotimes_{k=1}^{\log_2(n)} B^-_2
\]

Then the transforms are [FXT: walsh/arithtransform.h]
22.11.2 Relation to Walsh transform

To establish the relation to the Walsh transform recall that its decomposition as a Kronecker product is

\[
W_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \quad W_n = \bigotimes_{k=1}^{\log_2(n)} W_2
\] (22.11-6)

Now as \((W_2 A_2^+) A_2^- = W_2\) the expression in parenthesis is the matrix that transforms the 2-point arithmetic transform (with the negative sign) to the Walsh transform. Similarly, as \((\frac{1}{2} A_2^- W_2) W_2 = A_2^+\), the matrix leading to the conversion from Walsh to arithmetic transform can be determined. One finds:

\[
\begin{align*}
(W_2 A_2^+) A_2^- &= \begin{bmatrix} +2 & +1 \\ 0 & -1 \end{bmatrix} A_2^- \\
(W_2 A_2^-) A_2^+ &= \begin{bmatrix} 0 & +1 \\ +2 & -1 \end{bmatrix} A_2^+ \\
\left(\frac{1}{2} A_2^- W_2\right) W_2 &= A_2^- = \frac{1}{2} \begin{bmatrix} +1 & +1 \\ 0 & -2 \end{bmatrix} W_2 \\
\left(\frac{1}{2} A_2^+ W_2\right) W_2 &= A_2^+ = \frac{1}{2} \begin{bmatrix} +1 & +1 \\ +2 & 0 \end{bmatrix} W_2
\end{align*}
\] (22.11-7a-d)

The Kronecker product of the given matrices gives the converting transform. For example, using relation (22.11-7a) define

\[
T_n := \bigotimes_{k=1}^{\log_2(n)} \begin{bmatrix} +2 & +1 \\ 0 & -1 \end{bmatrix}
\] (22.11-8)

Then \(T_n\) converts a arithmetic (minus sign) transform to a Walsh transform: \(W_n = T_n A_n^-\). For the relations between the arithmetic transform, the Reed-Muller transform and the Walsh transform, see [257].

22.12 Reed-Muller transform

How to make a Reed-Muller transform out of the arithmetic transform:
‘Replace + and - by XOR, done.’

The Reed-Muller transform is can be obtained from the arithmetic transform by working modulo two. The transform is self-inverse, its basis functions are identical to those of the arithmetic transform \(A_x^+\), shown in figure 22.11-B on page 484. The implementation is almost identical to [FXT: walsh_wak_dif2()] in walsh/walshwak2.h. The only changes are
22.12: Reed-Muller transform

Walsh: \( f[t1] = u + v; \) \( \implies \) Reed-Muller: \( f[t1] = u; \)
Walsh: \( f[t2] = u - v; \) \( \implies \) Reed-Muller: \( f[t2] = u \triangleleft v; \)

There we go \([\text{FXT: walsh/reedmuller.h}]\):

```cpp
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

```cpp
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

The decimation in time algorithm can be obtained from \([\text{FXT: walsh_wak_dit2()} in \text{walsh/walshwak2.h}]\) by the very same changes. As given, the transforms work word-wise, if the bit-wise transform is wanted use

\[
\begin{align*}
R R R &= \text{id} & R^{-1} &= R R = E \\
E E E &= \text{id} & E^{-1} &= E E = R \\
R E &= = E R = \text{id}
\end{align*}
\]

In fact, all relations given in the referenced section hold.

As can be seen from the ‘atomic’ matrices (relations \([1.19-12c \ldots 1.19-12f on page 57]) the four transforms corresponding to the ‘color codes’ are obtained by

Walsh: \( f[t1] = u + v; \) \( \implies \) transposed Reed-Muller: \( f[t1] = u \triangleleft v; \)
B: \( f[t1] = u \triangleleft v; \) \( \implies \) transposed Reed-Muller: \( f[t1] = v; \)
Y: \( f[t1] = u; \) \( \implies \) (Reed-Muller transform)
R: \( f[t1] = v; \) \( \implies \) (Reed-Muller transform)
E: \( f[t1] = u \triangleleft v; \) \( \implies \) transposed Reed-Muller: \( f[t2] = u; \)
```

The basis functions of the transforms are shown in figure \(22.12-A\).

The transposed Reed-Muller transform can be obtained by setting

Walsh: \( f[t1] = u + v; \) \( \implies \) transposed Reed-Muller: \( f[t1] = u \triangleleft v; \)
Walsh: \( f[t2] = u - v; \) \( \implies \) transposed Reed-Muller: \( f[t2] = v; \)
Chapter 22: The Walsh transform and its relatives

The symbolic powering idea from section 1.19 on page 51 leads to transforms with bases (using eight element arrays and the yellow code):

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\(x=0\) \(x=1\) \(x=2\) \(x=3\) \(x=4\) \(x=5\) \(x=6\) \(x=7\)

The program [FXT: bits/bitxtransforms-demo.cc] gives the matrices for 64-bit words.

A function that computes the \(k\)-th base function of the transform is [FXT: walsh/reedmuller.h]:

```cpp
1 template<typename Type>
2 inline void reed_muller_basefunc(Type *f, ulong n, ulong k)
3 {
4     for (ulong i=0; i<n; ++i)
5         f[i] = ( (i & k)==k ? +1 : 0 ); // is k a bit-subset of i ?
6 }
7 }
```

Functions that are the word-wise equivalents of the Gray code are given in [FXT: aux1/wordgray.h]:

```cpp
1 template<typename Type>
2 void word_gray(Type *f, ulong n)
3 {
4     for (ulong k=0; k<n-1; ++k) f[k] ^= f[k+1];
5 }
```

and

```cpp
1 void inverse_word_gray(Type *f, ulong n)
2 {
3     ulong x = 0, k = n;
4     while ( k-- ) { x ^= f[k]; f[k] = x; }
5 }
```

As one might suspect, these are related to the Reed-Muller transform. Writing \(Y\) ("yellow") for the Reed-Muller transform, \(g\) for the word-wise Gray code and \(S_k\) for the cyclic shift by \(k\) words (word zero is moved to position \(k\)) one has

\[
\begin{align*}
YS_{+1}Y &= g \\
YS_{-1}Y &= g^{-1} \\
YS_kY &= g^k
\end{align*}
\]

(22.12-4)

(22.12-5)

(22.12-6)

These are exactly the relations 1.19-10a ... 1.19-10c on page 56 for the bit-wise transforms. For \(k \geq 0\) the operator \(S_k\) corresponds to the shift toward element zero (use [FXT: rotate_sign() in perm/rotate.h]).

The power of the word-wise Gray code is perfectly equivalent to the bit-wise version:
22.13: The OR-convolution, and the AND-convolution

Let \( e \) be the reversed Gray code operator, then we have for the transposed Reed-Muller transform \( B \):

\[
BS_{+1}B = e^{-1} \tag{22.12-7}
\]
\[
BS_{-1}B = e \tag{22.12-8}
\]
\[
BS_{k}B = e^{-k} \tag{22.12-9}
\]

Further,

\[
ES_{k}R = e^{k} \tag{22.12-10}
\]
\[
Ee^{k}R = S_{k} \tag{22.12-11}
\]

The transforms as Kronecker products (all operations are modulo two):

\[
B_{n} = \bigotimes_{k=1}^{\log_{2}(n)} B_{2} \text{ where } B_{2} = \begin{bmatrix}
1 & 1 \\
0 & 1 \\
\end{bmatrix} \tag{22.12-12a}
\]
\[
Y_{n} = \bigotimes_{k=1}^{\log_{2}(n)} Y_{2} \text{ where } Y_{2} = \begin{bmatrix}
1 & 0 \\
1 & 1 \\
\end{bmatrix} \tag{22.12-12b}
\]
\[
R_{n} = \bigotimes_{k=1}^{\log_{2}(n)} R_{2} \text{ where } R_{2} = \begin{bmatrix}
0 & 1 \\
1 & 1 \\
\end{bmatrix} \tag{22.12-12c}
\]
\[
E_{n} = \bigotimes_{k=1}^{\log_{2}(n)} E_{2} \text{ where } E_{2} = \begin{bmatrix}
1 & 1 \\
1 & 0 \\
\end{bmatrix} \tag{22.12-12d}
\]

22.13 The OR-convolution, and the AND-convolution

Let \( a \) and \( b \) be sequences of length a power of two. We define the OR-convolution \( h \) of \( a \) and \( b \), as

\[
h_{\tau} = \sum_{i \lor j = \tau} a_{i} b_{j} \tag{22.13-1}
\]

where \( \lor \) denotes bit-wise OR. The symbolic table for the OR-convolution is shown in figure 22.13-A (see figure 21.1-A on page 438 for an explanation of the scheme). The OR-convolution can be computed as

\[
h = A^{-} \left[ A^{+}[a] \cdot A^{+}[b] \right] \tag{22.13-2}
\]

where \( A^{+} \) and \( A^{-} \) are the arithmetic transforms given in section 22.11 on page 483. An implementation is [FXT: walsh/or-convolution.h]:

```
template<typename Type>
void word_gray_pow(Type *f, ulong n, ulong x)
{
    for (ulong s=1; s<n; s*=2)
    {
        if ( x & 1)
            // word_gray ** s:
            for (ulong k=0, j=k+s; j<n; ++k,++j) f[k] ^= f[j];
        x >>= 1;
    }
}
```
Chapter 22: The Walsh transform and its relatives

\[ h[k] = \sum(i \mid j = k, f[i]*g[j]) \]
\[ f[] \text{ and } g[] \text{ must not overlap.} \]
\[ \text{// Result written to } g[]. \]
\[ \{ \]
\[ \text{arith_transform_plus}(f, ldn); \]
\[ \text{arith_transform_plus}(g, ldn); \]
\[ \text{const ulong } n = (1UL<<ldn); \]
\[ \text{for (ulong } k=0; k<n; ++k) g[k] *= f[k]; \]
\[ \text{arith_transform_minus}(g, ldn); \]
\[ \} \]

We also have
\[ h = Y^{-1}[Y[a] \cdot Y[b]] = Y[Y[a] \cdot Y[b]] \] (22.13-3)

where \( Y \) is the Reed-Muller transform given in section 22.12 on page 486.

Define the AND-convolution \( h \) of two sequences \( a \) and \( b \) as
\[ h_\tau = \sum_{i \land j = \tau} a_i b_j \] (22.13-4)

where \( \land \) denotes the bit-wise AND. The symbolic scheme is shown in figure 22.13-B.

The AND-convolution can be computed as
\[ h = B^-\left[B^+[a] \cdot B^+[b]\right] \] (22.13-5)

Figure 22.13-A: Semi-symbolic scheme for the OR-convolution of two length-16 sequences.

Figure 22.13-B: Semi-symbolic scheme for the AND-convolution of two length-16 sequences.
where $B^+$ and $B^-$ are the transposed arithmetic transforms. The implementation of the AND-convolution is [FXT: walsh/and-convolution.h]:

```cpp
template <typename Type>
inline void and_convolution(Type * restrict f, Type * restrict g, ulong ldn)
// Compute the AND-convolution $h[k]$ of $f[]$ and $g[]$:
// $h[k] = \sum(i \& j == k, f[i]*g[j])$
// $f[]$ and $g[]$ must not overlap.
// Result written to $g[]$.
{
  transposed_arith_transform_plus(f, ldn);
  transposed_arith_transform_plus(g, ldn);
  const ulong n = (1UL<<ldn);
  for (ulong k=0; k<n; ++k) g[k] *= f[k];
  transposed_arith_transform_minus(g, ldn);
}
```

The same is true modulo two:

$$h = B^{-1}[B[a] \cdot B[b]] = B[B[a] \cdot B[b]]$$

(22.13-6)

here $B$ is the transposed Reed-Muller transform.
Chapter 23

The Haar transform

Haar transforms are invertible transforms that do not involve trigonometric factors. We present several variants of the transform whose computation involves just proportional $n$ operations. Haar transforms can be used as building blocks of the Walsh transform. We give two non-standard splitting schemes for Haar transforms, based on the Fibonacci and Mersenne numbers.

23.1 The ‘standard’ Haar transform

<table>
<thead>
<tr>
<th>Index</th>
<th>Basis Function</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +]</td>
<td>1/sqrt(32)</td>
</tr>
<tr>
<td>1</td>
<td>[+ + + + + + + + + + + + + + + + - - - - - - - - - - - - - - - -]</td>
<td>1/sqrt(32)</td>
</tr>
<tr>
<td>2</td>
<td>[+ + + + + + - - - - - - - - - -]</td>
<td>1/sqrt(16)</td>
</tr>
<tr>
<td>3</td>
<td>[+ + + + - - - - -]</td>
<td>1/sqrt(8)</td>
</tr>
<tr>
<td>4</td>
<td>[+ + + + - - - - -]</td>
<td>1/sqrt(8)</td>
</tr>
<tr>
<td>5</td>
<td>[+ + + + - - - - -]</td>
<td>1/sqrt(8)</td>
</tr>
<tr>
<td>6</td>
<td>[+ + + + - - - - -]</td>
<td>1/sqrt(8)</td>
</tr>
<tr>
<td>7</td>
<td>[+ + + + - - - - -]</td>
<td>1/sqrt(8)</td>
</tr>
<tr>
<td>8</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>9</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>10</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>11</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>12</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>13</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>14</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>15</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>16</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>17</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>18</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>19</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>20</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>21</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>22</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>23</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>24</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>25</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>26</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>27</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>28</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>29</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>30</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
<tr>
<td>31</td>
<td>[+ + - -]</td>
<td>1/sqrt(4)</td>
</tr>
</tbody>
</table>

Figure 23.1-A: Basis functions for the Haar transform. Only the signs of the nonzero entries are shown. The absolute value of the nonzero entries in each row is given at the right. The norm of each row is one.
Chapter 23: The Haar transform

The Haar transform of a length-n sequence \( f \) consists of \( \log_2(n) \) steps were the sums and differences of adjacent pairs of elements \( f_{2j}, f_{2j+1} \) are computed. The sums are then written to the lower half of the array \( f \), the differences to the higher half. Ignoring the order (and normalization), each step corresponds to a matrix multiplication:

\[
\begin{bmatrix}
+1 & +1 \\
+1 & -1 \\
+1 & +1 \\
+1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
f_0 \\
f_1 \\
f_2 \\
f_3 \\
\end{bmatrix}
\]

(23.1-1)

The step is applied to the full array, then to the lower half, the lower quarter, ..., the lower four elements, the lowest pair. (The array length \( n \) must be a power of two.) The computational cost of the transform is proportional \( n + n/2 + n/4 + \ldots + 4 + 2 \) which is \( \sim O(n) \). The basis functions for the Haar transform have finite support, they are shown in figure 23.1-A.

The following implementation involves \( 2n \) multiplications \( \sqrt{2} \) which make the transform orthogonal, corresponding to a scalar factor of \( \sqrt{2} \) in the relation 23.1-A:

```c
1 template <typename Type>
2 void haar(Type *f, ulong ldn)
3 {
4     ulong n = (1UL<<ldn);
5     const Type s2 = sqrt(0.5); // normalization factor
6     Type *g = new Type[n]; // scratch space
7     for (ulong m=n; m>1; m>>=1) // n, n/2, n/4, n/8, ..., 4, 2
8     {
9         ulong mh = (m>>1);
10        for (ulong j=0, k=0; j<m; j+=2, k++) // sums and differences of adjacent pairs
11           {
12               Type x = f[j];
13               Type y = f[j+1];
14               g[k] = x + y; // sums to lower half
15               g[mh+k] = (x - y) * s2; // differences to higher half
16           }
17         copy(g, f, m);
18     }
19     delete [] g;
20 }
```

We can reduce the number of multiplications to \( n \) by delaying the multiplies with the sums [FXT: haar() in haar/haar.h]:

```c
1 template <typename Type>
2 void haar(Type *f, ulong ldn, Type *ws=0)
3 {
4     ulong n = (1UL<<ldn);
5     const Type s2 = sqrt(0.5);
6     Type v = 1.0;
7     Type *g = ws;
8     if ( !ws ) g = new Type[n];
9     for (ulong m=n; m>1; m>>=1)
10        {
11            v *= s2;
12            ulong mh = (m>>1);
13            for (ulong j=0, k=0; j<m; j+=2, k++)
14                {
15                    Type x = f[j];
16                    Type y = f[j+1];
17                    g[k] = x + y;
18                    g[mh+k] = (x - y) * v;
19                }
20        }
21     delete [] g;
22 }
```
23.2: In-place Haar transform

The 'standard' Haar transform routines are not in-place, they use a temporary storage. A rather simple reordering of the basis functions, however, allows for to an in-place algorithm [FXT: haar_inplace() in haar/haar.h]:

\[
\begin{align*}
\text{template } & \text{<typename Type>}
\text{void haar_inplace(Type } *f, \text{ ulong } ldn) \\
\text{\{}
\text{\quad ulong } & n = (1UL<<ldn); \\
\text{\quad Type } & s2 = \sqrt{0.5}; \\
\text{\quad Type } & v = 1.0; \\
\text{\quad for (ulong } js=2; \text{ js<=n; js<<=1)} \\
\text{\quad \quad \{}
\text{\quad \quad \quad Type } x = f[k]; \\
\text{\quad \quad \quad Type } y = f[\text{mh+k}] \times v; \\
\text{\quad \quad \quad g[j] } & = x + y; \\
\text{\quad \quad \quad g[j+1] } & = x - y; \\
\text{\quad \quad \}} \\
\text{\quad v } & = s2; \\
\text{\quad f[0] } & = v; // v==1.0/sqrt(n); \\
\text{\}}
\end{align*}
\]

The inverse is [FXT: inverse_haar_inplace() in haar/haar.h]:

\[
\begin{align*}
\text{template } & \text{<typename Type>}
\text{void inverse_haar_inplace(Type } *f, \text{ ulong } ldn) \\
\text{\{}
\text{\quad ulong } & n = (1UL<<ldn); \\
\text{\quad Type } s2 = \sqrt{0.5}; \\
\text{\quad Type } v = 1.0; \\
\text{\quad for (ulong } js=2; \text{ js<=n; js<<=1)} \\
\text{\quad \quad \{}
\text{\quad \quad \quad Type } x = f[k]; \\
\text{\quad \quad \quad Type } y = f[\text{mh+k}] \times v; \\
\text{\quad \quad \quad f[k] } & = x + y; \\
\text{\quad \quad \quad f[t] } & = (x - y) \times v; \\
\text{\quad \quad \}} \\
\text{\quad f[0] } & = v; // v==1.0/sqrt(n); \\
\text{\}}
\end{align*}
\]
The in-place Haar transform $H_i$ is related to the ‘usual’ Haar transform $H$ by a permutation $P_H$ via the relations

$$H = P_H \cdot H_i \quad (23.2-1)$$

$$H^{-1} = H_i^{-1} \cdot P_H^{-1} \quad (23.2-2)$$

The permutation $P_H$ can be programmed as

```cpp
template <typename Type>
void haar_permute(Type *f, ulong n)
{
    Type s2 = sqrt(2.0);
    Type v = 1.0/sqrt(n);
    f[0] *= v;
    for (ulong js=n; js>=2; js>>=1)
        {  
            for (ulong j=0, t=js>>1; j<n; j+=js, t+=js)
                {  
                    Type x = f[j];
                    Type y = f[t] * v;
                    f[j] = x + y;
                    f[t] = x - y;
                }  
            v *= s2;
        }
}
```
The ordering is such that basis functions that are identical up to a shift appear consecutively.

The revbin permutations in the loop do not overlap, so the inverse Haar permutation is obtained by simply swapping the loop with the full-length revbin permutation [FXT: perm/haarpermute.h]:

```c++
template <typename Type>
void inverse_haar_permute(Type *f, ulong n)
{
  for (ulong m=4; m<=n/2; m*=2) revbin_permute(f+m, m);
  revbin_permute(f, n);
}
```

Then, as given above, `haar` is equivalent to

```
haar_inplace();haar_permute();
```

and `inverse_haar` is equivalent to

```
inverse_haar_permute();inverse_haar_inplace();
```

### 23.3 Non-normalized Haar transforms

Versions of the Haar transform without normalization are given in [FXT: haar/haarnn.h]. The basis functions are the same as for the normalized versions, only the absolute value of the nonzero entries are different.

```c++
template <typename Type>
void haar_nn(Type *f, ulong ldn, Type *ws=0)
```
Chapter 23: The Haar transform

The Haar transform is defined as

\[
\begin{align*}
\hat{x}_k &= \sum_{n=0}^{N-1} x_n \delta(n - k) \\
\hat{y}_k &= \sum_{n=0}^{N-1} x_n \delta(n - k) - \sum_{n=0}^{N-1} x_n \delta(n + k)
\end{align*}
\]

where \( \delta(n) \) is the Kronecker delta function. The inverse Haar transform is given by

\[
\begin{align*}
x_n &= \sum_{k=0}^{N-1} \hat{x}_k \delta(n - k) \\
x_n &= \sum_{k=0}^{N-1} \hat{y}_k \delta(n - k) + \sum_{k=0}^{N-1} \hat{y}_k \delta(n + k)
\end{align*}
\]

Implementations of the Haar transform in C++ are shown below.

```cpp
#include <vector>

using namespace std;

void haar_transform(vector<double>& x, int ldn) {
    if (ldn == 1) return x[0] = 1.0;
    haar_transform(x, ldn / 2);
    double s2 = 2.0;
    for (int j = 0, k = 0; j < ldn; j += 2, k++)
        x[k] = x[j] + x[j+1];
    x[k] = x[j] - x[j+1];
    s2 /= 2.0;
}

void inverse_haar_transform(vector<double>& x, int ldn) {
    if (ldn == 1) return x[0] = 0.5;
    inverse_haar_transform(x, ldn / 2);
    double s2 = 2.0;
    for (int j = 0, k = 0; j < ldn; j += 2, k++)
        x[k] = x[j] + x[j+1];
    x[k] = x[j] - x[j+1];
    s2 *= 2.0;
}
```

An unnormalized transform that works in-place is

```cpp
#include <vector>

void haar_inplace_transform(vector<double>& x, int ldn) {
    for (int js = 2; js <= ldn; js <<= 1)
        for (int j = 0, t = js >> 1; j < ldn; j += js, t += js)
            x[j] = x[j] + x[t];
}

void inverse_haar_inplace_transform(vector<double>& x, int ldn) {
    for (int js = ldn; js >= 2; js >>= 1)
        for (int j = 0, t = js >> 1; j < ldn; j += js, t += js)
            x[j] = x[j] + x[t];
}
```

The inverse routine is

```cpp
#include <vector>

void inverse_haar_inplace_transform(vector<double>& x, int ldn) {
    x[0] = 0.5;
    for (int js = 1; js <= ldn; js <<= 1)
        for (int j = 0, t = js >> 1; j < ldn; j += js, t += js)
            x[j] = x[j] + x[t];
}
```

The inverse is

```cpp
#include <vector>

void inverse_haar_transform(vector<double>& x, int ldn) {
    x[0] = 0.5;
    for (int js = 1; js <= ldn; js <<= 1)
        for (int j = 0, t = js >> 1; j < ldn; j += js, t += js)
            x[j] = x[j] + x[t];
}
```
The sequence of statements \{ haar_inplace_nn(); haar_permute(); \} is equivalent to \{ haar_nn(); \}. The sequence \{ inverse_haar_permute(); inverse_haar_inplace(); \} is equivalent to \{ inverse_haar(); \}.

23.4 Transposed Haar transforms

```
0: [ + + + + + + ]
1: [ + + + + + - ]
2: [ + + + + - + ]
3: [ + + + + - - ]
4: [ + + + - + + ]
5: [ + + + - + - ]
6: [ + + + - - + ]
7: [ + + + - - - ]
8: [ + + - + + + ]
9: [ + + - + + - ]
10: [ + + - + - + ]
11: [ + + - + - - ]
12: [ + + - - + + ]
13: [ + + - - + - ]
14: [ + + - - - + ]
15: [ + + - - - - ]
16: [ + - + + + + ]
17: [ + - + + + - ]
18: [ + - + + - + ]
19: [ + - + + - - ]
20: [ + - + - + + ]
21: [ + - + - + - ]
22: [ + - + - - + ]
23: [ + - + - - - ]
24: [ + - - + + + ]
25: [ + - - + + - ]
26: [ + - - + - + ]
27: [ + - - + - - ]
28: [ + - - - + + ]
29: [ + - - - + - ]
30: [ + - - - - + ]
31: [ + - - - - - ]
```

Figure 23.4-A: Basis functions for the transposed Haar transform. Only the signs of the basis functions are shown. At the blank entries the functions are zero.

Figure 23.4-A shows the basis functions of the transposed Haar transform. The shown routine are given in [FXT: haar/transposedhaarnn.h]. The following routine does an unnormalized Haar transform. The result is, up to normalization, the same as with inverse_haar(). The implementation uses a scratch array:

```
1  template <typename Type>
2  void transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
3  {
4      ulong n = (1UL<<ldn);
5      Type *g = ws;
```
The inverse transform is

\begin{verbatim}
template <typename Type>
void inverse_transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws ) g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0; j<m; j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k];
            g[j] = x + y;
            g[j+1] = x - y;
        }
        copy(g, f, m);
    }
    if ( !ws ) delete [] g;
}
\end{verbatim}

The next routine is equivalent to the sequence of statements \{ inverse_haar_permute(); transposed_haar_inplace_nn(); \}. Its advantage is that no scratch array is needed:

\begin{verbatim}
template <typename Type>
void transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1; j<n; j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j] = x + y;
            f[t] = x - y;
        }
    }
}
\end{verbatim}

The inverse transform is

\begin{verbatim}
template <typename Type>
void inverse_transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=2; js<=n; js<<=1)
    {
        for (ulong j=0, t=js>>1; j<n; j+=js, t+=js)
        {
            Type x = f[j] * 0.5;
            Type y = f[t] * 0.5;
            f[j] = x + y;
            f[t] = x - y;
        }
    }
}
\end{verbatim}
### 23.5 The reversed Haar transform

Let $H_{ni}$ denote the non-normalized in-place Haar transform ($\text{haar\_inplace\_nn}$), Let $H_{tni}$ denote the transposed non-normalized in-place Haar transform ($\text{transposed\_haar\_inplace\_nn}$), $R$ the revbin permutation, $\overline{H}$ the reversed Haar transform and $\overline{H}_t$ the transposed reversed Haar transform. Then

$$
\overline{H} = RH_{ni}R \quad (23.5-1a)
$$

$$
\overline{H}_t = RH_{tni}R \quad (23.5-1b)
$$

$$
\overline{H}^{-1} = RH_{ni}^{-1}R \quad (23.5-1c)
$$

$$
\overline{H}_t^{-1} = RH_{tni}^{-1}R \quad (23.5-1d)
$$

#### Code for the reversed Haar transform

```cpp
1 template <typename Type>
2 void haar_rev_nn(Type *f, ulong ldn)
3 {
4     // const ulong n = (1UL<<ldn);
5     for (ulong ldm=ldn; ldm>=1; --ldm)
6     {
7         const ulong m = (1UL<<ldm);
8         const ulong mh = (m>>1);
9         ulong r = 0;
10        // for (ulong r=0; r<n; r+=m) // almost walsh_wak_dif2()
11        {
12            ulong t1 = r;
13            ulong t2 = r + mh;
14            for (ulong j=0; j<mh; ++j, ++t1, ++t2)
15        }
```

**Figure 23.4-B:** Basis functions for the transposed in-place Haar transform. Only the signs of the basis functions are shown. At the blank entries the functions are zero.
Chapter 23: The Haar transform

Figure 23.5-A: Basis functions for the reversed Haar transform. Only the signs of the nonzero entries are shown.

```c
{  
    Type u = f[t1];
    Type v = f[t2];
    f[t1] = u + v;
    f[t2] = u - v;
}
```

Note that this is almost the radix-2 DIF implementation for the Walsh transform. The only change is that the line for (ulong r=0; r<n; r+=m) was replaced by ulong r = 0. The transform can also be computed via the following sequence of statements: { revbin_permute(); haar_inplace_nn(); revbin_permute(); }.

The inverse transform is obtained by the equivalent modification with the DIT implementation for the Walsh transform and normalization:

```c
  template <typename Type>
  void inverse_haar_rev_nn(Type *f, ulong ldn)
  {
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
      const ulong m = (1UL<<ldm);
      const ulong mh = (m>>1);
      ulong r = 0;
      // for (ulong r=0; r<n; r+=m) // almost walsh_wak_dit2()
      {
        ulong t1 = r;
        ulong t2 = r + mh;
        for (ulong j=0; j<mh; ++j, ++t1, ++t2)
        {
```

[fxtbook draft of 2008-August-17]
23.6 Relations between Walsh and Haar transforms

23.6.1 Walsh transforms from Haar transforms

A length-$n$ Walsh transform can be obtained from one length-$n$ Haar transform, one transform of length-$\frac{n}{2}$, two transforms of length-$\frac{n}{4}$, four transforms of length-$\frac{n}{8}$, . . . and $n$ transforms of length-2. Using the reversed Haar transform the implementation is most straightforward: A Walsh transform ($W_k$, the one with the Walsh Kronecker base) can be implemented as

The reversed transposed Haar transform is, up to normalization, the inverse of `haar_rev_nn()`. It is given in [FXT: haar/transposedhaarrevnn.h]:

```cpp
template <typename Type>
void transposed_haar_rev_nn(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
        // for (ulong r=0; r<n; r+=m) // almost walsh_wak_dit2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0; j<mh; ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The same result would be obtained by the following sequence of statements: `{ revbin_permute(); transposed_haar_inplace_nn(); revbin_permute(); }`. The inverse transform is

```cpp
template <typename Type>
void inverse_transposed_haar_rev_nn(Type *f, ulong ldn)
{
    // const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
        // for (ulong r=0; r<n; r+=m) // almost walsh_wak_dif2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0; j<mh; ++j, ++t1, ++t2)
            {
                Type u = f[t1] * 0.5;
                Type v = f[t2] * 0.5;
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```
Haar transforms:
\[
\begin{array}{cccc}
H(16) & H(8) & H(4) & H(2) \\
AAAAAAAaaaaaa & BBBbmmm & Ccc & Dd \\
AAAAaaa & BBbb & Cc \\
AAaa & Bb \\
Aa & \\
\end{array}
\]

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
AAAAAAAaaaaaa
AAAAaaaBBBbmmm
AAAAaaCCccBBbbCCcc
AAaDdCcDdCcDDd

Figure 23.6-A: Symbolic description of how to build a Walsh transform from Haar transforms.

Transposed Haar transforms:
\[
\begin{array}{cccc}
H(16) & H(8) & H(4) & H(2) \\
Aa & \\
AAaa & Bb \\
AAAAaaa & BBbb & Cc \\
AAAAAAAaaaaaa & BBBbmmm & Ccc & Dd \\
\end{array}
\]

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
AaDdCcDdCcDd
AAaCCccBBbbCCcc
AAAAaaaBBBbmmm
AAAAAAAaaaaaa

Figure 23.6-B: Symbolic description of how to build a Walsh transform from Haar transforms, transposed version.

```c
1 // algorithm WH1:
2 ulong n = 1UL<<ldn;
3 haar_rev_nn(f, ldn);
4 for (ulong ldk=ldn-1; ldk>0; --ldk)
5 {
6     ulong k = 1UL << ldk;
7     for (ulong j=k; j<n; j+=2*k) haar_rev_nn(f+j, ldk);
8 }
```

The idea, as a symbolic scheme, is shown in figure 23.6-A. The scheme obtained by reversing the order of the lines is shown in figure 23.6-B. It corresponds to the computation of \(W_k\) using the transposed version of the Haar transform:

```c
1 // algorithm WH1T:
2 ulong n = 1UL<<ldn;
3 inverse_transposed_haar_rev_nn(f, ldn);
4 for (ulong ldk=1; ldk<ldn; ++ldk)
5 {
6     ulong k = 1UL << ldk;
7     for (ulong j=k; j<n; j+=2*k) transposed_haar_rev_nn(f+j, ldk);
8 }
```

Two more methods are obtained by reversing the individual lines of the schemes seen so far, see figure 23.6-C. These correspond to the computation of the inverse Walsh transform \(W_k^{-1} = \frac{1}{n} W_k\) either as

```c
1 // algorithm WH2T:
2 ulong n = 1UL<<ldn;
3 inverse_transposed_haar_rev_nn(f, ldn);
4 for (ulong ldk=ldn-1; ldk>0; --ldk)
```

23.6: Relations between Walsh and Haar transforms

The (∼n log(n)) schemes given here are not a efficient method to compute the Haar transform (which is ∼n). Instead, they can be used to identify the type of Haar transform that is the building block of a given Walsh transform.

The non-normalized transposed reversed Haar transform can (up to normalization) be obtained via

```c
// algorithm HW1: transposed_haar_rev_nn(f, ldn); =^=
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_wak(f+k, ldk);
}
walsh_wak(f, ldn);
```

and its inverse as

```c
// algorithm HW1I: inverse_transposed_haar_rev_nn(f, ldn); =^=
walsh_wak(f, ldn);
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_wak(f+k, ldk);
}
```

The non-normalized transposed Haar transform can (again, up to normalization) be obtained via

```c
// algorithm HW2: transposed_haar_nn(f, ldn); =^=
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_wak(f+k, ldk);
}
```

Figure 23.6-C: Symbolic scheme of the four versions of the computation of the Walsh transform via Haar transforms.
Walsh transform:
\[
W(16) \\
AaDdCcDdBbDdCcDd \\
AAaaCCccBBbbCCcc \\
AAAAaaaBBBBbbb \\
AAAAAAAaaaaaaa
\]

Inverse (or transposed) Walsh transforms:
\[
BBBBbbbb CCcc Dd \\
BBbbCCcc CcDd \\
BbDdCcDd
\]


Figure 23.6-D: Symbolic description of how to build a Haar transform from Walsh transforms.

3
4     { ulong k = 1UL << ldk;
5     walsh_pal(f+k, ldk);
6     }
7     walsh_pal(f, ldn);

and its inverse as

1     // algorithm HW2I: inverse_transposed_haar_nn(f, ldn); ^==
2     walsh_pal(f, ldn); // ^= revbin_permute(f, n); walsh_wak(f, ldn);
3     for (ulong ldk=1; ldk<ldn; ++ldk)
4     {
5         ulong k = 1UL << ldk;
6         walsh_pal(f+k, ldk);
7     }

The symbolic scheme is given in figure 23.6-D.

23.7 Nonstandard splitting schemes *

All radix-2 transforms recursively split the length of the array into halves. The size of the transforms is limited to powers of two. In a recursive implementation we use the fact that \(2^k = 2^{k-1} + 2^{k-1}\). With \(N_k := 2^k\) we have \(N_0 = 1\), and \(N_k = N_{k-1} + N_{k-1}\). We use different recursive schemes to derive nonstandard variants of the Haar and Walsh transforms.

23.7.1 Fibonacci-Haar and Fibonacci-Walsh transform

One can use the Fibonacci numbers \(F_n = F_{n-1} + F_{n-2}\) (where \(F_0 = 0\) and \(F_1 = 1\)) to construct a Fibonacci-Haar transform as follows [FXT: fib-haar.h].
Figure 23.7-A: Basis functions for the non-normalized Fibonacci-Haar transform. Only the signs of the nonzero entries are shown. At the blank entries the functions are zero.

```c
inline void fibonacci_haar(double *a, ulong f0, ulong f1)
// In-place Fibonacci-Haar transform of a[0,...,f0-1].
// f0 must be a Fibonacci number, f1 the next smaller Fibonacci number.
{
    if ( f0 < 2 ) return;
    ulong f2 = f0 - f1;
    for (ulong j=0,k=f1; j<f2; ++j,++k)
        double u = a[j], v = a[k);
        a[j] = (u+v) * SQRT1_2;
        a[k] = (u-v) * SQRT1_2;
    fibonacci_haar(a, f1, f2);
}
```

A non-normalized version is obtained by omitting the multiplications with $\frac{1}{\sqrt{2}}$ ($=\text{SQRT1\_2}$). The basis functions for the non-normalized transform with length-21 ($=F_8$) are shown in figure 23.7-A (compare to figure 23.5-A on page 502). The second row corresponds to the rabbit sequence described in section 36.11 on page 750. Figure 23.7-A was created with the program [FXT: fft/fib-haar-demo.cc].

Figure 23.7-B: Basis functions for the non-normalized Fibonacci-Walsh transform.
A *Fibonacci-Walsh transform* can be obtained by adding one line in the recursive implementation of the Fibonacci-Haar transform [FXT: walsh/fib-walsh.h]:

```cpp
inline void fibonacci_walsh(double *a, ulong f0, ulong f1) {
    if ( f0 < 2 ) return;
    ulong f2 = f0 - f1;
    for (ulong j=0, k=f1; j<f2; ++j, ++k) {
        double u = a[j], v = a[k];
        a[j] = (u+v) * SQRT1_2;
        a[k] = (u-v) * SQRT1_2;
    }
    fibonacci_walsh(a, f1, f2);
    fibonacci_walsh(a+f1, f2, f1-f2);  // <--- omit line to obtain Haar transform
}
```

The basis functions for the length 21 transform are shown in figure 23.7-B which was created with the program [FXT: fft/fib-walsh-demo.cc].

The given routines can be optimized by inserting short-length transforms as recursion end.

One can obtain Haar-like and Walsh-like transforms for any linear recursive sequence that is increasing. A construction for recurrences $N_k = N_{k-1} + N_{k-1-p}$ is considered in [112].

### 23.7.2 Mersenne-Haar and Mersenne-Walsh transform

```
<table>
<thead>
<tr>
<th>Mersenne-Haar</th>
<th>Mersenne-Walsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: [+ + + + + + + +]</td>
<td>0: [+ + + + + + + +]</td>
</tr>
<tr>
<td>1: [+ + - + - - + -]</td>
<td>1: [+ + + + + + + +]</td>
</tr>
<tr>
<td>2: [+ + + + + + + +]</td>
<td>2: [+ - + - + - + -]</td>
</tr>
<tr>
<td>3: [+ + - + - - + -]</td>
<td>3: [ + + + + + + + ]</td>
</tr>
<tr>
<td>4: [+ + + + + + + +]</td>
<td>4: [ + - + - + - + -]</td>
</tr>
<tr>
<td>5: [+ + + + + + + +]</td>
<td>5: [ + - + - + - + -]</td>
</tr>
<tr>
<td>6: [+ + + + + + + +]</td>
<td>6: [ + - + - + - + -]</td>
</tr>
<tr>
<td>7: [+ + - + - - + -]</td>
<td>7: [ + - + - + - + -]</td>
</tr>
<tr>
<td>8: [+ + + + + + + +]</td>
<td>8: [ + - + - + - + -]</td>
</tr>
<tr>
<td>9: [+ + - + - - + -]</td>
<td>9: [ + - + - + - + -]</td>
</tr>
<tr>
<td>10: [ + + - - - + + -]</td>
<td>10: [ + - + - + - + -]</td>
</tr>
<tr>
<td>11: [ + + - - - + + -]</td>
<td>11: [ + - + - + - + -]</td>
</tr>
<tr>
<td>12: [ + + - - - + + -]</td>
<td>12: [ + - + - + - + -]</td>
</tr>
<tr>
<td>13: [ + + - - - + + -]</td>
<td>13: [ + - + - + - + -]</td>
</tr>
<tr>
<td>14: [ + + - - - + + -]</td>
<td>14: [ + - + - + - + -]</td>
</tr>
</tbody>
</table>
```

**Figure 23.7-C:** Basis functions for the non-normalized Mersenne-Haar transform (left), and Mersenne-Walsh transform (right). Only the signs of the nonzero entries are shown. At the blank entries the functions are zero.

For the Mersenne numbers $M_k = 2^k - 1$ we have the recursion $M_k = 2 \cdot M_{k-1} + 1$. This can be used to obtain a *Mersenne-Walsh transform* [FXT: walsh/mers-walsh.h]:

```cpp
inline void mersenne_walsh(double *a, ulong f0) {
    if ( f0 < 2 ) return;
    ulong f1 = f0 >> 1;  // next smaller Mersenne number
    for (ulong j=0, k=f1+1; j<f1; ++j, ++k) {
        double u = a[j], v = a[k];
        a[j] = (u+v) * SQRT1_2;
        a[k] = (u-v) * SQRT1_2;
    }
}
```
23.7: Nonstandard splitting schemes *

Figure 23.7-C (right) gives the basis functions for the non-normalized Mersenne-Walsh transform. The
Mersenne-Haar transform is obtained by deleting one line as indicated. The implementation is given
in [FXT: haar/mers-haar.h], the basis functions of the non-normalized version are shown at the left
of figure 23.7-C. The figure was created with the programs [FXT: fft/mers-walsh-demo.cc] and [FXT:
fft/mers-haar-demo.cc]. Note that both transforms leave the central element unchanged.
Chapter 24

The Hartley transform

The Hartley transform is a trigonometric transform whose practical importance comes from the fact that it maps real data to real data. While the fast algorithms for radix-2 can be found without great difficulty the higher radix algorithms are not obvious. Therefore it is appropriate to describe the Hartley transform in terms of the Fourier transform. A method for the conversion of FFT algorithms to fast Hartley transform (FHT) algorithms is given.

Routines for the conversion of Hartley transforms to and from Fourier transforms are described. Convolution routines based on the FHT are given for complex and real valued data. An efficient procedure for the computation of the negacyclic convolution is described.

24.1 Definition and symmetries

The discrete Hartley transform (HT) of a length-\(n\) sequence \(a\) is defined as

\[
c = \mathcal{H}[a]
\]

\[
c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \left( \cos \frac{2\pi k x}{n} + \sin \frac{2\pi k x}{n} \right)
\]

(24.1-1a)

(24.1-1b)

That is, almost like the Fourier transform but with ‘\(\cos + \sin\)’ instead of ‘\(\cos + i \cdot \sin\)’. The (continuous version of the) Hartley transform is treated in [143].

The Hartley transform of a purely real sequence is purely real:

\[
\mathcal{H}[a] \in \mathbb{R} \quad \text{for} \quad a \in \mathbb{R}
\]

(24.1-2)

It also is its own inverse:

\[
\mathcal{H}[\mathcal{H}[a]] = a
\]

(24.1-3)

Symmetry is conserved, like for the Fourier transform: the Hartley transform of a symmetric, antisymmetric sequence is symmetric, antisymmetric, respectively. Using the notation from section 20.7 on page 424 one has

\[
\mathcal{H}[a_S] = +\mathcal{H}[a_S] = +\mathcal{H}[a_S]
\]

(24.1-4a)

\[
\mathcal{H}[a_A] = -\mathcal{H}[a_A] = -\mathcal{H}[a_A]
\]

(24.1-4b)

An algorithm for the fast \((n \log(n))\)-computation of the Hartley transform is called a fast Hartley transform (FHT).
24.2 Radix-2 FHT algorithms

24.2.1 Decimation in time (DIT) FHT

Notation: For a length-$n$ sequence $a$ of length, let $X^{1/2}a$ denote the sequence with elements $a_x \cos \pi x/n + \pi x/n sin(x/n)$. The operator $X^{1/2}$ is the equivalent to the operator $S^{1/2}$ of the Fourier transform algorithms. We use the notation (even) and (odd) as introduced in section 20.3 on page 406. The radix-2 decimation in time (DIT) step for the FHT:

$$\mathcal{H}[a]^{(left)}(n/2) = \mathcal{H}[a^{(even)}] + \lambda^{1/2} \mathcal{H}[a^{(odd)}]$$

(24.2-1a)

$$\mathcal{H}[a]^{(right)}(n/2) = \mathcal{H}[a^{(even)}] - \lambda^{1/2} \mathcal{H}[a^{(odd)}]$$

(24.2-1b)

This is the equivalent to relations 20.3-3a and 20.3-3b on page 407.

Pseudo code for a recursive radix-2 DIT FHT (C++ version in FXT: fht/recfht2.cc):

```c
procedure rec_fht_dit2(a[], n, x[])
    // real a[0..n-1] input
    // real x[0..n-1] result
    real b[0..n/2-1], c[0..n/2-1] // workspace
    real s[0..n/2-1], t[0..n/2-1] // workspace
    if n == 1 then
        x[0] := a[0]
        return
    end
    nh := n/2;
    for k:=0 to nh-1
        s[k] := a[2*k] // even indexed elements
        t[k] := a[2*k+1] // odd indexed elements
    end
    rec_fht_dit2(s[], nh, b[])
    rec_fht_dit2(t[], nh, c[])
    hartley_shift(c[], nh, 1/2)
    for k:=0 to nh-1
        x[k] := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    end
end
```

The result is returned in the array in $x[]$. The procedure `hartley_shift()` implements the operator $X^{1/2}$, it replaces element $c_k$ of the input sequence $c$ by $c_k \cos(\pi k/n) + c_{n-k} \sin(\pi k/n)$. As pseudo code:

```c
procedure hartley_shift_05(c[], n)
    // real c[0..n-1] input, result
    { nh := n/2
      j := n-1
      for k:=0 to nh-1
        c := cos( PI*k/n )
        s := sin( PI*k/n )
        {c[k], c[j]} := {c[k]*c+c[j]*s, c[k]*s-c[j]*c} 
        j := j-1
    }
```

C++ implementations are given in FXT: fht/hartleyshift.h. A version that exploits the symmetry of the trigonometric factors is
24.2: Radix-2 FHT algorithms

#define Tdouble long double
#define Sin sinl

template<typename Type>
inline void hartley_shift_05_v2rec(Type *f, ulong n)
{
    const ulong nh = n/2;
    if ( n>=4 )
    {
        ulong i0=nh/2, j0=3*i0;
        Type fi = f[i0], fj = f[j0];
        double cs = SQRT1_2;
        f[i0] = (fi + fj) * cs;
        f[j0] = (fi - fj) * cs;
        if ( n>=8 )
        {
            const Tdouble phi0 = PI/n;
            Tdouble be = Sin(phi0), al = Sin(0.5*phi0); al *= (2.0*al);
            Tdouble s = 0.0, c = 1.0;
            for (ulong i=1, j=n-1, k=nh-1, l=nh+1; i<k; ++i, --j, --k, ++l)
            {
                Tdouble tt = c; c -= (al*tt+be*s); s -= (al*s-be*tt); }
            fi = f[1];
            fj = f[0];
            f[1] = fi * (double)c + fj * (double)s;
            f[0] = fi * (double)s - fj * (double)c;
            fi = f[k];
            fj = f[l];
            f[k] = fi * (double)s + fj * (double)c;
            f[l] = fi * (double)c - fj * (double)s;
        }
    }
}

#undef Tdouble
#undef Sin

Pseudo code for a non-recursive radix-2 DIT FHT:

procedure fht_depth_first_dit2(a[], ldn)
{ // real a[0..n-1] input,result
    n := 2**ldn // length of a[] is a power of 2
    revbin_permute(a[], n)
    for ldm:=1 to ldn
    {
        m := 2**ldm
        mh := m/2
        m4 := m/4
        for r:=0 to n-m step m
        {
            for j:=1 to m4-1 // hartley_shift(a[r+mh], mh, 1/2)
            {
                k := mh - j
                u := a[r+mh+j]
                v := a[r+mh+k]
                c := cos(j*PI/mh)
                s := sin(j*PI/mh)
                {u, v} := {u*c+v*s, u*s-v*c}
                a[r+mh+j] := u
                a[r+mh+k] := v
            }
        }
    for j:=0 to mh-1
    {
        u := a[r+j]
        v := a[r+j+mh]
    }
}
The derivation of the ‘usual’ DIT2 FHT algorithm starts by combining the Hartley-shift with the sum/diff-operations [FXT: fht/fhtdit2.cc]:

```c
void fht_depth_first_dit2(double *f, ulong ldn)
{
    const ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;
        for (ulong r=0; r<n; r+=m)
        {
            // j == 0:
            ulong t1 = r;
            ulong t2 = t1 + mh;
            sumdiff(f[t1], f[t2]);
        }
        if ( m4 )
        {
            ulong t1 = r + m4;
            ulong t2 = t1 + mh;
            sumdiff(f[t1], f[t2]);
        }
        for (ulong j=1, k=mh-1; j<k; ++j,--k)
        {
            double s, c;
            SinCos(phi0*j, &s, &c);
            ulong tj = r + mh + j;
            ulong tk = r + mh + k;
            double fj = f[tj];
            double fk = f[tk];
            f[tj] = fj * c + fk * s;
            f[tk] = fj * s - fk * c;
            ulong t1 = r + j;
            ulong t2 = tj; // == t1 + mh;
            sumdiff(f[t1], f[t2]);
            t1 = r + k;
            t2 = tk; // == t1 + mh;
            sumdiff(f[t1], f[t2]);
        }
    }
}
```

Finally, as with the FFT equivalent (see section 20.3.1.3 on page 409), the number of trigonometric computations can be reduced by swapping the innermost loops [FXT: fht/fhtdit2.cc]:

```c
void fht_dit2(double *f, ulong ldn)
// Radix-2 decimation in time (DIT) FHT.
{
    const ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const double phi0 = M_PI/mh;
        for (ulong r=0; r<n; r+=m)
        {
            // j == 0:
            ulong t1 = r;
            ulong t2 = t1 + mh;
            sumdiff(f[t1], f[t2]);
        }
        if ( m4 )
        {
            ulong t1 = r + m4;
            ulong t2 = t1 + mh;
            sumdiff(f[t1], f[t2]);
        }
        for (ulong j=1, k=mh-1; j<k; ++j,--k)
        {
            double s, c;
            SinCos(phi0*j, &s, &c);
            ulong tj = r + mh + j;
            ulong tk = r + mh + k;
            double fj = f[tj];
            double fk = f[tk];
            f[tj] = fj * c + fk * s;
            f[tk] = fj * s - fk * c;
            ulong t1 = r + j;
            ulong t2 = tj; // == t1 + mh;
            sumdiff(f[t1], f[t2]);
            t1 = r + k;
            t2 = tk; // == t1 + mh;
            sumdiff(f[t1], f[t2]);
        }
    }
}
```
const double phi0 = M_PI/mh;

for (ulong r=0; r<n; r+=m)
{
    // j == 0:
    ulong t1 = r;
    ulong t2 = t1 + mh;
    sumdiff(f[t1], f[t2]);
}

if (m4)
{
    ulong t1 = r + m4;
    ulong t2 = t1 + mh;
    sumdiff(f[t1], f[t2]);
}

for (ulong j=1, k=mh-1; j<k; ++j,--k)
{
    double s, c;
    SinCos(phi0*j, &s, &c);

    for (ulong r=0; r<n; r+=m)
    {
        ulong tj = r + mh + j;
        ulong tk = r + mh + k;
        double fj = f[tj];
        double fk = f[tk];
        f[tj] = fj * c + fk * s;
        f[tk] = fj * s - fk * c;
        ulong t1 = r + j;
        ulong t2 = tj; // == t1 + mh;
        sumdiff(f[t1], f[t2]);
        t1 = r + k;
        t2 = tk; // == t1 + mh;
        sumdiff(f[t1], f[t2]);
    }
}

### 24.2.2 Decimation in frequency (DIF) FHT

The radix-2 decimation in frequency step for the FHT is (compare to relations 20.3-6a and 20.3-6b on page 410):

\[
\mathcal{H}[a]^{(even)} \overset{n/2}{=} \mathcal{H} \left[ a^{(left)} + a^{(right)} \right] \tag{24.2-2a}
\]

\[
\mathcal{H}[a]^{(odd)} \overset{n/2}{=} \mathcal{H} \left[ \chi^{1/2} \left( a^{(left)} - a^{(right)} \right) \right] \tag{24.2-2b}
\]

Pseudo code for a recursive radix-2 DIF FHT (the C++ equivalent is given in [FXT: fht/recfht2.cc]):

```cpp
procedure rec_fht_dif2(a[], n, x[])
// real a[0..n-1] input
// real x[0..n-1] result
{
    real b[0..n/2-1], c[0..n/2-1] // workspace
    real s[0..n/2-1], t[0..n/2-1] // workspace

    if n == 1 then
    {
        x[0] := a[0]
        return
    }
    nh := n/2;

    for k:=0 to nh-1
```
Chapter 24: The Hartley transform

\{ s[k] := a[k] // 'left' elements \\
t[k] := a[k+nh] // 'right' elements \}

for k:=0 to nh-1 
\{ 
{s[k], t[k]} := {s[k]+t[k], s[k]-t[k]} 
\}
hartley_shift(t[], nh, 1/2)
rec_fht_dif2(s[], nh, b[])
rec_fht_dif2(t[], nh, c[])

j := 0
for k:=0 to nh-1 
\{ 
x[j] := b[k] 
x[j+1] := c[k] 
j := j+2 
\}

Pseudo code for a non-recursive radix-2 DIF FHT (C++ version in [FXT: fht/fhtdif2.cc]):

procedure fht_depth_first_dif2(a[], ldn)
// real a[0..n-1] input,result
\{ 
\n := 2**ldn // length of a[] is a power of 2 
\for ldm:=ldn to 1 step -1 
\{ 
\m := 2**ldm 
\mh := m/2 
\m4 := m/4 
\for r:=0 to n-m step m 
\{ 
\for j:=0 to mh-1 
\{ 
\u := a[r+j] 
\v := a[r+j+mh] 
\a[r+j] := u + v 
\a[r+j+mh] := u - v 
\} 
\for j:=1 to m4-1 
\{ 
\k := mh - j 
\u := a[r+mh+j] 
\v := a[r+mh+k] 
\c := \cos(j*\pi/mh) 
\s := \sin(j*\pi/mh) 
\{u, v\} := {u*c+v*s, u*s-v*c} 
\a[r+mh+j] := u 
\a[r+mh+k] := v 
\} 
\} 
\revbin_permute(a[], n) 
\}

The ‘usual’ DIF2 FHT algorithm then is again obtained by swapping the inner loops, a C++ implemen-
tation is [FXT: fht/diff2() in fht/fhtdif2.cc]:

void fht_dif2(double *f, ulong ldn)
// Radix-2 decimation in frequency (DIF) FHT
\{ 
\const ulong n = (1UL<<ldn); 
\for (ulong ldm=ldn; ldm>=1; --ldm) 
\{ 
\const ulong m = (1UL<<ldm); 
}
24.3: Complex FT by HT

The relations between the HT and the FT can be read off directly from their definitions and their symmetry relations. Let $\sigma$ be the sign of the exponent in the FT, then the HT of a complex sequence $d \in \mathbb{C}$ is

$$
\mathcal{F}[d] = \frac{1}{2} \left( \mathcal{H}[d] + \overline{\mathcal{H}[d]} + \sigma i \left( \mathcal{H}[d] - \overline{\mathcal{H}[d]} \right) \right)
$$

(24.3-1)

Written out for the real and imaginary part of $d = a + ib$ ($a, b \in \mathbb{R}$):

$$
\text{Re} \mathcal{F}[a + ib] = \frac{1}{2} \left( \mathcal{H}[a] + \overline{\mathcal{H}[a]} - \sigma \left( \mathcal{H}[b] - \overline{\mathcal{H}[b]} \right) \right)
$$

(24.3-2a)

$$
\text{Im} \mathcal{F}[a + ib] = \frac{1}{2} \left( \mathcal{H}[b] + \overline{\mathcal{H}[b]} + \sigma \left( \mathcal{H}[a] - \overline{\mathcal{H}[a]} \right) \right)
$$

(24.3-2b)

Using the symmetry relations [24.1-4a] and [24.1-4b] on page 511, one can recast the relations as

$$
\text{Re} \mathcal{F}[a + ib] = \frac{1}{2} \mathcal{H}[aS - \sigma bA]
$$

(24.3-3a)

$$
\text{Im} \mathcal{F}[a + ib] = \frac{1}{2} \mathcal{H}[bS + \sigma aA]
$$

(24.3-3b)
Both formulations lead to the very same conversion procedure. The following pseudo code is for a complex FT by HT conversion:

```plaintext
1 fht_fft_conversion(a[], b[], n, is)
2 // preprocessing to use two length-n FHTs
3 // to compute a length-n complex FFT
4 // or
5 // post-processing to use two length-n FHTs
6 // to compute a length-n complex FFT
7 // Self-inverse.
8 {
9     for k:=1 to n/2-1
10         {  
11             t := n-k
12             as := a[k] + a[t]
13             aa := a[k] - a[t]
14             bs := b[k] + b[t]
15             ba := b[k] - b[t]
16             aa := is * aa
17             ba := is * ba
18             a[k] := 1/2 * (as - ba)
19             a[t] := 1/2 * (as + ba)
20             b[k] := 1/2 * (bs + aa)
21             b[t] := 1/2 * (bs - aa)
22         }
23 }
```

The C++ implementations are given in [FXT: fft/fhtfft.cc] for type `double` and [FXT: fft/fhtcfft.cc] for type `complex`. Now we have two options to compute a complex FT by two HTs. Version 1 does the FHTs first:

```plaintext
1 fft_by_fht1(a[], b[], n, is)
2 // real a[0..n-1] input, result (real part)
3 // real b[0..n-1] input, result (imaginary part)
4 {
5     fht(a[], n)
6     fht(b[], n)
7     fht_fft_conversion(a[], b[], n, is)
8 }
```

Version 2 does the FHTs at the end of the routine:

```plaintext
1 fft_by_fht2(a[], b[], n, is)
2 // real a[0..n-1] input, result (real part)
3 // real b[0..n-1] input, result (imaginary part)
4 {
5     fht_fft_conversion(a[], b[], n, is)
6     fht(a[], n)
7     fht(b[], n)
8 }
```

Note that the real and imaginary parts of the FT are computed independently by this procedure. This can be very advantageous when the real and imaginary part of complex data lies in separate arrays. The C++ version is given in [FXT: fft/fhtfft.cc].

### 24.4 Complex FT by complex HT and vice versa

A complex valued HT is simply two HTs (one of the real, one of the imaginary part). So we can use either version from section 24.3 and there is nothing new. Really? If one has a type `complex` version of both the conversion and the FHT routine then the complex FFT can be computed as either

```plaintext
1 fft_by_fht1(c[], n, is)
2 // complex c[0..n-1] input, result
3 {
4     fht(c[], n)
5     fht_fft_conversion(c[], n, is)
6 }
```
or the same with swapped statements.

This may not make you scream but here is the message: it makes sense to do so. One saves half of the trigonometric computations and book keeping. It is pretty easy to derive a complex FHT from the real version and with a well optimized FHT you get an even better optimized FFT. C++ implementations of complex FHTs are given in [FXT: fht/cfhtdif.cc] (DIF algorithm), [FXT: fht/cfhtdit.cc] (DIT algorithm), and, for zero padded data, [FXT: fht/cfht0.cc]. The other way round: computation of a complex FHT using FFTs. Let \( T \) be the operator corresponding to the \texttt{fht\_fft\_conversion}. The operator is its own inverse: \( T = T^{-1} \). We have seen that

\[
\mathcal{F} = \mathcal{H} \cdot T \quad \text{and} \quad \mathcal{F} = T \cdot \mathcal{H} \quad (24.4-1)
\]

Thereby (multiply the relations with \( T \) and use \( T \cdot T = 1 \)):

\[
\mathcal{H} = T \cdot \mathcal{F} \quad \text{and} \quad \mathcal{H} = \mathcal{F} \cdot T \quad (24.4-2)
\]

Hence we have either

\begin{verbatim}
1  fht_by_fft(c[], n, is)
2  // complex c[0..n-1] input,result
3  {
4      fft(c[], n)
5      fht_fft_conversion(c[], n, is)
6  }
\end{verbatim}

or the same thing with swapped lines [FXT: fft/fhtcfft.cc]. The same ideas also work for separate real and imaginary parts but in that case one should rather use separate FHTs for the two arrays.

### 24.5 Real FT by HT and vice versa

To express the real and imaginary part of a Fourier transform of a purely real sequence \( a \in \mathbb{R} \) by its Hartley transform use relations 24.3-2a and 24.3-2b on page 517 and set \( b = 0 \):

\[
\begin{align*}
\Re \mathcal{F}[a] &= \frac{1}{2} (\mathcal{H}[a] + \overline{\mathcal{H}[a]}) \quad (24.5-1a) \\
\Im \mathcal{F}[a] &= \sigma \frac{1}{2} (\mathcal{H}[a] - \overline{\mathcal{H}[a]}) \quad (24.5-1b)
\end{align*}
\]

A C++ implementation is [FXT: fht\_real\_complex\_fft() in \texttt{realfft/realfftbyfht.cc}]:

\begin{verbatim}
1  template <typename Type>
2  static inline void sumdiff05(Type &a, Type &b)
3  // {a, b} <--| {0.5*(a+b), 0.5*(a-b)}
4  { Type t=(a-b)*0.5; a+=b; a*=0.5; b=t; }
5  
6  template <typename Type>
7  static inline void sumdiff05_r(Type &a, Type &b)
8  // {a, b} <--| {0.5*(a+b), 0.5*(b-a)}
9  { Type t=(b-a)*0.5; a+=b; a*=0.5; b=t; }
10 
11 void
12 fht_real_complex_fft(double *f, ulong ldn, int is/*=+1*/)
13 {
14    fht(f, ldn);
15    const ulong n = (1UL<<ldn);
16    if ( is>0 ) for (ulong i=1,j=n-1; i<j; i++,j--) sumdiff05(f[i], f[j]);
17    else for (ulong i=1,j=n-1; i<j; i++,j--) sumdiff05_r(f[i], f[j]);
18 }
\end{verbatim}
At the end of the procedure the ordering of the output data \( c = F[a] \in \mathbb{C} \) is
\[
\begin{align*}
a[0] &= \Re c_0 \\
a[1] &= \Re c_1 \\
a[2] &= \Re c_2 \\
\ldots \\
a[n/2] &= \Re c_{n/2} \\
a[n/2 + 1] &= \Im c_{n/2 - 1} \\
a[n/2 + 2] &= \Im c_{n/2 - 2} \\
a[n/2 + 3] &= \Im c_{n/2 - 3} \\
\ldots \\
a[n - 1] &= \Im c_1
\end{align*}
\]

The inverse procedure is given in [FXT: realfft/realfftbyfht.cc]:
\[
\begin{verbatim}
void fht_complex_real_fft(double *f, ulong ldn, int is/*=+1*/)
{
    const ulong n = (1UL<<ldn);
    if ( is>0 ) for (ulong i=1,j=n-1; i<j; i++,j--) sumdiff(f[i], f[j]);
    else for (ulong i=1,j=n-1; i<j; i++,j--) diffsum(f[i], f[j]);
    fht(f,ldn);
}
\end{verbatim}
\]
The function \texttt{sumdiff()} is defined in [FXT: aux0/sumdiff.h]:
\[
\begin{verbatim}
template <typename Type>
    static inline void sumdiff(Type &a, Type &b)
    // {a, b} <--| {a+b, a-b}
    { Type t=a-b; a+=b; b=t; }
\end{verbatim}
\]
The input has to be ordered as given above (relations \text{24.5-2}). The sign of the transform \((i)\) has to be the same as with the forward version.

Computation of a (real-valued) FHT using a real-valued FFT proceeds similar as for complex versions. Let \( T_{c2r} \) be the operator corresponding to the post-processing in \texttt{real_complex_fft_by_fht()}, and \( T_{r2c} \) correspond to the preprocessing in \texttt{complex_real_fft_by_fht()}. That is
\[
F_{c2r} = H \cdot T_{c2r} \quad \text{and} \quad F_{r2c} = T_{r2c} \cdot H
\]

The operators are mutually inverse: \( T_{r2c} = T_{c2r}^{-1} \) and \( T_{c2r} = T_{r2c}^{-1} \). Multiplying the relations and using \( T_{r2c} \cdot T_{c2r} = T_{c2r} \cdot T_{r2c} = 1 \) gives
\[
H = T_{c2r} \cdot F_{r2c} \quad \text{and} \quad H = F_{c2r} \cdot T_{r2c}
\]

The corresponding code should be obvious. Watch out for real-to-complex FFTs that use a different ordering of the output than given in relation \text{24.5-2}.

### 24.6 Higher radix FHT algorithms

Higher radix FHT algorithms seem to get complicated due to the structure of the Hartley shift operator. In fact there is a straightforward way to turn any FFT decomposition into an FHT algorithm.
For the moment assume that we want to compute a complex HT, further assume we want to use a radix-$r$ algorithm. At each step we have $r$ short HTs and want to combine them to a longer HT but we do not know how this might be done. In section 24.3 on page 517 we learned how to turn a HT into an FT using the $T$-operator. And we have seen radix-$r$ algorithms for the FFT. The crucial idea is to use the conversion operator $T$ as a wrapper around the FFT-step that combines several short FTs into a longer one. Here is how to turn a radix-$r$ FFT-step into an FHT-step, simply do the following:

1. first convert the $r$ short HTs into FTs (use $T$ on the subsequences)
2. then perform the radix-$r$ the FFT step
3. finally convert the FT into a HT (use $T$ on the sequence)

For efficient implementations one obviously wants to combine the computations.

To obtain real-valued FHTs note that the real and imaginary parts do not ‘mix’: one can use the identical algorithm with real input (and the corresponding data types). With a radix-$r$ step the scheme always accesses $2^r$ elements simultaneously. The symmetry of the trigonometric factors is thereby automatically exploited. Splitting steps for the radix-4 FHT and the split-radix FHT are given in [249].

### 24.7 Convolution via FHT

The convolution property of the Hartley transform can be stated as

$$
\mathcal{H}[a \ast b] = \frac{1}{2} \left( \mathcal{H}[a] \mathcal{H}[b] - \overline{\mathcal{H}[a]} \overline{\mathcal{H}[b]} + \overline{\mathcal{H}[a]} \mathcal{H}[b] + \mathcal{H}[a] \overline{\mathcal{H}[b]} \right)
$$

or, with $c := \mathcal{H}[a]$ and $d := \mathcal{H}[b]$, written element-wise:

$$
\mathcal{H}[a \ast b]_k = \frac{1}{2} \left( c_k d_k - \overline{c_k} \overline{d_k} + c_k \overline{d_k} + \overline{c_k} d_k \right)
$$

or

$$
\begin{align*}
\mathcal{H}[a \ast b]_k &= \frac{1}{2} \left( c_k (d_k + \overline{d_k}) + \overline{c_k} (d_k - \overline{d_k}) \right) \\
&= \frac{1}{2} \left( c_k (d_k + \overline{d_k}) + \overline{c_k} (d_k - \overline{d_k}) \right)
\end{align*}
$$

The latter form reduces the number of multiplications. When turning the relation into an algorithm one has to keep in mind that both elements $y_k = \mathcal{H}[a \ast b]$ and $y_{-k}$ must be computed simultaneously.

### 24.7.1 Implementation as pseudo code

Pseudo code for the cyclic convolution of two real valued sequences $x[]$ and $y[]$ via the FHT. $n$ must be even, the result is returned in $y[]$:

```plaintext
procedure fht_cyclic_convolution(x[], y[], n)
    // real x[0..n-1] input, modified
    // real y[0..n-1] result
    { // transform data:
        fht(x[], n)
        fht(y[], n)
        // convolution in transformed domain:
        j := n-1
        for i:=1 to n/2-1
            { // xi := x[i]
                xj := x[j]
                yp := y[i] + y[j] // == y[j] + y[i]
                y[i] := (xi*yp + xj*ym)/2
                y[j] := (xj*yp - xi*ym)/2
            }
    }
```

[fftbook draft of 2008-August-17]
Chapter 24: The Hartley transform

j := j-1
y[0] := x[0]*y[0]
if n>1 then y[n/2] := x[n/2]*y[n/2]

// transform back:
fht(y[], n)

// normalize:
for i:=0 to n-1
{ y[i] := y[i] / n }

It is assumed that the procedure fht() does no normalization. The C++ equivalent is given in [FXT: convolution/fhtenv1.cc].

Equation 24.7-2a (slightly optimized) for the auto convolution is

\[
\mathcal{H}[a \ast a]_k = \frac{1}{2} (c_k (c_k + c_k) + c_k (c_k - c_k)) \quad (24.7-3a)
\]

\[
= c_k c_k + \frac{1}{2} (c_k^2 - c_k^2) \quad (24.7-3b)
\]

where \( c = \mathcal{H}[a] \).

We give pseudo code for the cyclic auto convolution that uses a fast Hartley transform, \( n \) must be even:

```
procedure cyclic_self_convolution(x[], n)
  // real x[0..n-1] input, result
  { // transform data:
    fht(x[], n)
   // convolution in transformed domain:
    j := n-1
    for i:=1 to n/2-1
    { c[i] := x[i]
      c[j] := x[j]
      t1 := c[i]*c[j] // == cj*ci
      t2 := 1/2*(c[i]*c[i]-c[j]*c[j]) // == -1/2*(cj*cj-ci*ci)
      x[i] := t1 + t2
      x[j] := t1 - t2
      j := j-1
    }
    x[0] := x[0]*x[0]
    if n>1 then x[n/2] := x[n/2]*x[n/2]
  // transform back:
  fht(x[], n)
  // normalize:
  for i:=0 to n-1
  { y[i] := y[i] / n }
}
```

For odd \( n \) replace the line

```
for i:=1 to n/2-1
```

by

```
for i:=1 to (n-1)/2
```

and omit the line

```
if n>1 then x[n/2] := x[n/2]*x[n/2]
```

in both procedures above.
24.7.2 C++ implementations

The FHT based routine for the cyclic convolution of two real sequences is:

```cpp
void fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht(f, ldn);
    fht(g, ldn);
    fht_convolution_core(f, g, ldn);
    fht(g, ldn);
}
```

The equivalent to the element-wise multiplication is given in [FXT: convolution/fhtcnvlcore.cc]:

```cpp
void fht_convolution_core(const double * restrict f, double * restrict g, ulong ldn, double v/*=0.0*/)
// Auxiliary routine for the computation of convolutions
// via Fast Hartley Transforms.
// ldn := base-2 logarithm of the array length.
// v!=0.0 chooses alternative normalization.
{
    const ulong n = (1UL<<ldn);
    if ( v==0.0 ) v = 1.0/n;
    g[0] *= (v * f[0]);
    const ulong nh = n/2;
    if ( nh>0 )
    {
        g[nh] *= (v * f[nh]);
        v *= 0.5;
        for (ulong i=1,j=n-1; i<j; i++,j--) fht_mul(f[i], f[j], g[i], g[j], v);
    }
}
```

where [FXT: convolution/fhtmulsqr.h]:

```cpp
template <typename Type>
static inline void fht_mul(Type xi, Type xj, Type &yi, Type &yj, double v)
// yi <-- v*( 2*xi*xj + xi*xi - xj*xj )
// yj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type h1p = xi, h1m = xj;
    Type s1 = h1p + h1m, d1 = h1p - h1m;
    Type h2p = yi, h2m = yj;
    yi = (h2p * s1 + h2m * d1) * v;
    yj = (h2m * s1 - h2p * d1) * v;
}
```

A C++ implementation of the FHT based self-convolution is given in [FXT: convolution/fhtcnvla.cc]. It uses the routine:

```cpp
void fht_auto_convolution_core(double *f, ulong ldn, double v/*=0.0*/)
// v!=0.0 chooses alternative normalization
{
    const ulong n = (1UL<<ldn);
    if ( v==0.0 ) v = 1.0/n;
    f[0] *= (v * f[0]);
    if ( n>=2 )
    {
        const ulong nh = n/2;
        f[nh] *= (v * f[nh]);
        v *= 0.5;
        for (ulong i=1,j=n-1; i<nh; i++,j--) fht_sqr(f[i], f[j], v);
    }
}
```

where [FXT: convolution/fhtmulsqr.h]:

```cpp
template <typename Type>
static inline void fht_sqr(Type &xi, Type &xj, double v)
```

[fxtbook draft of 2008-August-17]
24.7.3 Avoiding the revbin permutations

The observation that the revbin permutations can be omitted with FFT based convolutions (see section 21.1.3 on page 439) applies again [FXT: convolution/fhtcnvlcore.cc]:

```c
void
fht_convolution_revbin_permuted_core(const double * restrict f,
                               double * restrict g,
                               ulong ldn,
                               double v/*=0.0*/)
// Same as fht_convolution_core() but with data access in revbin order.
{
    const ulong n = (1UL<<ldn);
    if ( v==0.0 ) v = 1.0/n;
    g[0] *= (v * f[0]); // 0 == revbin(0)
    if ( n>=2 ) g[1] *= (v * f[1]); // 1 == revbin(nh)
    if ( n<4 ) return;
    v *= 0.5;
    const ulong nh = (n>>1);
    ulong r=nh, rm=n-1; // nh == revbin(1), n1-1 == revbin(n-1)
    fht_mul(f[r], f[rm], g[r], g[rm], v);
    ulong k=2, km=n-2;
    while ( k<nh )
    {
        // k even:
        rm = nh; ulong tr = r;
        r=nh; for (ulong m=(nh>>1); !(r^=m)&m); m>>=1) {;}
        fht_mul(f[r], f[rm], g[r], g[rm], v);
        --km;
        ++k;
        // k odd:
        rm += (tr-r);
        r += nh;
        fht_mul(f[r], f[rm], g[r], g[rm], v);
        --km;
        ++k;
    }
}
```

The optimized version saving three revbin permutaions is [FXT: convolution/fhtcnvl.cc]:

```c
void fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht_dif_core(f, ldn);
    fht_dif_core(g, ldn);
    fht_convolution_revbin_permuted_core(f, g, ldn);
    fht_dit_core(g, ldn);
}
```

[fxtbook draft of 2008-August-17]
24.8 Negacyclic convolution via FHT

Pseudo code for the computation of the negacyclic (auto-) convolution via FHT:

```c
procedure negacyclic_self_convolution(x[], n)
    // real x[0..n-1] input, result
    { 
     hartley_shift_05(x, n) // preprocess
     fht(x, n) // transform data
     // convolution in transformed domain:
     j := n-1
     for i:=0 to n/2-1 // here i starts from zero
     {
         a := x[i]
         b := x[j]
         x[i] := a*b+(a*a-b*b)/2
         x[j] := a*b-(a*a-b*b)/2
         j := j-1
     }
     fht(x, n) // transform back
     hartley_shift_05(x, n) // postprocess
    }
```

C++ implementations for the negacyclic convolution and self convolution are given in \[FXT: convo-

tution/fhtnegacnvl.cc\]. The FHT-based negacyclic convolution is turns out to be extremely useful for

the computation of weighted transforms, for example in the MFA-based convolution for real input, see

section 21.4.3 on page 449.

24.9 Localized FHT algorithms

Localized routines for the FHT can be obtained by slight modifications of the corresponding algorithms

for the Walsh transform described in section 22.6 on page 469 The decimation in time (DIT) version is

\[FXT: fht/fhtloc2.h\]:

```c
template <typename Type>
 void fht_loc_dit2_core(Type *f, ulong ldn)
 { 
     if ( ldn<=13 ) // sizeof(Type)*(2**threshold) <= L1_CACHE_BYTES
     {
         fht_dit_core(f, ldn);
         return;
     }
     // Recursion:
     fht_dit_core_2(f+2); // ldm==1
     fht_dit_core_4(f+4); // ldm==2
     fht_dit_core_8(f+8); // ldm==3
     for (ulong ldm=4; ldm<=ldn; ++ldm) fht_loc_dit2_core(f+(1UL<<ldm), ldm);
     for (ulong ldm=1; ldm<=ldn; ++ldm)
     { 
         const ulong m = (1UL<<ldm);
         const ulong mh = (m>>1);
         hartley_shift_05(f+mh, mh);
         for (ulong t1=0, t2=mh; t1<mh; ++t1, ++t2) sumdiff(f[t1], f[t2]);
     }
 }
```

The routine \texttt{hartley\_shift\_05()} is described in 24.2.1 on page 512 One should choose a implementation

that uses trigonometric recursion as this improves performance considerably.

The decimation in frequency (DIF) version is:

```c
template <typename Type>
 void fht_loc_dif2_core(Type *f, ulong ldn)
 { 
     if ( ldn<=13 ) // sizeof(Type)*(2**threshold) <= L1_CACHE_BYTES
     {
         fht_dit_core(f, ldn);
         return;
     }
     // Recursion:
     fht_dit_core_2(f+2); // ldm==1
     fht_dit_core_4(f+4); // ldm==2
     fht_dit_core_8(f+8); // ldm==3
     for (ulong ldm=4; ldm<=ldn; ++ldm) fht_loc_dit2_core(f+(1UL<<ldm), ldm);
     for (ulong ldm=1; ldm<=ldn; ++ldm)
     { 
         const ulong m = (1UL<<ldm);
         const ulong mh = (m>>1);
         hartley_shift_05(f+mh, mh);
         for (ulong t1=0, t2=mh; t1<mh; ++t1, ++t2) sumdiff(f[t1], f[t2]);
     }
 } 
```
{ if ( ldn<=13 ) // sizeof(Type)*(2**threshold) <= L1_CACHE_BYTES
  
  fht_dif_core(f, ldn);
  return;
}

for (ulong ldm=ldn; ldm>=1; --ldm)
{
  const ulong m = (1UL<<ldm);
  const ulong mh = (m>>1);
  for (ulong t1=0, t2=mh; t1<mh; ++t1, ++t2) sumdiff(f[t1], f[t2]);
  hartley_shift_05(f+mh, mh);
}

// Recursion:
  fht_dif_core_2(f+2); // ldm==1
  fht_dif_core_4(f+4); // ldm==2
  fht_dif_core_8(f+8); // ldm==3
for (ulong ldm=4; ldm<ldn; ++ldm) fht_loc_dif2_core(f+(1UL<<ldm), ldm);

The (generated) short-length transforms are given in the files [FXT: fht/shortfhtdifcore.h] and [FXT: fht/shortfhtditcore.h]. For example, the length-8 decimation in frequency routine is

template<typename Type>
inline void
fht_dif_core_8(Type *f)
{
  Type g0, f0, f1, g1;
  sumdiff(f[0], f[4], f0, g0);
  sumdiff(f[2], f[6], f1, g1);
  sumdiff(g0, f0, f1);
  sumdiff(g1, f1, g0);
  Type s1, c1, s2, c2;
  sumdiff(f[1], f[5], s1, c1);
  sumdiff(f[3], f[7], s2, c2);
  sumdiff(s1, s2);
  sumdiff(f0, s1, f[0], f[1]);
  sumdiff(f1, s2, f[2], f[3]);
  c1 *= SQRT2;
  c2 *= SQRT2;
  sumdiff(g0, c1, f[4], f[5]);
  sumdiff(g1, c2, f[6], f[7]);
}

An additional revbin permutation is needed if the data is required in order. The FHT can be computed
by either

fht_loc_dif2_core(f, ldn);
revbin_permute(f, 1UL<<ldn);

or

revbin_permute(f, 1UL<<ldn);
  fht_loc_dit2_core(f+1UL<<ldn, ldm);

Performance for large arrays is excellent: the convolutions based on the transforms [FXT: convolution/fhtlocnvla.cc]

void
loc_fht_auto_convolution(double *f, ulong ldn)
{
  fht_loc_dif2_core(f, ldn);
  fht_auto_convolution_reвин_permuted_core(f, ldn);
  fht_loc_dit2_core(f, ldn);
}

and [FXT: convolution/fhtlocnvl.cc]

void
loc_fht_convolution(double *restrict f, double *restrict g, ulong ldn)
{
  fht_loc_dif2_core(f, ldn);
  fht_loc_dif2_core(g, ldn);
  fht_convolution_reвин_permuted_core(f, g, ldn);
  fht_loc_dit2_core(g, ldn);
}
gave a significant (more than 50 percent) speedup for the high precision multiplication routines (see section \[27.3\] on page \[560\]) used in the hfloat library \[21\].

## 24.10 Two-dimensional FHTs

A two-dimensional FHT can be computed almost as easy as a two-dimensional FFT, only a trivial additional step is needed. Start with the row-column algorithm described in section \[20.10.2\] on page \[433\] \[FXT: fht/twodimfht.cc\]:

```cpp
void row_column_fht(double *f, ulong nr, ulong nc)
// FHT over rows and columns.
// nr := number of rows
// nc := number of columns
{
    ulong n = nr * nc;
    // fht over rows:
    ulong ldc = ld(nc);
    for (ulong k=0; k<n; k+=nc) FHT(f+k, ldc);
    // fht over columns:
    double *w = new double[nr];
    for (ulong k=0; k<nc; k++) skip_fht(f+k, nr, nc, w);
    delete [] w;
}
```

Note that no attempt has been made to make the routine cache friendly: the routine \[\text{skip}_fht()\] \[FXT: fht/skipfht.cc\] simply copies a column into the scratch array, does the FHT and copies the data back. This is not yet a two-dimensional FHT; the following post-processing must be made:

```cpp
void y_transform(double *f, ulong nr, ulong nc)
// Transforms row-column-FHT to 2-dimensional FHT.
// Self-inverse.
// nr := number of rows
// nc := number of columns
{
    ulong rh = nr/2;
    if ( nr&1 ) rh++;
    ulong ch = nc/2;
    if ( nc&1 ) ch++;
    ulong n = nr*nc;
    for (ulong tr=1, ctr=nc; tr<rh; tr++,ctr+=nc) // ctr=nc*tr
        {
            double *pa = f + ctr;
            double *pb = pa + nc;
            double *pc = f + nc - ctr;
            double *pd = pc + nc;
            for (ulong tc=1, tcc=ch; tc<tc; tc++)
                {
                    pa++;
                    pb--;
                    pc++;
                    pd--;
                    double e = (*pa + *pd - *pb - *pc) * 0.5;
                    *pa -= e;
                    *pb += e;
                    *pc += e;
                    *pd -= e;
                }
        }
}
```

The canned routine is therefore

```cpp
void twodim_fht(double *f, ulong nr, ulong nc)
// Two dimensional fast Hartley transform (FHT)
```

[\textit{fxtbook draft of 2008-August-17}]
Chapter 24: The Hartley transform

// nr := number of rows
// nc := number of columns
{
    row_column_fht(f, nr, nc);
    y_transform(f, nr, nc);
}

24.11 Discrete cosine transform (DCT) by HT

The discrete cosine transform (DCT) with respect to the basis

\[ u(k) = \nu(k) \cdot \cos \left( \frac{\pi k(i + 1/2)}{n} \right) \]

where

\[ \nu(k) = \begin{cases} 
1 & \text{if } k = 0 \\
\sqrt{2} & \text{else}
\end{cases} \]

can be computed from the FHT using an auxiliary routine which is its own inverse. As pseudo code:

```plaintext
procedure cos_rot(x[], y[], n)
    // Real x[0..n-1] input
    // Real y[0..n-1] result
    nh := n/2
    y[0] := x[0]
y[nh] := x[nh]
    phi := PI/2/n
    for k:=1 to nh-1
        c := cos(phi*k)
s := sin(phi*k)
cps := (c+s)*sqrt(1/2)
cms := (c-s)*sqrt(1/2)
y[k] := cms*x[k] + cps*x[n-k]
y[n-k] := cps*x[k] - cms*x[n-k]
}
```

The C++ equivalent is [FXT: cos_rot() in dctdct/cosrot.cc].

Pseudo code for the computation of the DCT via FHT:

```plaintext
procedure dcth(x[], ldn)
    // real x[0..n-1] input,result
    n := 2**ldn
    real y[0..n-1] // workspace
    unzip_rev(x, y, n)
fht(y[], ldn)
cos_rot(y[], x[], n)
}
```

where unzip_rev() is the reversed unzip permutation (see section 2.6 on page 101):

```plaintext
procedure unzip_rev(a[], b[], n)
    // real a[0..n-1] input
    // real b[0..n-1] result
    nh := n/2
    for k:=0 to nh-1
        k2 := 2*k
        b[k] := a[k2]
b[nh+k] := a[n-1-k2]
}
```

Pseudo code for the computation of the inverse discrete cosine transform via FHT:
24.12: Discrete sine transform (DST) by DCT

The basis of the discrete sine transform (DST) is

\[ u(k) = \sin \left( \frac{\pi (k + 1) (i + 1/2)}{n} \right) \]  

(24.12-1)

Pseudo code for the computation of the DST via the discrete cosine transform (DCT):

```plaintext
procedure dst(x[], ldn)
  // real x[0..n-1] input, result
  { n := 2**ldn
    nh := n/2
    for k:=1 to n-1 step 2
      { x[k] := -x[k] }
    dct(x, ldn)
    for k:=0 to nh-1
      { swap(x[k], x[n-1-k]) }
  }
```

The corresponding C++ implementation is [FXT: dsth() in dctdst/dsth.cc]. Pseudo code for the computation of the inverse sine transform using the inverse cosine transform:

```plaintext
procedure idst(x[], ldn)
  // real x[0..n-1] input, result
  { n := 2**ldn
    nh := n/2
    for k:=0 to nh-1
      { swap(x[k], x[n-1-k]) }
  }
```

The C++ implementations of both the forward and the backward transform [FXT: dctdst/dcth.cc] avoid the temporary array if no scratch space is supplied. The algorithms are given in [198] and [199]. An alternative variant for the computation of the DCT that also uses the FHT is given in [FXT: dctdst/dctzapata.cc], the algorithm is described in [16].
The C++ version is [FXT: idsth() in dctdst/dsth.cc].

### 24.13 Automatic generation of transform code

FFT **generators** are programs that output FFT routines, usually for fixed (short) lengths. In fact the thoughts here are not at all restricted to FFT codes. However, fast transforms and routines that can be unrolled like those for matrix multiplication or convolution are prime candidates for automated generation.

One can write code generators that have a built-in algorithmic knowledge. We restrict our attention to a simpler method known as *partial evaluation*. Writing such a program is easy: take an existing FFT and change all computations into print statements that emit the necessary code. The process, however, is less than delightful and error-prone.

It would be much better to have another program that takes the existing FFT code as input and emit the code for the generator. Let us call this a *meta-generator*. Implementing such a meta-generator of course is highly nontrivial. It actually is equivalent to writing an interpreter for the language used plus the necessary data flow analysis. A practical compromise is to write a program that, while theoretically not even close to a meta-generator, creates output that, after a little hand editing, is a usable generator code.

One may further want to print the current values of the loop variables of the original code as comments at the beginning of a block. Thereby it is possible to locate the corresponding part (with respect to both file and temporal location) of a piece of generated code in the original file. In addition one may keep the comments of the original code.

With FFTs it is necessary to identify (‘reverse engineer’) the trigonometric values that occur in the process in terms of the corresponding argument (rational multiples of $\pi$). The actual values should be inclined to some greater precision than actually needed, thereby one avoids the generation of multiple copies of the (logically) same value with differences only due to numeric inaccuracies. Printing the arguments, both as they appear and in lowest terms, inside comments helps to understand (or further optimize) the generated code:

```plaintext
double c1=.9807852804032049126182236134; // == cos(Pi*1/16) == cos(Pi*1/16)
double s1=.19509032201612826784284868476; // == sin(Pi*1/16) == sin(Pi*1/16)
double c2=.923879532521126756128183189397; // == cos(Pi*2/16) == cos(Pi*1/8)
double s2=.382683432365089771728459984029; // == sin(Pi*2/16) == sin(Pi*1/8)
```

Automatic verification of the generated codes against the original is a mandatory part of the process.

A level of abstraction for the array indices is of great use: when the print statements in the generator emit some function of the index instead of its plain value it is easy to generate modified versions of the code for permuted input. That is, instead of

```c
    cout << "sumdiff(f0, f2, g[" << k0 << "]", g[" << k2 << "]);" << endl;
    cout << "sumdiff(f1, f3, g[" << k1 << "]", g[" << k3 << "]);" << endl;
```

use

```c
    cout << "sumdiff(f0, f2, " << idxf(g,k0) << ", " << idxf(g,k2) << ");" << endl;
    cout << "sumdiff(f1, f3, " << idxf(g,k1) << ", " << idxf(g,k3) << ");" << endl;
```

where `idxf(g, k)` can be defined to print a modified (for example, revbin-permuted) index `k`.

A generated length-8 DIT FHT core (from [FXT: fht/shortfhtditcore.h]) shall serve as an example:
24.13: Automatic generation of transform code

`template <typename Type>
inline void fht_dit_core_8(Type *f)
// unrolled version for length 8
{
  // start initial loop
  { // fi = 0 g1 = 1
    Type g0, f0, f1, g1;
    sumdiff(f[0], f[1], f0, g0);
    sumdiff(f[2], f[3], f1, g1);
    sumdiff(f0, f1);
    sumdiff(g0, g1);
    Type s1, c1, s2, c2;
    sumdiff(f[4], f[5], s1, c1);
    sumdiff(f[6], f[7], s2, c2);
    sumdiff(s1, s2);
    sumdiff(f0, s1, f[0], f[4]);
    sumdiff(f1, s2, f[2], f[6]);
    c1 *= SQRT2;
    c2 *= SQRT2;
    sumdiff(g0, c1, f[1], f[5]);
    sumdiff(g1, c2, f[3], f[7]);
  } // end initial loop
} // end initial loop
// opcount by generator: #mult=2=0.25/pt  #add=22=2.75/pt

Generated DIF FHT codes for lengths up to 64 are given in [FXT: fht/shortfhtdifcore.h].

The generated codes can be useful when one wants to spot parts of the original code that need further
optimization. Especially repeated trigonometric values and unused symmetries tend to be apparent in
the unrolled code.

It is a good idea to let the generator count the number of operations (multiplications, additions, loads
and stores) of the code it emits. It is even better if those numbers are compared to the corresponding
values found in the compiled assembler code.

Checking the generated machine code

It is possible to have GCC produce the assembler code with the original source interlaced. This is a great
tool for code optimization. The necessary commands are (include- and warning flags omitted)

```
# create assembler code:
c++ -S -fverbose-asm -g -O2 test.cc -o test.s
# create asm interlaced with source lines:
as -ahmd test.s > test.lst
```

For example, the generated length-4 DIT FHT core from [FXT: fht/shortfhtditcore.h] is

```
1 template <typename Type>
2 inline void fht_dit_core_4(Type *f)
3 // unrolled version for length 4
4 {
5  Type f0, f1, f2, f3;
6  sumdiff(f[0], f[1], f0, f1);
7  sumdiff(f[2], f[3], f2, f3);
8  sumdiff(f0, f2, f[0], f[2]);
9  sumdiff(f1, f3, f[1], f[3]);
10 }
```

With Type set to double the generated assembler is (some editing for readability)

```
1 void fht_dit_core_4(double *f)
2 {
3  double f0, f1, f2, f3;
4  sumdiff(f[0], f[1], f0, f1);
5  movlpd (%rdi), %xmm1  ## f, tmp63
6  movlpd 8(%rdi), %xmm0  #, tmp64
7  movlpd 16(%rdi), %xmm2 #, tmp67
8  movsd  %xmm1, %xmm3  # tmp63, f0
9  subsd  %xmm0, %xmm1  # tmp64, f1
```

[fxtbook draft of 2008-August-17]
28. movsd %xmm2, %xmm4  # tmp73, f2
27. addsd %xmm1, 24(%rdi)  # f1,
26. addsd %xmm2, %xmm0  # f3, tmp73
25. subsd %xmm4, %xmm3  # f2, f0
24. movlpd 24(%rdi), %xmm0  #, tmp68
23. addsd %xmm0, %xmm4  # tmp68, f2
22. movsd %xmm1, %xmm0  # f1, tmp73
21. sumdiff(f0, f2, f[0], f[2]);
20. movsd %xmm3, 16(%rdi)  # f0,
19. sumdiff(f1, f3, f[1], f[3]);
18. subsd %xmm2, %xmm0  # f0, f2
17. movsd %xmm1, %xmm0  # f1, tmp73
16. movsd %xmm3, %xmm0  # f0, tmp71
15. movlpd 16(%rdi), %xmm0  #, tmp64
14. addsd %xmm0, %xmm4  # tmp64, f0
13. addsd %xmm0, %xmm4  # tmp71, f3
12. movlpd 24(%rdi), %xmm0  #, tmp68
11. movsd %xmm2, %xmm0  # tmp67, f2
10. subsd %xmm0, %xmm2  # tmp68, f3
9. sumdiff(f1, f3, f[1], f[3]);
8. movsd %xmm1, 24(%rdi)  # f1,
7. movsd %xmm0, 8(%rdi)  # tmp73,
6. movsd %xmm0, %xmm2  # f2,
5. movsd %xmm3, %xmm0  # f3, f0
4. addsd %xmm0, %xmm3  # tmp64, f0
3. addsd %xmm0, %xmm4  # tmp67, f2
2. movlpd 24(%rdi), %xmm0  #, tmp68
1. addsd %xmm0, %xmm3  # tmp64, f0

Note that the assembler code is not always in sync with the corresponding source lines, especially with higher levels of optimization.

24.14 Eigenvectors of the Fourier and Hartley transform *

Let \( a_S := a + \pi \) be the symmetric part of a sequence \( a \), then

\[
F[ F[a_S] ] = a_S \tag{24.14-1}
\]

Now let \( u_+ := a_S + F[a_S] \) and \( u_- := a_S - F[a_S] \), then

\[
F[u_+] = F[a_S] + a_S = a_S + F[a_S] = +1 \cdot u_+ \tag{24.14-2a}
\]
\[
F[u_-] = F[a_S] - a_S = -(a_S - F[a_S]) = -1 \cdot u_- \tag{24.14-2b}
\]

Both \( u_+ \) and \( u_- \) are symmetric. For \( a_A := a - \pi \), the antisymmetric part of \( a \), we have

\[
F[ F[a_A] ] = -a_A \tag{24.14-3}
\]

Therefore with \( v_+ := a_A + iF[a_A] \) and \( v_- := a_A - iF[a_A] \):

\[
F[v_+] = F[a_A] - i_a a_A = -i(a_A + iF[a_A]) = -i \cdot v_+ \tag{24.14-4a}
\]
\[
F[v_-] = F[a_A] + i_a a_A = +i(a_A - iF[a_A]) = +i \cdot v_- \tag{24.14-4b}
\]

Both \( v_+ \) and \( v_- \) are antisymmetric. The sequences \( u_+, u_-, v_+, \) and \( v_- \) are eigenvectors of the FT, with eigenvalues \( +1, -1, -i \) and \( +i \) respectively. The eigenvectors are pairwise perpendicular. Using the relation

\[
a = \frac{1}{2} (u_+ + u_- + v_+ + v_-) \tag{24.14-5}
\]

we can, for a given sequence, find a transform that is a ‘square root’ of the FT: compute \( u_+, u_-, v_+, \) and \( v_- \), and a transform \( F^\lambda[a] \) for \( \lambda \in \mathbb{R} \) as

\[
F^\lambda[a] = \frac{1}{2} \left( (-1)^\lambda u_+ + (-1)^\lambda u_- + (-i)^\lambda v_+ + (+i)^\lambda v_- \right) \tag{24.14-6}
\]

This transform is called the fractional (order) Fourier transform (or simply fractional FT, but see section 21.5.3 on page 452). Then \( F^0[a] \) is the identity and \( F^1[a] \) is the usual FT. The transform \( F^{1/2}[a] \) is a transform so that \( F^{1/2}[F^{1/2}[a]] = F[a] \), that is, a ‘square root’ of the FT. The transform \( F^{1/2}[a] \) is not unique as the expressions \( \pm 1^{1/2} \) and \( \pm i^{1/2} \) are not.
24.14: Eigenvectors of the Fourier and Hartley transform *

The eigenvectors of the Hartley Transform are

\[ u_+ := a + \mathcal{H}[a] \]  \hspace{1cm} (24.14-7a)
\[ u_- := a - \mathcal{H}[a] \]  \hspace{1cm} (24.14-7b)

The eigenvalues are ±1, one has \( \mathcal{H}[u_+] = +1 \cdot u_+ \) and \( \mathcal{H}[u_-] = -1 \cdot u_- \).

Let \( M \) be the \( n \times n \) matrix corresponding to the length-\( n \) Fourier transform with positive sign \( \sigma \), that is \( M_{r,c} = 1/\sqrt{n} \exp \left( 2 \pi i r c/n \right) \). Then its characteristic polynomial (see relation 40.5-2 on page 898) is

\[ p(x) = (x - 1)^{(n+4)/4} (x + 1)^{(n+2)/4} (x - i)^{(n+1)/4} (x + i)^{(n-1)/4} \]  \hspace{1cm} (24.14-8)

We write \( p(x) = x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0 \). The trace of the matrix \( M \) is

\[ \text{Tr}(M) = \sqrt{n} \sum_{k=0}^{n-1} \exp \left( 2i \pi k^2/n \right) \]  \hspace{1cm} (24.14-9)

It equals \((-c_{n-1}, \text{the negative sum of all roots of } p(x), \text{and})\)

\[ 1 + i, \ +1, \ 0, \ +i \]  \hspace{1cm} (24.14-10)

for \( n \mod 4 \equiv 0, 1, 2, 3 \), respectively. A closed form is \((1 + i^{-n}) / (1 - i)\). The generating function for the sequence of values is \(((1 + i) - x) / (1 + (-1 + i) x - i x^2)\).

The determinant of \( M \) equals \((-1)^n c_0, \ (-1)^n \times \text{the product of all roots of } p(x), \text{and})\)

\[ + i, \ +1, \ -1, \ -i, \ -i, \ +1, \ +i \]  \hspace{1cm} (24.14-11)

for \( n \mod 8 \equiv 0, 1, 2, \ldots, 7 \). A closed form is \((1 + i) / 2 (1 + (-i)^n)\). The generating function for the sequence is \((i + x - x^2 - i x^3) / (1 + x^4)\). 

[fxtbook draft of 2008-August-17]
Chapter 25

Number theoretic transforms (NTTs)

We introduce the number theoretic transforms (NTTs). After understanding the necessary concepts from number theory (see also chapter 37) it turns out that the routines for the fast NTTs are rather straightforward translations of the FFT algorithms. We give radix-2 and radix-4 routines but there should be no difficulty to translate any given (complex valued) FFT algorithm into the equivalent NTT algorithm. For the translation of real valued FFT (or FHT) routines one needs to express sines and cosines in modular arithmetic, this is presented in sections 37.12.6 and 37.12.7.

As no rounding errors occur with the underlying modular arithmetic the main application of NTTs is the fast computation of exact convolutions.

25.1 Prime moduli for NTTs

How to make a number theoretic transform out of your FFT:
'Replace \( \exp(\pm 2 \pi i/n) \) by a primitive \( n \)-th root of unity, done.'

We want to implement FFTs in \( \mathbb{Z}/m\mathbb{Z} \) (the ring of integers modulo some integer \( m \)) instead of \( \mathbb{C} \), the (field of the) complex numbers. These FFTs are called number theoretic transforms (NTTs), mod \( m \) FFTs or (if \( m \) is a prime) prime modulus transforms.

There is a restriction for the choice of \( m \): for a length \( n \) NTT we need a primitive \( n \)-th root of unity. A number \( r \) is called an \( n \)-th root of unity if \( r^n = 1 \). It is called a primitive \( n \)-th root if \( r^k \neq 1 \forall k < n \).

In \( \mathbb{C} \) matters are simple: \( e^{\pm 2 \pi i/n} \) is a primitive \( n \)-th root of unity for arbitrary \( n \). For example, \( e^{2 \pi i/21} \) is a primitive 21-th root of unity. Now \( r = e^{2 \pi i/3} \) is also 21-th root of unity but not a primitive root, because \( r^3 = 1 \). A primitive \( n \)-th root of 1 in \( \mathbb{Z}/m\mathbb{Z} \) is also called an element of order \( n \). The ‘cyclic’ property of the elements \( r \) of order \( n \) lies in the heart of all FFT algorithms: \( r^{n+k} = r^k \).

In \( \mathbb{Z}/m\mathbb{Z} \) things are not that simple: for a given modulus \( m \) primitive \( n \)-th roots of unity do not exist for arbitrary \( n \). They exist for some maximal order \( R \) only. Roots of unity of an order different from \( R \) are available only for the divisors \( d_i \) of \( R \): \( r^{R/d_i} \) is a \( d_i \)-th root of unity because \( (r^{R/d_i})^{d_i} = r^R = 1 \).

Therefore \( n \), the length of the transform, must divide the maximal order \( R \). This is the first condition for NTTs.

The operations needed in FFTs are addition, subtraction and multiplication. Division is not needed, except for division by \( n \) for the final normalization after transform and back-transform. Division by \( n \) is multiplication by the inverse of \( n \), so \( n \) must be invertible in \( \mathbb{Z}/m\mathbb{Z} \).
Therefore $n$, the length of the transform, must be coprime to the modulus $m$: $\gcd(n, m) = 1$. This is the second condition for NTTs.

We restrict our attention to prime moduli, though NTTs are also possible with composite moduli. If the modulus is a prime $p$ then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = \text{GF}(p)$: all elements except 0 have inverses and ‘division is possible’. Thereby the second condition is trivially fulfilled for all NTT lengths $n < p$: a prime $p$ is coprime to all integers $n < p$.

Roots of unity are available for the maximal order $R = p - 1$ and its divisors: Therefore the first condition on $n$ for a length-$n$ mod $p$ NTT being possible is that $n$ divides $p - 1$. This restricts the choice for $p$ to primes of the form $p = v n + 1$: for length-$n = 2^k$ NTTs one will use primes like $p = 3 \cdot 5 \cdot 2^{27} + 1$ (31 bits), $p = 3 \cdot 29 \cdot 2^{56} + 1$ (63 bits) or $p = 27 \cdot 2^{59} + 1$ (64 bits).

<table>
<thead>
<tr>
<th>$n$</th>
<th>Factorization</th>
<th>$\log_2(n-1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$2^1$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$3$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>$5$</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>$7$</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>$11$</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>$13$</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>$17$</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>$19$</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>$23$</td>
<td>6</td>
</tr>
<tr>
<td>29</td>
<td>$29$</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>$31$</td>
<td>5</td>
</tr>
</tbody>
</table>

Primes suitable with NTTs (sometimes called FFT-primes) can be generated with the program [FXT: mod/fftprimes-demo.cc]. A shortened sample output is shown in figure 25.1-A. A few moduli that allow for transforms of lengths dividing $2^{40} \cdot 3^2 \cdot 5^2 \cdot 7$ are shown in figure 25.1-B, the data is taken from [FXT: mod/moduli.txt]. We note that primality of moduli suitable for NTTs can easily by tested using Proth’s theorem, see section 37.11.3.1 on page 794.

Figure 25.1-A: Primes suitable for NTTs of lengths dividing $2^{44}$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Factorization + 1</th>
<th>$\log(n-1)/\log(2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$2^1$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$3$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>$5$</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>$7$</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>$11$</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>$13$</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>$17$</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>$19$</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>$23$</td>
<td>6</td>
</tr>
<tr>
<td>29</td>
<td>$29$</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>$31$</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 25.1-B: Primes suitable for NTTs of lengths dividing $2^{40} \cdot 3^2 \cdot 5^2 \cdot 7$. 
25.2 Implementation of NTTs

To implement NTTs (modulo $m$, length $n$) one has to implement modular arithmetics and replace $e^{±2\pi i/n}$ by an primitive $n$-th root $r$ of unity in $\mathbb{Z}/m\mathbb{Z}$ in the code. A C++ class implementing modular arithmetics is [FXT: class mod in mod/mod.h].

For the inverse transform one uses the (mod $m$) inverse $r^{-1}$ of $r$ that was used for the forward transform. The element $r^{-1}$ is also a primitive $n$-th root. Methods for the computation of the modular inverse are described in section 37.1.4 on page 766 (GCD algorithm) and in section 37.6.4 on page 778 (powering algorithm).

While the notion of the Fourier transform as a ‘decomposition into frequencies’ appears to be meaningless for NTTs the algorithms are denoted with ‘decimation in time/frequency’ in analogy to those in the complex domain.

The nice feature of NTTs is that there is no loss of precision in the transform as with the floating point FFTs. Using the trigonometric recursion in its most naive form is mandatory, as the computation of roots of unity is expensive.

25.2.1 Radix-2 DIT NTT

Pseudo code for the radix-2 decimation in time (DIT) NTT (to be called with $\text{ldn}=\log_2(n)$):

```plaintext
procedure mod_fft_dit2(f[], ldn, is)
    // mod_type f[0..2**ldn-1]
    n := 2**ldn
    rn := element_of_order(n) // (mod_type)
    if is<0 then rn := rn**(-1)
    revbin_permute(f[], n)
    for ldm:=1 to ldn
        m := 2**ldm
        mh := m/2
        dw := rn**(2**(ldn-ldm)) // (mod_type)
        w := 1 // (mod_type)
        for j:=0 to mh-1
            for r:=0 to n-m step m
                t1 := r+j
                t2 := t1+mh
                v := f[t2]*w // (mod_type)
                u := f[t1] // (mod_type)
                f[t1] := u+v
                f[t2] := u-v
            w := w*dw // trig recursion
    for ldm:=1 to ldn
        for r:=0 to n-1 step 2
            {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
```

As shown in section 20.3.1 on page 406 it is a good idea to extract the $\text{ldm}=1$ stage of the outermost loop: Replace

```
for ldm:=1 to ldn
```

by

```
for r:=0 to n-1 step 2
```

```
{f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
```
Chapter 25: Number theoretic transforms (NTTs)

The C++ implementation is given in [FXT: ntt/nttdit2.cc]:

void ntt_dit2_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dit2()
// Decimation in time (DIT) radix-2 FFT
// Input data must be in revbin_permuted order
// ldn := base-2 logarithm of the array length
// is := sign of the transform
{
    const ulong n = 1UL<<ldn;
    for (ulong i=0; i<n; i+=2) sumdiff(f[i], f[i+1]);
    for (ulong ldm=2; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r + j;
                const ulong t2 = t1 + mh;
                mod v = f[t2] * w;
                mod u = f[t1];
                f[t1] = u + v;
                f[t2] = u - v;
            }
            w *= dw;
        }
    }
}

void ntt_dit2(mod *f, ulong ldn, int is)
// Radix-2 decimation in time (DIT) NTT
{
    revbin_permute(f, 1UL<<ldn);
    ntt_dit2_core(f, ldn, is);
}

The elements of order $2^k$ are precomputed upon initialization of the mod class. The call to mod::root2pow() is a simple table lookup.

25.2.2 Radix-2 DIF NTT

Pseudo code for the radix-2 decimation in frequency (DIF) NTT:

procedure mod_fft_dif2(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{n := 2**ldn
 dw := element_of_order(n) // (mod_type)
 if is<0 then dw := rn**(-1)
 for ldm=ldn to 1 step -1
 { m := 2**ldm
   mh := m/2
   w := 1 // (mod_type)
   for j=0 to mh-1
}
25.2: Implementation of NTTs

```cpp
void ntt_dif2_core(mod *f, ulong ldn, int is)
   // Auxiliary routine for ntt_dif2().
   // Decimation in frequency (DIF) radix-2 NTT.
   // Output data is in revbin_permuted order.
   // ldn := base-2 logarithm of the array length.
   // is := sign of the transform
   { const ulong n = (1UL<<ldn);
     mod dw = mod::root2pow( is>0 ? ldn : -ldn );
     for (ulong ldm=ldn; ldm>1; --ldm)
     { const ulong m = (1UL<<ldm);
       const ulong mh = (m>>1);
       mod w = mod::one;
       for (ulong j=0; j<mh; ++j)
       { for (ulong r=0; r<n; r+=m)
         { const ulong t1 = r + j;
           const ulong t2 = t1 + mh;
           mod v = f[t2];
           mod u = f[t1];
           f[t1] = (u + v);
           f[t2] = (u - v) * w;
         }
       w *= dw;
     }
     dw *= dw;
   }
for (ulong i=0; i<n; i+=2) sumdiff(f[i], f[i+1]);
```

As in section 20.3.2 on page 409 extract the ldm=1 stage of the outermost loop: Replace the line

```cpp
for ldm:=ldn to 1 step -1
```

by

```cpp
for ldm:=ldn to 2 step -1
```

and insert

```cpp
for r:=0 to n-1 step 2
{ {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
}
```

before the call of revbin_permute(f[], n).

The C++ implementation is given in [FXT: ntt/nttdif2.cc]:
void ntt_dif2(mod *f, ulong ldn, int is)
// Radix-2 decimation in frequency (DIF) NTT
{
    ntt_dif2_core(f, ldn, is);
    revbin_permute(f, 1UL<<ldn);
}

25.2.3 Radix-4 NTTs

The radix-4 versions of the NTT are straightforward translations of the routines that use complex numbers. We simply give the C++ implementations

25.2.3.1 Decimation in time (DIT) algorithm

Code for a radix-4 decimation in time (DIT) NTT [FXT: ntt/dit4.cc]:

```c
static const ulong LX = 2;

void ntt_dit4_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dit4()
// Decimation in time (DIT) radix-4 NTT
// Input data must be in revbin_permuted order
// ldn := base-2 logarithm of the array length
// is := sign of the transform
{
    const ulong n = (1UL<<ldn);
    if ( ldn & 1 ) // n is not a power of 4, need a radix-2 step
    { for (ulong i=0; i<n; i+=2) sumdiff(f[i], f[i+1]); }
    const mod imag = mod::root2pow( is>0 ? 2 : -2 );
    ulong ldm = LX + (ldn&1);
    for ( ; ldm<=ldn ; ldm+=LX)
    { const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);
        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w;
        mod w3 = w;
        for (ulong j=0; j<m4; j++)
        { for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            { const ulong i1 = i0 + m4;
                const ulong i2 = i1 + m4;
                const ulong i3 = i2 + m4;
                mod a0 = f[i0];
                mod a2 = f[i1] * w2;
                mod a1 = f[i2] * w;
                mod a3 = f[i3] * w3;
                mod t02 = a0 + a2;
                mod t13 = a1 + a3;
                f[i0] = t02 + t13;
                f[i2] = t02 - t13;
                t02 = a0 - a2;
                t13 = a1 - a3;
                t13 *= imag;
            }
        }
    }
```

[fxtbook draft of 2008-August-17]

25.2.3.2 Decimation in frequency (DIF) algorithm

Code for a radix-4 decimation in frequency (DIT) NTT [FXT: ntt/nttdif4.cc):

```c
static const ulong LX = 2;

void ntt_dif4_core(mod *f, ulong ldn, int is)
  // Auxiliary routine for ntt_dif4().
  // Decimation in frequency (DIF) radix-4 NTT.
  // Output data is in revbin_permuted order.
  // ldn := base-2 logarithm of the array length.
  // is := sign of the transform
  {
    const ulong n = (1UL<<ldn);
    const mod imag = mod::root2pow( is>0 ? 2 : -2 );
    for (ulong ldm=ldn; ldm>=LX; ldm-=LX)
      {
        const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);
        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w * w;
        mod w3 = w * w2;
        for (ulong j=0; j<m4; j++)
          { // for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            const ulong i1 = i0 + m;
            const ulong i2 = i1 + m;
            const ulong i3 = i2 + m;
            mod a0 = f[i0];
            mod a1 = f[i1];
            mod a2 = f[i2];
            mod a3 = f[i3];
            mod t02 = a0 + a2;
            mod t13 = a1 + a3;
            f[i0] = (t02 + t13) * w;
            f[i1] = (t02 - t13) * w3;
            t02 = a0 - a2;
            t13 = a1 - a3;
            t13 *= imag;
            f[i2] = (t02 + t13) * w;
            f[i3] = (t02 - t13) * w3;
            t02 = a0 - a2;
            t13 = a1 - a3;
          }
      }
    w *= dw;
    w2 = w * w;
```

[ftxbook draft of 2008-August-17]
54        w3 = w * w2;
55    }
56  }
57  if ( ldn & 1 ) // n is not a power of 4, need a radix-2 step
58  {
59      for (ulong i=0; i<n; i+=2) sumdiff(f[i], f[i+1]);
60  }
61  }
62 }

```c

25.3 Convolution with NTTs

The NTTs are natural candidates for the computation of exact integer convolutions, as used in high precision multiplication algorithms. One must keep in mind that all computations are modulo \( m \), the largest value that can be represented is \( m - 1 \). Choosing a modulus that is greater than the maximal possible value of the result avoids any truncation. If \( m \) does not fit into a single machine word this may slow down the computation unacceptably.

It is better to choose \( m \) as the product of several coprime moduli \( m_i \) that are all just below machine word size, compute the convolutions for each modulus \( m_i \), and finally use the Chinese Remainder Theorem (see section 37.7 on page 778) to obtain the result modulo \( m \).

If length-\( n \) FFTs are used for convolution there must be an inverse element for \( n \). This imposes the condition \( \gcd(n, m) = 1 \), the modulus must be coprime to \( n \). For length-\( 2^k \) FFTs this simply means that \( m \) must be odd.

C++ code for the NTT based exact convolution can be found in [FXT: ntt/nttcnvl.cc]. The routines are virtually identical to their complex equivalents. For example, a routine for cyclic self-convolution can be given as

```
1 void
2 ntt_auto_convolution(mod *f, ulong ldn)
3 // Cyclic (self-)convolution.
4 // Use zero padded data for linear convolution.
5 {
6   assert_two_invertible(); // so we can normalize later
7     const int is = +1;
8     const ulong n = (1UL<<ldn);
9     ntt_dif4_core(f, ldn, is); // transform
10    for (ulong i=0; i<n; ++i) f[i] *= f[i]; // multiply element-wise
11    ntt_dit4_core(f, ldn, -is); // inverse transform
12    multiply_val(f, n, (mod(n)).inv() ); // normalize
13 }
```

The revbin permutations are avoided as explained in section 21.1.3 on page 439.

For further applications of the NTT see the survey article [139] and the references given there.
Chapter 26

Fast wavelet transforms

The discrete wavelet transforms are a class of transforms that can be computed in linear time. We treat wavelet transforms whose basis functions have compact support. These can be derived as a generalization of the Haar transform.

26.1 Wavelet filters

We motivate the wavelet transform as a generalization of the 'standard' Haar transform given in section 23.1 on page 493. We reformulate the Haar transform as a sequence of filtering steps.

We consider only (moving average) filters \( F \) defined by \( n \) coefficients ('taps') \( f_0, f_1, \ldots, f_{n-1} \). Let \( A \) be the length-\( N \) sequence \( a_0, a_1, \ldots, a_{N-1} \). We define \( F_k(A) \) as the weighted sum

\[
F_k(A) := \sum_{j=0}^{n-1} f_j a_{k+j \mod N} \tag{26.1-1}
\]

That is, \( F_k(A) \) is the result of applying the filter \( F \) to the \( n \) elements \( a_k, a_{k+1}, a_{k+2}, \ldots a_{k+n-1} \), possibly wrapping around.

Now assume that \( N \) is a power of two. Let \( H \) be the low-pass filter defined by \( h_0 = h_1 = +1/\sqrt{2} \), and \( G \) be the high-pass filter defined by \( g_0 = +1/\sqrt{2}, g_1 = -1/\sqrt{2} \). A single filtering step of the Haar transform consists of

- Computing the sums \( s_0 = H_0(A), s_2 = H_2(A), s_4 = H_4(4), \ldots, s_{N-2} = H_{N-2}(A) \)
- Computing the differences \( d_0 = G_0(A), d_2 = G_2(A), d_4 = G_4(4), \ldots, d_{N-2} = G_{N-2}(A) \)
- Writing the sums to the left half of \( A \), and the differences to the right half:
  \( A = [s_0, s_2, s_4, s_6, \ldots, s_{N-2}, d_0, d_2, d_4, d_6, \ldots, d_{N-2}] \)

The Haar transform is obtained by applying the step to the whole sequence, then to its left half, then to its left quarter, \ldots, the left four elements, the left two elements. With the Haar transform no wrap-around occurs.

A the analogous filtering step for the wavelet transform is obtained by defining two length-\( n \) filters \( H \) (low-pass) and \( G \) (high-pass) subject to certain conditions. Firstly, we consider only filters with an even number \( n \) of coefficients.

Secondly, we define coefficients of \( G \) to be the reversed sequence of the coefficients of \( H \) with alternating signs:

\[
g_0 = +h_{n-1}, g_1 = -h_{n-2}, g_2 = +h_{n-3}, g_4 = -h_{n-3}, \ldots, g_{n-3} = -h_2, g_{n-2} = +h_1, g_{n-1} = -h_0.
\]
Thirdly, we require that the resulting transform is orthogonal. Let \( S \) be the matrix corresponding to one filtering step, ignoring the order:

\[
SA = [s_0, d_0, s_2, d_2, s_4, d_4, s_6, d_6, \ldots, s_{N-2}, d_{N-2}]
\]  

(26.1-2)

With length-6 filters and \( N = 16 \) the matrix \( S \) would be

\[
S = \begin{bmatrix}
  h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

(26.1-3a)

\[
= \begin{bmatrix}
  h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{bmatrix}
\]

(26.1-3b)

The orthogonality requires that \( SS^T = I \), that is (setting \( h_j = 0 \) for \( j < 0 \) and \( j \geq n \))

\[
\sum_j h_j^2 = 1 \tag{26.1-4a}
\]

\[
\sum_j h_j h_{j+2} = 0 \tag{26.1-4b}
\]

\[
\sum_j h_j h_{j+4} = 0 \tag{26.1-4c}
\]

In general, the following \( n/2 \) wavelet conditions are obtained:

\[
\sum_j h_j^2 = 1 \tag{26.1-5a}
\]

\[
\sum_j h_j h_{j+2i} = 0 \text{ where } i = 1, 2, 3, \ldots, n/2 - 1 \tag{26.1-5b}
\]

We call a filter \( H \) satisfying these conditions a wavelet filter.

For the wavelet transform with \( n = 2 \) filter taps there is only condition, \( h_0^2 + h_1^2 = 1 \), leading to the parametric solution \( h_0 = \sin(\phi), h_1 = \cos(\phi) \). Setting \( \phi = \pi/4 \) one obtains \( h_0 = h_1 = 1/\sqrt{2} \), corresponding to the Haar transform.
26.2 Implementation

A container class for wavelet filters is [FXT: class wavelet_filter in wavelet/waveletfilter.h]:

```cpp
class wavelet_filter
{
public:
  double *h_; // low-pass filter
  double *g_; // high-pass filter
  ulong n_; // number of taps

  void ctor_core()
  {
    h_ = new double[n_];
    g_ = new double[n_];
  }

  wavelet_filter(const double *w, ulong n=0)
  {
    if ( 0!=n ) n_ = n;
    else // zero terminated array w[]
    {
      n_ = 0;
      while ( w[n_]! = 0 ) ++n_;
    }
  ctor_core();

  for (ulong i=0, j=n_-1; i<n_; ++i, --j)
  {
    h_[i] = w[i];
    if ( !(i&1) ) g_[j] = -h_[i]; // even indices
    else g_[j] = +h_[i]; // odd indices
  }
}

bool check(double eps=1e-6) const
{
  if ( fabs(norm_sqr(0)-1.0) > eps ) return false;
  for (ulong i=1; i<n_/2; ++i)
    if ( fabs(norm_sqr(i)) > eps ) return false;
  return true;
}
```

where `norm_sqr()` computes the sums in the relations (26.1-5a) and (26.1-5b):

```cpp
static double norm_sqr(const double *h, ulong n, ulong s=0)
{
  s *= 2; // Note!
  if ( s>=n ) return 0.0;
  double v = 0;
  for (ulong k=0, j=s; j<n; ++k, ++j) v += (h[k]*h[j]);
  return v;
}
```

double norm_sqr(ulong s=0) const { return norm_sqr(h_, n_, s); }

A wavelet step can be implemented as [FXT: wavelet/wavelet.cc]:

```cpp
void wavelet_step(double *f, ulong n, const wavelet_filter &wf, double *t)
{
  const ulong nh = (n>>1);
  const ulong n = n-1; // mask to compute modulo n (n is a power of two)
  for (ulong i=0, j=0; i<n; i+=2, ++j) // i \in [0,2,4,...,n-2]; j \in [0,1,2,...,n/2-1]
  {
    double s = 0.0, d = 0.0;
    for (ulong k=0; k<wf.n_; ++k)
    {
```

[fxtbook draft of 2008-August-17]
The wavelet transform itself is

```c
void wavelet(double *f, ulong ldn, const wavelet_filter &wf, ulong minm/*=2*/)
{
    ulong n = (1UL<<ldn);
    ALLOCA(double, t, n);
    for (ulong m=n; m>=minm; m>>=1) wavelet_step(f, m, wf, t);
}
```

The step for the inverse transform is [FXT: wavelet/invwavelet.cc]:

```c
void inverse_wavelet_step(double *f, ulong n, const wavelet_filter &wf, double *t)
{
    const ulong nh = (n>>1);
    const ulong m = n-1; // mask to compute modulo n (n is a power of two)
    null(t, n); // t[] := [0,0,...,0]
    for (ulong i=0, j=0; i<n; i+=2, ++j)
    {
        const double x = f[j], y = f[nh+j];
        for (ulong k=0; k<wf.n_; ++k)
        {
            ulong w = (i+k) & m;
            t[w] += (wf.h_[k] * x);
            t[w] += (wf.g_[k] * y);
        }
        copy(t, f, n); // f[] := t[]
    }
}
```

The inverse transform itself now is

```c
void inverse_wavelet(double *f, ulong ldn, const wavelet_filter &wf, ulong minm/*=2*/)
{
    ulong n = (1UL<<ldn);
    ALLOCA(double, t, n);
    for (ulong m=minm; m<=n; m<<=1) inverse_wavelet_step(f, m, wf, t);
}
```

A readable source about wavelets is [279].

### 26.3 Moment conditions

As the wavelet conditions do not uniquely define the wavelet filters on can impose additional properties for the filters used. We require that, for an \(2n\)-tap wavelet filter, the first \(n/2\) moments vanish:

\[
\sum_j (-1)^j h_j = 0 \quad (26.3-1a)
\]

\[
\sum_j (-j)^k h_j = 0 \quad \text{where} \quad k = 1, 2, 3, \ldots, n/2 - 1 \quad (26.3-1b)
\]

One motivation for these moment conditions is that for reasonably smooth signals (for which a polynomial approximation is good) the transform coefficients from the high-pass filter (the \(d_k\)) will be close to zero. With compression schemes that simply discard transform coefficients with small values this is a desirable property.
The class [FXT: class wavelet::wavelet_filter in wavelet/waveletfilter.h] has a method to compute the moments of the filter:

```cpp
static double moment(const double *h, ulong n, ulong x=0)
{
  if (0 == x)
  {
    double v = 0.0;
    for (ulong k=0; k<n; k+=2) v += h[k];
    for (ulong k=1; k<n; k+=2) v -= h[k];
    return v;
  }

  double dk;
  double ve = 0;
  dk = 2.0;
  for (ulong k=2; k<n; k+=2, dk+=2.0) ve += (pow(dk,x) * h[k]);
  double vo = 0;
  dk = 1.0;
  for (ulong k=1; k<n; k+=2, dk+=2.0) vo += (pow(dk,x) * h[k]);
  return ve - vo;
}
```

double moment(ulong x=0) const { return moment(h_, n_, x); }

Filter coefficients that satisfy the moment conditions are given in [FXT: wavelet/daubechies.cc]:

```cpp
extern const double Daub1[] = {
  +7.071067811865475244008443621048e-01,
  +7.071067811865475244008443621048e-01,
};
```

```cpp
extern const double Daub2[] = {
  +4.829629131445341433748715998644e-01,
  +8.365163037378079055752937809168e-01,
  +2.241438680420133810259727622404e-01,
  -1.294095225512603811744494188120e-01,
};
```

```cpp
extern const double Daub3[] = {
  +3.326705529500826159985115891390e-01,
  +8.068915093110295769444936040887e-01,
  +4.598775201184937579515919241476e-01,
  -1.3501102001254588663899069993e-01,
  -8.544127388202666162916918177e-02,
  +3.5226291885709536602740664715516e-02,
};
```

```cpp
extern const double Daub4[] = {
  +2.30377813308896600632911830404e-01,
  +7.14865705291564708999219552739e-01,
  +6.3080767929589078817163383006e-01,
  +2.7983694168595842114374718007e-02,
  +1.870348117190930840795706727890e-01,
  +3.04138135607636327193623495e-02,
  +3.28830166888519973540751354924e-02,
  -1.059740178506930210488320852402e-02,
};
```

The names reflect the number \(n/2\) of vanishing moments. Reversing or negating the sequence of filter coefficients leads to trivial variants that also satisfies the moment conditions.

For the filters of length \(n \geq 6\) there are solutions that are essentially different. For \(n = 6\) there is one complex solution besides Daub3:\

```cpp
-0.09556007476957763 + 0.050862777264*1
+0.081216602500826159985115891390e-01,
+8.068915093110295769444936040887e-01,
+4.598775201184937579515919241476e-01,
-1.3501102001254588663899069993e-01,
-8.544127388202666162916918177e-02,
+3.5226291885709536602740664715516e-02,
```

For \(n = 8\) there is, besides Daub4\[], an additional real solution (left), and a complex one (right):

```cpp
-0.07576571478950221 + 0.02152475910155493*1 + 0.0176769547520*1
+0.03222310060405146 + 0.050862777264*1
+0.081216602500826159985115891390e-01,
+0.09556007476957763 + 0.050862777264*1
+0.4978186763277598*1 - 0.19397617446607878 - 0.131959625451651*1
+0.8037387518061320881250751354924e-02,
+0.012789496439072861*1 - 0.021483019612*1
+0.03222310060405146 + 0.050862777264*1
+0.09921954357666393 - 0.01995817857353184 + 0.264584925288*1
+0.02152475910155493*1 + 0.0176769547520*1
-0.09921954357666393 - 0.01995817857353184 + 0.264584925288*1
+0.03222310060405146 + 0.050862777264*1
```

[fxtbook draft of 2008-August-17]
Chapter 26: Fast wavelet transforms

The numbers of solutions grows exponentially with \( n \) (the minimal polynomial of any tap value has degree \( 2^n \)). The filters given in [FXT:wavelet/daubechies.cc] are the filters for the so-called Daubechies wavelets (some closed form expressions for the filter coefficients are given in [RS]).

Filter coefficients that satisfy the wavelet and the moment conditions can be found by a Newton iteration for zeros of the function \( F : \mathbb{R}^n \to \mathbb{R}^n, F(\vec{h}) := \vec{w} \) where \( w_i = F_i(\vec{h}) = F_i(h_0, h_1, \ldots, h_5) \). For example, with \( n = 6 \) the \( F_i \) are defined by

\[
\begin{align*}
1. & \quad F[1] = h_0^2 + h_1^2 + h_2^2 + h_3^2 + h_4^2 + h_5^2 - 1 \\
2. & \quad F[2] = h_2 h_0 + h_3 h_1 + h_4 h_2 + h_5 h_3 \\
3. & \quad F[3] = h_4 h_0 + h_5 h_1 \\
4. & \quad F[4] = -h_0 + h_1 - h_2 + h_3 - h_4 + h_5 \\
5. & \quad F[5] = h_1 - 2 h_2 + 3 h_3 - 4 h_4 + 5 h_5 \\
6. & \quad F[6] = h_1 - 4 h_2 + 9 h_3 - 16 h_4 + 25 h_5
\end{align*}
\]

The derivative is given by the Jacobi matrix \( J \). It has the components \( J_{r,c} := \frac{dF_r}{dh_c} \). Its rows are

\[
\begin{align*}
1. & \quad J[1] = [2 h_0, 2 h_1, 2 h_2, 2 h_3, 2 h_4, 2 h_5] \\
2. & \quad J[2] = [h_2, h_3, h_0 + h_4, h_1 + h_5, h_2, h_3] \\
3. & \quad J[3] = [h_4, h_5, 0, 0, h_0, h_1] \\
4. & \quad J[4] = [-1, 1, -1, 1, -1, 1] \\
5. & \quad J[5] = [0, 1, -2, 3, -4, 5] \\
6. & \quad J[6] = [0, 1, -4, 9, -16, 25]
\end{align*}
\]

Now iterate (the equivalent to Newton's iteration, \( x_{k+1} := x_k - f(x_k)/f'(x_k) \))

\[
\vec{h}_{k+1} := \vec{h}_k - J^{-1}(\vec{h}_k) F(\vec{h}_k)
\]

The computations have to be carried out with a rather great precision to avoid loss of accuracy.
Part IV

Fast arithmetic
Chapter 27

Fast multiplication and exponentiation

The usual scheme for multiplication needs proportional $N^2$ operations for the multiplication of two $N$-digit numbers. This chapter describes multiplication algorithms that are asymptotically better than this, the Karatsuba algorithm, the Toom-Cook algorithms, and multiplication via FFTs. In addition, the left-to-right and right-to-left schemes for binary exponentiation are described.

27.1 Asymptotics of algorithms

An important feature of an algorithm is the number of operations that must be performed for the completion of a task of a certain size. For high precision computations one will take $N$ as the length of the numbers counted in decimal digits or bits. For computations with square matrices one may take for $N$ the number of rows, or the number of entries in the matrix. An operation is typically a (machine word) multiplication plus an addition, one could also simply count machine instructions.

We now consider the computational cost of a given algorithm. An algorithm is said to have some asymptotics $f(N)$ if it needs proportional $f(N)$ operations for a task of size $N$. We express proportionality with $f(N)$ as $\sim f(N)$. Some examples:

- Addition of two $N$-digit numbers needs $\sim N$ operations (here: machine word additions).
- Ordinary multiplication of two $N$-digit numbers needs $\sim N^2$ operations.
- Counting elements equal to a given value in an unsorted array of length $N$ costs $\sim N$ operations. The operations are reads from memory and comparisons.
- Deciding whether a sorted array of length $N$ contains a given value costs $\sim \log(N)$ operations when binary search is used.
- Computing the Fourier transform of a length-$N$ sequence by definition, that is, as $N$ sums each of length $N$, costs $\sim N^2$.
- The FFT algorithms compute the Fourier transform of a length-$N$ sequence in $\sim N \log(N)$ operations.
- Matrix multiplication (of two $N \times N$ matrices by the obvious algorithm) is $\sim N^3$ ($N^2$ sums, each of $N$ products).
- When the problem size $M$ for matrix multiplication is taken to be the number of elements of the matrix ($M = N^2$), then the cost is ‘only’ $\sim M^{3/2}$.
• Deciding whether a \( N \)-digit binary number is even or odd costs 1 operation (lookup whether the lowest bit is zero). The cost is independent of the problem size.

We have simplified the considerations by assuming that for a given algorithm \( \sim f(N) \) there is a constant \( C \) so that

\[
\lim_{N \to \infty} \frac{\text{work}(N)}{f(N)} = C
\]  

(27.1-1)

where \( \text{work}(N) \) is the actual cost for problem size \( N \). The approximate cost of an algorithm is expressed by the symbol \( \sim C \cdot f(N) \). The constant \( C \) is often referred to as the hidden constant.

The algorithm with the ‘best’ asymptotics wins for some, possibly huge, problem size \( N \). For smaller \( N \) another algorithm will be superior. For the exact break-even point the hidden constants are, of course, important. For example, let the algorithm \texttt{mult1} take \( \approx 1.0 \cdot N^2 \) operations \((C = 1.0)\), \texttt{mult2} take \( \approx 8.0 \cdot N \log_2(N) \) operations \((C = 8.0)\). Then for \( N < 64 \) the algorithm \texttt{mult1} is faster, while for \( N > 64 \) the algorithm \texttt{mult2} is faster. Completely different algorithms may be optimal for the same task at different problem sizes. The hidden constants can be so large that the asymptotically better algorithm is never practical for any feasible problem size.

With many algorithms it is only possible to give lower and upper (asymptotic) bounds, especially if the program flow depends on the processed data. Upper bounds are usually denoted with \( O(f(N)) \), for example, FFT multiplication is \( O(N \log(N)) \) and computations that take a constant number of operations independent of the problem size are \( O(1) \).

The space requirements in all methods given is a constant times the problem size \( N \). In-place algorithms (like the FFT) use space \( \sim N \), the remaining cases (like radix sort) typically need \( \sim 2N \). Note that it is possible to construct algorithms where the computational cost alone is not of much value: integer factorization (up to a certain maximal value) costs just one lookup if a precomputed table of factorizations is stored in an (insanely) big array.

For a more fine-grained approach to measuring the costs of algorithms see [95].

### 27.2 Splitting schemes for multiplication

Ordinary multiplication is \( \sim N^2 \). Assuming the hidden constant equals one the computation of the product of two million-digit numbers would require \( \approx 10^{12} \) operations. On a machine that does 1 billion operations per second the multiplication would need 1000 seconds. The following schemes leads to algorithms with superior asymptotics.

#### 27.2.1 2-way splitting: the Karatsuba algorithm

The following algorithm is due to A. Karatsuba and Y. Ofman, it was given 1963 in [163].

Split the numbers \( A \) and \( B \) (assumed to have approximately the same length) into two pieces

\[
\begin{align*}
A &= x A_1 + A_0 \\
B &= x B_1 + B_0
\end{align*}
\]  

(27.2-1)

where \( x \) is a power of the radix (for decimal numbers the radix is 10) close to the half length of \( A \) and \( B \). The usual multiplication scheme needs 4 multiplications with half precision for one multiplication with full precision:

\[
A B = A_0 \cdot B_0 + x (A_0 \cdot B_1 + B_0 \cdot A_1) + x^2 A_1 \cdot B_1
\]  

(27.2-2)
27.2: Splitting schemes for multiplication

Only the multiplications \( A_i \cdot B_j \) need to be considered. The multiplications by \( x \), a power of the radix, are only shifts. If we use the relation

\[
AB = (1 + x) A_0 \cdot B_0 + x (A_1 - A_0) \cdot (B_0 - B_1) + (x + x^2) A_1 \cdot B_1
\]

we need 3 multiplications with half precision for one multiplication with full precision. Applying the scheme recursively until the numbers to multiply are of machine size we obtain an algorithm whose asymptotic cost is \( \sim N^{\log_2(3)} \approx N^{1.585} \). An alternative form of relation 27.2-3 is

\[
AB = (1 - x) A_0 \cdot B_0 + x (A_1 + A_0) \cdot (B_0 + B_1) + (x^2 - x) A_1 \cdot B_1
\]

For squaring use either of the following schemes

\[
A^2 = (1 + x) A_0^2 - x (A_1 - A_0)^2 + (x + x^2) A_1^2
\]  
\[
A^2 = (1 - x) A_0^2 + x (A_1 + A_0)^2 + (x^2 - x) A_1^2
\]

We compute \( 8231^2 = 67749361 \) with the first relation (27.2-5a):

\[
8231^2 = (100*82+31)^2 = (1+100)*31^2 - 100*(82-31)^2 + (100+100^2)*82^2
\]

Assume that the hidden constant equals 2 as there is more bookkeeping overhead than with the usual algorithm. Computing the product of two million-digit numbers would require \( \approx 2 \cdot (10^6)^{1.585} \approx 6.47 \cdot 10^9 \) operations, taking about 6.5 seconds on our computer.

The Karatsuba scheme for polynomial multiplication is given in section 38.2 on page 824.

27.2.2 3-way splitting

One can extend the above idea by splitting \( U \) and \( V \) into more than two pieces each, the resulting method is called the **Toom-Cook algorithm** (the method is called **Toom algorithm** in [94], and **Cook-Toom algorithm** in [2]).

27.2.2.1 Zimmermann’s 3-way multiplication

\[
A = a_2 \cdot x^2 + a_1 \cdot x + a_0
B = b_2 \cdot x^2 + b_1 \cdot x + b_0
\]

\[
S0 = a_0 \cdot b_0
S1 = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0)
S2 = (4 \cdot a_2 + 2 \cdot a_1 + a_0) \cdot (4 \cdot b_2 + 2 \cdot b_1 + b_0)
S3 = (a_2 - a_1 + a_0) \cdot (b_2 - b_1 + b_0)
S4 = a_2 \cdot b_2
\]

\[
T1 = 2 \cdot S3 + S2
T1 /= 3 \quad \text{// division by 3}
T1 *= S0
T1 /= 2
T1 -= 2 \cdot S4
T2 = (S1 + S3) / 2
S1 = T1
S2 = T2 - S0 - S4
S3 = T1 - T2
P = S4 * x^4 + S3 * x^3 + S2 * x^2 + S1 * x + S0
P -= A * B \quad \text{// == zero}
\]

**Figure 27.2-A:** Implementation of Zimmermann’s 3-way multiplication scheme in pari/gp.
The good scheme for 3-way splitting is due to Paul Zimmermann. We compute the product $C = A \cdot B$ of two numbers, $A$ and $B$

\[
A = a_2 x^2 + a_1 x + a_0 \quad (27.2-6a)
\]

\[
B = b_2 x^2 + b_1 x + b_0 \quad (27.2-6b)
\]

\[
C = A \cdot B = c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 \quad (27.2-6c)
\]

by the following scheme (taken from [87]): set

\[
S_0 := a_0 \cdot b_0 \quad (27.2-6d)
\]

\[
S_1 := (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0) \quad (27.2-6e)
\]

\[
S_2 := (4a_2 + 2a_1 + a_0) \cdot (4b_2 + 2b_1 + b_0) \quad (27.2-6f)
\]

\[
S_3 := (a_2 - a_1 + a_0) \cdot (b_2 - b_1 + b_0) \quad (27.2-6g)
\]

\[
S_4 := a_2 \cdot b_2 \quad (27.2-6h)
\]

This costs 5 multiplications of length $N/3$. We already have found $c_0 = S_0$ and $c_4 = S_4$. We determine $c_1$, $c_2$, and $c_3$ by the following assignments (in the given order):

\[
T_1 := 2S_3 + S_2 \quad (= 18c_4 + 6c_3 + 6c_2 + 3c_0) \quad (27.2-6i)
\]

\[
T_1 := T_1/3 \quad (= 6c_4 + 2c_3 + 2c_2 + c_0) \quad \text{exact division by 3} \quad (27.2-6j)
\]

\[
T_1 := T_1 + S_0 \quad (= 6c_4 + 2c_3 + 2c_2 + 2c_0) \quad (27.2-6k)
\]

\[
T_1 := T_1/2 \quad (= 3c_4 + c_3 + c_2 + c_0) \quad (27.2-6l)
\]

\[
T_1 := T_1 - 2S_4 \quad (= c_4 + c_3 + c_2 + c_0) \quad (27.2-6m)
\]

\[
T_2 := (S_1 + S_3)/2 \quad (= c_4 + c_2 + c_0) \quad (27.2-6n)
\]

\[
S_1 := S_1 - T_1 \quad (= c_1) \quad \text{wrong in cited paper} \quad (27.2-6o)
\]

\[
S_2 := T_2 - S_0 - S_4 \quad (= c_2) \quad (27.2-6p)
\]

\[
S_3 := T_1 - T_2 \quad (= c_3) \quad (27.2-6q)
\]

Now we have

\[
C = A \cdot B = S_4 x^4 + S_3 x^3 + S_2 x^2 + S_1 x + S_0 \quad (27.2-6r)
\]

The complexity of recursive multiplication based on this splitting scheme is $N^{\log_3(5)} \approx N^{1.465}$. Assume that the hidden constant again equals 2. Then the computation of the product of two million-digit numbers would require $\approx 2 \cdot (10^6)^{1.465} \approx 1.23 \cdot 10^9$ operations, taking about 1.2 seconds on our computer.

Note the division by 3 in relation [27.2-6]. A division by a constant (that is not a power of two) cannot be avoided in 4-way splitting schemes for multiplication for $n \geq 3$. There are squaring schemes that do not involve such divisions.

### 27.2.2.2 3-way multiplication by Bodrato and Zanoni

An alternative algorithm for 3-way splitting is suggested in [51]: setup $S_0$, $S_1$, $\ldots$, $S_4$ as in relations [27.2-6a], [27.2-6b], then compute, in the given order,

\[
S_2 := (S_2 - S_3)/3 \quad (= 5c_4 + 3c_3 + c_2 + c_1) \quad \text{exact division by 3} \quad (27.2-7a)
\]

\[
S_3 := (S_1 - S_3)/2 \quad (= c_4 + c_1) \quad (27.2-7b)
\]

\[
S_1 := S_1 - S_0 \quad (= c_4 + c_3 + c_2 + c_1) \quad (27.2-7c)
\]

\[
S_2 := (S_2 - S_1)/2 \quad (= 2c_4 + c_3) \quad (27.2-7d)
\]

\[
S_1 := S_1 - S_3 - S_4 \quad (= c_2) \quad (27.2-7e)
\]

\[
S_2 := S_2 - 2S_4 \quad (= c_3) \quad (27.2-7f)
\]

\[
S_3 := S_3 - S_2 \quad (= c_1) \quad (27.2-7g)
\]
27.2: Splitting schemes for multiplication

\[ A = a_2 \times x^2 + a_1 \times x + a_0 \]
\[ B = b_2 \times x^2 + b_1 \times x + b_0 \]
\[ S_0 = a_0 \times b_0 \]
\[ S_1 = (a_2 + a_1 + a_0) \times (b_2 + b_1 + b_0) \]
\[ S_2 = (4 \times a_2 + 2 \times a_1 + a_0) \times (4 \times b_2 + 2 \times b_1 + b_0) \]
\[ S_3 = (a_2 - a_1 + a_0) \times (b_2 - b_1 + b_0) \]
\[ S_4 = a_2 \times b_2 \]
\[ S_2 = (S_2 - S_3) / 3 \quad \text{\small division by 3} \]
\[ S_3 = (S_1 - S_3) / 2 \]
\[ S_1 = S_1 - S_0 \]
\[ S_2 = (S_2 - S_1) / 2 \]
\[ S_1 = S_1 - S_3 - S_4 \]
\[ S_2 = S_2 - 2 \times S_4 \]
\[ S_3 = S_3 - S_2 \]

\[ P = S_4 \times x^4 + S_2 \times x^3 + S_1 \times x^2 + S_3 \times x + S_0 \]

\[ P = A \times B \quad \text{\small \color{red}== zero} \]

**Figure 27.2-B:** Implementation of the 3-way multiplication scheme of Bodrato and Zanoni.

Now we have (note the order of the coefficients \( S_i \))

\[ C = A \times B = S_4 \times x^4 + S_2 \times x^3 + S_1 \times x^2 + S_3 \times x + S_0 \]  
(27.2-7h)

The scheme requires only one multiplication by two, Zimmermann’s scheme involves two.

### 27.2.2.3 3-way squaring

The following scheme is taken from \( \text{[8]} \). To compute the square \( C = A^2 \) of a number \( A \)

\[ A = a_2 \times x^2 + a_1 \times x + a_0 \]  
(27.2-8a)
\[ C = A^2 = S_4 \times x^4 + S_3 \times x^3 + S_2 \times x^2 + S_1 \times x + S_0 \]  
(27.2-8b)

set

\[ S_0 := a_0^2 \]  
(27.2-8c)
\[ S_1 := (a_2 + a_1 + a_0)^2 \]  
(27.2-8d)
\[ S_2 := (a_2 - a_1 + a_0)^2 \]  
(27.2-8e)
\[ S_3 := 2 \times a_1 \times a_2 \]  
(27.2-8f)
\[ S_4 := a_2^2 \]  
(27.2-8g)

This costs 4 squarings and 1 multiplication of length \( N/3 \). The quantities \( S_0, S_3, \) and \( S_4 \) are already correct. Determine \( S_1 \) and \( S_2 \) via

\[ T_1 := (S_1 + S_2) / 2 \]  
(27.2-8i)
\[ S_1 := S_1 - T_1 - S_3 \]  
(27.2-8j)
\[ S_2 := T_1 - S_4 - S_0 \]  
(27.2-8k)

### 27.2.3 4-way splitting

#### 27.2.3.1 4-way multiplication

An elegant and clean scheme for 4-way splitting of a multiplication is given by Bodrato and Zanoni in \( \text{[52]} \). A pari/gp implementation is shown in figure 27.2-C. The algorithm has asymptotics \( \sim O(n^{\log_4(7)}) \approx O(n^{1.403}) \). In general, an \( s \)-way splitting scheme will be \( \sim O(n^{\log_s(2s+1)}) \).
Chapter 27: Fast multiplication and exponentiation

A = \( a_3 x^3 + a_2 x^2 + a_1 x + a_0 \)
B = \( b_3 x^3 + b_2 x^2 + b_1 x + b_0 \)

S1 = \( a_3 b_3 \)
S2 = \((8 a_3 + 4 a_2 + 2 a_1 + a_0) \cdot (8 b_3 + 4 b_2 + 2 b_1 + b_0)\)
S3 = \((+a_3 + a_2 + a_0) \cdot (+b_3 + b_2 + b_1 + b_0)\)
S4 = \((-a_3 + a_2 + a_0) \cdot (-b_3 + b_2 + b_1 + b_0)\)
S5 = \((8 a_0 + 4 a_1 + 2 a_2 + a_3) \cdot (8 b_0 + 4 b_1 + 2 b_2 + b_3)\)
S6 = \((-8 a_0 + 4 a_1 - 2 a_2 + a_3) \cdot (-8 b_0 + 4 b_1 - 2 b_2 + b_3)\)
S7 = \( a_0 b_0 \)

S2 += S5
S4 -= S3
S6 -= S5
S4 /= 2
S5 -= S1
S5 -= (64 * S7)
S5 *= 2; S5 += S6
S2 += (65 * S3)
S3 = S1
S6 = S6 \//
S2 += (45 * S3)
S5 = (8 * S3)
S5 /= 24 \// division by 24
S6 = S2
S2 = (16 * S4)
S3 = S5
S4 = S3
S6 += (30 * S2)
S6 = 60 \// division by 60
S2 = S6

P = S1 * x^6 + S2 * x^5 + S3 * x^4 + S4 * x^3 + S5 * x^2 + S6 * x + S7;
P - A * B \// == zero

Figure 27.2-C: Implementation of the 4-way multiplication scheme in pari/gp.

27.2.3.2 4-way squaring

The following scheme is taken from [87]

\[
A = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \quad \quad (27.2-9a)
\]
\[
C = A^2 = c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 \quad \quad (27.2-9b)
\]

Set

\[
S_1 := a_0^2 \quad \quad (27.2-9c)
\]
\[
S_2 := 2 a_0 \cdot a_1 \quad \quad (27.2-9d)
\]
\[
S_3 := (a_0 + a_1 - a_2 - a_3) \cdot (a_0 - a_1 - a_2 + a_3) \quad \quad (27.2-9e)
\]
\[
S_4 := (a_0 + a_1 + a_2 + a_3)^2 \quad \quad (27.2-9f)
\]
\[
S_5 := 2 (a_0 - a_2) \cdot (a_1 - a_3) \quad \quad (27.2-9g)
\]
\[
S_6 := 2 a_3 \cdot a_2 \quad \quad (27.2-9h)
\]
\[
S_7 := a_3^2 \quad \quad (27.2-9i)
\]
27.2: Splitting schemes for multiplication

\[
A = a_3 x^3 + a_2 x^2 + a_1 x + a_0
\]

\[
S_1 = a_0^2
\]
\[
S_2 = 2 * a_0 * a_1
\]
\[
S_3 = (a_0 + a_1 - a_2 - a_3) * (a_0 - a_1 - a_2 + a_3)
\]
\[
S_4 = (a_0 + a_1 + a_2 + a_3)^2
\]
\[
S_5 = 2 * (a_0 - a_2) * (a_1 - a_3)
\]
\[
S_7 = a_3^2
\]

\[
T_1 = S_3 + S_4
\]
\[
T_2 = (T_1 + S_5)/2
\]
\[
T_3 = S_2 + S_6
\]
\[
T_4 = T_2 - T_3
\]
\[
T_5 = T_3 - S_5
\]
\[
T_6 = T_4 - S_3
\]
\[
T_7 = T_4 - S_1
\]
\[
T_8 = T_6 - S_7
\]

\[
P = S_7 * x^6 + S_6 * x^5 + T_7 * x^4 + T_5 * x^3 + T_8 * x^2 + S_2 * x + S_1
\]

\[
P - A^2 \Rightarrow \text{zero}
\]

Figure 27.2-D: Implementation of the 4-way squaring scheme in pari/gp.

Then set, in the given order,

\[
T_1 := S_3 + S_4
\]
\[
T_2 := (T_1 + S_5)/2
\]
\[
T_3 := S_2 + S_6
\]
\[
T_4 := T_2 - T_3
\]
\[
T_5 := T_3 - S_5
\]
\[
T_6 := T_4 - S_3
\]
\[
T_7 := T_4 - S_1
\]
\[
T_8 := T_6 - S_7
\]

The square then equals

\[
C = S_7 x^6 + S_6 x^5 + T_7 x^4 + T_5 x^3 + T_8 x^2 + S_2 x + S_1
\]

27.2.4 5-way splitting

27.2.4.1 5-way multiplication

The scheme for 5-way splitting of a multiplication shown in figure 27.2-E is given in [52]. As for the 4-way multiplication scheme, no temporaries are used.

27.2.4.2 5-way squaring

We describe the 5-way squaring scheme given in [51]. Let

\[
A = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0
\]
\[
C = A^2 = c_8 x^8 + c_7 x^7 + c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0
\]

[fxtbook draft of 2008-August-17]
Chapter 27: Fast multiplication and exponentiation

Figure 27.2-E: Implementation of the 5-way multiplication scheme in pari/gp.
27.2: Splitting schemes for multiplication

\[ A = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]
\[ S_1 = a_0^2 \]
\[ S_2 = a_4^2 \]
\[ S_3 = (a_0 + a_1 + a_2 + a_3 + a_4)^2 \]
\[ S_4 = (a_0 - a_1 + a_2 - a_3 + a_4)^2 \]
\[ S_5 = 2 \cdot (a_0 - a_2 + a_4) \cdot (a_1 - a_3) \]
\[ S_6 = (a_0 + a_1 - a_2 - a_3 + a_4) \cdot (a_0 - a_1 - a_2 + a_3 + a_4) \]
\[ S_7 = (a_1 + a_2 - a_4) \cdot (a_1 - a_2 - a_4 + 2(a_0 - a_3)) \]
\[ S_8 = 2a_0 \cdot a_1 \]
\[ S_9 = 2a_3 \cdot a_4 \]

Further set, in the order given,

\[ S_4 = (S_4 + S_3)/2 \quad (= c_0 + c_2 + c_4 + c_6 + c_8) \]
\[ S_3 = S_3 - S_4 \quad (= c_1 + c_3 + c_5 + c_7) \]
\[ S_6 = (S_6 + S_4)/2 \quad (= c_0 + c_4 + c_8) \]
\[ S_5 = (-S_5 + S_3)/2 \quad (= c_3 + c_7) \]
\[ S_4 = S_4 - S_6 \quad (= c_2 + c_6) \]
\[ S_3 = S_3 - S_5 - S_6 \quad (= c_5) \]
\[ S_6 = S_6 - S_2 - S_1 \quad (= c_4) \]
\[ S_5 = S_5 - S_9 \quad (= c_3) \]
\[ S_7 = S_7 - S_2 - S_8 - S_9 + S_6 + S_3 \quad (= c_2) \]
\[ S_4 = S_4 - S_7 \quad (= c_6) \]

Now we have (note the order of the coefficients \( S_i \))

\[ C = A^2 = S_2 x^8 + S_9 x^7 + S_4 x^6 + S_3 x^5 + S_6 x^4 + S_5 x^3 + S_7 x^2 + S_8 x + S_1 \]
Chapter 27: Fast multiplication and exponentiation

A = a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0
S1 = a0^2
S2 = a4^2
[...S9, as before]
T1 = S1 + 2*S2 - S7 + 2*S8 + S9
T2 = S3 - S4
T3 = 2*S5
T4 = T2 + T3
T5 = T2 - T3
T6 = T4/4
T7 = T5/4 - S9
T8 = T1 - T6 - S6
T9 = T6 - S8
T10 = S3 + S6
T11 = (T10 + S4 + S6) / 4
T12 = T11 - S1 - S2
T13 = (T10 + S5) / 2
T14 = T13 - T1
P = S2*x^8 + S9*x^7 + T8*x^6 + T9*x^5 + T12*x^4 + T7*x^3 + T14*x^2 + S8*x + S1
P - A^2 \equiv zero

Figure 27.2-G: Implementation of the alternative 5-way squaring scheme in pari/gp. Definition of S1, ..., S9 as in figure 27.2-F.

27.2.4.3 Alternative 5-way squaring scheme

The following scheme is taken from [87], we correct some errors in the paper. Setup S1, ..., S9 as given by relations 27.2-12a ... 27.2-12i then compute, in the given order,

\[ T_1 := S_1 + 2S_2 - S_7 + 2S_8 + S_9 \] (27.2-14a)
\[ T_2 := S_3 - S_4 \] (27.2-14b)
\[ T_3 := 2S_5 \] (27.2-14c)
\[ T_4 := T_2 + T_3 \] (27.2-14d)
\[ T_5 := T_2 - T_3 \] (27.2-14e)
\[ T_6 := T_4/4 \] (27.2-14f)
\[ T_7 := T_5/4 - S_9 \] (27.2-14g)
\[ T_8 := T_1 - T_6 - S_6 \] (27.2-14h)
\[ T_9 := T_6 - S_8 \] (27.2-14i)
\[ T_{10} := S_3 + S_6 \] (27.2-14j)
\[ T_{11} := (T_{10} + S_4 + S_6) / 4 \] (27.2-14k)
\[ T_{12} := T_{11} - S_1 - S_2 \quad \text{wrong in cited paper} \] (27.2-14l)
\[ T_{13} := (T_{10} + S_5) / 2 \] (27.2-14m)
\[ T_{14} := T_{13} - T_1 \] (27.2-14n)

We have (note that the coefficients for x^4 and x^2 are wrong in the cited paper):

\[ C = S_2 x^8 + S_9 x^7 + T_8 x^6 + T_9 x^5 + T_{12} x^4 + T_7 x^3 + T_{14} x^2 + S_8 x + S_1 \] (27.2-14o)

27.3 Fast multiplication via FFT

Multiplication of two numbers is essentially a convolution of the sequences of their digits. The (linear) convolution of the two sequences \( a_k, b_k, k = 0 \ldots N - 1 \) is defined as the sequence \( c \) where

\[ c_k := \sum_{i,j=0; \ i+j=k}^{N-1} a_i b_j \quad k = 0 \ldots 2N - 2 \] (27.3-1)
A \((n\text{-digit})\) number written in radix \(R\) as

\[ a_{n-1} \ a_{n-2} \ldots a_2 \ a_1 \ a_0 \]  

(27.3-2)

denotes a quantity of

\[
\sum_{i=0}^{n-1} a_i \cdot R^i = a_{n-1} \cdot R^{n-1} + a_{n-2} \cdot R^{n-2} + \ldots + a_1 \cdot R + a_0
\]  

(27.3-3)

This means, the digits can be identified with coefficients of a polynomial in \(R\). For example, with decimal numbers one has \(R = 10\), and the number 578 equals \(5 \cdot 10^2 + 7 \cdot 10^1 + 8 \cdot 10^0\). The product of two numbers is almost the polynomial product

\[
\sum_{k=0}^{2N-2} c_k R^k := \sum_{i=0}^{N-1} a_i R^i \cdot \sum_{j=0}^{N-1} b_j R^j
\]  

(27.3-4)

The \(c_k\) are found by comparing coefficients. One easily checks that the \(c_k\) must satisfy the convolution equation 27.3-1. As the \(c_k\) can be greater than ‘nine’ (that is, \(R - 1\)), the result has to be ‘fixed’ using carry operations: Go from right to left, replace \(c_k\) by \(c_k' = c_k \mod R\) and add \(\lfloor (c_k - c_k')/R \rfloor\) to its left neighbor.

An example: usually one would multiply the numbers 82 and 34 as follows:

<table>
<thead>
<tr>
<th>(82\times34)</th>
<th>3</th>
<th>32</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

\[ = 2 \quad 7 \quad 8 \quad 8 \]

We have seen that the carries can be delayed to the end of the computation:

<table>
<thead>
<tr>
<th>(82\times34)</th>
<th>32</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

\[ = 2 \quad 7 \quad 8 \quad 8 \]

\[
\cdots \text{which is really polynomial multiplication (which in turn is a convolution of the coefficients):}
\]

\[
(8x+2) \times (3x+4)
\]

<table>
<thead>
<tr>
<th>(24x^2)</th>
<th>32x</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x</td>
<td>+38x</td>
<td>+8</td>
</tr>
</tbody>
</table>

\[ = 24x^2 \quad +38x \quad +8 \]

The value of the polynomial \(24x^2 + 38x + 8\) for \(x = 10\) is 2788.

Convolution can be done efficiently using the Fast Fourier Transform (FFT): convolution is a simple (element-wise) multiplication in Fourier space. The FFT itself takes \(\sim N \cdot \log N\) operations. Instead of the direct convolution (\(\sim N^2\)) one proceeds as follows:

- Compute the FFTs of both factors.
- Multiply the transformed sequences element-wise.
- Compute inverse transform of the product.

To understand why this actually works note that (1) the multiplication of two polynomials can be achieved by the (more complicated) scheme:

- evaluate both polynomials at sufficiently many points (at least one more point than the degree of the product polynomial \(c\): \(\deg c = \deg a + \deg b\))
• element-wise multiply the values found
• find the polynomial corresponding to those (product-)values

and (2) that the FFT is an algorithm for the parallel evaluation of a given polynomial at many points, namely the roots of unity. (3) the inverse FFT is an algorithm to find (the coefficients of) a polynomial whose values are given at the roots of unity.

You might be surprised if you always thought of the FFT as an algorithm for the ‘decomposition into frequencies’. There is no problem with either of these notions.

Re-launching our example \((82 \cdot 34 = 2788)\), we use the fourth roots of unity \(\pm 1\) and \(\pm i\):

\[
\begin{array}{ccc}
a = (8x + 2) & \times & b = (3x + 4) \\
+1 & +10 & +7 \\
+i & +8i + 2 & +3i + 4 \\
-1 & -6 & +1 \\
-i & -8i + 2 & -3i + 4 \\
\end{array}
\]

\[
c = (24x^2 + 38x + 8)
\]

This table has to be read as follows: first the given polynomials \(a\) and \(b\) are evaluated at the points given in the left column, thereby the columns below \(a\) and \(b\) are filled. Then the values are multiplied to fill the column below \(c\), giving the values of \(c\) at the points. Finally, the actual polynomial \(c\) is found from those values, resulting in the lower right entry. You may find it instructive to verify that a 4-point FFT really evaluates \(a\), \(b\) by transforming the sequences 0, 0, 8, 2 and 0, 0, 3, 4 by hand. The backward transform of 70, 38\(i\)−16, −6, −38\(i\)−16 should produce the final result given for \(c\).

The operation count is dominated by that of the FFTs (the element-wise multiplication is of course \(\sim N\)), so the whole fast convolution algorithm takes \(\sim N \cdot \log N\) operations. The following carry operation is also \(\sim N\) and can therefore be neglected when counting operations.

Assume the hidden constant equals five. Multiplying our million-digit numbers will need about

\[
5 \cdot 10^6 \log_2(10^6) \approx 5 \cdot 10^6 \cdot 20 = 10^8 = 0.1 \cdot 10^9
\]

operations, taking approximately a tenth second on our computer.

Strictly speaking \(N \cdot \log N\) is not really the truth: it has to be \(N \cdot \log N \cdot \log \log N\). This is because the sums in the convolutions have to be represented as exact integers. The biggest term \(C\) that can possibly occur is approximately \(NR^2\) for a number with \(N\) digits (see next section). Therefore, working with some fixed radix \(R\) one has to compute the FFTs with \(\log N\) bits precision, leading to an operation count of \(N \cdot \log N \cdot \log \log N\). The slightly better \(N \cdot \log N \cdot \log \log N\) can be obtained by recursive use of FFT multiplies. For realistic applications (where the sums in the convolution all fit into the machine type floating point numbers) it is safe to think of FFT multiplication being proportional \(N \cdot \log N\).

Several alternative multiplication algorithms are given in [172, ch.4.3.3]. See [180] on how far the idea “polynomials for numbers” can be carried and where it fails.

### 27.4 Radix/precision considerations with FFT multiplication

This section describes the dependencies between the radix of the number and the achievable precision when using FFT multiplication.

We use (unsigned) 16-bit words for the digits. Thereby the radix of the numbers can be in the range 2, 3, ..., 65536 (= 2\(16\)). When working in base ten one will actually use ‘super-digits’ of base 10,000, the largest power of ten that fits into a 16-bit word. These super-digits are called LIMBs in hfloat.

With very large precision one cannot always use the greatest power of the desired base. This is due to the fact that the components of the convolution must be representable as integer numbers with the data type
used for the FFTs: The cumulative sums $c_k$ have to be represented precisely enough to distinguish every (integer) quantity from the next bigger (or smaller) value. The highest possible value for a $c_k$ will appear in the middle of the product and when multiplicand and multiplier consist of ‘nines’ (that is $R - 1$) only. For radix $R$ and a precision of $N$ LIMBs let the maximal possible value be $C$, then

$$C = N(R - 1)^2$$  \hspace{1cm} (27.4-1)

Note that with FFT based convolution the absolute value of the central term can in fact equal $|C| = N^2(R - 1)^2$. But there is no need to distinguish that many integers. After dividing by $N$ we are back at relation 27.4-1.

The number of bits to represent $C$ exactly is the integer greater or equal to

$$\log_2(N(R - 1)^2) = \log_2 N + 2 \log_2(R - 1)$$  \hspace{1cm} (27.4-2)

Due to numerical errors there must be a few more bits for safety. If computations are made using double-precision floating point numbers (C-type double) one typically has a mantissa of 53 bits (only the 52 least significant bits are physically present, the most significant bit is implied to be always set) then we need to have

$$M \geq \log_2 N + 2 \log_2(R - 1) + S$$  \hspace{1cm} (27.4-3)

where $M$ := mantissa-bits and $S$ := safety-bits. Using $\log_2(R - 1) < \log_2(R)$:

$$N_{\text{max}}(R) = 2^{M-S-2 \log_2(R)}$$  \hspace{1cm} (27.4-4)

Suppose we have $M = 53$ mantissa-bits and require $S = 3$ safety-bits. With base 2 numbers one could use radix $R = 2^{16}$ for precisions up to a length of $N_{\text{max}} = 2^{53-3-2\cdot16} = 256k$ LIMBs. Corresponding are 4096 kilo bits and = 1024 kilo hex digits. For greater lengths smaller radices have to be used according to the following table (extra horizontal line at the 16 bit limit for LIMBs):

<table>
<thead>
<tr>
<th>Radix $R$</th>
<th>max # LIMBs</th>
<th>max # hex digits</th>
<th>max # bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>1048,576 $k$</td>
<td>2621,440 $k$</td>
<td>10240 $M$</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>262,144 $k$</td>
<td>720,896 $k$</td>
<td>2816 $M$</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>65,536 $k$</td>
<td>196,608 $k$</td>
<td>768 $M$</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>16384 $k$</td>
<td>53,248 $k$</td>
<td>208 $M$</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>4096 $k$</td>
<td>14,336 $k$</td>
<td>56 $M$</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>32768 $k$</td>
<td>3840 $k$</td>
<td>15 $M$</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>65536 $k$</td>
<td>1024 $k$</td>
<td>4 $M$</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>128 $k$</td>
<td>272 $k$</td>
<td>1062 $k$</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>256 $k$</td>
<td>72 $k$</td>
<td>281 $k$</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>512 $k$</td>
<td>19 $k$</td>
<td>74 $k$</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>1 $M$</td>
<td>5 $k$</td>
<td>19 $k$</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>2 $M$</td>
<td>256</td>
<td>5120</td>
</tr>
</tbody>
</table>

For decimal numbers:

<table>
<thead>
<tr>
<th>Radix $R$</th>
<th>max # LIMBs</th>
<th>max # digits</th>
<th>max # bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^2$</td>
<td>110 $G$</td>
<td>220 $G$</td>
<td>730 $G$</td>
</tr>
<tr>
<td>$10^3$</td>
<td>1100 $M$</td>
<td>3300 $M$</td>
<td>11 $G$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>11 $M$</td>
<td>44 $M$</td>
<td>146 $M$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>110 $k$</td>
<td>550 $k$</td>
<td>1826 $k$</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1 $k$</td>
<td>6,597</td>
<td>22 $k$</td>
</tr>
<tr>
<td>$10^7$</td>
<td>11</td>
<td>77</td>
<td>255</td>
</tr>
</tbody>
</table>

Summarizing:

- For decimal digits and precisions up to 11 million LIMBs use radix 10,000. (corresponding to more than 44 million decimal digits), for even greater precisions choose radix 1,000.
- For hexadecimal digits and precisions up to 256,000 LIMBs use radix 65,536 (corresponding to more than 1 million hexadecimal digits), for even greater precisions choose radix 4,096.
27.5 The sum-of-digits test

With high-precision calculations it is mandatory to add a sanity check to the multiplication routines. That is, false results due to loss of accuracy should (with high probability) be detected via the *sum-of-digits test*:

When computing the product \( c = a \cdot b \) with radix-\( R \) numbers then compute the values (‘sums of digits’) \( s_a = a \mod (R-1) \) and \( s_b = b \mod (R-1) \) after the multiplication compute \( s_c = c \mod (R-1) \) and compare \( s_c \) to \( s_m = s_a \cdot s_b \mod (R-1) \). If \( s_c \neq s_m \) then an error has occurred in the computation of \( c \).

The sum-of-digits function \( s_a \) can for a radix-\( R \), length-\( n \) number \( a \) be computed as [FXT: *mult/auxil.cc*):

```cpp
1 ulong
2 sum_of_digits(const LIMB *a, ulong n, ulong nine, ulong s)
3 {
4     for (ulong k=0; k<n; ++k) s += a[k];
5     s %= nine;
6     return s;
7 }
```

where the variable \( nine \) has to be set to \( R-1 \) and \( s \) to zero.

The computation of \( s_m = s_a \cdot s_b \) is done in

```cpp
1 ulong
2 mult_sum_of_digits(const LIMB *a, ulong an,
3     const LIMB *b, ulong bn,
4     ulong nine)
5 {
6     ulong qsa = sum_of_digits(a, an, nine, 0);
7     ulong qsb = sum_of_digits(b, bn, nine, 0);
8     ulong qsm = (qsa*qsb) % nine;
9     return qsm;
10 }
```

The checks in multiplication routine [FXT: *mult/fxtmultiply.cc*] can be outlined as:

```cpp
1 fxt_multiply(const LIMB *a, ulong an,
2     const LIMB *b, ulong bn,
3     LIMB *c, ulong cn,
4     uint rx)
5 {
6     const ulong nine = rx-1;
7     ulong qsm=0, qsp=0;
8     qsm = mult_sum_of_digits(a, an, b, bn, nine);
9     // Multiply: c=a*b
10     // If carrying through c gives an additional (leading) digit,
11     // then set cy to that value, else set cy=0.
12     if ( (qsm!=qsp) ) { /* FAILED */ }
13 }
```

If we assume that a failed multiplication produces ‘random’ digits in \( c \) then the probability that a failed multiplication goes unnoticed equals \( 1/R \).

Omitting the sum-of-digits test is *not* an option: the situation that some number contains mainly ‘nines’ in the course of a high-precision calculation is very common. Thereby insufficient precision in the FFT-multiplication will almost certainly result in an error.

The simplicity of the sum-of-digits test that uses the modulus \( R-1 \) can be seen from the polynomial identity

\[
\sum_k a_k R^k \equiv \sum_k a_k \mod R - 1 \quad (27.5-1)
\]
27.6 Binary exponentiation

One can use other moduli, like

$$\sum_k a_k R^k \equiv \sum_k (-1)^k a_k \mod R + 1$$ (27.5-2)

Moduli $R^n - 1$ for small $n$ are especially convenient:

$$\sum_k a_k R^k \equiv \sum_{k \equiv 0 \mod 2} a_k + R \sum_{k \equiv 1 \mod 2} a_k \mod R^2 - 1$$ (27.5-3a)

$$= \sum_{k \equiv 0 \mod 3} a_k + R \sum_{k \equiv 1 \mod 3} a_k + R^2 \sum_{k \equiv 2 \mod 3} a_k \mod R^3 - 1$$ (27.5-3b)

$$= \sum_{U=0}^{n-1} R^U \left( \sum_{k \equiv U \mod n} a_k \right) \mod R^n - 1$$ (27.5-3c)

One can keep the sums corresponding to the powers $R^U$ in separate variables. Note that the probability of an unrecognized error is reduced to approximately $1/R^n$. The multiplication of the residues involves additional work proportional to $n^2$.

27.6 Binary exponentiation

The binary exponentiation (or binary powering) scheme is a method to compute the $e$-th power of a number $a$ using about $\log(e)$ multiplications and squarings. The term ‘number’ can be replaced by about anything one can multiply. That includes integers, floating point numbers, polynomials, matrices, integer remainders modulo some modulus, polynomials modulo a polynomial and so on. In fact, the given algorithms work for any group: we do not need commutativity but $a^n \cdot a^m = a^{n+m}$ must hold (power-associativity).

27.6.1 Right-to-left powering

The algorithm uses the binary expansion of the exponent: let $e \geq 0$, write $e$ the base 2 as $e = [e_j, e_{j-1}, \ldots, e_1, e_0]$, $e_i \in \{0, 1\}$. Then

$$a^e = a^{1 \cdot e_0} a^{2 \cdot e_1} a^{4 \cdot e_2} a^{8 \cdot e_3} \cdots a^{2^j \cdot e_j}$$ (27.6-1a)

$$= 1 (a^1)^{e_0} (a^2)^{e_1} (a^4)^{e_2} (a^8)^{e_3} \cdots (a^{2^j})^{e_j}$$ (27.6-1b)

We initialize a variable $t$ by one, generate the powers $s_i = a^{2^i}$ by successive squarings $s_i = s_{i-1}^2 = (a^{2^{i-1}})^2$ and multiply $t$ by $s_i$ if $e_i$ equals one. The following C++ code computes the $e$-th power of the (double precision) number $a$:

```cpp
1 double power_r2l(double a, ulong e)
2 {
3     double t = 1;
4     if ( e )
5         {
6             double s = a;
7                 while ( 1 )
8                     {
9                     if ( e & 1 ) t *= s;
10                     e /= 2;
11                     if ( 0==e ) break;
12                     s *= s;
13                 }
14             }
15     return t;
16 }
```

A trivial optimization is to avoid the multiplication by 1 if the exponent is a power of two.
double power_r2l(double a, ulong e)
{
    if ( e == 0 ) return 1;
    double s = a;
    while ( 0 == (e & 1) )
    {
        s *= s;
        e /= 2;
    }
    a = s;
    while ( 0 != (e /= 2) )
    {
        s *= s;
        if ( e & 1 ) a *= s;
    }
    return a;
}

The program [FXT: \texttt{arith/power-r2l-demo.cc}] shows the quantities that occur with the computation of \( p = 2^{38} \):

\[
\begin{array}{c|c}
\text{arg 1: } 2 & \text{[number to exponentiate]} \text{ default=2} \\
\text{arg 2: } 38 & \text{[exponent]} \text{ default=38} \\
0 & 1.11. \\
2 & 2 \\
4 & 2 \\
8 & 2 \\
16 & 2 \\
32 & 2 \\
\hline
0 & 5532 \\
1 & 4294967296 \\
\hline
p=a^{38} = 274877906944 & 274877906944 \\
\end{array}
\]

In the right-to-left powering scheme the exponent is scanned starting from the lowest bit.

### 27.6.2 Left-to-right powering

The left-to-right binary powering algorithm scans the exponent starting from the highest bits. We use the facts that \( a^{2k} = (a^k)^2 \) and \( a^{2k+1} = (a^k)^2 a \). Implementation is simple:

double power_l2r(double a, ulong e)
{
    if ( e == 0 ) return 1;
    double s = a;
    ulong b = highest_one(e);
    while ( b > 1 )
    {
        b >>= 1;
        s *= s;
        if ( e & b ) s *= a;
    }
    return s;
}

The program [FXT: \texttt{arith/power-l2r-demo.cc}] shows the quantities that occur with the computation of \( p = 2^{38} \) when the left-to-right scan is used:

\[
\begin{array}{c|c}
\text{arg 1: } 2 & \text{[number to exponentiate]} \text{ default=2} \\
\text{arg 2: } 38 & \text{[exponent]} \text{ default=38} \\
0 & 1.11. \\
2 & 2 \\
4 & 2 \\
8 & 2 \\
16 & 2 \\
32 & 2 \\
\hline
0 & 2652 \\
1 & 274877906944 \\
\hline
p=a^{38} = 274877906944 & 274877906944 \\
\end{array}
\]

All multiplications apart from the squarings happen with the unchanged value of \( a \). This is an advantage if \( a \) is a small (integer) value so that the multiplications are cheap. As a slightly extreme example, if one computes \( 7^{77} \approx +0.3759823526783 \cdot 10^{695975} \) to full precision, then the left-to-right powering is about 3 times faster. If \( a \) is a full-precision number (and multiplication is done via FFTs) then the FFT of \( a \) only need to be computed once. Thereby all multiplications except for the first count a squarings. This technique is called \textit{FFT caching}. 

[fxtbook draft of 2008-August-17]
The given powering algorithms are good enough for most applications. There are schemes that improve further. For repeated power computations, especially for very large exponents the schemes based on the so-called *addition chains* lead to better algorithms, see [172]. The so-called ‘flexible window powering method’ is described and analyzed in [93]. A readable survey of exponentiation methods is given in [131]. Techniques for accelerating computations of factorials and binomial coefficients are described in [167].

### 27.6.3 Cost of binary exponentiation of full-precision numbers

<table>
<thead>
<tr>
<th>$e$</th>
<th>$e$ (radix 2)</th>
<th>#S</th>
<th>#M</th>
<th>#F</th>
<th>#C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.. . . . 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.. . . . 1.</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.. . . . 11</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>.. . . . 11.</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.. . . 1. . .</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>.. . . 1.1</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>.. . . 111</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>.. . . . 1...</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>.. . . . 11...</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>.. . . . 11.</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>.. . . 111</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>.. . . . 11...</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>. . . . 11.1</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>. . . . 111</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>. . . . 1111</td>
<td>3</td>
<td>3</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>16</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>22</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>23</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>3</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>24</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>26</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>27</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>3</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>28</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>29</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>3</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>30</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>31</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>3</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>. . . . . 1...</td>
<td>4</td>
<td>2</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 27.6-A: Cost of binary powering of full-precision numbers for small exponents $e$ in terms of squarings (#S), multiplications (#M) and FFTs (#F). If the left-to-right exponentiation algorithm with FFT caching needs less FFTs then the number is given under (#C).

With full-precision numbers the cost of binary powering is the same for both the left-to-right and the right-to-left algorithm. As an example, to raise $x$ to the 26-th power, note that $e = 26 = 11010_2$ and we can write

$$x^{26} = x^{16} \cdot x^8 \cdot x^2 = ((x^2)^2)^2 \cdot ((x^2)^2)^2 \cdot (x^2)$$

(27.6-2)

Here we need four squarings and two multiplications. In general one needs $\lfloor \log_2 e \rfloor$ squarings and $h(e) - 1$ multiplications were $h(e)$ is the number of set bits in the binary expansion of $e$. Figure 27.6-A lists the cost of the exponentiation for small exponents $e$ in terms of squarings and multiplications and, assuming a squaring costs two FFTs and multiplication three, in terms of FFTs. The table was created with the program [FXT: arith/power-costs-demo.cc].
Chapter 28

Root extraction

We describe methods to compute the inverse, square root, and higher roots of a given number. The computation of any of these costs just the equivalent of a few full-precision multiplications.

28.1 Division, square root and cube root

28.1.1 Inverse and division

The ordinary division algorithm is far too expensive for numbers of extreme precision. Instead one replaces the division \( \frac{a}{d} \) by the multiplication of \( a \) with the inverse of \( d \). The inverse of \( d \) is computed by finding a starting approximation \( x_0 \approx \frac{1}{d} \) and then iterating

\[
x_{k+1} = x_k + x_k (1 - dx_k)
\]

until the desired precision is reached. The convergence is quadratic (second order), which means that the number of correct digits is doubled with each step: if \( x_k = \frac{1}{d} (1 + e) \) then \( x_{k+1} = \frac{1}{d} (1 - e^2) \).

Moreover, each step only requires computations with twice the number of digits that were correct at its beginning. Still better: the multiplication \( x_k (\ldots) \) needs only to be done with half of the current precision as it computes the correcting digits (which alter only the less significant half of the digits). Thus, at each step we have 1.5 multiplications of the current precision: one full precision multiplication for \( dx_k \) plus a half precision multiplication for \( x_k (\ldots) \). The total work amounts to \( 1.5 + 1.5/2 + 1.5/4 + \ldots = 1.5 \cdot \sum_{n=0}^{N} \frac{1}{2^n} \) which is less than 3 full precision multiplications. The cost of a multiplication is set to \( \sim N \) for the estimates made here, this gives a realistic picture for large \( N \). Together with the final multiplication a division costs as much as 4 multiplications.

The numerical example given in figure 28.1-A shows the first steps of the computation of an inverse starting from a two-digit initial approximation.

The achieved precision can be determined by the absolute value of \( 1 - dx_k \). In hfloat, when the achieved precision is below a certain limit a third order correction is used to assure maximum precision at the last step:

\[
x_{k+1} = x_k + x_k (1 - dx_k) + x_k (1 - dx_k)^2
\]

One should in general not use algebraically equivalent forms like \( x_{k+1} = 2 x_k - dx_k^2 \) (for the second order iteration) because computationally there is a difference: cancellation can occur and the information on the achieved precision is not found easily.
Chapter 28: Root extraction

\[ d := 3.1415926 \]
\[ x_0 := 0.31 \quad \text{[initial 2-digit approximation for } 1/d] \]
\[ d \cdot x_0 := 3.141 \cdot 0.3100 = 0.9737 \]
\[ y_0 := 1.000 - d \cdot x_0 = 0.02629 \]
\[ x_0 \cdot y_0 := 0.3100 \cdot 0.02629 = 0.0081(49) \]
\[ x_1 := x_0 + x_0 \cdot y_0 = 0.3100 + 0.0081 = 0.3181 \]
\[ d \cdot x_1 := 3.1415926 \cdot 0.31810000 = 0.9993406 \]
\[ y_1 := 1.0000000 - d \cdot x_1 = 0.0006594 \]
\[ x_1 \cdot y_1 := 0.31810000 \cdot 0.0006594 = 0.0002097(5500) \]
\[ x_2 := x_1 + x_1 \cdot y_1 = 0.31810000 + 0.0002097 = 0.31830975 \]
\[ d \cdot x_2 := 3.1415926 \cdot 0.31830975 = 0.99999955 \]
\[ y_2 := 1.0000000 - d \cdot x_2 = 0.00000014 \]
\[ x_2 \cdot y_2 := 0.31830975 \cdot 0.00000014 = 0.000000044 \]
\[ x_3 := x_2 + x_2 \cdot y_2 = 0.31830975 + 0.000000044 = 0.31830979399 \]

\[ \text{Figure 28.1-A: First steps of the computation of the inverse of } \pi. \]

### 28.1.2 Inverse square root

Computation of inverse square roots can be done using a similar scheme: find a starting approximation \( x_0 \approx \frac{1}{\sqrt{d}} \) then iterate

\[ x_{k+1} = x_k + x_k \cdot \left(1 - \frac{d \cdot x_k^2}{2}\right) \quad (28.1-3) \]

until the desired precision is reached. Convergence is again second order: if \( x_k = \frac{1}{\sqrt{d}}(1 + \epsilon) \) then

\[ x_{k+1} = \frac{1}{\sqrt{d}} \left(1 - \frac{3}{2} \epsilon^2 - \frac{1}{2} \epsilon^3\right) \quad (28.1-4) \]

When the achieved precision is below a certain limit a third order correction should be applied:

\[ x_{k+1} = x_k + x_k \cdot \left(1 - \frac{d \cdot x_k^2}{2}\right) + x_k \cdot \frac{3(1 - d \cdot x_k^2)^2}{8} \quad (28.1-5) \]

To compute the square root first compute \( \frac{1}{\sqrt{d}} \) then a final multiply with \( d \) gives \( \sqrt{d} \).

Similar considerations as above (with squaring considered as expensive as multiplication, while with FFT multiplication it costs about 2/3 of a multiplication) give an operation count of 4 multiplications for computing \( \frac{1}{\sqrt{d}} \) and 5 for \( \sqrt{d} \). Note that this algorithm is considerably better than iterating \( x_{k+1} := \frac{1}{2} \left(x_k + \frac{d}{x_k}\right) \) because no long divisions are involved.

A unified routine that implements the computation of the inverse \( a \)-th roots is given in [hfloat: src/hf/itiroot.cc]. The general form of the divisionless iteration for the \( a \)-th root of \( d \) is, up to third
28.1. Division, square root and cube root

order:
\[ x_{k+1} = x_k \left( 1 + \frac{(1 - d x_k^2)}{a} + \frac{(1 + a)(1 - d x_k^2)^2}{2 a^2} \right) \] (28.1-6)

The initial approximation is obtained using ordinary floating point numbers (type double) with special precautions to avoid overflow with exponents that cannot be represented with doubles. Third order corrections are made whenever the achieved precision falls below a certain limit.

28.1.3 Cube root extraction

We use the relation \( d^{1/3} = d (d^2)^{-1/3} \). That is, we compute the inverse third root of \( d^2 \) using the iteration
\[ x_{k+1} = x_k + x_k \frac{(1 - d^2 x_k^3)}{3} \] (28.1-7)
and finally multiply with \( d \). Convergence is second order: if \( x_k = \frac{1}{\sqrt[d]{d}} (1 + e) \) then
\[ x_{k+1} = \frac{1}{\sqrt[d]{d}} \left( 1 - 2 e^2 - \frac{4}{3} e^3 - \frac{1}{3} e^4 \right) \] (28.1-8)

28.1.4 Improved iteration for the square root

Actually, the ‘simple’ version of the square root iteration \( x_{k+1} := \frac{1}{2} (x_k + \frac{d}{x_k}) \) can be used for practical purposes when rewritten as a coupled iteration for both \( \sqrt{d} \) and its inverse. Using for \( \sqrt{d} \) the iteration
\[ x_{k+1} = x_k - \frac{x_k^2 - d}{2 x_k} \] (28.1-9)
and for the auxiliary \( v \approx 1/\sqrt{d} \) the iteration
\[ v_{k+1} = v_k + v_k \left( 1 - x_k v_k \right) \] (28.1-11)
where one starts with approximations
\[ x_0 \approx \sqrt{d} \] (28.1-12)
\[ v_0 \approx 1/x_0 \] (28.1-13)
and the \( v \)-iteration step precedes that for \( x \). When carefully implemented this method turns out to be significantly more efficient than the computation via the inverse root. An implementation is given in [hfloat: src/hf/itsqrt.cc]. The idea is due to Schönhage.

28.1.5 A different view on the iterations

Let \( p \) be a prime and assume you know the inverse \( x_0 \) of a given number \( d \) modulo \( p \). With (the iteration for the inverse, relation 28.1-1 on page 569) \( \Phi(x) := x (1 + (1 d x)) \) the number \( x_1 := \Phi(x_0) \) is the inverse of \( d \) modulo \( p^2 \). Modulo \( p^2 \) we know that \( x_0 d \equiv (1 + k p) \) so we can write \( x_0 \equiv 1/d (1 + k p) \), thereby
\[ \Phi(x_0) = \Phi \left( \frac{1}{d} (1 + k p) \right) = \frac{1}{d} (1 - k p^2) \equiv \frac{1}{d} \mod p^2 \] (28.1-14)
The very same computation (with \( x_1 = 1/d (1+j p^2) \)) shows that for \( x_2 := \Phi(x_1) \) one has \( x_2 \equiv 1/d \mod p^4 \). Each application of \( \Phi \) doubles the exponent of the modulus.

The equivalent scheme works for root extraction. An example for the inverse square root: with \( p = 17 \) and \( x_0 = 3 \) we have \( x_0^2 \equiv 1 (x_0 \equiv 1/\sqrt{d} \mod p \) where \( d = 2 \). Now use the iteration \( \Phi(x) := x (1+(1-d x)/2) \) to compute \( x_1 = \Phi(x_0) \) to obtain \( x_1 = -45/2 \equiv 122 \mod p \) and observe that \( x_1^2 d \equiv 1 \mod p^2 \). Compute \( x_2 = \Phi(x_1) \) to obtain \( x_2 = -1815665 \equiv 21797 \mod p^2 \) and check that \( x_2^2 d \equiv 1 \mod p^4 \).

We will not go into the details (of the theory of \( p \)-adic numbers) but note that pari/gp can work with them:

\[
\text{\texttt{? 1/sqrt(2+O(17^5))}}
\]

\[
3 + 7*17 + 7*17^-2 + 4*17^-3 + 11*17^-4 + O(17^-5)
\]

\( \text{\texttt{\textbackslash Note that 21797 = 3 + 7*17 + 4*17^-2 + 11*17^-3 + 11*17^-4}} \)

\( \text{\texttt{\textbackslash and 122 = 3 + 7*17}} \)

Section 1.22 on page 61 describes the case \( p = 2 \). The computation of a square root modulo \( p^2 \), given a square root modulo \( p \), is described in section 37.9 on page 783.

### 28.2 Root extraction for rationals

We give expression for the extraction of the \( a \)-th root of a rational quantity.

#### 28.2.1 Extraction of the square root

A general formula for an \( k \)-th order \( (k \geq 2) \) iteration for \( \sqrt{d} \) is

\[
\Phi_k(x) = \sqrt{d} \left( \frac{x + \sqrt{d}}{x + \sqrt{d}} \right)^k = \sqrt{d} \left( \frac{p + q \sqrt{d}}{p + q \sqrt{d}} \right)^k \quad (28.2-1)
\]

where \( x = p/q \). All \( \sqrt{d} \) vanish when expanded:

\[
\Phi_2(x) = x^2 + d = \frac{p^2 + dq^2}{2pq} \quad (28.2-2a)
\]

\[
\Phi_3(x) = x^3 + 3\frac{d}{x} = \frac{p^2 + 3dq^2}{3pq} \quad (28.2-2b)
\]

\[
\Phi_4(x) = x^4 + 6\frac{d}{x^2} = \frac{p^4 + 6dpq^2}{4pq^2} \quad (28.2-2c)
\]

\[
\Phi_5(x) = x^5 + 10\frac{d}{x^2} = \frac{p^5 + 10dpq^2}{5pq^2} \quad (28.2-2d)
\]

\[
\Phi_k(x) = x \sum_{j=0}^{[k/2]} \frac{k}{2j} \frac{x^{k-2j} d^j}{\sum_{j=0}^{[k/2]} \frac{k}{2j+1} x^{k-2j-1} d^j} \quad (28.2-2e)
\]

The denominators and numerators of \( \Phi_k \) are terms of the second order recurrence

\[
a_k = 2x a_{k-1} - (x^2 - d) a_{k-2} \quad (28.2-3)
\]

with initial terms \( a_0 = 1 \), \( a_1 = x \) for the numerators, and \( a_0 = 0 \), \( a_1 = 1 \) for the denominators (that is, \( \Phi_0 = 1/0, \Phi_1 = x/1 \)). An equivalent form of relation 28.2-1 is

\[
\Phi_k(x) = \sqrt{d} \cot \left( k \arccot \frac{x}{\sqrt{d}} \right) \quad (28.2-4)
\]
28.2: Root extraction for rationals

Setting \( d = -1 \) and \( x = \cot(z) \) in relation 28.2-4 we obtain

\[
\cot(kz) = \Phi_k(\cot(z)) \tag{28.2-5}
\]

From this relation we obtain the following composition law:

\[
\Phi_m(\Phi_n(x)) = \Phi_{mn}(x) \tag{28.2-6}
\]

There is a nice expression for the error behavior of the \( k \)-th order iteration:

\[
\Phi_k\left(\sqrt{d} \cdot \frac{1+e}{1-e}\right) = \sqrt{d} \cdot \frac{1+e^k}{1-e^k} \tag{28.2-7}
\]

28.2.2 Extraction of the \( r \)-th root

A second order iteration for \( \sqrt[r]{z} \) is given by

\[
\Phi_2(x) = x + \frac{d - x^r}{r x^{r-1}} = \frac{(r-1) x^r + d}{r x^{r-1}} = \frac{1}{r} \left( (r-1) x + \frac{d}{x^{r-1}} \right) \tag{28.2-8}
\]

A third order iteration for \( \sqrt[d]{z} \) is

\[
\Phi_3(x) = x \cdot \frac{\alpha x^r + \beta d}{\beta x^r + \alpha d} = \frac{p}{q} \cdot \frac{\alpha p^r + \beta q^r d}{\beta p^r + \alpha q^r d} \tag{28.2-9}
\]

where \( x = p/q, \alpha = r - 1 \) and \( \beta = r + 1 \). An alternative form is

\[
\Phi_3(x) = \frac{d}{x} \cdot \frac{\beta x^r + \alpha d}{\alpha x^r + \beta d} \tag{28.2-10}
\]

where again \( \alpha = r - 1 \) and \( \beta = r + 1 \).

28.2.3 More iterations via Padé approximants

The iterations can also be obtained using Padé-approximants. Let \( P_{[i,j]}(z) \) be the Padé-expansion of \( \sqrt{z} \) around \( z = 1 \) of order \([i,j]\). An iteration of order \( i + j + 1 \) is given by \( x P_{[i,j]}(\frac{d}{z^2}) \). Different combinations of \( i \) and \( j \) result in alternative iterations:

\[
\begin{array}{l}
[i,j] \mapsto x P_{[i,j]} \left( \frac{d}{x^2} \right) \\
[1,0] \mapsto \frac{x^2 + d}{2x} \\
[0,1] \mapsto \frac{2x^3}{3x^2 - d} \\
[1,1] \mapsto \frac{x^2 + 3d}{3x^2 + d} \\
[2,0] \mapsto \frac{3x^4 + 6dx^2 - 3d^2}{8x^3} \\
[0,2] \mapsto \frac{15x^4 - 10dx^2 + 3d^2}{8x^5}
\end{array} \tag{28.2-11a-f}
\]
Still other forms are obtained by using $\frac{d}{x} P_{[i,j]} \left( \frac{x^2}{d} \right)$:

\[
[i,j] \quad \mapsto \quad \frac{d}{x} P_{[i,j]} \left( \frac{x^2}{d} \right) \quad (28.2-12a)
\]

\[
[1,0] \quad \mapsto \quad \frac{x^2 + d}{2x} \quad (28.2-12b)
\]

\[
[0,1] \quad \mapsto \quad \frac{2d^2}{3dx - x^3} \quad (28.2-12c)
\]

\[
[1,1] \quad \mapsto \quad \frac{d(d + 3x^3)}{x(3d + x^2)} \quad (28.2-12d)
\]

\[
[2,0] \quad \mapsto \quad \frac{-x^4 + 6dx^2 + 3d^2}{8xd} \quad (28.2-12e)
\]

\[
[0,2] \quad \mapsto \quad \frac{8d^3}{3x^4 - 10dx^2 + 15d^2} \quad (28.2-12f)
\]

Pade approximations for the $r$-th root can be given in terms of hypergeometric functions. Rewrite relation 35.2-30 on page 700 as

\[
F \left( \frac{u, v}{u + v + 1/r} \bigg| z \right) / F \left( \frac{u + 1/r, v + 1/r}{u + v + 1/r} \bigg| z \right) = (1 - z)^{1/r} \quad (28.2-13)
\]

The expression on the left gives the approximant $[i,j]$ if we set $u = -i$ and $v = -j - 1/r$ (so both series terminate). An iteration of order $i + j + 1$ is obtained as $\Phi_{[i,j]} = x P_{[i,j]} \left( \frac{d}{x^r} \right)$. We compute the third order iteration for the fourth root (relation 28.2-9 with $r = 4$):

1. $r=4; \quad r$-th root
2. $i=1; \quad \text{degree of denominator}$
3. $j=1; \quad \text{degree of numerator}$
4. $u=-i; v=-j-1/r; \quad \text{setup parameters so that series terminate}$
5. $N=\text{hypergeom}(\{u,v\},\{u+v+1/r\},x,i)$
6. $-5/8*x + 1$
7. $D=\text{hypergeom}(\{u+1/r,v+1/r\},\{u+v+1/r\},x,j)$
8. $-3/8*x + 1$
9. $t=N/D \quad \text{Pade approximant [i,j]}$
10. $(-5*x + 8)/(3*x + 8)$
11. $\text{check } t = (1-x)^{-1/r} + \text{order}(i+j+1)$
12. $n=i+j+2;$
13. $t=\text{hypergeom}([-i/r],[],x,n+0(x^{-n}))$
14. $5/256*x^3 + 0(x^{-4})$
15. $t=\text{hypergeom}(\{u,v\},\{u+v+1/r\},x,i)$
16. $t=\text{hypergeom}(\{u,v\},\{u+v+1/r\},x,i)$
17. $t=x*(3*x^4 + 5*d)/(5*x^4 + 3*d)$

Now we check the order of the iteration, we set $d = 1$ and compute $f \left( \Phi \left( d^{1/r}(1 + e) \right) \right) = f \left( \Phi(1 + e) \right)$:

1. $f(x)=x^r-1; \quad \text{for } d=1$
2. $g=\text{substitute}(g, d, 1); \quad d=1$
3. $e=\text{substitute}(e, (1+e)); \quad \Phi( d^{1/r}(1+e) ) = \Phi(1+e)$
4. $\text{taylor}(f(e), e) \quad f( \Phi(1+e) ) =? 0(e^{-1+j+1})$
5. $5*e^3 - 15/2*e^4 + [\ldots] \quad 0K$

An alternative form of iterations can be obtained as $\Phi_{[i,j]} = \frac{d}{x} P_{[i,j]} \left( \frac{x^r}{d} \right)$. The $[1,1]$-approximant leads to the third order iteration given as relation 28.2-10.

In section 29.6.1 on page 599 Pade approximants are used to obtain iterations for arbitrary functions $f$. 

[fxtbook draft of 2008-August-17]
28.3 Divisionless iterations for the inverse \( a \)-th root

There is a nice general formula that gives iterations with arbitrary order of convergence for \( 1/\sqrt[d]{a} = d^{-1/a} \) that involve no long division.

One uses the identity

\[
d^{-1/a} = x (1 - (1 - x^a d))^{-1/a}
\]

(28.3-1)

\[
d^{-1/a} = x(1 - y)^{-1/a} \quad \text{where} \quad y := (1 - x^a d)
\]

(28.3-2)

Taylor expansion gives

\[
d^{-1/a} = x \sum_{k=0}^{\infty} (1/a)_k y^k
\]

(28.3-3)

where \( z_k := z (z + 1) (z + 2) \ldots (z + k - 1) \) (and \( z^0 := 1 \), \( z^k \) is called the rising factorial power). Written out:

\[
d^{-1/a} = x \frac{1}{\sqrt[1 - y]} = x \left( 1 + \frac{y}{a} + \frac{(1 + a) y^2}{2 a^2} + \frac{(1 + a)(1 + 2a) y^3}{6 a^3} + \frac{(1 + a)(1 + 2a)(1 + 3a) y^4}{24 a^4} + \cdots + \frac{\prod_{k=1}^{n-1} (1 + k a) y^n}{n! a^n} + \cdots \right)
\]

(28.3-4)

A \( n \)-th order iteration for \( d^{-1/a} \) is obtained by truncating the above series after the \((n - 1)\)-th term:

\[
\Phi_n(x) := x \sum_{k=0}^{n-1} (1/a)_k y^k
\]

(28.3-5)

\[
x_{k+1} = \Phi_n(x_k)
\]

(28.3-6)

Convergence is \( n \)-th order:

\[
\Phi_n(d^{-1/a}(1 + e)) = d^{-1/a}(1 + O(e^n))
\]

(28.3-7)

For example, the second order iteration is

\[
\Phi_2(x) := x + x \frac{1 - d x^a}{a}
\]

(28.3-8)

Convergence is indeed quadratic: if \( x = \frac{1}{\sqrt[d]{a}} (1 + e) \) then

\[
\Phi_2(x) = \frac{1}{\sqrt[d]{a}} \left( (1 + e) \left[ (1 + e)^a - (a + 1) \right] \right)
\]

(28.3-9)

\[
= \frac{1}{\sqrt[d]{a}} \left( 1 - \frac{a + 1}{2} e^2 + O(e^3) \right)
\]

(28.3-10)

28.3.1 Iterations for the inverse

Set \( a = 1 \), \( y = 1 - d x \) to compute the inverse of \( d \).

\[
\frac{1}{d} = x \frac{1}{1 - y}
\]

(28.3-11a)

\[
\Phi_k(x) = x \left( 1 + y + y^2 + y^3 + y^4 + \cdots + y^{k-1} \right)
\]

(28.3-11b)
Chapter 28: Root extraction

\( \Phi_2(x) = x(1 + y) \) is the second order iteration 28.1-3 on page 570.

Composition is particularly simple with the iterations for the inverse:

\[
\Phi_{n,m} = \Phi_n(\Phi_m)
\]

(28.3-12)

There are simple closed forms for this iteration:

\[
\Phi_k = \frac{1 - y^k}{d} = x \frac{1 - y^k}{1 - y}
\]

(28.3-13a)

\[
\Phi_\infty = 1 + x + x^2 + x^3 + x^4 + \ldots
\]

(28.3-13b)

\[
= x(1 + y)(1 + y^2)(1 + y^4)(1 + y^8) \ldots
\]

(28.3-13c)

\[
= x(1 + y + y^2)(1 + y^3 + y^6)(1 + y^9 + y^{18}) \ldots
\]

(28.3-13d)

The expression for the convergence of the \( k \)-th order iteration is

\[
\Phi_k \left( \frac{1}{d} \right) (1 + e) = \frac{1}{d} (1 - (e)^k)
\]

(28.3-14)

The iteration converges if one has \(|e| < 1\) for the start value \( x_0 = \frac{1}{d} (1 + e) \). That is, the basin of attraction is the open disc of radius \( r = 1/d \) around the point \( 1/d \), independent of the order \( k \). For other iterations, the basin of attraction usually has a fractal boundary and further depends on the order.

28.3.2 Iterations for the inverse square root

Set \( a = 2 \), \( y = 1 - d x^2 \) to compute the inverse square root of \( d \).

\[
\frac{1}{\sqrt{d}} = x \frac{1}{\sqrt{1 - y}}
\]

(28.3-15a)

\[
= x \left( 1 + \frac{y}{2} + \frac{3 y^2}{8} + \frac{5 y^3}{16} + \frac{35 y^4}{128} + \ldots + \frac{(2^k) y^k}{4^k} + \ldots \right)
\]

(28.3-15b)

\[
\Phi_{k+1}(x) = x \left( 1 + \frac{y}{2} + \frac{3 y^2}{8} + \ldots + \frac{(2^k) y^k}{4^k} \right)
\]

(28.3-15c)

\( \Phi_2(x) = x(1 + y/2) \) is the second order iteration 28.1-1 on page 569.

28.3.3 Computation of the \( a \)-th root

The following (second order) iteration computes \( \sqrt[d]{a} \) directly:

\[
\Phi(x) = x + \frac{1}{a} \left( x - \frac{x^{a+1}}{d} \right)
\]

(28.3-16)

Figure 28.3-A shows the quantities occurring in the iterative computation of \( \sqrt[d]{a} \). The iteration is involves no long division for small (rational) \( d \).

To compute the \( a \)-root of a full-precision number \( d \) one can use the iteration for the inverse root and invert afterwards. Another possibility is to compute the inverse \( a \)-th root of \( d^{a-1} \) and multiply with \( d \) afterwards:

\[
[(d^{a-1})^{-1/a} d]^{1/a} = [d^{(1-a)/a}] d = d^{1/a}
\]

(28.3-17)

If \( a \) is small the cost is lower than with the final iteration for the inverse (which costs about 3 Multiplications or 9 FFTs). The powering-method is not more expensive than inversion if the rightmost column in figure 27.6-A on page 567 for \( e = a - 1 \) is less or equal 6. In case the iteration for the inverse involves a loss of precision the method might be preferred also if its cost is higher.
28.3: Divisionless iterations for the inverse \( a \)-th root

\[
\begin{align*}
\text{28.3-4 Error expressions for inverse square root iterations *}
\end{align*}
\]

An expression for the error behavior of the \( n \)-th order iteration similar to relation 28.2-7 on page 573 is

\[
F_n := \Phi_n \left( d - \frac{1}{2} \frac{1 + e}{1 - e} \right) / d^{-1/2}
\]

(28.3-18a)

\[
= \sum_{k=0}^{n} \binom{2n-1}{k} (-e)^k - \sum_{k=n+1}^{2n+1} \binom{2n-1}{k} (-e)^k
\]

(28.3-18b)

\[
= \frac{c + 1 + e}{1 - c}
\]

(28.3-18c)

Now define \( c := \frac{F_n - 1}{F_{n+1}} \), then

\[
F_n = \frac{1 + c}{1 - c}
\]

where

\[
c = e^n \sum_{k=0}^{n} \binom{2n-1}{n-k} (-e)^k
\]

(28.3-18d)

The coefficients of the Taylor expansion of \( F_n \) in \( e \) are always integers.

[\text{fxtbook draft of 2008-August-17}]
Chapter 28: Root extraction

For \( n = 4 \):

\[
F_4 := \Phi_4(d^{-1/2} \frac{1 + e}{1 - e}) / d^{-1/2}
\]

\[
= 1 - 7e + 21e^2 - 35e^3 - 35e^4 + 21e^5 - 7e^6 + e^7
\]

\[
= 1 - 70e^4 - 448e^5 - 1680e^6 - 4800e^7 - 11550e^8 - \ldots
\]

\[
= 1 + c \quad \text{where} \quad c = e^4 - 7e^2 + 21e - 35
\]

Two curious formulas related to the error behavior of \( \Phi_2 \) are

\[
\Phi_2 \left( \frac{1}{\sqrt{d}} \left[ e + \frac{1}{e} \right] \right) = \frac{1}{\sqrt{d}} \left[ -\frac{1}{2} \cdot \left( e^3 + \frac{1}{e^3} \right) \right]
\]

\[
\Phi_2 \left( \frac{1}{\sqrt{d}} \left[ e - \frac{2}{3} \cdot \frac{1}{e} \right] \right) = \frac{1}{\sqrt{d}} \left[ +\frac{1}{2} \cdot \left( e^3 - \frac{2}{3} \cdot e^3 \right) \right]
\]

28.4 Initial approximations for iterations

With the iterative schemes one always needs an initial approximation for the root that is to be computed. Assume we want to compute \( f(d) \), for example, \( f(d) = \sqrt{d} \) or \( f(d) = \exp(d) \). One can convert the high precision number \( d \) to a machine floating point number and use the FPU to compute an initial approximation. However, when \( d \) cannot be represented with a machine float, the method fails. The method will also fail if the result causes an overflow, which is likely to happen with \( f(d) = \exp(d) \).

28.4.1 Inverse roots

With \( f(d) = d^{1/a} \) one can use the following technique. Write \( d \) in the form

\[
d = M \cdot R^X
\]

were \( M \) is the mantissa, \( R \) the radix and \( X \) the exponent. We have \( 0 \leq M < 1 \) and \( X \in \mathbb{Z} \). Now use

\[
d^{1/a} = M^{1/a} \cdot R^{X/a} = M^{1/a} \cdot R^{Y/a} \cdot R^Z
\]

where \( Z = \lfloor X/a \rfloor \) and \( Y = X - a \cdot Z \) (so \( X = a \cdot Z + Y \)).

The algorithm computes the three quantities in relation 28.4-2 separately and finally computes the product as result. A C++ implementation is given in \[hf\float: src/hf/itiroot.cc\]:

```c

void approx_invpow(const hfloat &d, hfloat &c, long a) {
  double dd;
  double dd;
  dd = pow(dd, 1.0/(double)a); // M^(1/a)
  dd *= tt; // M^(1/a) * R^(Y/a)
  dd %= tt; // M^(1/a) * R^(Y/a)
  d2hfloat(dd, c); // c = M^(1/a) * R^(Y/a)
  c.exp( c.exp()+Z ); // c *= R^(Z)
}
```
One can also subtract \( a \cdot Z \) from the exponent before the iteration and add \( Z \) to the exponent afterwards: 
\[
(R^{X-aZ})^{1/a} = R^{Y/a} = R^{X/a}/R^Z.
\]
In that case the initial approximation can be computed via the straight forward approach.

### 28.4.2 Exponential function

With \( f(d) = \exp(d) \) write
\[
\exp(d) = M \cdot R^X
\]
Then use \( X = \lfloor d / \log(R) \rfloor \) and \( M = \exp(d - X \cdot \log R) \). Note that \( d \) must fit into a machine float which is not a real restriction as the \( \exp(d) \) will not fit into a \texttt{hfloat} type already with smaller values as the exponent of the result would overflow.

A C++ implementation is given in [hfloat: src/tz/itexp.cc]:

```cpp
void approx_exp(const hfloat &d, hfloat &c) {
    double dd;
    hfloat2d(d, dd);
    double lr = log( hfloat::radix() );
    double X = floor( dd/lr );
    double M = exp( dd-X*lr );
    d2hfloat(M, c);
    c.exp( c.exp()+(long)X );
}
```

The iteration for computation of the exponential function is given in section 31.2 on page 633.

### 28.5 Some applications of the matrix square root

We give applications of the iteration for the (inverse) square root to compute re-orthogonalized matrices, the polar decomposition, the sign decomposition, and the pseudo-inverse of a matrix.

#### 28.5.1 Re-orthogonalization

A task from graphics applications: a rotation matrix \( A \) that deviates from being orthogonal (for example, due to cumulative errors resulting from many multiplications with rotation matrices) shall be transformed to the closest orthogonal matrix \( E \). One has (see [233]):
\[
E = A (A^T A)^{-\frac{1}{2}}
\]
(28.5-1)

With the division-free iteration for the inverse square root
\[
\Phi(x) = x \left( 1 + \frac{1}{2} (1 - dx^2) + \frac{3}{8} (1 - dx^2)^2 + \frac{5}{16} (1 - dx^2)^3 + \ldots \right)
\]
(28.5-2)
the given task is pretty easy: as \( A^T A \) is close to unity (the identity matrix) we can use the (second order) iteration with \( d = A^T A \) and \( x = 1 \)
\[
(A^T A)^{-\frac{1}{2}} \approx \left( 1 + \frac{1}{2} A^T A \right)
\]
(28.5-3)
and multiply by \( A \) to get a ‘closer-to-orthogonal’ matrix \( A_+ \):
\[
A_+ = A \left( 1 + \frac{1}{2} A^T A \right) \approx E
\]
(28.5-4)
The step can be repeated with $A_+ \ (\text{or higher orders can be used})$ if necessary. Note the identical equation would be obtained when trying to compute the inverse square root of 1:

$$x_+ = x \left(1 + \frac{1 - x^2}{2}\right) \rightarrow 1 \quad (28.5-5)$$

It is instructive to write things down in the singular value decomposition (SVD) representation

$$A = U \Omega V^T \quad (28.5-6)$$

where $U$ and $V$ are orthogonal and $\Omega$ is a diagonal matrix with non-negative entries, see [288]. We note that the SVD is not unique, using the $1 \times 1$ matrix $\begin{bmatrix}-2 \end{bmatrix} = \begin{bmatrix}-1 \end{bmatrix} \begin{bmatrix}2 \end{bmatrix} = \begin{bmatrix}1 \end{bmatrix} \begin{bmatrix}-1 \end{bmatrix}$. The SVD is a decomposition of the action of the matrix as: rotation – element-wise stretching – rotation. Now

$$A^T A = (V \Omega U^T) (U \Omega V^T) = V \Omega^2 V^T \quad (28.5-7)$$

Thereby (using the fact that $(V \Omega U^T)^n = V \Omega^n U^T$)

$$A^T A^{-\frac{1}{2}} = \left((V \Omega U^T) (U \Omega V^T)\right)^{-\frac{1}{2}} = (V \Omega^2 V^T)^{-\frac{1}{2}} = V^{-1} V^T \quad (28.5-8)$$

and we have

$$A (A^T A)^{-\frac{1}{2}} = (U \Omega V^T) (V^{-1} V^T) = U V^T \quad (28.5-9)$$

that is, the ‘stretching part’ was removed.

A numerical example: Let

$$A = \begin{bmatrix}+1.0000000 & +1.0000000 & +0.7500000 \\-0.5000000 & +1.5000000 & +1.0000000 \\
+0.7500000 & +0.5000000 & -1.0000000 \end{bmatrix} \quad (28.5-10)$$

then

$$E = \begin{bmatrix}+0.803114165 & +0.291073143 & +0.519888513 \\-0.486897253 & +0.823533541 & +0.291073143 \\
+0.343422053 & +0.486897252 & -0.803114166 \end{bmatrix} \quad (28.5-11)$$

and $EE^T = 1$.

### 28.5.2 Polar decomposition

The so-called polar decomposition of a matrix $A$ is a representation of the form

$$A = ER \quad (28.5-12)$$

where the matrix $E$ is orthogonal and $R = R^T$. It is an analogue to the representation of a complex number $z \in \mathbb{C}$ as $z = e^{i \phi} r$ (identify $R \sim r$ and $E \sim e^{i \phi}$).

The polar decomposition can be defined by

$$A = E R := \left(A (A^T A)^{-1/2}\right) \left((A^T A)^{1/2}\right) \quad (28.5-13)$$

where $R = (A^T A)^{1/2}$ and $E = A (A^T A)^{-1/2}$. The matrix $E$ is computed as before:

$$E = A \cdot \left(1 + \frac{1 - A^T A}{2}\right) \cdot \left(1 + \frac{1 - A^T_{+} A_{+}}{2}\right) \cdot \ldots \quad (28.5-14)$$
The matrix $R$ equals $E^{-1}A = E^T A$, that is

$$A = ER = E \left( E^T A \right)$$

$$= UV^T \left( V \Omega V^T \right)$$

(28.5-15)

(28.5-16)

Compute the polar decomposition as

$$E_0 = A$$

(28.5-17a)

$$Y_k = \left( 1 + \frac{1 - E_k^T E_k}{2} \right)$$

(28.5-17b)

$$E_{k+1} = E_k Y_k \rightarrow E$$

(28.5-17c)

$$R_{k+1} = E_{k+1}^T A \rightarrow R$$

(28.5-17d)

$$E_{k+1} R_{k+1} \rightarrow A$$

(28.5-17e)

Higher orders can be added in the computation of $Y_k$. If you prefer $z = r e^{i \phi}$ over $e^{i \phi} r$ then iterate as above but set $R' = AE^T$ so that

$$A = R' E = \left( AE^T \right) E$$

$$= \left( U \Omega U^T \right) UV^T$$

(28.5-18)

(28.5-19)

Numerical example: Let

$$A = \begin{bmatrix} +1.0000 & +1.0000 & +0.75000 \\ -0.5000 & +1.5000 & +1.0000 \\ +0.75000 & +0.50000 & -1.0000 \end{bmatrix}$$

then

$$A = ER$$

$$= \begin{bmatrix} +0.80311 & +0.29107 & +0.51988 \\ -0.48689 & +0.82353 & +0.29107 \\ +0.34342 & +0.48689 & -0.80311 \end{bmatrix} \begin{bmatrix} +1.30412 & +0.24447 & -0.22798 \\ +0.24447 & +1.76982 & +0.55494 \\ -0.22798 & +0.55494 & +1.48410 \end{bmatrix}$$

(28.5-20)

$$A = R' E$$

$$= \begin{bmatrix} +1.48410 & +0.55494 & +0.22798 \\ +0.55494 & +1.76982 & -0.24447 \\ +0.22798 & -0.24447 & +1.30412 \end{bmatrix} \begin{bmatrix} +0.80311 & +0.29107 & +0.51988 \\ -0.48689 & +0.82353 & +0.29107 \\ +0.34342 & +0.48689 & -0.80311 \end{bmatrix}$$

(28.5-21a)

(28.5-21b)

(28.5-21c)

(28.5-21d)

### 28.5.3 Sign decomposition

The *sign decomposition* can be defined as

$$A = SN = \left( A(A^2)^{-1/2} \right) \left( (A^2)^{1/2} \right)$$

(28.5-22)

where $N = (A^2)^{-1/2}$ and $S = A (A^2)^{-1/2}$. The square root has to be chosen such that all its eigenvalues have positive real parts. The sign decomposition is undefined if $A$ has eigenvalues on the imaginary axis. The matrix $S$ is its own inverse (its eigenvalues are $\pm 1$). The matrices $A$, $S$ and $N$ commute pairwise: $SN = NS$, $AN = NA$ and $AS = SA$.

Use

$$S_0 = A$$

(28.5-23a)

$$Y_k = \left( 1 + \frac{1 - S_k^2}{2} \right)$$

(28.5-23b)

$$S_{k+1} = S_k Y_k \rightarrow S$$

(28.5-23c)

$$N_{k+1} = S_{k+1} A \rightarrow N$$

(28.5-23d)
Chapter 28: Root extraction

Numerical example: Let

$$A = \begin{bmatrix}
+1.0000 & +1.0000 & +0.7500 \\
0.5000 & +1.5000 & +1.0000 \\
+0.5000 & +0.5000 & -1.0000
\end{bmatrix}$$

then

$$A = SN = \begin{bmatrix}
+0.90071 & -0.01706 & +0.29453 \\
-0.24065 & +0.95862 & +0.71389 \\
+0.62679 & +0.10775 & -0.85933
\end{bmatrix} \begin{bmatrix}
+1.13014 & +1.02237 & +0.36392 \\
-0.18454 & +1.55423 & +0.06423 \\
-0.07158 & +0.35875 & +1.43718
\end{bmatrix}$$

where $SS = 1$. See [146] and also [147].

28.5.4 Pseudo-inverse

While we are at it: define a matrix $A^+$ as

$$A^+ := (AA^T)^{-1}A^T = (V\Omega^{-2}V^T)(V\Omega U^T) = V\Omega^{-1}U^T$$

This looks suspiciously like the inverse of $A$. In fact, this is the pseudo-inverse of $A$:

$$A^+ A = (V\Omega^{-1}U^T)(U\Omega V^T) = 1 \text{ but wait}$$

$A^+$ has the nice property to exist even if $A^{-1}$ does not. If $A^{-1}$ exists, it is identical to $A^+$. If not, $A^+A \neq 1$ but $A^+$ will give the best possible (in a least-square sense) solution $x^+ = A^+b$ of the equation $Ax = b$ (see [95, p.770]). To find $(AA^T)^{-1}$ use the iteration for the inverse:

$$\Phi(x) = x \left(1 + (1-dx) + (1-dx)^2 + \ldots\right)$$

with $d = AA^T$ and the start value $x_0 = 2 - n(AA^T)/\|AA^T\|^2$ where $n$ is the dimension of $A$.

A pari/gp implementation of the pseudo-inverse using the SVD:

```
1 matpseudoinv(A)=
2 \ Return pseudo-inverse of A
3 { local(t, x, U, d, V);
4    t = matSVD(A);
5    U = t[1]; d = t[2]; V = t[3];
6    for (k=1, matsize(d)[1],
7      x=d[k,k]; if (x>1e-15, d[k,k]=1/x, d[k,k]=0);
8    );
9    return( V*d*U~ );
10 }
```

Where the SVD is computed with the help of a routine (qfjacobi()) that returns the eigenvectors of a real symmetric matrix:

```
1 matSVDcore(A)=
2 \ Singular value decomposition:
3 \ Return [U, d, V] so that U*d*V~==A
4 \ d is a diagonal matrix
5 \ U, V are orthogonal
6 { local(U, d, V); \ returned quantities
7    local(t, R, d1);
8    R = conj(A'*A) \ R=V*d*V~
9    t = qfjacobi( R ); \ fails with eigenvalues==zero
10    V = t[2];
11    d = real(sqrt(t[1]));
12    d1 = d;
13    for (k=1, length(d1), t=d1[k]; if (abs(t)>1e-15, t=1/t, t=0); d1[k]=t );
```

[fxtbook draft of 2008-August-17]
28.5: Some applications of the matrix square root

A =
[+1.00  +1.00  +0.75  +2.00]
[ -0.50  +1.50  +1.00  +3.00]
[ +0.75  +0.50  -1.00  -3.00]

A =
[+0.64401153492 +0.438818890 +0.6262468643]
[-0.69537213279 +0.676976586 +0.2411644651]
[-0.318126941467 -0.590881276 +0.7413869203]

t = matSVD(A); U = t[1]; d = t[2]; V = t[3];

U =
[+0.644401153492 +0.438818890 +0.6262468643]
[-0.695372132379 +0.676976586 +0.2411644651]
[-0.318126941467 -0.590881276 +0.7413869203]

d = [+0.95641003 0 0]
[ 0 +5.09161169 0]
[ 0 0 +1.74234618]

V =
[+0.787833655771 -0.067332385426 +0.60935354299]
[-0.583139548860 +0.227598489665 +0.77980313700]
[-0.318126941467 -0.590881276 +0.7413869203]

Ax = matpseudoinv(A)

A*A =
[+1.000000000000  +3.78653234 E-29  -4.41762106 E-29]
[ +2.52435489 E-29  +1.000000000000  -2.52435489 E-29]
[ -2.52435489 E-29  -2.52435489 E-29  +1.000000000000]

A*X

Figure 28.5-A: Numerical example for the pseudo-inverse computed by the SVD. We use a 3 × 4 matrix which is definitely not invertible. A working precision of 25 decimal digits was used, so \( AA^+ = 1 \) to within that precision. On the other hand, \( A^+ A \) is not close to the unit matrix.

```plaintext
15  d1 = matdiagonal(d1);
16  d = matdiagonal(d);
17  U = (A*d1);  # U = A*d1;
18  return( [U, d, V] );
19 }
```

The core routine is always called with a matrix \( A \) whose number of rows is greater or equal to its number of rows. Thereby we make sure that when the main routine with argument \( A \) computes \( U, d, V \) so that \( A = U d V^T \) then with argument \( A^T \) we obtain \( X, d, Y \) where \( A^T = X d Y^T \), \( X = V \), and \( Y = U \).

```plaintext
1  matSVD(A)=
2  {
3    local(tq, t, U, d, V);
4    t = matsize(A);
5    tq=0; if ( t[1]<t[2], tq=1; A=A~; );
6    t = matSVDcore(A);
7    d = t[2];
8    if ( tq,
9      U = t[3]; V=t[1];
10      /* else */
11      U = t[1]; V=t[3];
12    );
13    return( [U, d, V] );
14 }
```

For a numerical example see figure 28.5-A. The connection between the SVD of a matrix \( a \) and the eigenvectors of \( A^T a \) is described in [161].
28.6 Goldschmidt’s algorithm

A framework for the so-called Goldschmidt algorithm can be stated as follows. Initialize

\[ x_0 = d^A, \quad E_0 = d^B \]  \hspace{1cm} (28.6-1a)

then iterate

\[ P_{k+1} = 1 + \frac{1 - E_k}{a} \]  \hspace{1cm} (28.6-1b)
\[ x_{k+1} = x_k P_k^b \rightarrow d^{A-B/a} \]  \hspace{1cm} (28.6-1c)
\[ E_{k+1} = E_k P_k^a \rightarrow 1 \]  \hspace{1cm} (28.6-1d)

The algorithm converges quadratically. The updates for \( x \) and \( E \) (last two relations) can be computed independently. The iteration is not self-correcting, so the computations have to be carried out with full precision in all steps.

An invariant of the algorithm is given by \( x_k^a/E_k^b \):

\[ \frac{x_{k+1}^a}{E_{k+1}^b} = \frac{\left(x_k \cdot P_k^b\right)^a}{\left(E_k \cdot P_k^a\right)^b} = \frac{x_k^a}{E_k^b} \]  \hspace{1cm} (28.6-2a)

Using

\[ \frac{x_0^a}{E_0^b} = \frac{d^{A-a}}{d^{Bb}} = d^{A-a-Bb} \]  \hspace{1cm} (28.6-2b)

and, as \( E \) converges to 1, we find that

\[ \frac{x_\infty^a}{E_\infty^b} = \frac{x_0^a}{E_0^b} \]  \hspace{1cm} (28.6-2c)

That is,

\[ x_\infty = \left(\frac{x_0^a}{E_0^b}\right)^{1/a} = \frac{x_0}{E_0^{b/a}} = \frac{d^A}{d^{B/a}} = d^{A-B/a} \]  \hspace{1cm} (28.6-2d)

One can now look for interesting special cases, \( b \) is set to one in what follows.

28.6.1 Algorithm for the \( a \)-th root

Solving \( A - B/a = 1/a \) gives \( B = A a - 1 \) and especially \( A = 0, B = 1 \). That is, set

\[ x_0 = d, \quad E_0 = d^{a-1} \]  \hspace{1cm} (28.6-3a)

then iterate

\[ P_k := 1 + \frac{1 - E_k}{a} \rightarrow 1 \]  \hspace{1cm} (28.6-3b)
\[ x_{k+1} := x_k \cdot P_k \]  \hspace{1cm} (28.6-3c)
\[ E_{k+1} := E_k \cdot P_k^a \rightarrow 1 \]  \hspace{1cm} (28.6-3d)

until \( x \) close enough to \( x_\infty = d^{1/a} \)
Computation of the square root

For $a = 2$ one obtains an algorithm for the computation of the square root:

$$\sqrt{d} = d \prod_{k=0}^{\infty} \frac{3 - E_k}{2}$$ \hfill (28.6-4)

where $E_0 = d$, $E_{k+1} := E_k (\frac{3-E_k}{2})^2$.

28.6.2 Algorithm for the inverse $a$-th root

Solving $A - B/a = -1/a$ gives $B = A a + 1$ and especially $A = 1, B = a - 1$. That is, set

$$x_0 = 1, \quad E_0 = d$$ \hfill (28.6-5)

then iterate as in formulas 28.6-3b, 28.6-3d until $x$ close enough to $x_{\infty} = d^{-\frac{1}{a}}$.

Computation of the inverse

Setting $a = 1$ one obtains an algorithm for the inverse ($P_k = 1 + (1 - E_k) = 2 - E_k$):

$$\frac{1}{d} = \prod_{k=0}^{\infty} (2 - E_k)$$ \hfill (28.6-6)

where $E_0 = d$, $E_{k+1} := E_k (2 - E_k)$.

Computation of the inverse square root

For $a = 2$ one obtains an algorithm for the inverse square root ($P_k = 1 + (1 - E_k)/2 = (3 - E_k)/2$):

$$\frac{1}{\sqrt{d}} = \prod_{k=0}^{\infty} \frac{3 - E_k}{2}$$ \hfill (28.6-7)

where $E_0 = d$, $E_{k+1} := E_k (\frac{3-E_k}{2})^2$.

28.6.3 Higher order algorithms for the inverse $a$-th root

Higher order iterations are obtained by appending higher terms to the expression $(1 + \frac{1-E_k}{a})$ in the definitions of $P_{k+1}$ as suggested by equation 28.3-4 on page 578 and the identification $y = 1 - E$:

$$E_{k+1} = E_k P_k^a \quad \text{where}$$

$$P_k = 1 + \frac{1 - E_k}{a} \quad \text{[second order]}$$ \hfill (28.6-8)

$$+ \frac{(1+a)(1 - E_k)^2}{2 a^2} \quad \text{[third order]}$$

$$+ \frac{(1+a)(1 + 2a)(1 - E_k)^3}{6 a^3} \quad \text{[fourth order]}$$

$$+ \ldots +$$

$$+ \frac{(1+a)(1 + 2a) \ldots (1 + (n-1)a)(1 - E_k)^n}{n! a^n} \quad \text{[order $(n+1)$]}$$

[fxtbook draft of 2008-August-17]
\[ x_0 = 1 \]
\[ E_0 = 2.0 \]
\[ P_0 = 0.90625 \]
\[ b_0 = 0.0 \]

\[ x_1 = 0.90625 \]
\[ E_1 = 1.3490314483642578125 \]
\[ P_1 = 0.9317769741506936043151654303073883056640625 \]
\[ b_1 = 1.5185 \]

\[ x_2 = 0.844422882824066078910618671216070652008056640625 \]
\[ E_2 = 1.01688061936626457410433320872039209492188542219092968 \]
\[ P_2 = 0.9958243694256508418788315054034553239996718905629355821 \]
\[ b_2 = 5.8884 \]

\[ x_3 = 0.840896884168655820212846605006387710597528718627830956 \]
\[ E_3 = 1.0000022336332328355958387702492100754068879685671957 \]
\[ P_3 = 0.9999999999999999986940807264200713050026299021477654907 \]
\[ b_3 = 57.409 \]

\[ x_4 = 0.8408964152537145441292683119973118637849080485731336497 \]
\[ E_4 = 1.000000000000000005223677094319714797043731882484224637 \]
\[ P_4 = 0.999999999999999999999999999999999999999999999999999833 \]
\[ b_4 = 173.32 \]

\[ 1/\sqrt{2} = 0.8408964152537145430311254762332148950400342623567845108... \]

Figure 28.6-A: Numerical quantities occurring in the computation of \(1/\sqrt{2}\) using a third order Goldschmidt algorithm. The value of \(b_k\) gives the number of correct bits at step \(k\).

As an example, the inverse fourth root of \(d = 2\) can be obtained via the third order algorithm

\[
\begin{align*}
x_0 &= 1 \\
E_0 &= d = 2 \\
E_{k+1} &= E_k P_k^4 = E_k \left( \frac{45 - 18 E_k + 5 E_k^2}{32} \right)^4 \\
x_{k+1} &= x_k P_k
\end{align*}
\]

Figure 28.6-A shows the numerical values of \(x_k\), \(E_k\) and \(P_k\) up to step \(k = 6\). The approximate precision in bits of \(x_k\) is computed as \(b_k = -\log(|1 - E_k|)/\log(2)\).

### 28.7 Products for the \(a\)-th root

Rewrite the well-known product form

\[
\frac{1}{1-y} = (1 + y) (1 + y^2) (1 + y^4) (1 + y^8) \ldots
\]

as

\[
\frac{1}{1-y} = \prod_{k>0} (1 + Y_k) \text{ where } Y_1 := y, \quad Y_{k+1} := Y_k^2
\]

We give product forms for \(a\)-th roots and their inverses that generalize the relations above.
28.7: Products for the $a$-th root

28.7.1 Second order products

Products for the square root and its inverse

For the inverse square root use $1/\sqrt{1-y} = (1+y/2) \cdot 1/\sqrt{1-y^2/4(3+y)}$, thereby

$$\frac{1}{\sqrt{1-y}} = \prod_{k>0} (1 + Y_k) \quad \text{where} \quad Y_1 := \frac{y}{2}, \quad Y_{k+1} := Y_k^2 \left( \frac{3}{2} + Y_k \right) \quad (28.7-3)$$

For the square root use $\sqrt{1-y} = (1-y/2) \cdot \sqrt{1-(y/(y-2))^2}$, so

$$\sqrt{1-y} = \prod_{k>0} (1 + Y_k) \quad \text{where} \quad Y_1 := -\frac{y}{2}, \quad Y_{k+1} := -\frac{1}{2} \left( \frac{Y_k}{1+Y_k} \right)^2 \quad (28.7-4)$$

Products for the $a$-th root and its inverse

The relation for the inverse $a$-th root is

$$\frac{1}{\sqrt[a]{1-y}} = (1-y)^{-1/a} = \prod_{k>0} (1 + Y_k) \quad \text{where}$$

$$Y_1 := \frac{y}{a}, \quad Y_{k+1} := \frac{1}{a} \left( 1 - (1 - a Y_k) (Y_k + 1)^a \right) \quad (28.7-5a)$$

Alternatively,

$$\frac{1}{\sqrt[a]{d}} = x (1-y)^{-1/a} = x \prod_{k>0} (1 + Y_k) \quad (28.7-6)$$

with $y := 1 - d x$ and the definitions [28.7-5b] for $Y_k$. For the $a$-th root we obtain

$$\sqrt[a]{1-y} = (1-y)^{1/a} = \prod_{k>0} (1 + Y_k) \quad \text{where}$$

$$Y_1 := \frac{y}{a}, \quad Y_{k+1} := \frac{1}{a} \left( 1 + a Y_k - (1+Y_k)^a \right) (1+Y_k)^a \quad (28.7-7a)$$

28.7.2 Products of arbitrary order

We want to obtain an $n$-th order product for the inverse $a$-th root

$$\frac{1}{\sqrt[a]{1-y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad Y_1 = y, \quad Y_{k+1} = N(Y_k) \quad (28.7-8)$$

The functions $T$ and $N$ have to be determined. Set

$$[1-y]^{-1/2} = (1+T(Y_1)) \cdot [(1+T(Y_1))^a (1-y)]^{-1/2} =: (1+T(Y_1)) \cdot [1-Y_2]^{-1/2} \quad (28.7-9a)$$

where $1+T(Y_1)$ is the Taylor expansion

$$[1-y]^{-1/2} = 1 + y/a + \frac{(1+a)y^2}{2a^2} + \frac{(1+a)(1+2a)y^3}{6a^4} + \ldots + \frac{\prod_{k=1}^{n-1} (1+k a)}{n! a^n} y^n + \ldots \quad (28.7-10)$$
up to order \( n - 1 \). The Taylor expansion of \( Y_2 \) starts with a term \( \sim y^n \). Using \( Y_{k+1} = N(Y_k) \) as suggested by the relation between \( Y_2 \) and \( Y_1 \) gives a product with \( n \)-th order convergence. For example, for a third order product for \( 1/\sqrt{1-y} \), set

\[
T(Y) := \frac{1}{2} Y + \frac{3}{8} Y^2
\]

Now solve \((1 + T(Y_1)) \cdot (1 - y) = (1 - Y_2)\) for \( Y_2 \) to obtain

\[
Y_2 = \frac{5 y^3}{8} + \frac{15 y^4}{64} + \frac{9 y^5}{64} =: N(y)
\]

Then, finally, \((Y_1 := y\) and

\[
\frac{1}{\sqrt{1-y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where}
\]

\[
T(Y) := \frac{1}{2} Y + \frac{3}{8} Y^2
\]

\[
Y_{k+1} = N(Y_k) := \frac{Y_k^3}{64} (40 + 15 Y_k + 9 Y_k^2)
\]

Replace relation\((28.7-13c)\) by \( Y_{k+1} = 1 - (1 + T(Y_k))^a (1 - Y_k)\) to obtain the general formula for the inverse \( a \)-th root.

The second order products lead to expressions that are especially nice:

\[
\frac{1}{\sqrt{1-y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := +\frac{y}{a} \quad \text{and} \quad (28.7-14a)
\]

\[
Y_{k+1} = N(Y_k) := 1 - (1 + \frac{y}{a})^a (1 - y)
\]

\[
\frac{1}{\sqrt{1+y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := -\frac{y}{a} \quad \text{and} \quad (28.7-15a)
\]

\[
Y_{k+1} = N(Y_k) := (1 - \frac{y}{a})^a (1 + y) - 1
\]

\[
\sqrt[3]{1-y} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := -\frac{y}{a} \quad \text{and} \quad (28.7-16a)
\]

\[
Y_{k+1} = N(Y_k) := \frac{(a - Y_k)^a - (1 - Y_k) a^a}{(a - Y_k)^a} = 1 - \frac{(1 - Y_k) a^a}{(a - Y_k)^a}
\]

\[
\sqrt[3]{1+y} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := +\frac{y}{a} \quad \text{and} \quad (28.7-17a)
\]

\[
Y_{k+1} = N(Y_k) := \frac{(1 + Y_k) a^a - (a + Y_k)^a}{(a + Y_k)^a} = \frac{(1 + Y_k) a^a}{(a + Y_k)^a} - 1
\]

The third order product for \( \frac{1}{\sqrt{1-y}} \) is

\[
\frac{1}{\sqrt{1-y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) = +\frac{y}{a} + \frac{y^2 (1 + a)}{2 a^2} \quad \text{and} \quad (28.7-18a)
\]

\[
Y_{k+1} = N(Y_k) := 1 - (1 + \frac{y}{a} + \frac{y^2 (1 + a)}{2 a^2})^a (1 - y)
\]

\[
= 1 - (1 + T(y))^a (1 - y)
\]
28.7.3 Third order product for the $a$-th root

\begin{verbatim}
\> \texttt{\textcopyright{} default(\texttt{realpreci...}
\> \texttt{\textcopyright{} a=3;d=2;al=a-1;be=a+1;
\> \texttt{\textcopyright{} F(x)=(al*x^a+be*d)/(be*x^a+al*d) \ \ == \ (x^3 + 4)/(2*x^3 + 2)\n\> \texttt{\textcopyright{} p=99.0; \ \ \ \ \ \ \ \ \ \ very \ bad \ approximation \ to \ the \ root\n\> \texttt{\textcopyright{} for(k=0,25,p*=F(p);print(" \ \ \ \ \ \ \ \ \ ");
\> \texttt{\textcopyright{} 49.500153045449860867728537657013294857260641038853963
\> \texttt{\textcopyright{} 24.75068686653235797325980776085903190585296493460
\> \texttt{\textcopyright{} 12.377792780128382589873092283828965637377202554041925
\> \texttt{\textcopyright{} 6.19868172946797330989039489353983190191173732016583057
\> \texttt{\textcopyright{} 3.1382160966257716458713612717562126960309193786721790
\> \texttt{\textcopyright{} 1.716643430397499234556143236039595682000253452248
\> \texttt{\textcopyright{} 1.283328514322784323283037711640101526459592858280490032
\> \texttt{\textcopyright{} 1.25992628457115327949135992475385658266816159208667105
\> \texttt{\textcopyright{} 1.25992104989487325007759793665422076373275039651898
\> \texttt{\textcopyright{} 1.259921049894873164767210607278226350570251464701509790
\> \texttt{\textcopyright{} 1.259921049894873164767210607278226350570251464701507980
\> \texttt{\textcopyright{} 1.259921049894873164767210607278226350570251464701507980
\> \texttt{\textcopyright{}
\end{verbatim}

Figure 28.7-A: Computation of $\sqrt[3]{2}$ with a very bad initial approximation.

The third order iteration given as relation \ref{28.2.9} on page \pageref{28.2.9} gives a simple product for $\sqrt[3]{d}$. Let

\begin{equation}
P_k := \prod_{j=0}^{k} Y_k \quad \quad (28.7-19a)
\end{equation}

where $Y_0$ is sufficiently near to $\sqrt[3]{d}$ and $Y_k = F(P_{k-1})$ where

\begin{equation}
F(x) := \frac{\alpha x^a + \beta d}{\beta x^a + \alpha d} \quad \quad (28.7-19b)
\end{equation}

with $\alpha = a - 1$ and $\beta = a + 1$. Then $P_\infty = \sqrt[3]{d}$. Figure 28.7-A shows the numerical quantities with the computation of $\sqrt[3]{2}$ with a starting value $Y_0 = 99$ that is not at all close to the root. We have $F(x) = x^3 + 4$ which is $\approx \frac{1}{2}$ for large values of $x$. Therefore the big initial values are repeatedly halved before the third order convergence begins.

28.8 Divisionless iterations for polynomial roots

Let $f(x)$ be a polynomial with simple roots only, then

\begin{equation}
\Phi(x) := x - p(x)f(x) \quad \quad \text{where} \quad p(x) := f'(x)^{-1} \mod f(x)
\end{equation}

is a second order iteration for the roots of $f(x)$. The iteration involves no long division if all coefficients are small rationals. Instead of dividing by $f'(x)$ a multiplication by the modular inverse $p(x)$ is used. As deg($p$) < deg($f$) we have deg($\Phi$) $\leq$ deg($f$) - 1.

For example, for $f(x) = ax^2 + bx + c$ we obtain

\begin{equation}
\Phi(x) = x - \frac{2ax + b}{\Delta} f(x) \quad \quad \text{where} \quad \Delta = b^2 - 4ac
\end{equation}

The general expressions for polynomials of orders $> 2$ get complicated. However, for fixed polynomial coefficients the iteration is more manageable. For example, with $f(x) = x^3 + 5x + 1$ we obtain

\begin{equation}
\Phi(x) = x + \frac{f(x)}{527} (30x^2 - 9x + 100)
\end{equation}

For the polynomial $x^n - d$ we have $p = x/(nd)$ and the iteration is (relation \ref{28.3.16} on page \pageref{28.3.16}:

\begin{equation}
\Phi(x) = x - \frac{x}{nd} (x^n - d) = x + \frac{1}{n} \left( x - \frac{x^{n+1}}{d} \right)
\end{equation}
The construction is given in [150] where a method to construct divisionless iterations of arbitrary order is given: let \( p f' + q f = 1 \), and

\[
\Phi_1 := x - p_1 f, \quad p_1 := p \\
p_r := pp'_r - (r - 1) q p_{r-1} \\
\Phi_r := \Phi_{r-1} + (-1)^r p_r f^r / r!
\]

then \( \Phi_r \) is an iteration of order \( r + 1 \).
Chapter 29

Iterations for the inversion of a function

We study some general expressions for iterations for the zero of a function. Two schemes that give arbitrary order one-point iterations, Householder’s formula and Schröder’s formula, are given. Several methods to construct alternative iterations are described. Moreover, iterations that also converge for multiple roots and a technique to turn a linear iteration into a super-linear one are presented.

29.1 Iterations and their rate of convergence

An iteration for a zero $r$ (or root, $f(r) = 0$) of a function $f(x)$ can be given as a function $\Phi(x)$ that, when used like

$$x_{k+1} = \Phi(x_k)$$

will make $x_k$ converge towards the root: $x_\infty = r$. Convergence is subject to the condition that $x_0$ is close enough to $r$. The function $\Phi(x)$ must (and can) be constructed such that it has an attracting fixed point where $f(x)$ has a zero:

$$\Phi(r) = r \quad \text{(fixed point)}$$

$$|\Phi'(r)| < 1 \quad \text{(attracting)}$$

This type of iteration is called a one-point iteration. There are also multi-point iterations, these are of the form $x_{k+1} = \Phi(x_k, x_{k-1}, \ldots, x_{k-j}), j \geq 1$. An example is the two-point iteration known as the secant method

$$x_{k+1} = \Phi(x_k, x_{k-1}) = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

We are mainly concerned with one-point iterations in what follows.

The order of convergence (or simply order) of a given iteration can be defined as follows: let $x = r \cdot (1+e)$ with $|e| \ll 1$ and $\Phi(x) = r \cdot (1 + \alpha e^n + O(e^{n+1}))$, then the iteration $\Phi$ is called linear (or first order) if $n = 1$ (and $|\alpha| < 1$). A linear iteration improves the result by (roughly) adding a constant amount of correct digits with every step.

A super-linear iteration does better than that: The number of correct digits grows exponentially (to the base $n$) at each step. Super-linear convergence of order $n$ should better be called exponential of order $n$. 

[fxtbook draft of 2008-August-17]
Chapter 29: Iterations for the inversion of a function

Iterations of second order \((n = 2)\) are often called quadratic (or quadratically convergent), those of third order cubic iterations. Fourth, fifth and sixth order iterations are called quartic, quintic and sextic and so on. We note that the two-point iteration relation (29.1-4) has order \((\sqrt{5}+1)/2 \approx 1.618\), see [151, p.152].

It is conceivable to find iterations that do converge better than linear but less than exponential to any base: imagine an iteration that produces proportional \(k^2\) digits at step \(k\) (this is not quadratical convergence which produces proportional \(2^k\) correct digits at step \(k\)). That case is not covered by the ‘order-\(n\)’ notion just introduced. However, those (super-linear but sub-exponential) iterations are not usually encountered and we actually won’t meet one. In fact, the constructions used here cannot produce such an iteration. For a more fine-grained definition of the concept of order see [64, p.21].

For \(n \geq 2\) the iteration function \(\Phi\) has a super-attracting fixed point at \(r\): \(\Phi'(r) = 0\). For an iteration of order \(n\) one has

\[
\Phi'(r) = 0, \quad \Phi''(r) = 0, \quad \ldots, \quad \Phi^{(n-1)}(r) = 0
\] (29.1-5)

There is no standard term for emphasizing the number of derivatives vanishing at the fixed point: super-attracting of order \(n\) might be appropriate.

To any iteration of order \(n\) for a function \(f\) one can add a term \(f(x)^n \cdot \varphi(x)\) (where \(\varphi(x)\) is an arbitrary function that is analytic in a neighborhood of the root) without changing the order of convergence. That term is assumed to be zero in what follows. The statement can easily be checked by verifying that the first \(n-1\) derivatives of \(\Phi_n(x) + f(x)^n \cdot \varphi(x)\), evaluated at the root \(r\), equal zero.

Any two one-point iterations of the same order \(n\) differ by a term \(f(x)^n \cdot \varphi(x)\).

Any two iterations of the same order \(n\) differ by a term \((x-r)^n \nu(x)\) where \(\nu(x)\) is a function that is finite at \(r\) [151, p.174, ex.3].

Any one-point iteration of order \(n\) must explicitly evaluate \(f, f', \ldots, f^{(n-1)}\) [260, p.98]. For methods to find zeros and extrema without evaluating derivatives see [64].

### 29.2 Schröder’s formula

For \(n \geq 2\) the expression

\[
S_n(x) := x + \sum_{k=1}^{n-1} (-1)^k \frac{f(x)^k}{k!} \left( \frac{1}{f'(x)} \frac{\partial}{\partial x} \right)^{k-1} \frac{1}{f'(x)}
\] (29.2-1)

gives a \(n\)-th order iteration for a (simple) root \(r\) of \(f\) [239, p.13]. That is,

\[
S := S_\infty(x) = x - \frac{f}{1! f'} - \frac{f^2}{2! f'^2} \cdot f'' - \frac{f^3}{3! f'^3} \cdot (3f''^2 - f' f''')
\] (29.2-2)

\[
- \frac{f^4}{4! f'^4} \cdot (15f'''^2 - 10f' f'' f''' + f'^2 f''')
\]

\[
- \frac{f^5}{5! f'^5} \cdot (105f'''^4 - 105f' f'''^2 f'' - 10f'^2 f'' f''' + 15f'^2 f'' f'''' - f'^3 f''''') - \ldots
\]

The second order iteration is the Newton iteration. The third order iteration, obtained upon truncation after the third term on the right hand side, and written as

\[
S_3 = x - \frac{f}{f'} \left( 1 + \frac{f f''}{2f'^2} \right)
\] (29.2-3)
29.3: Schröder’s formula and series reversion

is sometimes referred to as Householder’s method. Approximating the second term on the right hand side gives Halley’s formula:

\[ H_3 = x - f f' \left( 1 - \frac{f''}{2 f'^2} \right)^{-1} \]  

(29.2-4)

Write

\[ S = x - U_1 \frac{f}{f'} - U_2 \frac{f^2}{2 f'^3} - U_3 \frac{f^3}{3! f'^5} - \ldots - U_n \frac{f^n}{n! f'^{2n-1}} - \ldots \]  

(29.2-5)

then \( U_1 = 1, U_2 = f'', U_3 = 3 f'^2 - f' f''', \) and we have the recursion (see [239, p.16] or [151, p.148])

\[ U_n = (2n - 3) f'' U_{n-1} - f' U_{n-1}' \]  

(29.2-6)

An alternative recursion is given in [260, p.83], write

\[ S = x - Y_1 \frac{f}{f'} - Y_2 f \left( \frac{f}{f'} \right)^2 - Y_3 f f' \left( \frac{f}{f'} \right)^3 - \ldots - Y_n f f' \left( \frac{f}{f'} \right)^n - \ldots \]  

(29.2-7)

then \( Y_1 = 1 \) and

\[ Y_n = \frac{1}{n} \left( 2 (n - 1) \frac{f''}{2 f'} Y_{n-1} - Y_{n-1}' \right) \]  

(29.2-8)

Relation 29.2-1 with \( f(x) = 1/x^a - d \) gives the division-free iteration 28.3-5 on page 575 for arbitrary order. For \( f(x) = \log(x) - d \) one obtains the iteration 31.2-9 on page 633. For \( f(x) = x^2 - d \) one obtains

\[ S(x) = x - \left( \frac{x^2 - d}{2x} + \frac{(x^2 - d)^2}{8 x^3} + \frac{(x^2 - d)^3}{16 x^5} + \frac{5 (x^2 - d)^4}{128 x^7} + \ldots \right) \]  

(29.2-9a)

\[ = x - 2x \cdot (Y + 2 Y^2 + 5 Y^3 + 14 Y^4 + 42 Y^5 + \ldots) \quad \text{where} \quad Y := \frac{x^2 - d}{(2x)^2} \]  

(29.2-9b)

The coefficients of the powers of \( Y \) are the Catalan numbers, see section 13.3 on page 324.

29.3 Schröder’s formula and series reversion

We give three ways of obtaining Schröer’s iteration via (implicit or explicit) power series reversion. The reversion of a series

\[ A(x) = \sum_{k=1}^{\infty} a_k x^k \]  

is the series

\[ B(x) = \sum_{k=1}^{\infty} b_k x^k \]  

(29.3-2)

such that \( A(B(x)) = x \). That is, \( B(x) = A^{-1}(x) \) is the functional inverse of \( A(x) \) (reversion is inversion with respect to composition). Note that \( A^{-1}(x) \) is not the same as \( A^{-1}(x) = 1/A(x) \). A useful relation is given in [104, p.634]:

\[ A^{-1}(x) = \sum_{k=1}^{\infty} \frac{x^k}{k!} \left[ \left( \frac{\partial}{\partial x} \right)^{k-1} \left( \frac{x}{f(x)} \right)^k \right]_{x=0} \]  

(29.3-3)
Equivalently, from [172] p.527,

\[ kb_k = [k - 1] \left( \frac{x}{A(x)} \right)^k \]  
(29.3-4)

where \([k - 1] Q\) denotes the \((k - 1)\)-st series coefficient of \(Q\). We use the expression to give a few terms of the reversed series explicitly:

\[
\begin{align*}
\text{? } n=5; \ R=O(x^{n+1}); \\
\text{? } A=\text{sum}(k=1,n,x^k*\text{eval(Str("a"*k))})+R \\
\text{? } A=\text{sum}(k=1,n,x^k*(\text{polcoeff(truncate((x/A)^k),k-1)})+R \\
+ 1/a1 * x \\
+ (-a2/a1^3) * x^2 \\
+ ((-a3*a1 + 2*a2^2)/a1^5) * x^3 \\
+ ((-a4*a1^2 + 5*a3*a2*a1 - 5*a2^3)/a1^7) * x^4 \\
+ ((-a5*a1^3 + (4*a4*a2 + 3*a3^2)*a1^2 - 21*a3*a2^2*a1 + 14*a2^4)/a1^9) * x^5 + O(x^6)
\end{align*}
\]

The same result is obtained with pari/gp’s builtin function `serreverse()`. Relation [29.3-4] can be generalized for the \(n\)-th power of the reversed series:

\[ [k] (B(x))^n = \frac{n}{k} [k - n] \left( \frac{x}{A(x)} \right)^k \]  
(29.3-5)

This is one way to state the Lagrange inversion formula.

### 29.3.1 Method of deriving the inverse function

The starting point is the Taylor series of a function \(f\) around \(x_0\):

\[ f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} f^{(k)}(x_0) (x - x_0)^k \]  
(29.3-6a)

\[ = f(x_0) + f'(x_0) (x - x_0) + \frac{1}{2} f''(x_0) (x - x_0)^2 + \frac{1}{6} f'''(x_0) (x - x_0)^3 + \ldots \]  
(29.3-6b)

Now let \(f(x_0) = y_0\) and \(r\) be the zero of \(f\) (that is, \(f(r) = 0\)). We expand the inverse function \(g = f^{-1}\) around \(y_0\):

\[ g(0) = \sum_{k=0}^{\infty} \frac{1}{k!} g^{(k)}(y_0) (0 - y_0)^k \]  
(29.3-7a)

\[ = g(y_0) + g'(y_0) (0 - y_0) + \frac{1}{2} g''(y_0) (0 - y_0)^2 + \frac{1}{6} g'''(y_0) (0 - y_0)^3 + \ldots \]  
(29.3-7b)

Using \(x_0 = g(y_0)\) and \(g(0) = r\) we obtain

\[ r = x_0 - g'(y_0) f(x_0) + \frac{1}{2} g''(y_0) f(x_0)^2 - \frac{1}{6} g'''(y_0) f(x_0)^3 + \ldots \]  
(29.3-8)

Remains to express the derivatives of the inverse \(g\) in terms of (derivatives of) \(f\). Set

\[ f \circ g = \text{id}, \quad \text{that is: } f(g(x)) = x \]  
(29.3-9)

and derive the equation (chain rule) to obtain \(g'(f(x)) f'(x) = 1\), so \(g'(y) = \frac{1}{f'(y)}\). Derive \(f(g(x)) - x\) multiple times to obtain (arguments \(y\) of \(g\) and \(x\) of \(f\) are omitted for readability):

\[
\begin{align*}
1 &= f'g' \\
0 &= g'f'' + f'^2 g'' \\
0 &= g'f''' + 3f'f''g'' + f'^3 g''' \\
0 &= g'f'''' + 4f'g''f''' + 3f'^2 g'' + 6f'^2 f''g''' + f'^3 g'''
\end{align*}
\]  
(29.3-10a-d)
This system of linear equations in the derivatives of $g$ can be solved successively for $g'$, $g''$, $g'''$, etc.:

\[
g' = \frac{1}{f'} \quad (29.3-11a)
\]
\[
g'' = -\frac{f''}{f'^3} \quad (29.3-11b)
\]
\[
g''' = \frac{1}{f'^5} \left(3f'^2 - f'f''\right) \quad (29.3-11c)
\]
\[
g'''' = \frac{1}{f'^7} \left(10f'f''f''' - 15f'^3 - f'^2f''''\right) \quad (29.3-11d)
\]
\[
g''''' = \frac{1}{f'^9} \left(105f'^4 - f'^3f'''' - 15f'^6 + 15f'^5f'' + 10f'^2f''''\right) \quad (29.3-11e)
\]

Thereby equation 29.3-8 can be written as

\[
r = x - \frac{1}{f'} + \frac{1}{2} \left(-\frac{f''}{f'^3}\right) f^2 - \frac{1}{6} \left(\frac{1}{f'^5} \left(3f'^2 - f'f''\right)\right) f^3 + \ldots \quad (29.3-12)
\]

which is Schröder’s iteration, equation 29.2-2 on page 592.

29.3.2 Method of reversing power series

Schröder’s formula can be obtained as the reversion of the series

\[
E(W) := -\sum_{k=1}^{\infty} \frac{f(k)}{k!} (-W)^k = [1 - \exp(-W \partial) \circ f] \quad (29.3-13)
\]

Let $L(W)$ be the reversion of $E(W)$, then $x - L(f)$ is Schröder’s iteration:

\[
\begin{align*}
\text{let } L &= \text{serreverse}(E) \\
\text{for }(j=1,n, \text{print}(-\text{polcoeff}(L,j,x)*(f0)^j))
\end{align*}
\]

Here we use ‘fk’ as a symbol for the $k$-th derivative of $f$.

29.3.3 Method of writing power series as operator functions

Write the power series of the function $f$ symbolically as

\[
T(f) = \sum_{k=0}^{\infty} \frac{f(k)}{k!} x^k = \left[\exp \left(\frac{+h \partial}{\partial x}\right) \circ f\right]_{h=x} \quad (29.3-14)
\]

In this notation Schröders formula becomes

\[
S(x) = \left[\exp \left(-\frac{h \partial}{\partial f}\right) \circ x\right]_{h=f} \quad (29.3-15)
\]
First expand as a series

\[ S(x) = \sum_{k=0}^{\infty} \left( -h \frac{\partial f}{\partial f} \right)^k x \bigg|_{h=f} = \sum_{k=0}^{\infty} \frac{(-f)^k}{k!} \left( \frac{\partial}{\partial f} \right)^k x \] (29.3-16)

Now use

\[ \frac{\partial}{\partial f} = \frac{\partial x}{\partial f} \frac{\partial}{\partial x} = 1 \] (29.3-17)

and separate the term for \( k = 0 \) to obtain

\[ S(x) = x + \sum_{k=1}^{\infty} \frac{(-f)^k}{k!} \left( \frac{1}{f'} \frac{\partial}{\partial x} \right)^k x = x + \sum_{k=1}^{\infty} \frac{(-f)^{k-1}}{k!} \left( \frac{1}{f'} \frac{\partial}{\partial x} \right)^{k-1} \frac{1}{f'} \] (29.3-18)

Truncation gives relation 29.2-1.

### 29.4 Householder’s formula

The following expression gives for \( n \geq 2 \) a \( n \)-th order iteration for a (simple) root \( r \) of \( f \) [151, p.169]:

\[ H_n(x) := x + (n-1) \left( \frac{1}{f'} \right)^{(n-2)} \left( \frac{1}{f'} \right)^{(n-1)} \] (29.4-1)

We refer to iterations of this type as Householder iterations, the name König iteration function is used in [266]. We have

\[ H_2 = x - \frac{f}{f'} \] (29.4-2a)

\[ H_3 = x - \frac{2ff'}{2f'^2 - ff''} \] (29.4-2b)

\[ H_4 = x - \frac{3f(f'' - 2f'^2)}{6f'^2f'' - 6f'^3 - 2f^3} \] (29.4-2c)

\[ H_5 = x + \frac{4f \left( 6f'^3 - 6ff'' + f^2f''' \right)}{f^3f''' - 24f'^4 + 36f'^2f'' - 8f^2f'f''' - 6f^2f''^2} \] (29.4-2d)

The second order variant is Newton’s formula, the third order iteration is Halley’s formula.

Kalantari and Gerlach [159] define the iteration

\[ B_m = x - f \frac{D_{m-2}}{D_{m-1}} \] (29.4-3a)

where \( m \geq 2 \), \( D_0 = 1 \), \( D_1 = f' \), and

\[ D_m = \det \begin{pmatrix} f' & f'' & \cdots & f^{(m-1)} & f^{(m)} \\ f & f' & \cdots & f^{(m-1)} & f^{(m)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & f & \cdots & f^{(m-1)} & f'' \\ 0 & 0 & \cdots & f & f' \end{pmatrix} \] (29.4-3b)
The iteration turns out to be identical to the one of Householder ($B_n = H_n$). A recursive definition for $D_m$ is given by

$$D_m = \sum_{i=1}^{m} (-1)^{i-1} f^{(i)}(r) / i! \cdot D_{m-i} \quad (29.4-4)$$

The derivation of Halley’s formula by applying Newton’s formula to $f/\sqrt{f}$ can be generalized to produce $m$-order iterations as follows: Let $F_1 = f$ and for $m \geq 2$ let

$$F_m = \frac{F_{m-1}}{\sqrt{F'_{m-1}}} \quad (29.4-5a)$$

$$H_m = x - \frac{F_{m-1}}{F'_{m-1}} \quad (29.4-5b)$$

This recursion indeed gives the Householder iteration, as shown in [159].

An alternative recursive formulation (also given in [159], ascribed to Ford/Pennline) is

$$Q_2 = 1 \quad (29.4-6a)$$

$$Q_m = f' Q_{m-1} - \frac{1}{m-2} f Q'_{m-1} \quad (29.4-6b)$$

$$H_m = x - f \frac{Q_m}{Q_{m-1}} \quad (29.4-6c)$$

The Taylor series of the $k$-th order Householder iteration around $f = 0$ up to order $k-1$ gives the $k$-th order Schröder iteration.

An extraneous fixed point of an iteration for a function $f$ is a fixed point at $z$ such that $f(z) \neq 0$. All extraneous fixed points for the iterations $H_n$ are repelling ($|H(z)'| > 1$), see [160] and [266].

### 29.5 Dealing with multiple roots

The iterations given so far will not converge at the stated order if $f$ has a multiple root at $r$. As an example consider the function

$$f(x) = (x^2 - d)^m \quad \text{where} \quad m \in \mathbb{N}_+ \quad (29.5-1)$$

The iteration $\Phi(x) = x - f/f'$ is

$$\Phi(x) = x - \frac{x^2 - d}{m 2x} \quad (29.5-2)$$

Its convergence is only linear for $m > 1$: $\Phi(\sqrt{d}(1 + e)) = \sqrt{d}(1 + \frac{m-1}{m} e + O(e^2))$.

Householder [151, p.161, ex.6] gives a second order iteration for a root of known multiplicity $m$ as

$$\Phi_2(x) = x - m \cdot \frac{f}{f'} \quad (29.5-3)$$

Note that with the example above we obtain a quadratic iteration.

For roots of unknown multiplicity use the general expressions for iterations with $F := f/f'$ instead of $f$. Both $F$ and $f$ have the same set of roots, but all roots of $F$ are simple. To illustrate this consider a function $f$ that has a root of multiplicity $m$ at $r$: $f(x) := (x - r)^m h(x)$ with $h(r) \neq 0$. Then

$$f'(x) = m \cdot (x - r)^{m-1} h(x) + (x - r)^m h'(x) \quad (29.5-4a)$$

$$= (x - r)^{m-1} \left( m h(x) + (x - r) h'(x) \right) \quad (29.5-4b)$$
The iteration which holds independent of

\[ F(x) = \frac{f(x)}{f'(x)} = \frac{x - r}{m h(x) + (x - r) h'(x)} \]  

(29.5-5)

The fraction on the right hand side does not vanish at the root \( r \).

Plugging \( F = f / f' \) into Householder’s formula (relation 29.4-1) we get the following iterations denoted by \( H_k^\% \), the iterations \( H_k \) are given for comparison:

\[
H_2 = x - \frac{f f'}{f'^2 - f f''}
\]  

(29.5-6a)

\[
H_3^\% = x - \frac{2 f f'}{2 f'^2 - f f''}
\]  

(29.5-6b)

\[
H_3 = x - \frac{f f'}{f'^2 - f f''}
\]  

(29.5-6c)

\[
H_4^\% = x + \frac{2 f f'' - 2 f f'^2}{2 f'^3 - 3 f f' f'' + f^2 f'''}
\]  

(29.5-6d)

\[
H_4 = x + \frac{3 f^2 f'' - 6 f f'^2}{6 f'^3 - 6 f f' f'' + f^2 f'''}
\]  

(29.5-6e)

\[
H_5^\% = x + \frac{6 f f'^3 + 3 f^3 f''' - 9 f^2 f' f''}{f'^3 f''' - 6 f f'^4 + 12 f f'^2 f'' - 4 f^2 f' f''' - 3 f^2 f'^2 f'''}
\]  

(29.5-6f)

\[
H_5 = x + \frac{24 f f'^5 + 4 f^3 f''^3 - 24 f^2 f' f''^2}{f'^3 f''' - 24 f f'^4 + 36 f f'^2 f'' - 8 f^2 f' f''' - 6 f^2 f'^2 f'''}
\]  

(29.5-6g)

The terms in the numerators and denominators of \( H_k^\% \) and \( H_{k+1} \) are identical up to the integral constants.

The iteration \( H_k^\% \) can also be written as

\[
H_k^\% = x + (k - 1) \frac{(\log(f))^{(k-1)}}{(\log(f))^{(k)}}
\]  

(29.5-7)

Schröder’s formula (relation 29.2-1), when inserting \( f / f' \), becomes:

\[
S^\% = x - \frac{f f'}{(f'^2 - f f'')} - \frac{f^2 f'(f'^3 f''' + f'' f'^2 - f f''^2)}{2(f f'' - f'^2)^3} - \frac{6(f f'' - f'^2)^3}{f^3 f'(f'^3 f''' + f'' f'^2 - f f''^2)} - \frac{24(f f'' - f'^2)^4}{f^4 f'(f'^3 f''' + f'' f'^2 - f f''^2)} - \cdots - \frac{k!}{(f f'' - f'^2)^{2k-1}}
\]  

(29.5-8)

We check the convergence with our example (relation 29.5-1), the second order iteration is

\[
\Phi_2^\%(x) = S_2 = H_2 = x + \frac{d - x^2}{d + x^2} = \frac{2d x}{x^2 + d}
\]  

(29.5-9)

Convergence is indeed second order, as we have (compare to relation 28.2-7 on page 573)

\[
\Phi_2^\%(\sqrt{d} \cdot \frac{1 - e}{1 + e}) = \sqrt{d} \cdot \frac{1 - e^2}{1 + e^2}
\]  

(29.5-10)

which holds independent of \( m \). In general, with \( H_k^\% \) one obtains for the square root:

\[
\Phi_k^\%(\sqrt{d} \cdot \frac{1 - e}{1 + e}) = \sqrt{d} \cdot \frac{1 - e^k}{1 + e^k}
\]  

(29.5-11)
Using Schröder’s third order formula for \( f/f' \) with \( f \) as in \((29.5-1)\) we obtain a beautiful fourth order iteration for \( \sqrt{d} \):

\[
S_3^A(x) = x + x \frac{d - x^2}{d + x^2} + x^2 \frac{(d - x^2)^2}{(d + x^2)^3} \tag{29.5-12a}
\]

\[
S_3^A(\sqrt{d} \frac{1 - c}{1 + c}) = \sqrt{d} \frac{1 + 3e^2 - 3e^4 - e^6}{1 + 3e^2 + 3e^4 + e^6} \tag{29.5-12b}
\]

\[
= \sqrt{d} \frac{1 - c}{1 + c} \text{ where } c = e^4 \frac{e^2 + 3}{3e^2 + 1} \tag{29.5-12c}
\]

In general, the \((1 + \alpha k)\)-th order Schröder iteration for \( 1/\sqrt{d} \) obtained through \( f/f' \) has an order of convergence that exceeds the expected order by one. The third order Schröder iteration for \( f(x) = 1 - dx^2 \) is

\[
S_3^A(x) = x + x \frac{1 - dx^2}{1 + dx^2} + x^2 \frac{(1 - dx^2)^2}{(1 + dx^2)^3} \tag{29.5-13}
\]

The iteration also has 4th order convergence and the error expression \( S_3^A(\sqrt{d} \frac{1 - c}{1 + c}) \) is the same (replacing \( \sqrt{d} \) by \( 1/\sqrt{d} \)) as in relation \((29.5-12b)\)

### 29.6 More iterations

We give expressions for iterations via Padé approximants, radicals, and show how iterations can be obtained from given ones. Finally we give one form of a multi-point iteration.

#### 29.6.1 Rational iterations from Padé approximants

The \([i, j]\)-th Padé approximant of \( \Phi \) in \( f \) gives an iteration of order \( p = i + j + 1 \) (if \( n \geq p \)). Write \( P_{[i,j]} \) for an iteration (of order \( i + j + 1 \)) that is obtained using the Padé approximant \([i,j]\]. For the second order (where the Newton iteration is \( P_{[1,0]}(x) = x - f/\dot{f} \)) this method gives one alternative form, namely

\[
P_{[0,1]}(x) = x^2 \frac{f'}{f + x f'} = x - \frac{xf}{f + x f'} = x - \frac{xf}{(xf')} = x \left(1 + \frac{f}{xf'}\right)^{-1} \tag{29.6-1}
\]

For the third order we find \( P_{[2,0]}(x) = S_3(x) \), \( P_{[1,1]}(x) = H_3(x) \), and

\[
P_{[0,2]}(x) = \frac{2x^3 f'^3}{2f^2 f' + 2xf f'^2 + x^2 f'^3 + 2x^2 f'^3} \tag{29.6-2a}
\]

\[
= x - \frac{xf \left(2ff' + xff'' + 2x^2 f'^3\right)}{(2f^2 f' + 2xf f'^2 + x^2 f'^3 + 2x^2 f'^3)} \tag{29.6-2b}
\]

\[
= x \left(1 + \frac{f}{xf'} + \frac{f^2}{x^2 f'^2} + \frac{ff''}{2x f'^3}\right)^{-1} \tag{29.6-2c}
\]

\[
= x \left(1 + \frac{f}{xf'} + \frac{f^2 (x^2 f''')}{2(x f')^3} + \frac{f^2}{(xf')^2}\right)^{-1} \tag{29.6-2d}
\]

Alternatively one can use the Padé approximant \( A_{[i,j]} \) of \((\Phi(x) - x)/f \) in \( f \) where \( \Phi(x) \) is a given iteration
of order $\geq i + j + 2$. Then $\Phi_{[i,j]}^+ := x + f \cdot A_{[i,j]}$ is an iteration which has order $n = i + j + 2$.

$$\Phi_{[0,1]}^+(x) = x - \frac{2ff'}{2f^2 - f'f''} = H_3(x) \quad (29.6-3a)$$

$$\Phi_{[1,0]}^+(x) = x - \frac{f (f'' + 2f'^2)}{2f'^3} = S_3(x) \quad (29.6-3b)$$

The iterations $\Phi^+$ of order $n$ are expressions in $x, f, f', \ldots, f^{(n-1)}$. Fourth order iterations are

$$\Phi_{[2,0]}^+(x) = x - \frac{f \left(6f'^4 + 3f f'^2 f'' - f'^2 f'^3 + 3f'^2 f'^2 f'' \right)}{6f'^5} \quad (29.6-4a)$$

$$\Phi_{[1,1]}^+(x) = x - \frac{f \left(2ff' f'' - 3f f'^2 + 6f'^2 f'' \right)}{f' \left(2ff' f'' - 6f'^2 f'' + 6f'^2 f'' \right)} \quad (29.6-4d)$$

$$\Phi_{[0,2]}^+(x) = x - \frac{12ff'^3}{(6f'^4 - 6f f'^2 f'' + 2f'^2 f'^3 - 3f'^2 f'^2 f''}) \quad (29.6-4e)$$

$$\Phi_{[1,1]}^+(x) = x - \left(\frac{f'}{f} - \frac{f''}{2f'} - \frac{f \left(3f'^2 - 2f'^2 f'' \right)}{12f'^3} \right)^{-1} \quad (29.6-4f)$$

The iteration $\Phi_{[n,0]}^+$ always coincides with Schröder’s iteration. In general one obtains $n - 1$ additional forms of iterations using the approximants $[0, n - 2], [1, n - 3], \ldots, [n - 3, 1]$.

Neglecting terms that contain the third derivative in relation 29.6-4d we obtain the third order iteration

$$\Phi_3 = x - \frac{f \left(2f'^2 - f f'' \right)}{f' \left(2f'^2 - 2f f'' \right)} \quad (29.6-5)$$

A closed form for the Padé approximants of the $r$-th root is given in relation 28.2-13 on page 574.

### 29.6.2 An iteration involving radicals

By directly solving the truncated Taylor expansion

$$f(r) = f(x) + f'(x) (r - x) + \frac{1}{2} f''(x) (r - x)^2$$

of $f(r) = 0$ around $x$ one obtains the following third order iteration:

$$\Phi_3 = x - \frac{1}{f''} \left(f' \pm \sqrt{f'^2 - 2f f''} \right) = x - \frac{f'}{f''} \left(1 \pm \sqrt{1 - \frac{2f f''}{f'^2}} \right) \quad (29.6-7)$$

For $f(x) = ax^2 + bx + c$ this gives the two solutions of the quadratic equation $f(x) = 0$; for other functions one obtains an iterated square root expression for the roots.
The following form, given in [260, p.94], avoids possible cancellation:

\[
\Phi_3 = x - \frac{2u}{1 + \sqrt{1 - 4Au}} \quad \text{where} \quad u = f/f' \quad \text{and} \quad A = f''/(2f) \quad (29.6-8)
\]

It can be obtained by observing that

\[
\frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \quad (29.6-9)
\]

### 29.6.3 Iterations from iterations

Alternative rational forms can also be obtained in a way that generalizes the method used for multiple roots: if we emphasize the so far notationally omitted dependency from the function \(f\) as \(\Phi\{f\}\). The iteration \(\Phi\{f\}\) has fixed points where \(f\) has a root \(r\), so \(x - \Phi\{f\}\) again has a root at \(r\). Hence we can build more iterations that will converge to those roots as \(\Phi\{x - \Phi\{f\}\}\). For dealing with multiple roots we used \(\Phi\{x - \Phi_2\{f\}\}\). An iteration \(\Phi_k\{x - \Phi_j\{f\}\}\) can only be expected to have a \(k\)-th order convergence.

### 29.6.4 A multi-point iteration

A multi-point iteration of order \(r\) can be given [260, p.165] as

\[
\Phi_r(x) = \Phi_{r-1}(x) - \frac{f(\Phi_{r-1}(x))}{f'(x)} \quad (29.6-10)
\]

where \(\Phi_{r-1}(x)\) is an iteration of order \(r - 1\). For example, choose \(\Phi_2(x) = x - f(x)/f'(x)\) to obtain the third order iteration

\[
\Phi_3(x) = \Phi_2(x) - \frac{f(\Phi_2(x))}{f'(x)} \quad (29.6-11)
\]

Apply the method again to obtain

\[
\Phi_4(x) = \Phi_3(x) - \frac{f(\Phi_3(x))}{f'(x)} = \Phi_2(x) - \frac{f(\Phi_2(x))}{f'(x)} - \frac{f(\Phi_3(x))}{f'(x)} \quad (29.6-12)
\]

The \(r\)-th order iteration is

\[
\Phi_r(x) = \Phi_2(x) - \sum_{k=2}^{r-1} \frac{f(\Phi_k(x))}{f'(x)} \quad (29.6-13a)
\]

\[
= x - \frac{1}{f'(x)} \left[ f(x) + \sum_{k=2}^{r-1} f(\Phi_k(x)) \right] \quad (29.6-13b)
\]

The function \(f\) is evaluated at \(r - 1\) points but the derivative is only evaluated at \(x\). The iteration also involves only one inversion.

### 29.7 Convergence improvement by the delta squared process

Given a sequence of partial sums \(x_k\) the so-called \textit{delta squared process} computes a new sequence \(x_k^*\) of extrapolated sums:

\[
x_k^* = x_{k+2} - \frac{(x_{k+2} - x_{k+1})^2}{x_{k+2} - 2x_{k+1} + x_k} \quad (29.7-1)
\]
Chapter 29: Iterations for the inversion of a function

The method is due to Aitken. The name ‘delta squared process’ is due to the fact that the formula has the form

$$x^* = x - \frac{(\Delta x)^2}{(\Delta^2 x)}$$

(29.7-2)

where $\Delta$ is the difference operator. Note that the mathematically equivalent form

$$x^*_k = \frac{x_k x_{k+2} - x_{k+1}^2}{x_{k+2} - 2 x_{k+1} + x_k}$$

(29.7-3)

sometimes given should be avoided with numerical computations due to possible cancellation.

If $x_k = \sum_{i=0}^{k} a_i$ and the ratio of consecutive summands $a_i$ is approximately constant (that is, $a$ is close to a geometric series) then $x^*$ converges significantly faster to $x_\infty$ than $x$. Rewrite relation [29.7-1] with $a_k := x_k - x_{k-1}$:

$$x^*_k = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}}$$

(29.7-4)

For a geometric series (where $a_{k+1}/a_k = q$) we have

$$x^*_k = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}} = x_{k+2} - \frac{(a_0 q^{k+2})^2}{a_0 (q^{k+2} - q^{k+1})}$$

(29.7-5a)

$$= a_0 \frac{1 - q^{k+3}}{1 - q} + a_0 q^{k+2} \cdot \frac{q^{k+2}}{q^{k+1} - q^{k+2}} = a_0 \frac{1}{1 - q} (1 - q^{k+3} + q^{k+3})$$

(29.7-5b)

$$= a_0 \frac{1}{1 - q}$$

(29.7-5c)

which is the exact sum. Now consider the sequence

$$x_0, \quad x_1 = \Phi(x_0), \quad x_2 = \Phi(x_1) = \Phi(\Phi(x_0)), \quad \ldots$$

(29.7-6)

of successively better approximations to some root $r$ of a function $f$. Think of the $x_k$ as partial sums of a series whose sum is the root $r$. Apply the idea to define an improved iteration $\Phi^*$ from a given one $\Phi$:

$$\Phi^* (x) = \frac{\Phi(\Phi(x)) - \Phi(x)}{\Phi(\Phi(x)) - 2 \Phi(x) + x}$$

(29.7-7)

The good news is that $\Phi^*$ will give quadratic convergence even if $\Phi$ only has linear convergence. For example, take $f(x) = (x^2 - d)^2$, forget that its root $\sqrt{d}$ is a double root, and happily define $\Phi(x) = x - f(x)/f'(x) = x - (x^2 - d)/(4x)$. Convergence is only linear:

$$\Phi(\sqrt{d} \cdot (1 + e)) = \sqrt{d} \cdot \left(1 + \frac{e}{2} + \frac{e^2}{4} + O(e^3)\right)$$

(29.7-8)

Then try

$$\Phi^* (x) = \frac{d \frac{7 x^2 + d}{3x^2 + 5d}}{x(3x^2 + 5d)}$$

(29.7-9)

and find that it has quadratic convergence

$$\Phi(\sqrt{d} \cdot (1 + e)) = \sqrt{d} \cdot \left(1 - \frac{e^2}{4} + \frac{e^3}{16} + O(e^4)\right)$$

(29.7-10)

In general, if $\Phi_n$ has convergence of order $n > 1$ then $\Phi^*_n$ will be of order $2n - 1$, but linear convergence ($n = 1$) is turned into second order, see [151, p.165].
Chapter 30

The arithmetic-geometric mean (AGM)

The arithmetic-geometric mean (AGM) is the basis for fast algorithms for the computation of \(\pi\) to high precision. We give AGM based algorithms for the computation of certain hypergeometric functions. AGM-based algorithms for the computation of the logarithm are given in section 31.1.1 on page 627, and for the exponential function in section 31.2.1 on page 633.

30.1 The AGM

The arithmetic-geometric mean (AGM) plays a central role in the high precision computation of logarithms and \(\pi\). The AGM\((a, b)\) is defined as the limit of the iteration

\[
\begin{align*}
    a_{k+1} &= \frac{a_k + b_k}{2} \\
    b_{k+1} &= \sqrt{a_k b_k}
\end{align*}
\]

starting with \(a_0 = a\) and \(b_0 = b\). Both of the values converge quadratically to a common limit. The related quantity \(c_k\) used in many AGM based computations is defined as

\[
c_k^2 = a_k^2 - b_k^2 = (a_{k-1} - a_k)^2
\]

An alternative way for the computation for the AGM iteration is

\[
\begin{align*}
    c_{k+1} &= \frac{a_k - b_k}{2} \\
    a_{k+1} &= \frac{a_k + b_k}{2} \\
    b_{k+1} &= \sqrt{a_{k+1}^2 - c_{k+1}^2}
\end{align*}
\]

Schönhage gives the most economic variant of the AGM, which, apart from the square root, only needs
Chapter 30: The arithmetic-geometric mean (AGM)

one squaring per step:

\[ A_0 = a_0^2 \] (30.1-4a)
\[ B_0 = b_0^2 \] (30.1-4b)
\[ t_0 = 1 - (A_0 - B_0) \] (30.1-4c)
\[ S_k = \frac{A_k + B_k}{4} \] (30.1-4d)
\[ b_k = \sqrt{B_k} \] [square root] (30.1-4e)
\[ a_{k+1} = \frac{a_k + b_k}{2} \] (30.1-4f)
\[ A_{k+1} = a_{k+1}^2 \] [squaring] (30.1-4g)
\[ A_{k+1} = \frac{(\sqrt{A_k} + \sqrt{B_k})^2}{2} = \frac{A_k + B_k}{4} + \frac{\sqrt{A_k B_k}}{2} \] (30.1-4h)
\[ B_{k+1} = 2 (A_{k+1} - S_k) = b_{k+1}^2 \] (30.1-4i)
\[ c_{k+1}^2 = A_{k+1} - B_{k+1} = a_{k+1}^2 - b_{k+1}^2 \] (30.1-4j)
\[ t_{k+1} = t_k - 2^{k+1} c_{k+1}^2 \] (30.1-4k)

Starting with \( a_0 = A_0 = 1, B_0 = 1/2 \) one has \( \pi \approx (2 a_n^2)/t_n \). The importance of the AGM is related to the fact that it can be used to compute certain hypergeometric functions fast. Indeed, one has

\[ F \left( \begin{array}{c}
\frac{1}{2} \\
\frac{1}{2}
\end{array} \middle| 1 - \frac{b^2}{a^2} \right) = \frac{a}{\text{AGM}(a, b)} = \frac{1}{\text{AGM}(1, b/a)} \] (30.1-5)

The relation is usually written as

\[ F \left( \begin{array}{c}
\frac{1}{2} \\
\frac{1}{2}
\end{array} \middle| k \right) = \frac{1}{\text{AGM}(1, \sqrt{1 - k})} \] (30.1-6)

corresponding to \( \text{AGM}(1, k) \), that is

\[ a_0 = 1, \quad b_0 = k, \quad \text{and} \quad c_0 = \sqrt{1 - k^2} \] (30.1-7)

The quantity

\[ R'(k) := 1 - \frac{1}{2} \sum_{n=0}^{\infty} 2^n c_n^2 \] (30.1-8)

together with the AGM leads to a fast algorithm for the function \( F \left( \begin{array}{c}
-\frac{1}{2} \\
\frac{1}{2}
\end{array} \middle| k \right) \), see section 30.2

Combining two steps of the AGM iteration leads to the fourth order AGM iteration:

\[ \alpha_0 = \sqrt{a_0} \] (30.1-9a)
\[ \beta_0 = \sqrt{b_0} \] (30.1-9b)
\[ \alpha_{k+1} = \frac{\alpha_k + \beta_k}{2} \] (30.1-9c)
\[ \beta_{k+1} = \left( \frac{\alpha_k \beta_k (\alpha_k^2 + \beta_k^2)}{2} \right)^{1/4} \] (30.1-9d)
\[ c_k^4 = \alpha_k^4 - \beta_k^4 = c_{2k}^2 \] (30.1-9e)
30.2: The elliptic functions $K$ and $E$

We have $\alpha_k = \sqrt{a_{2k}}$, $\beta_k = \sqrt{b_{2k}}$. An alternative formulation of the iteration is:

$$\gamma_{k+1} = \frac{\alpha_k - \beta_k}{2} \quad (30.1-10a)$$
$$\alpha_{k+1} = \frac{\alpha_k + \beta_k}{2} \quad (30.1-10b)$$
$$\beta_{k+1} = \left(\alpha_{k+1}^4 - \gamma_{k+1}^4\right)^{1/4} \quad (30.1-10c)$$
$$c_{2k}^2 + 2c_{2k+1}^2 = \alpha_k^4 - \gamma_k^4 \quad (30.1-10d)$$

Compute $R'$ via

$$R'(k) = 1 - \sum_{n=0}^{\infty} 4^n \left(\frac{\alpha_n^4 + \beta_n^2}{2}\right) \quad (30.1-11)$$

30.2 The elliptic functions $K$ and $E$

The elliptic functions $K(k)$ and $E(k)$ can be computed via the AGM which gives super-linear convergence. The logarithmic singularity of $K(k)$ at the point $k = 1$ (relation 30.2-4, see also relation 31.1-1a on page 627) is the key to the fast computation of the logarithm. The exponential function could be computed by inverting the logarithm but also as described in section 31.2.1 on page 633. For computations with very high precision the algorithms based on the elliptic functions are the fastest known today for the logarithm, the number $\pi$, and the exponential function.

30.2.1 Elliptic $K$

The function $K$ can be defined as

$$K(k) = \int_0^{\pi/2} \frac{d\vartheta}{\sqrt{1 - k^2 \sin^2 \vartheta}} = \int_0^1 \frac{dt}{\sqrt{(1-t^2)(1-k^2 t^2)}} \quad (30.2-1)$$

One has

$$K(k) = \frac{\pi}{2} F\left(\frac{1}{2}, \frac{1}{2} \left| k^2 \right. \right) \quad (30.2-2a)$$

$$= \frac{\pi}{2} \sum_{i=0}^{\infty} \frac{(2i-1)!!}{2^i i!}^2 k^{2i} = \frac{\pi}{2} \sum_{i=0}^{\infty} \left(\frac{2i}{4^i}\right)^2 k^{2i} \quad (30.2-2b)$$

$$= \frac{\pi}{2} \left(1 + \frac{1}{2} k^2 + \frac{1}{2} \cdot 3 k^2 + \frac{1}{2} \cdot 3 \cdot 5 k^4 + \frac{1}{2} \cdot 3 \cdot 5 \cdot 7 k^6 + \ldots\right) \quad (30.2-2c)$$

$$= \frac{\pi}{2} \left(1 + \frac{1}{4} k^2 + \frac{9}{64} k^4 + \frac{25}{256} k^6 + \frac{1225}{16384} k^8 + \frac{3969}{65536} k^{10} + \ldots\right) \quad (30.2-2d)$$

See section 38.2.7 on page 711 for transformations in terms of hypergeometric functions. Special values are $K(0) = \frac{\pi}{2}$ and $\lim_{k \to 1^-} K(k) = +\infty$.

The computational interesting form is $F\left(\frac{1}{2}, \frac{1}{2} \left| z \right.\right) = 1/\text{AGM}(1, \sqrt{1-z})$ (see section 30.1 on page 603):

$$K(k) = \frac{\pi}{2} \frac{1}{\text{AGM}(1,k')} = \frac{\pi}{2} \frac{1}{\text{AGM}(1, \sqrt{1-k^2})} \quad (30.2-3a)$$

One defines $k' = \sqrt{1-k^2}$ and $K'(k')$ as $K(k')$:

$$K'(k') := K(\sqrt{1-k'^2}) = \frac{\pi}{2} \frac{1}{\text{AGM}(1,k')} \quad (30.2-3b)$$
Chapter 30: The arithmetic-geometric mean (AGM)

A C++ implementation of the AGM based computation is given in [hfloat: src/tz/elliptic-k.cc].

For \(k\) close to 1 we have

\[
K(k) \approx \log \frac{4}{\sqrt{1-k^2}} \tag{30.2-4}
\]

The following estimate is given in [57, p.11]:

\[
\left| K'(k) - \log \frac{4}{k} \right| \leq 4k^2 (8 + \log k) \quad \text{where} \quad 0 < k \leq 1 \tag{30.2-5}
\]

Product forms for \(K\) and \(K'\) that are also candidates for fast computations are, for \(0 < k_0 \leq 1\),

\[
\frac{2}{\pi} K'(k_0) = \prod_{n=0}^{\infty} \frac{2}{1 + k_n} = \prod_{n=1}^{\infty} 1 + k_n \quad \text{where} \quad k_{n+1} := \frac{2 \sqrt{k_n}}{1 + k_n}, \quad k_\infty = 1 \tag{30.2-6a}
\]

\[
\frac{2}{\pi} K'(k_0) = \prod_{n=0}^{\infty} \frac{1}{\sqrt{k_n}} = \prod_{n=1}^{\infty} 1 + k_n \quad \text{where} \quad k_{n+1} := \frac{1 + k_n}{2 \sqrt{k_n}}, \quad k_\infty = 1 \tag{30.2-6b}
\]

The second form is computationally especially attractive since, apart from the multiplication with the main product, only an inverse square root needs to be computed per step. The product formulas follow directly from relation 30.2-3b \((\text{and AGM}(a, b) = a \text{ AGM}(1, b/a) = b \text{ AGM}(a/b, 1))\):

\[
\frac{1}{\text{AGM}(1, k)} = \left[ \text{AGM} \left( \frac{1+k}{2}, \sqrt{k} \right) \right]^{-1} \tag{30.2-7a}
\]

\[
= \left[ \frac{1+k}{2} \text{ AGM} \left( 1, \frac{2 \sqrt{k}}{1+k} \right) \right]^{-1} \quad \text{(first form)} \tag{30.2-7b}
\]

\[
= \left[ \sqrt{k} \text{ AGM} \left( \frac{1+k}{2 \sqrt{k}}, 1 \right) \right]^{-1} \quad \text{(second form)} \tag{30.2-7c}
\]

Similarly, for \(0 < k_0 \leq 1\),

\[
\frac{2}{\pi} K(k_0) = \prod_{n=0}^{\infty} \frac{2}{1 + k_n'} = \prod_{n=1}^{\infty} 1 + k_n \quad \text{where} \quad k_{n+1} := \frac{1 - k_n'}{1 + k_n}, \quad k_\infty = 0 \tag{30.2-8}
\]

Certain values of the gamma function can be expressed in \(K\) \((\text{taken from [261] p.12})\):

\[
\Gamma \left( \frac{1}{3} \right) = \frac{\pi^{1/3} 2^{7/9}}{3^{1/12}} K \left( \frac{\sqrt{3} - 1}{2 \sqrt{2}} \right)^{1/3} \tag{30.2-9a}
\]

\[
\Gamma \left( \frac{1}{4} \right) = \pi^{1/4} 2 K \left( \frac{1}{\sqrt{2}} \right)^{1/2} \tag{30.2-9b}
\]

\[
\Gamma \left( \frac{1}{8} \right) = \pi^{1/8} 2^{17/8} K \left( \frac{1}{\sqrt{2}} \right)^{1/4} K \left( \sqrt{2} - 1 \right)^{1/2} \tag{30.2-9c}
\]

### 30.2.2 Elliptic \(E\)

The function \(E\) can be defined as

\[
E(k) = \int_0^{\pi/2} \sqrt{1-k^2 \sin^2 \theta} \, d\theta = \int_0^1 \sqrt{1-k^2 t^2} \, dt \tag{30.2-10}
\]
One has

\[
E(k) = \frac{\pi}{2} F\left(\frac{-\frac{1}{2}}{1} \mid k^2 \right) \quad (30.2-11a)
\]

\[
= \frac{\pi}{2} \left( -\sum_{i=0}^{\infty} \frac{(2i-1)!!}{2^i i!} k^{2i} \right) = \frac{\pi}{2} \sum_{i=0}^{\infty} \frac{(2i)!}{4^i} \frac{k^{2i}}{2i-1} \quad (30.2-11b)
\]

\[
= \frac{\pi}{2} \left( 1 - \frac{1}{2} k^2 - \frac{1}{3} k^4 - \frac{1}{3 \cdot 2} \frac{k^6}{3} - \frac{1}{\frac{3}{2} \cdot 4 \cdot 6} \frac{k^8}{5} - \ldots \right) \quad (30.2-11c)
\]

\[
= \frac{\pi}{2} \left( 1 - \frac{1}{4} k^2 - \frac{3}{64} k^4 - \frac{5}{256} k^6 - \frac{175}{16384} k^8 - \frac{441}{65536} k^{10} - \ldots \right) \quad (30.2-11d)
\]

Special values are \(E(0) = \frac{\pi}{2}\) and \(E(1) = 1\). The latter leads to a (slowly converging) series for \(2/\pi\):

\[
\frac{2}{\pi} = F\left(\frac{-\frac{1}{2}}{1} \mid 1 \right) \quad (30.2-12)
\]

Similarly as for \(K'\), one defines \(E'\) as

\[
E'(k) := E(k') = E(\sqrt{1-k^2}) \quad (30.2-13)
\]

The key to fast computation of \(E\) is the relation

\[
\frac{E}{K} = 1 - \frac{1}{2} \sum_{n=0}^{\infty} 2^n c_n^2 \quad (30.2-14)
\]

The terms \(c_n^'\) in the sum occur naturally during the computation of the AGM, see relation \[30.1-2\] on page 603. One defines

\[
R := \frac{E}{K}, \quad R' := \frac{E'}{K'} \quad (30.2-15)
\]

Then \(E\) can be computed via

\[
E(k) = R(k) K(k) = \frac{\pi}{2 AGM(1, \sqrt{1-k^2})} \cdot \left( 1 - \sum_{n=0}^{\infty} 2^{n-1} c_n^2 \right) \quad (30.2-16)
\]

Legendre’s relation between \(K\) and \(E\) is (arguments omitted for readability, choose your favorite form):

\[
\frac{E}{K} + \frac{E'}{K'} - 1 = \frac{\pi}{2 K K'} \quad (30.2-17a)
\]

\[
E K' + E' K - K K' = \frac{\pi}{2} \quad (30.2-17b)
\]

Equivalently,

\[
AGM(1, k) = \frac{E/K}{(1-E'/K')} = \frac{R}{(1-R')} \quad (30.2-18)
\]

For \(k = \frac{1}{\sqrt{2}} =: s\) we have \(k = k'\), thereby \(K = K'\) and \(E = E'\), so

\[
\frac{K(s)}{\pi} \left( \frac{2 E(s)}{\pi} - \frac{K(s)}{\pi} \right) = \frac{1}{2 \pi} \quad (30.2-19)
\]

As expressions \[30.2-3a\] and \[30.2-16\] provide a fast AGM based computation of \(K/\pi\) and \(E/\pi\) the above formula can be used to computer \(\pi\).
Using $E - K = kk^{\prime} \frac{dK}{dk} - k^2 K$ one can express the derivative of $K$ in terms of $E$ and $K$ and thereby compute that quantity fast:

$$\frac{dK}{dk} = \frac{E - k^2 K}{kk^{\prime}}$$  \hspace{1cm} (30.2-20)

For the derivative of $E$ we have

$$\frac{dE}{dk} = \frac{E - K}{k}$$  \hspace{1cm} (30.2-21)

We note the following generalization of Legendre’s relation in terms of hypergeometric functions (see section 35.2 on page 696):

$$\frac{\Gamma(1 + a + b)}{\Gamma(\frac{1}{2} + a + b + c)} \frac{\Gamma(1 + c + b)}{\Gamma(\frac{1}{2} + a + b + c)} = \frac{\Gamma(1 + \frac{1}{2} + a, -\frac{1}{2} - c)}{\Gamma(1 + a + b)} \frac{\Gamma(1 + \frac{1}{2} + a, +\frac{1}{2} + c)}{\Gamma(1 + c + b)} + \frac{\Gamma(1 + \frac{1}{2} + a, +\frac{1}{2} - c)}{\Gamma(1 + a + b)} \frac{\Gamma(1 + \frac{1}{2} + a, +\frac{1}{2} + c)}{\Gamma(1 + c + b)}$$

$$\frac{\Gamma(1 + a + b)}{\Gamma(\frac{1}{2} + a + b + c)} \frac{\Gamma(1 + c + b)}{\Gamma(\frac{1}{2} + a + b + c)} = \frac{\Gamma(1 + \frac{1}{2} + a, -\frac{1}{2} - c)}{\Gamma(1 + a + b)} \frac{\Gamma(1 + \frac{1}{2} + a, +\frac{1}{2} + c)}{\Gamma(1 + c + b)} - \frac{\Gamma(1 + \frac{1}{2} + a, +\frac{1}{2} - c)}{\Gamma(1 + a + b)} \frac{\Gamma(1 + \frac{1}{2} + a, +\frac{1}{2} + c)}{\Gamma(1 + c + b)}$$

This equation is given in [15, p.138]. For $a = b = c = 0$ one obtains Legendre’s relation.

### 30.3 AGM-type algorithms for hypergeometric functions

We give AGM based algorithms for $F\left(\frac{1}{2}, \frac{1}{2}, 1 | \frac{1}{2} \pm 0 \left| \frac{z}{1} \right. \right)$ where $s \in \{0, 1/6, 1/4, 1/3\}$, and $F\left(\frac{1}{4}, t, \frac{1}{4} + t | z \right)$ where $t \in \{1/12, 1/6\}$. These are taken from [60] and [125], both papers are recommended for further studies. See also [195], [59], [82], and [81]. The limit of a three-term iteration as a generalized hypergeometric function is determined in [176]. A four-term iteration is considered in [58].

The following transformations can be applied to the functions, these are special cases of relations 35.2-33a and 35.2-33b on page 700:

$$F\left(\frac{1}{2} + s, \frac{1}{2} - s | \frac{z}{1} \right) = F\left(\frac{1}{4} + \frac{s}{2}, \frac{1}{2} - \frac{s}{2} | 4z(1 - z) \right)$$  \hspace{1cm} (30.3-1a)

$$F\left(\frac{1}{4} + t, \frac{1}{4} - t | \frac{z}{1} \right) = F\left(\frac{1}{2} + 2t, \frac{1}{2} - 2t | \frac{1 - \sqrt{1 - z}}{2} \right)$$  \hspace{1cm} (30.3-1b)

**Algorithms for** $F\left(\frac{1}{2}, \frac{1}{2} | \frac{1}{2} \pm 0 \left| \frac{z}{1} \right. \right)$

The following is relation 30.2-3a on page 805 the classical AGM algorithm which has quadratic convergence:

$$F\left(\frac{1}{2}, \frac{1}{2} | \frac{z}{1} \right) = 1/M \left(1, \sqrt{1-z} \right)$$  \hspace{1cm} (30.3-2a)

$$M(a, b) := \left[(a + b)/2, \sqrt{a b} \right]$$  \hspace{1cm} (30.3-2b)

We write the AGM as $M := [f(a, b), g(a, b)]$ in the obvious way.
A fourth order algorithm obtained by combining two steps of the classical AGM:

\[ F\left(\frac{1}{2}, \frac{1}{2} \mid z\right) = 1/M \left(1, \sqrt{1-z}\right)^2 \]  

where

\[ M(a,b) := \left[(a+b)/2, \sqrt{a b (a^2 + b^2)}/2\right] \]

For comparison, we give the quadratic transform for the hypergeometric function

\[ F\left(\frac{1}{2}, \frac{1}{2} \mid z'\right) = (1 + z) F\left(\frac{1}{2}, \frac{1}{2} \mid z^2\right) \]  

where

\[
\begin{align*}
  z &= 1 - (1 - z')^{1/2} \\
  z' &= 1 - \left(\frac{1-z}{1+z}\right)^2
\end{align*}
\]

It is the special case \(a = 1/2\) and \(b = 1/2\) of the transformation

\[ F\left(\frac{1}{2} \mid \frac{4z}{(1+z)^2}\right) = F\left(\frac{1}{2}, \frac{1}{2} \mid 1 - \left(\frac{1-z}{1+z}\right)^2\right) = (1+z)^2 a F\left(\frac{a}{b} + \frac{1}{2} \mid z^2\right) \]

Algorithms for \( F\left(\frac{1}{3}, \frac{2}{3} \mid z\right) \) \([1/2 \pm 1/6]\)

A third order algorithm:

\[ F\left(\frac{1}{3}, \frac{1}{3} \mid z\right) = 1/M \left(\sqrt[3]{1-z}, 1\right) = \sqrt[3]{1-z} F\left(\frac{2}{3}, \frac{2}{3} \mid z\right) \]  

where

\[ M(a,b) := \left[(a+2b)/3, \sqrt[3]{b (a^2 + a b + b^2)}/3\right] \]

One further has

\[ F\left(\frac{1}{3}, \frac{2}{3} \mid z\right) = 1/M \left(1, \sqrt[3]{1-z}\right) \]

A quadratic algorithm:

\[ F\left(\frac{1}{3}, \frac{2}{3} \mid z\right) = 1/M \left(1, \sqrt[3]{1-z}\right) \]  

where

\[ M(a,b) := \left[\frac{1}{2} \left(\sqrt[3]{2p-a^3} + \sqrt[3]{2m-a^3}\right), \frac{1}{2} (\sqrt[3]{p} + \sqrt[3]{m})\right] \]

and

\[
\begin{align*}
  p &= b^3 + t, \\
  m &= b^3 - t, \\
  t &= \sqrt{b^6 - a^3 b^3}
\end{align*}
\]

And again (see relation 30.3-6a):

\[ F\left(\frac{1}{3}, \frac{1}{3} \mid z\right) = 1/M \left(\sqrt[3]{1-z}, 1\right) = \sqrt[3]{1-z} F\left(\frac{2}{3}, \frac{2}{3} \mid z\right) \]

We note the following hypergeometric transformation due to Ramanujan:

\[ F\left(\frac{1}{3}, \frac{2}{3} \mid z'\right) = (1 + 2z) F\left(\frac{1}{3}, \frac{2}{3} \mid z^3\right) \]
where
\[
 z = \frac{1 - (1 - z')^{1/3}}{1 + 2 (1 - z')^{1/3}} 
\] (30.3-10b)

\[
 z' = 1 - \left(\frac{1 - z}{1 + 2z}\right)^3 
\] (30.3-10c)

The general form is given in [40]:

\[
 F\left(\frac{c}{3}, \frac{c + 1}{3}, \frac{1}{c + 3}\right| 1 - \left(\frac{1 - z}{1 + 2z}\right)^3\right) = (1 + 2z)^{3c} F\left(\frac{c}{3c + 1},\frac{1}{3c} \right| z^3\right) 
\] (30.3-11)

For \( c = 1/3 \) one obtains relation 30.3-10a. A computer algebra proof that relation 30.3-11 is the only possible generalization of relation 30.3-10c is given in [175].

An alternative quadratic algorithm is

\[
 F\left(\frac{1}{3}, \frac{2}{3}, \frac{1}{1} \right| z\right) = 1/M (1, W) 
\] (30.3-12a)

\[
 M(a, b) := \left[\frac{(a + b) / 2, (3 \sqrt{b(b + 2a)/3} - b)}{2}\right] \text{ and} 
\] (30.3-12b)

\[
 W := \frac{1 - R + R^2}{R}, \quad R := \left[\sqrt{u^2 - 1 + u}\right]^{1/3}, \quad u := 1 - 2z 
\] (30.3-12c)

It is given in the form

\[
 F\left(\frac{1}{3}, \frac{2}{3}, \frac{1}{1} \right| (1 - x)(1 + x/2)^2\right) = 1/M (1, x) 
\] (30.3-13)

A product form can be derived from [39] Theorem 6.1: Let

\[
 \alpha(z) := \frac{z (3 + z)^2}{2 (1 + z)^3} 
\] (30.3-14a)

\[
 p(z) := \frac{r^2 - r + 1}{r} \quad \text{where} \quad r := \left[2z + 2\sqrt{z^2 - z - 1}\right]^{1/3} 
\] (30.3-14b)

then, with \( t_0 := 1 - z \) and \( t_{k+1} := \alpha(p(t_k)) \),

\[
 F\left(\frac{1}{3}, \frac{2}{3}, \frac{1}{1} \right| z\right) = \left[\prod_{k=0}^{\infty} \frac{1 + p(t_k)}{2}\right]^{-1} 
\] (30.3-14c)

Convergence is quadratic. The function \( p(z) \) is the real solution of \( p(\beta(z)) = z \) where \( \beta(z) := (z^2(3+z))/4 \).

**Algorithms for** \( F\left(\frac{1}{4}, \frac{3}{4}, \frac{1}{1} \right| z\right) [1/2 \pm 1/4] \)

A quadratic algorithm:

\[
 F\left(\frac{1}{4}, \frac{3}{4}, \frac{1}{1} \right| z\right) = 1/M (1, \sqrt{1 - z})^{1/2} \quad \text{where} \quad M(a, b) := \left[\frac{(a + 3b) / 4, \sqrt{b(a + b)}/2}\right] 
\] (30.3-15a)

(30.3-15b)

One further has (note the swapped arguments in the mean)

\[
 F\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{1} \right| z\right) = 1/M (\sqrt{1 - z}, 1)^{1/2} = \sqrt{1 - z} F\left(\frac{3}{4}, \frac{3}{4}, \frac{1}{1} \right| z\right) 
\] (30.3-16)
Now set \( A_k := \sqrt{(a_k + b_k)/2} \) and \( B_k := \sqrt{b_k} \), then
\[
A_{k+1} = \frac{1}{2} (A_k + B_k) \quad (30.3-17a)
\]
\[
B_{k+1} = \sqrt{A_k B_k} \quad (30.3-17b)
\]
This is the iteration of the classical AGM. Thereby
\[
M(a, b)^{1/2} = \text{AGM} \left( \sqrt{\frac{a + b}{2}}, \sqrt{b} \right) \quad (30.3-18)
\]
and one can employ the AGM scheme of Schönhage (relations 30.1-4a . . . 30.1-4k on page 604). Equivalently,
\[
F \left( \frac{1}{4}, \frac{3}{4} \left| \frac{z}{1} \right. \right) = \frac{1}{\text{AGM} \left( \sqrt{1 + \sqrt{1 - z^2}}, \sqrt{1 - z} \right)} \quad (30.3-19)
\]
We also have
\[
F \left( \frac{1}{4}, \frac{1}{4} \left| \frac{z}{1} \right. \right) = \frac{1}{\text{AGM} \left( \sqrt{1 + \sqrt{1 - z}}, 1 \right)} \quad (30.3-20)
\]
For comparison, we give the hypergeometric transformation
\[
F \left( \frac{1}{4}, \frac{3}{4} \left| \frac{z}{1} \right. \right) = \sqrt{1 + 3 z} F \left( \frac{1}{4}, \frac{3}{4} \left| \frac{z^2}{1} \right. \right) \quad (30.3-21a)
\]
where
\[
z = \frac{1 - (1 - z')^{1/2}}{1 + 3 (1 - z')^{1/2}} \quad (30.3-21b)
\]
\[
z' = 1 - \left( \frac{1 - z}{1 + 3 z} \right)^2 \quad (30.3-21c)
\]
is the special case \( d = 1/4 \) of the transformation
\[
F \left( \frac{d}{3d+2} \left| \frac{z'}{1} \right. \right) = (1 + 3 z)^{2d} F \left( \frac{d}{3d+5} \left| \frac{z^2}{6} \right. \right) \quad (30.3-22)
\]
This is taken from [126] where various such transformations and their generalizations are given.

**Algorithm for** \( F \left( \frac{1/6, 1/3}{1} \left| \frac{z}{1/4 \pm 1/12} \right. \right) \)

A quadratic algorithm is
\[
F \left( \frac{1/6, 1/3}{1} \left| \frac{z}{1} \right. \right) = 1/M (1, W)^{1/2} \quad \text{where}
\]
\[
M(a, b) := \left[ (a + 3b)/4, (\sqrt{ab} + b)/2 \right] \quad \text{and}
\]
\[
W := \frac{1 + R + R^2}{3 R}, \quad R := \left[ \sqrt{z^2 - 1 + z} \right]^{1/3}
\]
It is given [60] p.515] as
\[
\left[ F \left( \frac{1/6, 1/3}{1} \left| 27 x^2 (1 - x)/4 \right. \right) \right]^2 = 1/M (1, x)
\]
One can solve for the argument of \( F \) to obtain the explicit form.
Algorithm for \( F\left(\frac{1/12, 5/12}{1} | z \right) [1/4 \pm 1/6] \)

The following algorithm has quadratic convergence, \( W \) is defined by relation \( 30.3-23c \):

\[
\begin{align*}
F\left(\frac{1/12, 5/12}{1} | z \right) &= \frac{1}{M(1, W)^{1/4}} \\
M(a, b) &= \left[ (a + 15b)/16, \left( \frac{\sqrt{b} (a + 3b)/4 + b)}{2} \right) \right]
\end{align*}
\]

The following relations are given in [195, p.17]:

\[
\begin{align*}
F\left(\frac{1}{4}, \frac{1}{4} \right) \left( -\frac{z}{64} \right) &= \left[ \frac{1}{16^k (z + 16)^3} \right]^{-1/12} F\left(\frac{1/12, 5/12}{1} \right) \left( -\frac{4z}{(1-z)^2} \right) \\
F\left(\frac{1}{3}, \frac{1}{3} \right) \left( -\frac{z}{27} \right) &= \left[ \frac{1}{36^k (z + 3)^3 (z + 27)} \right]^{-1/12} F\left(\frac{1/12, 5/12}{1} \right) \left( -\frac{1728z}{(z + 3)^2 (z + 27)} \right) \\
F\left(\frac{1}{2}, \frac{1}{2} \right) \left( -\frac{z}{16} \right) &= \left[ \frac{1}{16^k (z^2 + 16z + 16)^3} \right]^{-1/12} F\left(\frac{1/12, 5/12}{1} \right) \left( -\frac{1728z (z + 16)}{(z^2 + 16z + 16)^2} \right)
\end{align*}
\]

30.4 Computation of \( \pi \)

We give various iterations for computing \( \pi \) with super-linear convergence. The number of full precision multiplications (FPM) is an indication of the efficiency of the algorithm. The approximate number of FPMs that were counted with a computation of \( \pi \) to 4 million decimal digits (using radix 10,000 and 1 million LIMBs) is indicated like this: \#FPM=123.4. The number is computed as three times the FFT-work in terms of full precision real valued FFTs.

30.4.1 Super-linear iterations for \( \pi \)

AGM implemented in [hfloat: src/pi/piagm.cc], \#FPM=98.4:

\[
\begin{align*}
a_0 &= 1, & b_0 &= \frac{1}{\sqrt{2}} \\
a_{k+1} &= \frac{a_k + b_k}{2} \\
b_{k+1} &= \sqrt{a_k b_k} \\
p_n &= \frac{2 a_{n+1}^2}{1 - \sum_{k=0}^{n} 2^k c_k} \rightarrow \pi \\
\pi - p_n &= \frac{\pi^2 2^{n+4} e^{-2^{n+1}}}{\text{AGM}^2(a_0, b_0)}
\end{align*}
\]

Convergence is second order. Computing \( \pi \) based on the fourth order AGM (relations \( 30.1-9a \ldots 30.1-9e \) on page 604) is possible by setting the second argument of the routine (\#FPM=149.3 for the quartic variant). Schönhage’s variant of the AGM computation (relations \( 30.1-4a \ldots 30.1-4k \) on page 604) is implemented in [hfloat: src/pi/piagmsch.cc] (\#FPM=78.424).
30.4: Computation of $\pi$

The AGM method goes back to Gauss, a facsimile of the entry in his 1809 handbook 6 is given in [18, p.101]. The entry states that

$$\pi = \frac{\text{AGM}(1, k) \text{AGM}(1, k')}{{1 - \sum_{k=0}^{\infty} 2^{k-1} (c_k^2 + c_k'^2)}}$$

(30.4-2)

where $k' = b_0/a_0$ and $k = \sqrt{1 - b_0^2/a_0^2} = c_0/a_0$. For $k = k' = 1/\sqrt{2}$ one obtains relation [30.4-1d]. The formula appeared also 1924 in [166, p.39]. The algorithm was rediscovered 1976 independently by Brent [65] (reprinted in [37, p.424]) and Salamin [232] (reprinted in [37, p.418]).

AGM variant given in [55], [hfloat: src/pi/piagm3.cc], #FPM=99.5 (#FPM=155.3 for the quartic variant):

$$a_0 = 1, \quad b_0 = \frac{\sqrt{6} + \sqrt{2}}{4}$$

(30.4-3a)

$$p_n = \frac{2 a_{n+1}^2}{\sqrt{3} (1 - \sum_{k=0}^{n} 2^{k} c_k^2) - 1} \rightarrow \pi$$

(30.4-3b)

$$\pi - p_n < \frac{\sqrt{3} \pi^2 2^{n+4} e^{-\sqrt{3} \pi 2^{n+1}}}{\text{AGM}(a_0, b_0)^2}$$

(30.4-3c)

AGM variant given in [55], [hfloat: src/pi/piagm3.cc], #FPM=108.2 (#FPM=169.5 for the quartic variant):

$$a_0 = 1, \quad b_0 = \frac{\sqrt{6} - \sqrt{2}}{4}$$

(30.4-4a)

$$p_n = \frac{6 a_{n+1}^2}{\sqrt{3} (1 - \sum_{k=0}^{n} 2^{k} c_k^2) + 1} \rightarrow \pi$$

(30.4-4b)

$$\pi - p_n < \frac{\sqrt{3} \pi^2 2^{n+4} e^{\sqrt{3} \pi 2^{n+1}}}{\text{AGM}(a_0, b_0)^2}$$

(30.4-4c)

Second order iteration from [57, p.170], [hfloat: src/pi/pi2nd.cc], #FPM=255.7:

$$y_0 = \frac{1}{\sqrt{2}}, \quad a_0 = \frac{1}{2}$$

(30.4-5a)

$$y_{k+1} = \frac{1 - (1 - y_k^2)^{1/2}}{1 + (1 - y_k^2)^{1/2}} \rightarrow 0 +$$

(30.4-5b)

$$= \frac{(1 - y_k^2)^{-1/2} - 1}{(1 - y_k^2)^{-1/2} + 1}$$

(30.4-5c)

$$a_{k+1} = a_k (1 + y_{k+1})^2 - 2^{k+1} y_{k+1} \rightarrow \frac{1}{\pi}$$

(30.4-5d)

$$a_k - \pi^{-1} \leq 16 \cdot 2^{k+1} e^{-2^{k+1}} \pi$$

(30.4-5e)

Relation [30.4-5c] shows how to save 1 multiplication per step (see section 28.1 on page 569). A simple proof of this iteration is given in [140].

Borwein’s quartic (fourth order) iteration from [57, p.170], variant $r = 4$, implemented in [hfloat:
Cubic AGM iteration (third order) from [61], implemented in [hfloat: src/pi/picubagm.cc], #FPM=182.7:

\[ y_0 = \sqrt{2} - 1, \quad a_0 = 6 - 4\sqrt{2} \quad (30.4-6a) \]
\[ y_{k+1} = \frac{1 - (1 - y_k^3)^{1/4}}{1 + (1 - y_k^3)^{1/4}} \to 0 + \quad (30.4-6b) \]
\[ a_{k+1} = a_k (1 + y_{k+1})^4 - 2^{2k+3} y_{k+1} (1 + y_{k+1} + y_{k+1}^2) \to \frac{1}{\pi} \quad (30.4-6c) \]

Identities 30.4-6c and 30.4-6c show how to save operations.

Borwein’s quartic (fourth order) iteration, variant \( r = 16 \), implemented in [hfloat: src/pi/pi4th.cc], #FPM=164.4:

\[ y_0 = \frac{1 - 2^{-1/4}}{1 + 2^{-1/4}}, \quad a_0 = \frac{8/\sqrt{2} - 2}{(2^{-1/4} + 1)^4} \quad (30.4-7a) \]
\[ y_{k+1} = \frac{(1 - y_k^4)^{-1/4} - 1}{(1 - y_k^4)^{-1/4} + 1} \to 0 + \quad (30.4-7b) \]
\[ a_{k+1} = a_k (1 + y_{k+1})^4 - 2^{2k+4} y_{k+1} (1 + y_{k+1} + y_{k+1}^2) \to \frac{1}{\pi} \quad (30.4-7c) \]
\[ 0 < a_k - \pi^{-1} \leq 16 \cdot 4^n e^{-4^n \pi^2/4} \quad (30.4-7d) \]

The operation count is unchanged, but this variant gives approximately twice as much precision after the same number of steps. The general form of the quartic iterations (relations 30.4-6a …, and 30.4-7a …) is given in [57] pp.170ff:

\[ y_0 = \sqrt{x^*(r)}, \quad a_0 = \alpha(r) \quad (30.4-8a) \]
\[ y_{k+1} = \frac{(1 - y_k^4)^{-1/4} - 1}{(1 - y_k^4)^{-1/4} + 1} \to 0 + \quad (30.4-8b) \]
\[ a_{k+1} = a_k (1 + y_{k+1})^4 - 2^{2k+2} \sqrt{r} y_{k+1} (1 + y_{k+1} + y_{k+1}^2) \to \frac{1}{\pi} \quad (30.4-8c) \]
\[ 0 < a_k - \pi^{-1} \leq 16 \cdot 4^n \sqrt{r} e^{-4^n \sqrt{r} \pi} \quad (30.4-8d) \]

Derived AGM iteration (second order, from [57] pp.46ff), implemented in [hfloat: src/pi/pideriv.cc], #FPM=276.2:

\[ x_0 = \sqrt{2}, \quad p_0 = 2 + \sqrt{2}, \quad y_1 = 2^{1/4} \quad (30.4-9a) \]
\[ x_{k+1} = \frac{1}{2} \left( \sqrt{x_k} + \frac{1}{\sqrt{x_k}} \right) \quad (k \geq 0) \to 1 + \quad (30.4-9b) \]
\[ y_{k+1} = \frac{y_k \sqrt{x_k} + 1}{y_k + 1} \quad (k \geq 1) \to 1 + \quad (30.4-9c) \]
\[ p_{k+1} = p_k \frac{x_k^{1/4} + 1}{y_k + 1} \quad (k \geq 1) \to \pi + \quad (30.4-9d) \]
\[ p_k - \pi < 10^{-2k+1} \quad (30.4-9e) \]

Cubic AGM iteration (third order) from [61], implemented in [hfloat: src/pi/picubagm.cc], #FPM=182.7:
30.4: Computation of \( \pi \)

\begin{align*}
a_0 &= 1, \quad b_0 = \frac{\sqrt{3} - 1}{2} \tag{30.4-10a} \\
a_{n+1} &= \frac{a_n + 2b_n}{3} \tag{30.4-10b} \\
b_{n+1} &= \sqrt[3]{\frac{b_n (a_n^2 + a_n b_n + b_n^2)}{3}} \tag{30.4-10c} \\
p_n &= \frac{3}{1 - \sum_{k=0}^{\infty} 3^k (a_k^2 - a_{k+1}^2)} \to \pi \tag{30.4-10d}
\end{align*}

Quintic (5th order) iteration from [57, p.310], [hfloat: src/pi/pi5th.cc], #FPM=353.2:

\begin{align*}
s_0 &= 5(\sqrt{5} - 2), \quad a_0 = \frac{1}{2} \tag{30.4-11a} \\
x &= \frac{5}{s_n} - 1 \to 4 \tag{30.4-11b} \\
y &= (x - 1)^2 + 7 \to 16 \tag{30.4-11c} \\
z &= \left(\frac{x}{2} \left(y + \sqrt{y^2 - 4x}\right)\right)^{1/5} \to 2 \tag{30.4-11d} \\
s_{n+1} &= \frac{25}{s_n(z + x/z + 1)^2} \to 1 \tag{30.4-11e} \\
a_{n+1} &= s_n^2 a_n - 5^n \left(s_n^2 - 5 + \sqrt{s_n (s_n^2 - 2s_n + 5)}\right) \to \frac{1}{\pi} \tag{30.4-11f} \\
a_n - \frac{1}{\pi} &< 16 \cdot 5^n e^{- \pi 5^n} \tag{30.4-11g}
\end{align*}

Cubic (third order) iteration from [27], implemented in [hfloat: src/pi/pi3rd.cc], #FPM=200.3:

\begin{align*}
a_0 &= \frac{1}{3}, \quad s_0 = \frac{\sqrt{3} - 1}{2} \tag{30.4-12a} \\
r_{k+1} &= \frac{3}{1 + 2 (1 - s_k^3)^{1/3}} \tag{30.4-12b} \\
s_{k+1} &= \frac{r_{k+1} - 1}{2} \tag{30.4-12c} \\
a_{k+1} &= r_{k+1}^2 a_k - 3^k (r_{k+1}^2 - 1) \to \frac{1}{\pi} \tag{30.4-12d}
\end{align*}

Nonic (9th order) iteration from [27], implemented in [hfloat: src/pi/pi9th.cc], #FPM=273.7:

\begin{align*}
a_0 &= \frac{1}{3}, \quad r_0 = \frac{\sqrt{3} - 1}{2}, \quad s_0 = (1 - r_0^3)^{1/3} \tag{30.4-13a} \\
t &= 1 + 2 r_k \tag{30.4-13b} \\
u &= (9 r_k (1 + r_k + r_k^2))^{1/3} \tag{30.4-13c} \\
v &= t^2 + t u + u^2 \tag{30.4-13d} \\
m &= \frac{27 (1 + s_k + s_k^2)}{v} \tag{30.4-13e} \\
a_{k+1} &= m a_k + 3^2 k^{-1} (1 - m) \to \frac{1}{\pi} \tag{30.4-13f} \\
s_{k+1} &= \frac{(1 - r_k)^3}{(t + 2 u) v} \tag{30.4-13g} \\
r_{k+1} &= (1 - s_k^3)^{1/3} \tag{30.4-13h}
\end{align*}
Chapter 30: The arithmetic-geometric mean (AGM)

30.4.2 Measured timings and operation counts

<table>
<thead>
<tr>
<th>#FPM</th>
<th>order</th>
<th>routine in hfloat</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>78.424</td>
<td>2</td>
<td>pi_agm_sch()</td>
<td>76 sec</td>
</tr>
<tr>
<td>98.424</td>
<td>2</td>
<td>pi_agm()</td>
<td>93 sec</td>
</tr>
<tr>
<td>99.510</td>
<td>2</td>
<td>pi_agm3(fast variant)</td>
<td>94 sec</td>
</tr>
<tr>
<td>108.241</td>
<td>2</td>
<td>pi_agm3(slow variant)</td>
<td>103 sec</td>
</tr>
<tr>
<td>149.324</td>
<td>4</td>
<td>pi_agm(quartic)</td>
<td>139 sec</td>
</tr>
<tr>
<td>155.265</td>
<td>4</td>
<td>pi_agm3(quartic, fast variant)</td>
<td>145 sec</td>
</tr>
<tr>
<td>164.359</td>
<td>4</td>
<td>pi_4th_order(r=16 variant)</td>
<td>154 sec</td>
</tr>
<tr>
<td>169.544</td>
<td>4</td>
<td>pi_agm3(quartic, slow variant)</td>
<td>159 sec</td>
</tr>
<tr>
<td>170.519</td>
<td>4</td>
<td>pi_4th_order(r=4 variant)</td>
<td>160 sec</td>
</tr>
<tr>
<td>182.710</td>
<td>3</td>
<td>pi_cubic_agm()</td>
<td>173 sec</td>
</tr>
<tr>
<td>200.261</td>
<td>3</td>
<td>pi_3rd_order()</td>
<td>189 sec</td>
</tr>
<tr>
<td>255.699</td>
<td>2</td>
<td>pi_2nd_order()</td>
<td>240 sec</td>
</tr>
<tr>
<td>273.763</td>
<td>9</td>
<td>pi_9th_order()</td>
<td>256 sec</td>
</tr>
<tr>
<td>276.221</td>
<td>2</td>
<td>pi-derived_agm()</td>
<td>259 sec</td>
</tr>
<tr>
<td>353.202</td>
<td>5</td>
<td>pi_5th_order()</td>
<td>329 sec</td>
</tr>
</tbody>
</table>

Figure 30.4-A: Measured operations counts and timings for various iterations for the computation of \( \pi \) to 4 million decimal digits.

The operation counts and timings for the algorithms given so far when computing \( \pi \) to 4 million decimal digits (using 1 million LIMBs and radix 10,000) are shown in figure 30.4-A. In view of these figures it seems surprising that the quartic algorithms \( \text{pi}_4\text{th\_order()} \) and the quartic AGM \( \text{pi\_agm(quartic)} \) are usually considered close competitors to the second order AGM schemes.

Apart from the operation count the number of variables used has to be taken into account. The algorithms using more variables (like \( \text{pi}_5\text{th\_order()} \)) cannot be used to compute as many digits as those using only a few (notably the AGM-schemes) given a fixed amount of RAM. Higher order algorithms tend to require more variables.

A further disadvantage of the algorithms of higher order is the more discontinuous growth of the work: if just a few more digits are to be computed than are available after step \( k \) then an additional step is required. Consider an extreme case where an algorithm \( T \) of order 1,000 would compute 1 million digits after the second step, at a slightly lower cost than the most effective competitor. Then algorithm \( T \) would likely be the ‘best’ one only for small ranges in the number of digits around the values \( 10^3, 10^6, 10^9, \ldots \).

Finally, it is much easier to find special arithmetical optimizations for the ‘simple’ (low order) algorithms, Schönhage’s AGM variant being the prime example.

30.4.3 More iterations for \( \pi \)

The following iterations are not implemented in hfloat.
Third order algorithm from [54]:

\[ v_0 = 2^{-1/8}, \quad v_1 = 2^{-7/8} \left( (1 - 3^{1/2}) 2^{-1/2} + 3^{1/4} \right) \]  
\[ w_0 = 1, \quad \alpha_0 = 1, \quad \beta_0 = 0 \]  
\[ v_{n+1} = v_n^3 - \left\{ v_n^6 + \left[ 4 v_n^2 (1 - v_n^8)^{1/3} \right]^{1/2} \right\} + v_{n-1} \]  
\[ w_{n+1} = \frac{2 v_n^3 + v_{n+1} (3 v_n^2 + v_{n+1}^2 - 1)}{2 v_{n+1} - v_n (3 v_n^2 + v_{n+1}^2 - 1)} w_n \]  
\[ \alpha_{n+1} = \left( \frac{2 v_{n+1}^3 + v_n}{v_{n+1}} + 1 \right) \alpha_n \]  
\[ \beta_{n+1} = \left( \frac{2 v_{n+1}^3 + v_n}{v_{n+1}} + 1 \right) \beta_n + \left( 6 v_{n+1} v_n - 2 v_{n+1} w_n \right) \frac{v_{n+1}^2 \alpha_n}{v_n^2} \]  
\[ \pi_n = 8 \cdot 2^{1/8} \frac{\alpha_n \beta_n}{v_n} \rightarrow \pi \]  

Second order algorithm from [62]:

\[ \alpha_0 = 1/3, \quad m_0 = 2 \]  
\[ m_{n+1} = \frac{4}{1 + \sqrt{(4 - m_n) (2 + m_n)}} \]  
\[ \alpha_{n+1} = m_n \alpha_n + \frac{2^n}{3} (1 - m_n) \rightarrow \frac{1}{\pi} \]  

Implicit second order algorithm from [62] (also in [59, p.700]):

\[ (s_n)^2 + (s_n^*)^2 = 1 \]  
\[ (1 + 3 s_n + 1)(1 + 3 s_n^*) = 4 \]  
\[ \alpha_{n+1} = (1 + 3 s_{n+1}) \alpha_n - 2^n s_{n+1} \rightarrow \frac{1}{\pi} \]  

It is trivial to turn this algorithm into an explicit form as with the next algorithm. However, there exist iterations that cannot be turned into explicit forms.

Implicit fourth order algorithm from [62] (also in [59, p.700]):

\[ \alpha_0 = 1/3 \quad s_1 = \sqrt{2} - 1 \]  
\[ (s_n)^4 + (s_n^*)^4 = 1 \]  
\[ (1 + 3 s_n + 1)(1 + 3 s_n^*) = 2 \]  
\[ \alpha_{n+1} = (1 + s_{n+1})^4 \alpha_n + \frac{4^n+1}{3} \left[ 1 - (1 + s_{n+1})^4 \right] \rightarrow \frac{1}{\pi} \]  

Combining two steps of the fourth order iteration leads to an algorithm of order 16. The following form
Chapter 30: The arithmetic-geometric mean (AGM)

is given in [62, p.111]:

\[ \alpha_0 = \frac{1}{3}, \quad s_1 = \sqrt{2} - 1 \]  
\[ s_n^* = (1 - s_n^4)^{1/4} \]  
\[ x_n = \frac{1}{(1 + s_n^*)^4} \]  
\[ y_n = x_n (1 + s_n^4) \]  
\[ \alpha_n = 16 y_n \alpha_{n-1} - 4^{2n-1} [1 - 12 x_n - 4 y_n] - \frac{1}{\pi} \]  
\[ t_n = 1 + s_n^* \]  
\[ u_n = \left[ 8 s_n^* \left( 1 + s_n^* \right)^2 \right]^{1/4} \]  
\[ s_{n+1} = \frac{(1 - s_n^*)^4}{(t + u)^2 (t^2 + u^2)} \]  

Quadratic iteration by Christian Hoffmann, given in [149, p.5]:

\[ a_0 = \sqrt{2}, \quad b_0 = 0, \quad p_0 = 2 + \sqrt{2} \]  
\[ a_{n+1} = \frac{1}{2} \left( \sqrt{a_n + 1/\sqrt{a_n}} \right) \to 1 + \]  
\[ b_{n+1} = \sqrt{a_n} \frac{b_n + 1}{b_n + a_n} \to 1 - \]  
\[ p_{n+1} = p_n \frac{b_{n+1} + 1}{1 + b_{n+1}} \to \pi \]  

Note that relation 30.4-19b deviates from the one given in the cited paper which seems to be incorrect. This is a variant of the iteration given as relations 30.4-9a . . . 30.4-9e on page 614. The values \( p_k \) are identical in both iterations.

Cubic iteration given in [56, p.125]:

\[ s_0 = \sqrt{3 + 2 \sqrt{3}}, \quad a_0 = \frac{1}{2} \]  
\[ m_n = \frac{3}{s_n} \]  
\[ a_{n+1} = \left[ (s_n^2 - 1)^{1/3} + 2 \right] / s_n \]  
\[ a_{n+1} = m_n^2 a_n - 3^n \left( m_n^2 + 2 m_n - 3 \right) / 2 \to \frac{1}{\pi} \]  

The cited paper actually gives a more general form, here we take \( N = 1 \) for simplicity.

Cubic iteration given in [82, p.1506, it-1.2]:

\[ t_0 = \frac{1}{3}, \quad s_0 = \left( \sqrt{3} - 1 \right) / 2 \]  
\[ s_n = \frac{1 - (1 - s_{n-1}^3)^{1/3}}{1 + 2 (1 - s_{n-1}^3)^{1/3}} \]  
\[ = \frac{(1 - s_{n-1}^3)^{-1/3} - 1}{(1 - s_{n-1}^3)^{-1/3} + 2} \]  
\[ t_n = (1 + 2 s_n)^2 t_{n-1} - 3^{n-1} \left( (1 + 2 s_n)^2 - 1 \right) \to \frac{1}{\pi} \]  

Note the corrected denominator in relation 30.4-21b (exponent of \( s_{n-1} \) is wrongly given as 2).
Quadratic iteration given in [82] p.1507, it-1.3]:

\[\begin{align*}
k_0 &= 0, \quad s_0 = 1/\sqrt{2} \quad (30.4-22a) \\
s_n &= \frac{1 - \sqrt{1 - s_n^2}}{1 + \sqrt{1 - s_n^2}} \quad (30.4-22b) \\
k_n &= (1 + s_n)^2 k_{n-1} + 2^n (1 - s_n) s_n \rightarrow \frac{1}{\pi} \quad (30.4-22c)
\end{align*}\]

Cubic iteration given in [82] p.1507, it-1.4]:

\[\begin{align*}
k_0 &= 0, \quad s_0 = 1/\sqrt{3} \quad (30.4-23a) \\
s_n &= \frac{1 - 3\sqrt{1 - s_n^3}}{1 + 3\sqrt{1 - s_n^3}} \quad (30.4-23b) \\
k_n &= (1 + 2s_n)^2 k_{n-1} + 8 \cdot 3^n - 2\sqrt{3} s_n \rightarrow \frac{1}{\pi} \quad (30.4-23c)
\end{align*}\]

Quadratic iteration given in [82] p.1508, it-1.5]:

\[\begin{align*}
k_0 &= 0, \quad y_0 = 8/9 \quad (30.4-24a) \\
y_n &= 2^6 y_{n-1}^2 - 5 y_{n-1} + \sqrt{y_{n-1} (4 - y_{n-1})} \\
Q_n &= \frac{2^n \sqrt{3} y_{n-1} (1 - y_{n-1}) + (4 - 3 y_{n-1}) k_{n-1}}{\sqrt{4 - 3 y_{n-1}}} \quad (30.4-24c) \\
k_n &= 2^n \sqrt[3]{y_{n-1} (1 - y_{n-1}) + (4 - 3 y_{n-1}) k_{n-1}} \rightarrow \frac{1}{\pi}
\end{align*}\]

Quadratic iteration given in [82] p.1508, it-1.6]:

\[\begin{align*}
k_0 &= 0, \quad y_0 = 4/5 \quad (30.4-25a) \\
y_n &= 2^{y_{n-1}^2} - y_{n-1} + \sqrt{4 y_{n-1}^2 - 3 y_{n-1}^2} \\
Q_n &= 2^n \sqrt[3]{y_{n-1} (1 - y_{n-1}) + (4 - 3 y_{n-1}) + 4} \\
k_n &= \frac{2^n y_{n-1} (1 - y_{n-1})}{2 - y_{n-1}} Q_n + (2 - y_{n-1})^2 k_{n-1} \rightarrow \frac{1}{\pi
}\end{align*}\]

Quadratic iteration (as product, two forms) given in [58] p.324]:

\[\begin{align*}
x_1 &= \frac{2}{9} (\sqrt{6} + 2), \quad y_1 = \frac{1}{6} (\sqrt{6} + 4) \quad (30.4-26a) \\
x_{n+1} &= 2 \left( \frac{x_n + x_n}{1 + 3x_n} \right) \quad (30.4-26b) \\
y_{n+1} &= \frac{2y_{n+1}}{1 + 3y_n} \quad (30.4-26c) \\
\pi &= \frac{27}{8} \prod_{n=1}^{\infty} \frac{(1 + 3x_n)^2}{(1 + 3y_n)/4} \quad (30.4-26d) \\
\pi &= \frac{5 + 2\sqrt{6}}{3} \prod_{n=1}^{\infty} \frac{(1 + 1/\sqrt{x_n})^2}{1 + 3y_n} \quad (30.4-26e)
\end{align*}\]

The definitive source for iterations to compute \(\pi\) and the underlying mathematics is [57].
30.5 Arctangent relations for $\pi$ *

This section is about relations of the form

$$k \frac{\pi}{4} = m_1 \arctan \frac{1}{x_1} + m_2 \arctan \frac{1}{x_2} + \ldots + m_n \arctan \frac{1}{x_n} \quad (30.5-1)$$

where $k$, $m_1, \ldots, m_n$, $x_1, \ldots, x_n \in \mathbb{Z}$ (in fact, $k = 1$ almost always). This is an $n$-term relation. For example, a 4-term relation, found 1896 by Størmer [253], is

$$\frac{\pi}{4} = +44 \arctan \frac{1}{57} + 7 \arctan \frac{1}{239} - 12 \arctan \frac{1}{682} + 24 \arctan \frac{1}{12943} \quad (30.5-2)$$

We use the following compact notation

$$m_1[x_1] + m_2[x_2] + \ldots + m_n[x_n] \equiv k \cdot \pi/4$$

for relation 30.5-1. For example, Størmer’s relation 30.5-2 would be written as

$$+44[57] +7[239] -12[682] +24[12943] \equiv 1 \cdot \pi/4$$

We write the relations so that the arguments $x_j$ are strictly increasing. Further, $n$-term relations are sorted so that the first arguments $x_1$ are in decreasing order (if $x_1, \ldots, x_j$ coincide with two relations then the arguments $x_{j+1}$ are used for sorting). For example, a few 6-term relations are

$$+322[577] +76[682] +139[1393] +156[12943] +132[32807] +44[1049433] \equiv 1 \cdot \pi/4$$
$$+183[268] +32[682] +95[1568] +44[1049433] +126[16612943] +51[32807] \equiv 1 \cdot \pi/4$$

Note that the second and third relation are sorted according to their fifth arguments (3458 and 2943). Among all $n$-term relations we consider a relation better than another if it precedes it. The first one is the best relation. Our goal is to find the best $n$-term relation for $n$ small. For example, the relation

$$+322[577] +76[682] +139[1393] +156[12943] +132[32807] +44[1049433] \equiv 1 \cdot \pi/4$$

is the best (known!) 6-term relation. The best $n$-term relations for $2 \leq n \leq 12$ currently known are shown in figure 30.5-A. Note that $k = -1$ in the 10-term relation, and $k = 2$ in the 12-term relation. The best relations for $13 \leq n \leq 21$ (shortened to save space) are shown in figure 30.5-B. Figure 30.5-C gives just the first argument ($x_1$) of the best relations for $2 \leq n \leq 27$. 

<table>
<thead>
<tr>
<th>Relation</th>
<th>Value (top)</th>
<th>Value (bottom)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+4[5] -1[239]$</td>
<td>$1 \cdot \pi/4$</td>
<td>$1 \cdot \pi/4$</td>
</tr>
<tr>
<td>$+12[18] +8[57] -5[239]</td>
<td>$1 \cdot \pi/4$</td>
<td>$1 \cdot \pi/4$</td>
</tr>
<tr>
<td>$+44[57] +7[239] -12[682] +24[12943]</td>
<td>$1 \cdot \pi/4$</td>
<td>$1 \cdot \pi/4$</td>
</tr>
</tbody>
</table>

Figure 30.5-A: Best $n$-term arctan relations currently known for $2 \leq n \leq 12$. 

[fxtbook draft of 2008-August-17]
30.5: Arctangent relations for $\pi$

$\pi \approx 3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067982148086513282306647093844609550582231725359408128481117450284102701938521105559644622948954930381964428810975665933446128475648230674609843 \approx 1 \times \frac{\pi}{4}$

$$+12872838[201229582] +27205340[203420807] \ldots +35839320[134520516108] \approx 1 \times \frac{\pi}{4}$$

+14215326[53141564] +6973645[54610269] \ldots +8735690[34840696582] \approx 1 \times \frac{\pi}{4}

Figure 30.5-B: The best $n$-term arctan relations (shortened) currently known for $13 \leq n \leq 21$. The 4-term entry corresponds to relation 30.5-2.
In the 5-term relation
factor \(x_j^2 + 1\) for all (inverse) arguments \(x_j\):

\[
\begin{align*}
192^2+1 & == 36865 == 5 \cdot 73 \cdot 101239^2+1 == 57122 == 2 \cdot 13 \cdot 13 \cdot 13 \\
515^2+1 & == 265226 == 2 \cdot 13 \cdot 101^2 \cdot 101 \\
1068^2+1 & == 1140625 == 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 73 \\
173932^2+1 & == 30252340625 == 5 \cdot 5 \cdot 5 \cdot 5 \cdot 13 \cdot 73 \cdot 101 \cdot 101
\end{align*}
\]

Note that all odd prime factors are the four primes 5, 13, 73, 101. The coefficients \(m_j\) can be computed as follows. Write (for all arguments \(x_j\))
\[x_j^2 + 1 = 2^{e(j,0)} \cdot 5^{e(j,1)} \cdot 13^{e(j,2)} \cdot 73^{e(j,3)} \cdot 101^{e(j,4)}\]

Now define a matrix \(M\) using the exponents \(e(j, u)\) (ignoring the prime 2):
\[M^T_{j,i} := \pm e(j, i)\]

The sign of \(M_{j,i}\) is minus if \((x_j \mod p_i) < p_i/2\). With our example we obtain
\[
\text{transpose}(M) := \begin{bmatrix}
-56, & -32, & 100, & 39, & 88
\end{bmatrix}
\]

This tells us that

We determine that \(k = 1\) by a floating point computation of the left hand side. Quite often one finds a relation where \(k = 0\), which we are not interested in. For example, the candidates 12943, 1068, 682, 538, 239 all factor into (2 and) the odd primes 5, 13, 61, 73. The matrix \(M\) is
\[
\text{transpose}(M) = \begin{bmatrix}
+4, & +3, & -1, & 0, & 1 \end{bmatrix}
\]

The nullspace of \(M\) is
\[\begin{bmatrix} 1, & -1, & -1, & -1, & 1 \end{bmatrix}\]

and the relation is
30.5.3 Exhaustive search for sets of candidate arguments

We want to find all \( x \) where \( x^2 + 1 \) factors into (2 and) the first 64 primes of the form \( 4i+1 \) (\( S = \{5, 13, 17, 29, \ldots, 761\} \)). Call the resulting set of candidates \( A \). We will later try (for small \( n \)) all \( (n-1) \)-subsets of \( S \) and test whether the corresponding subset of \( A \) leads to an arctan relation.

The most simple approach is to factor (for \( x \) up to a practical maximum) all \( x^2 + 1 \) and add \( x \) to the set \( A \) if all odd prime factors of \( x^2 + 1 \) are in \( S \). The method, however, is rather slow: about 11,000 CPU cycles are needed for each test.

The April-2006 computations were done with a more exhaustive search described in the next section.
A much faster approach is the following sieving method. We can determine exactly when a prime \( p \) divides \( x^2 + 1 \) by solving \( x^2 \equiv -1 \pmod{p} \) as shown in section 37.9 on page 783. We can further solve \( x^2 \equiv -1 \pmod{p^h} \) for all \( h \) as shown in the cited section. Initialize an array with the value 1 for even indices, else with 2 \((x^2 + 1) \) is even exactly if \( x \) is odd. For each prime \( p \in S \) do, for all powers \( p^h \), as follows: multiply the array entries with indices \( s, s + p^h, s + 2p^h, s + 3p^h, \ldots \) where \( s^2 \equiv -1 \pmod{p^h} \) by \( p \). Finally find the entries with index \( x \) that are equal to \( x^2 + 1 \), these are the candidates.

There are three useful improvements. Firstly, we can use the logarithm of a prime and add it instead of multiplying by the prime. The final test is then whether entry \( x \) is (approximately) equal to \( \log(x^2 + 1) \). Secondly, the array can be avoided altogether by using priority queues (see section 4.5.2 on page 155). An event of the priority queue will be to add \( \log(p) \) (initially at index \( x = s \) where \( s^2 \equiv -1 \pmod{p^h} \)). The event must then be rescheduled to \( x + p^h \). Thirdly, almost all computations of the logarithm can be avoided by observing that both \( x^2 + 1 \) and the logarithm are strictly increasing functions. We call a number \( x \) so that \( x^2 + 1 \) has all odd prime factors in \( S \) is a candidate. The sum of logarithms (of primes) for candidates \( x \) are equal to \( \log(x^2 + 1) \). If \( a \) was the last candidate then for the next candidate \( b \) the sum of logarithms must be strictly greater than that for \( a \). Thereby we only need to compute \( \log(x^2 + 1) \) if a new sum of logarithms is greater than the one for the candidate found most recently. It turns out that a logarithm is computed exactly whenever a new candidate is found.

The search costs about 250 cycles per test, which is a good improvement over the first attempt. Analysis of the machine code shows that most of the time is spent in the reschedule operations.

The final improvement comes from the separation of the frequent event (small prime powers) from the rare events (big prime powers). One has to use an array again (but only a small one that fits into level-1 cache and a segmented search should be used). Now the optimum value has to be found from which candidates are considered rare! This means we can forget about the priority queues. A better suited algorithm (and implementation) for a priority queue might give different results.

The resulting routine is remarkably fast, it uses just slightly more than 11 cycles per test. It was used to determine all candidates \( x \leq 10^{14} \). The search took about 8 days. The last entries in the list of candidates are:

\[
\begin{align*}
99205431802196^{2+1} & = [13.29.37.53.89.157.241.257.337.373.401^2.761] \\
99238108604548^{2+1} & = [5.29.37.61^2.101.349.397^2.433.557^2.661] \\
99311314035643^{2+1} & = [2.5^2.13.29.73.113.181.233.281.293.317.349.397.433.557.677] \\
9939576752881^{2+1} & = [2.13.29.37.53.149.173.181.193.313.353.373.401.449] \\
9950123975665^{2+1} & = [2.5^4.13.29.73.113.233.241.269.293.401^2.557^2.621.761] \\
99627378461772^{2+1} & = [5.13^2.29.37.41.73^2.137.277.281.521.557.617.761] \\
99795820688082^{2+1} & = [5^2.17.29.37.109.181.257.269.337.389.409.457.653] \\
9987586159917^{2+1} & = [2.5^2.89.101.181.233.257.293.389.457.521.557.677] \\
99955223464153^{2+1} & = [2.5^2.13.29.61^2.101^2.210.373.421.433.509.709.757]
\end{align*}
\]

The search produced 43,936 candidates (including 0 and 1). Exactly that many logarithms were computed. This means that on average one logarithm was computed for one in \( 10^{9} \). The search took about 8 days. The last entries in the list of candidates are

\[
\begin{align*}
99205431802196^{2+1} & = [13.29.37.53.89.157.241.257.337.373.401^2.761] \\
99238108604548^{2+1} & = [5.29.37.61^2.101.349.397^2.433.557^2.661] \\
99311314035643^{2+1} & = [2.5^2.13.29.73.113.181.233.281.293.317.349.397.433.557.677] \\
9939576752881^{2+1} & = [2.13.29.37.53.149.173.181.193.313.353.373.401.449] \\
9950123975665^{2+1} & = [2.5^4.13.29.73.113.233.241.269.293.401^2.557^2.621.761] \\
99627378461772^{2+1} & = [5.13^2.29.37.41.73^2.137.277.281.521.557.617.761] \\
99795820688082^{2+1} & = [5^2.17.29.37.109.181.257.269.337.389.409.457.653] \\
9987586159917^{2+1} & = [2.5^2.89.101.181.233.257.293.389.457.521.557.677] \\
99955223464153^{2+1} & = [2.5^2.13.29.61^2.101^2.210.373.421.433.509.709.757]
\end{align*}
\]

The search produced 43,936 candidates (including 0 and 1). Exactly that many logarithms were computed. This means that on average one logarithm was computed for one in \( 10^{14}/43,936 > 2 \cdot 10^9 \) values tested.

One can further extend the list by testing (for each element \( x \)) whether the right hand side formula

\[
[x] == [x+d] + [x+(x^2+1)/d] \quad \text{where } d \text{ divides } x^2+1
\]

leads to new candidate \( x+d \) and \( x + (x^2 + 1)/d \). Additionally one can try the arguments on the right hand side of relations like

\[
[x] == 2[4*x] - [4*x^3+3*x^2] \\
[x] == [2*x-1] + [2*x+1] - [2*x^3+x] \\
[x] == 3[3*x] - [(9*x^3+7*x)/2] - [(27*x^3+9*x)/2]
\]

Michael Roby Wetherfield has developed a more sophisticated approach for extending the list and sent me a big set of candidates beyond \( 10^{14} \). His methods are described in [277] (see also [257], [253], and [183]). We note that a single value, \( x = 276, 914, 859, 479, 857, 813, 947 \) where

\[
x^2+1 = [2.5^2.13.17.29^3.41.53^2.73^2.101.157.181.229.241.313.397.401.509.577]
\]

was discarded because it is bigger than \( 2^{64} = 18, 446, 744, 073, 709, 551, 616 \).
30.5: Arctangent relations for π *

We note the curious relation

\[ [k a] = [(k + 1) a] + [(k + 1) k a] - [(k^4 + 2 k^3 + k^2) a^3 + (k^2 + k + 1) a] \quad (30.5-5) \]

Set \( f(a, k) := (k^4 + 2 k^3 + k^2) a^3 + (k^2 + k + 1) a \), then

\[ f(a, k)^2 + 1 = \left( (k a)^2 + 1 \right) \cdot \left( ((k + 1) a)^2 + 1 \right) \cdot \left( ((k + 1) k a)^2 + 1 \right) \quad (30.5-6) \]

30.5.4 Searching for all \( n \)-term relations

To find all \( n \)-term relations whose arguments are a subset of our just determined list of candidates, we have to test all subsets of \((n - 1)\) (out of 64) odd primes, select the corresponding values \( x \), and compute the nullspace as described. Let \( A_j \) be the \( j \)-th candidate. An array \( M \) of 64-bit auxiliary values is used. Its \( j \)-th entry \( M_j \) is a bit-mask corresponding to the odd primes in the factorization of \( A_j^2 + 1 \): bit \( i \) of \( M_j \) is set if the \( i \)-th odd prime divides \( A_j^2 + 1 \).

To find \( n \)-term relations, we must try all \( \binom{64}{n-1} \) subsets of size \( n - 1 \) out of the 64 odd primes in our scope. The bit-combination routine from section 1.25 on page 67 was used for this task. The selection of the entries that factor completely in the subset of \( n - 1 \) primes under consideration can be done with a single bit-AND and a branch. The candidates with more than \( n - 1 \) odd primes in their factorization should be discarded before the search.

While the search is very fast for small \( n \), it does not finish in reasonable time for \( n > 8 \). A considerable speedup can be achieved by splitting our \( N = 64 \) odd primes into a group of the 20 smallest and \( b = 64 - 20 = 44 \) ‘big’ primes. Write \( (q = n - 1 \text{ and}) \)

\[
\binom{N}{q} = \binom{b}{0} \binom{N-b}{q} + \binom{b}{1} \binom{N-b}{q-1} + \binom{b}{2} \binom{N-b}{q-2} + \ldots \quad (30.5-7a)
\]

\[
= \sum_{j=0}^{q} \binom{b}{j} \binom{N-b}{q-j} \quad (30.5-7b)
\]

This means, we first select the \( j = 0, 1, 2, \ldots \)-subsets of the big primes. We copy the corresponding candidates whose big prime factors are in the current subset into a new array \( B \). The size of \( B \) will be significantly smaller than the size of \( A \). From this array we select the arguments according to subsets of the small primes (leaving the subset of big primes fixed). This results in a much improved memory locality and accelerates the search by a factor of about 25.

Still, the limit for \( n \) so that an exhaustive search can be done has only been moved a little. But if we look at the prime sets that lead to the best relations, we observe that small primes are much ‘preferred’:

<table>
<thead>
<tr>
<th>( n )</th>
<th>prime set of best relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{13}</td>
</tr>
<tr>
<td>3</td>
<td>{5, 13}</td>
</tr>
<tr>
<td>4</td>
<td>{5, 13, 61}</td>
</tr>
<tr>
<td>5</td>
<td>{5, 13, 73, 101}</td>
</tr>
<tr>
<td>6</td>
<td>{5, 13, 61, 89, 197}</td>
</tr>
<tr>
<td>7</td>
<td>{5, 13, 17, 29, 97, 433}</td>
</tr>
<tr>
<td>8</td>
<td>{5, 13, 29, 37, 61, 97, 337}</td>
</tr>
<tr>
<td>9</td>
<td>{5, 13, 17, 29, 41, 53, 97, 269}</td>
</tr>
<tr>
<td>10</td>
<td>{5, 13, 17, 41, 53, 73, 97, 101, 157}</td>
</tr>
<tr>
<td>11</td>
<td>{5, 13, 17, 29, 37, 53, 61, 97, 101}</td>
</tr>
<tr>
<td>12</td>
<td>{5, 13, 17, 29, 37, 53, 61, 89, 97, 101, 197}</td>
</tr>
<tr>
<td>13</td>
<td>{5, 13, 17, 29, 37, 53, 61, 89, 97, 101, 181, 281}</td>
</tr>
<tr>
<td>14</td>
<td>{5, 13, 17, 29, 37, 53, 61, 89, 97, 101, 181, 269, 457}</td>
</tr>
<tr>
<td>15</td>
<td>{5, 13, 17, 29, 37, 41, 53, 61, 89, 97, 101, 181, 337, 389}</td>
</tr>
</tbody>
</table>

The data suggests that the best possible relation is found long before the search space is exhausted. Therefore we will stop after the number of big primes in the subset is greater than, say, 4. In practice both parameters, the number \( b \) of primes considered big and the maximum number of primes taken from that set should be chosen depending on \( n \).

[fxtbook draft of 2008-August-17]
Another important improvement is to discard small candidates before the search. One thereby avoids the huge amount of uninteresting relations with small first arguments $x_1$. Obviously, the amount of nullspace computation is also reduced significantly.

The results of the searches can be found in [19]. While the searches for the $n$-term relations with $n > 11$ did not even exhaust the table of candidates (which in turn is incomplete!) one can be reasonably sure that the best relations within our scope (of the first 64 odd primes $4i + 1$) have been found. Indeed I do not expect to see a better relation for any $n \leq 15$.

To improve on the results, one should use the first 128 odd primes $4i + 1$, sieve up to $10^{16}$ (distributed on 100 machines), and use a 3-phase subset selection instead of the described 2-phase selection. The selection (an nullspace computation) stage should also be done in a distributed fashion to reasonably exhaust the table of candidates. Such a computation would likely improve on some of the relations with more than 17 terms and produce up to 35-term relations that are in the vicinity of the best possible.

A method for the simultaneous computation of logarithms of small primes that uses a similar method to the one given here is described in section 31.4 on page 638.

[fxtbook draft of 2008-August-17]
Chapter 31

Logarithm and exponential function

We describe algorithms for the computation of the exponential (and hyperbolic cosine) function, and the logarithm (and inverse tangent). Constructions of super-linear iterations the compute the functions from their inverses are given.

We give argument reduction schemes and methods for the fast computation of the exponential and logarithm of power series.

31.1 Logarithm

31.1.1 AGM-based computation

The (natural) logarithm can be computed using the relation (see [57, p.221])

\[ |\log(d) - R'(10^{-n}) + R'(10^{-n} d)| \leq \frac{n}{10^{2(n-1)}} \]  

(31.1-1a)

\[ \log(d) \approx R'(10^{-n}) - R'(10^{-n} d) \]  

(31.1-1b)

which holds for \( n \geq 3 \) and \( d \in [\frac{1}{2}, 1] \). Note that the first term on the right hand side is constant and can be saved for subsequent log-computations. One uses the relation

\[ \log(M r^X) = \log(M) + X \log(r) \]  

(31.1-2)

where \( M \) is the mantissa, \( r \) the radix, and \( X \) the exponent of the floating point representation. The value \( \log(r) \) is computed only once. If \( M \) is not in the interval \([1/2, 3/2]\) an argument reduction is made via

\[ \log(M) = \log(M s^f) - f \log(s) \]  

(31.1-3)

Where \( 0 < M < 1 \) for the mantissa \( M \), \( s = \sqrt{2} \), and \( f \in \mathbb{Z} \) so that \( M s^f \in [1/2, 3/2] \). The quantity \( \log(s) = \log(\sqrt{2}) \) can be precomputed directly via the AGM. A C++ implementation is given in [hfloat: src/tz/log.cc].
Chapter 31: Logarithm and exponential function

There is a nice way to compute the value of \( \log(r) \) if the value of \( \pi \) has been precomputed. One defines

\[
\Theta_2(q) = \sum_{n=-\infty}^{\infty} q^{(n+1/2)^2} = 2 \left( q^{1/4} + q^{9/4} + q^{25/4} + q^{81/4} + q^{121/4} + \ldots \right) \quad (31.1-4a)
\]

\[
= 2 q^{1/4} \left( 1 + q^2 + q^6 + q^{12} + q^{20} + \ldots \right) \quad (31.1-4b)
\]

\[
\Theta_3(q) = \sum_{n=-\infty}^{\infty} q^{n^2} = 1 + 2 \left( q + q^4 + q^9 + q^{16} + q^{25} + \ldots \right) \quad (31.1-4c)
\]

\[
\Theta_4(q) = \sum_{n=-\infty}^{\infty} (-1)^n q^{n^2} = 1 + 2 \left( -q + q^4 - q^9 + q^{16} - q^{25} + \ldots \right) \quad (31.1-4d)
\]

We have (with \( K \) the elliptic function, see section 30.2.1 on page 605)

\[
\log \frac{1}{q} = - \log q = \pi \frac{K'}{K} \quad (31.1-5)
\]

so \( q = \exp(-\pi K'/K) \), and

\[
\frac{\pi}{\log(1/q)} = - \frac{\pi}{\log(q)} = \text{AGM} \left( \Theta_3^2(q), \Theta_2^2(q) \right) = \frac{\text{AGM}(1, k)}{\text{AGM}(1, k')}. \quad (31.1-6)
\]

Computing \( \Theta_3(q) \) is easy if \( q = 1/r \):

\[
\Theta_3(q) = 1 + 2 \sum_{n=1}^{\infty} q^{n^2} = 2 \left( 1 + \sum_{n=1}^{\infty} q^{n^2} \right) - 1 \quad (31.1-7)
\]

However, the computation of \( \Theta_2(q) \) suggests to choose \( q = 1/r^4 =: b^4 \):

\[
\Theta_2(q) = 0 + 2 \sum_{n=0}^{\infty} q^{(n+1/2)^2} = 2 \sum_{n=0}^{\infty} b^{4n^2+4n+1} \quad \text{where} \quad q = b^4 \quad (31.1-8a)
\]

\[
= 2 b \sum_{n=0}^{\infty} q^{n^2+n} = 2 b \left( 1 + \sum_{n=1}^{\infty} q^{n^2+n} \right) \quad (31.1-8b)
\]

Functions that compute \( \Theta_2(b^4), \Theta_3(b^4) \) and \( \pi/\log(b) \) where \( b \) is the inverse fourth power of the used radix \( r \) are given in [hfloat: src/tz/pilogq.cc].

We give some relations involving the theta functions (see [278]):

\[
\Theta_3^2(q) = \Theta_2^2(q) + \Theta_1^4(q) \quad (31.1-9a)
\]

\[
\Theta_3(q) = \Theta_3(q^4) + \Theta_2(q^4) \quad \Theta_4(q) = \Theta_3(q^4) - \Theta_2(q^4) \quad (31.1-9b)
\]

\[
\Theta_2^2(q) = \frac{2k K}{\pi} \quad \Theta_3^2(q) = \frac{2 K}{\pi} \quad \Theta_4^2(q) = \frac{2 k' K}{\pi} \quad (31.1-9c)
\]

\[
k = \frac{\Theta_2^2(q)}{\Theta_3^2(q)} \quad k' = \frac{\Theta_2^2(q)}{\Theta_3^2(q)} \quad (31.1-9d)
\]

\[
1 = \text{AGM} \left( \Theta_3^2(q), \Theta_2^2(q) \right) \quad (31.1-9e)
\]

Expression for the eta-products (see section 35.1.4.2 on page 694) are

\[
q \eta(q)^{24} = q \prod_{n=1}^{\infty} \left( 1 - q^n \right)^{24} = \frac{256}{\pi^{12}} k^2 k^6 K^{12} \quad (31.1-10a)
\]

\[
q \eta_+(q)^{24} = q \prod_{n=1}^{\infty} \left( 1 + q^n \right)^{24} = \frac{k^2}{16 k'^4} \quad (31.1-10b)
\]
31.1.2 Computation by inverting the exponential function

31.1.2.1 Iterations from the power series

With an efficient algorithm for the exponential function one can compute the logarithm using

\[ y := 1 - d e^{-x} \]  

\[ \log(d) = x + \log(1 - y) \]  

\[ = x + \log \left( 1 - (1 - d e^{-x}) \right) = x + \log(e^{-x}d) = x + (-x + \log(d)) \]  

Thereby,

\[ \log(d) = x + \log \left( 1 - y \right) = x - \left( y + \frac{y^2}{2} + \frac{y^3}{3} + \frac{y^4}{4} + \ldots \right) \]  

Truncation of the series before the \( n \)-th power of \( y \) gives an iteration of order \( n \):

\[ x_{k+1} = \Phi_n(x_k) := x - \left( y + \frac{y^2}{2} + \frac{y^3}{3} + \ldots + \frac{y^{n-1}}{n-1} \right) \]  

31.1.2.2 Iterations from Padé approximants

Padé approximants \( P_{i,j}(z) \) of \( \log(1 - z) \) at \( z = 0 \) produce iterations of order \( i + j + 1 \). Compared to the power series based iteration one needs one additional long division but saves half of the exponentiations. This can be a substantial saving for high order iterations.

The approximants can be computed via the continued fraction expansion of \( \log(1 + z) \):

\[ \log(1 + z) = 0 + \frac{c_1 z}{1 + \frac{c_2 z}{1 + \frac{c_3 z}{1 + \ldots}}} \]  

where \( c_1 = 1 \) and

\[ c_k = \frac{k}{4(k-1)} \] if \( k \) even, \[ c_k = \frac{k-1}{4k} \] else

Using recurrence relations \( 35.3-7a \) and \( 35.3-7b \) on page 716 with \( a_0 = 0 \), \( a_k = 1 \) and \( b_k = c_k \cdot z \) we obtain (fractions in lowest terms):

1 \[ \rightarrow P_{[1,0]} = \frac{z}{1} \]  

2 \[ \rightarrow P_{[1,1]} = \frac{2z}{2 + z} \]  

3 \[ \rightarrow P_{[2,1]} = \frac{6z + z^2}{6 + 4z} \]  

4 \[ \rightarrow P_{[2,2]} = \frac{6z + 3z^2}{6 + 6z + z^2} \]  

5 \[ \rightarrow P_{[3,2]} = \frac{30z + 21z^2 + z^3}{30 + 36z + 9z^2} \]  

6 \[ \rightarrow P_{[3,3]} = \frac{60z + 60z^2 + 11z^3}{60 + 90z + 36z^2 + 3z^3} \]  

7 \[ \rightarrow P_{[4,3]} = \frac{420z + 510z^2 + 140z^3 + 3z^4}{420 + 720z + 360z^2 + 48z^3} \]  

8 \[ \rightarrow P_{[4,4]} = \frac{420z + 630z^2 + 260z^3 + 25z^4}{420 + 840z + 540z^2 + 120z^3 + 6z^4} \]
Chapter 31: Logarithm and exponential function

The expressions are Padé approximants correct up to order $k$. For even $k$ these are the diagonal approximants $[k/2,k/2]$ which satisfy the functional equation $\log(1/\omega) = -\log(\omega) : P(1/\omega - 1) = -P(\omega - 1)$. Further information like the error term of the diagonal approximants is given in [193].

The diagonal Padé approximants can be computed by setting $P_0 = 0$, $Q_0 = 1$, $P_2 = z$, $Q_2 = 1 + z/2$, and computing, for $k = 4, 6, \ldots, 2n$,

\[
P_k = A_k P_{k-2} + B_k P_{k-4} \tag{31.1-19a}
\]
\[
Q_k = A_k Q_{k-2} + B_k Q_{k-4} \tag{31.1-19b}
\]

(These are relations 35.3-14a and 35.3-14b on page 719). The $A_k$ and $B_k$ are defined as

\[
A_k = 1 + z/2 \tag{31.1-19c}
\]
\[
B_k = \frac{z^2}{16} \frac{(k-2)^2}{1-(k-2)^2} \tag{31.1-19d}
\]

Then $P_{2n}/Q_{2n}$ is the Padé approximant $[n,n]$ of log$(1+z)$ which is correct up to order $2n$. The following pari/gp function implements the algorithm:

```
1 log_pade(n, z='z)=
2 { /* Return Padé approximant [n,n] of log(1+z) */
3   local(P0,Q0,P2,Q2,tp,tq, t);
4   if ( n<1, return(0) );
5   P0=0; Q0=1;
6   P2=z; Q2=1+z/2;
7   forstep (k=4, 2*n, 2,
8     Ak = 1+z/2; \ \ \ \ \ \ \ \ \ \ \ \ =z*C(k-1)+z*C(k)+1;
9     t = (k-2)^2;
10     Bk = z^2/16*t/(1-t); \ \ \ \ \ \ \ \ \ \ \ \ =-z^2*C(k-1)*C(k-2);
11     tp = Ak*P2 + Bk*P0;
12     tq = Ak*Q2 + Bk*Q0;
13     P0=P2; P2=tp;
14     Q0=Q2; Q2=tq;
15   );
16   return( P2/Q2 );
17 }
```

31.1.2.3 Padé approximants for arctan *

A continued fraction for arctan is (given in [192, p.569])

\[
\text{arctan}(z) = z \frac{1}{1 + \frac{z^2/(1 \cdot 3)}{1 + \frac{2z^2/(3 \cdot 5)}{1 + \frac{3z^2/(5 \cdot 7)}}}} \tag{31.1-20}
\]

Padé approximants $P_k/Q_k$ for the inverse tangent can be obtained by setting $P_0 = 0$, $P_1 = z$, $Q_0 = 1$, $Q_1 = 1$, and the recurrences

\[
P_{k+1} = P_k + P_{k-1} \frac{k^2 z^2}{4k^2 - 1} \tag{31.1-21a}
\]
\[
Q_{k+1} = Q_k + P_{k-1} \frac{k^2 z^2}{4k^2 - 1} \tag{31.1-21b}
\]

[fxtbook draft of 2008-August-17]
The first few approximants are

\[
\begin{align*}
2 & \mapsto P_{1,2} = \frac{3x}{3 + x^2} & (31.1-22a) \\
3 & \mapsto P_{3,2} = \frac{15x + 4x^3}{15 + 9x^2} & (31.1-22b) \\
4 & \mapsto P_{3,4} = \frac{105x + 55x^3}{105 + 90x^2 + 9x^4} & (31.1-22c) \\
5 & \mapsto P_{5,4} = \frac{945x + 735x^3 + 64x^5}{945 + 1050x^2 + 225x^4} & (31.1-22d) \\
6 & \mapsto P_{5,6} = \frac{1155x + 1575x^3 + 231x^5}{1155 + 1575x^2 + 25x^6} & (31.1-22e) \\
k & \mapsto P_{[i,j]} = \arctan(z) + O(z^{2k+1}) & (31.1-22f)
\end{align*}
\]

31.1.3 Argument reduction for the logarithm

When the logarithm shall be computed for high but not extreme precision (up to several hundred decimal digits or so), the following scheme can beat the AGM algorithm. Use the functional equation for the logarithm

\[
\log(z^a) = a \log(z) \quad (31.1-23)
\]

to reduce the argument by setting \(a = 1/N\). Now with \(N\) big enough \(z^{1/N}\) will be close to one: \(r := z^{1/N} = 1 + e\) where \(e\) is small. Then a few terms of the Taylor series of \(\log(1 + e) = e - e^2/2 + e^3/3 \pm \ldots\) suffice to compute the logarithm. Compute the logarithm of \(z\) as follows

1. set \(r = z^{1/N}\) and \(e = r - 1\)
2. compute \(l := \log(1 + e)\) to the desired precision using the Taylor series
3. return \(L := Nl\)

One can also use a Padé approximant in step 2. With argument \(z = 2.0\), \(N = 2^{32}\), and four terms of the Taylor series we obtain:

\[
\begin{align*}
? z=2.0; & \quad \text{// argument for log()} \\
? n=32; & \quad N=2^n; \\
? r=z^(1/N); & \quad \text{// compute by 32 sqrt extractions} \\
1.0000000000161385904209659761203976631101985032744612016 & \\
? e=r-1; & \quad \text{// small} \\
1.61385904209659761203976631101985032744612016503265785 E-10 & \\
? l=e-1/2*e^2+1/3*e^3-1/4*e^4; & \quad \text{// approx log(1+e)} \\
1.6138590416637056166593013670822486594054133693140550 E-10 & \\
? L=N*l; & \quad \text{// final result} \\
0.69314718055994530941723212145817656807540609322656503655 & \\
? log(z); & \quad \text{// check with builtin log} \\
0.6931471805599453094172321214581765680754060932265650365
\end{align*}
\]

One may also use the following reduction for \(L(z) := \log(1 + z)\), which avoids loss of precision for small values of \(z\):

\[
L(z) = 2L \left( \frac{z}{1 + \sqrt{1 + z}} \right) \quad (31.1-24)
\]

31.1.4 Argument reduction for \(\arctan\)

We use the equation

\[
\arctan(z) = 2 \arctan \left( \frac{z}{1 + \sqrt{1 + z^2}} \right) \quad (31.1-25)
\]

Compute the inverse tangent of \(z\) as follows

[fxtbook draft of 2008-August-17]
1. set $r := z$

2. repeat $n$ times: $r = r/(1 + \sqrt{1+r^2})$ (for $n$ big enough)

3. compute $a := \arctan(r)$ to the desired precision using the Taylor series

4. return $A := 2^n a$

We compute $\arctan(1.0)$ using $n = 16$ and four terms of the Taylor series:

```
? z=1.0;  \\  \text{for}(k=1,n,r=r/(1+sqrt(1+r^2)));r
0.0000119842249059303478511634794650661319585128774402526189
? arcs=1/3*r^3+1/5*r^5-1/7*r^7
0.00001198422490535657210717255929058184974556765696760263
? A=2^n*arcs
0.7853981633974483096156608458198757210492552196703258299
```

All divisions in the reduction phase can be saved by using

$$\arctan\left(\frac{1}{z}\right) = 2 \arctan\left(\frac{1}{1 + \sqrt{1 + z^2}}\right)$$

The inverse sine and cosine can be computed as

$$\arcsin(z) = \arctan\left(\frac{z}{\sqrt{1 - z^2}}\right)$$

$$\arccos(z) = \arctan\left(\frac{\sqrt{1 - z^2}}{z}\right)$$

### 31.1.5 A curious series for the logarithm*

We finally note two relations resembling the well-known series

$$\frac{1}{2} \log\left(\frac{1 + x}{1 - x}\right) = x + \frac{1}{3} x^3 + \frac{1}{5} x^5 + \ldots + \frac{1}{2k+1} x^{2k+1} + \ldots$$

The first is

$$\frac{1}{6} \log\left(\frac{1 + 3x + 3x^2}{1 - 3x + 3x^2}\right) = x - \frac{3^2}{5} x^5 - \frac{3^3}{7} x^7 + \frac{3^5}{11} x^{11} + \frac{3^6}{13} x^{13} \pm \ldots =$$

$$= \sum_{k=0}^{\infty} \left(\frac{3^{2k+1}}{12k+1} - \frac{3^{2k+2}}{12k+5} x^{12k+5} - \frac{3^{2k+3}}{12k+7} x^{12k+7} + \frac{3^{2k+5}}{12k+11} x^{12k+11}\right)$$

The second, given in [28], is

$$\frac{1}{3} \log\left(\frac{1 + x + x^2}{1 - 2x + x^2}\right) = x + \frac{x^2}{2} + \frac{x^4}{4} + \frac{x^5}{5} + \ldots = \sum_{k=0}^{\infty} \left(\frac{x^{3k+1}}{3k+1} + \frac{x^{3k+2}}{3k+2}\right)$$

Relation 31.1-29a can be brought in a similar form, namely

$$\frac{1}{2 \sqrt{3}} \log\left(\frac{1 + \sqrt{3} x + x^2}{1 - \sqrt{3} x + x^2}\right) = x - \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^{11}}{11} + \frac{x^{13}}{13} \pm \ldots =$$

$$= \sum_{k=0}^{\infty} \left(\frac{x^{12k+1}}{12k+1} - \frac{x^{12k+5}}{12k+5} x^{12k+5} - \frac{x^{12k+7}}{12k+7} x^{12k+7} + \frac{x^{12k+11}}{12k+11}\right)$$

*[fxtbook draft of 2008-August-17]
31.2 Exponential function

31.2.1 AGM-based computation of the exponential function

We use the following relation (see [166], pp.35-36):

$$ q = \exp \left( -\pi \frac{K'}{K} \right) $$

(31.2-1)

Now write

$$ \frac{K'}{K} = \frac{\text{AGM}(1,k')}{\text{AGM}(1,k)} = \frac{\text{AGM}(1,b_0)}{\text{AGM}(1,b'_0)} $$

(31.2-2)

where $k' = b_0$ and $k = b'_0 = \sqrt{1 - b_0^2}$ and use ([166], p.38, note the missing ‘4’ there)

$$ \pi \frac{\text{AGM}(1,b_0)}{\text{AGM}(1,b'_0)} = \lim_{n \to \infty} \frac{1}{2^n} \log\left(\frac{4a_n}{c_n}\right) $$

(31.2-3)

thereby

$$ q = \exp \left( -2 \lim_{n \to \infty} \frac{1}{2^n} \log\left(\frac{4a_n}{c_n}\right) \right) = \lim_{n \to \infty} \exp \left( -\frac{1}{2^{n-1}} \log\left(\frac{4a_n}{c_n}\right) \right) $$

(31.2-4a)

$$ = \lim_{n \to \infty} \left( \exp \log\left(\frac{4a_n}{c_n}\right) \right)^{-1/2^{n-1}} = \lim_{n \to \infty} \left( \frac{4a_n}{c_n} \right)^{-1/(2^{n-1})} $$

(31.2-4b)

This gives

$$ q = \lim_{n \to \infty} \left( \frac{c_n}{4a_n} \right)^{1/(2^{n-1})} $$

(31.2-5)

One obtains an algorithm for $\exp(-x)$ by first solving for $k, k'$ so that $x = \pi K'/K$ (precomputed $\pi$) and applying the last relation that implies the computation of a $2^{n-1}$-th root. Note that the quantity $c$ should be computed via $c_{n+1} = \frac{c_n^2}{4a_{n+1}}$ throughout the AGM computation in order to preserve its accuracy.

For $k = 1/\sqrt{2} =: s$ one has $k = k'$ and so $q = \exp(-\pi)$. Thus the calculation of $\exp(-\pi) = 0.0432139182637\ldots$ can directly be done via a single AGM computation as $(c_n/(4a_n))^N$ where $N = 1/2^{(n-1)}$. The quantity $i^i = \exp(-\pi/2) = 0.2078795763507\ldots$ can be obtained using $N = 1/2^n$.

31.2.2 Computation by inverting the logarithm

31.2.2.1 Iterations from the power series

The exponential function can be computed using the iteration that is obtained as follows:

$$ \exp(d) = x \exp(d - \log(x)) $$

$$ = x \exp(y) \quad \text{where} \quad y := d - \log(x) $$

$$ = x \left( 1 + y + \frac{y^2}{2} + \frac{y^3}{3!} + \frac{y^4}{4!} + \ldots \right) $$

(31.2-6)

(31.2-7)

(31.2-8)

The corresponding $n$-th order iteration is

$$ x_{k+1} = \Phi_n(x_k) := x_k \left( 1 + y + \frac{y^2}{2} + \frac{y^3}{3!} + \ldots + \frac{y^{n-1}}{(n-1)!} \right) $$

(31.2-9)

As the computation of logarithms is expensive one should use an iteration of high order. The C++ implementation given in [hfloat: src/tz/itexp.cc] uses the iteration of order 20.
31.2.2.2 Iterations from Padé approximants

Padé approximants $P_{i,j}(z)$ of $\exp(z)$ around $z = 0$ give iterations of order $i + j + 1$. For $i = j$ we obtain

\[
P_{[1,1]} = \frac{2 + z}{2 - z} \tag{31.2-10a}
\]
\[
P_{[2,1]} = \frac{12 + 6z + z^2}{12 - 6z + z^2} \tag{31.2-10b}
\]
\[
P_{[3,1]} = \frac{120 + 60z + 12z^2 + z^3}{120 - 60z + 12z^2 - z^3} \tag{31.2-10c}
\]
\[
P_{[4,1]} = \frac{1680 + 840z + 180z^2 + 20z^3 + z^4}{1680 - 840z + 180z^2 - 20z^3 + z^4} \tag{31.2-10d}
\]

Note that the functional equation $\exp(-z) = \frac{1}{\exp(z)}$ holds for the diagonal approximants. In general, we have $P_{[i,j]}(-z) = 1/P_{[j,i]}(z)$. This can be seen from the following closed form for the Padé approximants:

\[
P_{[i,j]}(z) = \frac{\sum_{k=0}^{i} \frac{i!}{(i+j)!} z^k}{\sum_{k=0}^{j} \frac{j!}{(i+j)!} (-z)^k} \tag{31.2-11}
\]

The numerator for $i = j$, multiplied by $(2i)!/i!$ in order to avoid rational coefficients, equals

\[
\frac{(2i)!}{i!} \cdot \sum_{k=0}^{i} \frac{i!}{(2i)!} z^k \tag{31.2-12}
\]

The coefficients of the numerator and denominator in the diagonal approximant

\[
P_{[i,i]} = \frac{\sum_{k=0}^{i} c_k z^k}{\sum_{k=0}^{i} c_k (-z)^k} \tag{31.2-13}
\]

can be computed using $c_i = 1$ (the coefficient of the highest power of $z$) and the recurrence

\[
c_k = c_{k+1} \frac{(k+1)(2i-k)}{(i-k)} \tag{31.2-14}
\]

It is usually preferable to generate the coefficients in the other direction. To do so compute the constant $c_0$

\[
c_0 = \prod_{w=1}^{i} 4w - 2 = 2, 12, 120, 1680, 30240, \ldots \tag{31.2-15}
\]

and use the recurrence

\[
c_k = c_{k-1} \frac{i-k}{(2i-k)(k+1)} \tag{31.2-16}
\]

We generate the coefficients for $1 \leq i \leq 8$:

\[
\begin{align*}
&\textbf{? c0(i)=prod(w=1,i,4*w-2)} \\
&\textbf{? qq(i,k)=(i-k)/(2*i-k)*(k+1)} \\
&\textbf{? for (i=1, 8, c=c0(i)); print1("["i,"","i,"": ""); \} \\
&\textbf{? for (k=0, i, print1(" ", c); c=qq(i,k)); print();)} \\
&[1,1]: 2 1 \\
&[2,2]: 12 6 1 \\
&[3,3]: 120 60 12 1 \\
&[4,4]: 1680 840 180 20 1 \\
&[5,5]: 30240 15120 3360 420 30 1 \\
&[6,6]: 665280 332640 75600 10080 840 42 1 \\
&[7,7]: 17297280 8648640 1995840 277200 25200 1512 56 1 \\
&[8,8]: 518918400 259459200 60540480 8648640 831600 55440 2520 72 1
\end{align*}
\]
Finally, the Padé approximant $P_{[i,j]}$ can be expressed as ratio of hypergeometric functions:

$$P_{[i,j]} = \frac{F\left(\frac{-i}{-i-j} | z \right)}{F\left(\frac{-j}{-i-j} | -z \right)} \quad (31.2-17)$$

This is relation 35.2-53 on page 703 with $a = -i$ and $b = -j$ where $i$ and $j$ are positive integers.

### 31.2.3 Argument reduction for the exponential function

As for the logarithm an argument reduction technique can be useful with moderate precisions. We do not use the functional equation for the exponential functional ($\exp(2z) = \exp(z)^2$) because the loss of precision when adding up the Taylor series (one plus a tiny quantity). Instead we use the functional equation for $E(z) := \exp(z) - 1$:

$$E(2z) = 2E(z) + E^2(z) \quad (31.2-18)$$

Compute the exponential function of $z$ as follows

1. set $r = z/2^n$ (for $n$ big enough)
2. compute $E := \exp(r) - 1$ to the desired precision using the Taylor series
3. repeat $n$ times: $E = 2E + E^2$
4. return $E + 1$

We compute $\exp(1.0)$ using $n = 16$ and eight terms of the Taylor series:

```plaintext
? z=1.0;  \( z = 1.0 \)
? n=16;   \( n = 16 \)
? r=z/2^n \( r = z/2^n \)
0.0000152587890625000000000000000000000000000000000000000
? E=exp(r)*(1+r/2*(1+r/3*(1+r/4*(1+r/5*(1+r/6*(1+r/7*(1+r/8)))))))) \( E = \exp(r) \times \text{Taylor series} \)
0.000015258905478413984814004262248066173018701234845511622583
? for(k=1,n,E=2*E+E^2);E=E+1 \( \text{repeat } n \text{ times: } E = 2E + E^2 \)
2.718281828459045235360287471352662497757274709366995975
```

One can also compute the exponential function via the hyperbolic cosine using

$$\exp(z) = \cosh(z) + \sinh(z) = \cosh(z) + \sqrt{\cosh^2(z) - 1} \quad (31.2-19a)$$

Alternatively, compute hyperbolic sine and use

$$\exp(z) = \sinh(z) + \sqrt{\sinh(z)^2 + 1} \quad (31.2-19b)$$

The advantage is that half of the coefficients of the Taylor series are zero. Again we do not use the function equation for the cosine ($\cosh(2z) = 2 \cosh^2(z) - 1$) but that for $C(z) := \cosh(z) - 1$:

$$C(2z) = 2(C(z) + 1)^2 - 2 \quad (31.2-20)$$

Compute the hyperbolic cosine as follows

1. set $r = z/2^n$ (for $n$ big enough)
2. compute $C := \cosh(r) - 1$ to the desired precision using the Taylor series
3. repeat $n$ times: $C = 2[C - 1]^2 - 2$
4. return $C + 1$

We compute $\cosh(1.5)$ using $n = 16$ and four terms of the Taylor series:

```
? for(k=1,n,C=2*(C-1)^2-2);C=C+1 \( \text{repeat } n \text{ times: } C = 2(C - 1)^2 - 2 \)
2.718281828459045235360287471352662497757274709366995975
```
Chapter 31: Logarithm and exponential function

Let $f(x)$ be a power series in $x$ and $g(x) = \log(f(x))$. Then we have $\frac{dg(x)}{dx} = \frac{f'(x)}{f(x)}$. Thereby, symbolically,

$$g(x) = \log(f(x)) = \int \frac{f'(x)}{f(x)} \, dx$$

(31.3-1)

A few lines of pari/gp to demonstrate this:

? sp=8;default(seriesprecision,sp+1);
? f=taylor((1)/(1-x-x^2),x) /* shifted Fibonacci (with constant term) */
1 + x + 2*x^2 + 3*x^3 + 5*x^4 + 8*x^5 + 13*x^6 + 21*x^7 + 34*x^8 + O(x^9)
? d=deriv(f,x)
1 + 4*x + 9*x^2 + 20*x^3 + 40*x^4 + 78*x^5 + 147*x^6 + 272*x^7 + O(x^8)
? q=d/f /* the only nontrivial computation */
1 + 3*x + 4*x^2 + 7*x^3 + 11*x^4 + 18*x^5 + 29*x^6 + 47*x^7 + O(x^8)
? l=informal(q)
x + 3/2*x^2 + 4/3*x^3 + 7/4*x^4 + 11/5*x^5 + 3*x^6 + 29/7*x^7 + 47/8*x^8 + O(x^9)
? f=exp(l) /* check with builtin exp() */
0(x^9)

31.3.2 Inverse trigonometric functions

Now let $a(x) = \arctan(f(x))$. Then, symbolically,

$$a(x) = \int \frac{f'(x)}{1 + f(x)^2} \, dx$$

(31.3-2)

Verification for the trivial case $f(x) = x$: 

[fxtbook draft of 2008-August-17]
31.3: Logarithm and exponential function of power series

\[ \text{For } s(x) = \arcsin(f(x)) \text{ use} \]

\[ s(x) = \int \frac{f'(x)}{\sqrt{1 - f(x)^2}} \, dx \]  \hfill (31.3-3)

### 31.3.3 Exponential function

With \( e(x) = \exp(f(x)) \) we can use a scheme similar to those shown in section 28.7 on page 586. We express a function \( g(y) \) as

\[ g(y) = \prod_{k=1}^{\infty} [1 + T(Y_k)] \]  \hfill (31.3-4)

where \( Y_1 = y \), \( Y_{k+1} = N(Y_k) \) and \( 1 + T(y) \) is the truncated Taylor series of \( g \). A second order product is obtained by taking \( 1 + T(y) = 1 + y \) (the series of \( \exp(y) \) truncated before the second term) and

\[ N(y) = f^{-1}\left(\frac{f(y)}{1 + T(y)}\right) \]  \hfill (31.3-5)

For \( g(y) = \exp(y) \) we obtain \( N(y) = y - \log(1 + y) \) and

\[ \exp(y) = \prod_{k=1}^{\infty} [1 + Y_k] \]  \hfill (31.3-6)

where \( Y_1 = y = f(x) \) and \( Y_{k+1} = Y_k - \log(1 + Y_k) \). The product \( \prod_{k=1}^{N} \) is correct up to order \( y^{2N-1} \). The computation involves \( N - 2 \) logarithms and \( N - 1 \) multiplications. Implementation in pari/gp:

```python
1  texp(y, N=5)=
2  { local(Y, e, t);
3    Y = y; e = 1 + Y;
4    for (k=2, N,
5      t = deriv(1+Y,x)/(1+Y);
6      t = intformal(t); \ here: t = log(1+Y);
7      Y -= t;
8      e *= (1+Y);
9    );
10   return( e );
11 }
```

Check:

```plaintext
? f=taylor((x)/(1-x-x^2),x)
\[ x + x^2 + 2*x^3 + 3*x^4 + 5*x^5 + 8*x^6 + 13*x^7 + 21*x^8 + \ldots \]
? e=exp(f) /* builtin exp() */
\[ 1 + x + 3/2*x^2 + 19/6*x^3 + 145/24*x^4 + 467/40*x^5 + 16051/720*x^6 + \ldots \]
? t=texp(f,4);
\[ t=exp(f,4);
```

...
31.4 Simultaneous computation of logarithms of small primes

We describe a method to compute the logarithms of a given set of (small) primes simultaneously. We use a method similarly to the one for finding arctan-relations for \( \pi \) given in section 30.5 on page 620. We define

\[
L(z) := 2 \arccoth(z) = 2 \sum_{k=0}^{\infty} \frac{1}{(2k+1)z^{2k+1}}
\]

and note that (relation 35.2-8 on page 709)

\[
\log(z) = 2 \arccoth \frac{z+1}{z-1}
\]

We will determine a set of relations that express the logarithm of a prime as linear combination of terms \( L(X_i) \) where the \( X_i \) are large integers so that the series for \( L \) converges fast.

\[
S = \{ 5174295, 170918749, 265326335, 287083066, 362074049, 587270881, \\
61783151, 740512499, 831409151, 1752438401, 2151548801, 2470954914, 3222617399 \}
\]

\[
2: \{-1595639, -17569128, -8662593, -31112926, -13108464, -11209640, -12907342, +9745611, -1705229, -12058985, +4580610, +4775383, -12972664 \}
\]

\[
3: \{-2529028, -27846409, -13729885, -49312821, -20776424, -17766859, -20457653, +15446428, -2702724, -19113039, +7260095, +7568803, -20561186 \}
\]

\[
5: \{-3704959, -40794252, -20113918, -72241977, -30436911, -26027978, -2996920, +22628608, -3959419, -28000096, +10635847, +11088096, -30121593 \}
\]

\[
7: \{-4479525, -49322778, -24318973, -87345026, -36800111, -31469438, -36235490, +27359389, -4787183, -33853851, +12869098, +13406195, -36418872 \}
\]

\[
11: \{-5520004, -60779197, -29967340, -107633040, -45347835, -39767983, -44662067, +33714275, -5899123, -36859389, +12974958, +13406195, -36418872 \}
\]

\[
13: \{-6522115, -71813158, -35408027, -127172929, -53580360, -4818987, -52758281, +39834823, -72628724, -19159201, +53025828 \}
\]

\[
17: \{-7217972, -79475039, -39185776, -140741248, -59296949, -50707501, -58387161, +44088875, -7713709, -54549566, -20720673, +21601741, -58682649 \}
\]

\[
23: \{-7751584, -85350490, -42087212, -151146003, -63680669, -54456218, -62703622, +47343993, -8283970, -49290653, +18723073, +19519201, -53025828 \}
\]

\[
29: \{-904566, -65013499, -32055403, -115131507, -48507081, -41480597, -47762841, +36063046, -6310097, -44623547, +16960051, +17671017, -48004613 \}
\]

\[
37: \{-950596, -60779197, -29967340, -107633040, -45347835, -39767983, -44662067, +33714275, -5899123, -36859389, +12974958, +13406195, -36418872 \}
\]

\[
41: \{-9843719, -102172929, -53580360, -217127929, -95380360, -4818987, -52758281, +39834823, -72628724, -19159201, +53025828 \}
\]

Figure 31.4-A: Relations for the fast computation of the logarithms of the primes up to 41.

Compute \( \log(p_i) \) for the primes \( p_i \) in a predefined set \( P \) of \( n \) primes as follows:

1. Find a set \( S \) of numbers \( X \in \mathbb{Z} \) so that \( X^2 - 1 \) factor completely into the primes in \( P \).
2. Select a subset \( S \) of \( n \) (large) numbers \( X_k \) so that all \( L(X_k) \) are linearly independent.
3. Try to find, for each prime \( p_i \), a relation \( \log(p_i) = \sum_{j=1}^{n} m_j L(X_j) \). If this fails return to step 2.

For example, with the first 13 primes \( P = \{ 2, 3, 5, 7, 11, \ldots, 41 \} \) one can find

\[
S = \{ X_1, X_2, \ldots, X_{13} \} = \{ 5174295, 170918749, 265326335, 287083066, 362074049, 587270881, \\
61783151, 740512499, 831409151, 1752438401, 2151548801, 2470954914, 3222617399 \}
\]
We use the short form \( p: [m_1, m_2, m_3, \ldots, m_{13}] \) to denote a relation

\[
\log(p) = \sum_{j=1}^{13} m_j L(X_j) \quad (31.4-4)
\]

Now we have the relations given in figure 31.4-A, the first one is

\[
\log(2) = -1595639 L(51, 744, 295) - 17569128 L(170, 918, 749) \pm \ldots - 12972664 L(3, 222, 617, 399)
\]

Note that the series with slowest convergence already gives more than 15 digits per term (\( \log_{10}(51, 744, 295^2) \approx 15.4 \)), and the last series gives 19 digits per term.

<table>
<thead>
<tr>
<th>[51744295]</th>
<th>[51744295]</th>
<th>[-2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[170918749]</td>
<td>[170918749]</td>
<td>[+1, +4, -5, +1, +1, +1, 0, -1, -1, +1, 0, -1]</td>
</tr>
<tr>
<td>[265326335]</td>
<td>[265326335]</td>
<td>[-7, -2, 0, +1, -1, +1, 0, -2, 0, -1, +2, +1, +1]</td>
</tr>
<tr>
<td>[287080366]</td>
<td>[287080366]</td>
<td>[0, +1, +1, -5, +2, +1, 0, -1, +3, -1, -1, 0, 0]</td>
</tr>
<tr>
<td>[362074049]</td>
<td>[362074049]</td>
<td>[+5, -4, -2, +1, 0, -2, 0, 0, -2, +2, +2, 0, 0]</td>
</tr>
<tr>
<td>[587270881]</td>
<td>[587270881]</td>
<td>[+4, +2, +1, +3, -1, 0, -1, 0, 0, +1, -1, -3, +1]</td>
</tr>
<tr>
<td>[617831551]</td>
<td>[617831551]</td>
<td>[-6, +4, +2, +1, 0, -6, 0, +1, 0, 0, +1, +1, 0]</td>
</tr>
<tr>
<td>[740512499]</td>
<td>[740512499]</td>
<td>[-1, -1, -5, -2, +7, -1, 0, +1, 0, 0, -1, 0, 0]</td>
</tr>
<tr>
<td>[831409151]</td>
<td>[831409151]</td>
<td>[-9, -1, +2, -1, +3, +1, 0, 0, -1, 0, +2, 0, -2]</td>
</tr>
<tr>
<td>[1752438401]</td>
<td>[1752438401]</td>
<td>[+6, -4, +2, 0, -2, -2, 0, +2, -2, 0, 0, +1, +1]</td>
</tr>
<tr>
<td>[2151548801]</td>
<td>[2151548801]</td>
<td>[+6, -2, +2, 0, 0, -2, 0, 0, +2, -4, +1, 0, +1]</td>
</tr>
<tr>
<td>[2470944914]</td>
<td>[2470944914]</td>
<td>[0, 0, -1, +1, -3, -5, +2, 0, 0, +3, 0, +1]</td>
</tr>
<tr>
<td>[3222617399]</td>
<td>[3222617399]</td>
<td>[-2, -6, -2, +4, +1, +2, 0, +2, -1, 0, -2, 0, 0]</td>
</tr>
</tbody>
</table>

**Figure 31.4-B:** Values \( L(x) \) as linear combinations of logarithms of small primes.

Figure 31.4-B shows the linear combinations of logarithms of small primes that give the values \( L(x) \). The first row is the relation

\[
L(51, 744, 295) = -2 \log(2) + 2 \log(3) + 3 \log(7) \pm \ldots - 1 \log(41) \quad (31.4-5)
\]

The shown values, as a matrix, are the inverse the values in figure 31.4-A.

Precomputed logarithms of small primes can be used for the computation of the logarithms of integers \( k \) if one can determine a smooth number near \( k \). For example, the logarithm of 65537 (a prime) can be computed as

\[
\log(65537) = \log \left( \frac{65537}{65536} \right) = \log \left( \frac{65537}{65536} \right) + \log(65536) = \log \left( 1 + \frac{1}{65536} \right) + 16 \log(2) \quad (31.4-6a)
\]

The series of the first logarithm converges fast and \( \log(2) \) is precomputed. This idea has been given by Jim White [priv.comm.]. If \( k \) is not near a smooth number but \( u \cdot k \) is smooth where \( u \) factors into the chosen prime set, one can use the relation

\[
\log(k) = \log(u k) - \log(u) \quad (31.4-7)
\]

Here \( \log(u) \) is the sum of precomputed logarithms and with \( \log(u k) \) one can proceed as above.
Chapter 32

Numerical evaluation of power series

We give algorithms for the numerical evaluation of power series. When the series coefficients are rational the binary splitting (binsplit) algorithm can be applied for rational arguments, and the rectangular schemes with real (full-precision) arguments. As a special case of the binary splitting algorithm, a method for fast radix conversion is described. Finally we describe a technique for the summation of series with alternating coefficients.

32.1 The binary splitting algorithm for rational series

The straightforward computation of a series for which each term adds a constant amount of precision (for example, the arc-cotangent series with arguments \( > 1 \)) to a precision of \( N \) digits involves the summation of proportional \( N \) terms. To get \( N \) bits of precision one has to add proportional \( N \) terms of the sum, each term involves one (length-\( N \)) short division (and one addition). Therefore the total work is proportional \( N^2 \), which makes it impossible to compute billions of digits from linearly convergent series even if they are as ‘good’ as Chudnovsky’s famous series for \( \pi \) (given in [85]):

\[
\frac{1}{\pi} = \frac{641681608}{\sqrt{640320}} \sum_{k=0}^{\infty} \frac{(13591409 + k)(6k)!}{(k!)^3 (3k)!} \frac{(-1)^k}{640320^3k} 
\]

(32.1-1a)

\[
\frac{12}{\sqrt{640320}} \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!}{(k!)^3 (3k)!} \frac{13591409 + 545140134 \cdot k}{(640320)^3k} 
\]

(32.1-1b)

32.1.1 Binary splitting scheme for products

We motivate the binsplit algorithm by giving the analogue for the fast computation of the factorial. Define \( f_{m,n} := m \cdot (m+1) \cdot (m+2) \cdots (n-1) \cdot n \), then \( n! = f_{1,n} \). We compute \( n! \) by recursively using the relation \( f_{m,n} = f_{m,x} \cdot f_{x+1, n} \) where \( x = \lfloor (m+n)/2 \rfloor \):

```plaintext
1 indent(i)=for(k=1,8*i,print1(" ")); \ aux: print 8*i spaces
2 1
3 F(m, n, i=0)=
4 { /* Factorial, self documenting */
5    local(x, ret);
6    indent(i); print("F(", m, ", ", n, ", ")");
7    if (m==n, /* then: */
8      ret = m; /* then: */
9      x = floor((m+n)/2);
10     ret = F(m, x, i+1) * F(x+1, n, i+1);
11    );
```

[fxtbook draft of 2008-August-17]
Chapter 32: Numerical evaluation of power series

Figure 32.1-A: Quantities with the computation of 8!.

The function prints the intermediate values occurring in the computation. The additional parameter \( i \) keeps track of the calling depth, used with the auxiliary function \( \text{indent}() \). Figure 32.1-A shows the output with the computation of 8! = \( F(1, 8) \). A fragment like

\[
F(5, 6)
\]

\[
F(5, 5)
\]

\[
F(6, 6)
\]

\[
F(7, 8)
\]

\[
F(7, 7)
\]

\[
F(8, 8)
\]

\[
\text{``} == 12
\]

\[
\text{``} == 30
\]

\[
\text{``} == 56
\]

\[
\text{``} == 40320
\]

says “\( F(5, 6) \) called \( F(5, 5) \) [which returned 5], then called \( F(6, 6) \) [which returned 6]. Then \( F(5, 6) \) returned 30.” For the computation of other products modify the line \( \text{ret} = \text{m} \) as indicated in the code.

32.1.2 Binary splitting scheme for sums

For the evaluation of a sum \( \sum_{k=0}^{N-1} a_k \) we use the ratios \( r_k \) of consecutive terms:

\[
r_k := \frac{a_k}{a_{k-1}}
\]  

(32.1-2)

Set \( a_{-1} := 1 \) to avoid a special case for \( k = 0 \). One has

\[
\sum_{k=0}^{N-1} a_k = r_0 (1 + r_1 (1 + r_2 (1 + r_3 (1 + \ldots (1 + r_{N-1}) \ldots))))
\]  

(32.1-3)

Now define

\[
r_{m,n} := r_m (1 + r_{m+1} (\ldots (1 + r_n) \ldots)) \quad \text{where} \quad m < n
\]  

(32.1-4a)

\[
r_{m,m} := r_m
\]  

(32.1-4b)

then

\[
r_{m,n} = \frac{1}{a_{m-1}} \sum_{k=m}^{n} a_k
\]  

(32.1-5)
32.1: The binary splitting algorithm for rational series

and especially

\[ r_{0,n} = \sum_{k=0}^{n} a_k \]  

(32.1-6)

We have

\[ r_{m,n} = r_m + r_m \cdot r_{m+1} + r_m \cdot r_{m+1} \cdot r_{m+2} + \ldots \]

\[ \cdots + r_m \cdot \ldots \cdot r_x + r_m \cdot \ldots \cdot r_x \cdot [r_{x+1} + \ldots + r_{x+1} \cdot \ldots \cdot r_n] \]

\[ = r_{m,x} + \prod_{k=m}^{x} r_k \cdot r_{x+1,n} \]  

(32.1-7b)

The product telescopes, one gets (for \( m \leq x < n \))

\[ r_{m,n} = r_{m,x} + \frac{a_x}{a_{m-1}} \cdot r_{x+1,n} \]  

(32.1-8)

32.1.3 Implementation using rationals

![Figure 32.1-B](image)

Now we can formulate the binary splitting algorithm by giving a binsplit function using pari/gp:

```plaintext
1 R(m, n)=
2 { /* Rational binsplit */
3   local(x, ret);
4   if ( m==n, /* then: */
5     ret = A(m)/A(m-1);
6   , /* else: */
7     x = floor( (m+n)/2 );
8     ret = R(m, x) + A(x) / A(m-1) * R(x+1, n);
9   );
10   return( ret );
11 }
```

Here \( A(k) \) must be a function that returns the \( k \)-th term of the series we wish to compute, in addition one must have \( a(-1)=1 \). For example, to compute \( \arctan(1/10) \) one would use

\[ A(k)=if(k<0, 1, (-1)^k/(2\cdot k+1)\cdot 10^{-2\cdot (2\cdot k+1)}) \]

Figure 32.1-B shows the intermediate values with the computation of \( \sum_{k=0}^{6} 2^{-(k+1)} = 127/128 \).
32.1.4 Implementation using integers

In case the programming language used does not provide rational numbers one needs to rewrite formula [32.1-8] in separate parts for denominator and numerator. With

$$ a_i = p_i/q_i, \; p_{-1} = q_{-1} = 1 $$

and

$$ r_{m,n} = U_{m,n}/V_{m,n} $$

one gets

$$ U_{m,n} = p_{m-1} q_m U_{m,x} V_{x+1,n} + p_x q_{m-1} U_{x+1,n} V_{m,x} \tag{32.1-9a} $$

$$ V_{m,n} = p_{m-1} q_m V_{m,x} V_{x+1,n} \tag{32.1-9b} $$

The following implementation also contains code for reduction to lowest terms:

```python
Q(m, n) =
{ /* Integer binsplit */
  \local(x, ret, bm, bx, tm, tx);
  if (m==n, /* then: */
    bm = B(m); bx = B(m-1);
    ret = \[ bm[1]*bx[2], bx[1]*bm[2] \]; \ \== \ B(m)/B(m-1);
    x = gcd(ret[1], ret[2]); /* Reduction */
    ret = \[ret[1]/x, ret[2]/x\]; /* Reduction */
  , /* else: */
    x = floor((m+n)/2);
    tm = Q(m, x); \ \U_{m,x}, V_{m,x}\]
    tx = Q(x+1, n); \ \U_{x+1,n}, V_{m,n}\]
    bm = B(m-1); \ \[p_{m-1}, q_{m-1}\]
    bx = B(x); \ \[p_x, q_x\]
    \ret == Q(m, x) + B(x) / B(m-1) * Q(x+1, n);
    \(bm[1]*bx[2]*tm[2]*tx[2])/10 \];
    x = gcd(ret[1], ret[2]); /* Reduction */
    ret = \[ret[1]/x, ret[2]/x\]; /* Reduction */
  );
  return( ret );
}
```

The reduction step can do good or bad, depending on the terms of the sum. When computing \(\arctan(1/10)\) without the reduction, the intermediate quantities grow exponentially, as shown in figure 32.1-C. The square brackets are the quantities \([U_{m,n}, V_{m,n}]\). Such explosive growth will occur with all Taylor series unless the function argument equals one.

32.1.5 Performance

We compute the sum for \(\arctan(1/10)\) up to the 5,000th term with the direct method, the rational binsplit and the integer binsplit with and without reduction. The timings for the computation are:

- \(A(k) = \text{if}(k<0, 1, (-1)^{(k)}/(2k+1)*10^{/(2k+1)}))\); \ \for \ rational \ binsplit
- \(B(k) = \text{if}(k<0, [1,1], [[(-1)^{(k)}, (2k+1)*10^{/(2k+1)}])\)); \ \for \ integer \ binsplit
- \(N=5000\)
32.1. The binary splitting algorithm for rational series

\[ \text{sum}(k=0,N,A(k)); \ \text{// direct method: 69,385 ms.} \]
\[ R(0,N); \ \text{// rational binsplit: 2,532 ms.} \]
\[ Q(0,N); \ \text{// integer binsplit with gcd reduction: 4,152 ms.} \]
\[ Q(0,N); \ \text{// integer binsplit without gcd reduction: >8min, "forever"} \]

Things look quite different when computing the sum \[ \sum_{k=0}^{50,000} (-1)^k/(2k + 1)^2 \]. The intermediate quantities \( U \) and \( V \) have only small common factors, so it is better to omit the reduction step:

\[ B(k)=\text{if}(k<0, [1,1], [(-1)^k, (2k+1)^2]); \]
\[ A(k)=\text{if}(k<0,1,(-1)^k/(2k+1)^2); \]
\[ N=50000; \]
\[ \text{sum}(k=0,N,A(k)); \ \text{// direct method: 32,396 ms.} \]
\[ R(0,N); \ \text{// rational binsplit: 6,826 ms.} \]
\[ Q(0,N); \ \text{// integer binsplit with gcd reduction: 27,485 ms.} \]
\[ Q(0,N); \ \text{// integer binsplit without gcd reduction: 6,251 ms.} \]

With built-in routines for binsplit summation the advantage will be much more in favor than these figures suggest.

The reason why summation via binary splitting is better than the straightforward way is that its complexity is only \( O(\log NM(N)) \), where \( M(N) \) is the complexity of one \( N \)-bit multiplication (see [142]). If an FFT based multiplication algorithm is used (\( M(N) = N \log N \)) the work is \( O((\log N)^2 N) \) This means that sums of linear but sufficient convergence are again candidates for high precision computations. The algorithm should be implemented in the ‘depth first’ manner presented, and not via the naive pairs, pairs of pairs, etc. (breadth first) way. The reasons are better locality and less memory consumption. The naive way needs most memory after the first pass, when pairs have been multiplied.

32.1.6 Extending prior computations

The ratio \( r_{0,N-1} \) (that is, the sum of the first \( N \) terms) can be reused if one wants to evaluate the sum to a higher precision than before. To get twice the precision use

\[ r_{0,2N-1} = r_{0,N-1} + a_{N-1} \cdot r_{N,2N-1} \]  \hspace{1cm} (32.1-10)

This is formula [32.1-8] with \( m = 0, x = N - 1, n = 2N - 1 \). With explicit rational arithmetic:

\[ U_{0,2N-1} = q_{N-1} U_{0,N-1} V_{N,2N-1} + p_{N-1} U_{N,2N-1} V_{0,N-1} \]  \hspace{1cm} (32.1-11a)
\[ V_{0,2N-1} = q_{N-1} V_{0,N-1} V_{N,2N-1} \]  \hspace{1cm} (32.1-11b)

Thereby with the appearance of some new computer that can multiply two length-\( 2 \cdot N \) numbers (assuming the old model could multiply length-\( N \) numbers) we only need to combine the two ratios \( r_{0,N-1} \) and \( r_{N,2N-1} \) that had been precomputed by the last generation of computers. This costs only a few full-size multiplications, so we can improve on prior computations cheaply.

32.1.7 Computation of \( \pi \): binary splitting versus AGM-type iterations

Using formula [32.1-1a] on page 641 and the binary splitting scheme for the computation of \( \pi \) can outperform the AGM-style iterations given in section 30.4 on page 612. The reason is that the memory access pattern is more favorable than with the iterations. When computing \( N \) digits of \( \pi \) the iterations compute proportional \( \log_2(N) \) roots (and or inverses) to full precision. At the last phase of each root computation full-length multiplications have to be computed. These access all memory storing a few full-precision words. In contrast, the binary splitting involves full-precision multiplications only at the very last phase.

The drawback of the binary splitting scheme is that it may need significantly more memory than two full words. This may happen if the numerator and denominator grow fast which is more likely if no series so favorable as [32.1-1a] can be used for the quantity to be computed. The problem can be mitigated by computing the floating point value whenever the integer values become too large (as pointed out by Richard Kreckel [priv.comm.], see [177], and [83]). This technique is used in the CLN library [141].
32.1.8 Fast radix conversion

A binary splitting scheme for radix conversion of a radix-\(z\) integer \([a_Na_{N-1}\ldots a_2a_1a_0]_z\) can be obtained via recursive application of the scheme

\[
\sum_{k=M}^{N} a_k z^k = \sum_{k=M}^{M+X-1} a_k z^k + z^X \sum_{k=M+X}^{N} a_k z^k
\]  

(32.1-12)

where \(X\) is chosen to be the largest power of two that is smaller than \(d := N - M\).

\[
\begin{align*}
\text{Figure 32.1-D: Intermediate results when converting the number } & 2107654321076543_{16} \text{ to decimal.} \\
R(0, 15) & \\
R(0, 7) & \\
R(0, 3) & \\
R(0, 1) & ^{=} = 67 \\
R(2, 3) & ^{=} = 101 \\
R(4, 7) & ^{=} = 25923 \\
R(4, 5) & ^{=} = 7 \\
R(6, 7) & ^{=} = 33 \\
R(8, 15) & ^{=} = 554132803 \\
R(8, 9) & ^{=} = 25923 \\
R(10, 11) & ^{=} = 101 \\
R(12, 15) & ^{=} = 7 \\
R(14, 15) & ^{=} = 33 \\
R(15, 15) & ^{=} = 554132803 \\
& ^{=} = 237998226709943491
\end{align*}
\]

We define an auxiliary function that computes (for \(d > 1\)) the largest exponent \(s\) so that \(2^s < d\):

```c
1 ex2le(d)=
2 { /* return largest s so that 2^s < d */
3    local(s, t);
4    t=1; s=0;
5    while ( d>t, t<<=1; s+=1; );
6    t >>= 1; s--;
7    return(s);
8 }
```

We precompute \(z^2, z^4, z^8, \ldots, z^{2^w}\) where \(2^w < N\):

```
N=15;
z=16; \// \text{radix}
vz=vector(ceil(log(N)/log(2)));
vz[1]=z;
for (k=2, length(vz), vz[k]=vz[k-1]^2); \// N space
```

Now the conversion function can be defined as

```
1 Ri(m, n, i=0)=
2 { /* Radix conversion, self documenting */
3    local(x, d, ret, t);
4    indent(i); print( "R("m", ", ", n", ")\n");
5    d = n-m;
6    if ( d <= 1, /* then: */
7        if ( d==0, ret = A(m); , ret = A(m) + z*A(n); );
8    , /* else: */
9    t = ex2le(d);
```
32.2: Rectangular schemes for evaluation of power series

We compute temporaries \(z\) by precomputing the quantities \(z\), \(z^2\), \(z^3\), \(z^4\), ... \(z^C\) involving \(C - 1\) long multiplications. The sums in each row of expression 32.2-2 involve only short multiplications with series coefficients \(A_i\). The multiplication by \(z^C\) for each but the first row involves further \(R - 1\) long multiplications. The computation uses \(C\) temporaries \((z, z^2, \ldots, z^C)\) and proportional \(R + C\) long multiplications. Choosing \(R = C = \sqrt{N}\) leads to a complexity of \(2\sqrt{N}\) long multiplications, and also involves \(\sqrt{N}\) temporaries.

32.2.1 Implementation for arctan

We implement the scheme for the arctan in pari/gp:

```plaintext
We define \(A(k) = (k+3)\%8\); and convert the 16-digit, radix-16 number \([a_{15}a_{14} \ldots a_2a_1a_0]_{16} := 2107654321076543\) to decimal. The intermediate results are shown in figure 32.1-D.

32.2 Rectangular schemes for evaluation of power series

The rectangular scheme for the evaluation of polynomials was given in [216], and later in [246]. We use it for the evaluation of truncated power series up to a given power \(N - 1\) of the series variable. We give two variants, one for series whose coefficients are small rationals (as for the logarithm), and another for series where the ratios of successive coefficients are a small rationals (as for the exponential function). When the numbers of rows and columns in the schemes are identical, a method involving proportional \(\sqrt{N}\) full-precision multiplications is obtained. The schemes are very competitive up to very high precision in practice, even compared with AGM-based methods.

32.2.1 Rectangular scheme for arctan and logarithm

Computing the sum of the first \(N\) terms of a power series as

\[
S_N := \sum_{k=0}^{N-1} A_k z^k = A_0 + z (A_1 + z (A_2 + z (A_3 + \ldots z (A_{N-1} \ldots))) \quad (32.2-1)
\]

costs \(N\) long (full-precision) multiplications if \(z\) is a full-precision number. If the \(A_k\) are small rational values, and \(N = R \cdot C\) then one can rewrite \(S_N\) as

\[
S_N = A_0 C + A_0 C + z + A_0 C + z^2 + \ldots + A_0 C - z^{C-1} +
+ z^C \left[A_1 C + A_1 C + z + A_1 C + z^2 + \ldots + A_2 C - z^{C-1} +
+ z^C \left[A_2 C + A_2 C + z + A_2 C + z^2 + \ldots + A_3 C - z^{C-1} +
+ \ldots \right. \right.
+ z^C \left[A_{R-1} C + A_{R-1} C + z + A_{R-1} C + z^2 + \ldots + A_{R-1} C - z^{C-1} ] \ldots]]
\]

We compute \(S_N\) as

\[
\left[\left[\ldots [U_{R-1}] z^C + \ldots + U_3 \right] z^C + U_2 \right] z^C + U_1 \right] z^C + U_0 \quad (32.2-3)
\]

where \(U_r := \sum_{k=0}^{C-1} A_r C + z^k\) is the sum in one row of relation 32.2-2.

Precomputing the quantities \(z^2\), \(z^3\), \(z^4\), ... \(z^C\) involves \(C - 1\) long multiplications. The sums in each row of expression 32.2-2 involve only short multiplications with series coefficients \(A_i\). The multiplication by \(z^C\) for each but the first row involves further \(R - 1\) long multiplications. The computation uses \(C\) temporaries \((z, z^2, \ldots, z^C)\) and proportional \(R + C\) long multiplications. Choosing \(R = C = \sqrt{N}\) leads to a complexity of \(2\sqrt{N}\) long multiplications, and also involves \(\sqrt{N}\) temporaries.

32.2.1.1 Implementation for arctan

We implement the scheme for the arctan in pari/gp:
fa(n) = \ inverse of series coefficient
1 { /* fa(n) := (-1)^n/(2n+1) */
2    local(an);
3    an = (2*n+1);
4    if ( bitand(n,1), an=-an);
5    return( an );
6 }
7
8 atan_rect(z, R, C)=
9 { /* compute atan(z) as z*(1-z^2/3+z^4/4-z^6/5+...+-z^(2*(R*C-1))/(2*R*C-1) */
10    local(S, vz, s, ur, k);
11    vz = vector(C); \ vz == [z^2,z^4,z^6,...,z^(2*C)]
12    for (k=2, C, vz[k]=vz[1]*vz[k-1]); \ C-1 long multiplications
13    k = R*C; \ index of current coefficient
14    s = 0; \ sum
15    forstep (r=R-1, 0, -1,
16      ur = 0; \ sum of this row
17      forstep (c=C-1, 1, -1, k-=1; ur+=vz[c]/fa(k); );
18      k -= 1; ur += 1/fa(k);
19      if ( r!=R-1, s*=vz[C]; ); \ R-1 long multiplications
20      s += ur;
21    );
22    s *= z; \ 1 long multiplication (special for arctan)
23    return( s );
24 }
25
26 We compute \( \pi/16 \) as \( \arctan(z) \) where \( z = \sqrt{2}/\sqrt{2 + 4 - \sqrt{2} - 1} \approx 0.19891236 \) (using relation 31.1-26 on page 632 twice on \( z = 1 \)), using a precision of 30,000 decimal digits. We use \( R = C = \sqrt{N} =: S \):
27
28 \[ \pi/16 \approx \text{atan} \left( \sqrt{2}/\sqrt{2 + 4 - \sqrt{2} - 1} \right) \approx 0.19891236 \]
29 with \( R = C = S = 147 \), and \( N = S^2 = 21609 \).
30
31 ? a=atan(z); \ builtin arctan: computed in 1,123 ms.
32 ? r=atan_rect(z,S,S); \ computed in 2,377 ms.
33 ? a-r
34 0.E-30017 \ result OK
35
36 The given implementation involves about two times of the cost of the builtin routine. Argument reduction make the method much more competitive:
37
38 \[ \pi/16 \approx \text{atan} \left( \frac{\sqrt{2}}{\sqrt{2 + 4 - \sqrt{2} - 1}} \right) \]
39 with \( R = C = S = 147 \), and \( N = S^2 = 21609 \).
40
41 ? a=atan(z); \ computed in 1,123 ms.
42 ? z=1/z; \ for(k=1,32, z=1/z) \ computed in 204 ms.
43 ? z=1/z
44 \%45 = 1.8690050398194919519 E-30016 \ result OK
45
46 With 100,000 decimal digits the performance ratio is roughly the same. Note that one will have to limit the number \( C \) of temporaries according to the available memory.
47
48 Compute the inverse sine and cosine as \( \arcsin(z) = \arctan \left( \frac{z}{\sqrt{1-z^2}} \right) \) and \( \arccos(z) = \frac{\pi}{2} - \arcsin(z) \).
50
51 \[ \arcsin(z) = \arctan \left( \frac{z}{\sqrt{1-z^2}} \right) \]
52 \[ \arccos(z) = \frac{\pi}{2} - \arcsin(z) \]
53
54 32.2.1.2 Implementation for the logarithm
55
56 A routine for log(1 - z) is
57
58 log_rect(z, R, C)=
59 { /* compute log(1-z) as 1+x/2+x^2/3+...+x^(R*C-1)/(R*C) */
60    local(S, vz, s, ur, k);
61    vz = vector(C); \ vz == [z^2,z^4,z^6,...,z^(2*C)]
62    vz[1] = z;
63    for (k=2, C, vz[k]=vz[1]*vz[k-1]); \ C-1 long multiplications
64    k = R*C; \ index of current coefficient
65    s = 0; \ sum
66    forstep (r=R-1, 0, -1,
67      ur = 0; \ sum of this row
68      forstep (c=C-1, 1, -1, k-=1; ur+=vz[c]/(k+1); );
69      if ( r!=R-1, s*=vz[C]; ); \ R-1 long multiplications
70      s += ur;
71    );
72    s
73    return( s );
74 }
75
76 \[ \log(1-z) = \sum_{k=1}^{R*C-1} \frac{z^k}{k} \]
77 \[ \log(1-z) = \sum_{k=1}^{R*C-1} \frac{z^k}{k} \]
78 \[ \log(1-z) = \sum_{k=1}^{R*C-1} \frac{z^k}{k} \]
79 \[ \log(1-z) = \sum_{k=1}^{R*C-1} \frac{z^k}{k} \]
80 \[ \log(1-z) = \sum_{k=1}^{R*C-1} \frac{z^k}{k} \]
32.2: Rectangular schemes for evaluation of power series

We rewrite the sum of the first \( N \) terms of a power series

\[
S_N := \sum_{k=0}^{N-1} A_k z^k
\]

as

\[
S_N = 1 \left[ A_{0C+0} + A_{1C+0} z^{1C} + A_{2C+0} z^{2C} + \ldots + A_{(R-1)C+0} z^{(R-1)C} \right] + z^1 \left[ A_{0C+1} + A_{1C+1} z^{1C} + A_{2C+1} z^{2C} + \ldots + A_{(R-1)C+1} z^{(R-1)C} \right] + z^2 \left[ A_{0C+2} + A_{1C+2} z^{1C} + A_{2C+2} z^{2C} + \ldots + A_{(R-1)C+2} z^{(R-1)C} \right] + \ldots + z^{C-2} \left[ A_{1C-2} + A_{2C-2} z^{1C} + A_{3C-2} z^{2C} + \ldots + A_{(R-1)C-2} z^{(R-1)C} \right] + z^{C-1} \left[ A_{1C-1} + A_{2C-1} z^{1C} + A_{3C-1} z^{2C} + \ldots + A_{(R-1)C-1} z^{(R-1)C} \right]
\]

Compute the sum as (the transposed version of relation 32.2-2 on page 647)

\[
S_N = \left[ [\ldots [U_{C-1}] z + U_{C-2}] z + \ldots + U_3 \right] z + U_2 z + U_1 z + U_0
\]

where \( U_c = \sum_{k=0}^{R-1} A_{kC+c} z^{kc} \) (\( C \) temporary sums are computed). When proceeding column-wise the update \( A_i \rightarrow A_{i+1} \) involves only a short multiplication by the ratio \( A_{i+1}/A_i \). Only when going to the next column a long multiplication by \( z^{C} \) is required (\( R - 1 \) long multiplications). Finally, there are \( C - 1 \) long multiplications by \( z \).
32.2.2.1 Implementation for the exponential function

A routine for the computation of \(\exp(z) - 1\) can be given as follows:

```c
exp_rect(z, R, C)=
{ /* compute exp(z)-1 as z*[ 1+z/2!+z^2/3! +...+z^(R*C-1)/((R*C)!) ] */
  local(ur, zc, k, t);
  zc = z^C; \ \ proportional log(C) long multiplications
  ur = vector(C);
  k = 1; \ \ ratio of series coefficients /* set to zero for plain exp */
  t = 1.0;
  for (r=1, R, \ \ number of columns (!)
    for (c=1, C, ur[c] += t; k++; t /= (k); k++);
    if ( r!=R, t *= zc; ); \ \ R-1 long multiplications
  );
  t = ur[C];
  forstep (c=C-1, 1, -1, t*=z; t+=ur[c]); \ C-1 long multiplications
  t *= z; /* omit for plain exp */
  return( t );
}
```

We use the argument reduction given as relation 31.2-18 on page 635 and compute \(\exp(1/5)\) to a precision of 30,000 decimal digits. We use \(R = C = \sqrt{N} =: S\):

```plaintext
? z=0.2;
? e=exp(z) \ computed in 855 ms.
1.22140275816016983392107199464
? nred=32;
? z/=2^nred
4.65661287307739257812500000000 E-11
? S=48; \ \ N=S^2=2304
? e=exp_rect(z,S,S) \ computed in 395 ms.
4.6566128731858127953752333467 E-11
? for(k=1,nred,r=r+r^2); \ \ computed in 68 ms.
? t=1-
0.22140275816016983392107199464
0.980066577841241631124196516748
```

Using 100,000 digits, \(nred=112\), and \(S=52\) we obtain the timings:

```plaintext
? e=exp(z); \ \ computed in 8,601 ms.
? e=exp_rect(z,S,S); \ \ computed in 2,345 ms.
? for(k=1,nred,r=r+r^2); \ \ computed in 1,640 ms.
```

32.2.2.2 Implementation for the cosine

A routine for computing \(\cos(z) - 1\) can be given as

```c
cos_rect(z, R, C)=
{ /* compute cos(z)-1 as z*[ -1/2!+z^2/4! - z^4/6! + ... ] */
  local(ur, zc, k, t);
  zc = z^C; \ \ proportional log(C) long multiplications
  ur = vector(C);
  k = 2; \ \ ratio of series coefficients
  t = -0.5;
  for (r=1, R, \ \ number of columns (!)
    for (c=1, C, ur[c] += t; k++; t /= -(k); k++);
    if ( r!=R, t *= zc; ); \ \ R-1 long multiplications
  );
  t = ur[C];
  forstep (c=C-1, 1, -1, t*=z; t+=ur[c]); \ C-1 long multiplications
  t *= z; /* omit for plain exp */
  return( t );
}
```

We use the argument reduction as in relation 31.2-20 on page 635 and compute \(\cos(1/5)\) to 30,000 decimal digits:

```plaintext
? z=0.2;
? e=cos(z) \ \ computed in 788 ms.
0.980066577841241631124196516748
? nred=32;
```

[fxtbook draft of 2008-August-17]
The sine and tangent can be computed as \( \sin(z) = \sqrt{1 - \cos(z)^2} \), and \( \tan(z) = \sin(z)/\cos(z) \).

The routine is easily converted to compute the hyperbolic cosine. The relation \( \exp(z) = \cosh(z) - \sqrt{\cosh(z)^2 - 1} \) gives an alternative way to compute the exponential function.

### 32.3 The magic sumalt algorithm for alternating series

The following convergence acceleration algorithm for alternating series is due to Cohen, Villegas and Zagier, see [92]. As remarked in the cited paper, the algorithm often gives meaningful results also for non-alternating and even divergent series.

The algorithm computes an estimate of the sum \( s = \sum_{k=0}^{\infty} x_k \) as

\[
s_n = \sum_{k=0}^{n-1} c_{n,k} x_k \tag{32.3-1}
\]

The weights \( c_{n,k} \) do not depend on the values \( x_j \). With the following pseudo code the summands \( x_k \) have to be supplied in the array \( x[0,1,...,n-1] \):

```plaintext
1 function sumalt(x[], n)
2 {
3     d := (3+sqrt(8))^n
4     d := (d+1/d)/2
5     b := 1
6     c := d
7     s := 0
8     for k:=0 to n-1
9         c := c - b
10        s := s + c * x[k]
11        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1))
12     }
13     return s/d
14 }
```

With alternating sums the accuracy of the estimate will be \( (3 + \sqrt{8})^{-n} \approx 5.82^{-n} \). For example, the estimate for \( 4 \cdot \arctan(1) \) using the first 8 terms is

\[
\pi \approx 4 \cdot \left( \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} \right) = 3.017\ldots \tag{32.3-2}
\]

The sumalt-massaged estimate is

\[
\pi \approx 4 \cdot \left( \frac{665856}{1} - \frac{665728}{3} + \frac{663040}{5} - \frac{641536}{7} + \frac{557056}{9} - \frac{376832}{11} + \frac{163840}{13} - \frac{32768}{15} \right) / 665857
\]

\[
= 4 \cdot \frac{3365266048/4284789795}{3.141592658\ldots}
\]

and already gives 7 correct digits of \( \pi \). The linear but impressive growth of the accuracy of successive sumalt estimates with \( n \), the number of terms used, is illustrated in figure 32.3-A.
Chapter 32: Numerical evaluation of power series

<table>
<thead>
<tr>
<th>n</th>
<th>sumalt(n)</th>
<th>sumalt(n)−π</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.6667026666666666666</td>
<td>0.474925986923126571795</td>
</tr>
<tr>
<td>2</td>
<td>3.13725490196078431373</td>
<td>0.00433775162900892437</td>
</tr>
<tr>
<td>3</td>
<td>3.1414564036373968348</td>
<td>0.00000610718611927014</td>
</tr>
<tr>
<td>4</td>
<td>3.1415234215659403660</td>
<td>−0.0000006906255253116447</td>
</tr>
<tr>
<td>5</td>
<td>3.14159044115659403660</td>
<td>0.000000088652253116447</td>
</tr>
<tr>
<td>6</td>
<td>3.14159044115659403660</td>
<td>−0.000000011634522625555</td>
</tr>
<tr>
<td>7</td>
<td>3.14159044115659403660</td>
<td>0.000000000031214482948</td>
</tr>
<tr>
<td>8</td>
<td>3.14159044115659403660</td>
<td>−0.000000000004503100007</td>
</tr>
<tr>
<td>9</td>
<td>3.14159044115659403660</td>
<td>0.000000000000658657944</td>
</tr>
<tr>
<td>10</td>
<td>3.14159044115659403660</td>
<td>−0.000000000000097480163</td>
</tr>
<tr>
<td>11</td>
<td>3.14159044115659403660</td>
<td>0.00000000000014574087</td>
</tr>
<tr>
<td>12</td>
<td>3.14159044115659403660</td>
<td>−0.0000000000002198312</td>
</tr>
<tr>
<td>13</td>
<td>3.14159044115659403660</td>
<td>0.000000000000334177</td>
</tr>
<tr>
<td>14</td>
<td>3.14159044115659403660</td>
<td>−0.000000000000051151</td>
</tr>
<tr>
<td>15</td>
<td>3.14159044115659403660</td>
<td>0.000000000000007877</td>
</tr>
<tr>
<td>16</td>
<td>3.14159044115659403660</td>
<td>−0.000000000000001220</td>
</tr>
<tr>
<td>17</td>
<td>3.14159044115659403660</td>
<td>0.000000000000000334177</td>
</tr>
<tr>
<td>18</td>
<td>3.14159044115659403660</td>
<td>−0.000000000000000051151</td>
</tr>
<tr>
<td>19</td>
<td>3.14159044115659403660</td>
<td>0.00000000000000007877</td>
</tr>
<tr>
<td>20</td>
<td>3.14159044115659403660</td>
<td>−0.0000000000000001220</td>
</tr>
</tbody>
</table>

Figure 32.3-A: Sumalt-estimates of \( \pi = 4 \cdot \arctan(1) \) using \( n = 1, 2, \ldots, 20 \) terms.

Therefore even slowly converging series like

\[
\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 4 \cdot \arctan(1) \quad (32.3-4a)
\]

\[
C = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2} = 0.9159655941772190 \ldots \quad (32.3-4b)
\]

\[
\log(2) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k+1} = 0.6931471805599453 \ldots \quad (32.3-4c)
\]

\[
\zeta(s) = \frac{1}{1 - 2^{1-s}} \sum_{k=1}^{\infty} \frac{(-1)^k}{k^s} \quad (32.3-4d)
\]

can be used to compute estimates that are correct up to thousands of digits. The algorithm scales like \( n^2 \) if the series terms in the array \( x[] \) are small rational values and like \( n^3 \cdot \log(n) \) if they are full precision (rational or float) values.

In fact, pari/gp has a built-in sumalt routine, we use it to compute the Catalan constant:

```p有名的p
\texttt{? default(realprecision,1000);}  \texttt{;}
\texttt{? sumalt(k=0,(-1)^k/(2*k+1)^2); \ taking 60 ms.}
\texttt{? default(realprecision,2000);}  \texttt{;}
\texttt{? sumalt(k=0,(-1)^k/(2*k+1)^2); \ taking 376 ms.}
\texttt{? default(realprecision,4000);}  \texttt{;}
\texttt{? sumalt(k=0,(-1)^k/(2*k+1)^2); \ taking 2,730 ms.}
```

The time scales roughly with the third power of the precision used.

All values \( c_k \) and \( b_k \) occurring in the computation are integers. In fact, the \( b_k \) in the computation with \( n \) terms are the coefficients of the expanded \( n \)-th Chebyshev polynomial (of the first kind, see section 34.2 on page 677) with argument \( 1 + 2x \):
The magic sumalt algorithm for alternating series

For the given series, the table shows the values of $b_k$ and $c_k$ for $k = 0$ to $8$.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$b_k$</th>
<th>$c_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>66557</td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td>665856</td>
</tr>
<tr>
<td>2</td>
<td>2688</td>
<td>665728</td>
</tr>
<tr>
<td>3</td>
<td>21504</td>
<td>663040</td>
</tr>
<tr>
<td>4</td>
<td>84480</td>
<td>641536</td>
</tr>
<tr>
<td>5</td>
<td>180224</td>
<td>557056</td>
</tr>
<tr>
<td>6</td>
<td>212992</td>
<td>376832</td>
</tr>
<tr>
<td>7</td>
<td>131072</td>
<td>163840</td>
</tr>
<tr>
<td>8</td>
<td>32768</td>
<td>32768</td>
</tr>
</tbody>
</table>

The formulas are:

\[
T_k(1 + 2x) = 1 + 128x + 2688x^2 + 21504x^3 + 84480x^4 + 180224x^5 + 212992x^6 + 131072x^7 + 32768x^8 = T_{16}(\sqrt{1 + x})
\]

\[
T_{16}(x) = 1 - 128x^2 + 2688x^4 - 21504x^6 + 84480x^8 - 180224x^{10} + 212992x^{12} - 131072x^{14} + 32768x^{16}
\]

Now observe that one has always $c_n = b_n = 2^{2n-1}$ in a length-$n$ sumalt computation. Obviously, ‘going backwards’ avoids the computation of $(3 + \sqrt{8})^n$:

```plaintext
function sumalt(x[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + c * x[k]
        b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }
    return s/c
}
```

The backward variant is:

```plaintext
function sumalt_partial(p[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + c * p[k]
        b := b * (2*(n+k)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }
    return s/d
}
```

The $b_k$ and $c_k$ occurring in a length-$n$ sumalt computation can be given explicitly as

\[
b_k = \frac{n}{n+k} \binom{n+k}{2k} 2^{2k}
\]  

\[
c_k = \sum_{i=k}^{n} \frac{n}{n+i} \binom{n+i}{2i} 2^{2i}
\]

To compute an estimate of $\sum_{k=0}^{\infty} x_k$ using the first $n$ partial sums use the following pseudo code (the partial sums $p_k = \sum_{j=0}^{k} x_j$ are expected in $p[0,1,\ldots,n-1]$):

```plaintext
function sumalt_partial(p[], n)
{
    d := (3+sqrt(8))^n
    d := (d+1/d)/2
    b := d
    c := 0
    for k:=0 to n-1
    {
        s := s + b * p[k]
        b := b * ((2*(n+k)*(n-k)) / ((2*k+1)*(k+1)))
    }
    return s/d
}
```

The backward variant is:

```plaintext
function sumalt_partial(p[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + c * p[k]
        b := b * (2*(n+k)*(k+1)) / ((2*k+1)*(k+1))
        c := c + b
    }
    return s/d
}
```
Chapter 32: Numerical evaluation of power series

```plaintext
for (k = 0; k < n; ++k) {
    s := s + b * p[k]
    b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
    c := c + b
}
return s/c
```

For series of already geometrical rate of convergence (where $|a_k/a_{k+1}| \approx e$) it is better to use

```plaintext
function sumalt_partial(p[], n, e) {
    d := ( 2*e + 1 + 2*sqrt(e*(e+1)) )^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1 {
        s := s + b * p[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1)) * e
    }
    return s/d
}
```

Convergence is improved from $\sim e^{-n}$ to $\sim \left(2e + 1 + 2\sqrt{e(e+1)}\right)^{-n} \approx (4e + 2)^{-n}$. This algorithm specializes to the original one for $e = 1$.

Implementations of the sumalt algorithm and the variant for partial sums are given in [hfloat: src/hf/sumalt.cc].
Chapter 33

Computing the elementary functions with limited resources

This chapter presents two types of algorithms for computations with limited resources, the shift-and-add and the CORDIC algorithms. The algorithms allow the computations of the elementary functions as the logarithm, exponential function, sine, cosine and their inverses with only shifts, adds, comparisons and table lookups. Algorithms of this type are usually used for pocket calculators.

33.1 Shift-and-add algorithms for $\log_b(x)$ and $b^x$

In this section so-called shift-and-add algorithms for the computation of $\log_b(x)$ and $b^x$ are presented. These algorithms use only additions, multiplications by a power of two (‘shifts’) and comparisons. Pre-computed lookup table with as many entries as the desired accuracy in bits is required. The algorithms are especially useful with limited hardware capabilities.

The implementations given in this section use floating point numbers. They can be rewritten to scaled integer arithmetic without difficulty.

33.1.1 Computing the base-$b$ logarithm

We use a table that contains the values $A_n = \log_b \left( 1 + \frac{1}{2^n} \right)$ where $n > 0$, it is created as follows:

```c
double *shiftadd_ltab;
ulong ltab_n;

void make_shiftadd_ltab(double b)
{
    double l1b = 1.0 / log(b);
    double s = 1.0;
    for (ulong k=0; k<ltab_n; ++k)
    {
        shiftadd_ltab[k] = log(1.0+s) * l1b; // == log_b(1+1/2^k)
        s *= 0.5;
    }
}
```

The algorithm takes as input the argument $x \geq 1$ and the number of iterations $N$ and computes $\log_b(x)$. It proceeds as follows:

1. Initialize: set $t_0 = 0$, $e_0 = 1$. 

[fxtbook draft of 2008-August-17]
Chapter 33: Computing the elementary functions with limited resources

<table>
<thead>
<tr>
<th>( n )</th>
<th>( u_n )</th>
<th>( t_n )</th>
<th>( e_n )</th>
<th>( A_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>-</td>
<td>0.00000000</td>
<td>+1.00000000</td>
<td>+1.00000000</td>
</tr>
<tr>
<td>1</td>
<td>1.50000000</td>
<td>0.00000000</td>
<td>+1.00000000</td>
<td>+0.58496250</td>
</tr>
<tr>
<td>2</td>
<td>1.25000000</td>
<td>0.00000000</td>
<td>+1.00000000</td>
<td>+0.32192809</td>
</tr>
<tr>
<td>3</td>
<td>1.56250000</td>
<td>0.32192809</td>
<td>+1.25000000</td>
<td>+0.32192809</td>
</tr>
<tr>
<td>4</td>
<td>1.49414062</td>
<td>0.32192809</td>
<td>+1.25000000</td>
<td>+0.08746284</td>
</tr>
<tr>
<td>5</td>
<td>1.45019531</td>
<td>0.49185309</td>
<td>+1.40625000</td>
<td>+0.04439411</td>
</tr>
<tr>
<td>6</td>
<td>1.42822265</td>
<td>0.49185309</td>
<td>+1.40625000</td>
<td>+0.02236781</td>
</tr>
<tr>
<td>7</td>
<td>1.41723632</td>
<td>0.49185309</td>
<td>+1.40625000</td>
<td>+0.01122725</td>
</tr>
<tr>
<td>8</td>
<td>1.41174316</td>
<td>0.49185309</td>
<td>+1.40625000</td>
<td>+0.00562454</td>
</tr>
<tr>
<td>9</td>
<td>1.41450047</td>
<td>0.49185309</td>
<td>+1.40625000</td>
<td>+0.00281501</td>
</tr>
<tr>
<td>10</td>
<td>1.41312181</td>
<td>0.49185309</td>
<td>+1.40625000</td>
<td>+0.00140819</td>
</tr>
<tr>
<td>11</td>
<td>1.41450182</td>
<td>0.49747764</td>
<td>+1.41174316</td>
<td>+0.00070426</td>
</tr>
<tr>
<td>12</td>
<td>1.41415698</td>
<td>0.49747764</td>
<td>+1.41174316</td>
<td>+0.00035217</td>
</tr>
<tr>
<td>13</td>
<td>1.41424330</td>
<td>0.49994228</td>
<td>+1.41381182</td>
<td>+0.00008805</td>
</tr>
<tr>
<td>14</td>
<td>1.41420014</td>
<td>0.49994228</td>
<td>+1.41381182</td>
<td>+0.00004402</td>
</tr>
<tr>
<td>15</td>
<td>1.41424330</td>
<td>0.49986331</td>
<td>+1.41312181</td>
<td>+0.00004402</td>
</tr>
<tr>
<td>∞</td>
<td>1.41421356</td>
<td>0.50000000</td>
<td>+1.41421356</td>
<td>+0.00000000</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{let } x & = \log_2(\sqrt{2}) = \frac{1}{2}.
\end{align*}
\]

**Figure 33.1-A:** Numerical values occurring in the shift-and-add computation of \( \log_2(\sqrt{2}) = 1/2 \). The computation of \( \log_{1/2}(\sqrt{2}) = -1/2 \) corresponds to the same values but opposite signs for all entries \( A_n \) and \( y_n \).

2. Compute \( u_n = e_n \cdot (1 + 2^{-n}) \). If \( u_n \leq x \) the set \( d_n = 1 \), else set \( d_n = 0 \).

3. If \( d_n \neq 0 \) then set \( t_{n+1} = t_n + A_n \) and \( e_{n+1} = u_n \) and repeat the last step. Else set \( t_{n+1} = t_n \) and \( e_{n+1} = e_n \).

4. Increment \( n \). If \( n = N \) return \( t_n \), else goto step 2.

A C++ implementation is given in [FXT: arith/shiftadd-log-demo.cc], note that the variable \( n \) equals \( N \), and \( k \) equals \( n \):
The variable \( v \) is a power of \( 1/2 \) so all multiplies by it can with scaled integer arithmetic be replaced by shifts as indicated by the comments. The values for first steps of the computation for the argument \( x_0 = \sqrt{2} \) are given in figure \ref{log2table}. The columns of the figure correspond to the variables \( k (= n) \), \( u (= u_n) \), \( t (= t_n) \), \( e (= e_n) \), and \( \text{shiftadd_ltab}[k] (= A_n) \).

\[
\begin{array}{|c|c|c|c|}
\hline
n & u_n & t_n & e_n & A_n \\
\hline
\text{init} & - & 0.00000000 & +1.00000000 & +1.00000000 \\
1: & 1.50000000 & 0.00000000 & +1.00000000 & +0.58496250 \\
1: & 2.25000000 & 0.58496250 & +1.50000000 & +0.58496250 \\
1: & 3.37500000 & 1.16992500 & +2.25000000 & +0.58496250 \\
1: & 5.06250000 & 1.75488750 & +3.37500000 & +0.58496250 \\
1: & 11.39062500 & 2.92481250 & +7.59375000 & +0.58496250 \\
2: & 9.49218750 & 2.92481250 & +7.59375000 & +0.32192809 \\
3: & 8.06835937 & 2.92481250 & +7.59375000 & +0.16992500 \\
4: & 8.07577514 & 2.96920662 & +7.83105468 & +0.04439411 \\
5: & 7.95341491 & 2.99157443 & +7.95341491 & +0.02236781 \\
6: & 8.00000000 & 2.99999999 & +8.00000000 & +0.00000000 \\
\infty & = x & = \log_2(8) & = x & = 0 \\
\hline
\end{array}
\]

\textbf{Figure 33.1-B:} Values occurring in the first few steps of a shift-and-add computation of \( \log_2(8) = 3 \).

The algorithm has been adapted from \cite{[210]} (chapter 5) where the correction is made only once for each value \( A_n \) limiting the range of convergence to \( x < X \) where

\[
X = \prod_{n=0}^{\infty} \left( 1 + \frac{1}{2^n} \right) = 4.768462058062743448299798577356794477543 \ldots
\]

As given, the algorithm converges for any \( x > 0 \), \( x \neq 1 \). A numerical example for the argument \( x = 8 \) is given in figure \ref{log2table}. The basis \( b \) must satisfy \( b > 0 \) and \( b \neq 1 \).

### 33.1.2 Computing \( b^x \)

We can use the same precomputed table as with the computation of \( \log_b(x) \).

The algorithm takes as input the argument \( x \) and the number of iterations \( N \) and computes \( b^x \) for \( b > 1 \), \( x \in \mathbb{R} \). It proceeds as follows:

1. Initialize: set \( t_0 = 0, e_0 = 1 \).
2. Compute \( u_n = t_n + A_n \). If \( u_n \leq x \) the set \( d_n = 1 \), else set \( d_n = 0 \).
3. If \( d_n \neq 0 \) then set \( t_n+1 = u_n \) and \( e_n+1 = e_n \cdot (1 + 2^{-n}) \) and repeat the last step. Else set \( t_n+1 = t_n \) and \( e_n+1 = e_n \).
4. Increment \( n \). If \( n = N \) return \( e_n \), else goto step 2.

A C++ implementation is given in \cite{FXT: arith/shiftadd-exp-demo.cc}:

```cpp
double
shiftadd_exp(double x, ulong n)
{
    if ( n>=ltab_n ) n = ltab_n;
    
```
## Chapter 33: Computing the elementary functions with limited resources

### 33.1.3 An alternative algorithm for the logarithm

A slightly different method for the computation of the base-$b$ logarithm ($b > 0$, $b \neq 1$) is given in [171]. Here the table used has to contain the values $A_n = \log_b \left( \frac{2^n}{2^n-1} \right)$ where $n > 0$:

```c
double *briggs_ltab;
ulong ltab_len;

void make_briggs_ltab(ulong na, double b)
```
33.1: Shift-and-add algorithms for $\log_b(x)$ and $b^x$

<table>
<thead>
<tr>
<th>( n )</th>
<th>( x_n )</th>
<th>( y_n )</th>
<th>( z_n )</th>
<th>( A_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>1.41421356</td>
<td>0.00000000</td>
<td>+0.07071068</td>
<td>+0.00000000</td>
</tr>
<tr>
<td>2:</td>
<td>1.41421356</td>
<td>0.00000000</td>
<td>+0.35355339</td>
<td>+0.41503749</td>
</tr>
<tr>
<td>2:</td>
<td>1.06066017</td>
<td>0.41503749</td>
<td>+0.26516504</td>
<td>+0.41503749</td>
</tr>
<tr>
<td>3:</td>
<td>1.06066017</td>
<td>0.41503749</td>
<td>+0.06629126</td>
<td>+0.09310940</td>
</tr>
<tr>
<td>4:</td>
<td>1.06066017</td>
<td>0.41503749</td>
<td>+0.03314563</td>
<td>+0.04580368</td>
</tr>
<tr>
<td>5:</td>
<td>1.02751454</td>
<td>0.46084118</td>
<td>+0.03210982</td>
<td>+0.04580368</td>
</tr>
<tr>
<td>6:</td>
<td>1.02751454</td>
<td>0.46084118</td>
<td>+0.01605491</td>
<td>+0.02272007</td>
</tr>
<tr>
<td>7:</td>
<td>1.01145962</td>
<td>0.48356126</td>
<td>+0.01580405</td>
<td>+0.02272007</td>
</tr>
<tr>
<td>8:</td>
<td>1.01145962</td>
<td>0.48356126</td>
<td>+0.00790202</td>
<td>+0.01131531</td>
</tr>
<tr>
<td>9:</td>
<td>1.00355759</td>
<td>0.49487657</td>
<td>+0.00784029</td>
<td>+0.01131531</td>
</tr>
<tr>
<td>10:</td>
<td>1.00355759</td>
<td>0.49487657</td>
<td>+0.00392014</td>
<td>+0.00282051</td>
</tr>
<tr>
<td>11:</td>
<td>1.00061940</td>
<td>0.49910666</td>
<td>+0.00384029</td>
<td>+0.00282051</td>
</tr>
<tr>
<td>12:</td>
<td>1.00061940</td>
<td>0.49910666</td>
<td>+0.00196007</td>
<td>+0.00140957</td>
</tr>
<tr>
<td>13:</td>
<td>1.00013081</td>
<td>0.49981128</td>
<td>+0.00195829</td>
<td>+0.00140957</td>
</tr>
<tr>
<td>14:</td>
<td>1.00013081</td>
<td>0.49981128</td>
<td>+0.00097812</td>
<td>+0.00070461</td>
</tr>
<tr>
<td>15:</td>
<td>1.00000873</td>
<td>0.49998740</td>
<td>+0.00097716</td>
<td>+0.00070461</td>
</tr>
<tr>
<td>( \infty ):</td>
<td>1.00000000</td>
<td>0.50000000</td>
<td>+0.00000000</td>
<td>+0.00000000</td>
</tr>
</tbody>
</table>

\( \log_b(\sqrt{2}) = 1/2 \) but \( \log_b(\sqrt{2}) = 0 \).

\begin{verbatim}
6  {
7      double llb = 1.0 / log(b);
8      double s = 2.0; // == 2^k
9      briggs_ltab[0] = -1.0; // unused
10     for (ulong k=1; k<na; ++k)
11     {
12        briggs_ltab[k] = log(s/(s-1.0)) * llb;
13        s *= 2.0;
14     }
15 }

The algorithm terminates when a given precision (\( \text{eps} \)) is reached:

1 double briggs_log(double x, double eps)
2 {
3     double y = 0;
4     double z = x * 0.5;
5     // [PRINT]
6     ulong k = 1;
7     double v = 0.5; // v == 2^(-k)
8     while ( fabs(x-1.0)>=eps )
9     {
10        while ( fabs(x-z)<1.0 )
11          {
12            z *= 0.5;
13            ++k; v *= 0.5;
14            if ( k >= ltab_len ) goto done; // no more table entries
15          } // [PRINT1]
16        x -= z;
17        y += briggs_ltab[k];
18        z = x * v; // z=(x>>k)
19     }
20     x -= z;
21     y += briggs_ltab[k];
22     z = x * v; // z=(x>>k)
23     return y;
24 }
\end{verbatim}

Figure 33.1-D: Numerical values occurring in the computation of $\log_2(\sqrt{2}) = 1/2$. The value of $n$ is incremented in the inner loop (comment [PRINT1] in the code, the value of $z$ changes). The values of $x$ and $y$ change just before the location of the comment [PRINT2], corresponding to consecutive rows with same value of $n$. The computation of $\log_{1/2}(\sqrt{2}) = -1/2$ corresponds to the same values but opposite signs for all entries $A_n$ and $y_n$. 

[fxtbook draft of 2008-August-17]
Chapter 33: Computing the elementary functions with limited resources

```cpp
23 // invariant: y_k + log_b(x_k) == log_b(x_0)
24 // [PRINT2]
25 }
26 // [PRINT2]
27 }
28 done: return y;
29 }
30 }
```

The code is given in [FXT: arith/briggs-log-demo.cc]. The values for first steps of the computation for the argument \(x_0 = \sqrt{2}\) are given in figure 33.1-D. The argument \(x\) must be greater than or equal to 1. Knuth [171] gives \(1 \leq x < 2\) but the restriction to values smaller than 2 does not seem to be necessary.

### 33.2 CORDIC algorithms

The so-called CORDIC algorithms can be used for the computation of functions like sine, cosine, exp and log. The acronym CORDIC stands for Coordinate Rotation Digital Computer.

Similar to the shift-and-add algorithms (section 33.1) only multiplications by powers of two (shifts), additions, subtractions and comparisons are used. Again, a precomputed lookup table with as many entries as the desired accuracy in bits is required.

Some early floating point units (FPUs) used CORDIC algorithms and your pocket calculator surely does.

#### 33.2.1 The circular case: sine and cosine

We start with a CORDIC routine for the computation of the sine and cosine. The lookup table has to contain the values \(\arctan(2^{-n})\) for \(n = 0, 1, 2, 3, \ldots\), these shall be stored in the array `cordic_ctab[]`. An implementation of the function is given in [FXT: arith/cordic-circ-demo.cc]:

```cpp
1 void
cordic_circ(double theta, double &s, double &c, ulong n)
2 {
3   double x = cordic_1K;
4   double y = 0;
5   double z = theta;
6   double v = 1.0;
7   ```

<table>
<thead>
<tr>
<th>(n)</th>
<th>(x_n)</th>
<th>(y_n)</th>
<th>(z_n)</th>
<th>(-d \cdot A_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>0.60725293</td>
<td>0.00000000</td>
<td>+1.04719755</td>
<td>+0.00000000</td>
</tr>
<tr>
<td>0</td>
<td>0.60725293</td>
<td>0.60725293</td>
<td>+0.26179938</td>
<td>-0.78539816</td>
</tr>
<tr>
<td>1</td>
<td>0.30362646</td>
<td>0.91087940</td>
<td>-0.20184822</td>
<td>-0.46364760</td>
</tr>
<tr>
<td>2</td>
<td>0.53134631</td>
<td>0.83497278</td>
<td>+0.04313044</td>
<td>+0.24497866</td>
</tr>
<tr>
<td>3</td>
<td>0.42697471</td>
<td>0.90139107</td>
<td>-0.08122455</td>
<td>-0.46364760</td>
</tr>
<tr>
<td>4</td>
<td>0.48331166</td>
<td>0.87470515</td>
<td>-0.18805742</td>
<td>+0.62418800</td>
</tr>
<tr>
<td>5</td>
<td>0.51064619</td>
<td>0.85960166</td>
<td>+0.01236640</td>
<td>-0.00462270</td>
</tr>
<tr>
<td>6</td>
<td>0.49721492</td>
<td>0.86758051</td>
<td>-0.00026008</td>
<td>+0.00097656</td>
</tr>
<tr>
<td>7</td>
<td>0.50399289</td>
<td>0.86369602</td>
<td>+0.00462270</td>
<td>+0.00097656</td>
</tr>
<tr>
<td>8</td>
<td>0.49972149</td>
<td>0.86580510</td>
<td>-0.00002608</td>
<td>+0.00097656</td>
</tr>
<tr>
<td>9</td>
<td>0.49892833</td>
<td>0.86642510</td>
<td>+0.00002608</td>
<td>+0.00097656</td>
</tr>
<tr>
<td>10</td>
<td>0.49977466</td>
<td>0.86615528</td>
<td>+0.00000000</td>
<td>+0.00000000</td>
</tr>
<tr>
<td>11</td>
<td>0.50019758</td>
<td>0.86591124</td>
<td>+0.00002608</td>
<td>+0.00000000</td>
</tr>
<tr>
<td>12</td>
<td>0.49998618</td>
<td>0.86603336</td>
<td>+0.00000000</td>
<td>+0.00000000</td>
</tr>
<tr>
<td>13</td>
<td>0.50009190</td>
<td>0.86597233</td>
<td>+0.00001061</td>
<td>+0.00001061</td>
</tr>
<tr>
<td>14</td>
<td>0.50003904</td>
<td>0.86600285</td>
<td>+0.00000450</td>
<td>+0.00000450</td>
</tr>
<tr>
<td>15</td>
<td>0.50001261</td>
<td>0.86601811</td>
<td>+0.00000145</td>
<td>+0.00000145</td>
</tr>
<tr>
<td>(\infty)</td>
<td>0.50000000</td>
<td>0.86602540</td>
<td>+0.00000000</td>
<td>+0.00000000</td>
</tr>
</tbody>
</table>

\(= \cos(\pi/3)\) \hspace{1em} \(= \sin(\pi/3)\) \hspace{1em} \(= 0\) \hspace{1em} \(= 0\)

Figure 33.2-A: Numerical values occurring in the CORDIC computation of \(\cos(\pi/3)\) and \(\sin(\pi/3)\).
For the sake of clarity floating point types are used. All operations can easily be converted to integer arithmetic. The multiplications by $d$ are sign changes and should be replaced by an if-construct. The multiplications by $v$ are shifts.

The values for first 16 steps of the computation for the argument $z_0 = \theta = \pi/3 = 1.04719755\ldots$ are given in figure 33.2-A. While $z$ gets closer to zero (however, the magnitude of $z$ does not necessarily decrease with every step) the values of $x$ and $y$ approach $\sin(\pi/3) = 1/2$ and $\cos(\pi/3) = \sqrt{3}/2 = 0.86602540\ldots$, respectively.

More formally, one initializes

$$x_0 = 1/K = 0.607252935008881\ldots \quad (33.2-1a)$$
$$y_0 = 0 \quad (33.2-1b)$$
$$z_0 = \theta \quad (33.2-1c)$$

and iterates (starting with $n = 0$)

$$A_n = \arctan(2^{-n}) \quad \text{(precomputed)} \quad (33.2-1d)$$
$$v_n = 2^{-n} \quad (33.2-1e)$$
$$d_n = \text{sign}(z_n) \quad (33.2-1f)$$
$$x_{n+1} = x_n - d_n v_n y_n \rightarrow \cos(\theta) \quad (33.2-1g)$$
$$y_{n+1} = y_n + d_n v_n x_n \rightarrow \sin(\theta) \quad (33.2-1h)$$
$$z_{n+1} = z_n - d_n A_n \rightarrow 0 \quad (33.2-1i)$$

The scaling constant $K$ is

$$K = \prod_{k=0}^{\infty} \sqrt{1+2^{-2k}} \quad (33.2-2a)$$

$$K = 1.646760258121065648366051222282298435652376725701027409\ldots \quad (33.2-2b)$$

$$1/K = 0.6072529350088812561694467525049282631123908521500897724\ldots \quad (33.2-2c)$$

We note that $K$ can be computed more efficiently as $K = \sqrt{2 F(1/4)}$ where $F(z)$ is defined as

$$F(z) = \prod_{k=1}^{\infty} 1 + z^k \quad (33.2-3)$$

We use relation 14.4-12 on page 338 and relation 14.4-8 (pentagonal number theorem): $F(z) = P(z^2)/P(z)$ where

$$P(z) = 1 + \sum_{n=1}^{\infty} (-1)^n \left( z^{n(3n-1)/2} + z^{n(3n+1)/2} \right) \quad (33.2-4)$$

Using $n$ terms of the sum gives a precision of about $3 (n-1)^2$ bits:
Chapter 33: Computing the elementary functions with limited resources

\[ \text{pent}(z, n) = \sum_{k=1}^{n} \frac{(-1)^{k}(z^{k(3k-1)/2} + z^{k(3k+1)/2})}{k(3k-1)/2} \]

\[ n=30; \quad u=0.25; \quad K=\sqrt{2 \times \text{pent}(u^2,n)/\text{pent}(u,n)} \]

\[ K \approx 1.64676025812106564836605122228229843562376725701027409 \]

The CORDIC algorithm converges if \(-r \leq z_0 \leq r\) where

\[ r = \sum_{k=0}^{\infty} \arctan(2^{-k}) = \frac{\pi}{2} = 1.57079632 \ldots (33.2-5c) \]

With arguments \(x_0, y_0, z_0\) one has

\[ x \rightarrow K(\sqrt{x_0^2 + y_0^2}) \]
\[ y \rightarrow 0 \]
\[ z \rightarrow z_0 - \arctan\left(\frac{y_0}{x_0}\right) \]

The algorithm can be derived by writing

\[ \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} +\cos(d_n A_n) & -\sin(d_n A_n) \\ +\sin(d_n A_n) & +\cos(d_n A_n) \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \]

and noting that (using \(d_n = \pm 1\), so \(\cos(d_n A_n) = \cos(A_n)\) and \(\sin(d_n A_n) = d_n \sin(A_n)\))

\[ \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \cos(A_n) \begin{bmatrix} +1 & -d_n v_n \\ +d_n v_n & +1 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \]

where \(v_n = 2^{-n}\). The CORDIC algorithm postpones the multiplications by \(\cos(A_n)\). One has

\[ \cos(A_n) = \cos\left(\arctan(2^{-n})\right) = \frac{1}{\sqrt{1 + 2^{-2n}}} \]

Thereby

\[ K = \frac{1}{\prod_{k=0}^{\infty} \cos(A_n)} = \prod_{k=0}^{\infty} \sqrt{1 + 2^{-2n}} \]

### 33.2.2 The linear case: multiplication and division

A slight variation gives a base-2 multiply-add algorithm:

\[ A_n = 2^{-n} \]
\[ v_n = 2^{-n} \]
\[ d_n = \text{sign}(z_n) \]
\[ x_{n+1} = x_n \]
\[ y_{n+1} = y_n + d_n v_n x_n \]
\[ z_{n+1} = z_n - d_n A_n \]
then

\[
\begin{align*}
x & \rightarrow x_0 \\
y & \rightarrow y_0 + x_0 z_0 \\
z & \rightarrow 0
\end{align*}
\]

Going backwards (replace relation 33.2-12c on the facing page by \(d_n := -\text{sign}(y_n)\)) gives an algorithm for division:

\[
\begin{align*}
x & \rightarrow x_0 \\
y & \rightarrow 0 \\
z & \rightarrow z_0 - \frac{y_0}{x_0}
\end{align*}
\]

### 33.2.3 The hyperbolic case: \(\sinh\) and \(\cosh\)

<table>
<thead>
<tr>
<th>(n)</th>
<th>(x_n)</th>
<th>(y_n)</th>
<th>(z_n)</th>
<th>(A_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{init})</td>
<td>1.20749706</td>
<td>0.00000000</td>
<td>+1.00000000</td>
<td>+0.00000000</td>
</tr>
<tr>
<td>1</td>
<td>1.20749706</td>
<td>0.60374853</td>
<td>+0.45069385</td>
<td>-0.54930614</td>
</tr>
<tr>
<td>2</td>
<td>1.35843420</td>
<td>0.90562280</td>
<td>+0.19528104</td>
<td>-0.25541281</td>
</tr>
<tr>
<td>3</td>
<td>1.47163705</td>
<td>1.07542707</td>
<td>+0.06962382</td>
<td>-0.12565721</td>
</tr>
<tr>
<td>4</td>
<td>1.53885124</td>
<td>1.16740439</td>
<td>+0.00704225</td>
<td>-0.06258157</td>
</tr>
<tr>
<td>+4</td>
<td>1.61181401</td>
<td>1.26358259</td>
<td>-0.0553931</td>
<td>-0.06258157</td>
</tr>
<tr>
<td>5</td>
<td>1.57232706</td>
<td>1.21321340</td>
<td>-0.0242791</td>
<td>+0.03126017</td>
</tr>
<tr>
<td>6</td>
<td>1.55337060</td>
<td>1.1884579</td>
<td>-0.00865286</td>
<td>+0.01562627</td>
</tr>
<tr>
<td>7</td>
<td>1.54408430</td>
<td>1.17651008</td>
<td>-0.0008402</td>
<td>+0.00781256</td>
</tr>
<tr>
<td>8</td>
<td>1.53948856</td>
<td>1.17047850</td>
<td>+0.00306606</td>
<td>+0.00390626</td>
</tr>
<tr>
<td>9</td>
<td>1.54177465</td>
<td>1.17348532</td>
<td>+0.0011293</td>
<td>-0.00195312</td>
</tr>
<tr>
<td>10</td>
<td>1.54292063</td>
<td>1.17499096</td>
<td>+0.00013637</td>
<td>-0.00097656</td>
</tr>
<tr>
<td>11</td>
<td>1.54349436</td>
<td>1.17574434</td>
<td>-0.00035190</td>
<td>-0.00048828</td>
</tr>
<tr>
<td>12</td>
<td>1.54320731</td>
<td>1.17536751</td>
<td>-0.00010776</td>
<td>+0.00024414</td>
</tr>
<tr>
<td>13</td>
<td>1.54306383</td>
<td>1.17517913</td>
<td>-0.00001430</td>
<td>+0.00012207</td>
</tr>
<tr>
<td>+13</td>
<td>1.54320729</td>
<td>1.17536749</td>
<td>-0.00010776</td>
<td>-0.00012207</td>
</tr>
<tr>
<td>14</td>
<td>1.54313555</td>
<td>1.17527330</td>
<td>-0.00004673</td>
<td>+0.00060103</td>
</tr>
<tr>
<td>15</td>
<td>1.54309968</td>
<td>1.17522621</td>
<td>-0.00001621</td>
<td>+0.0003051</td>
</tr>
<tr>
<td>(\infty)</td>
<td>1.54308063</td>
<td>1.17520119</td>
<td>+0.00000000</td>
<td>+0.00000000</td>
</tr>
</tbody>
</table>

Figure 33.2-B: Numerical values occurring in the CORDIC computation of \(\cosh(1)\) and \(\sinh(1)\). Note that steps 4 and 13 are executed twice.

The versions presented so far can be unified as

\[
\begin{align*}
v_n &= 2^{-n} \\
x_{n+1} &= x_n - m \, d_n \, v_n \, y_n \\
y_{n+1} &= y_n + d_n \, v_n \, x_n \\
z_{n+1} &= z_n - d_n \, A_n
\end{align*}
\]

where the linear case corresponds to \(m = 0\) and \(A_n = 2^{-n}\), the circular case to \(m = 1\) and \(A_n = \arctan(2^{-n})\). The forward direction (‘rotation mode’) is obtained by setting \(d_n = \text{sign}(z_n)\), the backward direction (‘vectoring mode’) by setting \(d_n = -\text{sign}(y_n)\).

Setting \(m = -1\) gives a CORDIC algorithm that computes the hyperbolic sine and cosine or their inverses. The lookup table has to contain the values \(\text{arctanh}(2^{-n})\) for \(n = 1, 2, 3, \ldots\), stored in the array \(\text{cordic_htab}[]\). The algorithm needs a modification in order to converge: the iteration starts with index
one and some steps have to be executed twice. The sequence of the indices that need to be processed twice is 4, 13, 40, 121, ... \( (i_0 = 4, i_{n+1} = 3i_n + 1, \text{entry A003462 of [245])} \).

A sample implementation is given in [FXT: arith/cordic-hyp-demo.cc]:

```c
void cordic_hyp(double theta, double &s, double &c, ulong n)
{
  double x = cordic_1Kp;
  double y = 0;
  double z = theta;
  double v = 1.0;
  // [PRINT]
  ulong i = 4;
  for (ulong k=1; k<n; ++k)
  {
    v *= 0.5;
    again:
    double d = ( z>=0 ? +1 : -1 );
    double tx = x + d * v * y;
    double ty = y + d * v * x;
    double tz = z - d * cordic_htab[k];
    x = tx; y = ty; z = tz;
    // [PRINT]
    if ( k==i ) { i=3*i+1; goto again; }
  }
  c = x;
  s = y;
}
```

The values for first steps of the computation for the argument \( \theta = z_1 = 1.0 \) are given in figure 33.2-B.

The scaling constant corresponding to \( K \) is \( K' \), one has

\[
K' = \prod_{k=1}^{\infty} \sqrt{1 - 2^{-2k}} \cdot \prod_{k=0}^{\infty} \sqrt{1 - 2^{-2i_k}} \quad (33.2-16a)
\]

\[
K' = 0.8281593609602156270761983277591751468694538376908425291\ldots \quad (33.2-16b)
\]

\[
\frac{1}{K'} = 1.207497067763072128877721011310915836812783221769813422\ldots \quad (33.2-16c)
\]

The duplicated indices appear twice in the product. The algorithm can be used for the computation of the exponential function using \( \exp(x) = \sinh(x) + \cosh(x) \). The algorithm converges if \(-r' \leq z_1 \leq r'\) where

\[
r' = \sum_{k=1}^{\infty} \arctanh(2^{-k}) + \sum_{k=0}^{\infty} \arctanh(2^{-i_k}) \quad (33.2-17a)
\]

\[
r' = 1.118173015526503803610627556783092451806572942929536106\ldots \quad (33.2-17b)
\]

With arguments \( x_1, y_1, z_1 \) one has

\[
x \rightarrow K'(x_1 \cosh(z_1) + y_1 \sinh(z_1)) \quad (33.2-18a)
\]

\[
y \rightarrow K'(y_1 \cosh(z_1) + x_1 \sinh(z_1)) \quad (33.2-18b)
\]

\[
z \rightarrow 0 \quad (33.2-18c)
\]

which, for \( x_1 = 1/K', y_1 = 0, z_1 = \theta \) specializes to the computation as above.

The backward version \( (d_n := -\text{sign}(y_n)) \) computes

\[
x \rightarrow K' \sqrt{x_1^2 - y_1^2} \quad (33.2-19a)
\]

\[
y \rightarrow 0 \quad (33.2-19b)
\]

\[
z \rightarrow z_1 - \arctanh \left( \frac{y_1}{x_1} \right) \quad (33.2-19c)
\]
For the computation of the natural logarithm use \( \log(w) = 2 \arctanh \frac{w-1}{w+1} \). That is, start with \( x_1 = w + 1 \) and \( y_1 = w - 1 \), then \( z \to \frac{1}{2} \log(w) \).

The computation of the square root \( \sqrt{w} \) can be obtained by starting with \( x_1 = w + 1/4 \) and \( y_1 = w - 1/4 \) then \( z \to K' \sqrt{w} \).

For further information see [14], [145], and chapter 6 of [210]. An algorithm working with complex numbers is given in [30].
Chapter 34

Recurrences and Chebyshev polynomials

This chapter presents algorithms and material concerning recurrences. Firstly, several algorithms for recurrences, mostly for the case of constant coefficients, are given. Secondly, the Chebyshev polynomials are described. These are an important special case of a recurrence.

34.1 Recurrences

A sequence \([a_0, a_1, a_2, \ldots]\) so that a recurrence relation

\[ a_n = \sum_{j=1}^{k} m_j a_{n-j} \]  

(34.1-1)

with given \(m_j\) holds for all \(a_j\) is called a \(k\)-th order recurrence. The recurrence is linear, homogeneous, with constant coefficients. The sequence is defined by both the recurrence relation and the first \(k\) elements.

An example, the second order recurrence relation \(a_n = a_{n-1} + a_{n-2}\) together with \(a_0 = 0\) and \(a_1 = 1\) gives the Fibonacci numbers \(F_n\), starting with \(a_0 = 2\) and \(a_1 = 1\) gives the Lucas numbers \(L_n\):

<table>
<thead>
<tr>
<th>(n) : (0)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
<th>(10)</th>
<th>(11)</th>
<th>(12)</th>
<th>(13)</th>
<th>(14)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F(n)) :</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>233</td>
<td>377</td>
</tr>
<tr>
<td>(L(n)) :</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>11</td>
<td>18</td>
<td>29</td>
<td>47</td>
<td>76</td>
<td>123</td>
<td>199</td>
<td>322</td>
<td>521</td>
</tr>
</tbody>
</table>

The characteristic polynomial of the recurrence relation [34.1-1] is given by

\[ p(x) = x^k - \sum_{j=1}^{k} m_j x^{k-j} \]  

(34.1-2)

The definition can be motivated by writing down the recurrence relation for the element with index \(n = k\):

\[ 0 = a_k - \left( \sum_{j=1}^{k} m_j a_{k-j} \right) \]  

(34.1-3)

34.1.1 Fast computation using matrix powers

For the recurrence defined by the recurrence relation

\[ a_n := m_1 a_{n-1} + m_2 a_{n-2} \]  

(34.1-4)
Chapter 34: Recurrences and Chebyshev polynomials

and the start \( a_0, a_1 \) use the relation

\[
\begin{bmatrix}
  a_0 & a_1 \\
  0 & m_2 \\
  1 & m_1
\end{bmatrix}
\begin{bmatrix}
  0 \\
  1
\end{bmatrix}
= 
\begin{bmatrix}
  a_k \\
  a_{k+1}
\end{bmatrix}
\]

(34.1-5)

for the fast computation of an individual element \( a_k \). The algorithm is fast when powering algorithms (see section 27.6) are used.

Note that with two consecutive terms of the recurrence in the resulting vector it is easy two compute the following terms \( a_{k+1}, a_{k+2}, \ldots \) using the original recurrence relation.

The generalization is straightforward. For example, a recurrence \( a_n = m_1 a_{n-1} + m_2 a_{n-2} + m_3 a_{n-3} \) corresponds to

\[
\begin{bmatrix}
  a_0 & a_1 & a_2 \\
  0 & 0 & m_3 \\
  1 & 0 & m_2 \\
  0 & 1 & m_1
\end{bmatrix}
\begin{bmatrix}
  0 \\
  1
\end{bmatrix}
= 
\begin{bmatrix}
  a_k \\
  a_{k+1} \\
  a_{k+2}
\end{bmatrix}
\]

(34.1-6)

The matrix is the companion matrix of the characteristic polynomial \( x^3 - (m_1 x^2 + m_2 x^1 + m_3 x^0) \), see relation 40.5-1 on page 898. Note that the indexing of the \( m_k \) is different here.

**Performance**

The computations are fast. As an example we give the timing of the computation of a few sequence terms with large indices. The following calculations were carried out with exact arithmetic, the post-multiply with the float 1.0 renders the output readable:

```plaintext
? M=[0,1;1,1] \ \ Fibonacci sequence
? #
# timer = 1 (on)
? (0,1)*M^10000 [1]*1.0
time = 1 ms.3.364476487643 E2089
? (0,1)*M^100000 [1]*1.0
time = 10 ms.2.597406934722 E20898
? (0,1)*M^1000000 [1]*1.0
time = 458 ms.1.953282128707 E208987
```

The powering algorithm can obviously be used also for polynomial recurrences such as for the Chebyshev polynomials \( T_n(x) \):

```plaintext
? M=[0,-1;1,2*x]
[0 -1]
[1 2*x]
? for(n=0,5,print(n," : ",([1,x]*M^n)[1]))
0: 1
1: x
2: 2*x^2 - 1
3: 4*x^3 - 8*x + 1
4: 8*x^4 - 32*x^2 + 1
5: 16*x^5 - 20*x^3 + 5*x
? p=([1,x]*M^1000)[1];
time = 1,027 ms.
? poldegree(p)
1000
? log(polcoeff(p,poldegree(p)))/log(10)
300.728965668317 \ \ The coefficient of x^1000 is a 301-digit number
```

With modular arithmetic the quantities remain bounded and the computations can be carried out for extreme large values of \( n \). We use the modulus \( m = 2^{1279} - 1 \) and compute the \( n = (m + 1)/4 \) element of the sequence 2, 4, 14, 52, ... where \( a_n = 4a_{n-1} - a_{n-2} \):

```plaintext
? m=2^1279-1; \ \ a 1279-bit number
? log(m)/log(10)
385.0173 \ \ 306 decimal digits
? M=Mod([0,-1;1,4],m); \ \ all entries modulo m
? component( ([2,4]*M^((m+1)/4))[1], 2)
0 time = 118 ms.
```

[fxtbook draft of 2008-August-17]
The result is zero which proves that \( m \) is prime, see section 37.11.4. Here is a one-liner that prints all exponents \( e < 1000 \) of Mersenne primes:

```
? forprime(e=3,1000,m=2^e-1;M=Mod([[0,-1;1,4],m);if(0==(([2,4]*M^((m+1)/4))[1]),print1(" ", e)))
```

3 5 7 13 17 19 31 61 89 107 127 521 607

The computation takes a few seconds only.

The connection of recurrences and matrix powers is investigated in [34].

### 34.1.2 Faster computation using polynomial arithmetic

The matrix power algorithm for computing the \( k \)-th element of a \( n \)-th order recursion involves proportional \( \log k \) multiplications of \( n \times n \) matrices. As matrix multiplication (with the straightforward algorithm) is proportional \( n^3 \) the algorithm is not optimal for recursions of high order. Note that the matrix entries grow exponentially, so the asymptotics as given is valid only for computations with bounded values such as with modular arithmetic. We will see that the involved work can be brought down from \( \log k \cdot n^3 \) to \( \log k \cdot n^2 \) and even to \( \log k \cdot n \cdot \log n \).

The characteristic polynomial for the recursion \( a_n := 3a_{n-1} + 1a_{n-2} + 2a_{n-3} \) is

\[
p(x) = x^3 - 3x^2 - 1x - 2 \quad (34.1-7)
\]

We list the first few powers of the companion matrix \( M \) of \( p(x) \):

\[
M^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad M^1 = \begin{bmatrix} 0 & 2 & 6 \\ 1 & 0 & 5 \\ 0 & 1 & 3 \end{bmatrix} \quad M^2 = \begin{bmatrix} 2 & 6 & 20 \\ 1 & 5 & 16 \\ 3 & 10 & 35 \end{bmatrix} \quad (34.1-8)
\]

Note that each power is a left shifted version of its predecessor, only the rightmost column is ‘new’. Now compare the columns of the matrix powers to the first few values \( x^k \) modulo \( p(x) \):

\[
\begin{align*}
x^0 \mod p(x) &= 0x^2 + 0x + 1 \\
x^1 \mod p(x) &= 0x^2 + 1x + 0 \\
x^2 \mod p(x) &= 1x^2 + 0x + 0 \\
x^3 \mod p(x) &= 3x^2 + 1x + 2 \\
x^4 \mod p(x) &= 10x^2 + 5x + 6 \\
x^5 \mod p(x) &= 35x^2 + 16x + 20
\end{align*} \quad (34.1-9a-d)
\]

Observe that \( x^k \mod p(x) \) corresponds to the leftmost column of \( M^k \).

We now turn the observation into an efficient algorithm. The main routines in this section take as arguments a vector \( v \) of initial values, a vector \( m \) of recursion coefficients and an index \( k \). The vector \( r = [a_k, a_{k+1}, \ldots , a_{k+n}] \) is returned. We compute the leftmost column of \( M^k \) as \( z := x^k \mod p(x) \) and compute \( a_k \) as the scalar product of \( z \) (as a vector) and \( v \). Our main routine is:

```plaintext
1  frec(v, m, k)=
2  {
3    local(n, pc, pv, pp, px, r, t);
4    n = length(m);
5    if ( k<=n, return( recstep(v, m, k) ) ); \ \ small indices by definition
6    pc = vec2charpol(m);
7    pp = Mod( x, pc );
8    px = pp^-x();
9    r = vector(n);
10    for (i=1, n,
11        t = component(px,2);
12        r[i] = sum(j=1,n, v[j]*polcoeff(t,j-1,x)));
13    px *= pp;
14    return( r );
15  }
```

[fftbook draft of 2008-August-17]
If only the value $a_k$ is of interest, skip the computations in the final for loop for the values $i > 1$.

For small indices $k$ the result is computed directly by definition, using the following auxiliary routine:

```fortran
recstep(v, m, k)=
{ /* update v by k steps according to the recursion coefficients in m */
  local(n,r);
  if ( k<=0, return(v) ); \ negative k is forbidden
  n = length(m);
  r = vector(n);
  for (i=1, k,
    for (j=1, n-1, r[j]=v[j+1] ); \ shift left
    r[n] = sum(j=1,n, m[n+1-j]*v[j]); \ new element (convolution)
    v = r;
  );
  return( r );
}
```

The auxiliary routine used to compute the characteristic polynomial corresponding to the vector $m$ is:

```fortran
vec2charpol(m)=
{ /* return characteristic polynomial for the recursion coefficients in m */
  local(d,p);
  d = length(m);
  p = x^d - Pol(m,x);
  return( p );
}
```

The computation of the $k$-th element of a $n$-term recurrence involves proportional $\log k$ modular polynomial multiplications. Thereby the total cost is $\log k \cdot M(n)$ where $M(n)$ is the cost of the multiplication of two polynomials of degree $n$. That is, the cost is $\log k \cdot n^2$ when usual polynomial multiplication is used, and $\log k \cdot n \cdot \log n$ if an FFT scheme is applied.

The matrix power algorithm, restated for the argument structure defined above, can be implemented as:

```fortran
mrec(v, m, k)=
{ local(p,M);
  p = vec2charpol(m);
  M = matcompanion(p);
  M = M^k;
  return ( v * M );
}
```

All main routines can be used with symbolic values:

```fortran
? frec([a0,a1],[m1,m2],3)
[2*m2*m1*a0 + (m1^2 + m2)*a1, (2*m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
? mrec([a0,a1],[m1,m2],3)
[2*m2*m1*a0 + (m1^2 + m2)*a1, (2*m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
? recstep([a0,a1],[m1,m2],3)
[2*m2*m1*a0 + (m1^2 + m2)*a1, (2*m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
```

Performance

We check the performance (suppressing output):

```fortran
? k=10^5;
? recstep([0,1],[1,1],k);
  time = 2,811 ms. \ time linear in k
? mrec([0,1],[1,1],k);
  time = 10 ms. \ time linear in \log(k)
? frec([0,1],[1,1],k);
  time = 4 ms. \ time linear in \log(k)
```

The relative performance of the routine `frec()` and `mrec()` differs more with higher orders $n$ of the recurrence, we use $n = 10$:

```fortran
? n=10; v=vector(n); v[n]=1; m=vector(n,j,1); k=10^5; \ tenth order recurrence
? mrec(v,m,k);
  time = 2,813 ms.
? f=frec(v,m,k);
  time = 159 ms.
```
? log(f)/log(10.0)
[30078.67, 30078.97, 30079.58, 30079.88, 30080.18, \ 
30080.48, 30080.78, 30081.08, 30081.38] \ about 30k decimal digits each

Somewhat surprisingly, we see a performance gain greater than \( n \) even though the computations were done using integers. Finally, we repeat the computations modulo \( p = 2^{521} - 1 \) for \( k = 10^{30} \):

\[
\begin{align*}
? \ n=10; \ v=vector(n); \ v[n]=1; \ m=vector(n,j,1); \\
? \ k=10^{30}; \ p=2^{521}-1; \ v=Mod(v,p); \ m=Mod(m,p); \\
? \ mrec(v,m,k); \\
\text{time} = 312 \text{ ms.} \\
? \ frec(v,m,k); \\
\text{time} = 14 \text{ ms.}
\end{align*}
\]

That the performance gain with integers is not smaller than with modular arithmetic can be motivated by the fact that the quantities in both algorithms grow with the same rate. Now at each step the performance ratio should approximately equal \( n \). Thereby the algorithms perform with the same ratio.

The computational advantage of powering modulo the characteristic polynomial versus matrix powering has been pointed out 1994 by Brent [67, p.392] (page 4 of the preprint).

34.1.3 Inhomogeneous recurrences

The fast algorithms for the computation of recurrences do only work with homogeneous recurrences as defined by relation \ref{34.1-1} on page 667. A inhomogeneous recurrence is defined by a relation

\[
a_n = \sum_{j=1}^{k} m_j a_{n-j} + P(n) \tag{34.1-10}
\]

where \( P(n) \) is a nonzero polynomial in \( n \). We will show how to transform an inhomogeneous recurrence into a homogeneous recurrence of greater order.

34.1.3.1 Recurrence relations with a constant

In case a constant is to be added in an \( k \)-th order relation, one can use a recurrence of order \( k+1 \). From the recurrence relation \( a_n = m_1 a_{n-1} + m_2 a_{n-2} + \ldots + m_k a_{n-k} + C \) subtract a shifted version \( a_{n-1} = m_1 a_{n-2} + m_2 a_{n-3} + \ldots + m_k a_{n-k-1} + C \) to obtain \( a_n = (m_1 + 1) a_{n-1} + (m_2 - m_1) a_{n-2} + \ldots + (m_k - m_{k-1}) a_{n-k} \).

An example should make the idea clear: with \( a_n = 34 a_{n-1} - a_{n-2} + 2 \) subtract a shifted version \( a_{n-1} = 34 a_{n-2} - a_{n-3} + 2 \) to obtain \( a_n = 35 a_{n-1} - 35 a_{n-2} + a_{n-3} \). Setting \( a_0 = 1, a_1 = 36 \) we get, using the original relation

\[
\begin{align*}
? \ n=7; \\
? \ ts=vector(n); \ ts[1]=1; \ ts[2]=36; \\
? \ for(k=3,n,ts[k]=34*ts[k-1]-ts[k-2]+2); \\
? \ ts \\
[1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
\end{align*}
\]

and, using the relation without constant,

\[
\begin{align*}
? \ for(k=4,n,ts[k]=35*ts[k-1]-35*ts[k-2]+ts[k-3]); \\
? \ ts \\
[1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
\end{align*}
\]

34.1.3.2 The general case

If the recurrence is of the form

\[
a_n = m_1 a_{n-1} + m_2 a_{n-2} + \ldots + m_k a_{n-k} + P(n) \tag{34.1-11}
\]
where $P(n)$ is a polynomial of degree $d$ in $n$ then a homogeneous recurrence of order $k + d + 1$

$$a_n = M_1 a_{n-1} + M_2 a_{n-2} + \ldots + M_{k+d+1} a_{n-k-d-1} \quad (34.1-12)$$

can be obtained by repeatedly subtracting a shifted relation.

The following pari/gp routine takes as input a vector of the multipliers $m_i$ ($i = 1, \ldots, k$) and a polynomial of degree $d$ in $n$. It returns a homogeneous recurrence relation as a vector $[M_1, \ldots, M_{k+d+1}]$:

```plaintext
1  ihom2hom(m, p)=
2  { local(d, M, k);
3    if ( p==0, return(m) );
4    d = poldegree(p, 'n);
5    k = length(m);
6    M = vector(k+d+1);
7    for (j=1, k, M[j]=m[j]);
8    for (s=1, d+1,
9      M[1] += 1; \ \left hand side
10      for (j=2, k+s, M[j] -= m[j-1]; );
11        m = M;
12    );
13    return(M);
14 }
```

In order to verify the output we use a (slow) routine that directly computes the values of an inhomogeneous recurrence:

```plaintext
1  ihom(v, m, k, p)=
2  { local(n, r);
3    if ( k<=0, return(v[1]) );
4    n = length(m);
5    r = vector(n);
6    for (i=1, k,
7      for (j=1, n-1, r[j]=v[j+1]); \ \shift left
8        r[n] = sum(j=1,n, m[n+1-j]*v[j]); \ \new element (convolution)
9        r[n] += subst(p, 'n, i+n-1); \ \add inhomogeneous term
10        v = r;
11    );
12    return( r[1] );
13 }
```

We use the recurrence relation $a_n = 3a_{n-1} + 2a_{n-2} + (n^3 - n^2 - 7)$. We compute the homogeneous equivalent (intermediate values of $M$ added):

? m=[3,+2];p=n^3-n^2-7;
? M=ihom2hom(m,p)

```
[3, 2, 0, 0, 0, 0]
[4, -1, -2, 0, 0, 0]
[5, -5, -1, 2, 0, 0]
[6, -10, 4, 3, -2, 0]
[7, -16, 14, -1, -5, 2]
\ \a_n = 7*a_{n-1} - 16*a_{n-2} + 14*a_{n-3} + ...```

We can compute the first few values for the sequence starting with $a_0 = 2$, $a_1 = 5$ by the direct method:

? v=[2,5];
? for(k=0,9,print(k,: "ihom(v,m,k,p))

```
0: 2
1: 5
2: 16
3: 280
4: 1071
5: 3946
6: 14267
7: 511349
8: 1825777
9: 6728277
```

A vector of start values and the homogeneous equivalent allow the fast computation using the powering algorithms:

? V=vector(length(M),j,ihom(v,m,j-1,p))

```
[2, 5, 16, 69, 280, 1071]
\ [- same output as with direct computation -]
```
The computation of $a_{10,000}$ now takes less than a second:

```
? z=frec(V,M,10^5)[1];  \ \ result computed in 156 ms.
? 1.0*z
1.72279531330182 E55164
? z=ihom(v,m,10^5,p);  \ \ result computed in 6,768 ms.
```

34.1.4 Recurrence relations for subsequences

34.1.4.1 Two term recurrences

The recurrence for the subsequence of every $k$-th element of a two term recurrence $a_n = \alpha a_{n-1} + \beta a_{n-2}$ can be obtained as follows. Write

$$a_{n+0} = A_0 a_n + B_0 a_{n-0} = 2a_n - 1a_{n-0} \quad (34.1-13a)$$

$$a_{n+1} = A_1 a_n + B_1 a_{n-1} = \alpha a_n + \beta a_{n-1} \quad (34.1-13b)$$

$$a_{n+2} = A_2 a_n + B_2 a_{n-2} = (\alpha^2 + 2\beta) a_n - \beta^2 a_{n-2} \quad (34.1-13c)$$

$$a_{n+3} = A_3 a_n + B_3 a_{n-3} = (\alpha^3 + 3\alpha\beta) a_n + \beta^3 a_{n-3} \quad (34.1-13d)$$

$$a_{n+4} = A_4 a_n + B_4 a_{n-4} = (\alpha^4 + 4\alpha^2\beta + 2\beta^2) a_n - \beta^4 a_{n-4} \quad (34.1-13e)$$

$$a_{n+k} = A_k a_n + B_k a_{n-k} \quad (34.1-13f)$$

We have $a_n = A_k a_{n-k} + B_k a_{n-2k}$ where $A_0 = 2$, $A_1 = \alpha$ and $A_{k+1} = \alpha A_k + \beta A_{k-1}$ (and $B_k = (-\beta)^k$).

That is, the first coefficient $A_k$ of the recursion relations for the subsequences can be computed by the original recurrence relation. For efficient computation use

$$[A_k, A_{k+1}] = \begin{bmatrix} 2, \alpha \end{bmatrix} \begin{bmatrix} 0 & \beta \alpha \\ 1 & \alpha \end{bmatrix}^k \quad (34.1-14)$$

A closed form for $A_k$ in terms of Chebyshev polynomials is given in [33 item 14]:

$$A_k = 2(-\beta)^{k/2} T_k \left( \alpha/\sqrt{-4\beta} \right) \quad (34.1-15)$$

A simple example, let $F_n$ and $L_n$ denote the $n$-th Fibonacci and Lucas number, respectively. Then $\alpha = \beta = 1$ and

$$[A_k, A_{k+1}] = \begin{bmatrix} 2, 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^k = [L_k, L_{k+1}] \quad (34.1-16)$$

That is

$$F_{kn+e} = L_k F_{(n-1)+e} - (-1)^k F_{(n-2)+e} \quad (34.1-17)$$

where $k \in \mathbb{Z}$ and $e \in \mathbb{Z}$. The variable $e$ expresses the shift invariance of the relation.

34.1.4.2 Recurrences of order $n$

For the stride-$s$ recurrence relations of order $n$ the following may be the most straightforward algorithm. Let $p(x)$ be the characteristic polynomial of the recurrence and $M$ its companion matrix. Then the characteristic polynomial of $M^n$ corresponds to the recurrence relation of the stride-$s$ subsequence.

```
1 \texttt{recsubseq(n, s, m=0)=}
2 \{ /* Return vector coefficients of the stride-s subsequence}
3 \* of the n-th order linear recurrence.
4 */
5 local(p, M, z, r);
6 if ( 0==m,

[fxtbook draft of 2008-August-17]
Chapter 34: Recurrences and Chebyshev polynomials

For the second order recurrence we get what we have already seen for $s = 0, \ldots, 4$:

For the third order recurrence we get:

In general, for a recurrence $a_n = \sum_{k=1}^{K} m_k a_{n-k}$ with given $a_0, a_1, \ldots, a_K$ one has

$$\frac{x}{1 - x - x^2} = 0 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + 13x^7 + 21x^8 + 34x^9 + \ldots$$

$$\frac{2 - x}{1 - x - x^2} = 2 + x + 3x^2 + 4x^3 + 7x^4 + 11x^5 + 18x^6 + 29x^7 + 47x^8 + \ldots$$

$$\sum_{k=0}^{\infty} F_k x^k$$

$$\sum_{k=0}^{\infty} L_k x^k$$

$$\sum_{j=0}^{K} b_j x^j = \sum_{j=0}^{\infty} a_j x^j$$
34.1: Recurrences

where the denominator is the reciprocal polynomial of the characteristic polynomial and

\[ b_0 = a_0 \]  
\[ b_1 = a_1 - (a_0 m_1) \]  
\[ b_2 = a_2 - (a_0 m_2 + a_1 m_1) \]  
\[ b_3 = a_3 - (a_0 m_3 + a_1 m_2 + a_2 m_1) \]  
\[ b_k = a_k - \sum_{j=0}^{k-1} a_j m_{k-j} \]

As an example we choose the sequence

\[ [0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, \ldots] \]

with the recurrence relation \( a_n = a_{n-1} + a_{n-2} + a_{n-2} \):

\[
\begin{align*}
\text{? } a &= [0, 0, 1]; \\
\text{? } m &= [1, 1, 1]; \\
\text{? } K &= \text{length}(m); \\
\text{? } b &= \text{vector}(K, k, a[k] - \sum_{j=0}^{k-2} a[j+1] \cdot m[k-j-1]) \\
\text{? } \text{pb} &= \text{sum}(j=0, K-1, b[j+1] \cdot x^j) \\
\text{? } \text{pr} &= 1 - \sum_{k=1}^{K} m[k] \cdot x^k \\
\text{? } \text{gen} &= \frac{\text{pb}}{\text{pr}} \text{ \ (the generating function)} \\
\text{t} &= \text{taylor} (\text{gen}, x); \\
\text{t} &= \text{truncate}(\text{t}); \\
\text{? } \text{for}(j=0, \text{poldegree}(\text{t}), \text{print}(j, " ", \text{polcoeff}(\text{t}, j))) \\
0 & \text{ a0} \\
1 & \text{ a1} \\
2 & \text{ a1} \\
3 & \text{ a1} \\
4 & \text{ a1} \\
5 & \text{ a1} \\
6 & \text{ a1} \\
7 & \text{ a1} \\
\end{align*}
\]

Note that the denominator is the reciprocal of the characteristic polynomial. The general form of the expressions for a two term linear recurrence can be obtained using symbols:

\[
\begin{align*}
\text{? } a &= [a_0, a_1]; \\
\text{? } m &= [m_1, m_2]; \\
\text{? } K &= \text{length}(m); \\
\text{? } b &= \text{vector}(K, k, a[k] - \sum_{j=0}^{k-2} a[j+1] \cdot m[k-j-1]) \\
\text{? } \text{pb} &= \text{sum}(j=0, K-1, b[j+1] \cdot x^j) \\
\text{? } \text{pr} &= 1 - \sum_{k=1}^{K} m[k] \cdot x^k \\
\text{? } \text{gen} &= \frac{\text{pb}}{\text{pr}} \text{ \ (the generating function)} \\
\text{t} &= \text{taylor} (\text{gen}, x); \\
\text{t} &= \text{truncate}(\text{t}); \\
\text{? } \text{for}(j=0, \text{poldegree}(\text{t}), \text{print}(j, " ", \text{polcoeff}(\text{t}, j))) \\
0 & \text{ a0} \\
1 & \text{ a1} \\
2 & \text{ a1} \\
3 & \text{ a1} \\
4 & \text{ a1} \\
5 & \text{ a1} \\
6 & \text{ a1} \\
7 & \text{ a1} \\
\end{align*}
\]

34.1.6 Binet forms for recurrences

A closed form expression for the Fibonacci numbers is

\[ F_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right] \]
For a two-term recurrence \( a_n = m_1 a_{n-1} + m_2 a_{n-2} \) a closed form solution is given by
\[
a_n = \frac{1}{w} \left[ (a_1 - a_0 M)^n - (a_1 - a_0 P)^n \right] 
\]
where \( w = \sqrt{m_1^2 + 4m_2}, \ P = (m_1 + w)/2 \) and \( M = (m_1 - w)/2 \).

In general such formulas can be obtained as exemplified using a three-term recurrence: let \( a_n = m_1 a_{n-1} + m_2 a_{n-2} + m_3 a_{n-3} \), its characteristic polynomial is \( p(x) = x^3 - (m_1 x^2 + m_2 x + m_3) \). Let \( r_0, r_1, r_2 \) be the roots of \( p(x) \), then \( a_n = c_0 r_0^n + c_1 r_1^n + c_2 r_2^n \) if \( c_0, c_1, c_2 \) satisfy
\[
\begin{align*}
a_0 &= c_0 + c_1 + c_2 \\
a_1 &= r_0 c_0 + r_1 c_1 + r_2 c_2 \\
a_2 &= r_0^2 c_0 + r_1^2 c_1 + r_2^2 c_2
\end{align*}
\]
That is, we have to solve the matrix equation \( Z \cdot c = a \) for the vector \( c \) where \( a \) is the vector of starting values and
\[
Z = \begin{bmatrix} 1 & 1 & 1 \\ r_0 & r_1 & r_2 \\ r_0^2 & r_1^2 & r_2^2 \end{bmatrix}
\]
Verification with the three term recurrence \( a_n = a_{n-1} + a_{n-2} + a_{n-3} \) starting with \( a_0 = a_1 = 0 \) and \( a_2 = 1 \):
\[
\begin{align*}
\text{?} & a=[0,0,1]^\sim; \\
\text{?} & m=[1,1,1]^\sim; \\
\text{?} & K=length(m); \\
\text{?} & p=x^K-sum(k=1,K,m[k]*x^-K(k)) \ \text{\textbackslash \ characteristic polynomial} \\
\text{?} & x^3 - x^2 - x - 1 \\
\text{?} & r=polroots(p)) \\
\text{\{1.8392867, -0.419643 - 0.606290*1I, -0.419643 + 0.6062907*1I\}^-} \\
\text{?} & Z=matrix(K,K,RI,CI,r[CI](r[RI]-1)) \\
\text{\{1.839286 -0.419643 - 0.6062907*1I, -0.419643 + 0.6062907*1I}} \\
\text{\{3.382975 -0.1914978 + 0.5088517*1I, -0.1914978 - 0.5088517*1I\}^-} \\
\text{?} & c=matlsolve(Z,a) \\
\text{\{0.1828035 + 1.8947 E-20*I, -0.09140176 - 0.3405465*1I, -0.0914017 + 0.3405465*1I\}^-} \\
\text{?} & norm(Z*c-a) \ \text{\textbackslash \ check solution} \\
\text{\{1.147 E-39, 6.795 E-39, 3.673 E-39\}^-} \\
\text{?} & seq(n)=sum(k=0,K-1,c[k]*r[k+1]*r[CI](r[RI]-1)) \\
\text{\{0 0 1 1 2 4 7 13 24 44 81 149 274 504 927 1705 3136 5768 10609 19513 35890\}^-} \\
\text{?} & n=0:0.2,print1(" ",round(seq(n)))) \\
\text{0 0 1 1 2 4 7 13 24 44 81 149 274 504 927 1705 3136 5768 10609 19513 35890}\end{align*}
\]
The method fails if the characteristic polynomial has multiple roots because then the matrix \( Z \) is singular.

### 34.1.6.1 The special case \( c_k = 1 \)

Let \( p(x) \) be the characteristic polynomial of a recurrence, with roots \( r_i \): \( p(x) = \prod_k (x - r_k) \). We want to determine the generating function for the recurrence such that \( a_j = \sum_k r_k^j \) (that is, all constants \( c_k \) are one). For the reciprocal polynomial \( h(x) \) of \( p(x) \) we have \( h(x) = \prod_k (1 - r_k x) \), and (using the product rule for differentiation)
\[
h'(x) = h(x) \sum_k \frac{-r_k}{1 - r_k x}
\]
With \( r/(1 - r x) = \sum_{j \geq 0} r^{j+1} x^j \) we find that
\[
- \frac{h'(x)}{h(x)} = \sum_{j \geq 0} \left( \sum_k r_k^{j+1} \right) x^j
\]
That is \( a_j = \sum_k r_k^{j+1} \), and \( c_k = 1 \) for all \( k \). The relation is the key to the fast computation of the trace vector in finite fields, see relation [40.3-6] on page 895.
34.1.6.2 Binet form with multiple roots of the characteristic polynomial

When the characteristic polynomial has multiple roots the Binet form has coefficients that are polynomials in \( n \). For example, for the characteristic polynomials \( p(x) = (x - r_0)^3(x - r_1) \) the Binet form would be

\[
\begin{align*}
    a_0 &= (c_0 + 0d_0 + 0^2 e_0) + c_1 \\
    a_1 &= r_0 (c_0 + 1d_0 + 1^2 e_0) + r_1 c_1 \\
    a_2 &= r_0^2 (c_0 + 2d_0 + 2^2 e_0) + r_1^2 c_1
\end{align*}
\]

With \( n = 0, 1, \) and \( 2 \) we obtain the system of equations

\[
\begin{align*}
    a_0 &= (c_0 + 0d_0 + 0^2 e_0) + c_1 \\
    a_1 &= r_0 (c_0 + 1d_0 + 1^2 e_0) + r_1 c_1 \\
    a_2 &= r_0^2 (c_0 + 2d_0 + 2^2 e_0) + r_1^2 c_1
\end{align*}
\]

In general, the coefficient of the power of the \( k \)-th root \( r_k \) in the Binet form must be a polynomial of degree \( m_k - 1 \) where \( m_k \) is the multiplicity of \( r_k \).

34.1.7 Logarithms of generating functions *

A seemingly mysterious relation for the generating function of the Fibonacci numbers

\[
f(x) := \frac{1}{1 - x - x^2} = 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + \ldots = \sum_{k=0}^{\infty} F_{k+1} x^k
\]

is

\[
\log(f(x)) = x + \frac{1}{2} 3x^2 + \frac{1}{3} 4x^3 + \frac{1}{4} 7x^4 + \frac{1}{5} 11x^5 + \ldots = \sum_{k=1}^{\infty} \frac{1}{k} L_k x^k
\]

where \( L_k \) are the Lucas numbers. Similarly,

\[
g(x) := \frac{1}{1 - 2x - x^2} = 1 + 2x + 5x^2 + 12x^3 + 29x^4 + 70x^5 + 169x^6 + \ldots
\]

\[
\log(g(x)) = 2 \left[ x + \frac{1}{2} 3x^2 + \frac{1}{3} 7x^3 + \frac{1}{4} 17x^4 + \frac{1}{5} 41x^5 + \frac{1}{6} 99x^6 + \ldots \right]
\]

Now set \( f(x) := \frac{1}{h(x)} \), then

\[
\frac{d}{dx} \log(f(x)) = \frac{d}{dx} \log \left( \frac{1}{h(x)} \right) = -\frac{h'(x)}{h(x)}
\]

The expression \( \frac{h'(x)}{h(x)} \) is again the generating function of a recurrence and formal integration of the Taylor series terms gives the factors \( \frac{1}{k} \). The observation is a special case of the algorithm for the computation of the logarithm for powers series given in section 31.3 on page 636.

34.2 Chebyshev polynomials

The *Chebyshev polynomials* of the first (\( T \)) and second (\( U \)) kind can be defined by the functions

\[
\begin{align*}
    T_n(x) &= \cos[n \arccos(x)] \\
    U_n(x) &= \frac{\sin[(n + 1) \arccos(x)]}{\sqrt{1 - x^2}}
\end{align*}
\]

For integral \( n \) both of them are polynomials. The first few polynomials are given in figure 34.2-A (first kind) and figure 34.2-B (second kind).
### Chapter 34: Recurrences and Chebyshev polynomials

\[
T_{-n}(x) = T_n(x) \\
T_{-1}(x) = x \\
T_0(x) = 1 \\
T_1(x) = x \\
T_2(x) = 2x^2 - 1 \\
T_3(x) = 4x^3 - 3x \\
T_4(x) = 8x^4 - 8x^2 + 1 \\
T_5(x) = 16x^5 - 20x^3 + 5x \\
T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1 \\
T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x \\
T_8(x) = 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1 \\
T_9(x) = 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x \\
T_{10}(x) = 512x^{10} - 1280x^8 + 1120x^6 - 400x^4 + 50x^2 - 1 \\
T_{11}(x) = 1024x^{11} - 2816x^9 + 2816x^7 - 1232x^5 + 220x^3 - 11x
\]

**Figure 34.2-A:** The first few Chebyshev polynomials of the first kind.

\[
U_{-n}(x) = -U_{n-2}(x) \\
U_{-2}(x) = -1 \\
U_{-1}(x) = 0 \\
U_0(x) = 1 \\
U_1(x) = 2x \\
U_2(x) = 4x^2 - 1 \\
U_3(x) = 8x^3 - 4x \\
U_4(x) = 16x^4 - 12x^2 + 1 \\
U_5(x) = 32x^5 - 32x^3 + 6x \\
U_6(x) = 64x^6 - 80x^4 + 24x^2 - 1 \\
U_7(x) = 128x^7 - 192x^5 + 80x^3 - 8x \\
U_8(x) = 256x^8 - 448x^6 + 240x^4 - 40x^2 + 1 \\
U_9(x) = 512x^9 - 1024x^7 + 672x^5 - 160x^3 + 10x \\
U_{10}(x) = 1024x^{10} - 2304x^8 + 1792x^6 - 560x^4 + 60x^2 - 1 \\
U_{11}(x) = 2048x^{11} - 5120x^9 + 4608x^7 - 1792x^5 + 280x^3 - 12x
\]

**Figure 34.2-B:** The first few Chebyshev polynomials of the second kind.
34.2: Chebyshev polynomials

One has

\[ T_n(x) = \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{(n-k-1)!}{k!(n-2k)!} (2x)^{n-2k} \]  
\[ = \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{1}{n-k} \binom{n-k}{k} (2x)^{n-2k} \]  
\[ = \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{n}{2k} x^{n-2k} (x^2-1)^k \]  

(34.2-4a)  
(34.2-4b)  
(34.2-4c)

and

\[ U_n(x) = \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{(n-k)!}{k!(n-2k)!} (2x)^{n-2k} \]  
\[ = \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \binom{n-k}{k} (2x)^{n-2k} \]  
\[ = \sum_{k=0}^{\lfloor n/2+1 \rfloor} \frac{n+1}{2k+1} x^{n-2k} (x^2-1)^k \]  

(34.2-5a)  
(34.2-5b)  
(34.2-5c)

The indexing of \( U \) seems to be slightly unfortunate, having \( U_0 = 0 \) would render many of the relations for the Chebyshev polynomials more symmetric.

The \( n+1 \) extrema of \( T_n(x) \) are located at the points \( x_k = \cos \frac{k \pi}{n} \) where \( k = 0, 1, 2, \ldots, n \) and \(-1 \leq x_k \leq +1\), which can be seen from the definition. The values at those points are \( \pm 1 \). The \( n \) zeros lie at \( x_k = \cos \frac{(k-1/2) \pi}{n} \) where \( k = 1, 2, 3, \ldots, n \).

The expansion of \( x^n \) in terms of Chebyshev polynomials of the first kind is, for \( n \) even,

\[ x^n = \frac{1}{2^n} \binom{n}{n/2} + \frac{1}{2^{n-1}} \sum_{k=0}^{n/2-1} \binom{n}{k} T_{n-2k}(x) \]  

(34.2-6a)

and, for odd \( n \),

\[ x^n = \frac{1}{2^{n-1}} \sum_{k=0}^{(n-1)/2} \binom{n}{k} T_{n-2k}(x) \]  

(34.2-6b)

For the Chebyshev polynomials of the first kind one has

\[ T_n \left( \frac{x + 1/x}{2} \right) = \frac{x^n + 1/x^n}{2} \]  

(34.2-7)

This relation can be used to find a solution of \( T_n(x) = z \) directly. Indeed

\[ x = \frac{R_n + 1/R_n}{2} \text{ where } R_n := \left( z + \sqrt{z^2 - 1} \right)^{1/n} \]  

(34.2-8)

is a solution which can be chosen to be real if \( z \in \mathbb{R} \) and \( z > 1 \). Thereby we have the closed form expression

\[ T_n(z) = \frac{r^n + r^{-n}}{2} \text{ where } r := \left( z + \sqrt{z^2 - 1} \right) \]  

(34.2-9)
34.2.1 Recurrence relation, generating functions, and the composition law

Both types of Chebyshev polynomials obey the same recurrence (omitting the argument \( x \))

\[
N_n = 2xN_{n-1} - N_{n-2}
\]  (34.2-10)

where \( N \) can be either symbol, \( T \) or \( U \). Recurrence relations for subsequences are:

\[
\begin{align*}
N_{n+1} &= [2x] \cdot N_n - N_{n-1} \quad (34.2-11a) \\
N_{n+2} &= \left[2(2x^2 - 1)\right] \cdot N_n - N_{n-2} \quad (34.2-11b) \\
N_{n+3} &= \left[2(4x^3 - 3x)\right] \cdot N_n - N_{n-3} \quad (34.2-11c) \\
N_{n+4} &= \left[2(8x^4 - 8x^2 + 1)\right] \cdot N_n - N_{n-4} \quad (34.2-11d) \\
N_{n+5} &= \left[2(16x^5 - 20x^3 + 5x)\right] \cdot N_n - N_{n-5} \quad (34.2-11e) \\
N_{n+s} &= \left[2T_s(x)\right] \cdot N_n - N_{n-s} \quad (34.2-11f)
\end{align*}
\]

The generating functions are

\[
\begin{align*}
\frac{1 - xt}{1 - 2xt + t^2} &= \sum_{n=0}^{\infty} t^n T_n(x) \quad (34.2-12a) \\
\frac{1}{1 - 2xt + t^2} &= \sum_{n=0}^{\infty} t^n U_n(x) \quad (34.2-12b)
\end{align*}
\]

Quick check of relation 34.2-12a using pari/gp:

\[
? \text{gen=truncate(taylor((1-t*x)/(1-2*x*t+t^2),t));} \\
? \text{for(k=0,5,print(k,":\",polcoeff(gen,k,t)))};
\]

\[
\begin{align*}
0: & 1 \\
1: & 2x - 1 \\
2: & 4x^2 - 3x + 1 \\
3: & 8x^3 - 5x^2 + 1 \\
4: & 16x^4 - 8x^3 + 5x^2 - 1 \\
5: & 32x^5 - 20x^4 + 5x^3 - 3x + 1
\end{align*}
\]

Binet forms for \( T \) (compare with relation 34.2-9) and \( U \) are

\[
\begin{align*}
T_n(z) &= \frac{1}{2} \left[ (z + \sqrt{z^2 - 1})^n + (z - \sqrt{z^2 - 1})^n \right] \quad (34.2-13a) \\
U_n(z) &= \frac{1}{2\sqrt{z^2 - 1}} \left[ (z + \sqrt{z^2 - 1})^{n+1} - (z - \sqrt{z^2 - 1})^{n+1} \right] \quad (34.2-13b)
\end{align*}
\]

Composition is multiplication of indices as can be seen by the definition (relation \( \text{34.2-1a} \) on page 677):

\[
T_n(T_m(x)) = T_{nm}(x) \quad (34.2-14)
\]

For example,

\[
\begin{align*}
T_{2n}(x) &= T_2(T_n(x)) = 2T_n^2(x) - 1 \quad (34.2-15a) \\
&= T_n(T_2(x)) = T_n(2x^2 - 1) \quad (34.2-15b)
\end{align*}
\]

34.2.2 Index-doubling and relations between \( T \) and \( U \)

Index-doubling relations for the polynomials of the first kind are

\[
\begin{align*}
T_{2n} &= 2T_n^2 - 1 \quad (34.2-16a) \\
T_{2n+1} &= 2T_n + T_{n-1} - x \quad (34.2-16b) \\
T_{2n-1} &= 2T_nT_{n-1} - x \quad (34.2-16c)
\end{align*}
\]

[fxtbook draft of 2008-August-17]
Similar relations for the polynomials of the second kind are

\[ U_{2n} = U_n^2 - U_{n-1}^2 = (U_n + U_{n-1}) (U_n - U_{n-1}) \]  
\[ = U_n (U_n - U_{n-2}) - 1 = U_{n-1} (U_{n+1} - U_{n-1}) + 1 \]  
\[ U_{2n+1} = U_n (U_{n+1} - U_{n-1}) \]  
\[ = 2U_n (U_{n+1} - xU_n) = 2U_n (xU_n - U_{n-1}) \]  
\[ U_{2n-1} = U_{n-1} (U_n - U_{n-2}) \]  
\[ = 2U_{n-1} (U_n - xU_{n-1}) = 2U_{n-1} (xU_{n-1} - U_{n-2}) \]

Some relations between \( T \) and \( U \) are

\[ T_n = U_n - xU_{n-1} = xU_{n-1} - U_{n-2} = \frac{1}{2} (U_n - U_{n-2}) \]  
\[ T_{n+1} = xT_n - (1 - x^2)U_{n-1} \]  
\[ U_{2n} = 2T_n U_n - 1 \]  
\[ U_{2n-1} = 2T_n U_{n-1} = 2 (T_{n+1} U_n + x) \]  
\[ U_{2n+1} = 2T_{n+1} U_n = 2 (T_{n+2} U_{n-1} + x) \]  
\[ U_{2n-1} = 2^{n-1} \prod_{k=0}^{n-1} T_{2k} \]

Relation 34.2-18b written as

\[ U_n = \frac{xT_{n+1} - T_{n+2}}{1 - x^2} = \frac{T_n - xT_{n+1}}{1 - x^2} \]

can be used to compute the polynomials of the second kind from those of the first kind. One further has:

\[ T_{n+m} + T_{n-m} = 2T_n T_m \]  
\[ T_{n+m} - T_{n-m} = 2 (x^2 + 1) U_{n-1} U_{m-1} \]  
\[ U_{n+m-1} + U_{n-m-1} = 2 U_{n-1} T_m \]  
\[ U_{n+m-1} - U_{n-m-1} = 2 T_n U_{m-1} \]

Expressions for certain sums:

\[ \sum_{k=0}^{n} T_{2k} = \frac{1}{2} (1 + U_{2n}) \]  
\[ \sum_{k=0}^{n-1} T_{2k+1} = \frac{1}{2} U_{2n-1} \]  
\[ \sum_{k=0}^{n} U_{2k} = \frac{1}{2} - \frac{T_{2n+2}}{1 - x^2} \]  
\[ \sum_{k=0}^{n-1} U_{2k+1} = \frac{x - T_{2n+1}}{2 (1 - x^2)} \]

Using \( \partial_x \cos(n \arccos(x)) = n \sin(n \arccos(x))/\sqrt{1-x^2} \) we obtain

\[ \partial_x T_n(x) = n U_{n-1}(x) \]
34.2.3 Fast computation of the Chebyshev polynomials

We give algorithms that improve on both the matrix power, and the polynomial based algorithms.

34.2.3.1 Chebyshev polynomials of the first kind

For even index use relation (34.2-16a) \( T_{2n} = 2 T_n^2 - 1 \). For odd index we use relations (34.2-16c) and (34.2-16b). We compute the pair \([T_{n-1}, T_n]\) recursively via

\[
[T_{n-1}, T_n] = \begin{cases} 
2 T_q - 1, T_q - x & \text{if } n \text{ even} \\
T_q^2 - 1, 2 T_q - x & \text{if } n \text{ odd}
\end{cases}
\]

where \( q = n/2 \), if \( n \) even \( (34.2-23a) \)

\[
[T_{n-1}, T_n] = \begin{cases} 
2 T_q^2 - 1, 2 T_q - x & \text{if } n \text{ even} \\
T_q^2 - 1, T_q - x & \text{if } n \text{ odd}
\end{cases}
\]

where \( q = (n+1)/2 \), if \( n \) odd \( (34.2-23b) \)

Note that no multiplication with \( x \) occurs thereby the computation is efficient also for floating point arguments. With integer \( x \) the cost of the computation of \( T_n(x) \) is \( \sim M(n) \) where \( M(n) \) is the cost of a multiplication of numbers with the precision of the result. When \( x \) is a floating point number the cost is \( \sim \log_2(n) M(n) \) where \( M(n) \) is the cost of a multiplication with the precision used.

The code for the pair computations is

```c
1 fvT(n, x)=
2 { /* return [ T(n-1,x), T(n,x) ] */
3    local(nr, t, t1, t2);
4    if ( n<=1,
5        if ( 1==n, return( [1, x] ) );
6        if ( 0==n, return( [x, 1] ) );
7        if ( -1==n, return( [2*x^2-1, x] ) );
8        return( 0 ); \ disallow negative index < -1
9    );
10   nr = (n+1) >> 1; \ if ( "n even", nr = n/2 , nr = (n+1)/2; );
11   vr = fvT(nr, x); \ recursion
12   t1 = vr[1]; t2 = vr[2];
13   if ( !bitand(n,1), \ n is even
14      t = [2*t1*t2-x, 2*t2^2-1];
15    ,
16      t = [2*t1^2-1, 2*t1*t2-x];
17    );
18   return( t );
19 }
```

The function to be called by the user is

```c
1 fT(n, x)=
2 {
3    local(q, t, v, T);
4    n = abs(n);
5    if ( n<=1,
6        if ( n>0, return(if(0==n,1,x)));
7        return( fT(-n, x) );
8    );
9    q = 0; t = 0;
10   while ( 0==bitand(q, 1), q++=1; t+=1; );
11   \ here: n=q^2*t
12   T = fvT(q, x)[2];
13   while ( t, T=2*T*T-1; t-=1; );
14    return( T );
15 }
```

We check the speedup by comparing with the matrix-power computation that gives identical results. We compute \( T_{3,455,967}(2) \), a number with more than 2,600,000 decimal digits:

\[ vT(n,x)= \text{return( ([1, x]*[0,-1; 1,2*x]^n) )} \]
\[ x=2; \ \ \text{\textbackslash want integer calculations} \]
\[ n=4645967; \]
\[ vT(n,x); \ \ \text{\textbackslash computed in 9,800 ms.} \]
\[ fvT(n,x); \ \ \text{\textbackslash computed in 2,241 ms.} \]

C++ implementations for the computation of \( T_n(2) \) and \( T_n(x) \) modulo \( m \) are given in [FXT: mod/chebyshev1.cc].
34.2.3.2 Chebyshev polynomials of the second kind

One can use the fast algorithm for the polynomials of the first kind and relation (34.2-19) \( U_n = (T_n - x T_{n+1})/(1 - x^2) \), involving a division:

```c
fU(n, x) =
{
    local(v);
    if ( 1==x, return(n+1) ); \ avoid division by zero
    v = fT(n+1, x);
    return( (v[1]-x*v[2])/(1-x^2) );
}
```

We give an additional algorithm that uses 3 multiplication for each reduction of the index \( n \). One multiplication is by the variable \( x \). We compute the pair \([U_{n-1}, U_n]\) recursively via

\[
M_q := (U_q + U_{q-1}) (U_q - U_{q-1})
\]

\[
[U_{n-1}, U_n] = [2U_{q-1} (U_q - x U_{q-1}), M_q] \quad \text{where} \quad q = n/2, \quad \text{if} \ n \text{ even}
\]

\[
[U_{n-1}, U_n] = [M_q, 2U_q (x U_q - U_{q-1})] \quad \text{where} \quad q = (n - 1)/2, \quad \text{if} \ n \text{ odd}
\]

The code for the pair computations is

```c
fvU(n, x)=
{
    /* return [ U(n-1,x), U(n,x) ] */
    local(nr, u1, u0, ue, t, u);
    if ( n<=1,
        if ( 1==n, return( [1, (2*x)] ) );
        if ( 0==n, return( [0, 1] ) );
        if ( -1==n, return( [-1, 0] ) );
        if ( -2==n, return( [-(-2*x), -1] ) );
        return( 0 ); \ disallow negative index < -2
    );
    nr = n >> 1; \ if ( "n even", nr = n/2 , nr = (n-1)/2; );
    vr = fvU(nr, x); \ recursion
    u1 = vr[1]; u0 = vr[2];
    ue = (u0+u1) * (u0-u1);
    if ( !bitand(n,1), \ n is even
        t = u1*(u0-x*u1); t+=t;
        u = [t, ue];
    ,
        t = u0*(x*u0-u1); t+=t;
        u = [ue, t];
    );
    return( u );
}
```

The function to be called by the user is

```c
fU(n, x)= return( fU(n,x)[2] );
```

The comparison with the matrix-power computation shows almost the same speedup as for the polynomials of the first kind:

```c
vU(n,x)= return( [0, 1]*[0,-1; 1,2*x]^n );
x=2; \ want integer calculations
n=4545967;
vU(n,x); \ computed in 9,783 ms.
fvU(n,x); \ computed in 2,704 ms.
```

C++ implementations for the computation of \( U_n(x) \) and \( U_n(x) \) modulo \( m \) are given in [FXT: mod/chebyshev2.cc].

34.2.3.3 Symbolic computation

For symbolic computations the explicit power series as in (34.2-4a) or (34.2-5a) on page 679 should be preferred. The following routine computes \( T_n \) as a polynomial in \( x \):

```c
[fxtbook draft of 2008-August-17]
34.2.4 Relations to approximations of the square root

34.2.4.1 Padé approximants for $\sqrt{x^2 + 1}$

We start with the relation (from the definitions \[34.2-1\]a and \[34.2-1\]b on page 677 and $\sin^2 + \cos^2 = 1$)

$$T_n^2 - (x^2 - 1) U_{n-1}^2 = 1 \quad (34.2-25)$$

which we write as

$$\sqrt{x^2 - 1} = \sqrt{\frac{T_n^2 - 1}{U_{n-1}^2}} \quad (34.2-26)$$

If we define $R_n = T_n/U_{n-1}$, then

$$R_n(x) = \frac{T_n}{U_{n-1}} \approx \sqrt{x^2 - 1} \quad (34.2-27)$$

A composition law holds for $R$:

$$R_{mn}(x) = R_m(R_n(x)) \quad (34.2-28)$$

We list the first few values of $R_k(2)$ and $R_k(x)$:

\[
\begin{array}{ccc}
 k & R_k(2) & R_k(x) \\
 1 & 2/1 & x/1 \\
 2 & 7/4 & \frac{2x^2 - 1}{2x} \\
 3 & 26/15 & \frac{4x^3 - 3x}{4x^2 - 1} \\
 4 & 97/56 & \frac{8x^4 - 8x^2 + 1}{8x^3 - 4x} \\
 5 & 362/209 & \frac{16x^5 - 20x^3 + 5x}{16x^4 - 12x^2 + 1} \\
 \infty & \sqrt{3} & \sqrt{x^2 - 1}
\end{array}
\]
If we define $T^+_n(x):= T(i x)/i^n$ and $U^+_n(x):= U(i x)/i^n$ then
\[ T^+_n - (x^2 + 1) U^+_{n-1} = 1 \] (34.2-30)

Defining $R^+_n := T^+_n/U^+_n$ we have
\[ R^+_mn(x) = R^+_m(R^+_n(x)) \] (34.2-31)

and
\[ \sqrt{x^2 + 1} = \sqrt{T^+_n U^+_{n-1}} \approx \frac{T^+_n}{U^+_{n-1}} = R^+_n(x) \] (34.2-32)

The first few values of $R^+_k(1)$ and $R^+_k(x)$ are

\[ \begin{array}{cccc}
  k: & R^+_k(1) & R^+_k(x) \\
  1: & 1/1 & x/1 \\
  2: & 3/2 & \frac{2x^2 + 1}{2x} \\
  3: & 7/5 & \frac{4x^3 + 3x}{4x^2 + 1} \\
  4: & 17/12 & \frac{8x^4 + 8x^2 + 1}{8x^3 + 4x} \\
  5: & 41/29 & \frac{16x^5 + 20x^3 + 5x}{16x^4 + 12x^2 + 1} \\
  \infty: & \sqrt{2} & \sqrt{x^2 + 1} \\
\end{array} \] (34.2-33)

Relations 34.2-30 and 34.2-25 can be used to power solutions of Pell’s Diophantine equation, see relation 37.13-13a on page 814.

34.2.4.2 Two products for the square root

For those fond of products: for $d > 0$, $d \neq 1$
\[ \sqrt{d} = \prod_{k=0}^{\infty} \left( 1 + \frac{1}{q_k} \right) \] where $q_0 = \frac{d+1}{d-1}$, $q_{k+1} = 2q_k^2 - 1$ (34.2-34)

(convergence is quadratic) and
\[ \sqrt{d} = \prod_{k=0}^{\infty} \left( 1 + \frac{2}{h_k} \right) \] where $h_0 = \frac{d+3}{d-1}$, $h_{k+1} = (h_k + 2)^2 (h_k - 1) + 1$ (34.2-35)

(convergence is cubic). These are given in [32] and also in [118], more expressions can be found in [109].

The paper gives $h_{k+1} = \frac{4d}{d-1} \prod_{i=0}^{k} (h_i^2) - 3$. Note that for relation 34.2-34 we have
\[ q_k = T_{2^k}(q_0) \] (34.2-36)
\[ \frac{1}{q_k} = \frac{(d-1)^N}{\sum_{i=0}^{N} \left( \frac{2^N}{2^i} \right) d^i} = \frac{2 (1 - d)^N}{(1 + \sqrt{d})^{2N} + (1 - \sqrt{d})^{2N}} \] where $N = 2^k$ (34.2-37)

where $T_n$ is the $n$-th Chebyshev polynomial of the first kind. One finds
\[ q_k = T_{2^k}(1/c) \] where $c = \frac{1 - d}{1 + d}$, $c < 1$ (34.2-38)
and

\[
\sqrt{\frac{1-c}{1+c}} \approx \frac{1-c}{c} \frac{U_{2^{k-1}}(1/c)}{T_{2^k}(1/c)} \tag{34.2-39}
\]

which can be expressed in \(d = \frac{1-c}{1+c}\) as

\[
\sqrt{d} \approx \frac{2d}{1-d} \frac{U_{2^{k-1}}(\frac{1+d}{1-d})}{T_{2^k}(\frac{1+d}{1-d})} \quad \text{where} \quad d > 1 \tag{34.2-40}
\]

where \(U_n\) is the \(n\)-th Chebyshev polynomial of the second kind. We have \(U_{2^{k-1}}(x) = 2^k \prod_{i=0}^{k-1} T_{2^i}(x)\). Successively compute \(T_{2^i} = 2 T_{2^{i-1}}^2 - 1\) and accumulate the product \(U_{2^{k-1}} = 2 U_{2^{k-1}-1} T_{2^{k-1}}\) until \(U_{2^k-1}\) and \(T_{2^k}\) are obtained. Alternatively use the relation \(U_k(x) = \frac{1}{k+1} \partial_x T_{k+1}(x)\) and use the recursion for the coefficients of \(T\) as shown in section 34.2.3.3 on page 683.

A systematic approach to find product expressions for roots is given in section 28.7 on page 586.
Chapter 35

Cyclotomic polynomials, Hypergeometric functions, and continued fractions

We describe the cyclotomic polynomials and some of their properties, together with the Möbius inversion principle. We also give algorithms to convert a power series into Lambert series and infinite products.

We describe the hypergeometric functions which contain most of the ‘useful’ functions such as the logarithm and the sine as special cases. The transformation formulas for hypergeometric functions can be used to obtain series transformations that are non-obvious. The computation of certain hypergeometric functions by AGM-type algorithms is described in section 30.3 on page 608.

Further continued fractions are described together with algorithms for their computation.

35.1 Cyclotomic polynomials, Möbius inversion, Lambert series

35.1.1 Cyclotomic polynomials

The roots (over \( \mathbb{C} \)) of the polynomial \( x^n - 1 \) are the \( n \)-th roots of unity:

\[
x^n - 1 = \prod_{k=0}^{n-1} \left( x - \exp \left( \frac{2 \pi i k}{n} \right) \right)
\]  

The degree of \( Y_n \) equals the number of primitive \( n \)-th roots of unity:

\[
\deg(Y_n) = \varphi(n)
\]

The coefficients are integers, for example,

\[
Y_{63}(x) = x^{36} - x^{33} + x^{27} - x^{24} + x^{18} - x^{12} + x^9 - x^3 + 1
\]
The first 30 cyclotomic polynomials are shown in figure 35.1-A. The first cyclotomic polynomial with a coefficient not in the set \( \{0, \pm 1\} \) is \( Y_{105} \):

\[
Y_{105}(x) = x^{48} + x^{47} + x^{46} - x^{43} - x^{42} - 2 \cdot x^{41} - x^{40} - x^{39} + \ldots
\]  

(35.1-5)

The cyclotomic polynomials are irreducible over \( \mathbb{Z} \). All except \( Y_1 \) are self-reciprocal.

For \( n \) prime the cyclotomic polynomial \( Y_n(x) \) equals \( (x^n - 1)/(x - 1) = x^{n-1} + x^{n-2} + \ldots + x + 1 \). For \( n = 2k \) and odd \( k \geq 3 \) we have \( Y_n(x) = Y_k(-x) \). For \( n = pk \) where \( p \) is a prime that does not divide \( k \) we have \( Y_n(x) = Y_k(x^p)/Y_p(x) \). The following algorithm for the computation of \( Y_n(x) \) is given in [128, p.403]:

1. Let \([p_1, p_2, \ldots, p_r]\) the distinct prime divisors of \( n \). Set \( y_0(x) = x - 1 \).

2. For \( j = 1, 2, \ldots, r \) set \( y_j(x) = y_j(x^{p_j})/y_j(x) \) (the division is exact).

3. Return \( y_r(x^{n/(p_1 p_2 \ldots p_r)}) \)

The last statement uses the fact that for \( n = kt \) where all prime factors of \( k \) divide \( t \) we have \( Y_n(x) = Y_t(x^k) \). An implementation is

```plaintext
1 polycyclo2(n, z='x)=
2 { local(fc, y);
3  fc = factor(n)[,1]; \ \ \ \ \text{prime divisors}
4  y = z - 1;
5  for (j=1, #fc, y=subst(y,z,z^fc[j]) \ y; n\=fc[j]; );
6  y = subst(y, z, z^n);
7  return( y );
8 }
```

Note that the routine will only work when the argument \( z \) is a symbol.
35.1.2 The Möbius inversion principle

The Möbius function \( \mu(n) \) is defined for positive integer arguments \( n \) as

\[
\mu(n) := \begin{cases} 
0 & \text{if } n \text{ has a square factor} \\
(-1)^k & \text{if } n \text{ is a product of } k \text{ distinct primes} \\
+1 & \text{if } n = 1
\end{cases}
\] (35.1-6)

The function satisfies

\[
\sum_{d \mid n} \mu(d) = \begin{cases} 
1 & \text{if } n = 0 \\
0 & \text{else}
\end{cases}
\] (35.1-7)

A function \( f(n) \) defined that satisfies

\[
f(n \cdot m) = f(n) \cdot f(m) \text{ if } \gcd(n, m) = 1
\] (35.1-8)

is said to be multiplicative. For a multiplicative function one always has \( f(1) = 1 \) and \( f(n) = f(p_1^{e_1}) \cdot f(p_2^{e_2}) \cdot \cdots f(p_k^{e_k}) \) where \( n = p_1^{e_1} \cdot p_2^{e_2} \cdot \cdots p_k^{e_k} \) is the factorization of \( n \) into distinct primes \( p_i \). If the equality holds also for \( \gcd(n, m) \neq 1 \) the function is said to be completely multiplicative. Such a function satisfies \( f(n) = f(p_1^{e_1}) \cdot f(p_2^{e_2}) \cdot \cdots f(p_k^{e_k}) \).

The Möbius function is multiplicative

\[
\mu(n) \mu(m) = \begin{cases} 
\mu(n \cdot m) & \text{if } \gcd(n, m) = 1 \\
0 & \text{else}
\end{cases}
\] (35.1-9)

For the cyclotomic polynomials one has

\[
x^n - 1 = \prod_{d \mid n} Y_d(x)
\] (35.1-10)

and

\[
Y_n(x) = \prod_{d \mid n} (x^d - 1)^{\mu(n/d)}
\] (35.1-11)

The relation implies a reasonably efficient algorithm for the computation of the cyclotomic polynomials. The method also works when the argument \( x \) is not a symbol.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \mu(n) )</th>
<th>( n )</th>
<th>( \mu(n) )</th>
<th>( n )</th>
<th>( \mu(n) )</th>
<th>( n )</th>
<th>( \mu(n) )</th>
<th>( n )</th>
<th>( \mu(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+1</td>
<td>11</td>
<td>-1</td>
<td>21</td>
<td>+1</td>
<td>31</td>
<td>-1</td>
<td>41</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>12</td>
<td>0</td>
<td>22</td>
<td>+1</td>
<td>32</td>
<td>0</td>
<td>42</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>13</td>
<td>-1</td>
<td>23</td>
<td>-1</td>
<td>33</td>
<td>+1</td>
<td>43</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>14</td>
<td>+1</td>
<td>24</td>
<td>0</td>
<td>34</td>
<td>+1</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>15</td>
<td>+1</td>
<td>25</td>
<td>0</td>
<td>35</td>
<td>+1</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>+1</td>
<td>16</td>
<td>0</td>
<td>26</td>
<td>+1</td>
<td>36</td>
<td>0</td>
<td>46</td>
<td>+1</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>17</td>
<td>-1</td>
<td>27</td>
<td>0</td>
<td>37</td>
<td>-1</td>
<td>47</td>
<td>-1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>28</td>
<td>0</td>
<td>38</td>
<td>+1</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>19</td>
<td>-1</td>
<td>29</td>
<td>-1</td>
<td>39</td>
<td>+1</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>+1</td>
<td>20</td>
<td>0</td>
<td>30</td>
<td>-1</td>
<td>40</td>
<td>0</td>
<td>50</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 35.1-B: Values of the Möbius function \( \mu(n) \) for \( n \leq 80 \).

The pair of relations 35.1-10 and 35.1-11 is actually a special case of the (multiplicative) Möbius inversion principle:

\[
g(n) = \prod_{d \mid n} f(d) \iff f(n) = \sum_{d \mid n} g(d)^{\mu(n/d)}
\] (35.1-12)
Relation 35.1-10 implies (considering the polynomial degrees only and using relation 35.1-3)

\[ n = \sum_{d|n} \varphi(d) \]  

(35.1-13)

while relation 35.1-11 corresponds to the equality

\[ \varphi(n) = \sum_{d|n} d \mu(n/d) \]  

(35.1-14)

Relations 35.1-13 and 35.1-14 are a special case of the additive version of the Möbius inversion principle:

\[ g(n) = \sum_{d|n} f(d) \iff f(n) = \sum_{d|n} g(d) \mu(n/d) \]  

(35.1-15)

More general, if \( h(ab) = h(a) h(b) \) (see [103, p.447]) then

\[ g(n) = \sum_{d|n} f(d) h(n/d) \iff f(n) = \sum_{d|n} g(d) h(n/d) \mu(n/d) \]  

(35.1-16)

Setting \( h(n) = 1 \) gives relation 35.1-15. The Möbius inversion principle is nicely explained in [134]. The sequence of the values of the Möbius function (see figure 35.1-B) is entry A008683 of [245].

We note two relations valid for multiplicative functions \( f \):

\[ \sum_{d|n} \mu(d) f(d) = \prod_{d|n, \text{d prime}} (1 - f(d)) \]  

(35.1-17a)

\[ \sum_{d|n} \mu(d)^2 f(d) = \prod_{d|n, \text{d prime}} (1 + f(d)) \]  

(35.1-17b)

Relation 35.1-17a with \( f(n) = 1/n \) gives relation 35.1-13 and also

\[ \varphi(n) = n \prod_{d|n, \text{d prime}} \left(1 - \frac{1}{d}\right) \]  

(35.1-18)

35.1.3 Lambert series

A Lambert series is an expansion of the form

\[ L(x) = \sum_{k>0} a_k x^k = \sum_{k>0} \sum_{j>0} a_k x^{kj} \]  

(35.1-19)

It can be converted to a Taylor series

\[ L(x) = \sum_{k>0} b_k x^k \quad \text{where} \quad b_k = \sum_{d|k} a_d \]  

(35.1-20)

The inversion principle allows us to transform a Taylor series to a Lambert series:

\[ a_k = \sum_{d|k} b_d \mu(k/d) \]  

(35.1-21)

With pari/gp the conversion to a Lambert series can be implemented as
Cylotomic polynomials, Möbius inversion, Lambert series

ser2lambert(t) =

/* Let t=[a1,a2,a3, ...], n=length(v), where t(x)=sum_{k=1}^{n}{a_k*x^k};
* Return L=[l1,l2,l3,...] so that (up to order n)
* t(x)=\sum_{j=1}^{n}{l_j*x^j/(1-x^j)}
*/
local(n, L);
 n = length(t);
 L = vector(n);
for (k=1, n, fordiv(k, d, L[k]+=moebius(k/d)*t[d]); );
return( L );
}

The conversion in the other direction is
lambert2ser(L)=

{ /* inverse of ser2lambert() */
local(n, t);
 n = length(L);
 t = sum(k=1, length(L), O('x^(n+1))+L[k]*'x^k/(1-'x^k) );
 t = Vec(t);
return( t );
}

For the Lambert series with \(a_k = 1\) for all \(k\) we have

\[
\sum_{k>0} d(k) x^k = \sum_{k>0} \frac{x^k}{1-x^k} = \sum_{k>0} \sum_{j>0} x^{kj} = \sum_{k>0} \frac{1+x^k}{1-x^k} x^{k^2}
\]  \hspace{1cm} (35.1-22)

where \(d(k)\) is the number of the divisors of \(k\), entry \([A000005]\) of \([245]\). More generally, we have

\[
\sum_{k>0} \frac{k^e x^k}{1-x^k} = \sum_{k>0} \sigma_e(k) x^k
\]  \hspace{1cm} (35.1-23)

where \(\sigma_e(n)\) is the sum of the \(e\)-th powers of the divisors of \(n\). We note two relations, the first is from \([173\text{ p.644, ex.27}]\):

\[
\sum_{k>0} \frac{x^k}{1-x^k} = \sum_{k>0} \left[ k x^k \prod_{j \geq k+1} (1-x^j) \right] = \sum_{k>0} \left[ 1 - \prod_{j \geq k} (1-x^j) \right]
\]  \hspace{1cm} (35.1-24a)

For the Lambert series with \(a_k = \mu(k)\) we have

\[
x = \sum_{k>0} \frac{\mu(k) x^k}{1-x^k}
\]  \hspace{1cm} (35.1-25)

For \(a_k = \alpha^k\) we have

\[
\sum_{k>0} \frac{\alpha^k x^k}{1-x^k} = \sum_{k>0} \frac{\alpha x^k}{1-\alpha x^k}
\]  \hspace{1cm} (35.1-26)

This is given in \([169\text{ p.468}]\), also the following: If \(\sum_{k>0} (a_k x^k) / (1-x^k) = f(x)\) and \(\sum_{k>0} a_k x^k = g(x)\) then \(f(x) = \sum_{k>0} g(x^k)\).

We note a relation that is a useful for the computation of the sum, it is given in \([173\text{ p.644, ex.27}]\),

\[
L(x) = \sum_{k>0} x^{k^2} \left[ a_k + \sum_{j>0} (a_k + a_{k+j}) x^{kj} \right]
\]  \hspace{1cm} (35.1-27)

For the related series

\[
P(x) = \sum_{k>0} \frac{a_k x^k}{1+x^k} = -\sum_{k>0} \sum_{j>0} (-1)^j a_k x^{kj}
\]  \hspace{1cm} (35.1-28)

[fxtbook draft of 2008-August-17]
we find (by computing the \( k \)-th term on both sides: \( a_k x^k/(1 + x^k) = a_k x^k/(1 - x^k) - 2 a_k x^{2k}/(1 - x^{2k}) \))

\[
P(x) = L(x) - 2 L(x^2)
\]  \(35.1-29\)

The other direction is obtained by repeatedly using \( L(x) = P(x) + 2 L(x^2) \):

\[
L(x) = \sum_{k=0}^{\infty} 2^k P(x^k)
\]  \(35.1-30\)

Use relations [35.1-27] and [35.1-29] to obtain

\[
P(x) = \sum_{k>0} x^k \left[ a_k (1 - 2 x^{2k}) + \sum_{j>0} (a_k + a_{k+j}) \left( x^{k+j} - 2 x^{(k+j)^2 - j^2} \right) \right]
\]  \(35.1-31\)

### 35.1.4 Conversion of series to infinite products

Given a series with constant term one,

\[
f(x) = 1 + \sum_{k>0} a_k x^k
\]  \(35.1-32\)

we want to find an infinite product such that

\[
f(x) = \prod_{k>0} (1 - x^k)^{b_k}
\]  \(35.1-33\)

We take the logarithm, differentiate, and multiply by \( x \):

\[
x \frac{f'(x)}{f(x)} = \sum_{k>0} \frac{(-k b_k) x^k}{1 - x^k}
\]  \(35.1-34\)

The expression on the right hand side is a Lambert series with coefficients \(-k b_k\), the expression on the left is easily computable as a power series, and we know how to compute a Lambert series from a power series. Thereby

\[
b_k = -\frac{1}{k} \sum_{d|k} q_k \mu(k/d)
\]  \(35.1-35\)

where the \( q_k \) are the coefficients of the power series for \( q(x) := x f'(x)/f(x) \). With pari/gp the conversion to a product can be implemented as

```plaintext
1  ser2prod(t)=
2  { /* Let t=[1,a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^n{a_k*x^k};
3   * Return p=[p1,p2,p3,...] so that (up to order n)
4   * t(x)=\prod_{j=1}^n{(1-x^j)^{p_j}}
5   */
6   local(v);
7   v = Ser(t);
8   v = v'/v;
9   v = vector(#t-1, j, polcoeff(v, j-1));
10  v = ser2lambert(v);
11  v = vector(#v, j, -v[j]/j);
12  return( v );
13 }
```

A simple example is \( f(x) = \exp(x) \), so \( x f'/f = x \), and

\[
\exp(x) = \prod_{k>0} (1 - x^k)^{-\mu(k)/k} = \frac{(1 - x^2)^{1/2} (1 - x^3)^{1/3} (1 - x^5)^{1/5} \ldots}{(1 - x)^{1/1} (1 - x^6)^{1/6} (1 - x^{10})^{1/10} \ldots}
\]  \(35.1-36\)
Taking the logarithm, we obtain

\[ x = - \sum_{k>0} \frac{\mu(k)}{k} \log(1 - x^k) \]  

(35.1-37)

Setting \( f(x) = 1 - 2x \) we obtain relation 17.2-6a on page 370 (number of binary Lyndon words):

? ser2prod(Vec(1-2*x+O(x^20)))

\[ [2, 1, 2, 3, 6, 9, 18, 30, 56, 99, 186, 335, 630, 1161, 2182, 4080, 7710, 14532, 27594] \]

Setting \( f(x) = 1 - x - x^2 \) gives the number of binary Lyndon words without the subsequence 00 (entry A006206 of [245]):

? ser2prod(Vec(1-x-x^2+O(x^20)))

\[ [1, 1, 1, 1, 2, 2, 4, 5, 8, 11, 18, 25, 40, 58, 90, 135, 210, 316, 492] \]

The ordinary generating function for the \( e_k \) corresponding to the product form \( f(x) = \prod (1 - x^k)^{c_k} \) is

\[ \sum_{k=1}^{\infty} e_k x^k = - \sum_{k=1}^{\infty} \frac{\mu(k)}{k} \log(f(x^k)) \]  

(35.1-38)

This can be seen by using the product form for \( f \) on the right hand side, using the power series \( \log(1 - x) = -(x + x^2/2 + x^3/3 + \ldots) \), and using the defining property of the Möbius function (relation 35.1-7 on page 689). An example is relation 17.2-6b on page 370. For the cyclotomic polynomials we obtain (via relation 35.1-11 on page 689):

\[ - \sum_{k=1}^{\infty} \frac{\mu(k)}{k} \log(Y_n(x^k)) = \sum_{d|n} \mu(d) x^{n/d} \]  

(35.1-39)

For example, setting \( n = 2 \) we obtain

\[ x^2 - x = - \sum_{k=1}^{\infty} \frac{\mu(k)}{k} \log(1 + x^k) \]  

(35.1-40)

### 35.1.4.1 An alternative product form

For the transformation into products of the form \( \prod (1 + x^k)^c_k \) we set

\[ f(x) = \prod_{k>0} (1 + x^k)^{c_k} \]  

(35.1-41)

and note that

\[ x \frac{f'(x)}{f(x)} = \sum_{k>0} \frac{(k c_k) x^k}{1+x^k} \]  

(35.1-42)

So we need a transformation into series of this type. As the Möbius transform is not (easily) applicable we use a greedy algorithm:

```plaintext
ser2lambertplus(t)=
{
    /* Let t=[a1,a2,a3, ...], n=length(v), where t(x)=sum_{k=1}^{n}{a_k*x^k};
    * Return L=[l1,l2,l3,...] so that (up to order n)
    * t(x)=sum_{j=1}^{n}{l_j*x^j/(1+x^j)}
    */
    local(n, L, k4);
    n = length(t);
    L = vector(n);
    for (k=1, n, tk = t[k]);
    for (k=1, n)
```
Chapter 35: Cyclotomic polynomials, Hypergeometric functions, and continued fractions

L[k] = tk;
\subtract tk * x^k/(1+x^k):
forstep(j=k, n, 2*k, t[j] -= tk);
forstep(j=k+k, n, 2*k, t[j] += tk);
);
return( L );
}

Now we can compute the product form via

\begin{verbatim}
ser2prodplus(t)=
{
/* Let t=[1,a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^{n}{a_k*x^k}; */
* Return p=[p1,p2,p3, ...] so that (up to order n)
* t(x)=\prod_{j=1}^{n}{(1+x^j)^{p_j}}
*/
local(v);
v = Ser(t);
v = v'/v;
v = vector(#t-1, j, polcoeff(v, j-1));
v = ser2lambertplus(v);
v = vector(#v, j, v[j]/j);
return( v );
}
\end{verbatim}

A product \( \prod_{k>0}(1-x^k)^{b_k} \) can be converted into a product \( \prod_{k>0}(1+x^k)^{c_k} \) via the relation 
\((1-x) = \prod_{k \geq 0}(1+x^{2k})^{1}\).  

35.1.4.2 Conversion to eta-products

\begin{verbatim}
r= 2: ( E(y^2)^3 ) / ( E(y^4) )
r= 3: ( E(y^3)^4 ) / ( E(y^9) )
r= 4: ( E(y^4)^7 ) / ( E(y^8)^3 )
r= 5: ( E(y^5)^6 ) / ( E(y^25) )
r= 6: ( E(y^6)^12 E(y^{36}) ) / ( E(y^{12})^4 E(y^{18})^3 )
r= 7: ( E(y^7)^8 ) / ( E(y^{49}) )
r= 8: ( E(y^8)^15 ) / ( E(y^{16})^7 )
r= 9: ( E(y^9)^13 ) / ( E(y^{27})^4 )
r=10: ( E(y^10)^18 E(y^{100}) ) / ( E(y^{20})^6 E(y^{50})^3 )
r=11: ( E(y^{11})^12 ) / ( E(y^{121}) )
r=12: ( E(y^{12})^28 E(y^{72})^3 ) / ( E(y^{24})^12 E(y^{36})^7 )
r=13: ( E(y^{13})^14 ) / ( E(y^{169}) )
r=14: ( E(y^{14})^24 E(y^{196}) ) / ( E(y^{28})^8 E(y^{98})^3 )
r=15: ( E(y^{15})^24 E(y^{225}) ) / ( E(y^{45})^6 E(y^{75})^4 )
r=16: ( E(y^{16})^31 ) / ( E(y^{32})^15 )
r=17: ( E(y^{17})^18 ) / ( E(y^{289}) )
r=18: ( E(y^{18})^39 E(y^{108})^4 ) / ( E(y^{36})^13 E(y^{54})^12 )
r=19: ( E(y^{19})^20 ) / ( E(y^{361}) )
r=20: ( E(y^{20})^42 E(y^{200})^3 ) / ( E(y^{40})^18 E(y^{100})^7 )
r=21: ( E(y^{21})^32 E(y^{441}) ) / ( E(y^{63})^8 E(y^{147})^4 )
r=22: ( E(y^{22})^36 E(y^{484}) ) / ( E(y^{44})^12 E(y^{242})^3 )
r=23: ( E(y^{23})^24 ) / ( E(y^{529}) )
r=24: ( E(y^{24})^60 E(y^{144})^7 ) / ( E(y^{48})^28 E(y^{72})^15 )
r=25: ( E(y^{25})^31 ) / ( E(y^{125})^6 )
r=26: ( E(y^{26})^42 E(y^{676}) ) / ( E(y^{52})^14 E(y^{338})^3 )
r=27: ( E(y^{27})^40 ) / ( E(y^{81})^13 )
r=28: ( E(y^{28})^56 E(y^{392})^3 ) / ( E(y^{56})^24 E(y^{196})^7 )
r=29: ( E(y^{29})^30 ) / ( E(y^{841}) )
r=30: ( E(y^{30})^72 E(y^{180})^6 E(y^{300})^4 E(y^{450})^3 ) / ( E(y^{60})^24 E(y^{90})^18 E(y^{150})^12 E(y^{900}) )
r=31: ( E(y^{31})^32 ) / ( E(y^{961}) )
r=32: ( E(y^{32})^63 ) / ( E(y^{64})^31 )
r=33: ( E(y^{33})^48 E(y^{1089}) ) / ( E(y^{99})^12 E(y^{363})^4 )
\end{verbatim}

Figure 35.1-C: Functions \( \eta_r(y) := \prod_{j=0}^{r-1} \eta(\omega_j y) \) as products of \( \eta \)-functions.
35.1: Cylotomic polynomials, Möbius inversion, Lambert series

The conversion of a series to a product of the form \((\eta \text{-product})\),

\[
\prod_{k=1}^{\infty} \left[ \eta(x^k) \right]^{u_k}
\]

where \(\eta(x) := \prod_{j=1}^{\infty} 1 - x^j\) (35.1-43)

can be done by a greedy algorithm:

```plaintext
1 ser2etaprod(v) =
2 { /* Let t=[a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^{n}{a_k*x^k}; */
3   local(n, t);
4   v = ser2prod(v);
5   n = length(v);
6   for (k=1, n,
7       t = v[k];
8       forstep (j=k+k, n, k, v[j]-=t; );
9     );
10   return( v );
11 }
```

Similarly, to convert into a product of the form

\[
\prod_{k=1}^{\infty} \left[ \eta_+(x^k) \right]^{u_k}
\]

where \(\eta_+(x) = \prod_{k>0} (1+x^k)\) use

```plaintext
1 ser2etaprodplus(v) =
2 { /* Let t=[1,a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^{n}{a_k*x^k}; */
3   local(n, t);
4   v = ser2prodplus(v);
5   n = length(v);
6   for (k=1, n,
7       t = v[k];
8       forstep (j=k+k, n, k, v[j]-=t; );
9     );
10   return( v );
11 }
```

The routines are useful for computations with the generating functions of partitions of certain types, see section 14.4 on page 336. Here we just give:

\[
\eta(-x) = \frac{\eta(x^2)^3}{\eta(x) \eta(x^4)} \quad (35.1-45a)
\]

\[
\eta(+i x) \eta(-i x) = \frac{\eta(x^4)^8}{\eta(x^2)^3 \eta(x^8)^3} \quad (35.1-45b)
\]

Figure 35.1-C gives more product formulas. We note two relations with Lambert series taken from [169, p.468]:

\[
\eta(x) = \exp \left( -\sum_{k>0} \frac{1}{k} \frac{x^k}{1-x^k} \right) = \exp \left( -\sum_{k>0} x^k \sum_{d|k} \frac{1}{d} \right) \quad (35.1-46a)
\]

\[
\eta_+(x) = \exp \left( -\sum_{k>0} \frac{(-1)^k}{k} \frac{x^k}{1-x^k} \right) \quad (35.1-46b)
\]
Further expression for $\eta$ and $\eta_+$ are

\[
\eta(x) = 1 - \sum_{k>0} x^k \prod_{j \geq k+1} (1 - x^j) = 1 - \sum_{k>0} x^k \prod_{j=1}^{k-1} (1 - x^j) \tag{35.1-47a}
\]

\[
\eta_+(x) = 1 + \sum_{k>0} x^k \prod_{j \geq k+1} (1 + x^j) = 1 + \sum_{k>0} x^k \prod_{j=1}^{k-1} (1 + x^j) \tag{35.1-47b}
\]

The eta functions are closely connected to the generating functions for integer partitions, see section 14.4 on page 336.

35.2 Hypergeometric functions

The hypergeometric function $F\left(\begin{array}{c} a, b \\ c \end{array} \mid z \right)$ can be defined as

\[
F\left(\begin{array}{c} a, b \\ c \end{array} \mid z \right) := \sum_{k=0}^{\infty} \frac{a^\overline{k} b^\overline{k}}{c^\overline{k}} \frac{z^k}{k!} \tag{35.2-1}
\]

where $z^\overline{k} := z(z+1)(z+2)\ldots(z+k-1)$ is the rising factorial power ($z^\overline{0} := 1$). Some sources use the so-called Pochhammer symbol $(x)_k$ which is the same: $(x)_k = x^\overline{k}$. We'll stick to the factorial notation.

The variable $z$ is the argument of the function, $a, b$ and $c$ are the parameters. Parameters in the upper and lower row are called upper and lower parameters, respectively.

Note the $k! = 1^\overline{k}$ in the denominator of relation 35.2-1. You might want to have the hidden lower parameter 1 in mind:

\[
F\left(\begin{array}{c} 2, 2 \\ 1 \end{array} \mid z \right) = _{a}F\left(\begin{array}{c} 2, 2 \\ 1 \end{array} \mid z \right)
\]

The expression is a sum of perfect squares if $z$ is a square.

We have

\[
F\left(\begin{array}{c} a, b \\ c \end{array} \mid z \right) = 1 + \frac{a}{1} z \left(1 + \frac{a+1}{2} \frac{b+1}{c+1} z \left(1 + \frac{a+2}{3} \frac{b+2}{c+2} z (1 + \ldots)\right)\right) \tag{35.2-3}
\]

so by formula 35.2-3 hypergeometric functions with rational arguments can be computed with the binary splitting method described in section 32.1.

Hypergeometric functions can have any number of parameters:

\[
F\left(\begin{array}{c} a_1, \ldots, a_m \\ b_1, \ldots, b_n \end{array} \mid z \right) = \sum_{k=0}^{\infty} \frac{a_1^\overline{k} \ldots a_m^\overline{k}}{b_1^\overline{k} \ldots b_n^\overline{k}} \frac{z^k}{k!} \tag{35.2-4}
\]

These are sometimes called generalized hypergeometric functions. The number of upper and lower parameters are often emphasized as subscripts left and right to the symbol $F$. For example, $mF_n$ for the hypergeometric function in the last relation.

The functions $F\left(\begin{array}{c} a \\ b \end{array} \mid z \right)$ (of type $1F_1$) are sometimes written as $M(a, b; z)$ or $\Phi(a; b; z)$. Kummer’s function $U(a, b, z)$ (or $\Psi(a; b; z)$) is related to hypergeometric functions of type $2F_0$:

\[
U(a, b, z) = z^{-a} F\left(\begin{array}{c} a, 1+a-b \\ -1 \end{array} \mid -1/z \right) \tag{35.2-5}
\]
Note that series $2F_0$ are not convergent. Still, they can be used as asymptotic series for large values of $z$.

The so-called Whittaker functions are related to hypergeometric functions as follows:

$$M_{a,b}(z) = e^{-z/2} z^{b+1/2} F\left(\frac{1}{2} + b - a \mid 1 + 2b, z\right)$$  \hspace{1cm} (35.2-6a)$$

$$W_{a,b}(z) = e^{-z/2} z^{b+1/2} U\left(\frac{1}{2} + b - a, 1 + 2b, z\right)$$  \hspace{1cm} (35.2-6b)$$

$$= e^{-z/2} z^a F\left(\frac{1}{2} + b - a, \frac{1}{2} - b - a \mid -1/z\right)$$  \hspace{1cm} (35.2-6c)$$

Negative integer parameters in the upper row lead to polynomials:

$$F\left(-3, \frac{3}{1} \mid z\right) = 1 - 9z + 18z^2 - 10z^3$$ (35.2-7)$$

The lower parameter must not be zero or a negative integer unless there is a negative upper parameter with smaller absolute value.

Sometimes one finds the notational convention to omit an argument $z = 1$:

$$F\left(a_1, \ldots, a_m \mid b_1, \ldots, b_n\right) := F\left(a_1, \ldots, a_m \mid 1\right)$$ (35.2-8)$$

In what follows the argument is never omitted.

An in-depth treatment of hypergeometric functions is [15].

### 35.2.1 Derivative and differential equation

Using the relation

$$\frac{d}{dz^n} F\left(a, b \ldots \mid c, \ldots\right) = \frac{a^n b^n \ldots}{c^n \ldots} F\left(a + n, b + n \ldots \mid c + n, \ldots\right)$$ (35.2-9)$$

one can verify that $f(z) = F\left(a, b \mid c\right)$ is a solution of the differential equation

$$z \left(1 - z\right) \frac{d^2 f}{dz^2} + \left[c - (1 + a + b) z\right] \frac{df}{dz} - abf = 0$$ (35.2-10)$$

A general form of the differential equation satisfied by $F\left(a, b, c, \ldots \mid u, v, w, \ldots\right)$ is

$$z (\vartheta + a) (\vartheta + b) (\vartheta + c) \ldots f(z) = \vartheta (\vartheta + u - 1) (\vartheta + v - 1) (\vartheta + w - 1) \ldots f(z)$$ (35.2-11)$$

where $\vartheta$ is the operator $z \frac{d}{dz}$. The leftmost $\vartheta$ on the right hand side of the equation takes care of the hidden lower parameter 1: $\vartheta = (\vartheta + 1 - 1)$. See [134] for a beautiful derivation. Relation 10.15-4a on page 284 can be used to rewrite powers of $\vartheta$ as polynomials in $\frac{d}{dz}$.

### 35.2.2 Evaluations for fixed $z$

A closed form (in terms of the gamma function) evaluation at $z = 1$ can be given for $2F_1$:

$$F\left(a, b \mid c\right) = \frac{\Gamma(c) \Gamma(c - a - b)}{\Gamma(c - a) \Gamma(c - b)} \quad \text{if} \quad \Re(c - a - b) > 0 \quad \text{or} \quad b \in \mathbb{N}, b < 0$$ (35.2-12)$$
Chapter 35: Cyclotomic polynomials, Hypergeometric functions, and continued fractions

When \( c - a - b < 0 \) then, see [278, ex.18, p.299]
\[
\lim_{z \to -1} F\left( \frac{a}{c}, \frac{b}{c} \bigg| z \right) = \frac{\Gamma(1 - a + b) \Gamma(1 + a/2)}{\Gamma(1 + a) \Gamma(1 + a/2 - b)} (35.2-13a)
\]
and, for \( c - a - b = 0 \),
\[
\lim_{z \to -1} F\left( \frac{a}{c}, \frac{b}{c} \bigg| z \right) = \frac{\Gamma(1 - a + b) \Gamma(1/a)}{\Gamma(1 + a/2) \Gamma((1 + a/2) - b)} (35.2-13b)
\]

For \( z = -1 \) there is an evaluation due to Kummer:
\[
F\left( \frac{a}{1 + a - b}, \frac{b}{1 + a - b} \bigg| -1 \right) = \frac{\Gamma(1 - a + b) \Gamma(1 + a/2)}{\Gamma(1 + a) \Gamma(1 + a/2 - b)} (35.2-14a)
\]
\[
= 2^{-a} \pi \frac{\Gamma(1 - a + b)}{\Gamma(1/2 + a/2) \Gamma(1 + a/2 - b)} (35.2-14b)
\]

Several evaluations at \( z = \frac{1}{2} \) are given in [1], we just give one:
\[
F\left( \frac{a}{2}, \frac{b}{2 + a + b} \bigg| \frac{1}{2} \right) = \sqrt{\pi} \frac{\Gamma\left( \frac{1}{2} + \frac{1}{2} a + \frac{1}{2} b \right)}{\Gamma\left( \frac{1}{2} + \frac{1}{2} a \right) \Gamma\left( \frac{1}{2} + \frac{1}{2} b \right)} (35.2-15)
\]

For further information see (chapter 15 of) [1], [278, ex.18, p.299]. Various evaluations of \( F\left( \frac{-a_n, b_n+b_1}{c_n+c_1} \bigg| z \right) \)
for integer \( a, b, c \) and \( n, 1 \leq a \leq 2, -4 \leq b \leq 4 \) and \(-4 \leq c \leq 4 \) can be found in [114].

35.2.3 Extraction of even and odd part

Let \( E[f(z)] = (f(z) + f(z))/2 \) (the even powers of the series of \( f(z) \)), and \( O[f(z)] = (f(z) - f(-z))/2 \) (the odd powers). We express the even and odd parts of a hypergeometric series as hypergeometric functions:

\[
E \left[ F\left( \frac{a}{c}, \frac{b}{c} \bigg| z \right) \right] = F\left( \frac{a}{2}, \frac{a+1}{2}, \frac{b}{2}, \frac{b+1}{2} \bigg| z^2 \right) (35.2-16a)
\]
\[
O \left[ F\left( \frac{a}{c}, \frac{b}{c} \bigg| z \right) \right] = \frac{ab}{c} z F\left( \frac{a+1}{2}, \frac{a+2}{2}, \frac{b+1}{2}, \frac{b+2}{2} \bigg| z^2 \right) (35.2-16b)
\]

The lowers parameters 1/2 and 3/2 are due to the hidden lower parameter 1. The general case for
\[
H(z) := F\left( \frac{a_1, \ldots, a_m}{b_1, \ldots, b_n} \bigg| z \right) (35.2-17a)
\]
is
\[
E [H(z)] = F\left( \frac{a_1}{b_1}, \frac{a_1+1}{b_1}, \ldots, \frac{a_m}{b_n}, \frac{a_m+1}{b_n} \bigg| X z^2 \right) (35.2-17b)
\]
\[
O [H(z)] = \frac{a_1 \cdots a_m}{b_1 \cdots b_n} z F\left( \frac{a_1+1}{b_1}, \frac{a_1+2}{b_1}, \ldots, \frac{a_m+1}{b_n}, \frac{a_m+2}{b_n} \bigg| X z^2 \right) (35.2-17c)
\]

where \( X = 4^{m-n-1} \). For example,
\[
E \left[ F\left( \frac{1}{2} \bigg| z \right) \right] = F\left( \frac{1}{2}, \frac{1}{2} \bigg| \frac{z^2}{4} \right) = F\left( \frac{3}{4}, \frac{3}{4} \bigg| \frac{z^2}{4} \right) = \frac{\sinh z}{z} (35.2-18)
\]
We indicate a further generalization by the extraction of all terms of $H(z)$ where the exponent of $z$ is divisible by 3:

$$
\frac{H(z) + H(\omega z) + H(\omega^2 z)}{3} = F \left( \frac{a_1}{3}, \frac{a_1+1}{3}, \frac{a_1+2}{3}, \ldots, \frac{a_m}{3}, \frac{a_m+1}{3}, \frac{a_m+2}{3} \left| \frac{X z^3}{3} \right. \right) \tag{35.2-19}
$$

where $\omega = \exp(2i\pi/3)$ and $X = 27^{m-n-1}$. For example, with $H(z) = \exp(z) = F \left( \left| \frac{z}{3} \right. \right)$ we obtain

$$
F \left( \left| \frac{1}{3}, \frac{2}{3} \right. \right) = \sum_{k=0}^{\infty} \frac{z^{3k}}{(3k)!} \tag{35.2-20}
$$

Define the power series $C_j(z)$, for $j \in \{0, 1, 2\}$, by

$$
C_s(z) = \sum_{k=0}^{\infty} \frac{z^{3k+s}}{(3k+s)!} \tag{35.2-21a}
$$

then (omitting arguments)

$$
\det \begin{bmatrix} C_0 & C_1 & C_2 \\ C_2 & C_0 & C_1 \\ C_1 & C_2 & C_0 \end{bmatrix} = C_0^3 + C_1^3 + C_2^3 - 3C_0 C_1 C_2 = 1 \tag{35.2-21b}
$$

which is a three power series analogy to the relation $\cosh^2 - \sinh^2 = 1$.

For the extraction of the coefficient at the positions $j \mod M$ replace every upper and lower parameter $A$ by the $M$ parameters $(A + j)/M$, $(A + j + 1)/M$, $(A + j + 2)/M$, $\ldots$, $(A + j + M - 1)/M$, and the argument $z$ by $X z^M$ where $X = (M^2)^{m-n-1}$.

### 35.2.4 Transformations

As obvious from the definition, parameters in the upper row can be swapped (capitalized symbols for readability):

$$
F \left( A, B, C \left| e, f, g \right. \right) = F \left( B, A, C \left| e, f, g \right. \right) \tag{35.2-22}
$$

The same is true for the lower row. Usually one writes the parameters in ascending order. Identical elements in the lower and upper row can be canceled:

$$
F \left( a, b, C \left| e, f, C \right. \right) = F \left( a, b \left| e, f \right. \right) \tag{35.2-23}
$$

These trivial transformations are true for any number of elements. The following transformations are only valid for the given structure, unless the list of parameters contain an ellipsis ‘…’.

#### 35.2.4.1 Elementary relations

$$
F \left( a, b, \ldots \left| c, \ldots \right. \right) = 1 + z \frac{a b \ldots}{c} F \left( a + 1, b + 1, \ldots, 1 \left| c + 1, \ldots, 2 \right. \right) \tag{35.2-24}
$$

$$
(a-b) F \left( a, b, \ldots \left| c, \ldots \right. \right) = a F \left( a + 1, b, \ldots \left| c, \ldots \right. \right) - b F \left( a, b + 1, \ldots \left| c, \ldots \right. \right) \tag{35.2-25}
$$
A transformation by Gauss

\[(a - c) F\left(\frac{a, b, \ldots}{c + 1, \ldots} | z\right) = a F\left(\frac{a + 1, b, \ldots}{c + 1, \ldots} | z\right) - c F\left(\frac{a, b, \ldots}{c, \ldots} | z\right)\]  

(35.2-26)

These are given in [134], the following is taken from [278].

\[F\left(\frac{a, b}{c} | z\right) = F\left(\frac{a, b + 1}{c} | z\right) - \frac{a z}{c} F\left(\frac{a + 1, b + 1}{c + 1} | z\right)\]  

(35.2-27)

More relations of this type are given in [1].

### 35.2.4.2 Pfaff’s reflection law and Euler’s identity

Pfaff’s reflection law can be given as either of:

\[
\frac{1}{(1 - z)^a} F\left(\frac{a, b}{c} | \frac{-z}{1 - z}\right) = F\left(\frac{a, c - b}{c} | z\right)  
\]

(35.2-28a)

\[
F\left(\frac{a, b}{c} | z\right) = \frac{1}{(1 - z)^a} F\left(\frac{a, c - b}{c} | \frac{-z}{1 - z}\right)  
\]

(35.2-28b)

\[
= \frac{1}{(1 - z)^b} F\left(\frac{c - a, b}{c} | \frac{-z}{1 - z}\right)  
\]

(35.2-28c)

Euler’s identity is obtained by applying the Pfaff reflection on both upper parameters:

\[
F\left(\frac{a, b}{c} | z\right) = (1 - z)^{(c - a - b)} F\left(\frac{c - a, c - b}{c} | z\right)  
\]

(35.2-29)

Now write Euler’s transform as

\[
F\left(\frac{a, b}{a + b + r} | z\right) / F\left(\frac{a + r, b + r}{a + b + r} | z\right) = (1 - z)^r  
\]

(35.2-30)

If both the numerator and the denominator terminate then the expression on the left is a Padé approximant for the 1/r-th root, see section 28.2.3 on page 573.

Euler’s transformation can generalized for hypergeometric functions \(\frac{r+1}{r} F_r\), see [151]. We give two transforms for hypergeometric functions \(\frac{3}{2} F_2\) where one upper parameter exceeds a lower parameter by one, taken from [196, p.17]. The first one is reminiscent to the Pfaff reflection, we have

\[
F\left(\frac{a_1, a_2, e + 1}{b, e} | z\right) = (1 - z)^{-a_1} F\left(\frac{a_1, b - a_2 - 1, f + 1}{b, f} | \frac{-z}{1 - z}\right)  
\]

(35.2-31)

where \(f = e \frac{b - a_2 - 1}{e - a_2}\). The second one is similar to Euler’s identity:

\[
F\left(\frac{a_1, a_2, e + 1}{b, e} | z\right) = (1 - z)^{b - a_1 - a_2 - 1} F\left(\frac{b - a_1 - 1, b - a_2 - 1, g + 1}{b, g} | z\right)  
\]

(35.2-32)

where \(g = \frac{(b - a_1 - 1)(b - a_2 - 1)e}{(b - a_1 - a_2 - 1)e + a_1 a_2}\).

### 35.2.4.3 A transformation by Gauss

\[
F\left(\frac{2a, 2b}{a + b + \frac{1}{2}} | z\right) = F\left(\frac{a, b}{a + b + \frac{1}{2}} | 4z(1 - z)\right) \text{ where } |z| < \frac{1}{2}  
\]

(35.2-33a)

\[
F\left(\frac{a, b}{a + b + \frac{1}{2}} | z\right) = F\left(\frac{2a, 2b}{a + b + \frac{1}{2}} | \frac{1 - \sqrt{1 - z}}{2}\right)  
\]

(35.2-33b)

[fxtbook draft of 2008-August-17]
Note that the right hand side of relation \[35.2-33a\] does not change if \(z\) is replaced by \(1 - z\), so it seems that

\[ F\left(\frac{2a, 2b}{a + b + \frac{1}{2}} \middle| z\right) = F\left(\frac{2a, 2b}{a + b + \frac{1}{2}} \middle| 1 - z\right) \quad (35.2-34) \]

However, the relation is true only for terminating series, that is, for polynomials. Rewriting relation \[35.2-33a\] for the argument \(\frac{1 - z}{2}\) we obtain

\[ F\left(\frac{2a, 2b}{a + b + \frac{1}{2}} \middle| \frac{1 - z}{2}\right) = F\left(\frac{a, b}{a + b + \frac{1}{2}} \middle| 1 - z^2\right) \quad (35.2-35) \]

### 35.2.4.4 Whipple’s identity and quadratic transformations

Whipple’s identity connects two hypergeometric functions \(_3F_2:\)

\[ F\left(\frac{1}{2}a, \frac{1}{2}a + \frac{1}{2}, 1 - a - b - c \middle| \frac{-4z}{(1 - z)^2}\right) = (1 - z)^a F\left(\frac{a, b, c}{1 + a - b, 1 + a - c, 1 + a - b - c} \middle| z\right) \quad (35.2-36) \]

Specializing \[35.2-36\] for \(c = (a + 1)/2\) (note the symmetry between \(b\) and \(c\) so specializing for \(c = (b + 1)/2\) produces the identical relation) gives

\[ F\left(\frac{a, b}{1 + a - b} \middle| \frac{z}{1 + c}\right) = \frac{1 - z)^a}{(1 - z)^a} F\left(\frac{\frac{1}{2}a, \frac{1}{2} - \frac{1}{2}a + c}{1 + c} \middle| \frac{-4z}{(1 - z)^2}\right) \quad (35.2-37) \]

\[ F\left(\frac{a, b}{a + b + \frac{1}{2}} \middle| \frac{z}{(1 - \sqrt{1 - z})^2}\right) = \frac{(2(1 - \sqrt{1 - z})^2)^{2a}}{a + b + \frac{1}{2}} F\left(\frac{2a, a - b + \frac{1}{2}}{a + b + \frac{1}{2}} \middle| \frac{(1 - \sqrt{1 - z})^2}{z}\right) \quad (35.2-38) \]

With \(c := a - b\) in \[35.2-37\] one obtains:

\[ F\left(\frac{a, a - c}{1 + c} \middle| \frac{z}{1 + c}\right) = \frac{1}{(1 - z)^a} F\left(\frac{\frac{1}{2}a, \frac{1}{2} - \frac{1}{2}a + c}{1 + c} \middle| \frac{-4z}{(1 - z)^2}\right) \quad (35.2-39) \]

Similarly as for the relations by Gauss, from relations \[35.2-37\] and \[35.2-38\]

\[ F\left(\frac{a, b}{1 + a - b} \middle| -\frac{1 - z}{1 + z}\right) = \frac{1 + z)^a}{2} F\left(\frac{\frac{1}{2}a, \frac{1}{2}a + \frac{1}{2} - b}{1 + a - b} \middle| 1 - z^2\right) \quad (35.2-40a) \]

\[ F\left(\frac{a, b}{1 + a - b} \middle| -\frac{1 - \sqrt{1 - z^2}}{1 + \sqrt{1 - z^2}}\right) = \frac{(1 + \sqrt{1 - z^2})^a}{2} F\left(\frac{\frac{1}{2}a, \frac{1}{2}a + \frac{1}{2} - b}{1 + a - b} \middle| z^2\right) \quad (35.2-40b) \]

Relations \[35.2-40b\] and \[35.2-40a\] can be obtained from each other by setting \(x = \sqrt{1 - y^2}\) (and replacing \(y\) by \(x\)). The same is true for the next pair of relations:

\[ F\left(\frac{a, b}{a + b + \frac{1}{2}} \middle| 1 - z^2\right) = \left(\frac{2}{1 + z}\right)^{2a} F\left(\frac{2a, a - b + \frac{1}{2}}{a + b + \frac{1}{2}} \middle| \frac{1 - z}{1 + z}\right) \quad (35.2-41a) \]

\[ F\left(\frac{a, b}{a + b + \frac{1}{2}} \middle| z^2\right) = \left(\frac{2}{1 + \sqrt{1 - z^2}}\right)^{2a} F\left(\frac{2a, a - b + \frac{1}{2}}{a + b + \frac{1}{2}} \middle| \frac{1 - \sqrt{1 - z^2}}{1 + \sqrt{1 - z^2}}\right) \quad (35.2-41b) \]

The transformations

\[ F\left(\frac{a, b}{a - b + 1} \middle| z\right) = (1 + z)^{-a} F\left(\frac{\frac{1}{2}a, \frac{1}{2}a + \frac{1}{2}}{a - b + 1} \middle| \frac{4z}{(1 + z)^2}\right) \quad (35.2-42a) \]

\[ = (1 - z)^{-a} F\left(\frac{\frac{1}{2}a, \frac{1}{2}a - b + \frac{1}{2}}{a - b + 1} \middle| \frac{-4z}{(1 - z)^2}\right) \quad (35.2-42b) \]

\[ = (1 \pm \sqrt{z})^{-2a} F\left(\frac{a, a - b + \frac{1}{2}}{2a - 2b + 1} \middle| \frac{\pm 4\sqrt{z}}{(1 \pm \sqrt{z})^2}\right) \quad (35.2-42c) \]
are given in [1]. Specializing for \( a = b \) gives

\[
F\left(\frac{a}{1}, \frac{a}{1} \bigg| z\right) = (1+z)^{-a} F\left(\frac{1}{2}a, \frac{1}{2}a + \frac{1}{2}a \bigg| \frac{4z}{(1+z)^2}\right) \quad (35.2-42d)
\]

\[
= (1-z)^{-a} F\left(\frac{1}{2}a, \frac{1}{2}a - \frac{1}{2}a \bigg| \frac{-4z}{(1-z)^2}\right) \quad (35.2-42e)
\]

\[
= (1 \pm \sqrt{z})^{-2a} F\left(\frac{a}{1} \bigg| \pm \frac{4\sqrt{z}}{1 \pm \sqrt{z}^2}\right) \quad (35.2-42f)
\]

Relation 35.2-42c can be obtained by setting \( c = 0 \) in relation 35.2-37. Observe that the hypergeometric function on the right hand side of relation 35.2-42c does not change when replacing \( a \) by \( 1 - a \). The next \( {}_3F_2 \) transformation is given in [194]:

\[
(1-z)^{-1} F\left(\frac{a}{2} + \frac{1}{2}a, \frac{1}{2}b, 2 \bigg| z\right) = F\left(\frac{1}{2} + \frac{1}{2}a, \frac{1}{2} + \frac{1}{2}b, 1 \bigg| 4z(1-z)\right) \quad (35.2-43)
\]

The following are special cases of this transformation:

\[
(1-z)^{-1} F\left(\frac{a}{2}, 1 - \frac{a}{2} \bigg| z\right) = F\left(\frac{1}{2} + \frac{1}{2}a, 1 - \frac{1}{2}a \bigg| 4z(1-z)\right) \quad (35.2-44a)
\]

\[
(1-z)^{-1} F\left(\frac{a}{2}, 1 \bigg| z\right) = F\left(\frac{1}{2} + \frac{1}{2}a, 1 + \frac{1}{2}a, 1 \bigg| 4z(1-z)\right) \quad (35.2-44b)
\]

\[
(1-z)^{-1} F\left(\frac{a}{2} + \frac{1}{2}a \bigg| z\right) = F\left(\frac{1}{2} + \frac{1}{2}a, \frac{3}{2}, 1 \bigg| 4z(1-z)\right) \quad (35.2-44c)
\]

More quadratic (and cubic) transformations are given in [115] vol.1, pp.110-114.

The nonlinear transformation (given in [219] p.21)

\[
F\left(\frac{a}{c}, \frac{b}{c} \bigg| z\right) = (1-\omega)^{2a} \sum_{n=0}^{\infty} d_n \omega^n \quad (35.2-45a)
\]

where

\[
\omega = \frac{-4z}{(1-z)^2}, \quad z = \frac{\sqrt{1-\omega} - 1}{\sqrt{1-\omega} + 1} \quad (35.2-45b)
\]

and

\[
d_0 = 1 \quad (35.2-45c)
\]

\[
d_1 = \frac{2a(c-2b)}{c} \quad (35.2-45d)
\]

\[
d_{n+2} = \frac{2(c-2b)(n+1+a) d_{n+1} + (n+2a)(n+2a+1-c) d_n}{(n+2)(n+1+c)} \quad (35.2-45e)
\]

maps the complex \((z-)\)plane into the unit circle. Thereby the \( \omega \)-form of the series converges for all \( z \neq 1 \).

### 35.2.4.5 Clausen’s product formulas

Clausen’s formulas connect hypergeometric functions of type \( {}_2F_1 \) and \( {}_3F_2 \):

\[
F\left(\frac{1}{2} + a, \frac{1}{2} + b \bigg| 1 + a + b \bigg| z\right) = F\left(\frac{1}{2} + a + b, \frac{1}{2} + b \bigg| 2a + 2b \bigg| z\right) \quad (35.2-46a)
\]

\[
F\left(\frac{1}{2} + a, \frac{1}{2} + b \bigg| 1 + a + b \bigg| z\right) F\left(\frac{1}{2} - a, \frac{1}{2} - b \bigg| 1 - a - b \bigg| z\right) = F\left(\frac{1}{2} + a + b, \frac{1}{2} - a + b \bigg| 1 + a + b, 1 - a - b \bigg| z\right) \quad (35.2-46b)
\]
If \( a = b + \frac{1}{2} \) in \(35.2-46a\) then (two parameters on the right hand side cancel)

\[
\left[ F\left( \frac{b + \frac{1}{2}}{2b+1}, b \right) \right]^2 = F\left( \frac{2b + \frac{1}{2}, 2b}{4b+1} \right) \quad (35.2-47)
\]

and the right hand side again matches the structure on the left. The corresponding function can be identified (see [57, p.190]) as \( G_b(z) := F\left( \frac{b + \frac{1}{2}, b}{2b+1} \right) \) where \( z = (1+\sqrt{1-z})^{-2b} \). We have \( G_{nm}(z) = [G_n(z)]^m \).

Specializing relation \(35.2-46b\) for \( b = -a \) we obtain

\[
\left[ F\left( \frac{1}{2} + a, \frac{1}{2} - a \right) \right]^2 = F\left( \frac{1}{2} + 2a, \frac{1}{2}, \frac{1}{2} - 2a \right) \quad (35.2-48)
\]

For \( a = 0, z = 1 \) (or \( z \) a sixth power) this relation is an identity between the square of a sum of squares and a sum of cubes:

\[
\left[ \sum_{n=0}^{\infty} \left[ \prod_{j=1}^{n} \frac{1}{2} + j - 1 \right] \right]^2 = \sum_{n=0}^{\infty} \left[ \prod_{j=1}^{n} \frac{1}{2} + j - 1 \right] = 1.39320392968 \ldots \quad (35.2-49)
\]

The relation can be obtained by setting \( \alpha = \beta = 1/4 \) and \( \gamma = 1/2 \) in exercise 16 in [278, p.298]. Setting \( a = 1/2 \) in exercise 28 in [278, p.301] we find that the quantity equals \( \pi/\Gamma(3/4)^4 = \Gamma(1/4)^4/(4 \pi^3) \). For the square root of the expressions \( (\sqrt{1.39320 \ldots} = 1.180340 \ldots) \) we have \([116, p.34]\):

\[
F\left( \frac{1}{2}, \frac{1}{2}, 1 \right) = \left[ \sum_{n=-\infty}^{\infty} e^{-n^2 \pi} \right]^2 = 1.180340599016 \ldots \quad (35.2-50)
\]

We note that relation \(35.2-46a\) on the preceding page can be obtained as the special case \( c = a + b \) of the following relation given in [278, ex.16, p.298]:

\[
F\left( a, b, \frac{c}{c + \frac{1}{2}} \right) = \sum_{k=0}^{\infty} A_k \frac{\Gamma(c+k)}{(c+\frac{1}{2})^k} z^k \quad (35.2-51a)
\]

where the \( A_k \) are defined by

\[
(1 - z)^{a+b-c} F\left( \frac{2a, 2b}{2c} \right) = \sum_{k=0}^{\infty} A_k z^k \quad (35.2-51b)
\]

The following relations are given in [13, p.184]:

\[
F\left( a, b, \frac{a+b}{2} \right) = F\left( a, b, \frac{a+b+\frac{1}{2}}{2} \right) = F\left( \frac{a+b}{2}, a+b-1, \frac{a+b+1}{2} \right) \quad (35.2-52a)
\]

\[
F\left( a, b, \frac{a+b}{2} \right) = F\left( a, b-\frac{1}{2}, \frac{a+b-\frac{1}{2}}{2} \right) = F\left( \frac{a+b}{2}, a+b-1, \frac{a+b+1}{2} \right) \quad (35.2-52b)
\]

### 35.2.4.6 The Kummer transformation

The Kummer transformation connects two hypergeometric functions of type 1\( F_1 \):

\[
\exp(z) F\left( a, b, \frac{c}{c+\frac{1}{2}} \right) = F\left( a+b, \frac{c}{c+\frac{1}{2}} \right) \quad (35.2-53)
\]

The relation is not valid if both \( a \) and \( b \) are negative integers. In that case one obtains the Padé approximants of \( \exp(z) \), see relation 31.2-17 on page 635.
Chapter 35: Cyclotomic polynomials, Hypergeometric functions, and continued fractions

A transformation from \(_1F_1\) to \(_2F_3\) is given by

\[
F\left(\frac{a}{b} \mid z\right) F\left(\frac{a}{b} \mid -z\right) = F\left(\frac{a}{b}, \frac{b - a}{b}, \frac{1}{2}(b + 1) \mid \frac{z^2}{4}\right) \quad (35.2-54)
\]

Setting \(b = 2a\) and using 35.2-53 gives

\[
\left[F\left(\frac{a}{2a} \mid z\right) \right]^2 = \exp(z) F\left(\frac{a}{a + \frac{1}{2}}, \frac{1}{2}a \mid \frac{z^2}{4}\right) \quad (35.2-55)
\]

The following transformation connects functions \(_0F_1\) and \(_2F_3\):

\[
F\left(\frac{a}{z} \mid z\right) F\left(\frac{b}{z} \mid z\right) = F\left(\frac{\frac{1}{2}(a + b)}{a, b, a + b - 1} \mid 4z\right) \quad (35.2-56)
\]

The relation is given in [115, vol.1, p.186], where also the following transformation is given:

\[
F\left(\frac{a}{z} \mid z\right) F\left(\frac{a}{-z} \mid z\right) = F\left(\frac{1}{2}a, \frac{1}{2}a + 1 \mid -\frac{z^2}{4}\right) \quad (35.2-57)
\]

Setting \(b = a\) in relation 35.2-56 gives (cancellation of parameters on the right hand side)

\[
\left[F\left(\frac{a}{z} \mid z\right) \right]^2 = F\left(\frac{a - \frac{1}{2}}{2a - 1} \mid 4z\right) \quad (35.2-58)
\]

From relations 35.2-55 and 35.2-58 one can obtain

\[
\exp(z) F\left(\frac{a}{z} \mid z^2\right) = F\left(\frac{a - \frac{1}{2}}{2a - 1} \mid 2z\right) \quad (35.2-59)
\]

The following relations can be derived from the preceding ones:

\[
F\left(\frac{a - \frac{1}{2}}{a, 2a - 1} \mid z\right) F\left(\frac{b - \frac{1}{2}}{b, 2b - 1} \mid z\right) = \left[F\left(\frac{\frac{1}{2}(a + b)}{a, b, a + b - 1} \mid z\right) \right]^2 \quad (35.2-60)
\]

\[
F\left(\frac{a - \frac{1}{2}}{a, 2a - 1} \mid z\right) F\left(\frac{a - \frac{1}{2}}{a, 2a - 1} \mid -z\right) = \left[F\left(\frac{1}{2}a, \frac{1}{2}a + 1 \mid -\frac{z^2}{64}\right) \right]^2 \quad (35.2-61)
\]

\[
F\left(\frac{a}{2a} \mid z\right) F\left(\frac{a}{2a} \mid -z\right) = \left[F\left(\frac{a + \frac{1}{2}}{a + \frac{1}{2}} \mid \frac{z^2}{16}\right) \right]^2 \quad (35.2-62a)
\]

\[
F\left(\frac{a}{2a + 1} \mid z\right) F\left(\frac{a}{2a + 1} \mid -z\right) = F\left(\frac{a}{2a + 1, a + \frac{1}{2}} \mid \frac{z^2}{4}\right) \quad (35.2-62b)
\]

\[
F\left(\frac{a}{z} \mid z\right) F\left(\frac{1}{z} \mid z\right) = \frac{1}{2} \left[1 + F\left(\frac{1}{a, 1 - a} \mid 4z\right) \right] \quad (35.2-63a)
\]

\[
F\left(\frac{a}{z} \mid z\right) F\left(\frac{a}{z} \mid z\right) = F\left(\frac{a + \frac{1}{2}}{a + 1, 2a} \mid 4z\right) \quad (35.2-63b)
\]
35.2.5 Examples: elementary functions

The ‘well-known’ functions like exp, log and sin are expressed as hypergeometric functions. In some cases a transformation is applied to give an alternative series.

35.2.5.1 Powers, roots, and binomial series

\[
\frac{1}{(1-z)^a} = F\left(\frac{a}{z}\right) = \sum_{k=0}^{\infty} \binom{a+k-1}{k} z^k = F\left(-\frac{a}{1-z}\right) \quad \text{(35.2-64a)}
\]

\[
(1+z)^a = F\left(-\frac{a}{1-z}\right) = \sum_{k=0}^{\infty} \binom{a}{k} z^k = F\left(\frac{a}{1+1}\right) \quad \text{(35.2-64b)}
\]

An important special case of relation 35.2-64a is

\[
\frac{1}{1-z} = F\left(\frac{1}{z}\right) = \sum_{k=0}^{\infty} z^k = F\left(\frac{1}{2}, \frac{1}{2} \Bigg | 4z(1-z)\right) \quad \text{where } z < \frac{1}{2} \quad \text{(35.2-65)}
\]

The last equality is obtained by setting \(a = 0\) in relation 35.2-44a on page 702.

\[
F\left(-s, s+1 \Bigg | \frac{1}{2}\right) = (1-2z)(1-z)^{s-1} \quad \text{(35.2-66)}
\]

\[
F\left(\frac{n}{2}, \frac{n+1}{2} \Bigg | \frac{1}{2}\right) = \frac{(1-\sqrt{z})^{-n} + (1+\sqrt{z})^{-n}}{2} \quad \text{(35.2-67a)}
\]

\[
F\left(\frac{n}{2}, \frac{n+1}{2} \Bigg | \frac{1}{n+1}\right) = \left(\frac{1+\sqrt{1-z}}{2}\right)^{-n} \quad \text{(35.2-67b)}
\]

\[
F\left(\frac{n}{2}, \frac{n+1}{n+1} \Bigg | \frac{1}{z}\right) = \frac{1}{\sqrt{1-z}} \left(\frac{2}{1+\sqrt{1-z}}\right)^{n-1} \quad \text{(35.2-67c)}
\]

\[
F\left(\frac{n+1}{2}, \frac{n+2}{2} \Bigg | \frac{2}{z}\right) = \frac{(1-\sqrt{z})^{-n} - (1+\sqrt{z})^{-n}}{2n\sqrt{z}} \quad \text{if } n \neq 0 \quad \text{(35.2-68a)}
\]

\[
= \frac{1}{2\sqrt{z}} \log \left(\frac{1+\sqrt{z}}{1-\sqrt{z}}\right) \quad \text{if } n = 0 \quad \text{(35.2-68b)}
\]

\[
\frac{(1+z)^n - (1-z)^n}{(1+z)^n + (1-z)^n} = nz F\left(\frac{1}{2} - \frac{n}{2}, \frac{1}{2} \Bigg | \frac{z^2}{z^2 - 1}\right) / F\left(\frac{1}{2} - \frac{n}{2}, \frac{1}{2} \Bigg | \frac{z^2}{z^2 - 1}\right) \quad \text{(35.2-69a)}
\]

\[
= nz F\left(\frac{1}{2} - \frac{n}{2}, \frac{1}{2} \Bigg | \frac{z^2}{z^2 - 1}\right) / F\left(\frac{1}{2} - \frac{n}{2}, \frac{1}{2} + \frac{n}{2} \Bigg | \frac{z^2}{z^2 - 1}\right) \quad \text{(35.2-69b)}
\]
35.2.5.2 Chebyshev polynomials

The Chebyshev polynomials are treated in section 34.2 on page 677, we have

\[ T_n(1-2z) = F\left(\frac{n}{2}, -\frac{n}{2} \Bigr| z\right) \quad (35.2-70a) \]
\[ T_n(z) = F\left(\frac{n}{2}, -\frac{n}{2} \Bigr| \frac{1-z}{2}\right) \quad (35.2-70b) \]
\[ U_n(z) = (n+1) F\left(-n, \frac{n+2}{2} \Bigr| \frac{1-z}{2}\right) \quad (35.2-70c) \]

Using relation 34.2-14 on page 680 (as \( T_n(T_{1/n}(z)) = z = \text{id}(z) \)) we find that

\[ F\left(\frac{n}{2}, -\frac{n}{2} \Bigr| \frac{1-z}{2}\right) \quad (35.2-71) \]

near \( z = 1 \) (here \( F^{-1} \) denotes the inverse function).

35.2.5.3 Hermite polynomials

The Hermite polynomials \( H_n(z) \) can be defined by the recurrence

\[ H_{n+1}(z) = 2z H_n(z) - 2n H_{n-1}(x) \quad (35.2-72) \]

where \( H_0(z) = 1 \) and \( H_1(z) = 2z \). The first few are

\[
\begin{align*}
H_0 &= 1 \\
H_1 &= 2z \\
H_2 &= 4z^2 - 2 \\
H_3 &= 8z^3 - 12z \\
H_4 &= 16z^4 - 48z^2 + 12 \\
H_5 &= 32z^5 - 160z^3 + 120z \\
H_6 &= 64z^6 - 480z^4 + 720z^2 - 120 \\
H_7 &= 128z^7 - 1344z^5 + 3360z^3 - 1680z \\
H_8 &= 256z^8 - 3584z^6 + 13440z^4 - 13440z^2 + 1680 \\
H_9 &= 512z^9 - 9216z^7 + 48384z^5 - 80640z^3 + 30240 \\
H_{10} &= 1024z^{10} - 23040z^8 + 161280z^6 - 403200z^4 + 302400z^2 - 30240
\end{align*}
\]

For nonnegative integer \( n \) we have

\[ H_n(z) = (2z)^n F\left(-\frac{1}{2}, -\frac{1}{2}(n-1) \Bigr| \frac{1}{z^2}\right) \quad (35.2-74) \]
35.2.5.4 Stirling numbers of the first kind

A generating hypergeometric function for the (unsigned) Stirling numbers of the first kind \( s(n, m) \) (see section 10.15 on page 283) is given by

\[
F\left(1, e \left| \frac{z}{2} \right. \right) = \sum_{n=0}^{\infty} e^{m} \frac{z^{n}}{n!} = \sum_{n=0}^{\infty} \sum_{m=1}^{n} e^{m} s(n, m) \frac{z^{n}}{n!} = 1 + e z + (e + e^{2}) z^{2} + (2e + 3e^{2} + e^{3}) z^{3} + (6e + 11e^{2} + 6e^{3} + e^{4}) z^{4} + (24e + 50e^{2} + 35e^{3} + 10e^{4} + e^{5}) z^{5} + (120e + 274e^{2} + 225e^{3} + 85e^{4} + 15e^{5} + e^{6}) z^{6} + \ldots
\]

35.2.5.5 Logarithm and exponential function

\[
\log(1 + z) = z F\left(1, \frac{1}{2} \left| \frac{z}{2} \right. \right) = \sum_{k=0}^{\infty} \frac{(-1)^{k} z^{k+1}}{k+1}
\]

\[
\log\left(\frac{1 + z}{1 - z} \right) = 2z F\left(\frac{1}{2} \left| \frac{z}{2} \right. \right) = \log\left(1 + \frac{2}{1 - z} \right)
\]

For large arguments \( z \) the following relation can be useful:

\[
\log(1 + z) = -\log\left(1 - \frac{z}{1 + z} \right) = \frac{z}{1 + z} F\left(1, \frac{1}{2} \left| \frac{z}{1 + z} \right. \right)
\]

\[
\exp(z) = F\left(\left| \frac{z}{2} \right. \right) = \sum_{k=0}^{\infty} \frac{z^{k}}{k!}
\]

35.2.5.6 Bessel functions and error function

The Bessel functions \( J_n \) of the first kind, and the modified Bessel functions \( I_n \) (as given in [1]):

\[
J_n(z) = \frac{(z/2)^{n}}{n!} F\left(n + 1 \left| \frac{z^{2}}{4} \right. \right)
\]

\[
I_n(z) = \frac{(z/2)^{n}}{n!} F\left(n + 1 \left| \frac{z^{2}}{4} \right. \right)
\]

\[
F\left(\left| \frac{z}{2} \right. \right) = \frac{(n - 1)!}{z^{(n-1)/2}} I_{n-1}(2\sqrt{z})
\]

\[
F\left(1 \left| \frac{z}{2} \right. \right) = I_0(2\sqrt{z}) = \sum_{k=0}^{\infty} \frac{z^{k}}{k!^{2}}
\]
Chapter 35: Cyclotomic polynomials, Hypergeometric functions, and continued fractions

Error function (the Kummer transformation, relation 35.2-53 on page 703 gives relation 35.2-80b):

\[
\sqrt{\frac{\pi}{2}} \text{erf}(z) := \int_{0}^{z} e^{-t^2} dt = z F\left(\frac{1}{2}, \frac{1}{2} \mid -z^2\right)
\]

\[
= z e^{-z^2} F\left(\frac{1}{2}, \frac{1}{2} \mid z^2\right) = z e^{-z^2} \sum_{k=0}^{\infty} \frac{(2z^2)^k}{1 \cdot 3 \cdot 5 \ldots (2k+1)}
\]

\[
= \frac{1}{2} z \left[ F\left(-\frac{1}{2}, \frac{1}{2} \mid -z^2\right) - e^{-z^2}\right]
\]

35.2.5.7 Trigonometric and hyperbolic functions

Series for sine and cosine:

\[
\sin(z) = z F\left(\frac{1}{2}, \frac{1}{2} \mid -\frac{z^2}{4}\right) = \sum_{k=0}^{\infty} (-1)^k \frac{z^{2k+1}}{(2k+1)!}
\]

\[
\sinh(z) = z F\left(\frac{1}{2}, \frac{1}{2} \mid \frac{z^2}{4}\right) = \sum_{k=0}^{\infty} \frac{z^{2k+1}}{(2k+1)!}
\]

Applying the transformation 35.2-58 on page 704 to relation 35.2-81a gives

\[
|\sin(z)|^2 = z^2 F\left(\frac{1}{2}, \frac{1}{2} \mid -z^2\right)
\]

\[
\cos(z) = F\left(\frac{1}{2}, \frac{1}{2} \mid -\frac{z^2}{4}\right) = \sum_{k=0}^{\infty} (-1)^k \frac{z^{2k}}{(2k)!}
\]

\[
\cosh(z) = F\left(\frac{1}{2}, \frac{1}{2} \mid \frac{z^2}{4}\right) = \sum_{k=0}^{\infty} \frac{z^{2k}}{(2k)!}
\]

\[
\exp(z) \frac{\sinh(z)}{z} = F\left(\frac{1}{2}, \frac{1}{2} \mid 2z\right)
\]

\[
\exp(-iz) \frac{\sin(z)}{z} = F\left(\frac{1}{2}, \frac{1}{2} \mid -2iz\right)
\]

Further expressions for the sine and cosine are

\[
\frac{\sin(az)}{a \sin(z)} = F\left(\frac{1+a}{2}, \frac{1-a}{2} \mid |\sin(z)|^2\right)
\]

\[
\frac{\cos(az)}{\cos(z)} = F\left(\frac{1+a}{2}, \frac{1-a}{2} \mid |\sin(z)|^2\right)
\]

\[
\frac{\sin(az)}{a \sin(z) \cos(z)} = F\left(\frac{1+a}{2}, \frac{1-a}{2} \mid |\sin(z)|^2\right)
\]

[fxtbook draft of 2008-August-17]
Relations for the hyperbolic sine and cosine are obtained by replacing \( \sin \mapsto \sinh \), \( \cos \mapsto \cosh \), and negating the sign of the argument of the hypergeometric function. For example, from relation 35.2-85b one obtains

\[
\cosh(az) = F \left( \frac{1 + a^2}{2}, \frac{1 - a^2}{2} \mid -\sinh(z)^2 \right)
\]

(35.2-85e)

### 35.2.5.8 Inverse trigonometric and hyperbolic functions

Series for the inverse tangent and cotangent:

\[
\arctan(z) = -i \frac{1}{2} \log \frac{1 + iz}{1 - iz} \quad \text{or} \quad \arctan(z) = \frac{1}{\sqrt{1 + z^2}} F \left( \frac{1}{2}, 1 \mid \frac{z^2}{1 + z^2} \right) = \arccos \frac{1}{\sqrt{1 + z^2}}
\]

(35.2-86c)

By Pfaff’s reflection law (relation 35.2-28b) on can obtain

\[
\arctan(z) = \frac{z}{1 + z^2} F \left( \frac{1}{2}, 1 \mid \frac{z^2}{1 + z^2} \right)
\]

by 35.2-28b (35.2-86d)

\[
\arctanh(z) = \frac{1}{2} \log \frac{1 + z}{1 - z}
\]

(35.2-87a)

\[
\arccoth(z) = \frac{1}{2} \log \frac{z + 1}{z - 1} = \sum_{k=0}^{\infty} \frac{1}{(2k + 1) z^{2k+1}}
\]

(35.2-87c)

\[
\log(z) = 2 \arctan \frac{z - 1}{z + 1} = 2 \arccoth \frac{z + 1}{z - 1}
\]

(35.2-87d)

\[
\arccot(z) = \arctan \left( \frac{1}{z} \right) = -i \frac{1}{2} \log \frac{z + i}{z - i}
\]

(35.2-88a)

\[
\arccot(z) = \frac{1}{z} F \left( \frac{1}{2}, 1 \mid \frac{1}{1 + z^2} \right) = \sum_{k=0}^{\infty} \frac{(-1)^k}{z^{2k+1}}
\]

(35.2-88b)

\[
= \frac{1}{\sqrt{1 + z^2}} F \left( \frac{1}{2}, 1 \mid \frac{1}{1 + z^2} \right) = \arcsin \frac{1}{\sqrt{1 + z^2}}
\]

(35.2-88c)

\[
= \frac{z}{1 + z^2} F \left( \frac{1}{2}, 1 \mid \frac{1}{1 + z^2} \right)
\]

(35.2-88d)

Applying Clausen’s product formula (relation 35.2-46a on page 702) gives

\[
[\arctan(z)]^2 = \frac{z^2}{1 + z^2} F \left( 1, 1, 1 \mid \frac{z^2}{1 + z^2} \right)
\]

(35.2-89a)

\[
[\arccot(z)]^2 = \frac{1}{1 + z^2} F \left( 1, 1, 1 \mid \frac{1}{1 + z^2} \right)
\]

(35.2-89b)
Series for the inverse sine and cosine are

\[
\arcsin(z) = z F \left( \frac{1}{2}, \frac{1}{2} \bigg| z^2 \right) = \arctan \frac{z}{\sqrt{1-z^2}} \quad (35.2-90a)
\]

\[
= z F \left( \frac{3}{2} \bigg| \frac{1 - \sqrt{1-z^2}}{2} \right) \quad \text{by 35.2-33b} \quad (35.2-90b)
\]

\[
= z \sqrt{1-z^2} F \left( \frac{1}{2}, \frac{1}{2} \bigg| z^2 \right) \quad (35.2-90c)
\]

The two latter relations suggest the following argument reduction applicable for the inverse sine (and tangent). Let \( G(z) = (1 - \sqrt{1-z})/2 \), then

\[
F \left( \frac{1}{2}, \frac{1}{2} \bigg| G(z) \right) = \sqrt{1-z} F \left( \frac{1}{2}, \frac{1}{2} \bigg| G(G(z)) \right) = \ldots
\]

\[
F \left( \frac{1}{2}, \frac{1}{2} \bigg| G(z) \right) = \left[ \prod_{k=0}^{\infty} (1-z_k) \right]^{-1/2} \quad \text{where } z_0 = z, \quad z_{k+1} = G(z_k) \quad (35.2-91c)
\]

Clausen’s product formula (relation 35.2-46a on page 702) can be applied to obtain

\[
[\arcsin(z)]^2 = z^2 F \left( \frac{1}{3}, \frac{1}{3}, \frac{1}{2} \bigg| z^2 \right) \quad (35.2-92)
\]

\[
\arccos(z) = \frac{\pi}{2} - \arcsin(z) = \arccot \frac{z}{\sqrt{1-z^2}} \quad (35.2-93)
\]

\[
\arcsinh(z) = \log(z + \sqrt{1+z^2}) = z F \left( \frac{1}{2}, \frac{1}{2} \bigg| -z^2 \right) \quad (35.2-94a)
\]

\[
= \frac{z}{\sqrt{1+z^2}} F \left( \frac{1}{2}, \frac{1}{2} \bigg| \frac{z^2}{1+z^2} \right) \quad \text{by 35.2-28b} \quad (35.2-94b)
\]

\[
= z F \left( \frac{3}{2} \bigg| \frac{1 - \sqrt{1+z^2}}{2} \right) \quad \text{by 35.2-33b} \quad (35.2-94c)
\]

### 35.2.6 The function \( x^x \)

Boldly setting \( a = 1 + z \) in \((1+z)^a = F \left( -a \bigg| -z \right)\) (relation 35.2-64b on page 705) gives

\[
(1+z)^{(1+z)} = F \left( -1 - z \bigg| -z \right) = \exp \left[ (1+z) \log(1+z) \right] \quad (35.2-95a)
\]

\[
= 1 + (z+1)z + \frac{(z+2)}{2} z^2 + \frac{(z+3)}{3} z^3 + \frac{(z+4)}{4} z^4 + \frac{(3+5)}{6} z^5 + \frac{(4+59)}{120} z^6 - \frac{1}{2520} z^7 + \frac{59}{5040} z^8 - \frac{71}{5040} z^9 + \ldots \quad (35.2-95c)
\]
This somewhat surprising expression allows the computation of \( x^z \) without computing \( \exp() \) or \( \log() \). The series converges for real \( z > 0 \) so we can compute \( x^z \) (where \( x = 1 + z \)) for real \( x > +1 \) as

\[
x^z = F \left( -x \left| -x + 1 \right. \right) = 1 + \frac{x - 0}{1} (x - 1) \left[ 1 + \frac{x - 1}{2} (x - 1) \left[ 1 + \frac{x - 2}{3} (x - 1) \left[ 1 + \ldots \right] \right] \right]
\]  

We denote the series obtained by truncating after the \( n \)-th term of the hypergeometric function by \( g_n(x) \). For example, with \( n = 2 \) and \( n = 4 \) we obtain:

\[
g_2(x) = \frac{1}{2} x^4 - \frac{3}{2} x^3 + \frac{5}{2} x^2 - \frac{3}{2} x + 1
\]

\[
g_4(x) = \frac{1}{24} x^8 - \frac{5}{12} x^7 + \frac{15}{8} x^6 - \frac{19}{4} x^5 + \frac{61}{8} x^4 - \frac{31}{4} x^3 + \frac{131}{24} x^2 - \frac{25}{12} x + 1
\]

For \( 0 < x < n \) the interpolating polynomials \( i_n(x) \) give an estimate that is consistently worse than \( g_n(x) \) for non-integer values of \( x \). The same is true even for the polynomials \( i_{2n}(x) \) that interpolate \( k^k \) for \( 0 \leq k \leq n \) (setting \( 0^0 := 1 \) for our purposes).

The polynomials actually give acceptable estimates for \( x^z \) also for non-integer \( x \), especially for \( x \) near 1. The (unique) degree-\( n \) polynomials \( i_n(x) \) that are obtained by interpolating the values \( k^k \) have much bigger coefficients and give values far away from \( x^z \) for non-integer arguments \( x \).

For \( 0 < x < n \) the interpolating polynomials \( i_n(x) \) give an estimate that is consistently worse than \( g_n(x) \) for non-integer values of \( x \). The same is true even for the polynomials \( i_{2n}(x) \) that interpolate \( k^k \) for \( 0 \leq k \leq 2n \) (so that \( \deg(i_{2n}) = \deg(g_{2n}) = 2n \)). In fact, the \( i_{2n}(x) \) approximate consistently worse than \( i_n(x) \) for non-integer \( x \).

Finally, the Padé approximants \( p_{n,n}(x) \) for \( g_n(x) \) give estimates that are worse than with both \( i_n(x) \) or \( g_n(x) \). Further, \( g_n(x) \neq x^z \) even for integer \( x \) and the \( p_{n,n}(x) \) have a pole on the real axis near \( x = 1 \). That is, we found a surprisingly good and compact polynomial approximation for the function \( x^z \).

The sequence of the \( n \)-th derivatives of \( (1 + z)^{1+z} \) at \( z = 0 \) is entry [A005727] of [245]:

\[
\text{Vec}(\text{serlaplace}(\exp((1+z)*\log(1+z))))
\]

\[
[1, 1, 2, 3, 8, 10, 54, -42, 944, -5112, 47160, -419760, 4297512, \ldots ]
\]

Many other expressions for the function \( x^z \) can be given, we note just one: set \( s = 2 + z \) in relation 35.2-66 on page 705 to obtain

\[
(1 + z)^{(1+z)} = \frac{1}{z + 2} x^z F \left( -z - 2, z + 3 \left| z + 2 \right. \right)
\]  

### 35.2.7 Elliptic \( K \) and \( E \)

In order to avoid the factor \( \frac{z}{2} \) we let \( \tilde{K} := \frac{2K}{\pi} \), \( \tilde{E} := \frac{2E}{\pi} \), then

\[
\tilde{K}(k) = F \left( \frac{1}{2}, \frac{1}{2} \left| k^2 \right. \right)
\]

\[
\tilde{E}(k) = F \left( -\frac{1}{2}, \frac{1}{2} \left| k^2 \right. \right)
\]

See section 30.2 on page 605 for explicit expansions. We further set

\[
\tilde{N}(k) = F \left( -\frac{1}{2}, -\frac{1}{2} \left| k^2 \right. \right)
\]

\[
= 1 + \frac{1}{4} k^2 + \frac{1}{64} k^4 + \frac{1}{256} k^6 + \frac{25}{16384} k^8 + \frac{49}{65536} k^{10} + \frac{441}{1048576} k^{12} + \ldots
\]  

[fxbook draft of 2008-August-17]
A special value is $N(1) = 4/\pi$. Most of the following relations can be written in several ways by one of the identities

$$k' = \sqrt{1-k^2}$$  
$$-\frac{k^2}{1-k^2} = -\left(\frac{k}{\sqrt{k'}}\right)^2 = -\frac{1-k'^2}{k'^2}$$  
$$-\left(\frac{k}{1-\sqrt{1-k^2}}\right)^2 = \frac{1+k'}{1-k'} = -\frac{1-k'^2}{(1-k')^2} = -\frac{(1+k')^2}{1-k'^2}$$  
$$\frac{2}{1+k'} = 1 + \frac{1-k'}{1+k'}$$  
$$\frac{4k}{(1+k)^2} = 1 - \left(\frac{1-k}{1+k}\right)^2$$  
$$\frac{-4k^2}{(1-k')^2} = -\frac{2k}{k'^2} = 1 - \left(\frac{1+k'}{1-k'^2}\right)^2$$

### 35.2.7.1 Relations for $K$

$$\tilde{K}(k) = F\left(\frac{1}{2}, \frac{1}{2} \mid k^2\right)$$

$$= \frac{1}{k'} F\left(\frac{1}{2}, \frac{1}{2} \mid -\left(\frac{k}{k'}\right)^2\right) \text{ by } 35.2-101$$

$$\hat{K}(k) = \frac{1}{1-k'} F\left(\frac{1}{2}, \frac{1}{2} \mid \frac{1-k'}{1-k'}\right)$$

From the product form (relation 30.2-8 on page 606) we obtain

$$\tilde{K}(k) = \frac{2}{1+k'} F\left(\frac{1}{2}, \frac{1}{2} \mid \left(\frac{1-k'}{1+k'}\right)^2\right)$$  

The relation can be written as

$$\tilde{K}(k) = (1+z(k)) \tilde{K}(z(k)) \text{ where } z(k) := \frac{1-k'}{1+k'}$$

Relation 35.2-42 on page 702 with $a = \frac{1}{2}$ gives

$$\tilde{K}(k) = \frac{1}{1+k} \tilde{K}\left(\frac{2\sqrt{k}}{1+k}\right)$$

$$\hat{K}(k) = \frac{1}{\sqrt{1-k^2}} F\left(\frac{3}{4}, \frac{3}{4} \mid -\frac{4k^2}{(1-k^2)^2}\right) = \frac{1}{k'} F\left(\frac{1}{4}, \frac{3}{4} \mid 1 - \left(\frac{1+k'}{1-k'}\right)^2\right)$$

Euler’s transform on 35.2-106a gives:

$$\tilde{K}(k) = \frac{1}{k'} \frac{1+k^2}{1-k^2} F\left(\frac{3}{4}, \frac{3}{4} \mid -\left(\frac{2k}{k'^2}\right)^2\right)$$

$$\tilde{K}(k) = F\left(\frac{1}{4}, \frac{1}{4} \mid (2k')^2\right) \text{ by } 35.2-33a$$
35.2.7.2 Relations for $E$

\[
\begin{align*}
\tilde{E}(k) &= F\left(\frac{-1}{2}, \frac{1}{2} \mid k^2\right) \\
\tilde{E}(k) &= k' F\left(\frac{-1}{2}, \frac{1}{2} \mid -\left(\frac{k}{k'}\right)^2\right)
\end{align*}
\]  
(35.2-109)  
(35.2-110)

The following relation resembles relation 35.2-104a:

\[
\tilde{E}(k) = \frac{1 + \sqrt{1 - k^2}}{2} F\left(-\frac{1}{2}, -\frac{1}{2} \mid \frac{1 - \sqrt{1 - k^2}}{1 + \sqrt{1 - k^2}}\right)
\]

by 35.2-28b

\[\text{by 35.2-111a}\]

\[
\tilde{E}(k) = \frac{1 + k'}{2} F\left(-\frac{1}{2}, -\frac{1}{2} \mid \frac{1 - k'}{1 + k'}\right)
\]

by 35.2-111b

\[\text{by 35.2-111b}\]

\[
\tilde{E}(k) = (1 + z(k))^{-1} \tilde{N}(z(k)) \quad \text{where} \quad z(k) := \frac{1 - k'}{1 + k'}
\]

(35.2-111c)

35.2.7.3 Relations for $N$

\[
\tilde{N}(k) = \sqrt{1 - k^2} F\left(-\frac{1}{4}, \frac{3}{4} \mid -\frac{4 k^2}{(1 - k^2)^2}\right) \quad \text{by 35.2-42c}
\]

(35.2-112a)

\[
\tilde{N}(k) = k' F\left(-\frac{1}{4}, \frac{3}{4} \mid -\frac{2 k^2}{k'^2}\right)
\]

(35.2-112b)

\[
\tilde{N}(k) = \sqrt{1 + k^2} F\left(-\frac{1}{4}, \frac{3}{4} \mid \frac{2 k^2}{k'^2 + 1}\right) \quad \text{by 35.2-28b}
\]

(35.2-112c)

Relation 35.2-25 on page 699 with $a = b = -1/2$ and $c = 1$ gives

\[
F\left(-\frac{1}{2}, -\frac{1}{2} \mid z\right) = \frac{1}{2} F\left(-\frac{3}{2}, -\frac{1}{2} \mid z\right) + \frac{1}{2} F\left(-\frac{1}{2}, -\frac{1}{2} \mid z\right)
\]

(35.2-113)

Applying 35.2-28c on page 700 to the second function ($F\left(-\frac{3}{2}, -\frac{1}{2} \mid z\right) = (1 - z) F\left(-\frac{1}{2}, -\frac{1}{2} \mid \frac{z}{1 - z}\right)$) and rearranging gives

\[
2 \tilde{E}(k) - \tilde{K}(k) = k' \tilde{N}\left(\frac{k}{k'}\right)
\]

(35.2-114)

Applying the transformation 35.2-42f on page 702 to the defining relation gives the key to fast computation of the function $\tilde{N}(k)$:

\[
\tilde{N}(k) = (1 + k) \tilde{E}\left(\frac{2\sqrt{k}}{1 + k}\right)
\]

(35.2-115)

The relation

\[
2 \tilde{E}(k) - k'^2 \tilde{K}(k) = \tilde{N}(k)
\]

(35.2-116)

can be used to rewrite Legendre’s relation (equation 30.2-17b on page 607) as either

\[
\frac{4}{\pi} = \tilde{N} \tilde{K}' + \tilde{K} \tilde{N}' - \tilde{K} \tilde{K}'
\]

(35.2-117)

or, by setting $N := \pi/2 \tilde{N}$,

\[
\pi = N K' + K N' - K K'
\]

(35.2-118)

[fatbook draft of 2008-August-17]
35.3 Continued fractions

A continued fraction is an expression of the form:

\[ K(a, b) = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \ldots}}} \]  

(35.3-1)

Continued fractions are sometimes expressed in the following form:

\[ K(a, b) = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \ldots}}} \]  

(35.3-2)

The \( a_k \) and \( b_k \) are called the \( k \)-th partial numerators and denominators.

For \( k > 0 \) let \( \frac{P_k}{Q_k} \) be the value of the above fraction if \( b_{k+1} \) is set to zero (that is, the continued fraction terminates at index \( k \)). The ratio is called the \( k \)-th convergent of the continued fraction:

\[ \frac{P_k}{Q_k} = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \ldots}}} = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \ldots}}} = \ldots \]  

(35.3-3)

We note that multiplication of \( a_i, b_i, b_{i+1} \) by some nonzero value does not change the value of the continued fraction. Thereby

\[ a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \ldots}}} = a_0 + \frac{c_1 b_1}{c_1 a_1 + \frac{c_2 b_2}{c_2 a_2 + \frac{c_3 b_3}{c_3 a_3 + \ldots}}} \]  

(35.3-4)

where all \( c_i \) are arbitrary nonzero constants.

35.3.1 Simple continued fractions

Continued fractions where all \( b_k \) are equal to one (and all the \( a_k \) are positive) are called simple continued fractions. Rational numbers have terminating continued fractions. Note that the expression of a rational number as simple continued fraction is not unique:

\[ [a_0, \ldots, a_n-1, a_n] = [a_0, \ldots, a_n-1, a_n-1, 1] \quad \text{if} \quad a_n > 1 \]  

(35.3-5a)

\[ [a_0, \ldots, a_n-1, 1] = [a_0, \ldots, a_n-1+1] \quad \text{if} \quad a_n = 1 \]  

(35.3-5b)

Solutions of the quadratic equation \( \alpha x^2 + \beta x + \gamma = 0 \) that are not rational (\( \Delta := \beta^2 - 4\alpha \gamma \) not a square) have simple continued fractions that are eventually periodic. For example:

\[ \text{contfrac(sqrt(5))} \]  

\[ [2, 4, 4, 4, 4, 4, \ldots] \]

\[ \text{contfrac(2+sqrt(3))} \]  

\[ [3, 1, 2, 1, 2, 1, 2, \ldots] \]

\[ \text{contfrac(sqrt(19))} \]  

\[ [4, 2, 1, 3, 2, 8, 2, 2, 1, 3, 1, 2, 8, 2, 2, 1, 3, 1, 2, 8, \ldots] \]

For the \( k \)-th convergent \( \frac{P_k}{Q_k} \) (in lowest terms) of the simple continued fraction expansion of some number \( x \) then the convergent is a best approximation in the following sense: if \( p/q \) is any better rational approximation to \( x \) (that is, \( \left| \frac{p}{q} - x \right| < \left| \frac{P_k}{Q_k} - x \right| \)), then one must have \( q > Q_k \).

For the simple continued fraction of \( x \) one has

\[ \left| x - \frac{P_n}{Q_n} \right| \leq \frac{1}{Q_n Q_{n-1}} < \frac{1}{Q_n^2} \]  

(35.3-6)

and equality can only occur with terminating continued fractions.
35.3.1.1 Computing the simple continued fraction of a number

Given a numerical quantity one can compute the sequence \( a_k \) of its simple continued fraction by the following algorithm:

1. Set \( k = 0 \), \( F_0(x) = F(x) \), and \( d = \deg(F) \).
2. Find the (unique) real positive root \( r_k \) of \( F_k(x) \), set \( a_k = \lfloor r \rfloor \). If \( k = n \) then stop.
3. Set \( G(x) = F_k(x + a_k) \), set \( F_{k+1} = -G^*(x) = -x^d G(1/x) \).
4. Set \( k = k + 1 \) and goto step 2.

Here \( n \) is the number of requested terms \( a_k \). Obviously some check has to be inserted in order to avoid possible division by zero (and indicate a terminating continued fraction as will occur for rational \( x \)). If in the process one keeps track of the exact rational values of the simple continued fraction convergents (using the recursion relations) then the algorithm can be used to produce a ‘best possible’ rational approximation where the denominator does not exceed a certain specified size. Alternatively one can stop as soon as the convergent is within some specified bound.

35.3.1.2 Continued fractions of polynomial roots

For example:

\[
\begin{align*}
F(x) &= x^3 - 2, \quad r = \text{RootOf}(z^3 - 2) = 1.25992104989487 \\
\text{contfrac}(r) &= [1, 3, 1, 5, 1, 1, 4, 1, 1, 8, 1, 14, 1, 10, 2, ...] \\
\end{align*}
\]

```
F(x) = x^3 - 2, r = 1.25992104989487 ...
F(r) = 0
F(r) = 0
```

Figure 35.3-A: Computation of the continued fraction of the positive real root of the polynomial \( z^3 - 2 \).

\[
\begin{align*}
F(x) &= x^2 - 29, \quad r = \text{RootOf}(z^2 - 29) = 5.38516480713450 \\
\text{contfrac}(r) &= [5, 2, 1, 1, 2, 10, 2, 1, 1, 2, 10, ...] \\
\end{align*}
\]

```
F(x) = x^2 - 29, r = 5.38516480713450 ...
F(r) = 0
F(r) = 0
```

Figure 35.3-B: Computation of the continued fraction of the positive real root of the polynomial \( z^2 - 29 \).
A simple demonstration is

\[ f = z^3 - 2 \]
\[ \text{ff}(y) = \text{subst}(f, z, y) \quad \text{\Comment{for solve() function}} \]

\[
\begin{align*}
\text{for } (k=1, 12, \\
\qquad \text{print1(" f=", f);}
\end{align*}
\]

\[
\begin{align*}
\text{r} &= \text{solve}(x=0.9, 1e9, \text{ff}(x)); \quad \text{\Comment{\textit{l}}azy implementation} \\
\text{ak} &= \text{floor}(\text{r}); \\
\text{g} &= \text{subst}(f, z, z+ak); \quad \text{\Comment{shifted polynomial}}
\end{align*}
\]

\[
\begin{align*}
\text{f} &= -\text{polrecip}(g); \quad \text{\Comment{negated reciprocal of g}} \quad \text{print();} \\
\end{align*}
\]

The output with \( F(x) = x^3 - 2 \) is shown in figure 35.3-A. With quadratic equations one obtains periodic continued fractions, figure 35.3-B shows the computation for \( F(x) = x^2 - 29 \). For a comparison of methods for the computation of continued fractions for algebraic numbers see [69].

35.3.2 Computation of the convergents (evaluation)

The computation of the sequence of convergents uses the recurrence

\[
\begin{align*}
P_k &= a_k P_{k-1} + b_k P_{k-2} \\
Q_k &= a_k Q_{k-1} + b_k Q_{k-2}
\end{align*}
\]

Set \( \frac{P_{-1}}{Q_{-1}} := \frac{1}{0} \) and \( \frac{P_0}{Q_0} := \frac{a_0}{1} \) to initialize. The following is a procedure that computes the sequences of values \( P_k \) and \( Q_k \) for \( k = -1 \ldots n \) for a given continued fraction:

\[
\begin{align*}
\text{procedure ratios_from_cf(a[0..n], b[0..n], n, P[-1..n], Q[-1..n])}
\end{align*}
\]

\[
\begin{align*}
\text{P[-1]} := 1 \\
\text{Q[-1]} := 0 \\
\text{P[0]} := a[0] \\
\text{Q[0]} := 1 \\
\text{for } k:=1 \text{ to } n \\
\text{\{ \\
\text{P}[k] := a[k] * P[k-1] + b[k] * P[k-2] \\
\text{Q}[k] := a[k] * Q[k-1] + b[k] * Q[k-2] \\
\text{\}}}
\end{align*}
\]

If only the last ratio is of interest, the ‘backward’ variant of the algorithm can be used. The version that computes the numerical value \( x \) from the first \( n \) terms of a simple continued fraction is:

\[
\begin{align*}
\text{function ratio_from_cf(a[0..n-1], n)}
\end{align*}
\]

\[
\begin{align*}
x := a[n-1] \\
\text{for } k:=n-2 \text{ to } 0 \text{ step } -1 \\
\text{\{ \\
\text{\quad x := 1/x + a[k] \\
\text{\}}}
\end{align*}
\]

\[
\begin{align*}
\text{return } x
\end{align*}
\]

Using rational arithmetic and a general (non-simple) continued fraction, the algorithm becomes:

\[
\begin{align*}
\text{function ratio_from_cf(a[0..n-1], b[0..n-1], n)}
\end{align*}
\]

\[
\begin{align*}
P := a[n-1] \\
Q := b[n-1] \\
\text{for } k:=n-2 \text{ to } 0 \text{ step } -1 \\
\text{\{ \\
\text{\quad \{P, Q\} := \{a[k]*P+b[k]*Q, P\} // x := b[k] / x + a[k] \\
\text{\}}}
\end{align*}
\]

\[
\begin{align*}
\text{return } P/Q
\end{align*}
\]
35.3: Continued fractions

Implementation

Converting a number to a simple continued fraction can be done with pari/gp’s builtin function `contfrac()`. The final convergent can be computed with `contfracpnqn()`:

```gp
? default(realprecision,23)
realprecision = 28 significant digits (23 digits displayed)
? Pi = 3.1415926535897932384626
? cf = contfrac(Pi)
[3, 7, 15, 1, 292, 1, 1, 1, 2, 2, 2, 2, 1, 84, 2, 1, 1, 15, 3]
? contfracpnqn
contfracpnqn(x): [p_n,p_{n-1}; q_n,q_{n-1}] corresponding to the continued fraction x.
? m = contfracpnqn(cf)
[428224593349304 139755218526789]
[136308121570117 44485467702853]
? 1.0*m[1,1]/m[2,1]
3.1415926535897932384626
```

The number of terms of the continued fraction depends on the precision used, with greater precision more terms can be computed. The computation of the \( m \)-th convergent of a continued fraction given as two vectors \( a[] \) and \( b[] \) can be implemented as (backward variant):

```gp
1 cfab2r(a,b, m=-2)=
2 { local(n, r);
3 n = length(a);
4 if ( m>-2, m = min(n, m) ); \ \ default: m=n
5 if ( m>=n, m=n-1 );
6 if ( m<0, return( 0 ) ); \ \ infinity
7 r = 0;
8 m += 1;
9 forstep (k=m, 2, -1, r = b[k]/(a[k]+r); );
10 r += a[1]; \ \ b[1] unused
11 return( r );
}
```

Alternatively, one can use the recursion relations 35.3-7a and 35.3-7b. We do not store all pairs \( P_n, Q_n \) but only return the final pair \( P_m, Q_m \):

```gp
1 cfab2pq(a,b, m=-2)=
2 {
3 local(n, p, p1, p2, q, q1, q2, i);
4 n = length(a)-1;
5 if ( m>-2, m = min(n, m) ); \ \ default: m=n
6 if ( m<0, return( [1, 0] ) ); \ \ infinity
7 p1 = 1;
8 q1 = 0;
9 p = a[1];
10 q = 1; \ \ b[1] unused
11 for (k=1, m,
12 i = k+1;
13 p2 = p1; p1 = p;
14 q2 = q1; q1 = q;
15 p = a[i]*p1 + b[i]*p2;
16 q = a[i]*q1 + b[i]*q2;
17 );
18 return( [p,q] );
}
```

We use our routines to compute the convergents of the continued fraction for \( 4/\pi \) given 1658 by Brouncker:

\[
\frac{4}{\pi} = 1 + \frac{1^2}{2 + \frac{2^2}{3 + \frac{3^2}{2 + \frac{5^2}{2 + \frac{7^2}{2 + \cdots}}}}} = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \cdots}}} \quad (35.3-8)
\]

Figure 35.3-C shows how to set up the vectors containing the \( a_k \) and \( b_k \) and check the convergents.
default(realprecision, 55); \ use enough precision
default(format, "g.11"); \ print with moderate precision
default(echo, 0);
\ cfab2pq.inc.gp \ functions cfab2pq and cfab2r
x=4.0/Pi
n=15
/*/ set up the continued fraction: */
a=vector(n, j, 2); a[1]=1;
b=vector(n, j, (2*j-3)^2);
/*/ print convergents and their error: */
{ for(k=0, n-1,
t=cfab2pq(a, b, k);
p=t[1]; q=t[2];
print1(k, " : ", p, " / ", q);
print1("\n d=", x-p/q);
print();
);
}
quit; /* ------ end of script -------- */
/* ------ start output: ------ */
15 /* =n */
a=[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] /* =a */
b=[1, 1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729] /* =b */
1.2732395447 /* = 4/Pi */
0: 1 / 1
d=0.27323954473
1: 3 / 2
d=0.22676045526
2: 15 / 13
d=0.11939339088
3: 105 / 76 /* =p3/q3 */
d=0.1038940263 /* =p3/q3-4/Pi */
4: 945 / 789
d=0.07552091356
5: 10395 / 7734
d=0.070825622061
[--snip--]
13: 213458046676875 / 163842638377950
d=0.029583998575
14: 6190283353629375 / 4964894559637425
d=0.026428906710

Figure 35.3-C: A pari/gp script demonstrating the function cfab2pq() that computes the convergents of a continued fraction (top) and its output (bottom, comments added). Here convergence is rather slow.

Fast evaluation as matrix product

For the evaluation of a continued fraction with a large number of terms rewrite relations 35.3-7a and 35.3-7b as

\[
\begin{bmatrix}
P_k & Q_k \\
P_{k-1} & Q_{k-1}
\end{bmatrix} = \prod_{j=0}^{k} \begin{bmatrix} a_j & 1 \\ b_j & 0 \end{bmatrix} \tag{35.3-9}
\]

Use the binary splitting algorithm (section 32.1 on page 641) for the computation of the matrix product \(M(k)\).

35.3.3 Miscellaneous relations for continued fractions

35.3.3.1 Determinantal expressions

The determinant formula for the numerators and denominators of successive convergents is

\[
\det \begin{bmatrix}
P_k & Q_k \\
P_{k-1} & Q_{k-1}
\end{bmatrix} = P_k Q_{k-1} - P_{k-1} Q_k = (-1)^{k-1} \prod_{j=1}^{k} b_j \tag{35.3-10}
\]
The relation is obtained by taking determinants on both sides of equation \[35.3-9\] The relation can also be written as
\[
\frac{P_k}{Q_k} - \frac{P_{k-1}}{Q_{k-1}} = \frac{(-1)^{k-1}}{Q_{k-1}} \prod_{j=1}^{k-1} b_j
\]
(35.3-11)

For simple continued fractions we have \(b_j = 1\) so the product in the numerator equals one. Further, by inserting \(P_{k-1} = (P_k - b_k P_{k-2})/a_k\) (relation \[35.3-10\]) and the equivalent expression for \(Q_{k-1}\) into relation \[35.3-11\] one obtains
\[
\det \begin{bmatrix} P_k & Q_k \\ P_{k-2} & Q_{k-2} \end{bmatrix} = P_k Q_{k-2} - P_{k-2} Q_k = (-1)^k a_k \prod_{j=1}^{k-1} b_j
\]
(35.3-12)

Equivalently,
\[
\frac{P_k}{Q_k} - \frac{P_{k-2}}{Q_{k-2}} = \frac{(-1)^k a_k \prod_{j=1}^{k-1} b_j}{Q_{k-2} Q_k}
\]
(35.3-13)

This relation tells us (provided all \(a_j\) and \(b_j\) are positive) that the sequence of even convergents is increasing and the sequence of odd convergents is decreasing. As both converge to a common limit we have \(P_e/Q_o \geq P_e/Q_o\) for all even \(e\) and odd \(o\). Equality can occur only for terminating continued fractions.

### 35.3.3.2 Subsequences of convergents

Sometimes the terms \(a_k, b_k\) of the continued fraction are given in the form “\(a_k = u(k)\) if \(k\) even, \(a_k = v(k)\) else” (and \(b_k\) equivalently). Then one might want to compute the \(x = K(a,b)\) in a ‘stride two’ manner as

\[
P_k = A_k P_{k-2} + B_k P_{k-4}
\]
(35.3-14a)

\[
Q_k = A_k Q_{k-2} + B_k Q_{k-4}
\]
(35.3-14b)

in order to regularize the involved expressions. We write the recurrence relation three times

\[
P_k = a_k P_{k-1} + b_k P_{k-2}
\]
(35.3-15a)

\[
P_{k-1} = a_{k-1} P_{k-2} + b_{k-1} P_{k-3}
\]
(35.3-15b)

\[
P_{k-2} = a_{k-2} P_{k-3} + b_{k-2} P_{k-4}
\]
(35.3-15c)

and eliminate the terms \(P_{k-1}\) and \(P_{k-3}\). This gives

\[
A_k = \frac{a_k b_{k-1} + b_k a_{k-1} a_k - a_k a_{k-1} a_{k-2}}{a_{k-2}} = \frac{a_k b_{k-1}}{a_{k-2}} + b_k + a_k a_{k-1}
\]
(35.3-16a)

\[
B_k = \frac{-a_k b_{k-1} b_{k-2}}{a_{k-2}}
\]
(35.3-16b)

The stride three version

\[
P_k = A_k P_{k-3} + B_k P_{k-6}
\]
(35.3-17a)

\[
Q_k = A_k Q_{k-3} + B_k Q_{k-6}
\]
(35.3-17b)

leads to the expressions (writing \(a_n\) for \(a_{k-n}\) to reduce line width):

\[
A_k = \frac{a_0 b_1 b_3 + b_0 a_2 b_5 + b_0 b_2 a_4 + a_0 a_1 a_2 b_3 + a_0 a_1 b_2 a_4 + a_0 b_1 a_3 a_4 + b_0 a_2 a_3 a_5 + a_0 a_1 a_2 a_3 a_4}{b_3 + a_3 a_4}
\]
(35.3-18a)

\[
B_k = \frac{b_0 b_2 b_4 + a_0 a_1 b_2 b_4}{b_3 + a_3 a_4}
\]
(35.3-18b)

When setting \(a_k = \alpha, b_k = \beta\) the expressions for \(A_k\) and \(B_k\) simplify to the coefficients in relations \[34.1-13c\] on page 673 (stride two) and \[34.1-13d\] (stride three) for recurrences.
35.3.3.3 Relation to alternating series

Using relation [35.3-11] it is possible to rewrite a continued fraction \( x = K(a, b) \) with positive \( a_k, b_k \) as an alternating series

\[
x = a_0 + \sum_{k=1}^{\infty} (-1)^{k+1} s_k
\]

\[
= a_0 + \frac{b_1}{Q_0 Q_1} - \frac{b_1 b_2}{Q_1 Q_2} + \frac{b_1 b_2 b_3}{Q_2 Q_3} \pm \ldots + (-1)^{k+1} \prod_{i=1}^{k} b_i \pm \ldots
\]

Thereby the algorithm for the accelerated summation of alternating series from section 32.3 can be applied to compute \( x \).

35.3.3.4 Continued fractions from infinite products

A continued fraction for the product

\[
P := \prod_{k=0}^{\infty} (1 + Y_k)
\]

in terms of

\[
a = [1, 1, +Y_1 + (1 + Y_1) \cdot Y_0, +Y_2 + (1 + Y_2) \cdot Y_1, +Y_3 + (1 + Y_3) \cdot Y_2, \ldots] \quad (35.3-20b)
\]

\[
b = [1, +Y_0, -1 \cdot Y_1 \cdot (1 + Y_0), -Y_0 \cdot Y_2 \cdot (1 + Y_1), -Y_1 \cdot Y_3 \cdot (1 + Y_2), \ldots] \quad (35.3-20c)
\]

Y(k) = eval(Str("Y" k)) \ \text{return symbol Yk}

yprod(n) = if (n<0, 1, prod(j=0, n-1, (1+Y(j))))
n=3
pr = yprod(n)
((Y2 + 1)*Y1 + (Y2 + 1))*Y0 + ((Y2 + 1)*Y1 + (Y2 + 1))
yv = vector(n, j, Y(j-1) )
\[Y_0, Y_1, Y_2\]
t = cfprod(yv);
a=t[1]
\[1, 1, (Y_1 + 1)*Y_0 + Y_1, (Y_2 + 1)*Y_1 + Y_2\]
b=t[2]
\[1, Y_0, -Y_1*Y_0 - Y_1, (-Y_2*Y_1 - Y_2)*Y_0\]

\{ for(k=0, n, 
t=cfabZpq(a,b, k); 
p=t[1]; q=t[2]; 
print1(k, ": (",p, ") / (", q,")); 
yp = yprod(k); 
print1("\n == ", simplify(p/q)); 
print(); 
); 
\}

0: (1) / (1) == 1
1: (Y0 + 1) / (1) \ \text{\(p1\)} / (q1) == Y0 + 1 \ \text{\(yprod(1)\)}
2: (((Y1 + 1)*Y0 + (Y1 + 1)*Y1) / Y0) \ \text{\(p2\)} / (q2) == (Y1 + 1)*Y0 + (Y1 + 1) \ \text{\(yprod(2)\)} == (1+Y0)*(1+Y1)
3: (((Y2 + 1)*Y1 + (Y2 + 1)*Y0) / Y0 + ((Y2 + 1)*Y1 + (Y2 + 1)*Y0) / (Y1*Y0) == ((Y2 + 1)*Y1 + (Y2 + 1)*Y0) * Y0 + ((Y2 + 1)*Y1 + (Y2 + 1))

Figure 35.3-D: Verification of relations 35.3-20b and 35.3-20c using pari/gp.

For a given a vector \( y = [Y_0, Y_1, \ldots] \) the computation of individual values \( a_k \) and \( b_k \) can be implemented as:

[fxtbook draft of 2008-August-17]
35.3: Continued fractions

\[
\text{cfproda}(yv, n) = \begin{cases} 
3 & \text{if } (n \leq 1, \text{return}(1) ); \\
2 & y3 = yv[n]; \\
1 & y2 = yv[n-1]; \\
0 & \text{return}( (y2+y3*(1+y2)) ); 
\end{cases}
\]

\[
\text{cfprodb}(yv, n) = \begin{cases} 
2 & \text{if } (0==n, \text{return}(1) ); \ \text{\textbackslash unused} \\
1 & y3 = yv[n]; \\
0 & y2 = yv[n-1]; \\
-1 & y1 = \text{if } ( n==2, 1, yv[n-2] ); \\
0 & \text{return}( -y1*y3*(1+y2) ); 
\end{cases}
\]

The routine \text{cfprod()} generates the vectors \(a\) and \(b\) with \(n+1\) terms where \(n\) is the length of \(y\):

\[
\text{cfprod}(yv) = \begin{cases} 
1 & \text{local}(n, a, b); \\
2 & n = \text{length}(yv); \\
3 & n += 1; \ \text{\textbackslash n+1 terms in continued fraction} \\
4 & a = \text{vector}(n); \\
5 & b = \text{vector}(n); \\
6 & \text{for } (k=0, n-1, \\
7 & a[k+1] = \text{cfproda}(yv, k); \\
8 & b[k+1] = \text{cfprodb}(yv, k); \\
9 & \text{return}( [a, b] ); 
\end{cases}
\]

Relations 35.3-20b and 35.3-20c can be verified using pari/gp as shown in figure 35.3-D.

35.3.3.5 An expression for a sum of products

Define \(Z_n\) as

\[
Z_n := z_1 + z_1 z_2 + z_1 z_2 z_3 + z_1 z_2 z_3 z_4 + \ldots = \sum_{k=1}^{n} \prod_{i=1}^{k} z_i = z_1 \left[ 1 + z_2 \left[ 1 + z_3 \left[ 1 + z_4 \left[ \ldots \right] \right] \right] \right] \quad (35.3-21a)
\]

Then \(Z_\infty\) has the continued fraction

\[
Z_\infty = \frac{z_1}{1 - \frac{z_2}{1 + z_2 - \frac{z_3}{1 + z_3 - \frac{z_4}{1 + z_4 - \frac{z_5}{1 + z_5 - \ldots}}}}} \quad (35.3-22)
\]

That is, \(Z = K(a, b)\) where

\[
a = [0, 1, z_2 + 1, z_3 + 1, z_4 + 1, z_5 + 1, z_6 + 1, \ldots] \quad (35.3-23a)
\]

\[
b = [1, z_1, -z_2, -z_3, -z_4, -z_5, -z_6, \ldots] \quad (35.3-23b)
\]

For the \(n\)-th convergent \(P_n/Q_n\) one has \(Q_n = 1\) and \(P_n = Z_n\).
To convert the power series of a hypergeometric function (see section 35.2 on page 696)

\[ F \left( \begin{array}{c} a_1, a_2, \ldots, a_u \\ b_1, b_2, \ldots, b_v \end{array} \bigg| z \right) \]  

into a continued fraction, set \( z_1 = 1 \), and for \( k \geq 1 \) set

\[ z_{k+1} = \frac{z}{k \prod_{j=1}^{u} (a_j + k)} \prod_{j=1}^{v} (b_j + k) \]  

An implementation is

```python
1 hyper2cf(va, vb, n, z='z')=
2 \ convert hypergeom(va,vb,z) to continued fraction
3 {
4    local(cfa, cfb, m);
5    n += 2;
6    cfa = vector(n);
7    cfb = vector(n);
8    cfa[1] = 0; cfa[2] = 1;
10   for (k=3, n,
11       m = 1/(k-2); \ hidden lower parameter 1: (n-2) == 1+(n-3)
12       m *= prod(j=1, #va, va[j]+(k-3)); \ upper parameters
13       m /= prod(j=1, #vb, vb[j]+(k-3)); \ lower parameters
14       m *= z; \ argument
15       cfa[k]=(m+1);
16       cfb[k]=-m;
17   );
18   return( [cfa, cfb] );
}
```

We convert \( \log(1-z)/z = F \left( \begin{array}{c} 1 \\ 2 \end{array} \bigg| z \right) \) to a continued fraction and check the result:

```plaintext
? N=7;
? va=[1,1];vb=[2];
? t=hyper2cf(va,vb,N);
? cfa=t[1]
[0, 1, 1/2*z + 1, 2/3*z + 1, 3/4*z + 1, 4/5*z + 1, 5/6*z + 1, 6/7*z + 1, 7/8*z + 1]
? cfb=t[2]
[1, 1, -1/2*z, -2/3*z, -3/4*z, -4/5*z, -5/6*z, -6/7*z, -7/8*z]
? t=cfab2pq(cfa,cfb)
[1/8*z^7 + 1/7*z^6 + 1/6*z^5 + 1/5*z^4 + 1/4*z^3 + 1/3*z^2 + 1/2*z + 1, 1]
? s1=t[1]/t[2]z^N)
1 + 1/2*z + 1/3*z^2 + 1/4*z^3 + 1/5*z^4 + 1/6*z^5 + 1/7*z^6 + 0(z^7)
? s2=hypergeom(va,vb,z,N)+O(z^N)
1 + 1/2*z + 1/3*z^2 + 1/4*z^3 + 1/5*z^4 + 1/6*z^5 + 1/7*z^6 + 0(z^7)
```

For further information on continued fractions see \[213\] and \[134\]. An in depth treatment is \[192\].
Chapter 36

Synthetic Iterations *

It is easy to construct arbitrary many iterations that converge super-linearly. Guided by some special constants that in base 2 can be obtained by recursive constructions we build iterations that allow the computation of the constant in a base independent manner. The iterations lead to functions that typically cannot be identified in terms of known (named) functions. Some of the functions can be expressed as infinite sums or products. For the constructions with repeated string substitutions see chapter 16.

36.1 A variation of the iteration for the inverse

We start with the product form for the most simple iteration, the one for the inverse:

\[
I(y) := \frac{1}{1-y}
\]

\[
= 1 + y + y^2 + y^3 + y^4 + \ldots
\]

\[
= (1 + y) (1 + y^2) (1 + y^4) (1 + y^8) \ldots (1 + y^{2^n}) \ldots
\]

\[
= (1 + Y_0) (1 + Y_1) (1 + Y_2) (1 + Y_3) \ldots (1 + Y_k) \ldots
\]

where \( Y_0 = y, Y_{k+1} = Y_k^2 \)

We now modify the signs in the infinite product:

\[
J(y) := (1 - y) (1 - y^2) (1 - y^4) (1 - y^8) \ldots (1 - y^{2^n}) \ldots
\]

\[
= 1 - y - y^2 + y^3 - y^4 + y^5 + y^6 - y^7 - y^8 \pm \ldots
\]

\[
= (1 - Y_0) (1 - Y_1) (1 - Y_2) (1 - Y_3) \ldots (1 - Y_k) \ldots
\]

\[
\text{where } Y_0 = y, Y_{k+1} = Y_k^2
\]

The value of the \( n \)-th coefficient equals plus one if the parity of \( n \) is zero, else minus one (sequence [A106400](http://oeis.org/A106400), the Thue-Morse sequence). The function \( J \) can be implemented as

```c
fj(y, N=5)="
6  { local(r);
5      r = 1;
6       for (k=1, N,
5           r -= r*y;
4           y *= y;
4         );
5     return(r);
4 })
```

[fxtbook draft of 2008-August-17]
Replacing the minus by a plus gives the implementation for the function $I$.

A related constant is the parity number (or Prouhet-Thue-Morse constant):

$$P = 0.4124540336401075977833613682584552830894783744557695575 \ldots$$  (36.1-3)

- $[\text{base 2}] = 0.0110, 1001, 1001, 0110, 0110, 1001, 1001, 1001, 0110, 0110, 1001, \ldots$
- $[\text{base 16}] = 0.6996, 9696, 6996, 9696, 6996, 9696, 9696, 6996, 9696, \ldots$
- $[\text{CF}] = [0, 2, 2, 2, 1, 4, 3, 5, 2, 1, 4, 2, 1, 5, 44, 1, 4, 1, 1, 2, 4, 1, 1, 5, 1, 15, 5, 1, 1, 4, 2, 1, \ldots]$

The sequence of zeros and ones in the binary expansions is entry \([A010060]\) of \([243]\). The constant $P$ can be computed defining

$$K(y) = \frac{[I(y) - J(y)]}{2} = y + y^2 + y^4 + y^7 + y^8 + y^{11} + y^{13} + y^{14} + y^{16} + \ldots$$  (36.1-4)

Then

$$P = \frac{1}{2} K \left( \frac{1}{2} \right) = \frac{1}{2} \left[ \frac{I(\frac{1}{2}) - J(\frac{1}{2})}{2} \right] = \frac{1}{2} - \frac{1}{4} J \left( \frac{1}{2} \right)$$  (36.1-5)

Thereby (see also \([33]\), item 125),

$$2 - 4P = J \left( \frac{1}{2} \right) = \prod_{k=0}^{\infty} \left( 1 - \frac{1}{2^{2k}} \right) = 0.35018386543956960886554526966178 \ldots$$  (36.1-6)

The sequence of bits of the parity number can also be obtained by starting with a single 0 and repeated application of the substitution rules $0 \rightarrow 01$ and $1 \rightarrow 10$.

The following relations are direct consequences of the definitions of the functions $I$ and $J$:

$$I(y) I(-y) = I(y^2)$$  (36.1-7a)

$$I(y) = \frac{J(y^2)}{J(y)}$$  (36.1-7b)

$$I(-y) = \frac{1 - y}{1 + y} I(y)$$  (36.1-7c)

$$J(-y) = \frac{1 + y}{1 - y} J(y)$$  (36.1-7d)

We have

$$I(y) = 1 + \sum_{k=0}^{\infty} \left[ y^{2^k} \prod_{j=0}^{k-1} \left( 1 + y^{2^j} \right) \right]$$  (36.1-8a)

$$J(y) = 1 - \sum_{k=0}^{\infty} \left[ y^{2^k} \prod_{j=0}^{k-1} \left( 1 - y^{2^j} \right) \right]$$  (36.1-8b)

A functional equation for $K$ is

$$K(y) = (1 - y) K(y^2) + \frac{y}{1 - y^2}$$  (36.1-9)

It is solved by

$$K(y) = \sum_{k=0}^{\infty} \left[ \frac{y^{2^k}}{1 - y^{2^{k+1}}} \prod_{j=0}^{k-1} \left( 1 - y^{2^j} \right) \right]$$  (36.1-10)
36.1: A variation of the iteration for the inverse

For the inverse of $J$ we have

$$\frac{1}{J(y)} = 1 + y + 2y^2 + 2y^3 + 4y^4 + 4y^5 + 6y^6 + 6y^7 + 10y^8 + 10y^9 + 14y^{10} + \ldots \quad (36.1-11a)$$

$$= \frac{[(1 - y)(1 - y^2)(1 - y^4)(1 - y^8)\ldots]^{-1}}{\prod_{k=0}^{\infty} I(y^{2^k})} \quad (36.1-11b)$$

$$= (1 - y) \prod_{k=0}^{\infty} \frac{1 + y^{2^k}}{1 - y^{2^k}} \quad (36.1-11c)$$

$$= (1 + y)(1 + y^2)^2 (1 + y^4)^3 (1 + y^8)^4 (1 + y^{16})^5 \ldots (1 + y^{2^k})^{k+1} \ldots \quad (36.1-11d)$$

Relation 36.1-11d can be used for a divisionless algorithm for the computation of $1/J$:

```plaintext
1   binpart(y,N=5)=
2   { local(r);
3       r = 1;
4       for (k=1, N,
5               for (j=1, k, r += r*y; );
6           y *= y;
7       );
8   return(r);
9 }
```

The sequence of coefficients of the even powers of $x$ in relation 36.1-11a is

$$1, 2, 4, 6, 10, 14, 20, 26, 36, 46, 60, 74, 94, 114, 140, 166, 202, \ldots$$

This is entry A000123 of [245], the number of binary partitions of the even numbers. The sequence $\frac{1}{2}[2, 4, 6, 10, 14, \ldots]$ modulo two equals the period-doubling sequence, see section 36.5 on page 732. The generating function $1 + 2y + 4y^2 + 6y^3 + 10y^4 + 14y^5 + 20y^6 + \ldots$ equals

$$\frac{I(y)}{J(y)} = (1 + y)^2 (1 + y^2)^3 (1 + y^4)^4 (1 + y^8)^5 (1 + y^{16})^6 \ldots (1 + y^{2^k})^{k+2} \ldots \quad (36.1-12)$$

It can be computed via (note the change in the inner loop)

```plaintext
1   binpart2(y,N=5)=
2   { local(r);
3       r = 1;
4       for (k=1, N,
5               for (j=1, k+1, r += r*y; );
6           y *= y;
7       );
8   return(r);
9 }
```

For the function $I$ we have

$$I(y) = \sum_{k=0}^\infty \frac{y^{2^k-1}}{1 - y^{2^{k+1}}} = \sum_{k=0}^\infty \frac{2^k y^{2^k-1}}{1 + y^{2^k}} \quad (36.1-13a)$$

Integration gives

$$-\log(1 - y) = \sum_{k=0}^\infty \frac{1}{2^{k+1}} \log \left( \frac{1 + y^{2^k}}{1 - y^{2^k}} \right) = \sum_{k=0}^\infty \log \left( 1 + y^{2^k} \right) \quad (36.1-14a)$$

For the derivative of $J$ we have

$$J'(y) = -J(y) \sum_{k=0}^\infty \frac{2^k y^{2^k-1}}{1 - y^{2^k}} \quad (36.1-15)$$
The following functional equations hold for $I(y)$:

$$
0 = B - 2AB + A^2 \text{ where } A = I(y), \quad B = I(y^2) \quad (36.1-16a)
$$
$$
0 = B - 2AB - A^2 + 2A^2B \text{ where } A = I(-y), \quad B = I(-y^2) \quad (36.1-16b)
$$
$$
0 = B - 3AB + 3A^2B - A^3 \text{ where } A = I(y), \quad B = I(y^3) \quad (36.1-16c)
$$
$$
0 = B - 5AB + 10A^2B - 10A^3B + 5A^4B - A^5 \text{ where } A = I(y), \quad B = I(y^5) \quad (36.1-16d)
$$
$$
A = I(y), \quad B = I(y^3) \quad (36.1-16e)
$$

From the functional equation for $K(y)$ (relation 36.1-9), the definition of $K(y)$, and relation 36.1-7b one can derive the following relation for $J(y)$:

$$
0 = J_2^3 - 2J_1J_2J_3 + J_4J_1^2 \text{ where } J_1 = J(y), \quad J_2 = J(y^2), \quad J_3 = J(y^3), \quad J_4 = J(y^4) \quad (36.1-17)
$$

This relation is given in entry [A106400] of [243], together with

$$
0 = J_0J_1^3 - 3J_0J_2J_3^2 + 3J_0J_2^2J_1 - J_3J_2^3 \quad (36.1-18)
$$

where $J_k = J(y^k)$. Relations between $J_1$, $J_2$, $J_k$, and $J_{2k}$ can be derived from relation 36.1-16e by replacing $I(y)$ by $J(y^3)/J(y)$. For example, $k = 5$ gives

$$
0 = J_{10}J_5^5 - 5J_{10}J_2J_4^2 + 10J_{10}J_2^2J_3^3 - 10J_{10}J_2^3J_1^2 + 5J_{10}J_4^2J_1 - J_5J_2^5 \quad (36.1-19)
$$

**The Komornik-Loreti constant**

One has $K(1/β) = 1$ for

$$
\frac{1}{β} = 0.5595245584967265251322097651574322858310764789686603076 \ldots \quad (36.1-20a)
$$

$[\text{base } 2] = 0.1000111110111000000001101000000000101101000101110 \ldots$

$$
β = 1.787231650182965933013274890337008385337931402961810997 \ldots \quad (36.1-20b)
$$

$[\text{base } 2] = 1.110010111000100000000101101111110100110111011000000 \ldots$

$[\text{CF}] = [1, 1, 3, 1, 2, 3, 188, 1, 12, 1, 1, 22, 33, 1, 10, 1, 1, 7, 1, 9, 1, 1, 20, 2, 15, 1, \ldots]$

The constant $β$ is the smallest real number in the interval $(1, 2)$ so that $1$ has a unique expansion of the form $\sum_{n=1}^{\infty} δ_n β^{-n}$ where $δ_n ∈ [0, 1]$. It is called the Komornik-Loreti constant (see [8]). The fact that $δ_n = 1$ exactly where the Thue-Morse sequence equals $1$ was used for the computation of $β$: one solves $K(y) = 1$ for $y$. The transcendence of $β$ is proved (using the fact that $J(y)$ is transcendental for algebraic $y$) in [9].

**Third order variants**

Variations of the third order iteration for the inverse

$$
I(y) := \frac{1}{1 - y} = 1 + y + y^2 + y^3 + y^4 + \ldots \quad (36.1-21a)
$$

$$
= (1 + y + y^2)(1 + y^3 + y^6)(1 + y^9 + y^{18}) \ldots (1 + y^{3k} + y^{2\cdot3k}) \ldots \quad (36.1-21b)
$$

$$
= (1 + Y_0 + Y_0^2)(1 + Y_1 + Y_1^2)(1 + Y_2 + Y_2^2) \ldots (1 + Y_k + Y_k^2) \ldots \quad (36.1-21c)
$$

where $Y_0 = y$, $Y_{k+1} = Y_k^3$

lead to series related to the base-3 analogue of the parity. The most simple example may be

$$
T(y) = (1 + Y_0 - Y_0^2)(1 + Y_1 - Y_1^2)(1 + Y_2 - Y_2^2) \ldots (1 + Y_k - Y_k^2) \ldots \quad (36.1-22a)
$$

$$
= 1 + y - y^2 + y^3 - y^4 - y^5 - y^6 + y^8 + y^9 + y^{10} - y^{11} + y^{12} \pm \ldots \quad (36.1-22b)
$$

[fxtbook draft of 2008-August-17]
The sign of the $n$-th coefficient is the parity of the number of twos in the radix-3 expansion of $n$. We have
\[
\frac{1}{2} \left( I(y) - T(y) \right) = y^2 + y^5 + y^6 + y^7 + y^{11} + y^{14} + y^{15} + y^{16} + y^{18} + y^{19} + y^{21} + \ldots \tag{36.1-23a}
\]
\[
\frac{1}{4} \left( I \left( \frac{1}{2} \right) - T \left( \frac{1}{2} \right) \right) = 0.1526445236254075825319249214757916793115045148714892548\ldots \tag{36.1-23b}
\]
\[
[\text{base } 2] = 0.0010011000100110110001001100010011101101100011\ldots
\]
\[
[\text{CF}] = [0, 6, 1, 4, 2, 1, 1, 2, 4, 1, 1, 4, 1, 4, 2, 1, 1, 2, 1, 1, 8, 3, 24, 1, 6, 1, 3, \ldots]
\]
The sequence of zeros and ones in the binary expansion is entry [A064990] of [215], the Mephisto Waltz sequence.

### 36.2 An iteration related to the Thue constant

We construct a sequence of zeros and ones that can be generated by starting with a single zero and repeated application of the substitution rules $0 \to 111$ and $1 \to 110$. The evolution starting with a single zero is:

\[
\begin{align*}
T_0 &= 0 \\
T_1 &= 111 \\
T_2 &= 110110110 \\
&= 3 \times \text{times } 11.0 \\
T_3 &= 110110111101110110111 \\
&= 3 \times \text{times } 110110111101111011011110110111110111101101111011111011111011111011111101101111101101101101101111101101111101101101101101111 \ldots
\end{align*}
\]

The crucial property is that $T_n$ is three time repeated the string $T_{n-1}T_{n-2}$ where $T_k$ consists of the first and second third of $T_k$. The length of the $n$-th string is $3^n$. Let $T(y)$ be the function whose power series corresponds to the string $T_\infty$:
\[
T(y) = 1 + y + y^3 + y^4 + y^6 + y^7 + y^8 + y^9 + y^{10} + y^{12} + y^{13} + y^{15} + y^{16} + y^{17} + y^{18} + \ldots \tag{36.2-1}
\]

It can be computed by the iteration
\[
\begin{align*}
L_0 &= 0, & A_0 &= 1 + y, & B_0 &= y^2, & Y_0 &= y \\
R_n &= A_n + y^2 L_n \\
L_{n+1} &= A_n + B_n \\
Y_{n+1} &= Y_n^3 \\
A_{n+1} &= R_n (1 + Y_{n+1}) \rightarrow T(y) \\
B_{n+1} &= R_n Y_{n+1}^2
\end{align*} \tag{36.2-2a}
\]

The implementation is slightly tricky:

```plaintext
1  th(y, N=5)=
2  {
3    local(L, R, A, B, y2, y3, t);
4    /* correct up to order 3^N(N+1)-1 */
5    L=0;
6    A=1+y; B=y^2; /* R = A.B */
7    for(k=1, N,
8      /* (L, A, B) --> (A.B, A.L.A.L, A.L) */
9       y2 = y^2;
10      R = A + y2*L; /* A.L */
11      L = A + B; /* next L = A.B */
12      y3 = y * y2;
13      B = R * (y3*y3);
14      A = R * (1+y3); /* next A = A.L.A.L */
15      y = y3;
16    );
17    return(A + B )
18  }
```

[fxtbook draft of 2008-August-17]
The *Thue constant* (which should be *Roth’s constant*, see entry \[A014578\] of \[245\]) can be computed as

\[
\frac{1}{2}T\left(\frac{1}{2}\right) = 0.859099796854703104903572502841974202614239955594390874\ldots
\]

\[\text{[base 2]} \quad 0.110, 110, 111, 110, 110, 111, 110, 110, 110, 111, 110, 110, 110, 111, 110, 110, 110, \ldots
\]

\[\text{[base 8]} \quad 0.667, 667, 666, 667, 666, 667, 667, 667, 666, 667, 666, 667, 667, \ldots
\]

\[\text{[CF]} \quad [0, 1, 6, 10, 3, 2, 513, 1, 1, 2, 1, 4, 2, 6576687699, 1, 1, 4,
\]

\[1, 2, 2, 256, 1, 1, 2, 1, 2, 3, 1, 3, 3, 241785163922958349412353,
\]

\[1, 1, 4, 1, 2, 2, 146, 2, 3, 3, 2, 1, 2, 1, 12, X, \ldots]

The term \(X\) in the continued fraction has 74 decimal digits. By construction the bits at positions \(n\) not divisible by three are one and otherwise the complement of the bit at position \(n/3\). As a functional equation (see also section \[36.5\] on page \[732\]):

\[
yT(y) + y^3T(y^3) = \frac{y}{1-y}
\]

From this relation we can obtain a series for \(T(y)\):

\[
T(y) = \sum_{n=0}^{\infty} (-1)^n \frac{y^{3n+1}}{1-y^{3n}}
\]

\[\text{[fxtbook draft of 2008-August-17]}\]

### 36.3 An iteration related to the Golay-Rudin-Shapiro sequence

Define \(Q(y)\) by

\[
L_0 = 1, \quad R_0 = y, \quad Y_0 = y
\]

\[
L_{n+1} = L_n + R_n \rightarrow Q(y)
\]

\[
Y_{n+1} = Y_n^2
\]

\[
R_{n+1} = Y_{n+1} (L_n - R_n)
\]

then

\[
Q(y) = 1 + y + y^2 - y^3 + y^4 + y^5 - y^6 + y^7 + y^8 + y^9 + y^{10} - y^{11} - y^{12} - y^{13} + y^{14} - y^{15} + \ldots
\]

The sequence of coefficients is the *Golay-Rudin-Shapiro sequence* (or *GRS sequence*, entry \[A020985\] of \[245\], see also section \[1.16.5\] on page \[46\]). The constant

\[
Q = 0.9292438695973788532539766447220507644128755395243255222\ldots
\]

\[\text{[base 2]} \quad 0.1110, 1101, 1110, 0010, 1110, 1101, 0001, 1101, 1110, 1110, 0010, \ldots
\]

\[\text{[base 16]} \quad 0.edes, ed1d, ede2, ed2e, ede2, ed1d, ed1d, ede2, ed2e, \ldots
\]

\[\text{[CF]} \quad [0, 1, 13, 7, 1, 1, 15, 4, 1, 3, 1, 2, 2, 1000, 12, 2, 1, 6, 1, 1, 1, 1, 1, 8, 2, 1, 1, 2, 4, 1, 1, 3, \ldots]
\]

can be computed as

\[
Q = \frac{1 + \frac{1}{2}Q^{(1/2)}}{2}
\]

The implementation using pari/gp:
36.3: An iteration related to the Golay-Rudin-Shapiro sequence

```plaintext
1  qq(y, N=8)=
2  { local(L, R, Lp, Rp);
3      /* correct up to order 2**(N+1) */
4      L=1; R=y;
5      for(k=0,N, Lp=L+R; y*=y; Rp=y*(L-R); L=Lp; R=Rp);
6      return( L + R )
7  }
```

The following functional relations hold for $Q$:

$$Q(y^2) = \frac{Q(y) + Q(-y)}{2} \quad (36.3-5a)$$

$$Q(y) = Q(y^2) + yQ(-y^2) \quad (36.3-5b)$$

$$Q(-y) = Q(y^2) - yQ(-y^2) \quad (36.3-5c)$$

Combining the latter two gives

$$Q(y) = (1 + y) Q(y^4) + (y^2 - y^3) Q(-y^4) \quad (36.3-6)$$

Counting zeros and ones in the binary expansion of $Q$

The number of ones and zeros in the first $4^k$ bits of the constant $Q$ can be computed via a string-substitution engine (see chapter 16 on page 357). The hexadecimal expansion can be obtained as:

```
Number of symbols = 4
Start:  e
Rules:
  e --> ed
  d --> e2
  1 --> 12
  2 --> 21
-------------
0:  (#=1) e
1:  (#=2) ed
2:  (#=4) ede2
3:  (#=8) ede2ed1d
4:  (#=16) ede2ed1dede212e2
5:  (#=32) ede2ed1dede212e2ede2ed1d121ded1d
6:  (#=64) ede2ed1dede212e2ede2ed1d121ded1dede2ed1d121ede2ede2ed12e2

A few lines of pari/gp code count the occurrences of the symbols (and thereby of zeros and ones):
```

```plaintext
1  /* e --> ed ; d --> e2 ; 2 --> 1d ; 1 --> 12 */
2  /* e  d ; e 2 ; d 1; 2 1; */
3  mg= [1, 1, 0, 0; 1, 0, 1, 0; 0, 1, 0, 1; 0, 0, 1, 1];
4  mg=mattranspose(mg)
5  { for (k=0, 40,
6      print1( k, " : ");
7      mm=mg^k;
8      t = sum(i=1,4, mm[1,i]);
9      /* e and d have three ones and one zero */
10     /* 1 and 2 have one one and three zeros */
11     n0 = 3*(mm[1]+mm[2]) + (mm[3]+mm[4]); /* # of zeros */
12     n1 = 3*(mm[1]+mm[2]) + (mm[3]+mm[4]); /* # of ones */
13     print ( t, " #0=", n0, " #1=", n1, " diff=", n1-n0, " #1/#0=", 1.0*n1/n0 );
14     ) }
```

We obtain the data shown in figure 36.3-A. The sequence of the numbers of ones is entry A005418 of [245]. It is identical to the sequence of numbers of equivalence classes obtained by identifying bit-strings that are mutual reverses or complements, see section 3.8.1.5 on page 136.
# Chapter 36: Synthetic Iterations *

### 36.4 Iterations related to the ruler function

The \textit{ruler function} \( r(n) \) can be defined to be the highest exponent \( e \) so that \( 2^e \) divides \( n \). Here we consider the function that equals \( r(n) + 1 \) for \( n \neq 0 \) and zero for \( n = 0 \). The partial sequences up to indices \( 2^n - 1 \) are:

\[
\begin{align*}
R_0 &= 0 \\
R_1 &= 0, 1 \\
R_2 &= 0, 1, 2, 1 \\
R_3 &= 0, 1, 2, 1, 3, 1, 2, 1 \\
R_4 &= 0, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1 \\
\vdots
\end{align*}
\]

The limiting sequence is entry \( \text{A001511} \) of [245]. Observe that \( R_n = R_{n-1}(R_{n-1} + [n, 0, 0, \ldots]) \). Define the function \( R(y) \) as the limit of the iteration

\[
\begin{align*}
R_1 &= y, \\
Y_{n+1} &= Y_n, \\
R_{n+1} &= R_n + Y_{n+1} [R_n + (1 + n)] \to R(y)
\end{align*}
\]

Implementation in pari/gp:

```gp
def r2(y, N=11)={ 
  \{ /* correct to order = 2^N-1 */
  local(A);
  A=y;
  for(k=2, N, y *= y; A += y*(A + k); );
  return( A );
}
```

If one replaces in the statement \( A += y*(A + k); \) by \( A += y*(A + 1); \) then the iteration computes \( \frac{y}{1-y} \).

For the function \( R \) we have

\[
\begin{align*}
R \left( \frac{1}{q^2} \right) &= \frac{1}{2q} \left[ (q-1) R \left( \frac{1}{q} \right) + (q+1) R \left( -\frac{1}{q} \right) \right] \\
R \left( \frac{1}{q} \right) &= R \left( \frac{1}{q^2} \right) + \frac{1}{q-1} \\
R(y) &= R(y^2) + \frac{y}{1-y} = R(y^4) + \frac{y}{1-y} + \frac{y^2}{1-y^2} = \ldots
\end{align*}
\]

and so

\[
R(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1-y^{2^n}}
\]

One further has

\[
R(y) = \sum_{n=0}^{\infty} (1+n) \frac{y^{2^n}}{1-y^{2^{n+1}}}
\]

---

*Figure 36.3-A: Number of symbols, zeros and ones with the \( n \)-th step of the string substitution engine for the GRS sequence. For long strings the ratio of the number of zeros and ones approaches one.*

[fxtbook draft of 2008-August-17]
36.4: Iterations related to the ruler function

Define the ruler constant as $R := R(1/2)/2$, then

$$R = 0.7019684139410891602881030370686046772688193807609450337 \ldots$$ (36.4-5)

$$[\text{base 2}] = 0.101100111011000110110100110110110100011001110 \ldots$$

$$[\text{CF}] = [0, 1, 2, 2, 1, 4, 2, 1, 1, 1, 2, 2, 1, 3, 3, 4, 5, 6, 1, 5, 1, 1, 9, 49, 1, 8, 1, 1, 5, 1, $$

$$6, 5, 1, 3, 3, 1, 2, 4, 3, 1, 2, 4, 2, 1, 1, 3, 1, 9, 1, 11, 18, 2, 4, 5, 1, 3, 2, 25, 9,$$

$$2, 3, 1, 2, 3, 1, 9, 1, 2, 8, 1, 3, 4, 1, 1, 1, 1, 31, 1, 1, 6, 1, 13, 1, 1, 14, 1, 6, 1, 1, \ldots]$$

We will now compute the function

$$P(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1 + y^{2^n}} = \sum_{n=0}^{\infty} (1 - n) \frac{y^{2^n}}{1 - y^{2^n+1}}$$ (36.4-6)

The partial sequences of coefficients of the Taylor expansion are

$$
\begin{align*}
{P_0} & = 0 \\
{P_1} & = 0 \\
{P_2} & = 0 \\
{P_3} & = 0 \\
{P_4} & = 0 \\
{P_5} & = 0 \\
{P_6} & = 0 \\
{P_7} & = 0 \\
{P_8} & = 0 \\
{P_9} & = 0
\end{align*}
$$

Observe that $P_n = P_{n-1} \cdot (P_{n-1} - [n-2, 0, 0, \ldots, 0])$. Compute $P(y)$ by

$$P_1 = y, \quad Y_1 = y$$

$$Y_{n+1} = Y_n^2$$

$$P_{n+1} = P_n + Y_{n+1} \cdot [P_n + (1 - n)] \quad \rightarrow P(y)$$ (36.4-7c)

Implementation in pari/gp:

```plaintext
1 \ p2(y, N+11)=
2 \ { /* correct to order = 2N-1 */
3 \ \ local(A);
4 \ \ A=y;
5 \ \ for(k=2, N, \ y *= y; \ A += y*(A - (k-2)); );
6 \ \ return( A );
7 }
```

One finds for $P$:

$$P \left( \frac{1}{q^2} \right) = \frac{1}{2q} \left[ (q + 1) P \left( \frac{1}{q} \right) + (q - 1) P \left( \frac{-1}{q} \right) \right]$$ (36.4-8a)

$$P \left( \frac{1}{q} \right) = P \left( \frac{1}{q^2} \right) + \frac{1}{q+1}$$ (36.4-8b)

Relations that involve both $P$ and $R$ are

$$P \left( \frac{1}{q^2} \right) = \frac{1}{2} \left[ \frac{q + 3}{q + 1} \right] \left[ \frac{q - 1}{q + 1} \right] P \left( \frac{1}{q} \right) - \frac{q - 1}{q + 1} R \left( \frac{1}{q} \right)$$ (36.4-8c)

$$R \left( \frac{1}{q} \right) = \frac{1}{2} \left[ (q + 1) P \left( \frac{1}{q} \right) + (q + 3) R \left( \frac{1}{q^2} \right) \right]$$ (36.4-8d)

$$P \left( \frac{1}{q} \right) = \frac{1}{2} \left[ (q + 1) P \left( \frac{1}{q^2} \right) + (q - 1) R \left( \frac{1}{q^2} \right) \right]$$ (36.4-8e)

$$\frac{R(y) + P(y)}{2} = \frac{y}{1 - y} = \sum_{n=1}^{\infty} y^n$$ (36.4-8f)

$$\frac{R(y) - P(y)}{2} = R \left( y^2 \right)$$ (36.4-8g)

$$R^2(y) - P^2(y) = 4 \frac{y}{1 - y} R \left( y^2 \right)$$ (36.4-8h)
36.5 An iteration related to the period-doubling sequence

Define
\[
T(y) = \sum_{n=0}^{\infty} \frac{y^{2n}}{1 + (-1)^n y^{2n}} = \sum_{n=0}^{\infty} (-1)^n \frac{y^{2n}}{1 - y^{2n}}
\]
\[
= y + y^3 + y^4 + y^5 + y^7 + y^9 + y^{11} + y^{12} + y^{13} + y^{15} + y^{16} + y^{17} + \ldots
\]

The function can be computed by the iteration
\[
\begin{align*}
A_1 &= 0, \quad L_1 = y, \quad R_1 = y, \quad Y_1 = y \\
A_{n+1} &= L_n + Y_n R_n \quad \rightarrow T(y) \\
L_{n+1} &= L_n + Y_n A_n \\
R_{n+1} &= R_n + Y_n A_n \\
Y_{n+1} &= Y_n^2
\end{align*}
\]

Implementation in pari/gp:

```plaintext
1 t2(y, N=11)=
2 { /* correct to order = 2*N-1 */
3 local(A, L, R, t);
4 A=0; L=y; R=y;
5 for(k=2, N,
6 t = y*A;
7 A = L + y*R;
8 L *= t; R *= t;
9 y *= y;
10 );
11 return( A );
12 }
```

The computed Taylor series can be obtained symbolically by starting with a single zero and applying the substitution rules $0 \rightarrow 11$ and $1 \rightarrow 10$. The evolution is:

```
T0 = 0
T1 = 0 1
T2 = 0 1 1 0 1
T3 = 0 1 1 0 1 0 1 1 0 1
T4 = 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1
T5 = 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0
T6 = 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1
T7 = 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1
T8 = 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 0
T9 = 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 0 1
```

Observe that $T_n = L(T_{n-1}), R(T_{n-2}), R(T_{n-1}) R(T_{n-2})$ where $L$ and $R$ denote the left and right half of their arguments. The limiting sequence is the so-called period-doubling sequence. It is entry [A035263](https://oeis.org/A035263) of [215](https://oeis.org/A215) where it is called the *first Feigenbaum symbolic sequence*. Define the period-doubling constant as $T \coloneqq T(1/2)$, then

\[
T = 0.7294270234949484057090662068940526170600269444658547417\ldots
\]

[base 2] = 0.101110101101110110101101011101011011011101011010110\ldots
[CF] = [0, 1, 2, 1, 2, 3, 2, 8, 1, 1, 1, 2, 1, 8, 6, 1, 2, 1, 2, 8, 1, 2, 2, 1, 1, 24, 2, 2, 2, 1, 8, 2, 1, 2, 1, 8, 6, 1, 2, 1, 2, 2, 1, 1, 13, 1, 1, 8, 2, 1, 3, 1, 1, 1, 2, 2, 1, 2, 1, 6, 8, 1, 2, 1, 2, 8, 1, 2, 2, 1, 1, 24, 2, 2, 2, 1, 8, 2, 1, 2, 1, 11, 1, 9, 2, 1, 116, \ldots]

The transcendence of this constant is proved in [162](https://oeis.org/A162). A functional equation for $T(y)$ is

\[
T(y) + T(y^2) = \frac{y}{1 - y}
\]

The Taylor series of $T(y^2)$ has coefficients one where the series of $T(y)$ has coefficients zero.
36.5: An iteration related to the period-doubling sequence

<table>
<thead>
<tr>
<th>pile_0</th>
<th>pile_+</th>
<th>pile_-</th>
<th>moved</th>
<th>summary</th>
<th>direction of move</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>1111</td>
<td>....</td>
<td>....</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>111.</td>
<td>....</td>
<td>...1</td>
<td>0 0 0 - 1</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>11.</td>
<td>....</td>
<td>...1</td>
<td>0 0 + + + 0</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>1.11</td>
<td>....</td>
<td>...1</td>
<td>0 - + + + 1</td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>1.1</td>
<td>....</td>
<td>...1</td>
<td>0 - + 0 0 0</td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>1.11</td>
<td>....</td>
<td>...1</td>
<td>0 - + 0 0 0</td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td>11.1</td>
<td>....</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>1111</td>
<td>....</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>8:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>9:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>10:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>12:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
<tr>
<td>15:</td>
<td>....</td>
<td>1111</td>
<td>...1</td>
<td>0 + + + + 1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 36.5-A: Solution of the towers of Hanoi puzzle for 4 disks. The rightmost column corresponds to the direction of the move made, it is the period-doubling sequence.

36.5.1 Connection to the towers of Hanoi puzzle

The *towers of Hanoi* puzzle consists of three piles and \( n \) disks of different size. The initial configuration is that all disks are on the leftmost pile ordered by size (smallest on top). The task is to move all disks to the rightmost pile by moving only one disk at a time and never putting a bigger disk on top of a smaller one.

The puzzle with \( n \) disks can be solved in \( 2^n - 1 \) steps. Figure 36.5-A shows the solution for \( n = 4 \) [FXT: bits/hanoi-demo.cc]. Here the piles are represented as binary words. Note that with each move the lowest bit in one of the three words is moved to another word where it is again the lowest bit.

A simple solution can be obtained by observing that the disk moved with step \( k = 1, \ldots, 2^n - 1 \) corresponds to the lowest set bit in the binary representation of \( k \) and the index of the untouched pile changes by +1 mod 3 for \( n \) even and -1 mod 3 for \( n \) odd. The essential part of the implementation is

```plaintext
void hanoi(ulong n)
{
    ulong f[3];
    f[0] = first_comb(n); f[1] = 0; f[2] = 0; // Initial configuration
    const int dr = (n&1 ? -1 : +1); // == +1 (if n even), else == -1
    // PRINT configuration
    int u; // index of tower untouched in current move
    if (dr<0) u=2; else u=1;
    ulong n2 = 1UL<n;
    for (ulong k=1; k<n2; ++k)
    {
        ulong s = lowest_one(k);
        ulong j = 3; while (j--) f[j] ^= s; // change all piles
        f[u] ^= s; // undo change for untouched pile
        u += dr;
        if (u<0) u=2; else if (u>2) u=0; // modulo 3
        // PRINT configuration
    }
}
```

Now with each step the transferred disk is moved by +1 or -1 position (modulo 3). The rightmost column in figure 36.5-A consists of zeros and ones corresponding to the direction of the move. It is the period-doubling sequence. A recursive algorithm for the towers of Hanoi puzzle can be given as [FXT: comb/hanoi-rec-demo.cc].

```plaintext
ulong f[3]; // the three piles
void hanoi(int k, ulong A, ulong B, ulong C)
// Move k disks from pile A to pile C
```


The piles are represented by the binary words \( f[A] \), \( f[B] \), and \( f[C] \), the variable \( k \) is the number of the disk moved. The routine is called as follows:

```c
ulong n = 5;
// Initial configuration:
ulong f[0] = first_comb(n); // n ones as lowest bits
f[1] = 0; f[2] = 0; // empty

// visit initial state
hanoi(n-1, 0, 1, 2); // solve
```

36.5.2 Generalizations of the period-doubling sequence

The functional equation for the period-doubling sequence, relation 36.5-4 can be generalized in several ways. For example, one can look for a function for which \( F_3(y) + F_3(y^3) = y \). It is given by

\[
F(y) = \sum_{k=0}^{\infty} (-1)^k \frac{y^{3^k}}{1 - y^{3^{k+1}}} = z + z^2 + z^4 + z^5 + z^7 + z^8 + z^{10} + z^{11} + z^{13} + z^{14} + z^{16} + z^{17} + \ldots
\]

We can compute a constant:

\[
F_3(1/2) = 0.8590997968547031049035725028419742026142399555594390874 \ldots
\]

But this is just the Thue constant, see section 36.2 on page 727. Large terms occur in the continued fraction expansion of this constant. Even larger terms occur in the continued fractions of \( F_n(1/2) \) for...
$m > 3$ (replace 3 by $m$ in relation 36.5-5a). For example, the 45-th term of $F_5(1/2)$ has 565 digits. In contrast, the greatest of the first 1630 terms of the continued fraction of $F_5(1/2) = T(1/2)$ equals 288. Some sequences corresponding to the higher order analogues of the period-doubling sequence are shown in figure 36.5-B.

A different way to generalize is to search functions for which, for example, the following functional equation holds:

$$F(y) + F(y^2) + F(y^3) = \frac{y}{1-y} \quad (36.5-7)$$

The equation can be solved by writing $F(y) = y/(1-y) - F(y^2) - F(y^3)$ and using recursions that terminate when a prescribed order is reached.

```c
1 F(z, R)=
2 { /* solve F(y) + F(y^2) + F(y^3) = y/(1-y) */
3     local(s, y);
4     y = z + R;
5     s = y/(1-y);
6     if ( y^2!=R, s -= F(z^2, R) );
7     if ( y^3!=R, s -= F(z^3, R) );
8     return(s);
9 }
```

We obtain

?- N=55;
?- default(seriesprecision,N);
?- s=F(y,0(y^N))

$$y + y^4 + y^5 + y^6 + y^7 + y^9 + y^{11} - y^{12} + y^{13} + y^{16} + y^{17} - y^{18} + y^{19} + y^{20} + y^{23} + 2y^{24} + y^{25} + y^{29} + y^{30} + y^{31} + y^{35} + 3y^{36} + y^{37} + y^{41} + y^{42} + y^{43} + y^{44} + y^{45} + y^{47} - 2y^{48} + y^{49} + y^{52} + y^{53} + 2y^{54} + 0(y^{55})$$

To verify that the function $F$ actually satisfies the given functional equation we show the sequences of coefficients of the power series of $F(y^2)$, $F(y^3)$ and their sum:

| $F(y)$ | [0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, -1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 2, 1, ...] |
| $F(y^2)$ | [0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...] |
| $F(y^3)$ | [0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, ...] |

**sum**

| [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...] |

**Figure 36.5-C:** Visual demonstration that $F(y)$ satisfies the functional equation $F(y) + F(y^2) + F(y^3) = y/(1-y)$. Dots are used for zeros.

The Taylor series of $F(y)$ where $F(y) + F(y^2) + F(y^3) + F(y^6) = y/(1-y)$ contains only ones and zeros. One has

$$F(y) = \sum_{k=1}^{\infty} R(k) \frac{x^k}{1-x^k} \quad \text{where} \quad R(k) = \begin{cases} \frac{(-1)^{e_2+e_3}}{2^{e_2}3^{e_3}} & \text{if}\ k = 2^{e_2}3^{e_3} \\ 0 & \text{else} \end{cases} \quad (36.5-8)$$

This is a Lambert series, it can be converted into a Taylor series by relation 35.1-20 on page 690. It turns out that with $k = u2^{e_2}3^{e_3}$ (where neither 2 nor 3 divides $u$), we have

$$r(k) = \begin{cases} 0 & \text{if either of } e_2 \text{ or } e_3 \text{ is odd} \\ 1 & \text{else (i.e. both } e_2 \text{ and } e_3 \text{ are even}) \end{cases} \quad (36.5-9)$$
36.6 An iteration from substitution rules with sign

Substitution rules: \( 0 \rightarrow L_1, 1 \rightarrow 10, \text{ and } L \rightarrow L_0. \)

\[
\begin{align*}
D_0 &= 1, \\
D_1 &= 10, \\
D_2 &= 10L_1, \\
D_3 &= 10L_1L_0, \\
D_4 &= 10L_1L_010L_1L_010L_010L_1L_0.
\end{align*}
\]

Identify \( L \) with \(-1\) and observe that for \( n > 1 \) one has \( D_n = D_{n-1}.(-D_{n-2}D_{n-2}) \). The length of the strings is \( 2^n \). Define \( D(y) \) by the iteration

\[
\begin{align*}
L_0 &= 1, & R_0 &= 1(+0y), & Y_0 &= y \quad (36.6-1a) \\
L_{n+1} &= R_n \quad (36.6-1b) \\
R_{n+1} &= R_n + Y_n^2 (\neg L_n + Y_n L_n) \rightarrow D(y) \quad (36.6-1c) \\
Y_{n+1} &= Y_n^2 \quad (36.6-1d)
\end{align*}
\]

Implementation in pari/gp:

```plaintext
1 dd(y, N=7)=
2 {
3   local(R, L, y2, t);
4   /* correct up to order 2^N */
5   L = 1;
6   R = 1 + 0*y;
7   for(k=1, N,
8       /* (L, R) --> (R, R.(-L).L) */
9       y2 = y^2;
10      t = R;
11      R = R + y2*(-L + y*L); /* R.(-L).L */
12      L = t;
13      y = y2;
14   );
15   return( R )
16 }
```

We have

\[
D(y) = 1 - y^2 + y^3 - y^4 + y^6 - y^8 + y^{10} - y^{11} + y^{12} - y^{14} + y^{15} - y^{16} + y^{18} \pm \ldots 
\]

and (see relation 36.1-4 on page 724)

\[
\frac{y}{1-y} D(y) = K(y) = y + y^2 + y^4 + y^7 + y^8 + y^{11} + \ldots 
\]

The coefficients are one where the Thue-Morse sequence equals minus one. Thereby the parity number can be computed as \( P = \frac{1}{2} D \left( \frac{1}{2} \right) \). A functional equation for \( D \) is

\[
D(y) = y \frac{1-y}{1+y} D(y^2) + \frac{1}{1+y} 
\]

36.7 Iterations related to the sum of digits

The sequence of the sum of binary digits of the natural numbers starting with zero (entry A035263 of 240) can be constructed as follows:

\[
\begin{align*}
S_0 &= 0, & S_1 &= 01, & S_2 &= 0112, \\
S_3 &= 0112112223, & S_4 &= 011211222312233232323234343445.
\end{align*}
\]

Observe that \( S_n = S_{n-1}.(S_{n-1} + I_{n-1}) \) where \( I_n \) is a sequence of \( n \) ones and addition is element-wise.
36.7: Iterations related to the sum of digits

Define \( S(y) \) by

\[
I_1 = 1, \quad A_1 = y, \quad Y_1 = y
\]

\[
I_{n+1} = I_n (1 + Y_n) = \sum_{k=0}^{n-1} y^k
\]

\[
Y_{n+1} = Y_n^2
\]

\[
A_{n+1} = A_n + Y_{n+1} (I_{n+1} + A_n) \rightarrow S(y)
\]

Implementation in pari/gp:

```plaintext
s2(y, N=7)=
{
  local(in, A);
  /* correct to order = 2^N-1 */
  in = 1; /* 1+y+y^2+y^3+...+y^(2^k-1) */
  A = y;
  for(k=2, N,
     in *= (1+y);
     y *= y;
     A += y*(in + A);
  );
  return( A );
}
```

The Taylor series is

\[
S(y) = 0 + y + y^2 + 2y^3 + y^4 + 2y^5 + 2y^6 + 3y^7 + y^8 + 2y^9 + 2y^{10} + 3y^{11} + 2y^{12} + 3y^{13} + \ldots
\]

Define the sum-of-digits constant as \( S := S(1/2)/2 \), then

\[
S = \frac{0.5906317211782167942379325862790645623612384781099326\ldots}{2}
\]

[base 2] = 0.1001100010010111001100010110100110001001011100110001 \ldots

[CF] = [0, 1, 1, 2, 9, 1, 3, 5, 1, 2, 1, 1, 1, 1, 1, 8, 2, 1, 1, 2, 1, 12, 19, 24, 1, 18, 12, 1, \ldots]

We have (see [252])

\[
S(y) = \frac{1}{1-y} \sum_{k=0}^{\infty} \frac{y^{2k}}{1+y^{2^k}}
\]

and also

\[
S(y) = \sum_{k=0}^{\infty} \left[ \frac{y^{2k}}{1+y^{2^k}} \prod_{j=0}^{k-1} \left[ 1 + y^{2^j} \right] \right]
\]

The last relation follows from the functional relation for \( S \),

\[
S(y) = (1 + y) S(y^2) + \frac{y}{1-y^2}
\]

It is of the form \( F(y) = A(y) F(y^2) + B(y) \) where \( A(y) = 1 + y \) and \( B(y) = y/(1 - y^2) \) and has the solution

\[
F(y) = \sum_{k=0}^{\infty} \left[ B(y^{2^k}) \prod_{j=0}^{k-1} A(y^{2^j}) \right]
\]

This can be seen by applying the functional equation several times:

\[
F(y) = A(y) F(y^2) + B(y)
\]

\[
= A(y) [A(y^2) F(y^4) + B(y)] + B(y)
\]

\[
= A(y) [A(y^2) [A(y^4) F(y^8) + B(y^2)] + B(y^2)] + B(y) = \ldots
\]

\[
= B(y) + A(y) B(y^2) + A(y) A(y^2) B(y^4) + \ldots
\]
Weighted sum of digits

Define $W(y)$ by

$$I_1 = \frac{1}{2}, \quad A_1 = 1, \quad Y_1 = y$$

$$I_{n+1} = I_n \cdot \frac{(1+Y_n)}{2} = \frac{1}{2^n} \sum_{k=0}^{2^n-1} y^k$$

$$Y_{n+1} = Y_n^2$$

$$A_{n+1} = A_n + Y_{n+1} (I_{n+1} + A_n) \to W(y)$$

Implementation in pari/gp:

```plaintext
1  w2(y, N=7)=
2  \{
3      local(in, y2, A);
4      /* correct to order = 2^N-1 */
5      in = 1/2; /* 1/2 = (1+y)^2+y^3+...+y^{(2^k-1)} */
6      A = y/2;
7      for(k=2, N,
8        in *= (1+y)/2;
9        y *= y;
10        A += y*(in + A);
11    );
12  return(A);
```

In the Taylor series

$$W(y) = 0 + \frac{1}{2} y + \frac{1}{4} y^2 + \frac{3}{4} y^3 + \frac{1}{8} y^4 + \frac{5}{8} y^5 + \frac{3}{8} y^6 + \frac{7}{8} y^7 +$$

$$+ \frac{1}{16} y^8 + \frac{9}{16} y^9 + \frac{5}{16} y^{10} + \frac{13}{16} y^{11} + \frac{3}{16} y^{12} + \frac{11}{16} y^{13} + \frac{7}{16} y^{14} + \frac{15}{16} y^{15} +$$

$$+ \frac{1}{32} y^{16} + \frac{17}{32} y^{17} + \frac{9}{32} y^{18} + \frac{25}{32} y^{19} + \frac{5}{32} y^{20} + \frac{21}{32} y^{21} + \frac{13}{32} y^{22} + \frac{29}{32} y^{23} +$$

$$+ \frac{3}{32} y^{24} + \frac{19}{32} y^{25} + \frac{11}{32} y^{26} + \frac{27}{32} y^{27} + \frac{7}{32} y^{28} + \frac{23}{32} y^{29} + \frac{15}{32} y^{30} + \frac{31}{32} y^{31} + \frac{1}{64} y^{32} + \ldots$$

the coefficient of the $y^n$ is the weighted sum of digits $w(n) = \sum_{i=0}^{n} 2^{-(i+1)} b_i$ where $b_0, b_1, \ldots$ is the base-2 representation of $n$. Note that the numerator in the $n$-th coefficient is the reversed binary expansion of $n$.

The corresponding weighted-sum-of-digits constant or revbin constant is $W := W(1/2)$. Then

$$W = 0.44852655067627237892368772125452660976162788135384481336 \ldots$$

[base 2] = [0.0111001011010010100010110100101001010010100010100010100001 \ldots]

[CF] = [0, 2, 4, 2, 1, 4, 18, 1, 2, 6, 5, 17, 2, 14, 1, 1, 1, 2, 1, 1, 2, 1, 1, 2, 1, 3, 1, 29, 4, 1, \ldots]

For the function $W$ we have the following functional relation:

$$W\left(\frac{1}{q}\right) = \frac{1}{2} \left[ \frac{1+q}{q} W\left(\frac{1}{q^2}\right) + \frac{q}{q^2-1} \right]$$

### 36.8 Iterations related to the binary Gray code

#### 36.8.1 Series where coefficients are the Gray code of exponents

We construct a function with Taylor series coefficients that are the binary Gray code of the exponent of $y$. A list of the Gray codes is given below (see section 1.16 on page 42):
The sequence of Gray codes is entry A003188 of [245]. Define the function \( G(y) \) as the limit of the iteration

\[
\begin{align*}
F_1 &= y, \quad B_1 = 1, \quad Y_1 = y, \quad I_1 = 1 + y \\
Y_n &= Y_{n-1}^2 \\
F_n &= (F_{n-1} + 2I_{n-1}) + Y_n (B_{n-1}) \quad \rightarrow G(y) \\
B_n &= (F_{n-1} + 2I_{n-1}) + Y_n (B_{n-1}) \\
I_n &= I_{n-1} (1 + Y_n)
\end{align*}
\]  

Implementation in pari/gp:

```gp
1 gg(y, N=15)=
2 { local(t, ii, F, B, Fp, Bp);
3     /* correct up to order 2^N-1 */
4     F=0+y; B=1+0; ii=1+y;
5     for(k=2,N,
6         y *= y; // delete for sum of digits
7         ii *= 2; // delete for sum of digits
8         Fp = (F ) + y * (B + ii);
9         Bp = (F + ii) + y * (B );
10        F = Fp; B = Bp;
11        ii *= (1+y);
12     );
13 return( F )
14 }
```

In the algorithm \( F \) contains the approximation so far and \( B \) contains the reversed polynomial:

\[
\begin{align*}
\text{----- k = 1 :} \\
F &= (y) \\
B &= (1)
\end{align*}
\]

\[
\begin{align*}
\text{----- k = 2 :} \\
F &= (2y^2+3y^3+2y^4+6y^5+5y^6+4y^7+4y^8+3y^9+2y^{10}+y^{11}) \\
B &= (y^2+3y^3+2y^4+6y^5+5y^6+4y^7+4y^8+3y^9+2y^{10}+y^{11})
\end{align*}
\]

\[
\begin{align*}
\text{----- k = 3 :} \\
F &= (4y^7+5y^8+6y^9+7y^{10}+5y^{11}+6y^{12}+4y^{13}+3y^{14}+2y^{15}) \\
B &= (y^6+3y^7+5y^8+2y^9+6y^{10}+7y^{11}+2y^{12}+5y^{13}+6y^{14}+4y^{15}+3y^{16}+2y^{17}+y^{18})
\end{align*}
\]

\[
\begin{align*}
\text{----- k = 4 :} \\
F &= (8y^{15}+9y^{16}+11y^{17}+12+14y^{18}+11+15y^{19}+10+13y^{20}+9+12y^{21}+8+4y^{22}+7+5y^{23}+6+6y^{24}+4+2y^{25}+3+3y^{26}+2+y) \\
B &= (y^{14}+3y^{15}+13+2y^{16}+12+6y^{17}+11+7y^{18}+10+6y^{19}+9+4y^{20}+8+12y^{21}+7+13y^{22}+6+15y^{23}+14y^{24}+10y^{25}+3+11y^{26}+2+9y^{27}+8)
\end{align*}
\]

We obtain the series

\[
G(y) = 0 + 1y + 3y^2 + 2y^3 + 6y^4 + 7y^5 + 5y^6 + 4y^7 + \\
+ 12y^8 + 13y^9 + 15y^{10} + 14y^{11} + 10y^{12} + 11y^{13} + 9y^{14} + 8y^{15} + \\
+ 24y^{16} + 25y^{17} + 27y^{18} + 26y^{19} + 30y^{20} + 31y^{21} + 29y^{22} + 28y^{23} + \\
+ 20y^{24} + 21y^{25} + 23y^{26} + 22y^{27} + 18y^{28} + 19y^{29} + 17y^{30} + 16y^{31} + \ldots
\]  

We define the Gray code constant as \( G := G(1/2) \):

\[
\begin{align*}
G &= 2.302218287787689301229333006391310761000431077704369505 \ldots \\
[\text{base 2}] &= 10.0100110101011110001011011111100100111000101 \ldots \\
[\text{CF}] &= [2, 3, 3, 4, 4, 1, 4, 4, 1, 2, 1, 1, 1, 2, 24, 205, 1, 4, 2, 2, 1, 1, 4, 10, 8, 1, 9, 1, \ldots]
\end{align*}
\]
For the function $G$ we have

$$G\left(\frac{1}{q^2}\right) = \frac{1}{4} \left[ \frac{q}{q+1} G\left(\frac{1}{q}\right) + \frac{q}{q-1} G\left(-\frac{1}{q}\right) \right]$$  \hspace{1cm} (36.8-4a)

$$G\left(\frac{1}{q}\right) = \frac{2(q+1)}{q} G\left(\frac{1}{q^2}\right) + \frac{q^2}{(q-1)(q^2+1)}$$  \hspace{1cm} (36.8-4b)

$$G(y) = 2 \left(1+y\right) G(y^2) + \frac{y}{(1-y)(1+y^2)}$$  \hspace{1cm} (36.8-4c)

$$G\left(-\frac{1}{q}\right) = \frac{2(q-1)}{q} G\left(\frac{1}{q^2}\right) - \frac{q^2}{(q+1)(q^2+1)}$$  \hspace{1cm} (36.8-4d)

$$G\left(\frac{1}{q}\right) = \frac{2(q^2+1)}{q(q-1)} G\left(-\frac{1}{q^2}\right) + \frac{q^2(q^4+4q^3+4q+1)}{(q^4+1)(q^2+1)(q-1)}$$  \hspace{1cm} (36.8-4e)

$$G\left(\frac{1}{q}\right) = \frac{1}{2q^2} \left[ (q+1) \left(q^4+4q^3+4q+1\right) \frac{1}{(q^2+1)} G\left(\frac{1}{q^2}\right) - \frac{q^4+1}{(q-1)} G\left(-\frac{1}{q^2}\right) \right]$$  \hspace{1cm} (36.8-4f)

The function $G(y)$ can be expressed as

$$G(y) = \frac{1}{1-y} \sum_{k=0}^{\infty} \frac{2^k y^{2^k}}{1+y^{2^{k+1}}}$$  \hspace{1cm} (36.8-5)

### 36.8.2 Differences of the Gray code

If one defines $F(y) = (1 - y) G(y)$ to obtain the successive differences of the Gray code itself, the Taylor coefficients are powers of two in magnitude:

$$F(y) = 0 + y + 2y^2 - y^3 + 4y^4 + y^5 - 2y^6 - y^7 + 8y^8 + y^9 + 2y^{10} - y^{11} - 4y^{12} + y^{13} \pm \ldots$$  \hspace{1cm} (36.8-6)

We have

$$F(y) = 2 F\left(y^2\right) + \frac{y}{1+y^2}$$  \hspace{1cm} (36.8-7)

Now, as $y/(1+y) = q/(1+q)$ for $q = 1/y$,

$$F(y) = F\left(\frac{1}{y}\right) = \sum_{k=0}^{\infty} 2^k \frac{y^{2^k}}{1+y^{2^{k+1}}}$$  \hspace{1cm} (36.8-8)

Thereby $F(y)$ can be computed everywhere except on the unit circle. The sum

$$\sum_{k=0}^{\infty} 2^k \frac{y^{2^k}}{1-y^{2^{k+1}}}$$  \hspace{1cm} (36.8-9)

leads to a series with coefficients

$$0 ~ 1 ~ 2 ~ 1 ~ 4 ~ 1 ~ 2 ~ 1 ~ 8 ~ 1 ~ 2 ~ 1 ~ 6 ~ 1 ~ 2 ~ 1 ~ 4 ~ 1 ~ 2 ~ 1 ~ 8 ~ 1 ~ 2 ~ 1 ~ 4 ~ 1 ~ 2 ~ 1 ~ 3 ~ 2 ~ 1 ~ \ldots$$

corresponding to the (exponential) version of the ruler function which is defined as the highest power of two that divides $n$. The ruler function (see section 36.4 on page 730)

$$0 ~ 1 ~ 0 ~ 2 ~ 0 ~ 1 ~ 0 ~ 3 ~ 0 ~ 1 ~ 0 ~ 2 ~ 0 ~ 1 ~ 0 ~ 4 ~ 0 ~ 1 ~ 0 ~ 2 ~ 0 ~ 1 ~ 0 ~ 3 ~ 0 ~ 1 ~ 0 ~ 2 ~ 0 ~ 1 \ldots$$

is the base-2 logarithm of that series.
36.8.3 Sum of Gray code digits

The sequence of the sum of digits of the Gray code of \( k \geq 0 \) is (entry [A005811 in [245]]):

\[
0 \ 1 \ 2 \ 1 \ 2 \ 3 \ 2 \ 1 \ 2 \ 3 \ 4 \ 3 \ 2 \ 3 \ 2 \ 1 \ 2 \ 3 \ 4 \ 3 \ 2 \ 3 \ 2 \ 1 \ 2 \ 3 \ 
\]

Omit the factor 2 in relations 36.8-1c and 36.8-1d on page 739. That is, in the implementation simply remove the line

\[
\text{ii} *= 2; \quad /* \text{delete for sum of digits} */ 
\]

Let \( R(y) \) be the corresponding function, and define the sum of Gray code digits constant as \( R := R(1/2)/2 \), then

\[
R = 0.70147237640373452073555521064133208822798986154212954 \ldots \quad (36.8-10) \\
[\text{base} 2] = 0.1011001100100111010001100110011100110011011000 \ldots \\
[\text{CF}] = [0, 1, 2, 2, 1, 6, 10, 1, 9, 53, 1, 1, 3, 10, 1, 2, 1, 3, 2, 14, 2, 1, 2, 1, 3, 4, 2, \\
1, 34, 1, 1, 3, 1, 1, 109, 1, 1, 4, 2, 9, 1, 642, 51, 4, 3, 2, 2, 2, 2, 1, 2, 3, \ldots] 
\]

One finds:

\[
R \left( \frac{1}{q^2} \right) = \frac{1}{2} \left[ \frac{q}{q+1} R \left( \frac{1}{q} \right) + \frac{q}{q-1} R \left( -\frac{1}{q} \right) \right] \quad (36.8-11a) \\
R \left( \frac{1}{q} \right) = \frac{q+1}{q} R \left( \frac{1}{q^2} \right) + \frac{q^2}{q^3-q^2+q-1} \quad (36.8-11b) \\
R \left( -\frac{1}{q} \right) = \frac{q-1}{q} R \left( \frac{1}{q^2} \right) - \frac{q^2}{q^3+q^2+q+1} \quad (36.8-11c) 
\]

The function \( R(y) \) can be expressed as

\[
R(y) = \frac{1}{1-y} \sum_{k=0}^{\infty} \frac{y^k}{1+y^{2^{k+1}}} \quad (36.8-12) 
\]

Define the constant \( P \) as \( P = (R+1)/2 \)

\[
P = 0.8507361882018672603677977605320666044113994930827106477 \ldots \quad (36.8-13) \\
[\text{base} 2] = 0.110110011100110100011001100111001100110011001101100 \ldots \\
[\text{CF}] = [0, 1, 5, 1, 2, 3, 21, 1, 4, 107, 7, 5, 2, 1, 2, 1, 1, 2, 1, 6, 1, 2, 6, 1, 1, 8, 1, \\
2, 17, 3, 1, 1, 3, 1, 54, 3, 1, 1, 1, 2, 1, 4, 2, 321, 102, 2, 6, 1, 4, 1, 5, 2, \ldots] 
\]

Its binary expansion is the paper-folding sequence (or dragon curve sequence), entry [A014577 of [245]].

36.8.4 Differences of the sum of Gray code digits

Now define \( E(y) = (1-y)R(y) \) to obtain the differences of the sum of Gray code digits. From this definition (and relation 36.8-12) one sees that

\[
E(y) = \sum_{k=0}^{\infty} \frac{y^k}{1+y^{2^{k+1}}} \quad (36.8-14) 
\]

All Taylor coefficients (except for the constant term) are either plus or minus one:

\[
E(y) = 0 + y + y^2 - y^3 + y^4 + y^5 - y^6 - y^7 + y^8 + y^9 + y^{10} - y^{11} - y^{12} + y^{13} \pm \ldots \quad (36.8-15) 
\]
Chapter 36: Synthetic Iterations *

We have

\[ E \left( \frac{1}{q} \right) = E \left( \frac{1}{q^2} \right) + \frac{q}{q^2 + 1} \]  
\[ E (y) = E (y^2) + \frac{y}{y^2 + 1} \]  

(36.8-16a)  

(36.8-16b)

(use \( \frac{y}{y^2 + 1} = \frac{q}{q^2 + 1} \) where \( q = \frac{1}{y} \) for the latter relation), thereby

\[ E (y) = E \left( \frac{1}{y} \right) \]  

(36.8-17)

So we can compute \( E(y) \) everywhere except on the unit circle. For \( y < 1 \) compute \( E(y) \) by the iteration

\[
\begin{align*}
L_0 &= 0, \quad R_0 = 1, \quad Y_0 = y \\
L_{n+1} &= L_n + Y_n R_n \rightarrow E(y) \\
R_{n+1} &= (L_n + 1) + Y_n (R_n - 2) \\
Y_{n+1} &= Y_n^2
\end{align*}
\]

(36.8-18a)  

(36.8-18b)  

(36.8-18c)  

(36.8-18d)

Implementation in pari/gp:

```plaintext
ge(y, N=7)=
{
local(L, R, Lp, Rp);
/* correct up to order 2^N-1 */
L=0; R=1;
for(k=2, N,
    Lp = (L ) + y * (R );
    Rp = (L + 1) + y * (R - 2);
    L = Lp; R = Rp;
    y *= y;
)
return( L + y*R )
}
```

The symbolic representations of the polynomials \( L \) and \( R \) should make the underlying idea clear:

\[
\begin{align*}
L &= 0R = + \\
L &= 0+R = +- \\
L &= 0++R = +-- \\
L &= 0+++R = ++-- \\
L &= 0++++R = +++-- \\
L &= 0+++++R = ++++- \\
L &= 0++++++R = +++++- \\
L &= 0+++++++R = ++++++-- \\
L &= 0++++++++R = +++++++- \\
L &= 0+++++++++R = ++++++++ \\
L &= 0++++++++++R = +++++++++
\end{align*}
\]

The limit of the sequence \( L \) is entry \([A034947]\) of [245]. It is a signed version of the *paper-folding sequence.* The sequence after the initial zero is identical to the sequence of the Kronecker symbols \( (-1)^n \) for \( n = 0, 1, 2, 3, \ldots \), see section 37.8 on page 780. Quick verification:

```plaintext
? for(n=1,88,print1(if(-1==kronecker(-1,n),"-","+"))
++-++--+++--+--+++-++---++--+--+++-++--+++--+---++-++---++--+--+++-++--+++--+--+++-++---++--+---++-++--+++--+---++-++---++--+--
```

We note that the (divisionless) algorithm is fast in practice, we compute the constant \( R \) as \( R = E(1/2) \) to more than 600 thousand decimal digits:

```plaintext
? N=21 ; \ number of iterations  
? B=2^N-1 \ precision in bits  
2097151  
? D=ceil(log(2)*log(10)) + 1 \ precision in decimal digits  
631307  
? default(realprecision,D);
```

[fxtbook draft of 2008-August-17]
36.8.5 Weighted sum of Gray code digits

Define $H(y)$ by its Taylor series $H(y) = \sum_{k=1}^{\infty} h(k)y^k$ where $h(k)$ is the weighted sum of digits of the Gray code:

$$ h(k) = \sum_{n=0}^{33} \text{if (bittest(g,n),1/2^{(n+1)},0)}; $$

An iteration for the computation of $H(y)$ is:

$$ F_1 = y/2, \quad B_1 = 1/2, \quad Y_1 = y, \quad I_1 = (1 + y)/2 $$.  

Implementation in pari/gp:

```pari
gw(y, N=11)=
{ local(t, ii, F, B, Fp, Bp);
  /* correct up to order 2^N-1 */
  F=0+y; B=1+y; ii=1+y;
  for(k=2,N, ii /= 2; F /= 2; B /= 2;
    for(n=0,33, t+=if(bittest(g,n),1/2^(n+1),0));
    return( t );
  }
}
```

We define the weighted sum of Gray code digits constant as $H := H(1/2)$, then

$$ H = 0.533700488639284991588804814821242858549193225456118911 \ldots \quad (36.8-20) $$

[base 2] = 0.10001000101000010011000001100000100100000101000000111100 \ldots

[CF] = [0, 1, 1, 6, 1, 11, 4, 5, 6, 1, 13, 1, 3, 1, 18, 5, 77, 1, 2, 2, 3, 1, 2, 1, 1, \ldots]
We have:

\[
H \left( \frac{1}{q^2} \right) = \frac{q}{q+1} H \left( \frac{1}{q} \right) + \frac{q}{q-1} H \left( -\frac{1}{q} \right) \tag{36.8-21a}
\]

\[
H \left( \frac{1}{q} \right) = \frac{1}{2} \left[ \frac{q+1}{q} H \left( \frac{1}{q^2} \right) + \frac{q^2}{q^2 - q^2 + q - 1} \right] \tag{36.8-21b}
\]

\[
H \left( -\frac{1}{q} \right) = \frac{1}{2} \left[ \frac{q-1}{q} H \left( \frac{1}{q^2} \right) - \frac{q^2}{q^3 + q^2 + q + 1} \right] \tag{36.8-21c}
\]

### 36.9 A function that encodes the Hilbert curve

We define a function \( H(y) \) by the following iteration:

\[
H_1 = +i y + 1 y^2 - i y^3 \tag{36.9-1a}
\]

\[
R_1 = +i y - 1 y^2 - i y^3 \tag{36.9-1b}
\]

\[
Y_1 = y \tag{36.9-1c}
\]

\[
Y_{n+1} = Y_n^4 \tag{36.9-1d}
\]

\[
H_{n+1} = -i R_n + Y_n (+i + H_n + Y_n (+1 + H_n + Y_n (-i + i R_n))) \tag{36.9-1e}
\]

\[
R_{n+1} = +i H_n + Y_n (+i + R_n + Y_n (-1 + R_n + Y_n (-i - i H_n))) \tag{36.9-1f}
\]

\[
H_{2n} \rightarrow H(y) \tag{36.9-1g}
\]

As the real and imaginary parts are swapped with each step we agree on iterating an even number of times. The resulting function \( H(y) \) is

\[
H(y) = 0 + y + i y^2 - y^3 + i y^4 + i y^5 + y^6 - i y^7 + y^8 + i y^9 + y^{10} - i y^{11} \pm \ldots \tag{36.9-2}
\]

The coefficients of the series are \( \pm 1 \) and \( \pm i \), except for the constant term which is zero. If the sequence of coefficients is interpreted as follows:

\[
0 := \text{goto start '0'},
\]

\[
+1 := \text{move right '>',}
\]

\[
-1 := \text{move left '<'},
\]

\[
+i := \text{move up '^'}
\]

\[
-i := \text{move down 'v'}
\]

Then, symbolically:

\[
H = 0>^<^^>^v>^>vv<v>>^>v>>^<^>^<<v<^^>v>>^<^>^<<v<^<<v>vv<^<v<^^>^< ... 
\]

Follow the signs to walk along the Hilbert curve, see figure 1.20-A on page 58. An implementation is

```plaintext
1 hh(y, N=4)=
2 { /* correct to order = 4^N-1 */
3   local(H, R, th, tr);
4   H=+iy + 1*y^2 -1*I*y^3; R=+1*y - 1*y^2 -1*I*y^3;
5   for(k=2, N,
6     y=y^4;
7     th = -I*R + y* (+1 + H + y* (+1 + H + y*(-I + I*R)));
8     tr = +I*H + y* (+1 + R + y*(-1 + R + y*(-I - I*H)));
9     H=th; R=tr;
10   );
11 return( H );
12 }
```

The value of \( H(y) \) for \( y < 1 \) gives the limiting point in the complex plane when the walk according to the coefficients is done with decreasing step lengths: step number \( k \) has step length \( y^k \). The least positive \( y \) where the real and imaginary part of the endpoint are equal is \( y_1 = 0.543689016290 ... \). It turns out that \( y_1 \) is the real solution of the polynomial \( y^3 + y^2 + y - 1 \). On might suspect that \( M(y) := \Re H(y) - \Im H(y) \), the difference between the real and the imaginary part of \( H(y) \), has the

[fxtbook draft of 2008-August-17]
36.9: A function that encodes the Hilbert curve

Indeed, we have $M(y) = y(y^3 + y^2 + y - 1)(y^{12} + y^8 + y^4 - 1)\cdots$, and a similar statement is true for the $P(y) := \Re H(y) + \Im H(y)$. We use this observation for the construction of a simplified and quite elegant algorithm for the computation of $H(y)$.

### 36.9.1 A simplified algorithm

Define the function $P(y)$ as the result of the iteration

\[
Y_1 = y, \quad P_1 = 1 \\
P_{n+1} = P_n (+1 + Y_n + Y_n^2 - Y_n^3) \\
Y_{n+1} = Y_n^4 \\
P_n - 1 \rightarrow P(y)
\]

and the function $M(y)$ by

\[
Y_1 = y, \quad M_1 = 1 \\
M_{n+1} = M_n (-1 + Y_n + Y_n^2 + Y_n^3) \\
Y_{n+1} = Y_n^4 \\
y M_n \rightarrow M(y)
\]

Now the function $H(y)$ can be computed as

\[
H(y) = \frac{1}{2} \left[ (P(y) + M(y)) + i (P(y) - M(y)) \right]
\]

The following implementations compute the series up to order $4^N - 1$:

```python
1 fpp(y, N=4)=
2   \{ local( t, Y );
3       t = 1; Y=y;
4       for (k=1, N, t *= (+1+Y+Y^2-Y^3); Y=Y^4; );
5       return( t-1 );
6   \}
7 fmm(y, N=4)=
8   \{ local( t, Y );
9       t = 1; Y=y;
10      for (k=1, N, t *= (-1+Y+Y^2+Y^3); Y=Y^4; );
11     return( t*y-Y );
12   \}
13 hphp(y, N=4)=
14   \{ local( tp, tm );
15       tp = fpp(y);
16       tm = fmm(y);
17     return( ((tp+tm) + I*(tp-tm))/2 );
18   \}
```

With a routine `tdir()` that prints a power series with coefficients $\in \{-1, 0, +1\}$ symbolically we obtain:

```plaintext
? N=4;
? tdir(fpp(y));tdir(fmm(y));
0+----------+-----------------+----------
0+----------+-----------------+----------
0+----------+-----------------+----------
0+----------+-----------------+----------
0+----------+-----------------+----------
0+----------+-----------------+----------
0+----------+-----------------+----------
0+----------+-----------------+----------
```

The $n$-th coefficient of the Taylor series of $P(y)$ equals the parity of the number of threes in the radix-4 representation of $n$. This fact can be used for an efficient bit level algorithm, see section 1.20.1 on page 57.

The coefficients of the power series of the functions $P$ and $M$ can be computed with a string substitution engine, see figure 36.9-A.
Chapter 36: Synthetic Iterations

36.9.2 The turns of the Hilbert curve

We compute a function with series coefficients $\in \{-1, 0, +1\}$ that correspond to the turns of the Hilbert curve. We use $+1$ for a right turn, $-1$ for a left turn and zero for no turn. The sequences of turns starts as

0--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+--+ \ 
0++-0--++--0-++0+--+0++--++0+--00--+0++--++0+--+0++-0--++--0-++0 \ 
-++-0--++--0-++0+--+0++--++0+--00--+0++--++0+--+0++-0--++--0-++0 \ 
+--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+-- ...

The computation is slightly tricky:

```fortran
1  hht(y, N=4)=
2  { /* correct to order = 4^N-1 */
3    local( t, Y, F, s, p );
4    t = 1; Y=y; p = 1;
5    F = y + y^2;
6    for(k=2, N,
7       Y = Y^4;
8       t = -F + Y*F + Y^2*F - Y^3*F;
9       p *= 4;
10      if ( 0==(k%2),
11         t += y^(1*p-1);
12         t += y^(3*p);
13         t -= (y+1)*y^(2*p-1);
14         /* else */
15         t += y^(1*p);
16         t += y^(3*p-1);
17       );
18    F = t;
19    if ( 1==N%2, F = -F ); \ \ same result for even and odd N
20    return( F );
21  }
```

Figure 36.9-A: Computation of the power series of the functions $P$ (top) and $M$ (bottom) with a string substitution engine.
36.10: Sparse variants of the inverse

747

36.10

Sparse variants of the inverse

36.10.1

A fourth order iteration

Define the function F (y) as the result of the iteration
F0

=

1,

Y0 = y

Fn+1

= Fn (1 + Yn )

Yn+1

= Yn4

(36.10-1a)

→ F (y)

(36.10-1b)
(36.10-1c)

The terms of the continued fraction of F (1/10) grow doubly exponentially:
[1, 9, 1, 90, 110, 909000, 11001100000, 9090000090900000000000,
1100110000000000110011000000000000000000000,
90900000909000000000000000000000909000009090000000000000000000000000000000000000000000
... ]

Let [t0 , t1 , t2 , . . .] be the continued fraction of F (1/q), then
t0

=

1

(36.10-2a)

t1

= q−1

(36.10-2b)

t2

=

(36.10-2c)

1
2

= q −q

t3

(36.10-2d)

2

t4

= q +q
6

(36.10-2e)
5

4

3

t5

= q −q +q −q

t6

= q 10 + q 9 + q 6 + q 5

t7

= q 22 − q 21 + q 20 − q 19 + q 14 − q 13 + q 12 − q 11

t8

= q

42

+q

86

41
85

+q

38

+q

84

(36.10-2f)
(36.10-2g)
37

+q

83

26

+q

78

25

+q

77

22
76

+q

(36.10-2h)

21

(36.10-2i)

75

(36.10-2j)

= q −q +q −q +q −q +q −q +
+q 54 − q 53 + q 52 − q 51 + q 46 − q 45 + q 44 − q 43

t9

For j ≥ 4 we have
tj
tj−2

= q 2J + q J = q J (q J + 1)

where

J = 2j−4

(36.10-3)

A functional relation for F is given by
F (y) F (y 2 )

=

1
1−y

(36.10-4)

The relation is (mutatis mutandis) also true for the truncated product. The binary expansions of F (y),
F (y 2 ) and their product for y = 1/2 are (dots for zeros):
11..11..........11..11..............................................
1.1.....1.1.....................1.1.....1.1.........................
1111111111111111111111111111111111111111111111111111111111111111....
Since the expansions are palindromes also their correlation is a sequence of ones. Now set Fk := F (y k )
then, by relations 36.10-4 and 36.1-16e on page 726 (for k = 2, 3, and 5),
0

= F12 F2 − 2 F1 F2 F4 + F4

0

= F13 F23 − F3 F6 3 F12 F22 − 3 F1 F2 + 1

0

= F15 F25 − F10 F5 5 F14 F24 − 10 F13 F23 + 10 F12 F22 − 5 F1 F2 + 1

(36.10-5a)


(36.10-5b)


(36.10-5c)

We
note that for power series over GF(2) relation 36.10-4 becomes F (y)3 = 1/(1 − y). That is, F (y) =
√
3
1 − y. In general, an iteration for the inverse (2w − 1)-st root is obtained by replacing relation 36.10-1c
w
with Yk+1 = Yk2 .
[fxtbook draft of 2008-August-17]


36.10.2 A different fourth order iteration

To define the function \( F(y) \) we modify the third order iteration for \( \frac{1}{1+y} \)

1 \texttt{inv3m(y, N=6)= /* third order --> 1/(1+y) */}
2 { /* correct to order 3^N */
3 local(T);
4 \texttt{T = 1;}
5 \texttt{for(k=1, N,}
6 \texttt{T *= ( 1 - y + y^2 );}
7 \texttt{y = y^3;}
8 \texttt{)}
9 \texttt{return( T );}
10 }

to obtain a fourth-order iteration:

1 \texttt{f43(y, N=6)=
2 { /* correct to order 4^N */
3 local(T, yt);
4 \texttt{T = 1;}
5 \texttt{for(k=1, N,}
6 \texttt{T *= ( 1 - y + y^2 );}
7 \texttt{y = y^4; /* ! */}
8 \texttt{)}
9 \texttt{return( T );}
10 }

That is,

\[
F_0 = 1, \quad Y_0 = y \quad (36.10-6a)
\]
\[
F_{n+1} = F_n (1 - Y_n + Y_n^2) \rightarrow F(y) \quad (36.10-6b)
\]
\[
Y_{n+1} = Y_n^4 \quad (36.10-6c)
\]

The first few terms of the power series are

\[
F(y) = 1 - y + y^2 - y^4 + y^5 - y^6 + y^8 - y^9 + y^{10} - y^{16} + y^{17} - y^{18} + y^{20} - y^{21} \pm \ldots \quad (36.10-7)
\]

Let \([t_0, t_1, t_2, \ldots]\) be the continued fraction of \( F(1/q) \), then

\[
t_0 = 0 \quad (36.10-8a)
\]
\[
t_1 = 1 \quad (36.10-8b)
\]
\[
t_2 = q \quad (36.10-8c)
\]
\[
t_3 = q \quad (36.10-8d)
\]
\[
t_4 = q^2 - q \quad (36.10-8e)
\]
\[
t_5 = q^4 + q^3 - q - 1 \quad (36.10-8f)
\]
\[
t_6 = q^8 - q^7 + q^6 - q^4 + q^3 - q^2 \quad (36.10-8g)
\]
\[
t_7 = q^{16} + q^{15} - q^{13} + q^{11} + q^{10} - q^8 - q^7 + q^5 - q^3 - q^2 \quad (36.10-8h)
\]

For \( j \geq 6 \) we have

\[
\frac{t_j}{t_{j-2}} = q^6 + q^4j + q^3j + q^j = q^j (q^j + 1) (q^{2j} - q^j + 1) (q^{2j} + 1) \quad (36.10-9)
\]

where \( J = 2^j - 6 \). The terms of the continued fraction of \( F(1/q) \) for integer \( q \) grow doubly exponentially:

\[
? \texttt{contfrac(f43(0.5))}
\]
\[ [0, 1, 2, 2, 2, 21, 180, 92820, 3032435520, 26126907554432456580, 24024294248527099500117907463345274880, 164001265750215347067944129734442019102853751066678216639025390799096507269120, ... ] \]

/* number of decimal digits of the terms in the CF: */

[*, 1, 1, 1, 1, 2, 3, 5, 10, 20, 39, 78, 154, 309, ... ]
By construction,

\[ F(y) = (1 - y + y^2) F(y^4) \]  
\[ F(-y) = (1 + y + y^2) F(y^4) \]  
(36.10-10a)

The equivalent forms with \( y = 1/q \) are

\[ F \left( \frac{1}{q} \right) = \frac{q^2 - q + 1}{q^2} F \left( \frac{1}{q^4} \right) \]  
(36.10-11a)

\[ F \left( -\frac{1}{q} \right) = \frac{q^2 + q + 1}{q^2} F \left( \frac{1}{q^4} \right) \]  
(36.10-11b)

Now \( q^2 - q + 1 = p^2 - p + 1 \) if \( p = 1 - q \), so

\[ F \left( 1 - \frac{1}{q} \right) = \frac{q^2 - q + 1}{q^2} F \left( \left( 1 - \frac{1}{q} \right)^4 \right) \text{ where } q > 1 \]  
(36.10-12a)

\[ F \left( 1 + \frac{1}{q} \right) = \frac{q^2 + q + 1}{q^2} F \left( \left( 1 + \frac{1}{q} \right)^4 \right) \text{ where } q < -1 \]  
(36.10-12b)

Adding relations \( \alpha \times \) (36.10-11a) and \( \beta \times \) (36.10-12a) and simplifying gives

\[ \frac{\alpha F(y) + \beta F(1 - y)}{\alpha F(y^4) + \beta F((1 - y)^4)} = y^2 - y + 1 \text{ where } \alpha, \beta \in \mathbb{C} \]  
(36.10-13)

### 36.10.3 A sixth order iteration

Define the function \( F(y) \) by the iteration

\[ F_0 = 1, \quad Y_0 = y \]  
(36.10-14a)

\[ F_{n+1} = F_n \left( 1 + Y_n + Y_n^2 \right) \rightarrow F(y) \]  
(36.10-14b)

\[ Y_{n+1} = Y_n^6 \]  
(36.10-14c)

Let \([t_0, t_1, t_2, \ldots]\) be the continued fraction of \( F(1/q) \), then

\[ t_0 = 1 \]  
(36.10-15a)

\[ t_1 = q - 1 \]  
(36.10-15b)

\[ t_2 = q + 1 \]  
(36.10-15c)

\[ t_3 = q^2 - q \]  
(36.10-15d)

\[ t_4 = q^{10} + q^9 + q^8 + q^4 + q^3 + q^2 \]  
(36.10-15e)

\[ t_5 = q^8 - q^7 + q^5 - q^4 \]  
(36.10-15f)

\[ t_6 = q^{64} + q^{63} + q^{62} + q^{58} + q^{57} + q^{56} + q^{52} + q^{51} + q^{50} + \]  
\[ + q^{28} + q^{27} + q^{26} + q^{22} + q^{21} + q^{20} + q^{16} + q^{15} + q^{14} \]  
(36.10-15g)

\[ t_7 = q^{44} - q^{43} + q^{41} - q^{40} + q^{26} - q^{25} + q^{23} - q^{22} \]  
(36.10-15h)

\[ t_8 = q^{388} + q^{387} + \ldots + q^{87} + q^{86} \]  
(36.10-15i)

For \( j \geq 4 \) we have

\[ \frac{t_j}{t_{j-2}} = \begin{cases} 
q^j + q^{j/2} \\
(q^{10j} + q^{9j} + q^8j + q^4j + q^3j + q^2j) / (q^j + 1) 
\end{cases} \text{ if } j \text{ odd} \]  
\begin{cases} 
\text{else} 
\end{cases}  
\]  
(36.10-16)

where \( J = 6^{j-4} \).
36.11 An iteration related to the Fibonacci numbers

The so-called *rabbit constant* is

\[ A = 0.750034428612913146417873994445755970125022057678605169 \ldots \] (36.11-1)

\[ \text{[base 2]} = 0.101101011011010110101101101011011010110101 \ldots \]

\[ \text{[CF]} = [0, 1, 2, 4, 8, 32, 256, 8192, 2097152, 17179869184, 36028797018963968, 618970019642690137449562112, \ldots] \]

The sequence of zeros and ones after the decimal point is referred to as *rabbit sequence* or *infinite Fibonacci word*, entry A005614 of [245]. The rabbit sequence can be obtained by starting with a single zero and repeated application of the following substitution rules: simultaneously replace all zeros by one (0 → 1, ‘young rabbit gets old’) and all ones by one-zero (1 → 10, ‘old rabbit gets child’), no sex, no death. The evolution is:

\[
\begin{align*}
A_0 &= 0 \\
A_1 &= 1 \\
A_2 &= 10 \\
A_3 &= 101 \\
A_4 &= 10110 \\
A_5 &= 10110101 \\
A_6 &= 1011010110110 \\
A_7 &= 10110101101101011010110110101101 \ldots
\end{align*}
\]

The crucial observation is that each element \( A_n \) can be obtained by appending \( A_{n-2} \) to \( A_{n-1} \), that is

\[
A_n = A_{n-1} A_{n-2}.
\]

To compute the value of the rabbit constant in base 2 to \( N \) bits precision the whole process requires only copying \( N \) bits of data is the minimal conceivable work for a (non-sparse) computation.

We define a function \( A(y) \) that has the special value

\[
A = \frac{1}{2} A \left( \frac{1}{2} \right)
\]

by the equivalent operation for power series. We obtain the iteration

\[
\begin{align*}
L_0 &= 0 & R_0 &= 1 & l_0 &= 1 & r_0 &= y \\
L_{n+1} &= r_n = yF_n \\
L_{n+1} &= r_n l_n = yF_{n+1} \\
L_{n+1} &= R_n \\
R_{n+1} &= R_n + r_{n+1} L_n = R_n + yF_{n+1} L_n \rightarrow A(y)
\end{align*}
\]

Here \( F_n \) denotes the \( n \)-th Fibonacci number (sequence [A000045 of [245]):

\[
F_n = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots
\]

After the \( n \)-th step the series in \( y \) is correct up to order \( F_{n+2} - 1 \). That is, the order of convergence equals \( \frac{\sqrt{5} + 1}{2} \approx 1.6180 \). The function \( A(y) \) has the power series

\[
A(y) = 1 + y^2 + y^3 + y^5 + y^7 + y^8 + y^{10} + y^{11} + y^{13} + y^{15} + y^{16} + y^{18} + y^{20} + y^{21} \ldots
\]

The sequence of exponents of \( y \) in the series is entry A022342 of [245], the Fibonacci-even numbers. A pari/gp implementation is

```plaintext
1 \ fa(y, N=10)= 2 \ { local(t, yl, yr, L, R, Lp, Rp); 3 \n/* correct up to order fib(N+2)-1 */ 4 \ 5 \ L=0; R=1; yl=1; yr=y; 6 \ for(k=1, N,
```
36.11: An iteration related to the Fibonacci numbers

```c
36.11.1 Fibonacci representation

```

Figure 36.11-A: Fibonacci representations of the numbers 0...80. A dot is used for zero. The two lower lines are the sum of digits and the sum of digits modulo two, the Fibonacci parity.

The greedy algorithm to obtain the Fibonacci representation (or Zeckendorf representation) of an integer repeatedly subtracts the largest Fibonacci number that is greater or equal to it until the number is zero. The Fibonacci representations of the numbers 0...80 are shown in figure 36.11-A.

The sequence of lowest Fibonacci bits is

```
01001010010010100101001001010010010100101001001010010100100101001...
```

Interpreted as the binary number

```
x = 0.1001010010010100101001001010010010100101001001010010100100101001...
```

It turns out that

```
A = 1 - x/2 (that is, x = 2 - A(1/2)). Alternatively, one can compute the number as x = A[1, 0, 1, 1/2].
```

The sequence of least significant bits in the Fibonacci representations is entry A003849 of [245]. The sequence of numbers of digits in the Fibonacci representations (second lowest row in figure 36.11-A) is entry A007895 of [245]. This sequence modulo two gives the Fibonacci parity (entry A095076). It can be computed by initializing

```
L_0 = 1 and changing relation 36.11-3e to
```

```
R_{n+1} = R_n - r_{n+1} L_n = R_n - y^{F_{n+1}} L_n \rightarrow A_p(y) \quad (36.11-5)
```

Let the corresponding function be \( A_p(y) \). We define the Fibonacci parity constant \( A_p \) as

```
A_p = 1 - A_p(1/2)/2 \quad (36.11-6a)
```

```
= 0.9105334708635617638046868867710980073445812290069376454... \quad (36.11-6b)
```

[base 2] = 0.111010010001001100100101101100010111101001001001001001001001001...

[CF] = [0, 1, 0, 1, 5, 1, 1, 1, 3, 4, 2, 6, 25, 4, 5, 1, 1, 3, 5, 1, 3, 2, 1, 1, 1, 3, 1, 3, 22, 1, 10, 1, 2, 3, 2, 73, 1, 111, 46, 1, 51, 2, 1, 1, 5, 1, 65, 3, 1, 3, 2, 5, 6, 1, 4, 1, 2, ...]

The sequence of the Fibonacci representations interpreted as binary numbers is

```
0, 1, 2, 4, 5, 8, 9, 10, 16, 17, 18, 20, 21, 32, 33, 34, 36, 37, 40, 41, 42, 64, 65, 66, 68, 69, 72, 73, 74, 80, 81, 82, 84, 85, 128, 129, ...
```

This is entry A003714 of [245], where the numbers are called Fibbinary numbers. Define \( F_2(y) \) to be the function that has the same sequence of power series coefficients:

```
F(y) = 0 + 1 y + 2 y^2 + 4 y^3 + 5 y^4 + 8 y^5 + 9 y^6 + 10 y^7 + 16 y^8 + 17 y^9 + 18 y^{10} + ... \quad (36.11-7)
```

```c
A slightly more general function \( F_b(y) \) (which for \( b = 2 \) gives the power series above) can be computed by the iteration

\[
\begin{align*}
L_0 &= 0, \quad R_0 = y, \quad l_0 = y, \quad r_0 = y \quad (36.11-8a) \\
A_0 &= 1, \quad B_0 = 1, \quad b = 2 \quad (36.11-8b) \\
A_{n+1} &= b B_n \quad (36.11-8c) \\
B_{n+1} &= b [B_n + r_n A_n] \quad (36.11-8d) \\
l_{n+1} &= r_n = y F_n+1 \quad (36.11-8e) \\
\quad r_{n+1} &= r_n l_n = y F_{n+1} \quad (36.11-8f) \\
L_{n+1} &= R_n \quad (36.11-8g) \\
R_{n+1} &= R_n + r_{n+1} [L_n + A_{n+1}] \rightarrow F_b(y) \quad (36.11-8h)
\end{align*}
\]

A pari/gp implementation is

```plaintext
1 fibb(y, b=2, N=13)=
2 { /* correct up to order fib(N+3)-1 */
3   local(t, yl, yr, L, R, Lp, Rp, Ri, Li);
4     L=0; R=0+1*y;
5     Li=1; Ri=1;
6     yl=y; yr=y;
7     for (k=1, N,
8       Li*=b; Ri*=b;
9       Lp=Ri; Rp=Ri+yr*Li; Li=Lp; Ri=Rp;
10      t=yr; yr*=yl; yl=t;
11      Lp=R; Rp=R+yr*(L+Li); L=Lp; R=Rp;
12    );
13   return( R )
14 }
```

The sequence of coefficients

1, 6, 14, 35, 90, 234, 611, 1598, 4182, 10947, 28658, 75026, 196419, 514230, ... coincides (disregarding the initial one) with entries \text{A032908} and \text{A093467} of [245]. Let \( B(x) \) be the function with power series coefficients equal to one if the exponent is a Fibonacci number and zero else:

\[
B(x) := 1 + x + x^2 + x^4 + x^5 + x^8 + x^9 + x^{10} + x^{16} + x^{17} + x^{18} + x^{20} + \ldots \quad (36.11-9)
\]

Then a functional equation for \( B(x) \) is (see entry \text{A003714} of [245])

\[
B(x) = x B(x^4) + B(x^2) \quad (36.11-10)
\]

We turn the relation into an recursion for the computation of \( B(x) \) correct up to the term \( x^N \):

```plaintext
1 fibbi(z, R)=
2 {  
3   if ( z+R==0, return( 1+R ) );
4   return( z*fibbi(z^4,R) + fibbi(z^2,R) );
5 }
```

We check the functional relation:

```plaintext
? N=30; R=O(x^(N+1)); \ \ \ // R is used to truncate terms of order >N
? t=fibbi(x,R)
1 + x + x^2 + x^4 + x^5 + x^8 + x^9 + x^{10} + x^{16} + x^{17} + x^{18} + x^{20} + x^{21} + O(x^{31})
? t2=fibbi(x^2,R)
1 + x^2 + x^4 + x^8 + x^{10} + x^{16} + x^{18} + x^{20} + O(x^{31})
? t4=x*fibbi(x^4,R)
x + x^5 + x^9 + x^{17} + x^{21} + O(x^{32})
? t-(t4+t2)
O(x^{31})
```

### 36.11.2 Digit extract algorithms for the rabbit constant

The spectrum of a real number \( x \) is the sequence of integers \( \lfloor k \cdot x \rfloor \) where \( k = 1, 2, 3, \ldots \). As mentioned in [273], the spectrum of the golden ratio \( g = \frac{\sqrt{5}+1}{2} = 1.61803 \ldots \) gives the exponents of \( y \) where the series for \( y A(y) \) has coefficient one.
An iteration related to the Fibonacci numbers

```plaintext
bt(x, n=25)=
    { local(v);
      v = vector(n);
      for (k=1, n, v[k]=floor(x*k));
      return ( v );
    }

g=(sqrt(5)+1)/2
1.618033988749894848204586834365638117720309179805762862

n=40;
bt(g, n)
[1, 3, 4, 6, 8, 9, 11, 12, 14, 16, 17, 19, 21, 22, 24, 25, 27, 29, 
  30, 32, 33, 35, 37, 38, 40, 42, 43, 45, 46, 48, 50, 51, 53, 55, 56, 
  58, 59, 61, 63, 64]

t=taylor(y*fa(y),y)
y + y^3 + y^4 + y^6 + y^8 + y^9 + y^11 + y^12 + y^14 + y^16 + y^17 + 
  y^19 + y^21 + y^22 + y^24 + y^25 + y^27 + y^29 + y^30 + y^32 + y^33 + 
  y^35 + y^37 + y^38 + y^40 + y^42 + y^43 + y^45 + y^46 + y^48 + y^50 + 
  y^51 + y^53 + y^55 + y^56 + y^58 + y^59 + y^61 + y^63 + y^64 + O(y^66)

The sequence [1, 3, 4, 6, ...] of exponents where the coefficient equals one is sequence A000201 of [245].

There is a digit extract algorithm for the binary expansion of the rabbit constant. We use a binary search algorithm:

```plaintext
bts(x, k)=
    { /* return 0 if k is not in the spectrum of x, else return index >=1 */
      local(nlo, nhi, t);
      if ( 0==k, return(0) );
      t = 1 + ceil(k/x); \ floor(t*x)>=k
      nlo = 1; nhi = t;
      while ( nlo!=nhi,
        t = floor( (nlo+nhi)/2 );
        if ( floor(t*x) < k, nlo=t+1, nhi=t);
      );
      if ( floor(nhi*x) == k, return(nhi), return (0));
    }

g=(sqrt(5)+1)/2
for(k=1,65,if(bts(g,k),print1("1"),print1("0")));print();
10110101101101101101101101101111011011011011011011011011011011011011011011
--snip--
*** last result computed in 236 ms.
```

The connection between the sequence of lowest Fibonacci bits and the rabbit constants allows even more. Subtracting the Fibonacci numbers >1 until zero or one is reached, gives the complement of the rabbit sequence:

```plaintext
fpn=999;
vpv=vector(fpn, j, fibonacci(j+2)); /* vpv=[2,3,5,8,...] */
t=vpv[length(vpv)]; /* log(t)/log(10)== 208.8471. OK for range up to >10^200 (!) */
flb(x)=
    { /* return the lowest bit of the Fibonacci representation */
      local(k, t);
      k=bsearchgeq(x, vpv);
      while ( k>0,
        t = vpv[k];
        if (x>=t, x-=t);
        k-- );
      return ( x );
    }

dd=0;
for(k=dd,dd+40,t=flb(k);print1(1-t)
10110101101101101101101101101111011011011011011011011011011011011011011011
```

[fxtbook draft of 2008-August-17]
The routine \texttt{bsearchgeq()} does a binary search (see section 3.2 on page 123) for the first element that is greater or equal to the element sought:

```plaintext
bsearchgeq(x, v)=
{ /* return index of first element in v[] that is >=x, return 0 if x>max(v[]) */
  local(nlo, nhi, t);
  nlo = 1; nhi = length(v);
  while ( nlo!=nhi, t = floor((nlo+nhi)/2 );
    if ( v[t] < x, nlo=t+1, nhi=t);
  );
  if ( v[nhi] >= x, return(nhi), return (0));
}
```

We compute the first 1000 bits starting from position 10$^{100}$:

```plaintext
dd=10^100-1;
for (k=dd, dd+1000, t=flb(k); print1(1-t))
```

---

### 36.12 Iterations related to the Pell numbers

Replacement rules: simultaneously replace all zeros by one (0 → 1) and all ones by one-one-zero (1 → 110).

```
B0 = 0
B1 = 1
B2 = 110
B3 = 1101101
B4 = 11011011101101110

Bn\rightarrow 11011011101101110110110111011011101101101
```

Now the construction is $B_n = B_{n-1} B_{n-2}$. The length of the $n$-th string is

$$p_n = 1, 1, 3, 7, 17, 41, 99, 239, \ldots$$

This sequence is entry \texttt{A001333} of \cite{245}, the numerators of the continued fraction of $\sqrt{2}$. The sequence $B$ of zeros and ones is entry \texttt{A080764}. The Pell numbers are the first differences (and the denominators of the continued fraction of $\sqrt{2}$), sequence \texttt{A000129}:

```
0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, \ldots
```

Now define the function $B(y)$ by the iteration

\begin{align*}
L_0 &= 1, & R_0 &= 1 + y, & l_0 &= y, & r_0 &= y \quad \text{(36.12-1a)} \\
l_{n+1} &= r_n, & l_0 &= y, & r_0 &= y \quad \text{(36.12-1b)} \\
r_{n+1} &= r_n^2, \quad \text{(36.12-1c)} \\
L_n &= R_n \quad \text{(36.12-1d)} \\
R_{n+1} &= R_n + r_{n+1} R_n + r_{n+1}^2 L_n \rightarrow B(y) \quad \text{(36.12-1e)}
\end{align*}

After the $n$-th step the series in $y$ is correct up to order $p_n$. That is, the order of convergence is $\sqrt{2} + 1 \approx 2.4142$. Implementation in pari/gp:

```plaintext
fb(y, N=8)=
{ local(t, yr, yl, L, R, Lp, Rp);
  L=1; R=1+y; yl=y; yr=y;
  for(k=1,N, t=yr; yr=yr*yl; yl=t;
    Lp=R; Rp=R+yr*R+y^2*L; L=Lp; R=Rp;
  );
  return( R )
}
```
36.12: Iterations related to the Pell numbers

755

We obtain the series
B(y)

=

1 + y + y 3 + y 4 + y 6 + y 7 + y 8 + y 10 + y 11 + y 13 + y 14 + y 15 + y 17 + y 18 + y 20 + . . .
1
2

Define the Pell constant B as B =

(36.12-2)

B( 12 ), then

B

=

0.8582676564610020557922603084333751486649051900835067786 . . .

[base 2]

=

0.11011011101101110110110111011011101101101110110111011011 . . .

[CF]

=

(36.12-3)

[0, 1, 6, 18, 1032, 16777344, 288230376151842816,
1393796574908163946345982392042721617379328, . . .]

For the terms of the continued fraction we note
6
18
1032
16777344
288230376151842816
1393796574908163946345982392042721617379328

36.12.1

==
2^2
==
2^4
== 2^10
== 2^24
== 2^58
== 2^140

+ 2^1
+ 2^1
+ 2^3
+ 2^7
+ 2^17
+ 2^41

Pell palindromes

Define the function P (y) by
L0

=

R0 = 1 + y 2 ,

1,

l0 = y,

r0 = y

(36.12-4a)

ln+1

= rn

(36.12-4b)

rn+1

= rn2 ln

(36.12-4c)

Ln+1

= Rn

(36.12-4d)

Rn+1

= Rn + rn+1 Ln + rn+1 ln+1 Rn

→ P (y)

(36.12-4e)

Note that R0 is a palindrome and in relation 36.12-4e the combination of the parts gives a palindrome.
1
For R0 = 1 + y + y 2 the iteration computes 1−y
.
Define the Pell palindromic constant as P = P (1/2)/2, then
P

=

0.7321604330635328371645901871773044657272986589604112390 . . .

[base 2]

=

0.1011101101101110110111011011101101101110110111011011011 . . .

[CF]

=

(36.12-5)

[0, 1, 2, 1, 2, 1, 3, 17, 1, 7, 2063, 1, 63, 268437503, 1, 8191, 590295810358974087167,
1, 1073741823, 374144419156711147060143317175958748842277436653567, 1, . . .]

By construction, the binary expansion is a palindrome up to lengths 1, 3, 7, 17, 41, 99, 239, . . ..
The sequence of zeros and ones in the binary expansion of P is entry A104521 of [245]. It can be obtained
by the replacement rules (0 → 1) and (1 → 101):
P0 =
P1 =
P2 =
P3 =
P4 =
P5 =
P -->

36.12.2

0
1
101
1011101
10111011011011101
10111011011011101101110110111011011011101
1011101101101110110111011011101101101110110111011011011101101...

Pell representation

To obtain the Pell representation of a given number n, set t = n repeatedly subtract from t the largest
number pk ∈ {1, 3, 7, 17, 41, 99, 239, . . .} that is not greater than t. Stop when t = 0. The number of
times that pk has been subtracted gives the k-th digit of the representation. The resulting digits are 0,
1, or 2. If the k-th digit equals 2 then the (k − 1)-th digit will be zero.
[fxtbook draft of 2008-August-17]


The function $S(y)$ that has Taylor series coefficients corresponding to the sum of Pell digits

$$S(y) = 0 + 1y + 2y^2 + 1y^3 + 2y^4 + 3y^5 + 2y^6 + 1y^7 + 2y^8 + 3y^9 + 2y^{10} + 3y^{11} + 4y^{12} + \ldots$$  \hspace{1cm} (36.12-6)$$
can be computed via the iteration (see section \[ \text{36.7} \] on page 736)

\begin{align*}
L_0 &= 0, \quad R_0 = 0 + y + 2y^2, \quad l_0 = y, \quad r_0 = y \quad \text{(36.12-7a)} \\
A_0 &= 1, \quad B_0 = 1 + y + y^2 \quad \text{(36.12-7b)} \\
l_{n+1} &= r_n, \quad r_{n+1} = r_n^2 l_n \quad \text{(36.12-7c)} \\
L_{n+1} &= R_n \quad \text{(36.12-7d)} \\
R_{n+1} &= R_n + r_{n+1} (R_n + B_n) + r_{n+1}^2 (L_n + 2A_n) \quad \rightarrow S(y) \quad \text{(36.12-7f)} \\
A_{n+1} &= B_n \quad \text{(36.12-7g)} \\
B_{n+1} &= B_n + r_{n+1} B_n + r_{n+1}^2 A_n \quad \text{(36.12-7h)} \\
\end{align*}

Implementation in pari/gp:

```plaintext
1 fs(y, N=8)=
2 { local(t, yr, yl, L, R, Lp, Rp, Li, Ri);
3 L = 0; R = 0+y+2*y^2;
4 Li = 1; Ri = 1+y+y^2;
5 yl = y; yr = y;
6 for(k=1,N,
7     t=yr; yr=yr*yl; yl=t;
8     Lp=R; Rp=R+yr*(R+Ri)+yr^2*(L+2*Li); L=Lp; R=Rp;
9     Lp=Li; Rp=Li+yr*Ri+yr^2*Li; Li=Lp; Ri=Rp;
10    );
11   return( R )
12 }
```

The series coefficients grow slowly, so the first few of them can nicely be displayed as

$$S\left(\frac{1}{10}\right) = 0.1213212323432412324323434545234345412324323434543\ldots$$  \hspace{1cm} (36.12-8)$$

### 36.12.3 Pell Gray code

Figure \[ \text{36.12-B} \] gives a Gray code for the Pell representations. The Gray code can be constructed recursively as shown in figure \[ \text{36.12-C} \]. In the algorithm each block is split into a left and a right part (indicated by the ‘‘^’’. The next block is created by appending to the current block its reverse with ones on top and appending the left part with twos on top. The iteration can actually be started with a block of a single
36.12: Iterations related to the Pell numbers

Figure 36.12-B: A Gray code for Pell representations. A dot is used for zero. The three following lines are the sum of digits and the sum of digits modulo three and two. The sequence is $0, 1, 2, 5, 4, 3, 6, 10, 11, 12, 9, 8, 7, 13, ..., \ldots$, the difference between successive elements is a Pell number. The lowest block gives the Pell representations of the (absolute) differences, the Pell ruler function.

Figure 36.12-C: Construction for a Gray code for Pell representations.

zero (the left part being also a single zero). This is done in the following algorithm.

\[
\begin{align*}
F_0 &= 0, & F'_0 &= 0, & B_0 &= 0, & B'_0 &= 0, \\
I_0 &= 1, & I'_0 &= 1, & Y_0 &= y, & Y'_0 &= y. \\
\end{align*}
\]

(36.12-9a)

(36.12-9b)

(36.12-9c)

(36.12-9d)

(36.12-9e)

(36.12-9f)

(36.12-9g)

(36.12-9h)

(36.12-9i)

Implementation in pari/gp:

```
1  pgr(y, N=11)= /* Pell Gray code */
2  {
3    local(iir, iil, yl, yr, Fl, F, Bl, B, b, c);
4    local(t, tf, tb);
5    /* correct up to order pell(N+1)-1 */
6    F=0; Fl=0; B=0; Bl=0;
7    iil=1; iir=1; yl=y; yr=y;
8    for(k=1, N,
9       b = 4^*(k-1); c = 2*b; /* b = pell(k);*/
10      tf = (F ) + yr * (B + b*iir) + yr*yr * (Fl + c*iil);
11      tb = (Bl + c*iil) + yl * (F + b*iir) + yl*yr * (B );
12      Fl = F; Fl = Fl; B = B;
13      F = tf; F = F; B = B;
14      t = iir; iir += yr*(iir + yr*iil); iil = t;
15      t = y; yr *= (yr*yl); yl = t;
16    )
```
Define the Pell Gray code constant

Section 12.4 on page 306 gives a recursive algorithm to compute the words of the Pell Gray code. When relation 36.12-9c is changed to $b$ occupying two bits (figure 36.12-B). The coefficients corresponds to the Pell representations interpreted as binary numbers, each Pell-digit $pell(k) = \begin{cases} 1, & k \leq 1 \\ 2 \cdot pell(k-1) + pell(k-2), & \text{otherwise} \end{cases}$

Pell Gray code sequence as coefficients is computed: $G$

The Taylor series for $G(y)$ is

It is instructive to look at the variables in the first few steps of the iteration, see figure 36.12-D.

The coefficients corresponds to the Pell representations interpreted as binary numbers, each Pell-digit occupying two bits (figure 36.12-B).

When relation 36.12-9c is changed to $b_n = P_n$ (indicated in the code, the function can be defined as $pell(k) = 1 \text{ if } k = 1, 1, \text{ return}(2 \cdot pell(k-1) + pell(k-2));$) then the function $G_P(y)$ which has the Pell Gray code sequence as coefficients is computed:

Section 12.4 on page 306 gives a recursive algorithm to compute the words of the Pell Gray code.

Define the Pell Gray code constant as

$$G_P = G_P \left( \frac{1}{2} \right)$$  \hspace{1cm} (36.12-1a)

$$= 2.245567348365072195720956572438998819867495229140192012 \ldots$$ \hspace{1cm} (36.12-1b)

$$[\text{base } 2] = 10.001111101101101100000001110010011000110000101000111 \ldots$$ \hspace{1cm} (36.12-1c)

$$[\text{CF}] = [2, 4, 13, 1, 5, 1, 1, 27, 1, 9, 1, 3, 8, 1, 2, 1, 1, 3, 14, 1, 8, 1, 1, 6, 3, 1, 1, 1, 2, 1, 7, 210, 1, 1, 3, 2, 1, 1, 10, 1, 1, 6, 1, 1, 2, 1, 2, 1, 4, 6, 12, 1, \ldots]$$
Setting $b_n = 1$, $c_n = 2$ in the algorithm gives a function whose series coefficients are the sum of Pell Gray code digits:

$$G_{[1,2]} \left( \frac{1}{10} \right) = 0.12321232343212345434323421234545654323454345 \ldots \quad (36.12-13)$$

Using $b_n = 1$, $c_n = 0$ counts the ones in the Pell Gray code:

$$G_{[1,0]} \left( \frac{1}{10} \right) = 0.101210123212101232343212101232123234323 \ldots \quad (36.12-14)$$

while $b_n = 0$, $c_n = 1$ counts the twos:

$$G_{[0,1]} \left( \frac{1}{10} \right) = 0.01100110011221100110011001122112211201100110 \ldots \quad (36.12-15)$$

The continued fraction of this constant has very large terms:

$$[0, 90, 1, 8, 1, 5501100164, 8, 5, 3, 2, 19, 2, 1, 2, 2, 1, 1, 5, 1, 54, 6, 1, 5, 22, 2, 6, 2, 2, 1, 22445, 1, 45, 2, 2, 5, 1, 5, 1, 8, 5446094555446094447167274, 1, 5, 2, 2, 7, 1, 1, 1, 2, 1, 27, 2, 2, 1, 1, 17, 1, 1, 1, 4, 2, 1, 4, 3, 3, 3, 1, 3, 1, 2, 1, 1, 29, 1, \ldots]$$

Finally, $(1 - y)G_P(y)$ gives the (signed) Pell ruler function, the first 100 series coefficients are:

0 +1 +1 +3 -1 -1 +3 +7 -3 +1 +1 -3 -1 -1 +7 +1 +1 +17 -1 -1 -7 +1 +1 +3 -1 -1 +3 +7 -3 +1 +1 -3 -1 -1 +7 +1 +1 -17 -1
-1 -7 +1 +1 +3 -1 -1 +3 -7 -3 +1 +1 -3 -1 -1 +41 +1 +1 +3 -1 -1 +3 +7 -3 +1 +1 -3 -1 -1 +7 +1 +1 -17 -1
-7 -3 +1 +1 -3 -1 -1 +7 +1 +1 +99 -1
Part V

Algorithms for finite fields
Chapter 37

Modular arithmetic and some number theory

We implement the arithmetical operations modulo \( m \), such as addition, subtraction, multiplication, and division. Basic concepts of number theory, like the order of an element, quadratic residues, and primitive roots are developed. Selected algorithms such as the Rabin-Miller compositeness test and several primality tests are presented. Finally we give the Cayley-Dickson construction for hypercomplex numbers and compute their multiplication tables.

Modular arithmetic and the concepts of number theory are fundamental for many areas like cryptography, error correcting codes, and digital signal processing.

37.1 Implementation of the arithmetic operations

We implement the basic operations of modular arithmetic: addition, subtraction, multiplication, powering, inversion and division.

37.1.1 Addition and subtraction

Addition and subtraction modulo \( m \) can easily be implemented as [FXT: mod/modarith.h]:

```c
inline umod_t sub_mod(umod_t a, umod_t b, umod_t m) {
    if ( a>=b ) return a - b;
    else return m - b + a;
}
```

The type \( \text{umod}_t \) is an unsigned 64-bit integer. Care has been taken to avoid any overflow of intermediate results. A 'set', increment-, decrement- and negation function will further be useful:

```c
inline umod_t add_mod(umod_t a, umod_t b, umod_t m) {
    if ( 0==b ) return a;
    // return sub_mod(a, m-b, m);
    b = m - b;
    if ( a>=b ) return a - b;
    else return m - b + a;
}
```

The type \( \text{umod}_t \) is an unsigned 64-bit integer. Care has been taken to avoid any overflow of intermediate results. A 'set', increment-, decrement- and negation function will further be useful:

```c
inline umod_t set_mod(umod_t x, umod_t m) {
    if ( x>=m ) x %= m; return x;
}
```

```c
inline umod_t incr_mod(umod_t a, umod_t m) {
    a++; if ( a==m ) a = 0; return a;
}
```


Chapter 37: Modular arithmetic and some number theory

inline umod_t decr_mod(umod_t a, umod_t m)
{
    if ( a==0 ) a = m - 1; else a--; return a;
}

inline umod_t neg_mod(umod_t b, umod_t m)
{
    if ( 0==b ) return 0; else return m - b;
}

Two addition tables for the moduli 13 and 9 are shown in figure 37.1-A

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 37.1-A: Addition modulo 13 (left) and modulo 9 (right).

37.1.2 Multiplication

Multiplication is a bit harder: if one would use something like

inline umod_t mul_mod(umod_t a, umod_t b, umod_t m)
{
    return (a * b) % m;
}

Then the modulus would be restricted to half of the word size. Almost all bits can be used for the modulus with the following trick. Let \( \langle x \rangle \) denote \( x \) modulo \( y \), let \( \lfloor x \rfloor \) denote the integer part of \( x \). For \( 0 \leq a, b < m \) we have

\[
a \cdot b = \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m + \langle a \cdot b \rangle_m
\] (37.1-1)

Rearranging and taking both sides modulo \( z > m \) (where \( z = 2^k \) on a \( k \)-bit machine):

\[
\langle a \cdot b - \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \rangle_z = \langle \langle a \cdot b \rangle_m \rangle_z
\] (37.1-2)

The right hand side equals \( \langle a \cdot b \rangle_m \) because \( m < z \).

\[
\langle a \cdot b \rangle_m = \langle \langle a \cdot b \rangle_z - \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \rangle_z
\] (37.1-3)

The expression on the right can be translated into a few lines of C-code. The code given here assumes that one has 64-bit integer types int64 (signed) and uint64 (unsigned) and a floating point type with 64-bit mantissa, float64 (typically long double).

```c
uint64 mul_mod(uint64 a, uint64 b, uint64 m)
{
    uint64 y = (int64)((float64)a*(float64)b/m+(float64)1/2); // floor(a*b/m)
    y = y * m; // m*floor(a*b/m) mod z
}
```
The technique uses the fact that integer multiplication computes the least significant bits of the result \( <a \cdot b> \), whereas float multiplication computes the most significant bits of the result. The above routine works if \( 0 \leq a, b < m < 2^{63} = \frac{z}{2} \). The normalization is not necessary if \( m < 2^{62} = \frac{z}{4} \).

When working with a fixed modulus the division by \( p \) may be replaced by a multiplication with the inverse modulus, that only needs to be computed once:

precompute: \[ \text{float64 } i = (\text{float64})\frac{1}{m}; \]

and replace the line \[ \text{uint64 } y = (\text{uint64})((\text{float64})a*(\text{float64})b/m+(\text{float64})\frac{1}{2}); \]

by \[ \text{uint64 } y = (\text{uint64})((\text{float64})a*(\text{float64})b*i+(\text{float64})\frac{1}{2}); \]

so any division inside the routine is avoided. Beware that the routine cannot be used for \( m \geq 2^{62} \): it very rarely fails for moduli of more than 62 bits, due to the additional error when inverting and multiplying as compared to dividing alone. An implementation in is [FXT: mod/modarith.h]:

```c
inline umod_t mul_mod(umod_t a, umod_t b, umod_t m)
{
    umod_t x = a * b;
    umod_t y = m * (umod_t)( (ldouble)a * (ldouble)b/m + (ldouble)1/2 );
    umod_t r = x - y;
    if ( (smod_t)r < 0 ) r += m;
    return r;
}
```

![Figure 37.1-B: Multiplication modulo 13 (left) and modulo 9 (right).](image)

Two multiplication tables for the moduli 13 and 9 are shown in figure 37.1-B. Note that for the modulus 9 some products \( a \cdot b \) are zero though neither of \( a \) or \( b \) is zero. The tables were computed with the program [FXT: mod/modarithtables-demo.cc].

For alternative multiplication (and reduction) techniques see [205, ch.14]. One method of great practical importance is the Montgomery multiplication described in [207].
37.1.3 Exponentiation

The algorithm used for exponentiation (powering) is the binary exponentiation algorithm shown in section 27.6 on page 565:

```cpp
inline umod_t pow_mod(umod_t a, umod_t e, umod_t m)
// Right-to-left scan
{
if ( 0==e ) { return 1; }
else
{
    umod_t z = a;
    umod_t y = 1;
    while ( 1 )
    {
        if ( e&1 ) y = mul_mod(y, z, m); // y *= z;
        e >>= 1;
        if ( 0==e ) break;
        z = sqr_mod(z, m); // z *= z;
    }
    return y;
}
```

37.1.4 Inversion and division

Subtraction is the inverse of addition. In order to subtract an element \(b\) from another element \(a\) one can add the additive inverse \(-b := m - b\) to \(a\). Each element has an additive inverse.

Division is the inverse of multiplication. In order to divide an element \(a\) by another element \(b\) one can multiply \(a\) by the multiplicative inverse. But not all elements have a multiplicative inverse, only those elements \(b\) that are coprime to the modulus \(m\) (that is, \(\gcd(b,m) = 1\)). These elements are called invertible (modulo \(m\)) or units. For a prime modulus all elements except zero are invertible.

The computation of the GCD uses the Euclidean algorithm [FXT: mod/gcd.h]:

```cpp
template <typename Type>
Type gcd(Type a, Type b)
// Return greatest common divisor of a and b.
{
if ( a < b ) swap2(a, b);
if ( b==0 ) return a;
Type r;
do
{
    r = a % b;
    a = b;
    b = r;
} while ( r!=0 );
return a;
}
```

A variant of the algorithm that avoids most of the (expensive) computations \(a \% b\) is called the binary GCD algorithm [FXT: mod/binarygcd.h]:

```cpp
template <typename Type>
Type binary_ugcd(Type a, Type b)
// Return greatest common divisor of a and b.
// Version for unsigned types.
{
if ( a < b ) swap2(a, b);
if ( b==0 ) return a;
Type r = a % b;
a = b;
b = r;
} while ( !((a|b)&1) ) // both even
```

[fxtbook draft of 2008-August-17]
37.2: Modular reduction with structured primes

```cpp
{ 
    k++; 
    a >>= 1; 
    b >>= 1; 
}
while ( !(a&1) ) a >>= 1;
while ( !(b&1) ) b >>= 1;
while ( 1 )
{
    if ( a==b ) return a << k;
    if ( a < b ) swap2(a, b);
    Type t = (a-b) >> 1; // t>0
    while ( !(t&1) ) t >>= 1;
    a = t;
}
```

The complexity of this algorithm for \( N \)-bit numbers is \( O(N^2) \). We note that an \( O(N \log(N)) \) algorithm is given in [251].

The least common multiple (LCM) of two numbers is

\[
\text{lcm}(a, b) = \frac{a \cdot b}{\text{gcd}(a, b)} = \left( \frac{a}{\text{gcd}(a, b)} \right) \cdot b
\]  

(37.1-4)

The latter form avoids overflow when using integer types of fixed size.

For modular inversion one can use the extended Euclidean algorithm (EGCD), which for two integers \( a \) and \( b \) finds \( d = \text{gcd}(a, b) \) and \( u, v \) so that \( au + bv = d \). Applying EGCD to \( b \) and \( m \) where gcd\((b, m) = 1 \) one obtains \( u \) and \( v \) so that \( mu + bv = 1 \). Reduce modulo \( m \) to obtain \( bv \equiv 1 \pmod{m} \). That is, \( v \) is the inverse of \( b \) modulo \( m \) and \( a/b := ab^{-1} = av \).

The following code implements the EGCD algorithm as given in [172]:

```cpp
template <typename Type>
Type egcd(Type u, Type v, Type &tu1, Type &tu2)
// Return u3 and set u1,v1 so that
// gcd(u,v) == u3 == u*u1 + v*u2
// Type must be a signed type.
{
    Type u1 = 1, u2 = 0;
    Type v1 = 0, v3 = v;
    Type u3 = u, v2 = 1;
    while ( v3!=0 )
    {
        Type q = u3 / v3;
        Type t1 = u1 - v1 * q;
        u1 = v1; v1 = t1;
        Type t3 = u3 - v3 * q;
        u3 = v3; v3 = t3;
        Type t2 = u2 - v2 * q;
        u2 = v2; v2 = t2;
    }
    tu1 = u1; tu2 = u2;
    return u3;
}
```

Another algorithm for the computation of the modular inversion uses exponentiation. It is given only after the concept of the order of an element has been introduced (section 37.4 on page 771).
Chapter 37: Modular arithmetic and some number theory

\[2^{64} - 2^{32} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{76} - 2^{32} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{96} - 2^{32} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{176} - 2^{48} + 1 = 2^{16} \cdot (2^{16} - 1) \cdot (2^{16} - 5) + 1\]
\[2^{320} - 2^{288} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{368} - 2^{336} + 1 = 2^{16} \cdot (2^{16} - 1) \cdot (2^{16} - 21) + 1\]
\[2^{512} - 2^{32} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{512} - 2^{288} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{544} - 2^{32} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{544} - 2^{96} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{544} - 2^{304} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{576} - 2^{512} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{672} - 2^{192} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{832} - 2^{448} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{992} - 2^{832} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{1088} - 2^{608} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{1184} - 2^{768} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{1376} - 2^{32} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{1664} - 2^{256} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{1856} - 2^{1056} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{1920} - 2^{384} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]
\[2^{1984} - 2^{544} + 1 = 2^{32} \cdot (2^{32} - 1) + 1\]

Figure 37.2-A: The complete list of primes of the form \(p = x^k - x^j + 1\) where \(x = 2^G, G = 2^i, G \geq 32\) and \(p\) up to 2048 bits (left), and the equivalent list for \(x = 2^{16}\) and \(p\) up to 1024 bits (right).

```plaintext
? M=x^20-x^15+x^10-x^5+1;
? n=poldegree(M);
? P=sum(i=0,2*n-1,eval(Str("p_"i))*x^i)
   p_39*x^39 + p_38*x^38 + ... + p_3*x^3 + p_2*x^2 + p_1*x + p_0
? R=P%M;
? for(i=0,n-1,print("r_"i," = ",polcoeff(R,i)))
   r_0 = p_0 + (-p_20 - p_25)
   r_1 = p_1 + (-p_21 - p_26)
   r_2 = p_2 + (-p_22 - p_27)
   r_3 = p_3 + (-p_23 - p_28)
   r_4 = p_4 + (-p_24 - p_29)
   r_5 = p_5 + (p_20 - p_30)
   r_6 = p_6 + (p_21 - p_31)
   r_7 = p_7 + (p_22 - p_32)
   r_8 = p_8 + (p_23 - p_33)
   r_9 = p_9 + (p_24 - p_34)
   r_10 = p_10 + (-p_20 - p_35)
   r_11 = p_11 + (-p_21 - p_36)
   r_12 = p_12 + (-p_22 - p_37)
   r_13 = p_13 + (-p_23 - p_38)
   r_14 = p_14 + (-p_24 - p_39)
   r_15 = p_15 + p_20
   r_16 = p_16 + p_21
   r_17 = p_17 + p_22
   r_18 = p_18 + p_23
   r_19 = p_19 + p_24
```

Figure 37.2-B: Computation of the reduction rule for the 640-bit prime \(Y_{50}(2^{32})\).
37.2 Modular reduction with structured primes

The modular reduction with Mersenne primes $M = 2^k - 1$ is especially easy: let $u$ and $v$ be in the range $0 \leq u, v < M = 2^k - 1$, then with the non-reduced product written as $uv = 2^kr + s$ (where $0 \leq r, s < M = 2^k - 1$) the reduction is simply $uv \equiv r + s \pmod{M}$.

A modular reduction algorithm that uses only shifts, additions and subtractions can also be found for structured primes (called generalized Mersenne primes in [247]). Let the modulus $M$ be of the form

$$M = \sum_{i=0}^{n} m_i x^i$$

where $x = 2^k$ and the leading coefficient is equal to one: $m_n = 1$. For simplicity we further assume that $m_i = \pm 1$ and $m_{n-1} = -1$ (so that the numbers fit into $n$ bits). The reduction algorithm can be found using polynomial arithmetic. Write the non-reduced product $P$ as

$$P = \sum_{i=0}^{2n-1} p_i x^i$$

where $0 \leq p_i < x$. Write the reduced product $R$ as

$$R = \sum_{i=0}^{n-1} r_i x^i := P \pmod{M}$$

where $0 \leq r_i < x$. We determine the reduction rules for moduli of the form $x^k - x^j + 1$ (for $k = 3$ and $j = 2$, the rules are the last three lines):

```plaintext
? k=3;j=2;
? M=x^k-x^j+1
? n=poldegree(M);
? P=sum(i=0,2*n-1,eval(Str("p_"i))*x^i) \ unreduced product
? R=P%M; \ reduced product
? for(i=0,n-1,print("r_"i," = ",polcoeff(R,i))
```

A list of primes of the form $p = x^k - x^j + 1$ where $x = 2^G$, $G$ a power of two and $G \geq 16$ is shown in figure 37.2-A. The equivalent list with $i$ a multiple of 8 is given in [FXT: data/structured-primes-2k2j1.txt]. The primes allow radix-2 number theoretic transforms up to a length of $x^j$.

Structured primes that are evaluations of cyclotomic polynomials are given in section 37.11.4.7 on page 801. The reduction rule for the 640-bit prime $M = Y_{50}(2^{32})$ is shown in figure 37.2-B. There is a choice for the ‘granularity’ of the rule: the modulus also equals $Y_{10}(2^{5-32})$, so we can obtain the reduction rule for groups of five 32-bit words

```plaintext
? M=x^4-x^3+x^2-x+1;
? for(i=0,n-1,print("r_"i," = ",polcoeff(R,i))
```

The rule in terms of single words seems to be more appropriate as it allows for easier code generation.

37.3 The sieve of Eratosthenes

Several number theoretic algorithms can take advantage of a precomputed list of primes. A simple and quite efficient algorithm, called the sieve of Eratosthenes computes all primes up to a given limit. It uses
a tag-array where all entries $\geq 2$ are initially marked as potential primes. The algorithm proceeds by searching for the next marked entry and deleting all multiples of it.

An implementation that uses the \texttt{bitarray} class (see section 4.6 on page 158) is given in \cite{FXT:mod/eratosthenes-demo.cc}:

```c
void eratosthenes(bitarray &ba)
{
  ba.set_all();
  ba.clear(0);
  ba.clear(1);
  ulong n = ba.n_;  
  ulong k = 0;
  while ( (k=ba.next_set(k+1)) < n )
  {
    for (ulong j=2, i=j*k; i<n; ++j, i=j*k) ba.clear(i);
  }
}
```

The program prints the resulting list of primes (code slightly simplified):

```c
int main(int argc, char **argv)
{
  ulong n = 100;
  NXARG(n, "Upper limit for prime search");
  bitarray ba(n);
  eratosthenes(ba);
  ulong k = 0;
  ulong ct = 0;
  while ( (k=ba.next_set(k+1)) < n )
  {
    ++ct;
    cout << " " << k;
  }
  cout << endl;
  cout << "Found " << ct << " primes below " << n << "." << endl;
  return 0;
}
```

The output for the default ($n = 100$) is:

```plaintext
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Found 25 primes below 100.
```

A little thought leads to a faster variant: when deleting the multiples $k \cdot p$ of the prime $p$ from the list we only need to care about the values of $k$ that are greater than all primes found so far. Further, values $k \cdot p$ containing only prime factors smaller than $p$ have already been deleted. That is, we only need to delete the values \{ $p^2$, $p^2+2p$, $p^2+2p$, $p^2+3p$, \ldots \}. This algorithmic improvement can be deduced from the series acceleration of the Lambert series given as relation 35.1-22 on page 691. If we further extract the loop for the prime 2 then for the odd primes, we need to delete only the values \{ $p^2$, $p^2+2p$, $p^2+4p$, $p^2+6p$, \ldots \}.

The implementation is

```c
void eratosthenes_opt(bitarray &ba)
{
  ba.set_all();
  ba.clear(0);
  ba.clear(1);
  ulong n = ba.n_;  
  ulong r = isqrt(n);
  ulong k = 0;
  while ( (k=ba.next_set(k+1)) < n )
  {
    if ( k > r ) break;
    for (ulong j=k*k; j<n; j+=k*2) ba.clear(j);
  }
}
```

The routine is included in the demo, the second argument chooses whether the optimized routine is used.
When computing the primes up to a limit $N$ then about $N/p$ values are deleted after finding the prime $p$. If we slightly overestimate the computational work $W$ by

$$W \approx N \sum_{p < N, \text{prime}} \frac{1}{p}$$ (37.3-1)

then $W \approx N \log \log(N)$ which is almost linear. Practically, much of the time used with larger values of $N$ is lost waiting for memory access. Thereby, further improvements should rather address machines specific optimizations than additional algorithmic refinements.

One can save half of the space by recording only the odd primes. A C++ implementation of the modified algorithm is [FXT: make_oddprime_bitarray() in mod/eratosthenes.cc]. The corresponding table is created upon startup of programs linking the FXT-library. The data can be used to verify the primality of small numbers [FXT: is_small_prime() in mod/primes.cc]. The function next_small_prime() in the same file uses the data to return the next prime greater or equal to its argument or zero if the argument is too big.

### 37.4 The order of an element

![Figure 37.4-A: Powers and orders modulo 13, the maximal order is $R(13) = 12 = \varphi(13)$.](image)

![Figure 37.4-B: Powers and orders modulo 9 (left), the maximal order is $R(9) = 6 = \varphi(9)$. The order modulo $m$ is defined only for elements $a$ where $\gcd(a,m) = 1$. The table of powers for the group of units $(\mathbb{Z}/9\mathbb{Z})^*$ (right) is obtained by dropping all elements for which the order is undefined.](image)
Chapter 37: Modular arithmetic and some number theory

The (multiplicative) order $r = \text{ord}(a)$ of an element $a$ is the smallest positive exponent so that $a^r = 1$. For elements that are not invertible ($\gcd(a, m) \neq 1$) the order is not defined. Figure 37.4-A shows the powers of all elements modulo the prime 13. The rightmost column gives the order of those elements that are invertible.

An element $a$ whose $r$-th power equals one is called an $r$-th root of unity: $a^r = 1$. Modulo 9 both elements 2 and 4 are a 6th roots of unity, see figure 37.4-B.

If $a^r = 1$ but $a^x \neq 1$ for all $x < r$ then $a$ is called a primitive $r$-th root of unity. Modulo 9 the element 2 is a primitive 6th root of unity; the element 4 is not, it is a primitive 3rd root of unity. An element of order $r$ is an $r$-th primitive root of unity.

The maximal order $R(m)$ is simply the maximum of the orders of all elements for a fixed modulus $m$. For prime modulus $p$ the maximal order equals $R(p) = p - 1$. Where it cannot cause confusion we omit the argument to the maximal order in what follows.

An element of maximal order is a $R$-th primitive root of unity. Roots of unity of an order different from $R$ are available only for the divisors $d_i$ of $R$: if $g$ is an element of maximal order $R$ then $g^{R/d_i}$ has order $d_i$ (it is a primitive $d_i$-th root of unity):

$$\text{ord} \left( g^{R/d_i} \right) = d_i \quad (37.4-1)$$

This is because $(g^{R/d_i})^{d_i} = g^R = 1$ and $(g^{R/d_i})^k \neq 1$ for $k < d_i$.

The factor by which the order of an element falls short of the maximal order is sometimes called the index of the element. Let $i$ be the index and $r$ the order, then $i \cdot r = R$.

The concept of the order comes from group theory. The invertible elements modulo $m$ with multiplication form a group: the multiplicative group. The neutral element is one. The (multiplicative) order defined above is the order in this group, it tells us how often we have to multiply the element to one to obtain one. We restrict orders to positive values, else every element would have order zero.

With addition things are simpler, all elements with addition form a group with zero as neutral element: the additive group. The additive order of an element in this group tells us how often we have to add the element to zero to obtain zero. The additive order of the element $a$ modulo $m$ is simply $m/\gcd(a, m)$. All elements coprime to $m$ (and especially 1 and $-1$) are generators of the additive group.

The maximal order $R$ of all elements of a group is sometimes called the exponent of the group. Elements of maximal order are also called primitive elements of the group, primitive roots of the group, or generators of the group. In what follows we describe under which conditions the multiplicative group has generators.

37.5 Prime modulus: the field $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p = \text{GF}(p)$

If the modulus is a prime $p$ then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = \text{GF}(p)$: all elements except 0 have inverses and thereby division is possible in GF($p$). The maximal order $R$ equals $p - 1$. Elements of order $R$ are called primitive roots modulo $p$ or generators modulo $p$.

If $g$ is a generator, then every element in GF($p$) different from 0 is equal to some power $g^e$ ($1 \leq e < p$) of $g$ and its order is $R/e$. To test whether $g$ is a primitive root we only need to check whether

$$g^{(p-1)/q_i} \neq 1 \mod p \quad (37.5-1)$$

for all prime divisors of $q_i$ of $p - 1$. To find a primitive root use a simple search:

```
function primroot(p)
{
  if p==2 then  return 1
  f[] := distinct_prime_factors(p-1)
  for r:=2 to p-1
    return r
}
```
In practice the root is found after only a few tries. Note that the factorization of $p - 1$ must be known. An element of order $n$ in $\text{GF}(p)$ is returned by the following function:

```plaintext
function element_of_order(n, p) {
    R := p - 1 // maxorder
    if (R/n)*n != R then error("order n must divide maxorder p-1")
    r := primroot(p)
    x := r**(R/n)
    return x
}
```

In what follows we will need the function $\varphi$, the totient function (or Euler's totient function). The function $\varphi(m)$ counts the number of integers coprime to and less than $m$:

$$\varphi(m) \ := \ \sum_{1 \leq k < m \text{gcd}(k,m) = 1} 1 \quad (37.6-1)$$

The sequence of values $\varphi(n)$ is entry $\text{A000010}$ in [231]. The values of $\varphi(n)$ for $n \leq 96$ are shown in figure 37.6-A. For $m = p$ prime one has $\varphi(p) = p - 1$. For $m$ composite $\varphi(m)$ is always less than $m - 1$. For $m = p^k$ a prime power

$$\varphi(p^k) = p^k - p^{k-1} = p^{k-1}(p - 1) \quad (37.6-2)$$

The totient function is a multiplicative function: one has $\varphi(x_1 x_2) = \varphi(x_1) \varphi(x_2)$ for coprime $x_1, x_2$, that is, $\text{gcd}(x_1, x_2) = 1$ (but $x_1$ and $x_2$ are not required to be primes). Thereby, if $p_i$ are distinct primes in the factorization of $n$, then

$$\varphi(n) = \prod_i \varphi(p_i^{e_i}) \quad \text{where} \quad n = \prod_i p_i^{e_i} \quad (37.6-3)$$
An alternative expression for \( \varphi(n) \) is

\[
\varphi(n) = n \prod_{p_i} \left(1 - \frac{1}{p_i}\right) \quad \text{where} \quad n = \prod_i p_i^{e_i} \quad (37.6-4)
\]

We note a generalization: the number of \( s \)-element sets of numbers \( \leq n \) whose greatest common divisor is coprime to \( n \) equals

\[
\varphi_s(n) = n^s \prod_{p_i} \left(1 - \frac{1}{p_i^{s}}\right) \quad \text{where} \quad n = \prod_i p_i^{e_i} \quad (37.6-5)
\]

Pseudo code to compute \( \varphi(m) \) for arbitrary \( m \):

1. function euler_phi(m)
2. \{ \{n, p[], x[]\} := factorization(m) \// m==product(i=0..n-1, p[i]**x[i])
3. ph := 1
4. for i:=0 to n-1
5. \{ \{k := x[i] \// exponent
6. \phi := ph * (p[i]**(k-1)) * (p[i]-1) \// ==ph * euler_phi(p[i]**x[i])
7. \}
8. \}
9. }

The multiplicative group consists of the invertible elements (or units) and is denoted by \((\mathbb{Z}/m\mathbb{Z})^*\). The size of the group \((\mathbb{Z}/m\mathbb{Z})^*\) equals the number of units:

\[
| (\mathbb{Z}/m\mathbb{Z})^* | = \varphi(m) \quad (37.6-6)
\]

If \( m \) factorizes as \( m = 2^{e_0} \cdot p_1^{e_1} \cdots \cdot p_q^{e_q} \) where \( p_i \) are pairwise distinct primes then

\[
| (\mathbb{Z}/m\mathbb{Z})^* | = \varphi(2^{e_0}) \cdot \varphi(p_1^{e_1}) \cdots \cdot \varphi(p_q^{e_q}) \quad (37.6-7)
\]

Further, the group \((\mathbb{Z}/m\mathbb{Z})^*\) is isomorphic to the direct product of the multiplicative groups modulo the prime powers:

\[
(\mathbb{Z}/m\mathbb{Z})^* \simeq (\mathbb{Z}/2^{e_0}\mathbb{Z}) \times (\mathbb{Z}/p_1^{e_1}\mathbb{Z}) \times \cdots \times (\mathbb{Z}/p_q^{e_q}\mathbb{Z}) \quad (37.6-8)
\]

The relation reflects that we can, instead of working modulo \( m \), do computations modulo all prime powers in parallel. The Chinese remainder theorem (section 37.7 on page 778) tells us how to find the element modulo \( m \) given the results modulo the prime powers. The other direction is simply modular reduction.

### 37.6.1 Cyclic and noncyclic multiplicative groups

If the maximal order \( R(m) \) is equal to \( |(\mathbb{Z}/m\mathbb{Z})^*| = \varphi(m) \) then the multiplicative group \((\mathbb{Z}/m\mathbb{Z})^*\) is called cyclic, else we call it noncyclic. The term cyclic reflects that the powers of any element of maximal order ‘cycle through’ all elements of \((\mathbb{Z}/m\mathbb{Z})^*\). An element of maximal order in a cyclic group is also called a generator as its powers ‘generate’ all elements.

Figure 37.6-B shows the powers and orders of the noncyclic group \((\mathbb{Z}/15\mathbb{Z})^*\) where no element generates all units. The groups \((\mathbb{Z}/13\mathbb{Z})^*\) and \((\mathbb{Z}/9\mathbb{Z})^*\) are cyclic, see figure 37.4-A on page 771 and figure 37.4-B.

For prime modulus \( m \) the group \((\mathbb{Z}/m\mathbb{Z})^*\) contains all nonzero elements and any element of maximal order is a generator of the group.

For \( m \) a power \( p^k \) of an odd prime \( p \) the maximal order \( R \) in \((\mathbb{Z}/m\mathbb{Z})^*\) is

\[
R(p^k) = \varphi(p^k) \quad (37.6-9)
\]
### 37.6: Composite modulus: the ring \( \mathbb{Z}/m\mathbb{Z} \)

#### 37.6.1 Computation of the maximal order

The maximal order \( R(m) \) of an element in \( (\mathbb{Z}/m\mathbb{Z})^* \) can be computed as follows:

```plaintext
function maxorder(m)
    {n, p[], k[]} := factorization(m) // m==product(i=0..n-1,p[i]**k[i])
    R := 1
    for i:=0 to n-1
        {t := euler_phi_pp(p[i], k[i]) // ==euler_phi(p[i]**k[i])
         if p[i]==2 AND k[i] >= 3 then t := t / 2
         R := lcm(R, t)
        }
    return R
```

Now we can see for which moduli \( m \) the multiple group \( (\mathbb{Z}/m\mathbb{Z})^* \) will be cyclic:

\[
(\mathbb{Z}/m\mathbb{Z})^* \text{ is cyclic for } m = 2, 4, p^k, 2 \cdot p^k \text{ where } p \text{ is an odd prime}
\] (37.6-12)

If the factorization of \( m \) contains two different odd primes \( p_a \) and \( p_b \) then

\[
R(m) = \text{lcm}(\ldots, \varphi(p_a), \ldots, \varphi(p_b), \ldots)
\]

---

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>13</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>13</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>13</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>14</td>
</tr>
</tbody>
</table>

**Figure 37.6-B:** Powers and orders modulo 15 (left). The group \( (\mathbb{Z}/15\mathbb{Z})^* \) is noncyclic: there are \( \varphi(15) = 8 \) invertible elements but no element generates all of them as the maximal order is \( R(15) = 4 < \varphi(15) \). The table of powers for the group \( (\mathbb{Z}/15\mathbb{Z})^* \) (right) is obtained by dropping all non-invertible elements.

For \( m \) a power of two an irregularity occurs:

\[
R(2^k) = \begin{cases} 1 & \text{for } k = 1 \\ 2 & \text{for } k = 2 \\ 2^{k-2} & \text{for } k \geq 3 \end{cases}
\] (37.6-10)

That is, for powers of two greater than 4 the maximal order falls short from \( \varphi(2^k) = 2^{k-1} \) by a factor of 2. For the general modulus \( m = 2^{k_0} \cdot p_1^{k_1} \cdot \ldots \cdot p_q^{k_q} \) the maximal order is

\[
R(m) = \text{lcm}(R(2^{k_0}), R(p_1^{k_1}), \ldots, R(p_q^{k_q}))
\] (37.6-11)

where \( \text{lcm} \) denotes the least common multiple.
is at least by a factor of two smaller than
\[ \varphi(m) = \ldots \varphi(p_a) \cdots \varphi(p_b) \ldots \]
because both \( \varphi(p_a) \) and \( \varphi(p_b) \) are even. Thereby \( (\mathbb{Z}/m\mathbb{Z})^* \) cannot be cyclic in that case. The same argument holds for \( m = 2^{k_0} \cdot p^k \) if \( k_0 > 1 \). For \( m = 2^k \) the group \( (\mathbb{Z}/m\mathbb{Z})^* \) is cyclic only for \( k = 1 \) and \( k = 2 \) because of the mentioned irregularity of powers of two (relation 37.6-10).

### 37.6.1.2 Computation of the order of an element

Pseudo code for a function that returns the order of a given element \( x \) in \( (\mathbb{Z}/m\mathbb{Z})^* \):

```plaintext
function order(x, m)
  if gcd(x,m)!=1 then return 0 // x not a unit
  h := euler_phi(m) // number of units
  e := h
  {n, p[], k[]} := factorization(h) // h==product(i=0..n-1,p[i]**k[i])
  for i:=0 to n-1
    f := p[i]**k[i]
    e := e / f
    g1 := x**e mod m
    while g1!=1
      g1 := g1**p[i] mod m
      e := e * p[i]
  return e
```

Pseudo code for a function that returns an element \( x \) in \( (\mathbb{Z}/m\mathbb{Z})^* \) of maximal order:

```plaintext
function maxorder_element(m)
  {R := maxorder(m)
   for x:=1 to m-1
     if order(x, m)==R then return x
  }
```

Again, while the function does a simple search it is efficient in practice. For prime \( m \) the function returns a primitive root. A C++ implementation is \([\text{FXT: maxorder_element_mod() in mod/maxorder.cc}]\). Note that for noncyclic groups the returned element does not necessarily have maximal order modulo all factors of the modulus. We list all elements of \( (\mathbb{Z}/15\mathbb{Z})^* \) together with their orders modulo 15, 3, and 5:

<table>
<thead>
<tr>
<th>x</th>
<th>r1</th>
<th>r3</th>
<th>r5</th>
<th>&lt;--</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1</td>
<td>r3</td>
<td>r5</td>
<td>&lt;--</td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r3</td>
<td>r5</td>
<td>&lt;--</td>
</tr>
<tr>
<td>4</td>
<td>r4</td>
<td>r5</td>
<td>&lt;--</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r6</td>
<td>r5</td>
<td>&lt;--</td>
<td></td>
</tr>
</tbody>
</table>

The two elements marked with an arrow have maximal order modulo 15 but not modulo 3. An element of maximal order modulo all factors of a composite modulus (equivalently, maximal order in all subgroups) can be found by computing a generator for all cyclic subgroups and applying the Chinese remainder algorithm given in section 37.7 on page 778.

### 37.6.2 Generators in cyclic groups

Let \( G \) be the set of all generators in a cyclic group modulo \( n \). Then the number of generators is given by

\[ |G| = \varphi(\varphi(n)) \quad (37.6-13) \]
Let \( g \) be a generator, then \( g^k \) is a generator exactly if \( \gcd(k, \varphi(n)) = 1 \), and there are \( \varphi(n) \) numbers \( k \) that are coprime to \( \varphi(n) \).

Let \( g \) be a generator modulo a prime \( p \). Then \( g \) is a generator modulo \( 2p^k \) for all \( k \geq 1 \) if \( g \) is odd. If \( g \) is even then \( g + p^k \) is a generator modulo \( 2p^k \).

Further, \( g \) is a generator modulo \( p^k \) if \( g^{p-1} \mod p^2 \neq 1 \). The only primes below \( 2^{36} \approx 68 \cdot 10^9 \) for which the smallest primitive root is not the a generator modulo \( p^2 \) are 2, 40487 and 6692367337. Such primes are called non-generous primes, see entry A055578 of [245].

The only known primes \( p \) below \( 32 \cdot 10^{12} \) where \( 2^{p-1} \mod p^2 = 1 \) are 1093 and 3511 (such primes are called Wieferich primes, see entry A001220 of [245]). As 2 is not a generator modulo either of the two we see that whenever 2 is a generator modulo \( p < 32 \cdot 10^{12} \) then it is also a generator modulo \( p^k \) for all \( k > 1 \).

### 37.6.3 Generators in noncyclic groups

\[
\text{? for(n=2,25,print(n," ",znstar(n)))}
\]

\[
\begin{align*}
2 &\ [1, [1, [\text{Mod}(1,2)]]] \quad /* \text{read: [1, [1], [Mod(1,2)]]} */ \\
3 &\ [2, [2], [\text{Mod}(2, 3)]] \\
4 &\ [2, [2], [\text{Mod}(3, 4)]] \\
5 &\ [4, [4], [\text{Mod}(2, 5)]] \\
6 &\ [2, [2], [\text{Mod}(5, 6)]] \\
7 &\ [6, [6], [\text{Mod}(3, 7)]] \\
8 &\ [4, [2, 2], [\text{Mod}(5, 8), \text{Mod}(3, 8)]] \\
9 &\ [6, [6], [\text{Mod}(2, 9)]] \\
10 &\ [4, [4], [\text{Mod}(7, 10)]] \\
11 &\ [10, [10], [\text{Mod}(2, 11)]] \\
12 &\ [4, [2, 2], [\text{Mod}(7, 12), \text{Mod}(5, 12)]] \\
13 &\ [12, [12], [\text{Mod}(2, 13)]] \\
14 &\ [6, [6], [\text{Mod}(3, 14)]] \\
15 &\ [8, [4, 2], [\text{Mod}(8, 15), \text{Mod}(11, 15)]] \\
16 &\ [8, [4, 2], [\text{Mod}(5, 16), \text{Mod}(7, 16)]] \\
17 &\ [16, [16], [\text{Mod}(3, 17)]] \\
18 &\ [6, [6], [\text{Mod}(11, 18)]] \\
19 &\ [18, [18], [\text{Mod}(2, 19)]] \\
20 &\ [8, [4, 2], [\text{Mod}(3, 20), \text{Mod}(11, 20)]] \\
21 &\ [12, [6, 2], [\text{Mod}(5, 21), \text{Mod}(8, 21)]] \\
22 &\ [10, [10], [\text{Mod}(13, 22)]] \\
23 &\ [22, [22], [\text{Mod}(5, 23)]] \\
24 &\ [8, [2, 2, 2], [\text{Mod}(13, 24), \text{Mod}(19, 24), \text{Mod}(17, 24)]] \\
25 &\ [20, [20], [\text{Mod}(2, 25)]]
\end{align*}
\]

**Figure 37.6-C**: Structure of the multiplicative group modulo \( n \) for \( 2 \leq n \leq 25 \), as reported by pari/gp’s function \texttt{znstar()}. When the group is cyclic an element of maximal order generates all invertible elements. With noncyclic groups one needs more than one generator. Pari/gp’s function \texttt{znstar()} gives the complete information about the multiplicative group of units. The help text reads:

\[
\text{\texttt{znstar(n): 3-component vector } v, \text{ giving the structure of } (\mathbb{Z}/n\mathbb{Z})^\times.} \\
v[1] \text{ is the order (i.e. eulerphi(n)),} \\
v[2] \text{ is a vector of cyclic components, and} \\
v[3] \text{ is a vector giving the corresponding generators.}
\]

Its output for \( 2 \leq n \leq 25 \) is shown in figure 37.6-C.

The group is cyclic when there is just one generator. In general, when \texttt{znstar(n)} returns

\[
[\varphi, [r_1, r_2, \ldots, r_k], [g_1, g_2, \ldots, g_k]] \quad (37.6-14)
\]

then the \( \varphi \) invertible elements \( u \) are of the form

\[
u = g_1^{r_1} g_2^{r_2} \ldots g_k^{r_k} \quad (37.6-15)
\]
where \(0 \leq e_i < r_i\) for \(1 \leq i \leq k\). For example, with \(n = 15\):

```plaintext
? znstar(15)
[8, [4, 2], [Mod(8, 15), Mod(11, 15)]]
? g1=Mod(8, 15); g2=Mod(11, 15);
? for(e1=0,4-1,for(e2=0,2-1,print(e1," ",e2," ",g1^e1*g2^e2)))
0 0 Mod(1, 15)
0 1 Mod(11, 15)
1 0 Mod(8, 15)
1 1 Mod(13, 15)
2 0 Mod(4, 15)
2 1 Mod(14, 15)
3 0 Mod(2, 15)
3 1 Mod(7, 15)
```

The multiplicative group modulo \(n = 2^k\) is cyclic only for \(k \leq 2\):

```plaintext
? for(i=1,6,print(i,": ",znstar(2^i)))
1: [1, [ ], [ ]]
2: [2, [2], [Mod(3, 4)]]
3: [4, [2, 2], [Mod(5, 8), Mod(3, 8)]]
4: [8, [4, 2], [Mod(5, 16), Mod(7, 16)]]
5: [16, [8, 2], [Mod(5, 32), Mod(15, 32)]]
6: [32, [16, 2], [Mod(5, 64), Mod(31, 64)]]
```

For \(k \geq 3\) the multiplicative group is generated by the two elements 5 and \(-1\).

### 37.6.4 Inversion by exponentiation

For a unit \(u\) of order \(r = \text{ord}(u)\) one has \(u^r = 1\). As \(r\) divides the maximal order \(R\) also \(u^R = 1\) holds and thereby \(u^{R-1} \cdot u = 1\). That is, the inverse of any invertible element \(u\) equals \(u\) to the \((R-1)\)-st power:

```plaintext
u^{-1} = u^{R-1} \quad (37.6-16)
```

In fact, one has also \(u^{-1} = u^{\varphi(m)-1}\) which may involve slightly more work if the group is noncyclic.

### 37.7 The Chinese Remainder Theorem (CRT)

Let \(m_1, m_2, \ldots, m_f\) be pairwise coprime (that is, \(\gcd(m_i, m_j) = 1\) for all \(i \neq j\)). If \(x \equiv x_i \pmod{m_i}\) for \(i = 1, 2, \ldots, f\) then \(x\) is unique modulo the product \(M = m_1 \cdot m_2 \cdot \ldots \cdot m_f\). This is the Chinese remainder theorem (CRT). Note that it is not assumed that any of the \(m_i\) is prime.

The theorem tells us that a computation modulo a composite number \(M\) can be split into separate computations modulo the coprime factors of \(M\). To evaluate a function \(y := F(x) \mod M\) where \(M = m_1 \cdot m_2 \cdot \ldots \cdot m_f\) (with \(\gcd(m_i, m_j) = 1\) for all \(i \neq j\)), proceed as follows

1. Splitting: compute \(x_1 = x \mod m_1, x_2 = x \mod m_2, \ldots, x_f = x \mod m_f\).
2. Separate computations: compute \(y_1 := F(x_1) \mod m_1, y_2 := F(x_2) \mod m_2, \ldots, F(x_f) = F(x_f) \mod m_f\).
3. Recombination: compute \(y\) from \(y_1, y_2, \ldots, y_f\) using the CRT

For example, when computing the exact convolution of a long sequence via number theoretic transforms (see section 25.3 on page 542) the largest term of the result must be smaller than the modulus. Assume that (efficient) modular arithmetic is available for moduli of at most word size. Now choose several coprime moduli that fit into a word and whose product \(M\) is greater than the largest element of the result. Compute the transforms separately and only at the very end compute, via the CRT, the result modulo \(M\). Note that only the result needs to be smaller than \(M\), we do not need to worry about any intermediate quantities.
37.7: The Chinese Remainder Theorem (CRT)

37.7.1 Efficient computation

For two moduli \( m_1, m_2 \) compute \( x \) with \( x \equiv x_1 \pmod{m_1} \) and \( x \equiv x_2 \pmod{m_2} \) as suggested by the following pseudo code:

```plaintext
function crt2(x1, m1, x2, m2)
{
    c := m1**(-1) \pmod{m2} // inverse of m1 modulo m2
    s := ((x2-x1)*c) \pmod{m2}
    return x1 + s * m1
}
```

For repeated CRT calculations with the same moduli one will use precomputed values \( c = m_1^{-1} \pmod{m_2} \). With more than two moduli use the above algorithm repeatedly. Pseudo code to perform the CRT for several moduli:

```plaintext
function crt(x[0,...,f-1], m[0,...,f-1], f)
{
    x1 := x[0]
    m1 := m[0]
    i := 1
    do
    {
        x2 := x[i]
        m2 := m[i]
        x1 := crt2(x1, m1, x2, m2)
        m1 := m1 * m2
        i := i + 1
    } while i < f
    return x1
}
```

A C++ implementation is given in [FXT: mod/chinese.cc]:

```plaintext
umod_t chinese(const umod_t *x, const factorization &f)
{
    // Return R modulo M where:
    // f[] is the factorization of M,
    // x[] := R modulo the prime powers of f[].
    const int n = f.nprimes();
    // (omitted test that gcd(m_0,...,m_{n-1})=1 )
    const umod_t M = f.product();
    umod_t R = 0;
    for (int i=0; i<n; ++i)
    {
        // Ti = prod(mk) (where k!=i); Ti==M/mi:
        const umod_t Ti = M / f.primepow(i); // exact division
        // ci = 1 / Ti:
        umod_t ci = inv_modpp(Ti, f.prime(i), f.exponent(i));
        // here: 0 <= ci < mi
        // Xi = x[i] * ci * Ti:
        umod_t Xi = ci * Ti;
        // add Xi to result:
        R = add_mod(R, Xi, M);
    }
    return R;
}
```

37.7.2 The underlying construction

We derive the algorithm for CRT recombination from a construction for \( k \) coprime moduli. Define \( T_i \) as

\[
T_i := \prod_{k \neq i} m_k \quad (37.7-1)
\]

[fxtbook draft of 2008-August-17]
and \( c_i \) as
\[
c_i := T_i^{-1} \mod m_i \quad (37.7-2)
\]
Then for \( X_i \) defined as
\[
X_i := x_i c_i T_i \quad (37.7-3)
\]
one has
\[
X_i \mod m_j = \begin{cases} x_i & \text{for } j = i \\ 0 & \text{else} \end{cases} \quad (37.7-4)
\]
Therefore
\[
x := \sum_k X_k = x_i \mod m_i \quad (37.7-5)
\]
For the special case of two moduli \( m_1, m_2 \) one has
\[
T_1 = m_2, \quad T_2 = m_1 \quad (37.7-6a)
\]
\[
c_1 = m_2^{-1} \mod m_1, \quad c_2 = m_1^{-1} \mod m_2 \quad (37.7-6b)
\]
The quantities are related by
\[
c_1 m_2 + c_2 m_1 = 1 \quad (37.7-7)
\]
and
\[
x = \sum_k X_k = x_1 c_1 T_1 + x_2 c_2 T_2 \quad (37.7-8a)
\]
\[
= x_1 c_1 m_2 + x_2 c_2 m_1 \quad (37.7-8b)
\]
\[
= x_1 (1 - c_2 m_1) + x_2 c_2 m_1 \quad (37.7-8c)
\]
\[
= x_1 + (x_2 - x_1) (m_1^{-1} \mod m_2) m_1 \quad (37.7-8d)
\]
The last equality is used in the code.

### 37.8 Quadratic residues

Let \( p \) be a prime. The **quadratic residues modulo** \( p \) are those values \( a \) so that the equation
\[
x^2 \equiv a \pmod{p} \quad (37.8-1)
\]
has a solution. If the equation has no solution then \( a \) is called a quadratic non-residue modulo \( p \). A quadratic residue is a square (modulo \( p \)) of some number, so we can safely just call it a square modulo \( p \).

Let \( g \) be a primitive root (the particular choice does not matter), then every nonzero element \( x \) can uniquely be written as \( x = g^e \) where \( 0 < e < p \). Rewriting equation \(37.8-1\) as \( x^2 = (g^e)^2 = g^{2e} = a \) it is apparent that the quadratic residues are the even powers of \( g \). The non-residues are the odd powers of \( g \). All generators are non-residues: \( g = g^1 \).

Let us compute \( f(x) := x^{(p-1)/2} \) for both residues and non-residues: With a quadratic residue \( g^{2e} \) we get \( f(g^{2e}) = g^{2e(p-1)/2} = 1^e = 1 \) where we used \( g^{p-1} = 1 \). With a non-residue \( a = g^k, \) \( k \) odd, we get \( f(a) = f(g^k) = g^{k(p-1)/2} = -1 \) where we used \( g^{(p-1)/2} = -1 \) (the only square root of 1 apart from 1 is \(-1\)) and \(-1^k = -1 \) for \( k \) odd.
37.8: Quadratic residues

Apparently we just found a function that can tell residues from non-residues. In fact, we rediscovered the so-called Legendre symbol usually written as \( \left( \frac{a}{p} \right) \). A surprising property of the Legendre symbol is the law of quadratic reciprocity: Let \( p \) and \( q \) be distinct odd primes, then

\[
\left( \frac{p}{q} \right) = (-1)^{\frac{p-1}{2} \frac{q-1}{2}} \left( \frac{q}{p} \right)
\]  

(37.8-2)

Also the following relations hold:

\[
\left( -\frac{1}{p} \right) = (-1)^{\frac{p-1}{2}} = \begin{cases} +1 & \text{if } p \equiv 1 \pmod{4} \\ -1 & \text{if } p \equiv 3 \pmod{4} \end{cases}
\]  

(37.8-3a)

\[
\left( \frac{2}{p} \right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} +1 & \text{if } p \equiv \pm 1 \pmod{8} \\ -1 & \text{if } p \equiv \pm 3 \pmod{8} \end{cases}
\]  

(37.8-3b)

\[
\left( \frac{3}{p} \right) = 1 \iff p \equiv \pm 1 \pmod{12}
\]  

(37.8-3c)

\[
\left( -\frac{3}{p} \right) = 1 \iff p = 2, p = 3, \text{ or } p \equiv 1 \pmod{3}
\]  

(37.8-3d)

If \( a \) is a square modulo \( p \) then the polynomial \( x^2 - a \) factors as \( (x - r_1)(x - r_2) \) where \( r_1^2 \equiv a \) and \( r_2^2 \equiv a \). Modulo \( 41 = 4 \cdot 10 + 1 \) minus one is a square and we have \( x^2 + 1 = (x - 9)(x - 32) \). The polynomial \( x^2 + 1 \) with coefficients modulo \( 43 = 4 \cdot 10 + 3 \) is irreducible.

The relation between the Legendre symbols of positive and negative arguments is

\[
\left( -\frac{a}{p} \right) = (-1)^{\frac{a-1}{2}} \left( \frac{a}{p} \right) = \begin{cases} +\left( \frac{a}{p} \right) & \text{if } p = 4k + 1 \\ -\left( \frac{a}{p} \right) & \text{if } p = 4k + 3 \end{cases}
\]  

(37.8-4)

Modulo a prime \( p = 4k + 3 \), if \( +a \) is a square then \( -a \) is not a square. The orders of any two elements \( +a \) and \( -a \) differ by a factor of two. Non-residues can easily be found: \(-\left( b^2 \right) \) is a non-residue for all \( b \).

Modulo a prime \( p = 4k + 1 \), if \( +a \) is a square then \( -a \) is also a square. The orders of two non-residues \( +a \) and \( -a \) are identical. The orders of two residues \( +a \) and \( -a \) can be identical or differ by a factor of two.

A special case are primes of the form \( p = 2^x + 1 \), the Fermat primes. Only five Fermat primes are known today: \( 2^1 + 1 = 3 \), \( 2^2 + 1 = 5 \), \( 2^4 + 1 = 17 \), \( 2^8 + 1 = 257 \) and \( 2^{16} + 1 = 65537 \). To be prime the exponent \( x \) must be a power of two. The primitive roots are exactly the non-residues: the maximal order equals \( R = \varphi(p) = 2^x \). There are \( \varphi(\varphi(p)) = 2^{x-1} \) primitive roots. There are \( (p-1)/2 = 2^{x-1} \) squares which all have order at most \( R/2 \). Remain \( 2^{x-1} \) non-residues which must all be primitive roots.

We will not pursue the issue, but it should be noted that there are more efficient ways than powering to determine the Legendre symbol. A generalization of the Legendre symbol for composite moduli is the so-called Kronecker symbol. An efficient implementation for its computation (following \[91, p.29\]) is given in \[FXT: kronecker() \] in \[mod/kronecker.cc\].
Chapter 37: Modular arithmetic and some number theory

Figure 37.8-A: Kronecker symbols $\left(\frac{a}{n}\right)$ for small positive $a$ and $b$.  

A table of Kronecker symbols $\left(\frac{a}{n}\right)$ for small $a$ and $b$ is shown in figure 37.8-A. It was created with the program [FXT: mod/kronecker-demo.cc].

The following relations hold for the Kronecker symbol:

\[
\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right) \cdot \left(\frac{b}{n}\right) \quad (37.8-5a)
\]

\[
\left(\frac{a}{mn}\right) = \left(\frac{a}{m}\right) \cdot \left(\frac{a}{n}\right) \quad (37.8-5b)
\]

Note we may have $\left(\frac{a}{mn}\right) = +1$ while $a$ is not a square modulo $mn$: If $\left(\frac{a}{m}\right) = \left(\frac{a}{n}\right) = -1$ ($a$ is a non-square modulo both $m$ and $n$) then (by relation 37.8-5b) $\left(\frac{a}{mn}\right) = +1$. But $a$ is not a square mod $mn$, as a square mod $mn$ must be a square both mod $m$ and mod $n$. For example, $\left(\frac{1}{14}\right) = +1$ but 2 is not a square modulo 143 = 11 · 13, we have $\left(\frac{2}{11}\right) = -1$ and $\left(\frac{2}{13}\right) = -1$, so 2 is a non-square modulo both primes and so modulo their product.

For a square $b = a^2$ the Kronecker symbol will always be $+1$: $\left(\frac{a}{n}\right) \cdot \left(\frac{a}{n}\right) = +1$ (by relation 37.8-5a).

Whether a given number $a$ is a square modulo $2^x$ can be determined via the simple routine [FXT: is_quadratic_residue_2ex in mod/quadresidue.cc]:

```c
bool is_quadratic_residue_2ex(umod_t a, ulong x)
// Return whether a is quadratic residue mod 2**x
{
  if ( x==1 ) return true;
  if ( (x%3)==0 ) kk (i==((a&7)) ) return true;
  if ( (x==2) ) kk (i==((a&3)) ) return true;
  return false;
}
```

[fxtbook draft of 2008-August-17]
A curious observation regarding quadratic residues is that exactly for the 29 moduli
2, 3, 4, 5, 8, 12, 15, 16, 24, 28, 40, 48, 56, 60, 72, 88, 112, 120,
168, 232, 240, 280, 312, 408, 520, 760, 840, 1320, 1848
all quadratic residues are non-prime. This sequence is entry A065428 of [245]. It can be generated using the program [FXT: mod/mod-residues-demo.cc].

See any textbook on number theory for the details of the theory of quadratic residues, and [91], [178],
and [244] for the corresponding algorithms. A method for watermarking that uses quadratic residues is discussed in [23]. An algorithm to compute conference matrices via quadratic residues is given in section 18.2 on page 375.

37.9 Computation of a square root modulo $m$

We give algorithms for computing square roots modulo primes, prime powers, and composites.

37.9.1 Prime modulus

The square root of a square $a$ modulo a prime $p = 4k + 3$ can be computed as

$$\sqrt{a} = \pm a^{(p+1)/4} \quad (37.9-1)$$

Write $(a^{(p+1)/4})^2 = a^{(p+1)/2} = a^{(p-1)/2+1} = \pm 1 \cdot a = \pm a$, if $a$ is not a square then the square root of $-a$ is obtained. Similar expressions for square root modulo $p$ are developed in [3]. An algorithm for the computation of a square root modulo a prime $p$ (without restriction on the form of $p$) is given in [91] p.32. We just give a C++ implementation [FXT: sqrt_modp() in mod/sqrtmod.cc]:

```c
1 umod_t
2 sqrt_modp(umod_t a, umod_t p)
3 // Return x such that x*x==a (mod p)
4 // p must be an odd prime.
5 // If a is not a square mod p then return 0.
6 {
7     if ( 1!=kronecker(a,p) ) return 0; // not a square mod p
8     // initialize q,t so that p == q * 2^t + 1
9     umod_t q; int t;
10     n2qt(p, q, t);
11     umod_t z = 0, n = 0;
12     for (n=1; n<p; ++n)
13         { if ( -1==kronecker(n, p) )
14             { z = pow_mod(n, q, p);
15                 break;
16             }
17         }
18     if ( n>=p ) return 0;
19     umod_t y = z;
20     uint r = t;
21     umod_t x = pow_mod(a, (q-1)/2, p);
22     umod_t b = x;
23     x = mul_mod(x, a, p);
24     b = mul_mod(b, x, p);
25     while ( 1 )
26         { if ( 1==b ) return x;
27         }
```

[fxtbook draft of 2008-August-17]
Chapter 37: Modular arithmetic and some number theory

```c
uint m;
for (m=1; m<r; ++m)
{
    if ( 1==pow_mod(b, 1ULL<<m, p) ) break;
}
if ( m==r ) return 0; // a is not a square mod p

umod_t v = pow_mod(y, 1ULL<<(r-m-1), p);
y = sqr_mod(v, p);
r = m;
x = mul_mod(x, v, p);
b = mul_mod(b, y, p);
}

37.9.2 Prime power modulus

For the computation of a square root modulo a prime power \( p^x \) the Newton iteration can be used (see section 28.1.5 on page 571). The case \( p = 2 \) has to be treated separately [FXT: sqrt_modpp() in mod/sqrtmod.cc]:

```c
umod_t sqrt_modpp(umod_t a, umod_t p, long ex)
// Return r so that r^2 == a (mod p^ex)
// return 0 if there is no such r
{
    umod_t r;
    if ( 2==p ) // case p==2
    {
        if ( false=is_quadratic_residue_2ex(a, ex) ) return 0; // no sqrt exists
        else r = 1; // (1/r)^2 = a mod 2
    }
    else // case p odd
    {
        umod_t z = a % p;
        r = sqrt_modp(z, p);
        if ( r==0 ) return 0; // no sqrt exists
    }
    // here r^2 == a (mod p)
    if ( 1==ex ) return r;
}
```

Here \( r \) is a square root of \( a \) modulo \( p \), Newton steps are used to compute \( \sqrt{a} \) modulo powers of \( p \):

```c
const umod_t m = ipow(p, ex);
if ( Z==p ) // case p==2
{
    long x = 1;
    while ( x<ex ) // Newton iteration for inverse sqrt, 2-adic case
    {
        umod_t z = a;
        z = mul_mod(z, r, m); // a*r
        z = mul_mod(z, r, m); // a*r*r
        z = sub_mod(3, z, m); // 3 - a*r*r
        r = mul_mod(r, z/2, m); // r*(3 - a*r*r)/2 = r*(1 + (1-a*r*r)/2)
        x *= 2; // (1/r)^2 == a mod 2^x
    }
    r = mul_mod(r, a, m);
}
else // case p odd
{
    const umod_t h = inv_modpp(2, p, ex); // 1/2
    long x = 1;
    while ( x<ex ) // Newton iteration for square root
    {
        umod_t ri = inv_modpp(r, p, ex); // 1/r
        umod_t ar = mul_mod(a, ri, m); // a/r
        r = add_mod(r, ar, m); // r+a/r
        r = mul_mod(r, h, m); // (r+a/r)/2
        x *= 2; // r^2 == a mod p^x
    }
}
```

[fxtbook draft of 2008-August-17]
37.10: The Rabin-Miller test for compositeness

37.10.1 Pseudoprimes and strong pseudoprimes

For a prime \( p \) the maximal order of an element equals \( p - 1 \). That is, for all \( a \neq 0 \)

\[
a^{p-1} \equiv 1 \mod p
\]  

(37.10-1)

If for a given number \( n \) one succeeds to find an \( a > 1 \) so that \( a^{n-1} \neq 1 \mod n \) then the compositeness of \( n \) has been proven. Composite numbers \( n \) for which \( a^{n-1} = 1 \mod n \) are called pseudoprime to base \( a \) (or \( a \)-pseudoprime). For example, for \( n = 15 \) one finds

\[
a : 2 3 4 5 6 7 8 9 10 11 12 13 14
a^{14}: 4 9 1 10 6 4 4 6 10 1 9 4 1
\]

We found that 15 is pseudoprime to the bases 4, 11 and 14 which we also could have read off the rightmost column of figure 37.6-B on page 775.

The bad news is that some composite numbers are pseudoprime to very many bases. The smallest such example is number 561 which is pseudoprime to all bases \( a \) with \( \gcd(a,n) = 1 \). Numbers with that property are called Carmichael numbers. The first few are 561, 1105, 1729, 2465, 2821, 6601, 8911, ..., this is sequence \([A002997]\) of [245]. There are infinitely many Carmichael numbers as proved in [27]. Finding a base that proves a Carmichael number composite is as difficult as finding a factor.

A significantly better algorithm can be found by a rather simple variation. Write \( n - 1 = q \cdot 2^t \) where \( q \) is odd, we examine the sequence \( b := a^q, b^2, b^4, \ldots, b^{2^{t-1}} = a^{(n-1)/2} \). We say that \( n \) is a strong pseudoprime to base \( a \) if either \( b \equiv 1 \) or \( b^{2^{e-1}} \equiv -1 \equiv n - 1 \) for some \( e \) where \( 0 \leq e < t \). We abbreviate strong pseudoprime as SPP. If neither of the conditions holds then \( n \) is proven composite. Then \( n \) is either not a pseudoprime to base \( a \) or we found a square root of 1 that is not equal to \( n - 1 \).
With two different square roots \( s_1, s_2 \) modulo \( n \) of some number \( z \) (here \( z = 1 \)) one has

\[
\begin{align*}
    s_1^2 - z &\equiv 0 \pmod{n} \quad (37.10-2a) \\
    s_2^2 - z &\equiv 0 \pmod{n} \quad (37.10-2b) \\
    s_1^2 - s_2^2 &= (s_1 + s_2) (s_1 - s_2) \equiv 0 \pmod{n} \quad (37.10-2c)
\end{align*}
\]

So both \( s_1 + s_2 \) and \( s_1 - s_2 \) are nontrivial factors of \( n \) if \( s_1 \neq n - s_2 \). Thereby a square root \( s \neq -1 \) of 1 proves compositeness because both \( \gcd(s + 1, n) \) and \( \gcd(s - 1, n) \) are nontrivial factors of \( n \).

Let \( B = [b, b^2, b^4, \ldots, b^{2^t}] \), then for \( n \) prime the sequence \( B \) must have one of the following forms: either

\[
\begin{align*}
    B &= [1, 1, \ldots, 1] \quad \text{or} \quad (37.10-3a) \\
    B &= [*, \ldots, *, -1, 1, \ldots, 1] \quad (37.10-3b)
\end{align*}
\]

where an asterisk denotes any number not equal to \( \pm 1 \pmod{n} \) (notation as in [178]). For \( n \) composite the sequence \( B \) can also be of the form

\[
\begin{align*}
    B &= [*, \ldots, *] \quad (a^{n-1} \neq 1, \text{not a pseudoprime to base } a) \quad \text{or} \quad (37.10-4a) \\
    B &= [*, \ldots, *, 1, \ldots, 1] \quad \text{(found square root of 1 not equal to } -1) \quad (37.10-4b)
\end{align*}
\]

If one of the latter two forms is encountered then \( n \) must be composite.

With our example \( n = 15 \) we have \( n - 1 = 7 \cdot 2^4 \), thereby \( q = 7 \) and \( t = 1 \). We only have to examine the value of \( b \). Values of \( a \) for which \( b \) is not equal to either \( +1 \) or \( -1 \) prove the compositeness of 15.

\[
\begin{align*}
    a: & \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \\
    b: & \quad 8 \quad 12 \quad 4 \quad 5 \quad 6 \quad 13 \quad 2 \quad 9 \quad 10 \quad 11 \quad 3 \quad 7 \quad -1
\end{align*}
\]

In our example all bases \( \neq 14 \) prove 15 composite. As \( n \) is always a SPP to base \( a = n - 1 \equiv -1 \) we restrict our attention to values \( 2 \leq a \leq n - 2 \).

A pari/gp implementation of the test whether \( n \) is a SPP to base \( a \):

```python
1  sppq(n, a)=
2  { /* Return whether n is a strong pseudoprime to base a */
3      local(q, t, b, e);
4      q = n-1;
5      t = 0;
6      while ( 0==bitand(q,1), q/=2; t+=1 );
7      /* here n==2^t*q+1 */
8      b = Mod(a, n)^q;
9      if ( 1==b, return(1) );
10     e = 1;
11     while ( e< t, 
12             if( (b==1) || (b==n-1), break(); );
13             b *= b;
14             e++;
15     );
16     return( if ( b!=(n-1), 0, 1 ) );
17  }
```

The Carmichael number 561 (\( 561 - 1 = 35 \cdot 2^4 \), so \( q = 35 \) and \( t = 4 \)) is a SPP to only 8 out of the 558 interesting bases, and not a SPP for any \( 2 \leq a \leq 20 \) as shown in figure 37.10-A. Note that with \( a = 4 \) we found \( s = 67 \) where \( s^2 \equiv 1 \pmod{561} \) and thereby the factors \( \gcd(67+1, 561) = 17 \) and \( \gcd(67-1, 561) = 33 \) of 561.

### 37.10.2 The Rabin-Miller test

The **Rabin-Miller test** is an algorithm to prove compositeness of a number \( n \) by testing strong pseudoprimality with several bases:
37.10: The Rabin-Miller test for compositeness

| a=2: | b=263 | b^2=166 | b^4=67 |
| a=3: | b=78  | b^2=474 | b^4=276 |
| a=4: | b=166 | b^2=67  | b^4=1   |
| a=5: | b=43  | b^2=529 | b^4=463 |
| a=6: | b=318 | b^2=144 | b^4=640 |
| a=7: | b=241 | b^2=298 | b^4=166 |
| a=8: | b=461 | b^2=463 | b^4=67  |
| a=9: | b=474 | b^2=276 | b^4=441 |
| a=10:| b=439 | b^2=298 | b^4=166 |
| a=11:| b=463 | b^2=463 | b^4=319 |
| a=12:| b=435 | b^2=342 | b^4=276 |
| a=13:| b=561 | b^2=100 | b^4=463 |
| a=14:| b=67  | b^2=67  | b^4=1   |
| a=15:| b=551 | b^2=100 | b^4=463 |
| a=16:| b=111 | b^2=540 | b^4=441 |
| a=17:| b=527 | b^2=34  | b^4=34  |
| a=18:| b=120 | b^2=375 | b^4=375 |
| a=19:| b=76  | b^2=166 | b^4=67  |
| a=20:| b=452 | b^2=100 | b^4=463 |

Figure 37.10-A: The Carmichael number $561 = 35 \cdot 2^4 + 1$ is a strong pseudoprime to 8 out of 558 bases $a$ (right) and no basis $2 \leq a \leq 20$ (left).

```c
rm(n, na=20)=
1 { /* Rabin Miller test */
2   local(a);
3     for (a=2, na+2,
4       if ( a>n-2, break() );
5         if ( 0==sppq(n, a), return(0) ); /* proven composite */
6     );
7     return(1); /* composite with probability less than 0.25^na */
8 }
```

It can be shown that for a composite number the probability to be a SPP to a ‘random’ base is at most $1/4$. Thereby the compositeness of number can in practice quickly be proven. While the algorithm does not prove primality, it can be used to rule out compositeness with a very high probability.

<table>
<thead>
<tr>
<th>Bases tested:</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>91:</td>
<td>[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>276:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>286:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>703:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>742:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>781:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>946:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1729:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2047:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2806:</td>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3277:</td>
<td>[3]</td>
<td></td>
<td></td>
<td>14</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4033:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>14</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4187:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>14</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5662:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>10</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5713:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>10</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6533:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>10</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6541:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>10</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7171:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>10</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8401:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>10</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8911:</td>
<td>[3]</td>
<td></td>
<td></td>
<td>12</td>
<td>13</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9073:</td>
<td>[2]</td>
<td></td>
<td></td>
<td>12</td>
<td>13</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 37.10-B: All numbers $\leq 10,000$ that are strong pseudoprimes to more than one base $a \leq 17$ (omitting bases $a$ that are prime powers).

A list (created with the program [FXT: mod/rabinmiller-demo.cc]) of composites $n \leq 10,000$ that are SPP to more than one base $a \leq 17$ is shown in figure 37.10-B. The table indicates how effective the Rabin-Miller algorithm actually is: it does not contain a single number pseudoprime to both 2 and 3. The first few odd composite numbers that are SPP to both bases $a = 2$ and $a = 3$ are shown in figure 37.10-C. There are 104 such composite $n < 2^{32}$, given in [FXT: data/pseudo-spp23.txt]. This sequence of numbers is entry A072276 of [245], entry A001262 gives the base-2 SPPs, and entry A020229 gives the base-3
Chapter 37: Modular arithmetic and some number theory

1,373,653 == 1 + 2^2 * 3^3 * 7 * 23 * 79
== 829 * 1657 == (1 + 2^2 * 3^2 * 23) * (1 + 2^3 * 3^2 * 23)

1,530,787 == 1 + 2 * 3 * 103 * 2477
== 619 * 2473 == (1 + 2 * 3 * 103) * (1 + 2^3 * 3 * 103)

1,987,021 == 1 + 2^2 * 3^2 * 5 * 7 * 19 * 83
== 997 * 1993 == (1 + 2^2 * 3 * 83) * (1 + 2^3 * 3 * 83)

2,284,453 == 1 + 2^2 * 3^2 * 23 * 31 * 89
== 1069 * 2137 == (1 + 2^2 * 3 * 89) * (1 + 2^3 * 3 * 89)

3,116,107 == 1 + 2 * 3^2 * 7^2 * 3533
== 883 * 3529 == (1 + 2 * 3^2 * 7^2) * (1 + 2^3 * 3^2 * 7^2)

5,173,601 == 1 + 2^5 * 5^2 * 29 * 223
== 929 * 5569 == (1 + 2 * 3^2 * 5^3) * (1 + 2 * 3^2 * 5^4)

6,787,327 == 1 + 2 * 3 * 7 * 13 * 31 * 401
== 1699 * 5209 == (1 + 2 * 3 * 7 * 31) * (1 + 2^3 * 3 * 7 * 31)

11,541,307 == 1 + 2 * 3 * 7 * 283 * 971
== 1699 * 6793 == (1 + 2 * 3 * 283) * (1 + 2^3 * 3 * 283)

13,694,761 == 1 + 2^3 * 3^2 * 5 * 109 * 349
== 2617 * 5233 == (1 + 2^3 * 3 * 109) * (1 + 2^4 * 3 * 109)

25,326,001 == 1 + 2^4 * 3^3 * 5^3 * 7 * 67
== 2251 * 11251 == (1 + 2 * 3^2 * 5^3) * (1 + 2 * 3^2 * 5^4)

161,304,001 == 1 + 2^6 * 3 * 5^3 * 11 * 13 * 47
== 7333 * 21997 == (1 + 2^2 * 3 * 11 * 47) * (1 + 2^3 * 3^2 * 11 * 47)

960,946,321 == 1 + 2^4 * 3 * 5 * 29 * 101 * 1367
== 11717 * 82013 == (1 + 2^2 * 3 * 29 * 101) * (1 + 2^2 * 3^2 * 29 * 101)

1,157,839,381 == 1 + 2^2 * 3^3 * 5 * 7 * 37 * 613
== 24681 * 48121 == (1 + 2 * 3 * 5 * 613) * (1 + 2^3 * 3^2 * 5 * 613)

3,215,031,751 == 1 + 2 * 3^3 * 5^3 * 7 * 37 * 613
== 151 * 751 * 28351

3,697,278,427 == 1 + 2 * 3^3 * 31 * 563 * 3923
== 30403 * 121609 == (1 + 2 * 3^3 * 31) * (1 + 2^3 * 3^3 * 3923)

Figure 37.10-C: The first composite numbers that are SPP to both bases 2 and 3.

25,326,001 == 1 + 2^4 * 3^3 * 5^3 * 7 * 67
== 2251 * 11251 == (1 + 2 * 3^2 * 5^3) * (1 + 2 * 3^2 * 5^4)

161,304,001 == 1 + 2^6 * 3 * 5^3 * 11 * 13 * 47
== 7333 * 21997 == (1 + 2^2 * 3 * 11 * 47) * (1 + 2^3 * 3^2 * 11 * 47)

960,946,321 == 1 + 2^4 * 3 * 5 * 29 * 101 * 1367
== 11717 * 82013 == (1 + 2^2 * 3 * 29 * 101) * (1 + 2^2 * 3^2 * 29 * 101)

1,157,839,381 == 1 + 2^2 * 3^3 * 5 * 7 * 37 * 613
== 24681 * 48121 == (1 + 2 * 3 * 5 * 613) * (1 + 2^3 * 3^2 * 5 * 613)

3,215,031,751 == 1 + 2 * 3^3 * 5^3 * 7 * 37 * 613
== 151 * 751 * 28351

3,697,278,427 == 1 + 2 * 3^3 * 31 * 563 * 3923
== 30403 * 121609 == (1 + 2 * 3^3 * 31) * (1 + 2^3 * 3^3 * 3923)

Figure 37.10-D: All composite numbers n ≤ 2^32 that are SPP to the three bases 2, 3 and 5.

SPPs. We note the uneven distribution modulo 12:

(n%12: num) (1: 75) (5: 9) (7: 18) (11: 2)

Composites that are SPP to the three bases 2, 3 and 5 are quite rare, figure 37.10-D shows all 6 such composite numbers smaller than 2^32 (values taken from [218] which lists all such numbers < 25 · 10^9). Thereby one can speed up the Rabin-Miller test for small values of n (say, n < 2^32) by only testing the bases a = 2, 3, 5 and, if n is a SPP to these bases, look up the composites in the table. The smallest odd composites that are SPP to the first k prime bases up to k = 8 are determined in [155], they are given as sequence A006945 of [245].

Note that if the probability of a base not proving compositeness was exactly 1/4 we would find much more entries in figure 37.10-D. Slightly overestimating the number of composites below N as N, there should be about (1/4)^N N = N/64 entries, that is 2^26 ≈ 6 · 10^7 for N = 2^32, but we have only six entries. Thereby the Rabin-Miller test is in practice significantly more efficient than one may initially assume.

Let p_k,t be the probability that a k-bit composite ‘survives’ t passes of the Rabin-Miller test. Then we
have, as shown in \[101\],

\[ p_{k,1} < k^{2 \frac{2}{4} - \sqrt{k}} \quad \text{for} \quad k > 2 \]  

(37.10-5)

For large numbers, the bound on the left hand side is much smaller than 1/4: for example, \( p_{1000,1} < 2^{-39} \).

Other bounds given in the cited paper are

\[ p_{100,10} < 2^{-44} \]  

(37.10-6a)

\[ p_{300,5} < 2^{-60} \]  

(37.10-6b)

\[ p_{600,1} < 2^{-75} \]  

(37.10-6c)

The last bound is stronger than that of relation 37.10-5. Still stronger bounds are given in \[76\], also the relation \( p_{k,t} < 4^{-t} \) for all \( k \geq 2 \) and \( t \geq 1 \).

<table>
<thead>
<tr>
<th>Bases tested:</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>11476:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>88831:</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>185191:</td>
<td></td>
<td></td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>697871:</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>736291:</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>765703:</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024651:</td>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1056331:</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1162271:</td>
<td></td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1314631:</td>
<td></td>
<td></td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1373653:</td>
<td></td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
<td>12</td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1530787:</td>
<td></td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1627921:</td>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1857241:</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1987021:</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2030341:</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2284453:</td>
<td></td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2741311:</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3116107:</td>
<td></td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
<td>12</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4181921:</td>
<td></td>
<td></td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 4226533:      |   |   |   | 6 | 7 | 10 | 11 |    |    | 14 | 15 | 17 | <=--=
| 5122133:      |   | 6 | 7 | 10 | 11 |    |    |    |    | 14 | 15 | 17 |
| 5173601:      |   | 2 | 3 | 6 |    |    |    |    |    |    | 12 |    |
| 5481451:      |   | 3 | 5 |    |    |    |    |    |    | 12 | 13 | 14 | 15 |
| 6594001:      |   | 5 |    |    |    | 12 | 13 | 14 | 15 |    |    |    |
| 6787327:      |   | 2 | 3 | 6 | 12 | 13 | 14 | 15 |    |    |    |    |
| 8086231:      |   | 5 | 11 | 12 | 13 | 15 | 17 |    |    |    |    |
| 9504191:      |   |   |   | 5 | 11 | 12 | 13 | 15 |    |    |    |    |
| 9863461:      |   | 2 | 5 | 6 | 10 | 11 | 14 | 15 | 17 |    |    |    |

Figure 37.10-E: Composites \( \leq 10^7 \) that are SPP to at least four bases.

The composites \( \leq 10^7 \) that are SPP to four or more bases \( a < 17 \) are shown in figure 37.10-E. We omit values of \( a \) that are perfect powers because if \( n \) is a base-\( a \) SPP then it is also a base-\( a^k \) SPP for all \( k > 1 \).

The entry for \( n = 4224533 \) (marked with an arrow) shows that a number that is not a SPP to two bases \( a_1 \) and \( a_2 \) may still be a SPP to the base \( a_1 \cdot a_2 \) (here \( a_1 = 2, a_2 = 3 \)). This indicates that one might want to restrict the tested bases to primes. All odd composite numbers \( \leq 10^7 \) that are SPP to four or more prime bases \( a \leq 17 \) are

<table>
<thead>
<tr>
<th>Bases tested:</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>1152271:</td>
<td>[4]</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1314631:</td>
<td>[4]</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2284453:</td>
<td>[4]</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that a number that is a SPP to bases \( a_1 \) and \( a_2 \) is not necessarily SPP to the base \( a_1 \cdot a_2 \). An example is \( n = 9,006,401 \) which is a SPP to bases 2 and 5 but not to base 10:

\[
\begin{array}{cccccccc}
9006401: & 2 & 4 & 5 & 8 & 16 & 18 \\
\end{array}
\]

All composites \( \leq 10^7 \) that are SPP to bases 2 and 3 are also SPP to base 6, same for bases 2 and 5. Out of six composites \( \leq 10^7 \) that are SPP to bases 2 and 7 three are not SPP to base 14:

<table>
<thead>
<tr>
<th>Bases tested:</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>14</th>
<th>16</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>314821:</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>3539101:</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>5388993:</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>14</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6388993:</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>14</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[fxtbook draft of 2008-August-17]
Chapter 37: Modular arithmetic and some number theory

Numbers which are SPP to several chosen bases are constructed in [17] where a composite 337-digit number is given that is SSP to all prime bases \( a < 200 \). See also [284] and [287].

### 37.10.3 Implementation of the Rabin-Miller test

A C++ implementation of the test for pseudoprimality is given in [FXT: mod/rabinmiller.cc]:

```c++
bool is_strong_pseudo_prime(const umod_t n, const umod_t a, const umod_t q, const int t)
// Return whether \( n \) is a strong pseudoprime to base \( a \).
// q and t must be set so that \( n = q \cdot 2^t + 1 \)
{
    umod_t b = pow_mod(a, q, n);
    if ( 1==b ) return true; // passed
    int e = 1;
    while ( (b!=1) && (b!=(n-1)) && (e<t) )
    {
        b = mul_mod(b, b, n);
        e++;
    }
    if ( b!=(n-1) ) return false; // =--> composite
    return true; // passed
}
```

It uses the routine

```c++
void n2qt(const umod_t n, umod_t &q, int &t)
// Set q,t so that \( n = q \cdot 2^t + 1 \)
// n must not equal 1, else routine loops.
{
    q = n - 1; t = 0;
    while ( 0==(q & 1) ) { q >>= 1; ++t; }
}
```

Now the Rabin-Miller test can be implemented as

```c++
bool rabin_miller(umod_t n, uint cm/*=0*/)
// Rabin-Miller compositeness test.
// Return true of none of the bases <=cm prove compositeness.
// If false is returned then \( n \) is proven composite (also for \( n=1 \) or \( n=0 \)).
// If true is returned the probability
// that \( n \) is composite is less than \( (1/4)^\text{cm} \)
{
    if ( n<=1 ) return false;
    if ( n < small_prime_limit ) return is_small_prime( (ulong)n );
    umod_t q;
    int t;
    n2qt(n, q, t);
    if ( 0==cm ) cm = 20; // default
    uint c = 0;
    while ( ++c<=cm )
    {
        umod_t a = c + 1;
        if ( is_small_perfpow(a) ) continue;
        if ( a >= n ) return true;
        if ( !is_strong_pseudo_prime(n, a, q, t) ) return false; // proven composite
    }
    return true; // strong pseudoprime for all tested bases
}
```

[fxtbook draft of 2008-August-17]
The function `is_small_perfpow()` [FXT: mod/perfpow.cc] returns `true` if its argument is a (small) perfect power. It uses a lookup in a precomputed bit-array.

A generalization of the Rabin-Miller test applicable when more factors (apart from 2) of \( n - 1 \) are known is given in [41]. Another generalization (named extended quadratic Frobenius primality test) is suggested in [102].

### 37.11 Proving primality

We describe several methods to prove primality. Only the first, Pratt’s certificate of primality, is applicable for numbers of arbitrary form but not practical in general because it relies on the factorization of \( n - 1 \). The Pocklington-Lehmer test only needs a partial factorization of \( n - 1 \). We give further tests applicable for numbers of a special forms: Pepin’s test, the Lucas-Lehmer test, and the Lucas test.

As already said, the Rabin-Miller test can only prove compositeness. Even if a candidate ‘survives’ many passes, we only know that it is prime with a high probability.

#### 37.11.1 Pratt’s certificate of primality

Only with a prime modulus \( p \) the maximal order equals \( R = p - 1 \). To determine the order of an element modulo \( p \) one needs the factorization of \( p - 1 \). Now if the factorization of \( p - 1 \) is known and we can find a primitive root then we do know that \( p \) is prime. Thereby it is quite easy to prove primality for numbers of certain special forms. For example, let \( p := 2 \cdot 3^{30} + 1 = 411,782,264,189,299 \). One finds that 3 is a primitive root and so we know that \( p \) is prime.

```
[314159311, [3], [2, 3, 5, 199, 1949]]
[2, "--"]
[3, [2], [2]]
[2, "--"]
[5, [2], [2]]
[2, "--"]
[199, [3], [2, 3, 11]]
[2, "--"]
[3, [2], [2]]
[2, "--"]
[11, [2], [2, 5]]
[2, "--"]
[5, [2], [2]]
[2, "--"]
[1949, [2], [2, 487]]
[2, "--"]
[487, [3], [2, 3]]
[2, "--"]
[3, [2], [2]]
[2, "--"]
```

**Figure 37.11-A:** A certificate for the primality of \( p = 314,159,311 \).

In general, the factorization of \( p - 1 \) can contain large factors whose primality needs to be proven. Recursion leads to a primality certificate in the form of a tree which is called Pratt’s certificate of primality.

A certificate for the primality of \( p = 314,159,311 \) is shown in figure 37.11-A. The first line says that 3 is a primitive root of \( p = 314159311 \) and \( p - 1 \) has the prime factors 2, 3, 5, 199, 1949 (actually, \( p - 1 = 2 \cdot 3^{30} \cdot 5 \cdot 199 \cdot 1949 \) but we can ignore exponents). In the second level (indented by 4 characters) appear the prime factors just determined together with their primality certificates: the prime 2 is trivially accepted, all other primes are followed by their (further indented) certificates.
The certificate was produced with the following pari/gp code:

```gp
1 indprint(x, ind)=
2 { /* print x, indented by ind characters */
3    for (k=1, ind, print1(" ");
4    print(x);
5 }
6
7 pratt(p, ind=0)=
8 {
9    local( a, p1, f, nf, t );
10    if ( p<2, \ \ 2 is trivially prime
11       indprint(p, "--", ind);
12       return();
13    );
14    \ p-1 is factored here:
15    a = component(znprimroot(p),2);
16    \ but we cannot access the factorization, so we do it "manually":
17    p1 = p-1;
18    f = factor(p1);
19    nf = matsize(f)[1];
20    t = vector(nf,j, f[j,1] ); f = t; \ \ prime factors only
21    indprint([p, [a], t], ind);
22    \ \ recurse on prime factors of p-1:
23    for (k=1, nf, pratt(f[k], ind+4));
24    return();
25 }
```

```gp
? p=nextprime(Pi*10^8);
? pratt(p)
```

The routine has to be taken with a grain of salt as we rely on the fact that `znprimroot(p)` will fail for composite `p`:

```gp
? pratt(1000)
*** primitive root does not exist in gener
```

The routine has an additional parameter `ind` determining the indentation used with printing. This parameter is incremented with the recursion level, resulting in the tree-like structure of the output. This little trick is often useful with recursive procedures.

With a precomputed table of small primes (see section 37.3 on page 769) the line
if ( p<=2, \ 2 is trivially prime \\
 can be changed to something like \\
 if ( (p<=ptable_max) && (ptable[p]==1), \ trivial to verify \\
 which will shorten the certificate significantly. A certificate for the smallest prime \( p \) greater than \( \pi \cdot 10^{50} \) and \( ptable\_max=100 \) is shown in figure 37.11-B the output of ‘trivial’ primes is suppressed. We note that \( p = \lfloor \pi \cdot 10^{50} \rfloor + 20 \).

Once a certificate is computed it can be verified very quickly. As this type of primality certificate needs the factorization of \( p - 1 \) its computation is in general not feasible for large values of \( p \).

### 37.11.2 The Pocklington-Lehmer test

Let \( p - 1 = F \cdot U \) where \( F > U \) and the complete factorization of \( F \) is known. If, for each prime factor \( q \) of \( F \), we can find \( a_q \) such that \( a_q^{p-1} \equiv 1 \mod p \) and \( \gcd(a_q^{(p-1)/q} - 1, p) = 1 \), then \( p \) is prime.

The corresponding algorithm is called the *Pocklington-Lehmer test* for primality. The following implementation removes entries from the list of prime factors \( q \) of \( F \) until the list is empty:

```plaintext
if ( p<=2, \ 2 is trivially prime \\
 can be changed to something like \\
 if ( (p<=ptable_max) && (ptable[p]==1), \ trivial to verify \\
 which will shorten the certificate significantly. A certificate for the smallest prime \( p \) greater than \( \pi \cdot 10^{50} \) and \( ptable\_max=100 \) is shown in figure 37.11-B the output of ‘trivial’ primes is suppressed. We note that \( p = \lfloor \pi \cdot 10^{50} \rfloor + 20 \).

Once a certificate is computed it can be verified very quickly. As this type of primality certificate needs the factorization of \( p - 1 \) its computation is in general not feasible for large values of \( p \).

### 37.11.2 The Pocklington-Lehmer test

Let \( p - 1 = F \cdot U \) where \( F > U \) and the complete factorization of \( F \) is known. If, for each prime factor \( q \) of \( F \), we can find \( a_q \) such that \( a_q^{p-1} \equiv 1 \mod p \) and \( \gcd(a_q^{(p-1)/q} - 1, p) = 1 \), then \( p \) is prime.

The corresponding algorithm is called the *Pocklington-Lehmer test* for primality. The following implementation removes entries from the list of prime factors \( q \) of \( F \) until the list is empty:

```plaintext
1 pocklington_lehmer(F, u, c=10000)=
2 { /* Pocklington-Lehmer test for the primality of p=f*u+1.
3 * Return last successful base, else zero.
4 * F must be the factorization of f.
5 * Test bases a=2...c
6 * Must have u<f.
7 */
8 local(n, f, C, p, t, ct);
9 n = matsize(F)[1];
10 f = prod(j=1, n, F[j,1]^F[j,2]);
11 if ( f<=u, return(0) );
12 p = f*u + 1;
13 C = vector(n, j, (p-1)/F[j,1]);
14 ct = n; \ number remaining prime divisors of f
15 for (a=2, c,
16   if ( 1==Mod(a,p)^(p-1),
17     for (j=1, n,
18       if ( C[j]!=0, \ skip entries already removed
19         t = component( Mod(a,p)^C[j], 2);
20         if( 1==gcd(t-1, p),
21           C[j] = 0; \ remove entry
22           ct -= 1; \ number of remaining entries
23         );
24       );
25     );
26   if ( ct==0, return(a) );
27   return( 0 );
28 )

We search all primes of the form \( p = F \cdot U + 1 \) where \( F = 100! \), \( U = F - d \), and \( d \) lies in the range \( 1, \ldots, 1000 \). Only candidates that are strong pseudoprimes to both bases 2 and 3 are tested:

```plaintext
f=100!;
F=factor(f);
{ for (d=1, 1000,
  u = f - d;
  p = f*u+1;
  if ( sppq(p, 2) && sppq(p,3),
    q2 = pocklington_lehmer(F, u);
    print1(d, " ");
    print1(" ", q2); 
    print();
  );
) }

We find five such primes \( \approx 8.70978248908948 \cdot 10^{315} \) (in about 10 seconds):
Chapter 37: Modular arithmetic and some number theory

The returned value $a_q$ is the one that did lead to the removal of the last entry in $C[]$. The value is smaller with less prime factors of $F$. Using $F = 2^{500}$ we find primes ($\approx 1.07150860718626 \cdot 10^{301}$) of the form $p = F \cdot U + 1$ where $U = F - d$ and $1 \leq d \leq 3000$ for the following $d$ and maximal $a_q$:

<table>
<thead>
<tr>
<th>d</th>
<th>last a</th>
<th>d</th>
<th>last a</th>
</tr>
</thead>
<tbody>
<tr>
<td>314</td>
<td>35</td>
<td>438</td>
<td>13</td>
</tr>
<tr>
<td>1023</td>
<td>3</td>
<td>3001</td>
<td>13</td>
</tr>
<tr>
<td>1114</td>
<td>15</td>
<td>2481</td>
<td>3</td>
</tr>
<tr>
<td>1327</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The search takes about 20 seconds. Discarding candidates that have small prime factors ($p < 1,000$) gives a four-fold speedup. The prime $2^{3340}(2^{3340} - 1633) + 1 \approx 7.59225935 \cdot 10^{2010}$ is found within 5 minutes.

A further refinement of the test is given in [91], see also [73].

### 37.11.3 Tests for $n = k \cdot 2^t + 1$

#### 37.11.3.1 Proth’s theorem and Pepin’s test

For numbers of the form $p = q \cdot 2^t + 1$ with $q$ odd and $2^t > q$ primality can be proven as follows: If there is an integer $a$ such that $a^{(p-1)/2} \equiv -1 \pmod{p}$ then $p$ must be prime. This is Proth’s theorem.

The ‘FFT-primes’ (see section 25.1 on page 635) are natural candidates for Proth’s theorem. For example, with $p := 2^{57} \cdot 29 + 1 = 4,179,340,454,199,820,289$ one finds that $a^{(p-1)/2} \equiv -1 \pmod{p}$ for $a = 3$, so $p$ must be prime. Note that Proth’s theorem is the special case $F = 2^t > k = U$ of the Pocklington-Lehmer test.

Numbers of the form $2^t + 1$ are composite unless $t$ is a power of two. The candidates are therefore restricted to the so-called Fermat numbers $F_n := 2^{2^n} + 1$. Here it suffices to test whether $3^x \equiv -1 \pmod{F_n}$ where $x = 2^{t-1}$:

```plaintext
for (tx=1,12, print(tx," ", pepin(tx)))
1 1 \ \ F_1 = 5
2 1 \ \ F_2 = 17
3 1 \ \ F_3 = 257
4 1 \ \ F_4 = 65537
5 0
6 0
7 0
```

No Fermat prime greater than $F_4 = 65537$ is known today and all $F_n$ where $5 \leq n \leq 32$ are known to be composite.

Note that $F_{n+1}$ has (about) twice as many bits as $F_n$. Further the number of squarings involved in the test $(t-1)$ is (about) doubled. If we underestimate the cost of multiplying $N$-bit numbers with $N$ operations we get a lower bound for the ratio of the costs of testing $F_{n+1}$ and $F_n$ of four. Assuming the computer power doubles every 18 month and Pepin’s test of $F_n$ is just feasible today we’d have to wait three years (36 month) before we can test $F_{n+1}$. The computation that proved $F_24$ composite is described in [97].
37.11.3.2 What to consider before doing the Pepin’s test

As \(2^t \equiv -1 \pmod{F_n} = 2^t + 1\) we see that the order of two equals \(2t = 2^{n+1}\). The same is true for factors of composite Fermat numbers. When searching factors of \(F_n\) we only need to consider candidates of the form \(1 + k2^{n+2}\). A routine that searches for small factors of \(f_n\) can be implemented as:

```plaintext
ord2pow2(p)=  \(\text{Return the base-2 logarithm of the order of two modulo } p\)
1 \ Must have: ord(2)==2^k for some k
2 { 
3     local(m, rx);
4     rx = 0;
5     m = Mod(2,p);
6     while ( m!=1, m*=m; rx++; );
7     return( rx );
8 } 

ftrialx(n, mm=10^5, brn=0)=
\(\text{Try to find small factors of the Fermat number } F_n=2^{2^n}+1\)
1 \ Try factors 1+ps, 1+2*ps, ..., 1+mm*ps where \(ps=2^{(n+2)}\)
2 \ Stop if \(brn\) factors were found (zero: do not stop)
3 { 
4     local(p,ps,ttx,fct);
5     ps = 2^{(n+2)}; \(\text{factors are of the form } 1+k*ps\)
6     p = ps+1; \(\text{trial factor}\)
7     ttx = 2^{(n+1)}; \(\text{will test whether } \text{Mod}(2,p)^{ttx}=1\)
8     fct = 0; \(\text{how many factors were found so far}\)
9     for (ct=1, mm,
10         if ( (Mod(2,p)^ttx)==1 \(\text{order condition}\)
11             && ( (rx=ord2pow2(p)) == n+1 ) \(\text{avoid factors of smaller Fermat numbers}\)
12             , /* then */
13             print1(n, ": ");
14             print1(p);
15             print1(\" p-1=",factor(p-1));
16             print();
17             fct++;
18             if ( fct==brn, break() );
19         );
20         p += ps;
21     };
22 return(fct); 
}
```

We create a list of small prime factors of \(F_n\) for \(5 \leq n \leq 32\) where the search is restricted to factors \(f \leq 1 + 10^52^{n+2}\) and stopped when a factor was found:

```plaintext
for(n=5,32, ftrialx(n, 10^5, 1); ); 
5: 641  \(p-1=[2, 7; 5, 1]\)
6: 274177  \(p-1=[2, 8; 3, 2; 7, 1; 17, 1]\)
9: 2424833  \(p-1=[2, 16; 37, 1]\)
10: 45592577  \(p-1=[2, 12; 1131, 1]\)
11: 319489  \(p-1=[2, 13; 3, 1; 13, 1]\)
12: 114689  \(p-1=[2, 14; 7, 1]\)
15: 1214281009  \(p-1=[2, 21; 3, 1; 193, 1]\)
16: 825753601  \(p-1=[2, 19; 3, 2; 5, 2; 7, 1]\)
18: 13631489  \(p-1=[2, 20; 13, 1]\)
19: 70525124609  \(p-1=[2, 21; 33629, 1]\)
23: 167772161  \(p-1=[2, 25; 5, 1]\)
32: 25409026523137  \(p-1=[2, 34; 3, 1; 17, 1; 29, 1]\)
```

A list for \(5 \leq n \leq 300\) is given in [FXT: data/small-fermat-factors.txt]. Note that an entry

```
201: 124569837190596286160012901398286924947521176076042100592562667521 \ p-1=[2, 204; 3, 1; 5, 1; 17, 1; 19, 1]
```

asserts the compositeness of the number \(F_{201}\) where Pepin’s test is out of reach by far. Indeed, its binary representation could not be stored in all existing computer memory combined: \(F_{201}\) is a \(\log_2(F_{201}) \approx 2^{201} = 3.2138 \cdot 10^{600}\)-bit number.

The status of the (partial) factorizations of Fermat numbers is given in [165].
37.11.4 Tests for \( n = k \cdot 2^t - 1 \)

37.11.4.1 The Lucas-Lehmer test for Mersenne numbers

Define the sequence \( H \) by \( H_0 = 1 \), \( H_1 := 2 \) and \( H_i = 4 H_{i-1} - H_{i-2} \). The Mersenne number \( n = 2^e - 1 \) is prime exactly if \( H_{2^e-2} \equiv 0 \mod n \). The first few terms of the sequence \( H \) are

\[
\begin{align*}
  k : & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  H_k : & 1 & 2 & 7 & 26 & 97 & 362 & 1351 & 5042 & 18817 & 70226 & 262087 & 978122 & 3650401 \\
\end{align*}
\]

The numbers \( H_k \) can be computed efficiently via the index doubling formula \( H_{2k} = 2 H_k^2 - 1 \). Starting with the value \( H_1 = 2 \) and computing modulo \( n \) the implementation is as simple as

```plaintext
1 LL(e)=
2  \{ local(n, h);
3      n = 2^e-1;
4      h = Mod(2,n);
5      for (k=1, e-2, h=2*h*h-1);
6      return( 0==h );
7  \}
```

? LL(521)
\`
1 \ 2^521-1 is prime
```

? LL(239)
\`
0 \ 2^239-1 is composite
```

? LL(9941)
\`
1 \ 2^9941-1 is prime
```

This is entry [245]: entry [A002812] gives the values \( V_k \):

\[
\begin{align*}
  2, & 7, 97, 18817, 70815877, 1002978273411373057, 1011930833587051801141281782051050497, \ldots \\
\end{align*}
\]

This is entry [A000043] of [245]:

\[
\begin{align*}
  2, & 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, \\
     & 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, \\
     & 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917 \\
\end{align*}
\]

Four more exponents for Mersenne primes are known: 24036583, 25964951, 30402457, and 32582657. They are not in the sequence as they might be preceded by currently unknown values \( e \). The list of exponents is also given in [FXT: mod/mersenne-exponents.cc].

37.11.4.2 What to consider before doing the Lucas-Lehmer test

The exponent \( e \) of a Mersenne prime must be prime, else \( n \) factors algebraically as

\[
2^e - 1 = \prod_{d|e} Y_d(2)
\]

where \( Y_k(x) \) is the \( k \)-th cyclotomic polynomial (see section 35.1.1 on page 687). For example, with \( 2^{21} - 1 = 2097151 \) the following factors are found:

? m=1; fordiv(21,d, y2=subst(polcyclo(d,x),x,2); m*=y2;print(d,"",y2)); m
\`
1 1
13 1
21 2097151 \ 
== 2^21-1
```

The factors found are not necessarily prime: here 2359 factors further into 7 · 337. More information on the multiplicative structure of \( b^e \pm 1 \) can be found in [74].
Before doing the Lucas-Lehmer test one should do a special version of trial division based on the following observation: any factor \( f \) of \( m = 2^e - 1 \) has the property that \( 2^e \equiv 1 \mod f \). That is, \( 2^e - 1 \equiv 0 \mod f \) and \( f \) divides \( m \). We further use the fact that factors are of the form \( 2ek + 1 \) and that \( f \equiv 1 \mod 8 \). The following routine does not try assert the primality of a candidate factor as this would render the computation considerably slower.

```plaintext
mers_trial(e, mct=10^7, bnf=0)=
\try to discover small factors of the Mersenne number 2^e-1
\ e : exponent of the Mersenne number
\ mct : how many factors are tried
\ bnf : stop with the factor found (zero: do not stop)
{
local(f, fi, ct, fct, m8);
print("exponent e=",e);
print("trying up to ", mct, " factors");
fi=2^e; \ factors are of the form 2*e*k+1
f=1;
ct=0;
fct=0; \ how many factors where found so far
while (ct < mct,
f += fi;
   m8 = bitand(f, 7); \ factor modulo 8
   if ( (1!=m8) && (7!=m8), next(); ); \ must equal +1 or -1
   if ( Mod(2, f)^e == Mod(1, f),
      print(f, " ", isprime(f)); \ give factor and tell whether it is prime
      fct++;
   );
   if ( fct==bnf , break(); );
ct++;
);
}
```

For \( m = 2^{10007} - 1 \) (3013 decimal digits) we find three factors of which all are prime:

```plaintext
? e=10007; mers_trial(e,,3);
exponent e=10007
trying up to 10000000 factors
240169 160282169 1136255313 1
? ##*** last result computed in 44 ms.
? ceil((e*log(2.0)/log(10.0)))
646456993 \ m=2^e-1 has 3,013 decimal digits
```

Sometimes one is lucky with truly huge numbers:

```plaintext
? e=2^31-1; mers_trial(e,,1);
exponent e=2147483647
trying up to 100000000 factors
2952657526626031 1
? ##*** last result computed in 583 ms.
? ceil((e*log(2.0)/log(10.0)))
646456993 \ m=2^e-1 has 646,456,993 decimal digits
```

Note that we found that \( m = 2^e - 1 \) is prime if and only if there is no prime \( f < m \) where the order of two equals \( e \). A special case is sometimes given as follows: if both \( p = 4k + 3 \) and \( q = 2p + 1 \) are prime then \( q \) divides \( 2^p - 1 \) (because the order of 2 modulo \( q \) equals \( p \)).

By the way, if both \( p = 4k + 1 \) and \( q = 2p + 1 \) are prime then \( q \) divides \( 2^p + 1 \) (because the order of 2 modulo \( q \) equals \( 2p \) and \( 2^{2p} - 1 = (2^p + 1)(2^p - 1) \)).

#### 37.11.4.3 Lucas-Lehmer test with floats *

The Binet form (see section 34.1.6 on page 675) of the sequence \( H_n \) is

\[
H_n = \frac{1}{2} \left[ (2 + \sqrt{3})^n + (2 - \sqrt{3})^{-n} \right]
\] (37.11-2)
We can rewrite the expression in the form

\[ H_n = \frac{1}{2} \left[ \exp(x)^n + \exp(x)^{-n} \right] = \frac{1}{2} \left[ \exp(n \cdot x) + \exp(-n \cdot x) \right] \] (37.11-3)

where \( x = \log(2 + \sqrt{3}) \). The hyperbolic cosine can be defined as

\[ \cosh(z) = \frac{1}{2} \left[ \exp(z) + \exp(-z) \right] \] (37.11-4)

and the expression equals \( H_n \) for \( z = n \log(2 + \sqrt{3}) \). Now we can give a criterion equivalent to the Lucas-Lehmer condition as follows:

\[ \cosh \left( 2^{m-2} \log \left( 2 + \sqrt{3} \right) \right) \equiv 0 \mod M_m \Rightarrow M_m \text{ is prime} \] (37.11-5)

The relation is computationally useless because the quantity to be computed grows doubly-exponentially with \( m \); the number of digits grows exponentially with \( m \). Already for \( m = 17 \) the calculation has to be carried out with more than 18,741 decimal digits:

\[
? \cosh(2^{17-2} \times \log(2 + \sqrt{3}))
1.8888939581139837726097538478056602 E18741
\]

The program \texttt{hfloat: examples/ex8.cc} does the computations in the obvious (insane) way. Using a precision of 32,768 decimal digits we obtain:

\[
cosh(...) = \begin{array}{c}
+1.888893958113983772609753847805660285946584431551 \\
[... about 18,000 digits ...] \\
... 5579750039800680284170000000000000 ... \\
[decimal point after 7] \\
00000000000000000000000000000000000000000000000000 \\
[...] \\
000000000000000015496956720461401504275898540185472 \times 10^{18742} \\
[nonzero due to numerical imprecision]
\end{array}
\]

After rounding and computing the modulus the program declares \( M_{17} = 2^{17} - 1 \) prime. All this using just 4 MB of memory and computations equivalent to about 35 FFTs of length 1 million, taking about 4 seconds. This is many many million times the work needed by the original (sane) version of the test. Even trial division would have been significantly faster.

The number \( M_{31} \) would need a bigger machine as the computations needs a precision of more than 300 million digits:

\[
? (2^{(31-2)} \times \log(2 + \sqrt{3}))/\log(10) /* approx decimal digits */
307062001.46039800926268312150009204
\]

Apart from being insane the computation can be used to test high precision floating point libraries.

37.11.4.4 The Lucas test

The Lucas-Lehmer test can be generalized for a less restricted set of candidates. The Lucas test can be stated as follows (taken from [227, p.131]):

Let \( n = k \cdot 2^t - 1 \) were \( k \) is odd, \( 2^t > k, n \neq 0 \mod 3 \) and \( k \neq 0 \mod 3 \) (so we must have \( n \equiv 1 \mod 3 \)). Then \( n \) is prime if and only if \( H_{(n+1)/4} \equiv 0 \mod n \) where \( H \) is as given above.

To turn this into an efficient algorithm use the relation \((n+1)/4 = k \cdot 2^t - 2 \). Firstly, compute \( H_k \) as described in section 34.1.1 on page 667:

\[
[H_k, H_{k+1}] = [H_0, H_1] \begin{bmatrix} 0 & -1 \\ 1 & 4 \end{bmatrix}^k
\] (37.11-6)

This is a one-liner in pari/gp:
37.11: Proving primality

? \( H(k) = \text{return}\left( \left[\begin{smallmatrix} 1, 2 \\ 0, -1; 1, 4 \end{smallmatrix}\right]^k [1] \right) \);
? \text{for}(k=0, 10, \text{print}(k, " : ", H(k), " = 1/2 * \), \text{\(2*H(k)\))}
0: 1 = 1/2 * 2
1: 2 = 1/2 * 4
2: 7 = 1/2 * 14
3: 26 = 1/2 * 52
4: 97 = 1/2 * 194 \quad /* = 2*7^2-1 = (14^2-2)/2 */
5: 362 = 1/2 * 724
6: 1351 = 1/2 * 2702 \quad /* = 2*26^2-1 = (52^2-2)/2 */
7: 5042 = 1/2 * 10084
8: 18817 = 1/2 * 37634
9: 70226 = 1/2 * 140452
10: 262087 = 1/2 * 524174

To compute \( H_{2^k+2} \) from \( H_k \) use \((t-2)\) times the index doubling relation \( H_{2k} = 2H_k^2 - 1 \). The test can be implemented as

\[
\begin{align*}
&\text{H}(k, n) = \text{return}\left( \left(\text{Mod}\left(\left[\begin{smallmatrix} 1, 2 \\ 0, -1; 1, 4 \end{smallmatrix}\right], n\right) * \left(\text{Mod}\left(\left[\begin{smallmatrix} 0, -1; 1, 4 \end{smallmatrix}\right]^k, n\right)\right)\right)[1] \right); \\
&\text{lucas}(k, t) = \\
&\quad \{ \\
&\quad \quad \text{local}(n, h); \\
&\quad \quad \text{/* check preconditions: */} \\
&\quad \quad \text{if} \ (0==\text{bitand}(k,1), \text{return}(0)); \quad \|\ k \text{ must be odd} \\
&\quad \quad \text{if} \ (k>=2^t, \text{return}(0)); \\
&\quad \quad n = k*2^t - 1; \\
&\quad \quad \text{if} \ (n%3!=1, \text{return}(0)); \quad \|\ \gcd(3,k)!=0 && \gcd(3,n)!=0 \\
&\quad \quad \text{/* main loop: */} \\
&\quad \quad h = \text{H}(k, n); \\
&\quad \quad \text{for}(j=1, t-2, h *= h; h += h; h -= 1); \quad \|\ \text{index doubling} \\
&\quad \quad \text{return} \ (0==h); \\
&\quad \}
\end{align*}
\]

Note that the routine returns ‘false’ even for primes if the preconditions are not met. With \( n = 5 \cdot 2^{12} - 1 = 20479 \) we obtain

\[
\begin{align*}
&n=20479 \quad k=5 \quad t=12 \\
&\text{H}(k*2^j) \text{ modulo n} \\
&0 \quad 362 \\
&4 \quad 1839 \\
&8 \quad 5686 \\
&12 \quad 96685 \\
&16 \quad 62229 \\
&20 \quad 5516 \\
&24 \quad 9402 \\
&28 \quad 0 \\
\end{align*}
\]

which shows that 20479 is prime. Proving \( n = 5 \cdot 2^{1340} - 1 \) prime takes about 10 milliseconds. The following code finds the first value \( t \geq 2500 \) so that \( n = 5 \cdot 2^t - 1 \) is prime:

\[
\begin{align*}
k=5; \quad t=2500; \quad \text{while} \ (0==\text{lucas}(k, t), t++\); \quad t
\end{align*}
\]

Within 1 second we obtain the result \( t = 2548 \).

37.11.4.5 Numbers of the form \( n = 24j + 7 \) and \( n = 24j + 19 \)

Numbers of the form \( n = 24j + 7 \) satisfy the preconditions of the Lucas test except for the condition that \( 2^t > k \) where \( n = k \cdot 2^t - 1 \). We test whether \( H_{n+1)/4} \equiv 0 \mod n \), as in the Lucas test. Note that \( H_n = T_n(2) \) where \( T_n(x) \) is the \( n \)-th Chebyshev polynomial of the first kind. We use the fast algorithm for its computation described in section 34.2.3 on page 682 for the test routine:

\[
\begin{align*}
&\text{bool test_7mod24(ulong n)} \\
&\quad \{ \\
&\quad \quad \text{ulong nu = (n+1) >> 2;} \\
&\quad \quad \text{umod_t t = chebyT2(nu, n);} \quad // == chebyT(nu, 2, n); \\
&\quad \quad \text{return} \ (0==nu1); \\
&\quad \}
\end{align*}
\]

The function \text{chebyT2()} is given in \[\text{FXT: mod/chebyshev1.cc}\]. Figure 37.11-C gives composite numbers \( n < 2^{32} \) that pass the test. The complete list of such numbers is given in \[\text{FXT: data/pseudo-7mod24.txt}\].
Chapter 37: Modular arithmetic and some number theory

there are just 64 entries. There are only five entries that are strong pseudoprimes to any basis $a < 100,000$, all shown in figure 37.11-C.

The data suggests that composites of the form $n = 24j + 7$ that pass the test and are pseudoprime to a small base are extremely rare. The implied test would cover $1/8$ of all candidates (that are not divisible by 2 or 3), as eight numbers (1, 5, 7, 11, 13, 17, 19 and 23) are coprime to 24.

For numbers of the form $n = 24j + 19$ we use a different test: here we check whether $U(n+1)/4-1 \equiv 0 \mod n$ where $U_0 = 0$, $U_1 = 1$ and $U_k = 4U_{k-1} - U_{k-2}$ (the Chebyshev polynomial of the second kind, $U_n(x)$, evaluated at $x = 2$). The function for testing is

```c
bool test_19mod24(ulong n)
{
    ulong nu = ((n+1) >> 2) - 1;
    umod_t t = chebyU2(nu, n); // == chebyU(nu, 2, n);
    return (0==t);
}
```

were the function `chebyU2()` is given in [FXT: mod/chebyshev2.cc]. The list [FXT: data/pseudo19mod24.txt] contains all (155) composites $n < 2^{32}$ that pass the test. An extract is shown in figure 37.11-D. Just four numbers $n < 2^{32}$ are also strong pseudoprimes to any base $a < 100,000$.

The application of second order recurrent sequences to primality testing is described in [29]: define the sequence $W_k$ by

$$W_k = PW_{k-1} - QW_{k-2}, \quad W_0 = 0, \quad W_1 = 1$$

Then $n$ is a Lucas pseudoprime (with parameters $P$ and $Q$) if $W_{n+1} \equiv 0 \mod n$, where the sign depends on whether $D = P^2 - 4Q$ is a square modulo $n$. For both cases considered here we have $n = 12j + 7$, $D = 16 - 4 = 12 = 4 \cdot 3$, and 3 is not a square modulo $n$. The test would be (note that $W_n = U_{n-1}(2)$)
37.11: Proving primality

```c
#include <umod.h>

bool lucas_7mod12(ulong n) {
    ulong nu = n;
    umod_t t = chebyU2(nu, n);
    return (0==t);
}
```

This test is passed by far more composites than the two tests considered before. A primality test combining a Lucas-type test and a test for strong pseudoprimality has been suggested in [218]. No composite that passes the test has been found so far.

37.11.4.6 An observation about Mersenne numbers

An interesting observation about Mersenne numbers is that the following seems to be true:

\[ M = 2^e - 1 \text{ prime } \iff 3^{2^{e-1}} \equiv -3 \mod M \]  

(37.11-8)

Note that for odd \( e \) the condition is equivalent to \( 3^{(M-1)/2} \equiv -1 \mod n \) and 3 is a non-residue. For prime exponents \( e \) it can be seen that it is very unlikely to find a composite \( M_e \) where \( 3^{(M-1)/2} \equiv -1 \mod n \): the number \( m \) is a strong pseudoprime (SPP) to base 2 by construction and the right hand side of condition 37.11-8 says that \( m \) is a SPP to base 3. Given the rarity of composites that are SPP to both bases (see section 37.10.2 on page 786) the chance to find such a number among the exponentially growing Mersenne numbers is very small. Tony Reix, who observed the statement of relation 37.11-8 independently verified it for prime exponents up to 132499.

37.11.4.7 Primes that are evaluations of cyclotomic polynomials

The Mersenne numbers and Fermat numbers are actually special cases of numbers obtained as cyclotomic polynomials \( Y_n(x) \) (see section 35.1.1 on page 687) evaluated at \( x = 2 \) (sequence A019320 of [245]). The first such numbers are shown in figure 37.11-E. The sequence of values \( n \) such that \( Y_n(2) \) is prime is entry A072226 of [245]:

\[ 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 19, 22, 24, 26, 27, 30, 31, 32, 33, 34, 38, 40, 42, 46, 49, 56, 61, 62, 65, 69, 77, 78, 80, 85, 86, 89, 93, 98, 99, 107, 120, 122, 126, 127, 129, 133, 148, 150, 158, 165, 170, 174, 184, 192, 195, 202, 208, 234, 254, 261, \ldots \]

Now set \( N := Y_n(2) \), testing whether \( 3^{(N-1)/2} \equiv -1 \mod N \) seems to prove primality for all values of \( n \). Note that for \( n \) a power of two the test is Pepin's test. Information about the primality of \( Y_n(2) \) is given in [122]. Theorems about factorizations of \( Y_n(x) \) where \( x \) is an integer are given in [130], see also [68] and [66].

The primes \( Y_n(2) \) are also of interest for number theoretic transforms (see section 25.1 on page 535) because of their special structure allowing for very efficient modular reduction (see section 37.2 on page 769). A prominent example is \( Y_{192}(2) = 2^{64} - 2^{32} + 1 \). Note that the order of 2 modulo \( Y_{192}(2) \) equals \( n \).

The structure of the primes becomes (in base 10) visible if we check evaluations at 10, the first primes \( Y_n(10) \) are

\[ \begin{array}{llllllllllllll}
2: & Y_2(10) = 11 \\
4: & Y_4(10) = 101 \\
10: & Y_{10}(10) = 9091 \\
12: & Y_{12}(10) = 9901 \\
14: & Y_{14}(10) = 909091 \\
19: & Y_{19}(10) = 99999999900000001 \\
23: & Y_{23}(10) = 900900900909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
### Figure 37.11-E: Evaluations $s$ of the first cyclotomic polynomials at two (left). Entries at prime $n$ are Mersenne numbers $M_n$, entries at $n = 2^k$ are Fermat numbers $F_k$. Composites that are strong pseudoprimes to prime bases other than two are marked with ‘SPP’. The right columns show the corresponding data for evaluations at three.
Finally, we do a silly thing: the factors of $Y_{2^7-1}(x)$ over GF(2) are the irreducible binary polynomials of degree 7. If we evaluate them as polynomials over $\mathbb{Z}$ at $x = 10$ and select the prime numbers we obtain 8-digit primes consisting of only zero and ones:

$$\begin{align*}
10011101 &
10111001 \\
11100101 &
11110111 \\
11111101 &
\end{align*}$$

The same procedure, with $Y_{3^5-1}(x)$ and factoring over GF(3) gives the primes

$$\begin{align*}
101221 &
102101 \\
111121 &
111211 \\
112111 &
120011 \\
122021 &
\end{align*}$$

The list is created via

```plaintext
n=3^5-1; f=lift(factor(polcyclo(n)*Mod(1,3))); f=f[,1];
for(k=1, #f, v=subst(f[k],x,10); if(isprime(v), print(v)));
```

### 37.12 Complex moduli: GF($p^2$)

Recall that with real numbers the equation $x^2 = -1$ has no solution, there is no real square root of minus one. The construction of complex numbers proceeds by taking pairs of real numbers $(a, b) = a + i b$ together with (component wise) addition $(a, b) + (c, d) = (a + c, b + d)$ and multiplication defined by $(a, b) (c, d) = (ac - bd, ad + bc)$. Indeed the pairs of real numbers together with addition and multiplication as given constitute a field.

We will now rephrase the construction in a way that shows how to construct an extension field from a given ground field (or base field). In the example above the ground field are the real numbers and the extension field are the complex numbers.

### 37.12.1 The construction of complex numbers

Equivalently to saying that there is no real square root of minus one we could have said that the polynomial $x^2 + 1$ has no linear factor. The construction of the complex numbers proceeds by taking numbers of the form $a + bi$ where $i$ is boldly defined to be a root of the polynomial $x^2 + 1$. Now observe that if we identify $a + bi = b i + a$ with the polynomial $b x + a$ and use polynomial addition and multiplication modulo the polynomial $x^2 + 1$ we obtain the arithmetic of complex numbers. Addition is component wise, no modular reduction occurs. The multiplication rule can be obtained with polynomial arithmetic:

$$
(b x + a) (d x + c) \equiv (bd) x^2 + (ad + bc) x + (ac) \pmod{x^2 + 1}
$$

We used the relation $x^2 = -1$, so $u x^2 \equiv -u \pmod{x^2 + 1}$. Identify $x$ with $i$ in the relations and observe that computations with a root (that is not in the ground field) of a polynomial is equivalent to computations with polynomials (whose coefficients are in the ground field) modulo the polynomial $x^2 + 1$. Some of the newer factorization algorithms can be found in [91], readable surveys are [186] and [208]. Tables of factorizations of numbers of the form $b^e \pm 1$ are given in [74] which also contains much historical information.

It remains to say that a deterministic polynomial-time algorithm for proving primality was published by Agrawal, Kayal and Saxena in August 2002 [4]. While this a major breakthrough in mathematics it does not render the Rabin-Miller test worthless. Practically, ‘industrial grade’ primes are still produced with it, see [71]. Good introductions into the ideas behind the so-called AKS algorithm and its improvements are [137] and [98, p.200ff].

### 37.11.4.8 Further reading

Excellent introductions into topics related to prime numbers and methods of factorization are [227], [282], and [98]. Primality tests and factorization algorithms are also described in [128]. Some of the newer factorization algorithms can be found in [91], readable surveys are [186] and [208]. Tables of factorizations of numbers of the form $b^e \pm 1$ are given in [74] which also contains much historical information.
When the ground field is the real numbers the story comes to an end: every polynomial of arbitrary
degree \( n \) with complex coefficients has exactly \( n \) complex roots (including multiplicity). That is, we
cannot use the given construction to extend the field \( \mathbb{C} \) of complex numbers as every polynomial \( p(x) \)
with coefficients in \( \mathbb{C} \) is a product of linear factors:
\[
p(x) = (x - r_1)(x - r_2) \ldots (x - r_n)
\]
with \( r_k \in \mathbb{C} \).

The field \( \mathbb{C} \) is algebraically closed.

If we choose the ground field to be \( \mathbb{F}_p = \text{GF}(p) \), the integers modulo a prime \( p \), and an irreducible
polynomial \( c(x) \) of degree \( n \) whose coefficients are in \( \mathbb{F}_p \) we get an extension field
\( \mathbb{F}_{p^n} = \text{GF}(p^n) \), a finite
field with \( p^n \) elements. The special case of the binary finite fields \( \text{GF}(2^n) \) is treated in chapter 40 on
page S85.

### 37.12.2 Complex finite fields

<table>
<thead>
<tr>
<th>( a )</th>
<th>( a^1 )</th>
<th>( a^2 )</th>
<th>( a^3 )</th>
<th>( a^4 )</th>
<th>( a^5 )</th>
<th>( a^6 )</th>
<th>( a^7 )</th>
<th>( a^8 )</th>
<th>( a^9 )</th>
<th>( R = \text{maxord} = 8 )</th>
<th>( r = \text{ord}(a) = 8 )</th>
<th>( R/r = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 1 + i )</td>
<td>( 1 + i )</td>
<td>( 0 + 2i )</td>
<td>( 1 + 2i )</td>
<td>( 2 + 0i )</td>
<td>( 3 + 2i )</td>
<td>( 0 + 1i )</td>
<td>( 2 + 1i )</td>
<td>( 1 + 0i )</td>
<td>( 1 + i )</td>
<td>( \text{Mat([2, 3])} )</td>
<td>( \text{mod}(3) )</td>
<td>( \text{mod}(3) )</td>
</tr>
</tbody>
</table>

Figure 37.12-A: The powers of the element \( 1 + i \) modulo \( x^2 + 1 \) and \( p = 3 \).

With primes of the form \( p = 4k + 3 \) it is possible to construct a field of complex numbers as minus one
is a quadratic non-residue and so the polynomial \( x^2 + 1 \) is irreducible. The field is denoted by \( \text{GF}(p^2) \).

The rules for complex addition, subtraction and multiplication are the ‘usual’ ones. The field has \( p^2 \)
elements of which \( R = p^2 - 1 \) are invertible. The maximal order equals \( R \), so the inverse of any element
\( a \) can be computed as \( a^{-1} = a^{R-1} = a^{p^2-2} \).

For example, the powers of \( a = 1 + x \) modulo \( c = x^2 + x + 1 = 0 = 3 + 3i \) are shown in figure 37.12-A.

Note that the modular reduction happens with both the polynomial \( x^2 + 1 \) and \( p = 3 \). The polynomial
reduction uses \( x^2 = -1 \).

<table>
<thead>
<tr>
<th>( a^1 )</th>
<th>( a^2 )</th>
<th>( a^3 )</th>
<th>( a^4 )</th>
<th>( a^5 )</th>
<th>( a^6 )</th>
<th>( a^7 )</th>
<th>( a^8 )</th>
<th>( a^9 )</th>
<th>( a^{10} )</th>
<th>( a^{11} )</th>
<th>( a^{12} )</th>
<th>( a^{13} )</th>
<th>( a^{14} )</th>
<th>( a^{15} )</th>
<th>( a^{16} )</th>
<th>( a^{17} )</th>
<th>( a^{18} )</th>
<th>( a^{19} )</th>
<th>( a^{20} )</th>
<th>( a^{21} )</th>
<th>( a^{22} )</th>
<th>( a^{23} )</th>
<th>( a^{24} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 1 + 3x )</td>
<td>( 1 + 3x )</td>
<td>( 2 + 2x )</td>
<td>( 3 + 2x )</td>
<td>( 4 + 0x )</td>
<td>( 5 )</td>
<td>( 6 )</td>
<td>( 7 )</td>
<td>( 8 )</td>
<td>( 9 )</td>
<td>( 10 )</td>
<td>( 11 )</td>
<td>( 12 )</td>
<td>( 13 )</td>
<td>( 14 )</td>
<td>( 15 )</td>
<td>( 16 )</td>
<td>( 17 )</td>
<td>( 18 )</td>
<td>( 19 )</td>
<td>( 20 )</td>
<td>( 21 )</td>
<td>( 22 )</td>
<td>( 23 )</td>
</tr>
</tbody>
</table>

Figure 37.12-B: The powers of the element \( 1 + 3x \) modulo \( x^2 + x + 1 \) and \( p = 5 \).

With primes of the form \( p = 4k + 1 \) it also possible to construct a field \( \text{GF}(p^2) \). But we have to use a
different polynomial as \( x^2 + 1 \) is reducible modulo \( p \) and thereby the multiplication rules is different. For
example, with \( p = 5 \) we find that \( x^2 + x + 1 \) is irreducible:
The complete list of powers is shown in figure 37.12-B. We see that \( a = 1 + 3x \) has the maximal order (24), it is a primitive root. The polynomial reduction uses the relation \( x^2 = -(x + 1) \).

The values of the powers of the primitive root can be used to ‘randomly’ fill a \( p \times p \) array. With \( a^k = u + xv \) we mark the entry at row \( v \), column \( u \) with \( k \):

\[
\begin{bmatrix}
4 & 11 & 9 & 19 & 8 \\
10 & 1 & 17 & 14 & 15 \\
22 & 3 & 2 & 5 & 13 \\
16 & 20 & 7 & 21 & 23 \\
-24 & 6 & 18 & 12
\end{bmatrix}
\]

The position 0, 0 (lower left) is not visited. Note row zero is the lowest row.

As described, the procedure fills an \( p \times p \) array where \( p \) is a prime. Working with an irreducible polynomial of degree \( n \) we can fill a \( p^2 \times p^2 \times p^2 \times \ldots \times p^k \) array if \( e + f + g + \ldots + k = n \): for the exponents that are one just choose one polynomial coefficient. For the exponents greater than one (say, \( h \)) combine \( h \) polynomial coefficients \( c_0, c_1, \ldots, c_{h-1} \) and use \( z_h = c_0 + c_1 p + c_2 p^2 + \ldots + c_{h-1} p^{h-1} \).

### 37.12.3 Efficient reduction modulo certain quadratic polynomials

The polynomial \( C = x^2 + 1 \) is irreducible for primes of the form \( 4k + 3 \) (−1 is not a square)

\[
(a + b) (Ax + B) = (aA) x^2 + (a B + b A) x + (b B) \equiv (a B + b A) x + (-a A + b B) \mod x^2 + 1 \quad (37.12-2a)
\]

The last equality shows how to multiply two complex numbers at the cost of three real multiplications and five real additions instead of four multiplications and two additions.

The polynomial \( C = x^2 + d \) is irreducible if \( -d \) is not a square. We have

\[
(a + b) (Ax + B) \equiv (a B + b A) x + (-d a A + b B) \mod x^2 + d \quad (37.12-3a)
\]

If the multiplication by \( d \) is cheap (for example, if \( d = 2 \)) the implied technique can be a gain.

The polynomial \( C := x^2 + x + 1 \) has the roots \( (-1 \pm \sqrt{-3}) / 2 \) so it is irreducible modulo \( p \) if −3 is not a square modulo \( p \). The first few such primes \( p \) are

\[
2 \ 5 \ 11 \ 17 \ 23 \ 29 \ 41 \ 47 \ 53 \ 59 \ 71 \ 83 \ 89 \ 101 \ 107 \ 113 \ 131 \ 137 \ 149 \ 167 \ 173 \ 179 \ 191 \ 197 \ 227 \ 233 \ 239
\]

Multiplication modulo \( C \) costs only three scalar multiplications:

\[
(a + b) (Ax + B) = (aA) x^2 + (a B + b A) x + (b B) \equiv (-a A + b B + a B) x + (-a A + b B) \mod x^2 + x + 1 \quad (37.12-4a)
\]

For the polynomial \( C = x^2 + x + d \) use

\[
(a + b) (Ax + B) \equiv (-a A + b A + b A) x + (-d a A + b B) \mod x^2 + x + d \quad (37.12-5a)
\]
The polynomial \( C = x^2 - x - 1 \) has the roots \( (1 \pm \sqrt{5})/2 \), so it is irreducible modulo \( p \) if 5 is not a square modulo \( p \). The first few such primes are:

\[
2 \quad 3 \quad 7 \quad 13 \quad 17 \quad 23 \quad 37 \quad 43 \quad 47 \quad 53 \quad 67 \quad 73 \quad 83 \quad 97 \quad 103 \quad 107 \quad 113 \quad 127 \quad 137 \quad 157 \quad 163 \quad 167 \quad 173 \quad 193 \quad 197 \quad 223 \quad 227 \quad 233
\]

Again, multiplication modulo \( C \) costs only three scalar multiplications:

\[
(a x + b) (A x + B) = (a A) x^2 + (a b A + B) x + (b B)
\]

\[
\equiv (a A + a B + b A) x + (a A + b B) \mod x^2 - x - 1
\]

\[
= ((a + b) (A + B) - b B) x + (a A + b B)
\]  

(37.12-6a)

(37.12-6b)

(37.12-6c)

With the polynomial \( C = x^2 - x - d \) use

\[
(a x + b) (A x + B) \equiv (a A + a B + b A) x + (d a A + b B) \mod x^2 - x - d
\]

\[
= ((a + b) (A + B) - d) x + (d a A + b B)
\]  

(37.12-7a)

(37.12-7b)

For polynomials of the form \( C = x^2 - e x - d \) one has

\[
(a x + b) (A x + B) = (a A) x^2 + (a B + b A) x + (b B)
\]

\[
\equiv (e a A + a B + b A) x + (d a A + b B) \mod x^2 - e x - d
\]

\[
= ((a + b) (A + B) - [e - 1] a A - b B) x + (d a A + b B)
\]  

(37.12-8a)

(37.12-8b)

(37.12-8c)

If the multiplications by \( e - 1 \) and \( d \) are cheap then the last equality can be useful. For example, with the polynomial \( C = x^2 - 2 x - 1 \) use

\[
(a x + b) (A x + B) \equiv (2 a A + a B + b A) x + (a A + b B) \mod x^2 - 2 x - 1
\]

\[
= ((a + b) (A + B) - b B + 2 a A) x + (a A + b B)
\]  

(37.12-9a)

(37.12-9b)

With \( C = x^2 - 3 x \pm 1 \) use

\[
(a x + b) (A x + B) \equiv (3 a A + a B + b A) x + (\mp a A + b B) \mod x^2 - 3 x \pm 1
\]

\[
= ((a + b) (A + B) - b B + 2 a A) x + (\mp a A + b B)
\]  

(37.12-10a)

(37.12-10b)

### 37.12.4 An algorithm for primitive \( 2^j \)-th roots

For primes with the lowest \( k \) bits set \( p \equiv (2^k - 1) \mod 2^k \) the largest power of two dividing the maximal order in \( \text{GF}(p^2) \) equals \( N = 2^{k+1} \); \( p = j 2^k - 1 \) with \( j \) odd, so \( p+1 = j 2^k \) and \( p-1 = j 2^k - 2 = 2 (j 2^{k-1} - 1) \), thereby \( p^2 - 1 = 2^{k+1} [j (j 2^k - 1)] \) where the term in square brackets is odd.

An algorithm for the construction of primitive \( 2^j \)-th roots in \( \text{GF}(p^2) \) for \( j = 2, 3, \ldots, a \) where \( 2^a \) is the largest power of two dividing \( p^2 - 1 \) is given in \([123]\) (and also in \([45]\)):

Let \( u_2 := 0 \) and for \( j > 2 \) define

\[
u_j := \begin{cases} 
(u_{j-1} + 1)/2 \quad & \text{if } j < a \\
(u_{j-1} - 1)/2 \quad & \text{if } j = a
\end{cases}
\]  

(37.12-11)

and (for \( j = 2, 3, \ldots, a \))

\[
v_j := \begin{cases} 
(1 - u_j^2)^{(p+1)/4} \quad & \text{if } j < a \\
(1 - u_j^2)^{(p+1)/4} \quad & \text{if } j = a
\end{cases}
\]  

(37.12-12)

where all operations are modulo \( p \). Then \( u_j + i v_j \) is a primitive \( 2^j \)-th root of unity in \( \text{GF}(p^2) \).

For example, with \( p = 127 \) (and field polynomial \( x^2 + 1 \)) we obtain
37.12: Complex moduli: $\text{GF}(p^2)$

\begin{align*}
\text{j:} & \quad u_j \quad v_j \\
2: & \quad \text{ord}(0 + i*1) = 4 \\
3: & \quad \text{ord}(8 + i*8) = 8 \\
4: & \quad \text{ord}(103 + i*21) = 16 \\
5: & \quad \text{ord}(68 + i*87) = 32 \\
6: & \quad \text{ord}(15 + i*41) = 64 \\
7: & \quad \text{ord}(32 + i*82) = 128 \\
8: & \quad \text{ord}(98 + i*38) = 256 \\
\end{align*}

For Mersenne primes $p = 2^e - 1$ one has $p^2 - 1 = (p + 1)(p - 1) = 2^e(2^e - 2) = 2^{e+1}(2^{e-1} - 1) = 2^a k$ where $k$ is odd. The highest power of two for which a primitive root exists is $2^a$ where $a = e + 1$ which checks with our example where $p = 127 = 2^7 - 1$.

### 37.12.5 Primitive $2^j$-th roots with Mersenne primes

For Mersenne primes $p = 131071 = 2^{17} - 1$ one has $p^2 - 1 = (p + 1)(p - 1) = 2^{18}$. By squaring $e - 2$ times, then compute $1/\sqrt{3} = 2^{e-1}/2$ which does not require modular reduction. Now

$$z := \frac{1}{\sqrt{2}} \left( 1 + \sqrt{3} \right) = \frac{1}{\sqrt{2}} \left( 1 + i \sqrt{3} \right)$$

is an element of order $2^{e+1}$. The number $z$ is sometimes called the Creutzburg-Tasche primitive root as the construction was given in [99, p.200]. We have $z^2 = 2 + \sqrt{3} = 2 + i \sqrt{3} = H_j + i \sqrt{3} U_j$, and

$$z^{2^k} = H_{2^{k-1}} + i \sqrt{3} U_{2^{k-1}} \quad \text{for} \quad (k \geq 1)$$

Figure 37.12-C shows the values of the successive $2^k$-th powers of $z$ in $\text{GF}(p^2)$ where $p = 2^{17} - 1$.

The sequences $H = 2, 7, 97, 18817, \ldots$ and $U = 1, 4, 56, 10864, \ldots$ are those which appear in the Lucas-Lehmer test. The order of $z^{2^k}$ is $2^{e+1-k}$.

We have $H_{2j} = 2 H_j^2 - 1$ and $H_j^2 - 3 U_j^2 = 1$, thereby

$$H_{2j} = H_j^2 + 3 U_j^2 \quad (37.12-15a)$$

$$H_{2j} = 2 H_j U_j \quad (37.12-15b)$$

These are the index doubling formulas for the convergents of the continued fraction of $\sqrt{3}$. 

[fatbook draft of 2008-August-17]
37.12.6 Cosine and sine in GF($p^2$)

Let $z$ be an element of order $n$ in GF($p^2$), we would like to identify $z$ with $\exp(2i\pi/n)$ and determine the values equivalent to $\cos(2\pi/n)$ and $\sin(2\pi/n)$. One can set

$$
\cos \frac{2\pi}{n} := \frac{z^2 + 1}{2z} \tag{37.12-16a}
$$

$$
i \sin \frac{2\pi}{n} := \frac{z^2 - 1}{2z} \tag{37.12-16b}
$$

For the choice of $\sin$ and $\cos$ the relations $\exp(x) = \cos(x) + i\sin(x)$ and $\sin(x)^2 + \cos(x)^2 = 1$ should hold. The first check is trivial: $\frac{z^2 + 1}{2z} + \frac{z^2 - 1}{2z} = 1$. The second is also easy if we write $i$ for some element that is the square root of $-1$: $(\frac{z^2 + 1}{2z})^2 + (\frac{z^2 - 1}{2z}i)^2 = 1$.

For the $2^j$-th roots in GF(127$^2$) we obtain

<table>
<thead>
<tr>
<th>$u + iv$</th>
<th>$\cos$</th>
<th>$i \sin$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: $1 + i\cdot 0$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1: $126 + i\cdot 0$</td>
<td>126</td>
<td>0</td>
</tr>
<tr>
<td>2: $0 + i\cdot 1$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3: $8 + i\cdot 8$</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4: $103 + i\cdot 21$</td>
<td>103</td>
<td>1</td>
</tr>
<tr>
<td>5: $68 + i\cdot 87$</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>6: $15 + i\cdot 41$</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>7: $32 + i\cdot 82$</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>8: $96 + i\cdot 38$</td>
<td>38</td>
<td>1</td>
</tr>
</tbody>
</table>

Note how the $i$ swaps side with the element of highest order $2^a$.

The construction allows us to mechanically convert fast Fourier (or Hartley) transforms with explicit trigonometric constants into the corresponding number theoretic transforms. The idea of expressing cosines and sines in terms of primitive roots was taken from [259].

37.12.7 Cosine and sine in GF($p$)

What about primes of the form $p = 4k + 1$ that are used anyway for NTTs? The same construction works. The polynomial $x^2 + 1$ is reducible modulo $p = 4k + 1$, equivalently, $-1$ is a quadratic residue so its square root lies in GF($p$). We could say: $i$ is real modulo $p$ if $p$ is of the form $4k + 1$.

In the implementation [FXT: class mod in mod/mod.h] the cosine and sine values are computed from the primitive roots of order $2^t$. The program [FXT: mod/modsincos-demo.cc] generates the list of $2^t$-th roots and inverse roots shown in figure 37.12-D.

Again we can translate a given FFT implementation in a mechanical way.

An element modulo a prime $p = k \cdot 2^t + 1$ whose order equals $2^t$ can be found by the following algorithm even if the factorization of $k$ is not known: Choose a random $a$ where $1 < a < p - 1$ and compute $s = a^k$, if $-1 = s^{t-1}$ then return $s$, else try another $a$.

The algorithm terminates when the first element $a$ is encountered whose order has the factor $2^t$. An implementation that tests $a = 2, 3, \ldots, p - 2$ sequentially is

```c
1  e12(k, t)=
2  {
3    local(p, s);
4    p = k2^t+1;
5    for(a=2, p-2, s = Mod(a,p)^k; if( Mod(-1,p)==s*(2^-(t-1)), return( s ); ); );
6  }
```

With $p = 314151729239163 \cdot 2^{26} + 1$ the algorithm terminates after testing $a = 5$ (of order $(p-1)/3$) and returning $s = 1858378138645552528042$ whose order is indeed $2^{26}$.

In general, if $p = u \cdot f + 1$, $\gcd(f, u) = 1$ and $f = \prod p_i^{c_i}$ is fully factored then an element of order $f$ can be determined by testing random values $a$:
37.12: Complex moduli: \( \text{GF}(p^2) \)

**modulus= 257 == 0x101**

**modulus is cyclic**

**modulus is prime**

bits(modulus)= 8.0056245 == 9 - 0.99437545
euler_phi(modulus)= 256 == 0x100 == 2^8
maxorder= 256 == 0x100
maxordelem= 3 == 0x3
max2pow= 8 (max FFT length = 2**8 == 256)
root2pow(max2pow)=3 root2pow(-max2pow)=86

\[ \sqrt{-1} =: i = 241 \]

\[ \begin{align*}
8: & \quad z= 3 = ( 173 + 87) = ( 173 + 107\cdot i) \\
7: & \quad z= 9 = ( 233 + 33) = ( 233 + 14\cdot i) \\
6: & \quad z= 81 = ( 123 + 215) = ( 123 + 99\cdot i) \\
5: & \quad z= 136 = ( 188 + 205) = ( 188 + 196\cdot i) \\
4: & \quad z= 249 = ( 12 + 237) = ( 12 + 194\cdot i) \\
3: & \quad z= 64 = ( 30 + 34) = ( 30 + 30\cdot i) \\
2: & \quad z= 241 = ( 0 + 241) = ( 0 + 1\cdot i) \\
1: & \quad z= 256 = ( 256 + 0) = ( 256 + 0\cdot i) \\
\end{align*} \]

**Figure 37.12-D:** Roots of order 2\(^j\) modulo \( p = 257 = 2^8 + 1 \).

1. Take a random \( a \) and set \( s = a^u \).
2. If \( s^{f/p_i} \neq 1 \) for all prime factors \( p_i \) of \( f \), then return \( s \) (an element of order \( f \)).
3. Go to step 1.

### 37.12.8 Decomposing \( p = 4k + 1 \) as sum of two squares

We give algorithms to decompose a prime \( p = 4k + 1 \) as a sum of two squares.

#### 37.12.8.1 Direct computation

The direct way to determine \( u \) and \( v \) with \( n = u^2 + v^2 \) is to check, for \( v = 1, 2, \ldots, \lfloor \sqrt{n} \rfloor \), whether \( n - v^2 \) is a perfect square. If so, return \( u = \sqrt{n - v^2} \) and \( v \):

```
1 sumofsquares_naive(k, t)=
2 { /* return [u,v] such that u^2+v^2==p =k*2^t+1 */
3    local(n, w);
4    n = k*2^t+1;
5    for (v=1, sqrtint(n), \ \ \ search until n-v^2 is a square
6      w = n-v^2;
7      if ( issquare(w), return( [sqrtint(w), v] );
8    );
9    return ( 0 ); \ \ \ not the sum of two squares
10 }
```

The routine needs at most \( \lfloor \sqrt{n} \rfloor \) steps which renders it rather useless for \( n \) large. With the prime \( n = 314151729239163 \cdot 2^{26}+1 \approx 2 \cdot 10^{22} \) we have \( n = u^2+v^2 \) where \( u = 132599472793 \) and \( v = 59158646772 \) and the routine would need \( v \) steps to find the solution. The method described next finds the solution immediately.

[fxtbook draft of 2008-August-17]
37.12.8.2 Computation using continued fractions

The square root $i$ of $-1$ can be used to find the representation of a prime $p = 4k + 1$ as a sum of two squares, $p = u^2 + v^2$, as follows:

1. Determine $i$ where $i^2 = -1$ modulo $p$. If $i \geq p/2$ then set $i = p - i$.

2. Compute the continued fraction of $p/i$, it has the form $[a_0, a_1, \ldots, a_n, a_n, \ldots, a_1, a_0]$.

3. Compute the numerators of the $(n-1)$-st and the $n$-th convergent, $P_{n-1}$ and $P_n$. Return $u = P_{n-1}$ and $v = P_n$.

Assume that $p = k \cdot 2^t + 1$ where $t \geq 2$. Use an element of order $2^t$ to find a square root of $-1$:

```plaintext
imag4k1(k, t)=
{ /* determine s such that s^2=-1 modulo p=k*2^t+1 */
  local(s);
  s = el2(k, t);
  s = s^(2^(t-2));
  return( s );
}
```

Now the decomposition as a sum of two squares can be found with

```plaintext
sumofsquares(k, t)=
{ /* return [u,v] such that u^2+v^2==p =k*2^t+1 */
  local(i, s, p, cf, q, u, v);
  i = component( imag4k1(k, t), 2);
  p = k*2^t+1;
  if ( i\geq p/2, i = p-i );
  cf = contfrac(p/i);
  cf = vector(length(cf)/2, j, cf\[j\]);
  q = contfracpnqn(cf);
  u = q\{1, 1\}; v = q\{1, 2\};
  return( \[u, v\] );
}
```

An example, for $p = 2281$ we obtain

```
1571 \ square root of -1
710 \ choose smaller square root
cf = [3, 4, 1, 2, 2, 1, 4, 3] \ contfrac(2281/710)
cf = [3, 4, 1, 2] \ first half on contfrac
q = contfracpnqn(cf) =
[45 16] \ P_4, P_3
[14 5] \ Q_4, Q_3 (unused)
```

```plaintext
u=45; v=16; \ u^2 + v^2 = 2025 + 256 = 2281
```

37.12.8.3 A memory saving version

An algorithm that avoids storing the continued fraction comes from the observation that $u$ and $v$ appear in the calculation of gcd($p, i$). With

```plaintext
gcd_print(p, i)=
{ local( t, s );
  if ( p<i, t=p; p=i; i=t; );
  s = sqrtint(p);
  while ( i,
    print (" ", p, " ", i);
    t = p % i; p = i; i = t;
  );
}
```

and $p = 2281, i = 710$ we obtain

```
2281 710
710 161
161 106
106 45
45 16
```
The marked pair is the first where $u^2 < p$. The resulting routine is

```c
1  sumofquares_gcd(k, t)=
2  { /* return [u,v] such that u^2+v^2==p =k*2^t+1 */
3    local(s, p, i, w);
4    i = component( imag4k1(k, t), 2);
5    p = k*2^t+1;
6    if ( i>=p/2, i = p-i );
7    w = sqrtint(p);
8    while ( i,
9      if ( p<=w, return( [p,i] ) );
10       t = p % i; p = i; i = t;
11     );
12    return( [0,0] ); \ failure
13  }
```

We note that by the relation $a^2 + b^2 = (a + ib)(a - ib)$ we can use the decomposition into two squares to obtain the factorization of a number over the complex integers. For example, we have $3141592653 = 3 \cdot 107 \cdot 9786893$ (over $\mathbb{Z}$) where the greatest prime factor is of the form $4k+1$. Using $9786893 = 2317^2 + 2102^2$ one can obtain $3141592653 = -i \cdot 3 \cdot 107 \cdot (2317 + 2102i) \cdot (2102 + 2317i)$. Pari/gp has a builtin routine for this task:

```plaintext
? factor(3141592653+0*I)
[-I 1] [3 1] [107 1] [2317 + 2102*I 1] [2102 + 2317*I 1]
```

### 37.13 Solving the Pell equation

Simple continued fractions can be used to find integer solutions of the equations

\[
x^2 - dy^2 = +1 \quad \text{(37.13-1a)}
\]
\[
x^2 - dy^2 = -1 \quad \text{(37.13-1b)}
\]

Equation (37.13-1a) is usually called the Pell equation. The name Bhaskara equation (or Brahmagupta-Bhaskara equation, used in [178]) has been suggested because Brahmagupta (ca. 600 AD) and Bhaskara (ca. 1100 AD) were the first to study and solve this equation.

#### 37.13.1 Solution via continued fractions

The convergents $P_k/Q_k$ of the continued fraction of $\sqrt{d}$ are close to $\sqrt{d}$: $(P_k/Q_k)^2 \approx d$. If we define $e_k := P_k^2 - dQ_k^2$ then solutions of relation (37.13-1a) correspond to $e_k = +1$, solutions of (37.13-1b) to $e_k = -1$.

As an example we set $d = 53$. The continued fraction of $\sqrt{53}$ is

\[
\text{CF}(\sqrt{53}) = [7, 3, 1, 1, 3, 14, 3, 1, 1, 3, 14, 3, 1, 1, 3, 14, \ldots] \quad \text{(37.13-2a)}
\]
\[
= [7, 3, 1, 1, 3, 14] \quad \text{(37.13-2b)}
\]

We observe that the sequence is periodic after the initial term and the last term of the period is twice the initial term. Moreover, disregarding the term 14, the terms in the period form a palindrome. These properties actually hold for all simple continued fractions of square roots $\sqrt{d}$ with $d$ not a perfect square, for the proofs see [213] or [178]. For the computation of the continued fraction of a square root a specialized version of the algorithm from section 35.3.1.2 on page 715 will be most efficient.

The table shown in figure 37.13-A gives the first convergents $P_k/Q_k$ together with $e_k := P_k^2 - 53Q_k^2$. The entry for $k = 4$ corresponds to the smallest solution $(x,y)$ of $x^2 - 53y^2 = -1$: $18^2 - 53 \cdot 25^2 = -1$. Entry $k = 9$ corresponds to $6624^2 - 53 \cdot 9100^2 = +1$, the smallest nontrivial solution to $x^2 - 53y^2 = +1$ (the trivial solution is $(P_{-1}, Q_{-1}) = (1,0))$.

[fxtbook draft of 2008-August-17]
### Chapter 37: Modular arithmetic and some number theory

#### Figure 37.13-A: The first convergents $P_k/Q_k$ of the continued fraction of $\sqrt{53}$.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$a_k$</th>
<th>$P_k$</th>
<th>$Q_k$</th>
<th>$e_k := P_k^2 - dQ_k^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>-4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>22</td>
<td>3</td>
<td>+7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>29</td>
<td>4</td>
<td>-7</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>51</td>
<td>7</td>
<td>+4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>182</td>
<td>25</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>2599</td>
<td>357</td>
<td>+4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>7979</td>
<td>1096</td>
<td>-7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>10578</td>
<td>1453</td>
<td>+7</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>18557</td>
<td>2549</td>
<td>-4</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>66249</td>
<td>9100</td>
<td>+1</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>946043</td>
<td>129949</td>
<td>-4</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>2904378</td>
<td>398947</td>
<td>+7</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>3850421</td>
<td>528896</td>
<td>-7</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>6754799</td>
<td>927843</td>
<td>+4</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>24114818</td>
<td>3312425</td>
<td>-1</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>344362251</td>
<td>47301793</td>
<td>+4</td>
</tr>
</tbody>
</table>

#### Figure 37.13-B: The first convergents $P_k/Q_k$ of the continued fraction of $\sqrt{19}$.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$a_k$</th>
<th>$P_k$</th>
<th>$Q_k$</th>
<th>$e_k := P_k^2 - dQ_k^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>+5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>13</td>
<td>3</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>48</td>
<td>11</td>
<td>+5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>61</td>
<td>14</td>
<td>-3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>170</td>
<td>39</td>
<td>+1</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>1421</td>
<td>326</td>
<td>-3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3012</td>
<td>691</td>
<td>+5</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>4433</td>
<td>1017</td>
<td>-2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>16311</td>
<td>3742</td>
<td>+5</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>20744</td>
<td>4759</td>
<td>-3</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>57799</td>
<td>13260</td>
<td>+1</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>483136</td>
<td>110839</td>
<td>-3</td>
</tr>
</tbody>
</table>
37.13: Solving the Pell equation

With \( d = 19 \) we obtain the continued fraction

\[
\text{CF}(\sqrt{19}) = [4, 2, 1, 3, 1, 2, 8, 2, 1, 3, 1, 2, 8, 2, 1, 3, 1, 2, 8, \ldots] \quad (37.13-3a)
\]

\[
= [4, 2, 1, 3, 1, 2, 8] \quad (37.13-3b)
\]

with period length \( l = 6 \). The corresponding table (figure 37.13-B) contains solutions with \( e_k = +1 \) but none with \( e_k = -1 \):

Let \( e \) correspond the minimal nontrivial solution of \( x^2 - dy^2 = \pm 1 \), if \( e = +1 \) then no solution for \( x^2 - dy^2 = -1 \) exists. Nontrivial solutions with \( e = +1 \) always exist, solutions with \( e = -1 \) only exist when the period length \( l \) of the continued fraction of \( \sqrt{d} \) is odd. The period length \( l \) is always odd for primes of the form \( p = 4k + 1 \) and never for numbers of the form \( n = 4k + 3 \) and \( 4k \). If any factor \( f_i \) of \( d \) is of the form \( f_i = 4k + 3 \) then no solution with \( e = -1 \) exists, because this would imply \( x^2 \equiv -1 \) mod \( f_i \), but \(-1\) is never a quadratic residue modulo \( f_i = 4k + 3 \) by relation 37.8-3a on page 781.

However, all prime factors being of the form \( 4k + 1 \) does not guarantee that \( e = -1 \), the smallest examples are \( 205 = 5 \cdot 41, 221 = 13 \cdot 17, 305 = 5 \cdot 61, \) and \( 377 = 13 \cdot 29 \). The list of such numbers up to 2500 is

\[
205, 221, 305, 377, 505, 545, 689, 725, 745, 793, 905, 1205, 1345, 1405, 1469, 1513, 1517, 1537, 1717, 1885, 1945, 1961, 2005, 2041, 2045, 2105, 2225, 2245, 2329, 2353
\]

The sequence of numbers \( d \) with no factor of the form \( 4k + 3 \) such that \( x^2 - dy^2 = -1 \) has no solution is entry A031399 of [245]:

\[
4, 8, 16, 20, 25, 32, 34, 40, 52, 64, 68, 80, 100, 104, 116, 128, 136, 146, 148, 160, 164, 169, 178, 194, 200, 205, 208, 212, 221, 232, 244, 256, 260, 272, 289, 292, 296, \ldots
\]

An algorithm for computing solutions \( (x, y) \) of the equation \( Ax^2 - By^2 = N \) is given in [200].

37.13.2 Multiplying and powering solutions

Consider two solutions \( (x, y) \) and \( (r, s) \) of the Pell equation

\[
x^2 - dy^2 = e \quad (37.13-4a)
\]

\[
r^2 - ds^2 = f \quad (37.13-4b)
\]

where \( e = \pm 1 \) and \( f = \pm 1 \). Now write

\[
x^2 - Dy^2 = (x + \sqrt{d} y) (x - \sqrt{d} y) \quad (37.13-5)
\]

and the same for \( (r, s) \). We compute the products

\[
\begin{align*}
(x + \sqrt{d} y) (r + \sqrt{d} y) &= (xr + dy s) + \sqrt{d} (xs + yr) \quad (37.13-6a) \\
(x - \sqrt{d} y) (r - \sqrt{d} y) &= (xr + dy s) - \sqrt{d} (xs + yr) \quad (37.13-6b)
\end{align*}
\]

By multiplying both relations we see that \( (U, V) := (xr + dy s, xs + yr) \) is also a solution:

\[
U^2 - \sqrt{d} V^2 = ef \quad (37.13-7)
\]

Now let \( (r, s) \) be the smallest nontrivial solution, and define \( (x_k, y_k) \) by

\[
\begin{pmatrix} x_k \\ y_k \end{pmatrix} := \begin{pmatrix} r & ds \k{+1} \\ s & r \end{pmatrix} \quad (37.13-8)
\]

Then \( (x_k, y_k) \) is the \( k \)-th solution of the Pell equation.

Let \( r^2 - ds^2 = e \), then we have \( x_k^2 - dy_k^2 = e^k \). Therefore, if \( r^2 - ds^2 = +1 \) then there is no solution \( (x, y) \) such that \( x^2 - dy^2 = -1 \). If \( r^2 - ds^2 = -1 \) then \( x_k^2 - dy_k^2 = -1 \) for all odd \( k \).

[fxtbook draft of 2008-August-17]
As we can multiply solutions, we can also raise them to any power. Let \((x, y)\) be such that \(x^2 - dy^2 = e\) where \(e = \pm 1\), and define the matrix \(M\) by

\[
M := \begin{bmatrix} x & dy \\ y & x \end{bmatrix}
\]  

(37.13-9)

The \(k\)-th power of the solution \((x, y)\) is

\[
(x, y)^k = M^k \begin{bmatrix} 1 \\ 0 \end{bmatrix}
\]  

(37.13-10)

Now write \((X_k, Y_k)\) for the \(k\)-th power of \((x, y)\). We have, for the squared solution,

\[
X_2 = x^2 + dy^2 = 2x^2 - e = 2dy^2 + e
\]

(37.13-11a)

\[
Y_2 = 2xy
\]

(37.13-11b)

And for the third power

\[
X_3 = x(x^2 + 3dy^2) = x(4x^2 - 3e)
\]

(37.13-12a)

\[
Y_3 = y(3x^2 + dy^2) = y(4x^2 - e) = y(4dy^2 + 3e)
\]

(37.13-12b)

Note that the last equality in the first relation expresses \(X_3\) solely in terms of \(x\), \(d\), and \(e\), and the last equality in the second relation expresses \(Y_3\) solely in terms of \(y\), \(d\), and \(e\). Therefore \(X_{3k}\) and \(Y_{3k}\) can be computed independently.

The relations 37.13-11a and 37.13-11a are just the numerator and denominator of the second order iteration for \(\sqrt{d}\), relation 28.2-2a on page 572. Relations 37.13-12a and 37.13-12b correspond to the third order iteration, relation 28.2-2b.

If the pair \((x, y)\) is a solution with \(e = +1\), then \((T_n(x), yU_{n-1}(x))\) is also a solution, where \(T_n\) and \(U_n\) are the Chebyshev polynomials of the first and second kind:

\[
T^2_n(x) - d(yU_{n-1}(x))^2 = T^2_n(x) - dy^2U_{n-1}(x)^2 = 1
\]

(37.13-13a)

\[
T^2_n(x) - (x^2 - 1)U_{n-1}(x)^2 = 1
\]

(37.13-13b)

The last equality is relation 34.2-25 on page 684. Similarly, if \((x, y)\) is a solution with \(e = -1\), then \((T_n^+(x), yU_{n-1}^+(x))\) is also a solution, by equation 34.2-30 on page 685.

### 37.14 Multiplication of hypercomplex numbers *

An \(n\)-dimensional vector space (over a field) together with component-wise addition and a multiplication table that defines the product of any two (vector) components defines an algebra.

The product of two elements \(x = \sum_k \alpha_k e_k\) and \(y = \sum_j \beta_j e_j\) of the algebra is defined as

\[
x \cdot y = \sum_{k,j=0}^{n-1} [(\alpha_k \cdot \beta_j) e_k e_j]
\]

(37.14-1)

The quantities \(e_k e_j\) are given in the multiplication table of the algebra. These can be arbitrary elements of the algebra, that is, linear combinations of the components. For example, a 2-dimensional algebra over the reals could have the following multiplication table:

<table>
<thead>
<tr>
<th></th>
<th>e0</th>
<th>e1</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0:</td>
<td>(5<em>e1 + 3</em>e0)</td>
<td>(239<em>e0 + 3.1415</em>e1)</td>
</tr>
<tr>
<td>e1:</td>
<td>(0)</td>
<td>(17<em>e1 + 2.71828</em>e0)</td>
</tr>
</tbody>
</table>

[fxtbook draft of 2008-August-17]
Note that there is no neutral element of multiplication (‘one’). Further, the algebra has zero divisors: the equation \( x \cdot y = 0 \) has a solution where neither element is zero, namely \( x = e_1 \) and \( y = e_0 \). As almost all randomly defined algebras, it is completely uninteresting.

In what follows we will only consider algebras over the reals where the product of two components equals \( \pm 1 \) times another component. For example, the complex numbers are a two-dimensional algebra (over the real numbers) with the multiplication table

\[
\begin{array}{c|cccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 1 & 0 & -3 & -2 & -1 & -6 & -5 & -4 \\
2 & -3 & -2 & 0 & 1 & 2 & 3 & 4 & 5 \\
3 & -9 & -8 & -7 & -6 & -5 & -4 & -3 & -2 \\
4 & -5 & -4 & -3 & -2 & 0 & 1 & 2 & 3 \\
5 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
6 & 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 \\
7 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\end{array}
\]

We will denote the components of an \( n \)-dimensional algebra by the numbers 0, 1, \ldots, \( n - 1 \). The multiplication table for the complex numbers would thus be written as

\[
\begin{array}{c|cccc}
  & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

\[37.14.1 \text{ The Cayley-Dickson construction}\]

The Cayley-Dickson construction recursively defines multiplication tables for certain algebras where the dimension is a power of two. Let \( a, A, b \) and \( B \) be elements of a \( 2^n-1 \)-dimensional algebra \( U \). Define the multiplication rule for an algebra \( V \) (of dimension \( 2^n \)) written as pairs of elements of \( U \) via

\[
(a, b) \cdot (A, B) := (a \cdot A - B \cdot b^*, a^* \cdot B + A \cdot b)
\]

where the conjugate \( C^* \) of an element \( C = (a, b) \) is defined as

\[
(a, b)^* := (a^*, -b)
\]

and the conjugate of a real number \( a \) equals \( a \) (unmodified). The construction leads to multiplications tables where the product of two units equals plus or minus one other unit only: \( e_i \cdot e_j = \pm e_k \forall i, j \).

**Figure 37.14-A:** Multiplication table for the sedenions. The entry in row \( R \), column \( C \) gives the product \( R \cdot C \) of the components \( R \) and \( C \) (hexadecimal notation).

Figure [37.14-A] gives the multiplication table for a 16-dimensional algebra, the so-called sedenions. The upper left (8 \( \times \) 8) quarter gives the multiplication rule for the octonions (or Cayley numbers), the upper left 4 \( \times \) 4 subtable gives the rule for the quaternions and the upper left 2 \( \times \) 2 subtable corresponds to the
complex numbers. Note that multiplication is in general neither commutative (only up to dimension 2) nor associative (only up to dimension 4).

The $2^n$-dimensional algebras are (for $n > 1$) referred to as *hypercomplex numbers*. There is no generally accepted naming scheme for the algebras beyond dimension 16. We will use the names ‘$2^n$-ions’.

This form (relation 37.14-2) of the construction is given in [23], an alternative form is used in [111]:

$$ (a, b) \cdot (A, B) := (a \cdot A - B^* \cdot b, b \cdot A^* + B \cdot a) $$

It leads to a table that is the transposed of figure 37.14-A.

![Figure 37.14-B: Signs in the multiplication table for sedenions.](image)

By construction, $e_k^2 = e_0$, $e_k^2 = -e_0$, $e_0 e_k = e_k e_0 = e_k$, and $e_k e_j = -e_j e_k$ whenever both of $k$ and $j$ are nonzero (and $k \neq j$). Further,

$$ e_k e_j = \pm e_x \quad \text{where} \quad x = k \text{XOR} \ j $$

where the sign is to be determined. Figure 37.14-B shows the pattern of the signs of the sedenion algebra. The lower left quarter is the transposed of the upper left quarter, so is the lower right quarter, except for its top row. The upper right quarter is (except for its first row) the negated upper left quarter. These observations, together with the partial antisymmetry can be cast into an algorithm to compute the signs.

```c
1 int CD_sign_rec(ulong r, ulong c, ulong n)
2 // Signs in the multiplication table for the
3 // algebra of n-ions (where n is a power of two)
4 // that is obtained by the Cayley-Dickson construction:
5 // If component r is multiplied with component c then the
6 // result is CD_sign_rec(r,c,n) * (r XOR c).
7 {
8     if ( (r==0) || (c==0) ) return +1;
9     if ( c>=r )
10        if ( c>r ) return -CD_sign_rec(c, r, n);
11           else return -1; // r==c
12     }
13     // here r>c (triangle below diagonal)
14     ulong h = n>>1;
15     if ( c==h ) // right
16        { return CD_sign_rec(c, r-h, h); // upper right not reached
17   } else // left
18        { return CD_sign_rec(r, c, h); // upper left
19   }
20 }
```
The function uses at most $2 \cdot \log_2(n)$ steps. Note that the second row in the table is (the signed version of) the Thue-Morse sequence, see section 1.16.1 on page 44. The matrix filled with entries $\pm 1$ according to figure 37.14-B is a Hadamard matrix, see section 18 on page 373. The sequence of signs, read by anti-diagonals, and setting $0 := +$ and $1 := -, is entry A118685 of [245].

An iterative version of the function is [FXT: arith/cayley-dickson-demo.cc]:

```cpp
inline void cp2(ulong a, ulong b, ulong &u, ulong &v) { u=a; v=b; }
//
inline int CD_sign_it(ulong r, ulong c, ulong n)
{
int s = +1;
start:
if ( (r==0) || (c==0) ) return s;
if ( c==r ) return -s;
if ( c>r ) { swap2(r,c); s=-s; }
n >>= 1;
if ( c>=n ) cp2(c-n, r-n, r, c);
else if ( r>=n ) cp2(c, r-n, r, c);
goto start;
}
```

The rate of generation with the computation of all $2^{24}$ signs in the multiplication table for the ‘$2^{12}$-ions’ is about 11.5 M/s, and about 9 M/s with the recursive routine.

![Figure 37.14-C: Alternative multiplication table for the octonions (left) and its sign pattern (right).](image)

An alternative multiplication table for the octonions is given in figure 37.14-C. Its sign pattern is the $8 \times 8$ Hadamard matrix shown in figure 18.1-A on page 374. Properties of this representation and the relation to shift register sequences are given in [105].

### 37.14.2 Fast multiplication of quaternions

![Figure 37.14-D: Scheme for the length-4 dyadic convolution (left), same with bucket zero negated (middle) and the multiplication table for the units of the quaternions (right). The asterisks mark those entries where the sign is different from the scheme in the middle.](image)

Quaternion multiplication can be achieved in eight real multiplications using the dyadic convolution (see section 22.7 on page 473): the scheme in figure 37.14-D suggests to use the dyadic convolution with bucket zero negated as a starting point which costs 4 multiplications. Some entries have to be corrected which costs four more multiplications.

```cpp
// f[] == [ re1, i1, j1, k1 ]
// g[] == [ re2, i2, j2, k2 ]
```
Chapter 37: Modular arithmetic and some number theory

Figure 37.14-E: Scheme for the length-8 dyadic convolution with bucket zero negated (left) and multiplication table for the octonions (middle, taken from [111]). There are 22 places where the signs differ (right, marked with '#'). This leads to an algorithm involving $8 + 22 = 30$ multiplications.

```plaintext
3 \hspace{1em} c0 := f[0] * g[0]
4 \hspace{1em} c1 := f[3] * g[2]
5 \hspace{1em} c2 := f[1] * g[3]
6 \hspace{1em} c3 := f[2] * g[1]
7 // length-4 dyadic convolution:
8 \hspace{1em} walsh(f[])
9 \hspace{1em} walsh(g[])
10 \hspace{1em} for i:=0 to 3 g[i] := (f[i] * g[i])
11 \hspace{1em} walsh(g[])
12 \hspace{1em} // normalization and correction:
13 \hspace{1em} g[0] := 2 * c0 - g[0] / 4
14 \hspace{1em} g[1] := - 2 * c1 + g[1] / 4
15 \hspace{1em} g[2] := - 2 * c2 + g[2] / 4
16 \hspace{1em} g[3] := - 2 * c3 + g[3] / 4

The algorithm is taken from [152] which also gives a second variant.

The complex multiplication by three real multiplications (relation 37.12-2c on page 805) corresponds to one length-2 Walsh dyadic convolution and the correction for the product of the imaginary units:

```plaintext
1 // f[] == [ re1, im1 ]
2 // g[] == [ re2, im2 ]
3 c0 := f[1] * g[1] // == im1 * im2
4 // length-2 dyadic convolution:
5 \{ f[0], f[1] \} := \{ f[0]+f[1], f[0]-f[1] \}
6 \{ g[0], g[1] \} := \{ g[0]+g[1], g[0]-g[1] \}
7 g[0] := f[0] * g[0]
8 g[1] := f[1] * g[1]
9 \{ g[0], g[1] \} := \{ g[0]+g[1], g[0]-g[1] \}
10 // normalization:
11 f[0] := f[0] / 2
12 g[0] := g[0] / 2
13 // correction:
14 g[0] := -2 * c0 + g[0]
15 // here: g[] == [ re1 * re2 - im1 * im2, re1 * im2 + im1 * re2 ]
```

For complex numbers of high precision multiplication is asymptotically equivalent to two real multiplications as one FFT based (complex linear) convolution can be used for the computation. Similarly, high precision quaternion multiplication is as expensive as four real multiplications. Figure 37.14-E shows an equivalent construction for the octonions leading to an algorithm with 30 multiplications.
Chapter 38

Binary polynomials

We introduce binary polynomials and their arithmetic. We describe tests for irreducibility, primitivity, and a method for factorization. Many of the algorithms shown can easily be implemented in hardware. The arithmetic operations with binary polynomials are the underlying methods for computations in binary finite fields which are treated in chapter 40. An important application are the linear feedback shift registers, they are described in chapter 39.

A polynomial with coefficients in the field GF(2) = \( \mathbb{Z}/2\mathbb{Z} \) (that is, ‘coefficients modulo 2’) is called a binary polynomial. The operations proceed as for usual polynomials except that the coefficients have to be reduced modulo two. To represent a binary polynomial in a binary computer one uses words where the bits are set at the positions where the polynomial coefficients are one. We stick to the convention that the constant term goes to the least significant bit. It turns out that the arithmetic operations can be implemented quite easily in an efficient manner.

38.1 The basic arithmetical operations

Addition of binary polynomials is the XOR operation. Subtraction is the very same operation.

Multiplication of a binary polynomial by its independent variable \( x \) is simply a shift to the left.

38.1.1 Multiplication and squaring

Multiplication of two polynomials \( A \) and \( B \) is identical to the usual (binary algorithm for) multiplication, except that no carry occurs [FXT: bpol/bitpol-arith.h]:

```c
inline ulong bitpol_mult(ulong a, ulong b)
{
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 ) t ^= a;
        b >>= 1;
        a <<= 1;
    }
    return t;
}
```

As for integer multiplication with the C-type `unsigned long`, the result will silently overflow if \( \deg(A) + \deg(B) \) is equal to or greater than the word length (BITS_PER_LONG). If the operation \( t^=a \) was replaced with \( t+=a \), the ordinary (integer) product would be returned [FXT: gf2n/bitpolmult-demo.cc], see figure 38.1-A (top). When a binary polynomial \( p = \sum_{k=0}^{d} a_k x^k \) is squared, the result
equals $p^2 = \sum_{k=0}^{d} a_k x^{2k}$, figure 38.1-A (bottom). Thereby, for squaring, we just have to move the bits from position $k$ to position $2k$:

```c
inline ulong bitpol_square(ulong a)
{
    ulong t = 0, m = 1UL;
    do
    {
        if ( a&1 ) t ^= m;
        m <<= 2; a >>= 1;
    }
    while ( a );
    return t; // == bitpol_mult(a, a);
}
```

### 38.1.2 Optimization of the squaring and multiplication routines

The routines for multiplication and squaring can be optimized by partially unrolling which avoids branches. As given, the function is compiled to:

```
0: 31 c9           xor %ecx,%ecx // t = 0
2: 48 85 ff       test %rdi,%rdi // a
5: ba 01 00 00 00  mov $0x1,%edx // m = 1
a: 74 1b          je 27 <_Z13bitpol_squarem+0x27> // a==0 ?
10: 48 89 c8      mov %rcx,%rax // tmp = t
13: 48 31 d0      xor %rdx,%rax // tmp ^= m
16: 40 f6 c7 01   test $0x1,%edx // if ( a&1 )
19: 48 0f 45 c8   cmovne %rax,%rcx // then t = tmp
1e: 48 c1 e2 02   shl $0x2,%rdx // m <<= 2
22: 48 d1 ef      shr %rdi // a >>= 1
25: 75 e9         jne 10 <_Z13bitpol_squarem+0x10> // a!=0 ?
27: 48 89 c8      mov %rcx,%rax
2a: c3           retq
```

The `if()`-statement does not cause a branch so we unroll the contents of the loop 4-fold. Further, we move the `while()` statement to the end the loop to avoid the initial branch:

```c
inline ulong bitpol_square(ulong a)
{
    ulong t = 0, m = 1UL;
    do
    {
        if ( a&1 ) t ^= m;
        m <<= 2; a >>= 1;
    }
    while ( a );
    return t; // == bitpol_mult(a, a);
}
```
38.1: The basic arithmetical operations

```c
    m <<= 2; a >>= 1;
    if ( a&1 ) t ^= m;
    m <<= 2; a >>= 1;
    if ( a&1 ) t ^= m;
    m <<= 2; a >>= 1;
 }
while ( a );
return t;
```

Now we obtain much better machine code:

```assembly
0: 31 c9 xor %ecx,%ecx // t = 0
2: ba 01 00 00 00 mov $0x1,%edx // m = 1
7: 48 89 c8 mov %rcx,%rax // tmp = t
a: 48 31 d0 xor %rdx,%rax // tmp ^= m
d: 48 0f 45 c8 cmovne %rax,%rcx // then t = tmp
11: 48 0f 45 c8 cmovne %rax,%rcx // then t = tmp
15: 48 c1 e2 02 shl $0x2,%rdx // m <<= 2
19: 48 d1 ef shr %rdi // a >>= 1
1c: 48 89 c8 mov %rcx,%rax
1f: 48 31 d0 xor %rdx,%rax
22: 40 f6 c7 01 test $0x1,%dil // if ( a&1 )
26: 48 0f 45 c8 cmovne %rax,%rcx // then t = tmp
2a: 48 c1 e2 02 shl $0x2,%rdx // m <<= 2
2e: 48 d1 ef shr %rdi
31: 48 89 c8 mov %rcx,%rax
[--snip--]
43: 48 d1 ef shr %rdi
46: 48 89 c8 mov %rcx,%rax
[--snip--]
58: 48 d1 ef shr %rdi
5b: 75 aa jne 7 <_Z13bitpol_squarem+0x7> // a!=0 ?
5d: 48 89 c8 mov %rcx,%rax
60: c3 retq
```

The multiplication algorithm is optimized in the same way. For squaring one can also use the bit-zip function given in section 1.15 on page 40:

```c
inline ulong bitpol_square(ulong a) { return bit_zip0( a ); }
```

The higher half of the bits of the argument must be zero.

### 38.1.3 Exponentiation

With a multiplication (and squaring) function at hand, it is straightforward (see section 27.6 on page 565) to implement the algorithm for binary exponentiation [FXT: `bpol/bipol-arith.h`]:

```c
inline ulong bitpol_power(ulong a, ulong e)
// Return A ** e
{
    if ( 0==e ) return 1;
    ulong s = a;
    while ( 0==(e&1) )
    {
        s = bitpol_square(s);
        e >>= 1;
    }
    a = s;
    while ( 0!=(e>>=1) )
    {
        s = bitpol_square(s);
        if ( e & 1 ) a = bitpol_mult(a, s);
    }
    return a;
}
```

Note that overflow will occur even for moderate exponents.
38.1.4 Quotient and remainder

The remainder \(a\) modulo \(b\) can be computed by initializing \(A = a\) and subtracting \(B = x^j \cdot b\) with \(\deg(B) = \deg(A)\) from \(A\) at each step. The computation is finished as soon as \(\deg b > \deg A\). As C-code

[FXT: bitpol/arith.h]:

```c
inline ulong bitpol_rem(ulong a, ulong b)
// Return R = A % B = A - (A/B)*B
// Must have: B!=0
{
    const ulong db = highest_one_idx(b);
    ulong da;
    while ( db <= (da=highest_one_idx(a)) )
        { if ( 0==a ) break; // needed because highest_one_idx(0)==highest_one_idx(1)
            a -= (b<<(da-db));
        }
    return a;
}
```

The function `highest_one_idx()` is given in section 1.6 on page 16. The following version may be superior if the degree of \(a\) is small or if no fast version of the function `highest_one_idx()` is available:

```c
while ( b <= a )
{
    ulong t = b;
    while ( (a^t) > t ) t <<= 1;
    // ^= while ( highest_one(a) > highest_one(t) ) t <<= 1;
    a ^= t;
}
return a;
```

The quotient and remainder of two polynomials is computed as follows:

```c
inline void bitpol_divrem(ulong a, ulong b, ulong &q, ulong &r)
// Set R, Q so that A == Q * B + R.
// Must have B!=0.
{
    const ulong db = highest_one_idx(b);
    q = 0; // quotient
    ulong da;
    while ( db <= (da=highest_one_idx(a)) )
        { if ( 0==a ) break; // needed because highest_one_idx(0)==highest_one_idx(1)
            a -= (b<<(da-db));
            q ^= (1UL<<(da-db));
        }
    r = a;
}
```

The division routine does the same computation but discards the remainder:

```c
inline ulong bitpol_div(ulong a, ulong b)
// Return Q = A / B
// Must have B!=0.
{
    [--snip--] // identical code
    return q;
}
```

38.1.5 Greatest common divisor (GCD)

The polynomial greatest common divisor (GCD) can be computed with the Euclidean algorithm [FXT: bitpol/gcd.h]:

```c
inline ulong bitpol_gcd(ulong a, ulong b)
// Return polynomial gcd(A, B)
{
    if ( a<b ) { ulong t=a; a=b; b=t; } // swap if deg(A)<deg(B)
    // here: b!=a
    while ( 0!=b )
        {
```
38.1: The basic arithmetical operations

```c
ulong c = bitpol_rem(a, b);
a = b;
b = c;
}
return a;
```

Note that the comment
```c
if ( a < b ) { ulong t=a; a=b; b=t; }; // swap if deg(A)<deg(B)
```
is not strictly correct as the swap can also happen with \( \deg(a) = \deg(b) \) but that does no harm.

The binary GCD algorithm can be implemented as follows:
```c
inline ulong bitpol_binary_gcd(ulong a, ulong b)
{
if ( a < b ) { ulong t=a; a=b; b=t; }; // swap if deg(A)<deg(B)
if ( b==0 ) return a;
ulong k = 0;
while ( !(a|b)&1 ) // both divisible by x
{
k++;
a >>= 1;
b >>= 1;
}
while ( !(a&1) ) a >>= 1;
while ( !(b&1) ) b >>= 1;
while ( a!=b )
{
if ( a < b ) { ulong t=a; a=b; b=t; } // swap if deg(A)<deg(B)
ulong t = (a^b) >> 1;
while ( !(t&1) ) t >>= 1;
a = t;
}
return a << k;
```

With a fast bit-scan instruction we can optimize the function:
```c
inline ulong bitpol_binary_gcd(ulong a, ulong b)
{
if ( (a==0) || (b==0) ) return a|b; // one (or both) of a, b zero?
ulong ka = lowest_one_idx(a);
a >>= ka;
ulong kb = lowest_one_idx(b);
b >>= kb;
ulong k = ( ka<kb ? ka : kb );
while ( a!=b )
{
if ( a < b ) { ulong t=a; a=b; b=t; } // swap if deg(A)<deg(B)
ulong t = (a^b) >> 1;
while ( !((t&1)) ) t >>= 1;
a = t;
}
return a << k;
```

38.1.6 Exact division

Let \( C \) be a binary polynomial in \( x \) with constant term one. We use the relation (for power series)

\[
\frac{1}{C} = \frac{1}{1 - Y} = (1 + Y)(1 + Y^2)(1 + Y^4)(1 + Y^8) \ldots (1 + Y^{2^n}) \mod x^{2^{n+1}} \tag{38.1-1}
\]

where \( Y = 1 - C \). Now let \( Y = x^{e_1} + x^{e_2} + \ldots + x^{e_k} \) where \( e_i \geq 1 \) and \( e_{i+1} > e_i \). Then \( Y^q = x^{qe_1} + x^{qe_2} + \ldots + x^{qe_k} \) whenever \( q \) is a power of two, and the multiplication by \( (1 - Y^q) \) is obtained by shifts and subtractions. If \( A \) is an exact multiple of \( C \) then \( R = A/C \) is a polynomial that can be computed as follows. We assume that arrays of \( N \) bits are used for the polynomials.
Chapter 38: Binary polynomials

1. Set $R := A$ and let $e_i$ (for $i = 1, 2, \ldots, k$) be the (ordered) positions of the nonzero coefficients of $C$. Set $q := 1$.

2. If $qe_1 \geq N$ then return $R$.

3. Set $T := 0$. For $j = 1, 2, \ldots, k$, set $T := T + Rx^{qe_j}$. The multiplications with $x^{qe_j}$ are left shifts by $qe_j$ positions. Set $R := T$.

4. Set $q := 2q$ and goto step 2.

The method is most efficient when $k$, the number of nonzero coefficients of $C−1$, is small. Sometimes one can reduce the work by dividing by $CD$ and finally multiplying by $D$ for some appropriate $D$. For example, with all-ones polynomials $C = 1 + x + x^2 + \ldots + x^k$ and $D = 1 + x$, then $CD = 1 + x^{k+1}$. If $C$ is of the form $x^n (1 + \ldots + x^k)$ then $A/C$ can be computed as $(A/x^n)/(C/x^n)$.

The most simple example is $C = 1 + x$ where the above procedure reduces to the inverse reversed Gray code given in section 1.16.6 on page 47:

```c
1 inline ulong bitpol_div_xp1(ulong a)
2 // Return power series A / (x+1)
3 // If A is a multiple of x+1, then the returned value
4 // is the exact division by x+1
5 {
6   a ^= a<<1; // rev_gray ** 1
7   a ^= a<<2; // rev_gray ** 2
8   a ^= a<<4; // rev_gray ** 4
9   a ^= a<<8; // rev_gray ** 8
10  a ^= a<<16; // rev_gray ** 16
11 #if BITS_PER_LONG >= 64
12    a ^= a<<32; // for 64bit words
13 #endif
14   return a;
15 }
```

For the division by $x^2 + 1$ use

```c
1 inline ulong bitpol_div_x2p1(ulong a)
2 // Return power series A / (x^2+1)
3 // If A is a multiple of x^2+1, then the returned value
4 // is the exact division by x^2+1
5 {
6   a ^= a<<2; // rev_gray ** 2
7   a ^= a<<4; // rev_gray ** 4
8   a ^= a<<8; // rev_gray ** 8
9   a ^= a<<16; // rev_gray ** 16
10 #if BITS_PER_LONG >= 64
11    a ^= a<<32; // for 64bit words
12 #endif
13   return a;
14 }
```

An algorithm for the exact division by $C = 2^k \pm 1$ (over $\mathbb{Z}$) is given in section 1.22.2 on page 62.

### 38.2 Multiplication for polynomials of high degree

We used the straightforward multiplication scheme whose asymptotic cost is $\sim N^2$ for polynomials of degree $N$. This is fine when working with polynomials of small degree. For the multiplication of two polynomials $U$ and $V$ both of (high, even) degree $N$ write $U = U_0 + U_1 x^{N/2}$, $V = V_0 + V_1 x^{N/2}$ and use the scheme

$$UV = U_0 \cdot V_0 (1 + x^{N/2}) + (U_1 - U_0) \cdot (V_0 - V_1) x^{N/2} + U_1 \cdot V_1 (x^{N/2} + x^N)$$ (38.2-1)

recursively. Only the three multiplications indicated by a dot are expensive, the multiplications by a power of $x$ are just shifts. The resulting scheme is the Karatsuba multiplication for polynomials, relation 27.2-3 on page 553 interpreted for polynomials (set $x^{N/2} = B$). Recursive application of the scheme leads to the
38.2: Multiplication for polynomials of high degree

asymptotic cost $\sim N^{\log_2(3)} \approx N^{1.585}$. When working with polynomials of high degree the implementation of the Karatsuba scheme is a must.

We give a generalization of the Karatsuba splitting that involves no constants, and several Toom-Cook schemes.

38.2.1 Splitting schemes that do not involve constants

A generalization of the Karatsuba scheme is given in [271] (see also [272]). It does not lead asymptotically better schemes than $\sim N^{\log_2(3)}$ but has a simple structure and avoids all multiplications by constants (the asymptotically better $n$-way splitting schemes method do involve multiplications by constants for all $n \geq 3$, see section 27.2 on page 552). We give the scheme for degree-2 (3-term) polynomials, recursive application for $3^n$-term polynomials should be straightforward. Let

$$A = a_2 x^2 + a_1 x + a_0$$

$$B = b_2 x^2 + b_1 x + b_0$$

$$C = A B = c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0$$

We want to compute $c_0, c_1, \ldots, c_4$. With

$$d_{0,0} = a_0 b_0$$

$$d_{1,1} = a_1 b_1$$

$$d_{2,2} = a_2 b_2$$

$$d_{0,1} = (a_0 + a_1) (b_0 + b_1)$$

$$d_{0,2} = (a_0 + a_2) (b_0 + b_2)$$

$$d_{1,2} = (a_1 + a_2) (b_1 + b_2)$$

the $c_k$ can be obtained as

$$c_0 = d_{0,0}$$

$$c_1 = d_{0,1} - d_{0,0} - d_{1,1}$$

$$c_2 = d_{0,2} - d_{0,0} - d_{2,2} + d_{1,1}$$

$$c_3 = d_{1,2} - d_{1,1} - d_{2,2}$$

$$c_4 = d_{2,2}$$

The scheme involves 6 multiplications and 13 additions. Recursive application leads to the asymptotic cost $\sim N^{\log_2(6)} \approx 1.6309$ which is slightly worse than for the 2-term scheme. However, applying this scheme first for a polynomial with $N = 3 \cdot 2^n$ terms and then using the Karatsuba scheme recursively can be advantageous.

We generalize the method for $n$-term polynomials and denote the scheme by KA-$n$. The 2-term scheme KA-2 is the Karatsuba algorithm. With

$$A = \sum_{k=0}^{n-1} a_k x^k$$

$$B = \sum_{k=0}^{n-1} b_k x^k$$

$$C = A B =: \sum_{k=0}^{2n-2} c_k x^k$$
Chapter 38: Binary polynomials

define
\[
\begin{align*}
  d_{s,s} & := a_s b_s & \text{for } s = 0, 1, \ldots, n - 1 \\
  d_{s,t} & := (a_s + b_s)(a_t + b_t) & \text{for } s + t = i, \ t > s \geq 0, \ 1 \leq i \leq 2n - 3 \\
  c_i^* & := \sum_{s+t=i, \ 0 \leq s < t} d_{s,t} - \sum_{s+t=i, \ 0 \leq s < n-1} (d_{s,s} + d_{t,t})
\end{align*}
\] (38.2-10a)

Then
\[
\begin{align*}
  c_0 & = d_{0,0} \quad (38.2-11a) \\
  c_{2n-2} & = d_{n-1,n-1} \quad (38.2-11b)
\end{align*}
\]

and for \(0 < i < 2n - 2\):
\[
\begin{align*}
  c_i & = \begin{cases} 
    c_i^* & \text{if } i \text{ odd} \\
    c_i^* + d_{i/2,i/2} & \text{else}
  \end{cases} \quad (38.2-11c)
\end{align*}
\]

The Karatsuba scheme is obtained for \(n = 2\).

We give pari/gp code whose output is the KA-n algorithm for given \(n\). We need to create symbols ‘\(a_k\)’ (for \(a_k\)), ‘\(b_k\)’, and so on:

1. \(fa(k)=\text{eval}(\text{Str}("a" k))\)
2. \(fb(k)=\text{eval}(\text{Str}("b" k))\)
3. \(fc(k)=\text{eval}(\text{Str}("c" k))\)
4. \(fd(k,j)=\text{eval}(\text{Str}("d" k "" j))\)

For example, we can create a symbolic polynomial of degree 3:

\[
\sum(k=0,3, \ fa(k) * x^k) = a3*x^3 + a2*x^2 + a1*x + a0
\]

The next routine generates the definitions of all \(d_{s,t}\). It returns the number of multiplications involved:

1. \(D(n)=\)
2. {  
3. \hspace{1em} \text{local(mct);} \\
4. \hspace{1em} mct = 0; \ \\ \text{// count multiplications} \\
5. \hspace{1em} for (i=0, n-1, \ mct+=1; print(fd(i,i), " = ", fa(i), " * ", fb(i) ) ); \\
6. \hspace{1em} for (t=1, n-1, \\
7. \hspace{2em} for (s=0, t-1, \\
8. \hspace{3em} mct += 1; \\
9. \hspace{3em} print(fd(s,t), " = (", fa(s)+fa(t), ") * (", fb(s)+fb(t), ",(") "; \\
10. \hspace{2em} ); \\
11. \hspace{1em} ); \\
12. \hspace{1em} return(mct); \\
13. }
\]

For \(n = 3\) the output is

\[
\begin{align*}
  d_{00} & = a0 * b0 \\
  d_{11} & = a1 * b1 \\
  d_{01} & = (a0 + a1) * (b0 + b1) \\
  d_{02} & = (a0 + a2) * (b0 + b2) \\
  d_{12} & = (a1 + a2) * (b1 + b2)
\end{align*}
\]

The following routine prints \(c_i\), the coefficient of the product, in terms of several \(d_{s,t}\). It returns the number of additions involved:

1. \(C(i, n)=\)
2. {  
3. \hspace{1em} \text{local(N, s, act);} \\
4. \hspace{1em} act = -1; \ \\ \text{// count additions} \\
5. \hspace{1em} print1(fc(i), " = "); \\
6. \hspace{1em} for (s=0, i-1, \\
7. \hspace{2em} t = i - s; \\
8. \hspace{3em} if ( (t>s) && (t<n), \\
9. \hspace{4em} act += 3; \\
10. \hspace{4em} print1(" + ", fd(s,t)); \\
11. \hspace{4em} print1(" - ", fd(s,s)); \\
12. \hspace{4em} print1(" - ", fd(t,t)); \\
\}
\]

[fxtbook draft of 2008-August-17]
38.2: Multiplication for polynomials of high degree

It has to be called for all $i$ where $0 \leq i \leq 2n - 2$. The algorithm is generated by the following routine:

```plaintext
KA(n)=
    { local(mct, act);
      act = 0; \ count additions
      mct = 0; \ count multiplications
      mct = D(n); \ generate definitions for the $d_{s,t}$
      \ generate rules for computation of $c_i$ in terms of $d_{s,t}$:
      for (i=0, 2*n-2, act+=C(i,n) );
      act += n*(n-1); \ additions when setting up $d(i,j)$ for $i!=j$
      return( [mct, act] );
    }

With $n = 3$ we obtain:

$$
c_0 = +d_{00}
c_1 = +d_{01} - d_{00} - d_{11}
c_2 = +d_{02} - d_{00} - d_{22} + d_{11}
c_3 = +d_{12} - d_{11} - d_{22}c_4 = +d_{22}
$$

$$
d_{00} = a_0 \cdot b_0
d_{11} = a_1 \cdot b_1
d_{22} = a_2 \cdot b_2
d_{33} = a_3 \cdot b_3
d_{44} = a_4 \cdot b_4
d_{01} = (a_0 + a_1) \cdot (b_0 + b_1)
d_{02} = (a_0 + a_2) \cdot (b_0 + b_2)
d_{12} = (a_1 + a_2) \cdot (b_1 + b_2)
d_{03} = (a_0 + a_3) \cdot (b_0 + b_3)
d_{13} = (a_1 + a_3) \cdot (b_1 + b_3)
d_{23} = (a_2 + a_3) \cdot (b_2 + b_3)
d_{04} = (a_0 + a_4) \cdot (b_0 + b_4)
d_{14} = (a_1 + a_4) \cdot (b_1 + b_4)
d_{24} = (a_2 + a_4) \cdot (b_2 + b_4)
d_{34} = (a_3 + a_4) \cdot (b_3 + b_4)
$$

Figure 38.2-A: Code for the algorithm KA-5.

Now we generate the definitions for the KA-5 algorithm:

```plaintext
n=5 /* n terms, degree=n-1 */
default(echo, 0);
KA(n);
```

We obtain the algorithm KA-5 shown in figure 38.2-A. The format is valid pari/gp input, so we add a few lines that print code to check the algorithm:

```plaintext
  1  print("A=",sum(k=0,n-1, fa(k) * x^k))
  2  print("B=",sum(k=0,n-1, fb(k) * x^k))
  3  print("/* direct computation of the product: */")
  4  print("C=A*B")
  5  print("/* Karatsuba computation of the product: */")
  6  print("K=",sum(k=0,2*n-2, fc(k) * x^k))
  7  print("qq=K-C")
  8  print("print( if(0==qq, "OK.", " **** OUCH!") )")
```

This gives for $n = 5$:

[fxtbook draft of 2008-August-17]
Chapter 38: Binary polynomials

\[ A = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \]
\[ B = b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \]

/* direct computation of the product: */
\[ C = A \times B \]

/* Karatsuba computation of the product: */
\[ K = c_8x^8 + c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \]

\[ qq = K - C \]

\[ \text{print( if(0==qq, "OK.", " **** OUCH!"") } ) \]

We can feed the output into another pari/gp session to verify the algorithm. We use the option ‘-f’ that prevents colorization of the output which would confuse the verification process, the option ‘-q’ suppresses the output of the version:

\[ \text{gp -f -q < karatsuba-n.gp | gp} \]

We obtain (shortened and comments added):

/* definitions of d(s,t): */
\[ b_0a_0 \]
\[ b_1a_1 \]
\[ b_2a_2 \]
\[ b_3a_3 \]
\[ b_4a_4 \]

\[ (b_0 + b_1)a_0 + (a_1b_0 + b_1a_1) \]
\[ (b_0 + b_2)a_0 + (a_2b_0 + b_2a_2) \]

/* the c_i in terms of d(s,t), evaluated: */
\[ b_0a_0 \]
\[ b_1a_0 + a_1b_0 \]
\[ b_2a_0 + (a_2b_0 + b_1a_1) \]
\[ b_3a_0 + (a_3b_0 + (b_2a_1 + a_2b_2)) \]
\[ b_4a_0 + (a_4b_0 + (b_3a_1 + (a_3*b_1 + b_2*a_2))) \]
\[ b_4a_1 + (a_4b_1 + (b_3*a_2 + a_3*b_2)) \]
\[ b_4a_2 + (a_4b_2 + b_3*a_3) \]
\[ b_4a_3 + a_4*b_3 \]
\[ b_4a_4 \]

/* polynomials: */
\[ a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \]
\[ b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \]

/* direct computation of product: */
\[ b_4a_4x^8 + (b_4a_3 + a_4*b_3)x^7 + (b_4*a_2 + (a_4*b_2 + b_3*a_3))x^6 + [...] \]

/* Karatsuba computation of product: */
\[ b_4a_4x^8 + (b_4a_3 + a_4*b_3)x^7 + (b_4*a_2 + (a_4*b_2 + b_3*a_3))x^6 + [...] \]

/* difference: */
\[ 0 \]

\[ \text{OK.} /* looks good */ \]

The number of multiplications with the KA-\( n \) splitting scheme is \( (n^2 + n)/2 \) which is suboptimal except for \( n = 2 \). However, recursive application can be worthwhile. One should start with the biggest prime factors as the number of additions is then minimized. The number of multiplications does not depend on the order of recursion (see \[271\] which also tabulates the number of additions and multiplications for \( n \leq 128 \)).

With \( n \) just below a highly composite number one should add (zero-valued) ‘dummy’ terms and recursively use KA-\( n \) algorithms for small \( n \). For example, with polynomials of degree 63 recursion with KA-2 (and \( n = 64 \)) will beat the scheme “KA-7, then KA-3”.

One could write code generators that create expanded versions of the recursions for \( n \) the product of small primes. When the cost of multiplication is much higher than for addition (as for binary polynomial multiplication on general purpose CPUs) substantial savings can be expected.

### 38.2.2 Toom-Cook algorithms for binary polynomials

We give a 3-way and a 4-way splitting scheme taken from \[50\].
38.2.3 3-way splitting

![Code snippet]

Figure 38.2-B: Implementation of the 3-way multiplication scheme for binary polynomials. The five expensive multiplications are commented with ‘Mult (n)’.

For the multiplication of two polynomials $A$ and $B$ both of degree $3N$ write

$$A = a_0 + a_1 x^N + a_2 x^{2N} =: a_0 + a_1 Y + a_2 Y^2$$  \hspace{1cm} (38.2-12)

and identically for $B$. A 3-way splitting scheme for multiplication is shown in figure 38.2-B. The multiplications and divisions by $x$ are shifts and the exact divisions are linear operations if we use the method of section 38.1.6 on page 823.

38.2.4 4-way splitting

For the multiplication of two polynomials $A$ and $B$ both of degree $4N$ write

$$A = a_0 + a_1 Y + a_2 Y^2 + a_3 Y^3$$  \hspace{1cm} (38.2-13)

where $Y := x^N$, and identically for $B$. The 4-way splitting multiplication scheme is shown in figure 38.2-C.

38.2.5 Multiplication methods for binary polynomials of high degrees

For polynomials of very high degree FFT-based algorithms can be used. The most simple method is to use integer multiplication without the carry phase (which is polynomial multiplication!). We give an example using decimal digits. The carry phase of the integer multiplication is replaced by a reduction modulo 2:

```
100110111 * 110101 == 11022223331211 // integer multiplication
== 11000001111011 // parity of digits
```

The scheme will work for polynomials of degree less than nine only. When using an FFT multiplication scheme (see section 27.3 on page 560) we can multiply polynomials up to degree $N$ as long as the integer
\[ A = a_3 Y^3 + a_2 Y^2 + a_1 Y + a_0; \]
\[ B = b_3 Y^3 + b_2 Y^2 + b_1 Y + b_0; \]
\[ S_1 = a_3 + a_2 + a_1 + a_0; \]
\[ S_2 = b_3 + b_2 + b_1 + b_0; \]
\[ S_3 = S_1 \ast S_2; \quad \text{\textbackslash Mult (1)} \]
\[ S_0 = a_1 + x \ast (a_2 + x \ast a_3); \]
\[ S_6 = b_1 + x \ast (b_2 + x \ast b_3); \]
\[ S_4 = (S_0 + a_3 \ast (x+1)) \ast x + S_1; \]
\[ S_5 = (S_6 + b_3 \ast (x+1)) \ast x + S_2; \]
\[ S_0 = S_0 \ast x + a_0; \]
\[ S_6 = S_6 \ast x + b_0; \]
\[ S_5 = S_5 \ast S_4; \quad \text{\textbackslash Mult (2)} \]
\[ S_4 = S_0 \ast S_6; \quad \text{\textbackslash Mult (3)} \]
\[ S_6 = b_0 \ast x^3 + b_1 \ast x^2 + b_2 \ast x; \]
\[ S_0 = a_0 \ast x^3 + a_1 \ast x^2 + a_2 \ast x; \]
\[ S_1 = S_1 + S_2 + S_0 \ast (x^2 + x^2 + 1); \]
\[ S_5 = (S_5 + S_4 + S_6 \ast (x^4 + x^2 + 2 + 1)) \ast (x^4 + x); \]
\[ S_2 = S_2 + S_6 + S_0 \ast x^6; \]
\[ S_1 = S_1 + S_3; \quad \text{\textbackslash Mult (4)} \]
\[ S_6 = a_3 \ast b_3; \quad \text{\textbackslash Mult (5)} \]
\[ S_0 = a_0 \ast b_0; \quad \text{\textbackslash Mult (6)} \]
\[ S_1 = S_1 + S_2 + S_0 \ast (x^4 + x^2 + 1); \]
\[ S_5 = (S_5 + S_4 + S_6 \ast (x^4 + x^2 + 2 + 1)) \ast (x^4 + x); \]
\[ S_2 = S_2 + S_6 + S_0 \ast x^6; \]
\[ S_1 = S_1 + S_3; \quad \text{\textbackslash Mult (7)} \]
\[ P = S_6 \ast Y^6 + S_5 \ast Y^5 + S_4 \ast Y^4 + S_3 \ast Y^3 + S_2 \ast Y^2 + S_1 \ast Y + S_0; \]
\[ \text{Mod(1,2)} \ast (P - A \ast B) \quad \text{\textbackslash Mult (8)} \]

Figure 38.2-C: Implementation of the 4-way multiplication scheme for binary polynomials. The seven expensive multiplications are commented with ‘Mult (n)’.

values 0, 1, 2 \ldots N + 1 can be distinguished after computing the FFT. This is hardly a limitation at all: with the C-type float (24 bit mantissa) polynomials up to degree one million can be multiplied assuming at least 20 bits are correct after the FFT. With type double (53-bit mantissa) there is no practical limit. While the algorithm is very easy to implement it is not competitive to well implemented splitting schemes and the FFT method described in [238] or the multiplication algorithm given in [77]. An excellent source for multiplication algorithms for binary polynomials is [72].

38.3 Modular arithmetic with binary polynomials

Here we consider arithmetic of binary polynomials modulo a binary polynomial. Addition and subtraction are again the XOR operation, and no modular reduction is required.

38.3.1 Multiplication and squaring

Multiplication of a polynomial \( A \) by \( x \) modulo (a polynomial) \( C \) can be done by shifting left and subtracting \( C \) if the coefficient shifted out is one [FXT: bpol/bitpolmod-arith.h]:

1  static inline ulong bitpolmod_times_x(ulong a, ulong c, ulong h)
2  // Return (A \ast x) \mod C
3  // where A and C represent polynomials over \( \mathbb{Z}/2\mathbb{Z} \)
38.3: Modular arithmetic with binary polynomials

4 // W = pol(w) =: \sum_k{ [bit_k(w)] \cdot x^k}
5 // h needs to be a mask with one bit set:
6 // h == highest_one(c) >> 1 == 1UL << (degree(C)-1)
7 {
8     ulong s = a & h;
9     a <<= 1;
10    if ( s ) a ^= c;
11   return a;
12 }

In order to avoid the repeated computation of the highest set bit we introduced the auxiliary variable \( h \) that has to be initialized as described in the comment. Section 1.6 on page 10 gives algorithms for the function \texttt{highest_one()}. Note that \( h \) needs to be computed only if the degree of the modulus \( C \) changes, which is usually only once for a series of calculations. By using the variable \( h \) we can obtain the correct result even if the degree of \( C \) equals the number of bits in a word in which case \( C \) does not fit into a word.

The routine for the multiplication of two polynomials \( a \) and \( b \) modulo \( C \) can be obtained by adding a reduction step to the binary multiplication routine:

1 inline ulong bitpolmod_mult(ulong a, ulong b, ulong c, ulong h)
2 // Return \((A \cdot B) \mod C\)
3 {
4     ulong t = 0;
5     while ( b )
6     {
7         if ( b & 1 ) t ^= a;
8     b >>= 1;
9     }
10    ulong s = a & h;
11   a <<= 1;
12   if ( s ) a ^= c;
13 }
14 return t;
15 }

38.3.2 Optimization of the squaring and multiplication routines

Squaring \( a \) can be done by the multiplication \( a \cdot a \). If many squaring have to be done with a fixed modulus then a optimization using a precomputed table of the residues \( x^{2k} \mod C \) shown in section 40.1 on page 885 can be useful. Squaring of the polynomial \( \sum_{k=0}^d a_k x^k \) is the computation the sum \( \sum_{k=0}^d a_k x^{2k} \) modulo \( C \). We use the auxiliary function

1 static inline ulong bitpolmod_times_x2(ulong a, ulong c, ulong h)
2 // Return \((A \cdot x \cdot x) \mod C\)
3 {
4     { ulong s=a&h; a<<=1; if (s) a^=c; }
5     { ulong s=a&h; a<<=1; if (s) a^=c; }
6   return a;
7 }

The squaring function, with a the 4-fold unrolled loop, is

1 static inline ulong bitpolmod_square(ulong a, ulong c, ulong h)
2 // Return \(A \cdot A \mod C\)
3 {
4     ulong t = 0, s = 1;
5     do
6     {
7         if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
8         if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
9         if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
10        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
11     }
12   while ( a );
13   return t;
14 }

Whether the unrolled code is used can be specified via the line

#define MULT_UNROLL // define to unroll loops 4-fold
Chapter 38: Binary polynomials

The optimization used for the multiplication routine is also unrolling as described in section 38.1.2 on page 820:

```c
static inline ulong bitpolmod_mult(ulong a, ulong b, ulong c, ulong h)
{
    ulong t = 0;
    do
    {
        if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c;
    } while ( b );
    return t;
}
```

It turns out that squaring via multiplication is slightly faster than via the described sum computation.

38.3.3 Exponentiation

A routine [FXT: bpol/bitpolmod-arith.h] for modular exponentiation can be obtained using the right-to-left powering algorithm from section 27.6.1 on page 565:

```c
inline ulong bitpolmod_power(ulong a, ulong e, ulong c, ulong h)
// Return (A ** e) mod C
{
    if ( 0==e ) return 1; // avoid hang with e==0 in next while()
    ulong s = a;
    while ( 0==(e&1) )
    {
        s = bitpolmod_square(s, c, h);
        e >>= 1;
    }
    a = s;
    while ( 0!=(e>>=1) )
    {
        s = bitpolmod_square(s, c, h);
        if ( e & 1 ) a = bitpolmod_mult(a, s, c, h);
    }
    return a;
}
```

The left-to-right powering algorithm given in section 27.6.2 on page 566 can be implemented as:

```c
inline ulong bitpolmod_power(ulong a, ulong e, ulong c, ulong h)
{
    ulong s = a;
    ulong b = highest_one(e);
    while ( b>1 )
    {
        b >>= 1;
        s = bitpolmod_square(s, c, h); // s *= s;
        if ( e & b ) s = bitpolmod_mult(s, a, c, h); // s *= a;
    }
    return s;
}
```

Computing a power of $x$ can be optimized with this scheme:

```c
inline ulong bitpolmod_xpower(ulong e, ulong c, ulong h)
// Return (x ** e) mod C
{
    ulong s = 2; // 'x'
    ulong b = highest_one(e);
    while ( b>1 )
    {
        b >>= 1;
        s = bitpolmod_square(s, c, h); // s *= s;
        if ( e & b ) s = bitpolmod_times_x(s, c, h); // s *= x;
    }
    return s;
}
```
38.3.4 Division by \( x \)

Division by \( x \) is possible if the modulus has a nonzero constant term (that is, \( \gcd(C, x) = 1 \)). The routine is quite simple [FXT: \texttt{bpol/bitpolmod-arith.h}]:

```c
static inline ulong bitpolmod_div_x(ulong a, ulong c, ulong h)
// Return \((A / x) \mod C\)
// \( C \) must have nonzero constant term: \((c\&1)\neq1\)
{
    ulong s = a & 1;
a >>= 1;
    if ( s )
    {
        a ^= (c>>1);
        a |= h; // so it also works for \( n = \text{BITS\_PER\_LONG} \)
    }
    return a;
}
```

If we do not insist on correct results for the case that the degree of \( C \) equals the number of bits in a word, we could simply use the following two-liner:

```c
if ( a & 1 ) a ^= c;
a >>= 1;
```

The operation needs only about two CPU cycles. The inverse of \( x \) can be computed with:

```c
static inline ulong bitpolmod_inv_x(ulong c, ulong h)
// Return \((1 / x) \mod C\)
// \( C \) must have nonzero constant term: \((c\&1)\neq1\)
{
    ulong a = (c>>1);
a |= h; // so it also works for \( n = \text{BITS\_PER\_LONG} \)
    return a;
}
```

38.3.5 Extended GCD, computation of the inverse, and division

The algorithm for the computation of extended GCD (EGCD) is taken from [172] [FXT: \texttt{bpol/bitpol-gcd.h}]:

```c
inline ulong bitpol_egcd(ulong u, ulong v, ulong &iu, ulong &iv)
// Return \( u3 \) and set \( u1, v1 \) so that \( \gcd(u,v) = u3 = u*u1 + v*u2 \)
{
    ulong u1 = 1, u2 = 0;
    ulong v1 = 0, v3 = v;
    ulong u3 = u, v2 = 1;
    while ( v3!=0 )
    {
        ulong q = bitpol_div(u3, v3); // == u3 / v3;
        ulong t1 = u1 ^ bitpol_mult(v1, q); // == u1 - v1 * q;
        u1 = v1; v1 = t1;
        ulong t3 = u3 ^ bitpol_mult(v3, q); // == u3 - v3 * q;
        u3 = v3; v3 = t3;
        ulong t2 = u2 ^ bitpol_mult(v2, q); // == u2 - v2 * q;
        u2 = v2; v2 = t2;
    }
uw = u1; iv = u2;
return u3;
```

The routine can be optimized using \texttt{bitpol\_divrem()} : remove the lines

```c
ulong q = bitpol_div(u3, v3); // == u3 / v3;
[---snip---]
ulong t3 = u3 ^ bitpol_mult(v3, q); // == u3 - v3 * q;
```

[fxtbook draft of 2008-August-17]
and insert at the beginning of the body of the loop:

```
ulong q, t3;
bitpol_divrem(u3, v3, q, t3);
```

The routine computes the GCD \( g \) and two additional quantities \( i_u \) and \( i_v \) so that

\[
g = u \cdot i_u + v \cdot i_v
\]  

(38.3-1)

When \( g = 1 \) we have

\[
1 \equiv u \cdot i_u \mod v
\]  

(38.3-2)

That is, \( i_u \) is the inverse of \( u \) modulo \( v \). Thereby [FXT: bpol/bitpolmod-arith.h]:

```c
1 inline ulong bitpolmod_inverse(ulong a, ulong c)  
2     // Returns the inverse of A modulo C if it exists, else zero.  
3     // Must have deg(A) < deg(C)  
4 {
5     ulong i, t; // t unused  
6     ulong g = bitpol_egcd(a, c, i, t);  
7     if (g!=1) i = 0;  
8     return i;  
9 }
```

Modular division is obtained by multiplication with the inverse:

```c
1 inline ulong bitpolmod_divide(ulong a, ulong b, ulong c, ulong h)  
2     // Return a/b modulo c.  
3     // Must have: gcd(b,c)==1  
4 {
5     ulong i = bitpolmod_inverse(b, c);  
6     a = bitpolmod_mult(a, i, c, h);  
7     return a;  
8 }
```

When working modulo prime \( m \) the inverse of a number \( a \) can be obtained as \( a^{-1} = a^{m-2} \) \((m - 1)\) is the maximal order of an element in \( \mathbb{Z}/m\mathbb{Z} \)). With an irreducible (see section 38.4) polynomial \( C \) of degree \( n \) the inverse modulo \( C \) of a polynomial \( A \) can be obtained by computing \( A^{-1} = A^{2^n-2} \), \((2^n - 1)\) is the maximal order modulo \( C \), see section 38.5 on page 838:

```c
1 inline ulong bitpolmod_inverse_irred(ulong a, ulong c, ulong h)  
2     // Return (A ** -1) mod C  
3     // Must have: C irreducible.  
4 {
5     ulong r1 = (h<<1) - 2; // max order minus one  
6     ulong i = bitpolmod_power(a, r1, c, h);  
7     return i;  
8 }
```

### 38.4 Irreducible polynomials

A polynomial is called *irreducible* if it has no non-trivial factors (trivial factors are the constant polynomial ‘1’ and the polynomial itself). A polynomial that has a non-trivial factorization is called *reducible*. The irreducible polynomials are the ‘primes’ among the polynomials.

The factorization of a polynomial depends on its coefficient field: The polynomial \( x^2 + 1 \) over \( \mathbb{R} \) (or \( \mathbb{Z} \)) is irreducible. Over \( \mathbb{C} \) it factors as \((x^2 + 1) = (x + i)(x - i)\). As a binary polynomial, the factorization is \((x^2 + 1) = (x + 1)^2\).

All polynomials except \( x \) with zero constant coefficient are reducible because they have the factor \( x \). A binary polynomial that is irreducible has at least one nonzero coefficient of odd degree (else it would be a square). All binary polynomials except for \( x + 1 \) that have an even number of nonzero coefficients are reducible because they have the factor \( x + 1 \).
38.4: Irreducible polynomials

38.4.1 Testing for irreducibility

Irreducibility tests for binary polynomials use the fact that the polynomial $x^{2^k} - x = x^{2^k} + x$ has all irreducible polynomials of degree $k$ as a factor. For example,

$$x^{2^6} + x = x^{64} + x = (x) \cdot (x + 1) \cdot (x^3 + x + 1) \cdot (x^3 + x^2 + 1) \cdot (x^6 + x + 1) \cdot (x^6 + x^4 + x^2 + x + 1) \cdot (x^6 + x^5 + x^3 + x + 1) \cdot (x^6 + x^5 + x^4 + x + 1) \cdot (x^6 + x^5 + x^4 + x^2 + 1)$$

We see that the factors of $x^{2^d} - x$ are all polynomials whose degrees divide $d$.

38.4.1.1 The Ben-Or test for irreducibility

A binary polynomial $C$ of degree $d$ is reducible if $\gcd(x^{2^k} - x \mod C, C) \neq 1$ for any $k < d$. We compute $u_k = x^{2^k}$ (modulo $C$) for each $k < d$ by successive squarings and test whether $\gcd(u_k + x, C) = 1$ for all $k$. But as a factor of degree $f$ implies another one of degree $d - f$ it suffices to do the first $\lfloor d/2 \rfloor$ of the tests. The algorithm is called the Ben-Or irreducibility test. A C++ implementation is given in [FXT: bpol/bitpol-irred-ben-or.cc]:

```cpp
// Return whether C is irreducible (via the Ben-Or irreducibility test_;
// h needs to be a mask with one bit set:
// h == highest_one(C) >> 1 == 1UL << (degree(C)-1)
bool bitpol_irreducible_q(ulong c, ulong h)
{
    if ( c<4 )
        if ( c>=2 ) return true; // x, and 1+x are irreducible
        else return false; // constant polynomials are reducible
    if ( 0==(1&c) ) return false; // x is a factor
    if ( 0==(c & 0xaaaaaaaaUL ) ) return 0; // at least one odd degree term
    if ( 0==parity(c) ) return 0; // need odd number of nonzero coeff.
    if ( 0!=bitpol_test_squarefree(c) ) return 0; // must be square free

    ulong d = h >> 1;
    ulong u = 2; // ^= x
    while ( 0 != d ) // floor( degree/2 ) times
    {
        // Square r-times for coefficients of c in GF(2^r).
        // We have r==1:
        u = bitpolmod_square(u, c, h);
        ulong upx = u ^ 2; // ^= u+x
        ulong g = bitpol_binary_gcd(upx, c);
        if ( 1!=g ) return false; // reducible
        d >>= 2;
    }
    return true; // irreducible
}
```

Commented out at the beginning are a few tests to check for trivial necessary conditions for irreducibility.

For the test `bitpol_test_squarefree` (for a square factor) see section 38.12.2 on page 857. The routine will fail if deg $c =$BITS_PER_LONG, because the gcd-computation fails in this case.
Rabin’s test for irreducibility

A binary polynomial \( C \) of degree \( d \) is irreducible if and only if

\[
x^{2^d} \equiv x \mod C
\]

and, for all prime divisors \( p_i \) of \( d \)

\[
\gcd \left( x^{2^d/p_i} - x \mod C, C \right) = 1
\]

The implied test is called Rabin’s algorithm for irreducibility testing, see [222, p.7]. The number of GCD computations equals the number of prime divisors of \( d \).

When the prime divisors are processed in decreasing order the successive exponents are increasing and the power of \( x \) can be updated via squarings. Thereby the total number of squarings equals \( d \) which is minimal.

A C++ implementation of Rabin’s test is given in [FXT: bitpol/irpol-irred-rabin.cc]. A table of auxiliary bitmasks gives the number of squarings between the GCD computations:

```c
1 static const ulong rabin_tab[] =
2 {
3 0UL, // x = 0 (bits: ............) OPS: 40UL, // x = 1 (bits: ............) OPS: 50UL, // x = 2 (bits: ............) OPS: 60UL, // x = 3 (bits: ............) OPS: 70UL, // x = 4 (bits: ........1...) OPS: 80UL, // x = 5 (bits: ............) OPS: 90UL, // x = 6 (bits: .........11...) OPS: 10UL, // x = 7 (bits: ............) OPS: 11UL, // x = 8 (bits: .........1...) OPS: 12UL, // x = 9 (bits: .........1...) OPS: 13UL, // x = 10 (bits: .........1...) 14 [--snip--]
```

The GCD computation with the divisor one can be avoided by noting that only the polynomial \( x^2 + x = (x + 1) x \) would wrongly pass the test, so we exclude the factor \( x \) explicitly. The testing routine is

```c
1 inline bool bitpol_irreducible_rabin_q(ulong c, ulong h)
2 // Return whether C is irreducible (via Rabin's irreducibility test).
3 // h needs to be a mask with one bit set:
4 // h == highest_one(C) >> 1 == 1UL << (degree(C)-1)
5 {
6  if ( c<4 ) // C is one of 0, 1, x, 1+x
7   { if ( c>=2 ) return true; // x, and 1+x are irreducible
8     else return false; // constant polynomials are reducible
9   }
10  if ( 0==(1&c) ) return false; // x is a factor
11  ulong d = 1 + lowest_one_idx(h); // degree
12  ulong rt = rabin_tab[d]; // degree
13  ulong m = 2; // =‘x’
14  while ( rt > 1 )
15    { do
16      { --d;
17        m = bitpolmod_square(m, c, h);
18        rt >>= 1;
19      } while ( 0 == (rt & 1) );
20    ulong g = bitpol_binary_gcd( m ^ 2UL, c );
21    if ( g!=1 ) return false;
22   } do { m = bitpolmod_square(m, c, h); } while ( --d );
23  if ( m ^ 2UL ) return false;
```
Rabin’s test will be faster than the Ben-Or test if the polynomial is irreducible. When the polynomial is reducible and has small factors (as often the case with ‘random’ polynomials) then the Ben-Or test will terminate faster. A comparison of the tests is given in \[124\].

### 38.4.1.3 Testing for irreducibility without GCD computations

Call a binary polynomial $C$ of degree $d$ that has no linear factors and for which

$$x^{2^k} \equiv x \mod C$$

and, for all $l < d$,

$$x^{2^l} \equiv x \mod C$$

a **strong pseudo irreducible** (SPI). The test whether a polynomial is SPI does not involve any GCD computation. The test for a polynomial $C$ of degree $d$ can be given as

1. If $C$ has a linear factor ($x$ or $x + 1$) then return false.
2. For $k = 1, \ldots, d$ compute $s_k := x^{2^k} \mod C$ by successive squarings.
3. If $s_k = x$ for any $k < d$ then return false.
4. If $s_d \neq x$ then return false.
5. Return true.

Now if $d$ is a prime, the power of a prime, or the product of two primes then strong pseudo irreducibility implies irreducibility (see \[22\]). We list the degrees $1 < d < 63$ where strong pseudo irreducibility does not imply irreducibility (and GCDs are needed for irreducibility testing):

$$12, 18, 20, 24, 28, 30, 36, 40, 42, 44, 45, 48, 50, 52, 54, 56, 60, 63$$

The sequence is entry \[A102467\] of \[245\]. For the degrees $44 = 4 \cdot 11$ and $52 = 4 \cdot 13$ no GCDs are needed because (see \[22\]) if $d = r^e s$ where $r$ and $s$ are distinct primes and $s > (2^e - 2)/r$. We have $r^e = 4$ so we need $s > (2^4 - 2)/2 = 7$ which holds for primes $s \geq 11$.

In the implementation of the SPI test an extra branch is needed if the polynomial $C$ does not fit into a word. In that case the parity must be even [FXT: bpol/bitpol-spi.cc]:

```cpp
bool bitpol_spi_q(ulong c, ulong h)
// Return whether C is a strong pseudo irreducible (SPI).
// A polynomial C of degree d is a SPI if
// it has no linear factors, x^2k!=x for 0<k<d, and x^(2^d)==x.
// h needs to be a mask with one bit set:
// h == highest_one(C) >> 1 == 1UL << (degree(C)-1)
{
    const bool md = (bool)((h<<1)==0); // whether degree == BITS_PER_LONG
    if ( md )
        { if ( (c&1)==0 ) return false; // factor x
          if ( 0 != parity(c) ) return false; // factor x+1
        }
    else
        {
            if ( c<4 ) // C is one of 0, 1, x, 1+x
                { if ( c==2 ) return true; // x, and 1+x are irreducible
                  else return false; // constant polynomials are reducible
                }
            if ( (c&1)==0 ) return false; // factor x
        }
    return true;
}
```

[fxtbook draft of 2008-August-17]
if ( 0 == parity(c) ) return false;  // factor x+1
}

ulong t = 1;
ulong m = 2;  // x
m = bitpolmod_square(m, c, h);
do {
    if ( m==2 ) return false;
    m = bitpolmod_square(m, c, h);
    t <<= 1;
} while ( t!=h );

if ( m!=2 ) return false;
return true;

An auxiliary function returns whether GCDs are needed with the irreducibility test (64-bit version):

bool bitpol_need_gcd(ulong h)
{// Return whether GCDs are needed for irreducibility test.
  // degrees where GCDs are needed:
  // 12, 18, 20, 24, 28, 30, 36, 40, 42, 45, 48, 50, 54, 56, 60, 63
  const ulong gn =
    (1UL<<12)|(1UL<<18)|(1UL<<20)|(1UL<<24)|(1UL<<28)|(1UL<<30)|
    (1UL<<36)|(1UL<<40)|(1UL<<42)|(1UL<<45)|(1UL<<48)|(1UL<<50)|
    (1UL<<56)|(1UL<<58)|(1UL<<60)|(1UL<<63);
  return 0 != ( h & (gn>>1) );
}

Now the irreducibility test can be implemented as follows:

inline bool bitpol_irreducible_q(ulong c, ulong h)
{  
  if ( bitpol_need_gcd(h) ) return bitpol_irreducible_ben_or_q(c, h);
  else return bitpol_spi_q(c, h);
}

As the SPI test also works for polynomials not fitting into a word we can test those for irreducibility.

38.5 Primitive polynomials

Let $C$ be an irreducible polynomial. Then the sequence $p_k = x^k \mod (C)$, $k = 1, 2, \ldots$ is periodic and the (smallest) period $m$ of the sequence is the order of $x$ modulo $C$. We call $m$ the period (or order) of the polynomial $C$. For a binary polynomial of degree $n$ the maximal period equals $2^n - 1$.

For the period $m$ of $C$ we have $x^m = 1 \mod C$, so $x^m - 1 = 0 \mod C$. That is, $C$ divides $x^m - 1$ but no polynomial $x^k - 1$ with $k < m$.

A polynomial is called primitive if its period is maximal. Then the powers of $x$ generate all nonzero binary polynomials of degree $\leq n - 1$. The polynomial $x$ is a generator (‘primitive root’) modulo $C$. Primitivity implies irreducibility, the converse is not true.

The situation is somewhat parallel to the operations modulo an integer:

- Among those integers $m$ that are prime some have the primitive root $2$: the sequence $2^k$ for $k = 1, 2, \ldots, m - 1$ contains all nonzero numbers modulo $m$ (see chapter 25 on page 535).
- Among those polynomials $C$ that are irreducible some are primitive: the sequence $x^k$ for $k = 1, 2, \ldots, 2^n - 1$ contains all nonzero polynomials modulo $C$.

Note that there is another notion of the term ‘primitive’, that of a polynomial for which the greatest common divisor of all coefficients is one.
38.5: Primitive polynomials

38.5.1 Roots of primitive polynomials have maximal order

A different characterization of primitivity is as follows. Suppose you want to do computations with linear combinations \( A = \sum_{k=0}^{n-1} a_k \alpha^k \) (where \( a_k \in \text{GF}(2) \)) of the powers of an (unknown!) root \( \alpha \) of an irreducible polynomial \( C = x^n + \sum_{k=0}^{n-1} c_k x^k \).

When multiplying \( A \) with the root \( \alpha \) we obtain a term \( \alpha^n \) which we want to get rid of. But we have

\[
\alpha^n = \sum_{k=0}^{n-1} c_k \alpha^k
\]  

(38.5-1)
as \( \alpha \) is a root of the polynomial \( C \). Therefore we can use exactly the same modular reduction as with polynomial computation modulo \( C \). The same is true for the multiplication of two linear combinations (of the powers of the same root \( \alpha \)).

We see that the order of a polynomial \( p \) is the order of its root \( \alpha \) modulo \( p \), and that a polynomial is primitive if and only if its root has maximal order. An irreducible polynomial \( C \) of degree \( n \) has \( n \) distinct roots, they are equal to \( \alpha^{2^k} \mod C \) for \( 0 \leq k < n \). The orders of all roots are identical.

38.5.2 Testing for primitivity

Checking a degree-\( d \) binary polynomial for primitivity by directly using the definition costs proportional \( 2^n \) operations which is prohibitive except for tiny \( d \).

A much better solution is a modification of the algorithm to determine the order in a finite field given in section 37.6.1.2 on page 776. The implementation given here uses the pari/gp language:

```gp
1 polorder(p) =
2    /* Order of x modulo p (p irreducible over GF(2)) */
3    { local(g, g1, te, tp, tf, tx);
4        g = 'x;
5        p *= Mod(1,2);
6        te = nn_;
7        for(i=1, np_,
8            tf = vf_[i]; tp = vp_[i]; tx = vx_[i];
9                te = te / tf;
10                g1 = Mod(g, p)^te;
11                while ( 1!=g1,
12                    g1 = g1^tp;
13                    te = te * tp;
14                );
15            );
16        return( te );
17    }
```

The function uses the following global variables that must be set up before call:

```gp
1 nn_ = 0; /* max order = 2^n-1 */
2 np_ = 0; /* number of primes in factorization */
3 vp_ = []; /* vector of primes */
4 vf_ = []; /* vector of factors (prime powers) */
5 vx_ = []; /* vector of exponents */
```

As given, the algorithm will do \( n_p \) exponentiations modulo \( p \) where \( n_p \) is the number of different primes in the factorization in \( m \). A C++ implementation of the algorithm is given in [FXT: bpol/bitpol-order.cc].

A shortcut that makes the algorithm terminate as soon as the computed order drops below maximum is

```gp
1 polmaxorder_q(p) =
2    /* Whether order of x modulo p is maximal (p irreducible over GF(2)) */
3    /* Early-out variant */
4    { local(g1, te, tp, tf, tx, ct);
5        p *= Mod(1,2);
```

[fxtbook draft of 2008-August-17]
Using \texttt{polmaxorder_q()} and pari's built-in \texttt{polisirreducible()} the search for the lexicographically minimal primitive polynomials up to degree \( n = 100 \) is a matter of about 10 seconds. Extending the list up to \( n = 200 \) takes 3 minutes. The computation of all polynomials up to degree \( n = 400 \) takes less than an hour.

Again, the algorithm depends on precomputed factorizations. The table [FXT: data/merсенne-factors.txt] taken from [74] was used in order to save computation time.

For prime \( m = 2^n - 1 \) (that is, \( m \) is a Mersenne prime) irreducibility suffices for primality: The one-liner

\begin{verbatim}
x=127; for(z=1,x-1,if(polisirreducible(Mod(1,2)+t^z+t^x),print1(" ",z)))
\end{verbatim}

finds all primitive trinomials whose degrees are Mersenne primes \( m \leq 607 \) in no time:

\begin{verbatim}
89: 38, 51, 112, 120, 126
127: 48, 158, 188, 363, 365, 473, 489
607: 105, 147, 273, 334, 460, 502
\end{verbatim}

The computation (for \( d = 607 \)) takes about a minute. Note we did not exploit the symmetry (reversed polynomials are also primitive). Techniques to find primitive trinomials whose degree are very big Mersenne exponents are described in [70].

Here is a surprising theorem: Let \( p(x) = \sum_{k=0}^{d} c_k x^k \) be an irreducible binary polynomial, and \( L_p(x) := \sum_{k=0}^{d} c_k x^{2^k} \). Then all irreducible factors of \( L_p(x)/x \) (a polynomial of degree \( 2^d - 1 \)) are of degree equal to \( \text{ord}(p) \) (the order of \( x \) modulo \( p(x) \)). Especially, if \( p(x) \) is primitive, then \( L_p(x)/x \) is irreducible. The theorem is proven in [259] and also in [189, p.110]. An example: \( x^7 + x + 1 \) is primitive, so \( x^{127} + x + 1 \) is irreducible. But, as \( 2^{127} - 1 \) is prime, \( x^{127} + x + 1 \) is also primitive. Thereby \( x^{2^{127} - 1} + x + 1 \) is irreducible.

### 38.6 The number of irreducible and primitive polynomials

<table>
<thead>
<tr>
<th>( n )</th>
<th>( I_n )</th>
<th>( n )</th>
<th>( I_n )</th>
<th>( n )</th>
<th>( I_n )</th>
<th>( n )</th>
<th>( I_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
<td>186</td>
<td>21</td>
<td>99858</td>
<td>31</td>
<td>69273666</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>335</td>
<td>22</td>
<td>190557</td>
<td>32</td>
<td>134215680</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>13</td>
<td>630</td>
<td>23</td>
<td>364722</td>
<td>33</td>
<td>260300986</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>14</td>
<td>1161</td>
<td>24</td>
<td>698870</td>
<td>34</td>
<td>505286415</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>15</td>
<td>2182</td>
<td>25</td>
<td>1342176</td>
<td>35</td>
<td>981706806</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>16</td>
<td>4080</td>
<td>26</td>
<td>2580795</td>
<td>36</td>
<td>190866960</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>17</td>
<td>7710</td>
<td>27</td>
<td>4971008</td>
<td>37</td>
<td>3714566310</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>18</td>
<td>14532</td>
<td>28</td>
<td>9586395</td>
<td>38</td>
<td>7233615334</td>
</tr>
<tr>
<td>9</td>
<td>56</td>
<td>19</td>
<td>27594</td>
<td>29</td>
<td>18512790</td>
<td>39</td>
<td>14096302710</td>
</tr>
<tr>
<td>10</td>
<td>99</td>
<td>20</td>
<td>52377</td>
<td>30</td>
<td>35790267</td>
<td>40</td>
<td>2748776474</td>
</tr>
</tbody>
</table>

Figure 38.6-A: The number of irreducible binary polynomials for degrees \( n \leq 40 \).
The number of irreducible and primitive polynomials

$$I_n = \frac{1}{n} \sum_{d|n} \mu(d) 2^{\frac{n}{d}} = \frac{1}{n} \sum_{d|n} \mu(n/d) 2^d \quad (38.6-1)$$

The M"obius function $\mu$ is defined by relation 35.1-6 on page 689. The expression is identical to the formula for the number of Lyndon words (relation 17.2-2 on page 369). If $n$ is prime then $I_n = 2^n - 2$.

Figure 38.6-A for gives $I_n$, $n \leq 40$, the sequence is entry A001037 of [245]. The list of all irreducible polynomials up to degree 11 is given in [FXT: data/all-irredpoly.txt].

For large degrees $n$ the probability that a randomly chosen polynomial is irreducible is about $1/n$. With polynomials in two or more variables the situation is very different: the probability that a random polynomial is irreducible tends to one for large $n$, see [19].

The number of primitive binary polynomials of degree $n$ equals

$$P_n = \frac{\phi(2^n - 1)}{n} \quad (38.6-2)$$

If $n$ is the exponent of a Mersenne prime we have $P_n = 2^n - 2 = I_n$. The values of $P_n$ for $n \leq 40$ are shown in figure 38.6-B. The sequence is entry A001037 of [245]. The list of all primitive polynomials up to degree 11 is given in [FXT: data/all-primpoly.txt].

The difference $D_n := I_n - P_n$ is the number of irreducible non-primitive polynomials (see figure 38.6-C). Whenever $n$ is the exponent of a Mersenne prime we have $D_n = 0$. The complete list of these polynomials up to degree 12 inclusive is given in [FXT: data/all-nonprim-irredpoly.txt].

Figure 38.6-D gives the probability that a randomly chosen irreducible polynomial of degree $n$ is primitive. A polynomial of prime degree is very likely primitive, so any conjecture suggesting that polynomials of
Chapter 38: Binary polynomials

<table>
<thead>
<tr>
<th>( n )</th>
<th>( P_n/I_n )</th>
<th>( n )</th>
<th>( P_n/I_n )</th>
<th>( n )</th>
<th>( P_n/I_n )</th>
<th>( n )</th>
<th>( P_n/I_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.50000000</td>
<td>26</td>
<td>0.66642256</td>
<td>51</td>
<td>0.84834222</td>
<td>76</td>
<td>0.52983738</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>27</td>
<td>0.85401171</td>
<td>52</td>
<td>0.9998234</td>
<td>77</td>
<td>0.93832726</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
<td>28</td>
<td>0.49462097</td>
<td>53</td>
<td>0.99982834</td>
<td>78</td>
<td>0.56391518</td>
</tr>
<tr>
<td>4</td>
<td>0.66666667</td>
<td>29</td>
<td>0.65329222</td>
<td>54</td>
<td>0.53329243</td>
<td>79</td>
<td>0.99962783</td>
</tr>
<tr>
<td>5</td>
<td>1.0</td>
<td>30</td>
<td>1.0</td>
<td>55</td>
<td>0.94019198</td>
<td>80</td>
<td>0.42915344</td>
</tr>
<tr>
<td>6</td>
<td>0.66666667</td>
<td>31</td>
<td>0.49462097</td>
<td>56</td>
<td>0.91393553</td>
<td>81</td>
<td>0.84506003</td>
</tr>
<tr>
<td>7</td>
<td>1.0</td>
<td>32</td>
<td>0.50000763</td>
<td>57</td>
<td>0.65165057</td>
<td>82</td>
<td>0.99991745</td>
</tr>
<tr>
<td>8</td>
<td>0.66666667</td>
<td>33</td>
<td>0.81066253</td>
<td>58</td>
<td>0.65165057</td>
<td>83</td>
<td>0.96773455</td>
</tr>
<tr>
<td>9</td>
<td>0.66666667</td>
<td>34</td>
<td>0.66666141</td>
<td>59</td>
<td>0.99994444</td>
<td>84</td>
<td>0.38979140</td>
</tr>
<tr>
<td>10</td>
<td>0.66666667</td>
<td>35</td>
<td>1.0</td>
<td>60</td>
<td>0.35253999</td>
<td>85</td>
<td>0.99962783</td>
</tr>
<tr>
<td>11</td>
<td>0.94623656</td>
<td>36</td>
<td>0.38011770</td>
<td>61</td>
<td>1.0</td>
<td>86</td>
<td>0.66505112</td>
</tr>
<tr>
<td>12</td>
<td>0.42985075</td>
<td>37</td>
<td>0.9551569</td>
<td>62</td>
<td>0.66666667</td>
<td>87</td>
<td>0.85207814</td>
</tr>
<tr>
<td>13</td>
<td>1.0</td>
<td>38</td>
<td>0.66666285</td>
<td>63</td>
<td>0.83624531</td>
<td>88</td>
<td>0.47128978</td>
</tr>
<tr>
<td>14</td>
<td>0.65116279</td>
<td>39</td>
<td>0.84618267</td>
<td>64</td>
<td>0.49921989</td>
<td>89</td>
<td>1.0</td>
</tr>
<tr>
<td>15</td>
<td>0.82493126</td>
<td>40</td>
<td>0.4303023</td>
<td>65</td>
<td>0.96762379</td>
<td>90</td>
<td>0.99991745</td>
</tr>
<tr>
<td>16</td>
<td>0.50196078</td>
<td>41</td>
<td>0.99992518</td>
<td>66</td>
<td>0.53157031</td>
<td>91</td>
<td>0.99091593</td>
</tr>
<tr>
<td>17</td>
<td>1.0</td>
<td>42</td>
<td>0.99999999</td>
<td>67</td>
<td>0.99999999</td>
<td>92</td>
<td>0.51925414</td>
</tr>
<tr>
<td>18</td>
<td>0.53509496</td>
<td>43</td>
<td>0.99756769</td>
<td>68</td>
<td>0.52894260</td>
<td>93</td>
<td>0.85714286</td>
</tr>
<tr>
<td>19</td>
<td>1.0</td>
<td>44</td>
<td>0.50216809</td>
<td>69</td>
<td>0.83890107</td>
<td>94</td>
<td>0.6638120</td>
</tr>
<tr>
<td>20</td>
<td>0.45821639</td>
<td>45</td>
<td>0.81138931</td>
<td>70</td>
<td>0.55834947</td>
<td>95</td>
<td>0.9627339</td>
</tr>
<tr>
<td>21</td>
<td>0.8492405</td>
<td>46</td>
<td>0.6527846</td>
<td>71</td>
<td>0.9999956</td>
<td>96</td>
<td>0.38730483</td>
</tr>
<tr>
<td>22</td>
<td>0.62990076</td>
<td>47</td>
<td>0.99935309</td>
<td>72</td>
<td>0.35544000</td>
<td>97</td>
<td>0.99991264</td>
</tr>
<tr>
<td>23</td>
<td>0.97871804</td>
<td>48</td>
<td>0.38932803</td>
<td>73</td>
<td>0.99772166</td>
<td>98</td>
<td>0.6403552</td>
</tr>
<tr>
<td>24</td>
<td>0.39561006</td>
<td>49</td>
<td>0.99212598</td>
<td>74</td>
<td>0.66330362</td>
<td>99</td>
<td>0.79553432</td>
</tr>
<tr>
<td>25</td>
<td>0.96559617</td>
<td>50</td>
<td>0.58273888</td>
<td>75</td>
<td>0.82216371</td>
<td>100</td>
<td>0.45025627</td>
</tr>
</tbody>
</table>

Figure 38.6-D: Ratios \( P_n/I_n \) for degrees \( n \leq 100 \): a random irreducible binary polynomial of prime degree \( n \) is likely primitive even if \( n \) is not a Mersenne exponent.

A certain type are always primitive for prime degree is dubious: if we take one random irreducible polynomial for each prime degree \( n \), then chances are that all of them are primitive.

38.7 Transformations that preserve irreducibility

38.7.1 The reciprocal polynomial

The reciprocal of a polynomial \( F(x) \) is the polynomial

\[
F^*(x) = x^{\deg F} F(1/x)
\]

The roots of \( F^*(x) \) are the inverses of the roots of \( F(x) \). The reciprocal of a binary polynomial is the reversed binary word:

```c
inline ulong bitpol_recip(ulong c)
{
    // Return x^{\deg(C)} * C(1/x) (the reciprocal polynomial)
    ulong t = 0;
    while ( c )
    {
        t <<= 1;
        t |= (c & 1);
        c >>= 1;
    }
    return t;
}
```

[fxtbook draft of 2008-August-17]
Alternatively, one can use the bit-reversion routines given in section 1.14 on page 35. The reciprocal of an irreducible polynomial is again irreducible. The order of the polynomial is preserved under the transformation.

### 38.7.2 The polynomial \( p(x + 1) \)

When a polynomial \( p(x) \) is irreducible then \( p(x + 1) \), the composition with \( x + 1 \), is also irreducible. The composition with \( x + 1 \) does not in general preserve order: the most simple example is the primitive polynomial \( p(x) = x^4 + x^3 + 1 \) where \( p(x + 1) = x^4 + x^3 + x^2 + x + 1 \) has the order 5. The order of \( x \) modulo \( p(x) \) equals the order of \( x + 1 \) modulo \( p(x + 1) \). The composition with \( x + 1 \) can be computed by [FXT: `bpol/bitpol-irred.h`]:

```c
inline ulong bitpol_compose_xp1(ulong c)
    // Return C(x+1).
    // Self-inverse.
    {
        ulong z = 1;
        ulong r = 0;
        while ( c )
            { if ( c & 1 ) r ^= z;
              c >>= 1;
              z ^= (z<<1);
            }
        return r;
    }
```

A faster routine that finishes in time \( \log_2(b) \) (where \( b = \) bits per word) is the `blue_code()` from section 1.19 on page 51.

In general the sequence of successive ‘compose’ and ‘reverse’ operations leads to 6 different polynomials:

\[
C = [11, 10, 4, 3, 0] \\
[11, 10, 4, 3, 0] \quad -- \text{recip} \quad (C=\text{bitpol_recip}(C)) \quad --> \\
[11, 8, 7, 1, 0] \quad -- \text{compose} \quad (C=\text{bitpol_compose_xp1}(C)) \quad --> \\
[11, 10, 9, 7, 6, 5, 4, 1, 0] \quad -- \text{recip} \quad --> \\
[11, 10, 7, 6, 5, 4, 2, 1, 0] \quad -- \text{compose} \quad --> \\
[11, 9, 7, 2, 0] \quad -- \text{recip} \quad --> \\
[11, 9, 4, 2, 0] \quad -- \text{compose} \quad --> \\
[11, 10, 4, 3, 0] \quad == \text{initial value}
\]

### 38.8 Self-reciprocal polynomials

A polynomial is called self-reciprocal if it is its own reciprocal. The irreducible self-reciprocal polynomials (SRPs), except for \( 1 + x \), are of even degree \( 2d \). They can be computed from the irreducible polynomials of degree \( d \) with nonzero linear coefficient. Let \( F(x) = \sum_{j=0}^{d} f_j x^j \) and \( S_F(x) \) the corresponding SRP, then

\[
S_F(x) = x^d F(x + 1/x) = \sum_{j=0}^{d} F_j x^{d-j} (1 + x^2)^j \tag{38.8-1}
\]

The irreducible SRPs of degree 18 and their corresponding polynomials are shown in figure 38.8-A [FXT: `gf2n/bitpol-srp-demo.cc`]. The conversion can be implemented as [FXT: `bpol/bitpol-srp.h`]:

```c
inline ulong bitpol_pol2srp(ulong f, ulong d)
    // Return the self-reciprocal polynomial S=x^d*F(x+1/x) where d=deg(f).
    // W = sum(j=0, d, F(j)*x^(d-j)*(1+x^2)^j ) where
    // F(j) is the j-th coefficient of F.
    // Must have: d=degree(f)
    {
        ulong w = 1; // == (x^2+1)^j
        ulong s = 0;
        ```
Chapter 38: Binary polynomials

Figure 38.8-A: All irreducible self-reciprocal binary polynomials of degree 18 (right) and the corresponding irreducible polynomials of degree 9 with constant linear coefficient (left).

The inverse function is given in [197]:

1. Set $F := 0$, and $j := 0$.
2. If $S \mod (x^2 + 1) \equiv 0$ then set $f_j := 0$, else set $f_j := 1$.
3. Set $S := (S - f_j x^{d-j})/(x^2 + 1)$ [the division is exact].
4. Set $j := j + 1$. If $j \leq d$ goto step 2.
5. Return $F (= \sum_{j=0}^{d} f_j x^j)$.

The computation of $S \mod (x^2 + 1)$ can be omitted because the quantity is zero exactly if the central coefficient of $S$ is zero. The assignment $S := (S - f_j x^{d-j})/(x^2 + 1)$ can be replaced by $S := S/(x^2 + 1)$ (as power series) because no coefficient beyond the position $d - j$ is needed by the following steps. We use the power series division shown in section 38.1.6 on page 823 for this computation:

```
inline ulong bitpol_srp2pol(ulong s, ulong hd)
// Inverse of bitpol_pol2srp().
// Must have: hd = degree(s)/2 (note: _half_ of the degree).
// Only the lower half coefficients are accessed, i.e.
// the routine works for degree(S) <= 2*BITS_PER_LONG-2.
{
    ulong f = 0;
    long mh = 1UL << hd;
    ulong ml = 1;
    do
    { 
        ulong b = s & mh; // central coefficient
        s -= b; // set central coefficient to zero (not needed)
        if ( b ) f |= ml; // positions 0,1,...,hd
        b <<= 2; // next coefficient to low end
        f >>= 1; // next coefficient to low end
        mh = mh << 2; // next coefficient to low end
        ml = ml << 1; // next coefficient to low end
    } while ( d-- );
    return s;
}
```
The self-reciprocal polynomials of degree $2n$ are factors of the polynomial $x^{2^n+1} - 1$ (see [206]). For example, for $n = 5$ we obtain

\[
\text{lift(factormod(x^(2^5+1)-1,2))}
\]

\[
\begin{align*}
[x + 1] \\
[x^2 + x + 1] \\
[x^4 + x + 1]
\end{align*}
\]

\[
\begin{align*}
[x^8 + x^7 + x^5 + x^3 + 1] \\
[x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1]
\end{align*}
\]

The order of a self-reciprocal polynomial of degree $2n$ is a divisor of $2^n + 1$. The list of all irreducible SRP up to degree 22 is given in [FXT: data/all-irred-srp.txt].

<table>
<thead>
<tr>
<th>$n$</th>
<th>$S_n$</th>
<th>$n$</th>
<th>$S_n$</th>
<th>$n$</th>
<th>$S_n$</th>
<th>$n$</th>
<th>$S_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>93</td>
<td>21</td>
<td>49929</td>
<td>31</td>
<td>34636833</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>170</td>
<td>22</td>
<td>95325</td>
<td>32</td>
<td>67108864</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>13</td>
<td>315</td>
<td>23</td>
<td>182361</td>
<td>33</td>
<td>130150493</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>14</td>
<td>585</td>
<td>24</td>
<td>349520</td>
<td>34</td>
<td>252645135</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>15</td>
<td>1091</td>
<td>25</td>
<td>671088</td>
<td>35</td>
<td>490853403</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>16</td>
<td>2048</td>
<td>26</td>
<td>1290555</td>
<td>36</td>
<td>954437120</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>17</td>
<td>3855</td>
<td>27</td>
<td>2485504</td>
<td>37</td>
<td>1857283155</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>18</td>
<td>7280</td>
<td>28</td>
<td>4793490</td>
<td>38</td>
<td>3616814565</td>
</tr>
<tr>
<td>9</td>
<td>28</td>
<td>19</td>
<td>13797</td>
<td>29</td>
<td>9256395</td>
<td>39</td>
<td>7048151355</td>
</tr>
<tr>
<td>10</td>
<td>51</td>
<td>20</td>
<td>26214</td>
<td>30</td>
<td>17895679</td>
<td>40</td>
<td>13743895344</td>
</tr>
</tbody>
</table>

Figure 38.8-B: Number of irreducible self-reciprocal polynomials of degree $2n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$T_n$</th>
<th>$n$</th>
<th>$T_n$</th>
<th>$n$</th>
<th>$T_n$</th>
<th>$n$</th>
<th>$T_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>62</td>
<td>21</td>
<td>32508</td>
<td>31</td>
<td>23091222</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>160</td>
<td>22</td>
<td>76032</td>
<td>32</td>
<td>67004160</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>13</td>
<td>210</td>
<td>23</td>
<td>121574</td>
<td>33</td>
<td>85342752</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>14</td>
<td>448</td>
<td>24</td>
<td>344064</td>
<td>34</td>
<td>200422656</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>15</td>
<td>660</td>
<td>25</td>
<td>405000</td>
<td>35</td>
<td>289531200</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>16</td>
<td>2048</td>
<td>26</td>
<td>1005888</td>
<td>36</td>
<td>892477440</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>17</td>
<td>2570</td>
<td>27</td>
<td>1569780</td>
<td>37</td>
<td>1237491936</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>18</td>
<td>5184</td>
<td>28</td>
<td>4511520</td>
<td>38</td>
<td>2874507264</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>19</td>
<td>9198</td>
<td>29</td>
<td>6066336</td>
<td>39</td>
<td>4697046000</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>20</td>
<td>24672</td>
<td>30</td>
<td>12672000</td>
<td>40</td>
<td>13690417152</td>
</tr>
</tbody>
</table>

Figure 38.8-C: Number of primitive self-reciprocal polynomials of degree $2n$.

The number $S_n$ of irreducible SRPs of degree $2n$ is

\[
S_n = \frac{1}{2^n} \sum_{d \mid n, \text{d odd}} \mu(d) 2^{n/d} \tag{38.8-2}
\]

Values of $S_n$ for $n \leq 40$ are shown in figure 38.8-B. The sequence of values $S_n$ is entry A000048 of [215].

The number of irreducible polynomials of degree $n$ with linear coefficient one is also $S_n$.

The number $\tilde{S}_n$ of irreducible SRPs of degree $n$ is

\[
\tilde{S}_n = \frac{-1}{n} \sum_{d \mid n, \text{d even}} \mu(d) 2^{n/d} \tag{38.8-3}
\]
Chapter 38: Binary polynomials

We have $\tilde{S}_n = 0$ for odd $n$ and else $\tilde{S}_n = S_{n/2}$.

The number $T_n$ of primitive SRPs of degree $2n$ is

$$T_n = \frac{\varphi(2^n + 1)}{2^n} \quad (38.8-4)$$

The sequence of values $T_n$ is entry A069925 of [245], values for $n \leq 40$ are shown in figure 38.8-C.

38.9 Irreducible and primitive polynomials of special forms *

We give lists of irreducible and primitive polynomials of special forms. The abbreviation ‘PP’ is used for ‘primitive polynomial’ in what follows. The weight of a binary polynomial is the sum of its coefficients. Polynomials of low weight allow for cheap modular reduction.

38.9.1 All irreducible and primitive polynomials for low degrees

<table>
<thead>
<tr>
<th>2,1,0</th>
<th>7,1,0</th>
<th>8,4,3,2,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,1,0</td>
<td>7,3,0</td>
<td>8,5,3,1,0</td>
</tr>
<tr>
<td>5,1,0</td>
<td>7,4,2,1,0</td>
<td>8,6,5,2,0</td>
</tr>
<tr>
<td>4,3,0</td>
<td>7,5,3,1,0</td>
<td>8,7,6,3,1,0</td>
</tr>
<tr>
<td># non-primitive: 4,3,2,1,0</td>
<td>7,5,4,3,2,1,0</td>
<td></td>
</tr>
<tr>
<td>5,2,0</td>
<td>7,6,0</td>
<td>8,7,7,0</td>
</tr>
<tr>
<td>5,3,0</td>
<td>7,6,3,1,0</td>
<td>8,7,7,5,0</td>
</tr>
<tr>
<td>5,4,2,1,0</td>
<td>7,6,4,2,0</td>
<td></td>
</tr>
<tr>
<td>5,4,3,1,0</td>
<td>7,6,5,2,1,0</td>
<td></td>
</tr>
<tr>
<td>5,4,3,2,0</td>
<td>7,6,5,4,0</td>
<td></td>
</tr>
<tr>
<td># non-primitive: 5,4,3,2,1,0</td>
<td>7,6,5,4,2,1,0</td>
<td></td>
</tr>
<tr>
<td>6,1,0</td>
<td>7,6,5,4,3,2,0</td>
<td></td>
</tr>
<tr>
<td>6,2,1,0</td>
<td>7,6,5,4,3,2,0</td>
<td></td>
</tr>
<tr>
<td>6,5,0</td>
<td>8,4,3,1,0</td>
<td></td>
</tr>
<tr>
<td>6,5,2,1,0</td>
<td>8,5,4,3,2,1,0</td>
<td></td>
</tr>
<tr>
<td>6,5,3,2,0</td>
<td>8,6,5,4,2,1,0</td>
<td></td>
</tr>
<tr>
<td>6,5,4,1,0</td>
<td>8,7,3,1,0</td>
<td></td>
</tr>
<tr>
<td># non-primitive: 6,5,4,3,2,1,0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6,3,0</td>
<td>8,7,5,1,0</td>
<td></td>
</tr>
<tr>
<td>6,4,2,1,0</td>
<td>8,7,5,4,0</td>
<td></td>
</tr>
<tr>
<td>6,5,4,2,0</td>
<td>8,7,6,4,3,2,0</td>
<td></td>
</tr>
<tr>
<td>6,5,4,3,0</td>
<td>8,7,6,5,4,1,0</td>
<td></td>
</tr>
<tr>
<td># non-primitive: 6,5,4,3,2,1,0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 38.9-A: All binary irreducible polynomials up to degree 8.

For degrees $n \leq 8$ the complete list of irreducible polynomials is shown in figure 38.9-A. The list up to degree $n = 11$ is given in [FXT: data/all-irredpoly.txt]. The list of PPs for $n \leq 11$ is given in [FXT: data/all-primpoly.txt]. The list of all irreducible polynomials that are not primitive for $n \leq 12$ is given in [FXT: data/all-nonprim-irredpoly.txt].

38.9.2 All irreducible and primitive trinomials for low degrees

A trinomial is a polynomial with exactly three nonzero coefficients. The irreducible binary trinomials for degrees $n \leq 49$ are shown in figure 38.9-B (there are no irreducible trinomials for degrees 50 and 51). A list of all irreducible trinomials up to degree $n = 400$ is given in [FXT: data/all-trinomial-irredpoly.txt]. A more compact form of the list can is given in [FXT: data/all-trinomial-primpoly-short.txt].
**38.9: Irreducible and primitive polynomials of special forms**

<table>
<thead>
<tr>
<th>n</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
</tr>
</tbody>
</table>

*Figure 38.9-B: All irreducible trinomials $x^n + x^k + 1$ for degrees $n \leq 49$. The format of the entries is $n, k$ for primitive trinomials, and $-n, k$ for non-primitive trinomials.*

A line starts with the entry for the degree followed by all possible positions of the middle coefficient. The corresponding files giving primitive trinomials only are [FXT: data/all-trinomial-primpoly.txt] and [FXT: data/all-trinomial-primpoly-short.txt]. A list of irreducible trinomials that are *not* primitive is [FXT: data/all-trinomial-nonprimpoly.txt].

The sequence of values $n$ such that a irreducible trinomial of degree $n$ exists is entry A073571 of [245], the values for which at least one trinomial is primitive is entry A073726. The values $n, k$ for primitive polynomials of the form $(x+1)^n + (x+1)^k + 1$ are listed in [FXT: data/all-t1-primpoly.txt]. Polynomials of that form are irreducible whenever $x^n + x^k + 1$ is. The list is not the same as for primitive trinomials as the transformation $p(x) \mapsto p(x+1)$ does in general not preserve the order. The sequence of degrees $n$ such that there is a primitive polynomial $(x+1)^n + (x+1)^k + 1$ where $0 < k < n$ is entry A136416 of [245].

Regarding trinomials, there is a theorem by Swan (given in [254]): The trinomial $x^n + x^k + 1$ over GF(2) has an even number of irreducible factors (and so is reducible) if

1. $n$ is even, $k$ is odd, $n \neq 2k$, and either $nk/2 \equiv 0 \mod 4$ or $nk/2 \equiv 1 \mod 4$
2. $n$ is odd, $k$ is even and does not divide $2n$, and $n \equiv \pm 3 \mod 8$
3. $n$ is even, $k$ is odd and does divide $2n$, and $n \equiv \pm 1 \mod 8$
4. Any of the above holds for $k$ replaced by $n - k$ (that is, for the reciprocal trinomial)

The first condition implies that no irreducible trinomial for $n$ a multiple of 8 exists (as $n$ is even, $k$ must be odd, else the trinomial is a perfect square; and $nk/2 \equiv 0 \mod 4$). Further, if $n$ is a prime with $n \equiv \pm 3 \mod 8$ then the trinomial can be irreducible only if $k = 2$ (or $n - k = 2$). In the note [89] it is shown that no irreducible trinomial exists for $n$ a prime such that $n \equiv 13 \mod 24$ or $n \equiv 19 \mod 24$.

For some applications one may want to use reducible trinomials whose period is close to that of a primitive...
one. For example, the trinomial (given in [90])

\[ x^{32} + x^{15} + 1 = (x^{11} + x^9 + x^7 + x^2 + 1) \cdot (x^{21} + x^{19} + x^{15} + x^{13} + x^{12} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^4 + x^2 + 1) \]

has the period \( p = 4, 292, 868, 097 \) which is very close to \( 2^{32} - 1 = 4, 294, 967, 295 \). Note that the degree is a multiple of eight, so no irreducible trinomial of that degree exists.

### 38.9.3 Irreducible trinomials of the form \( 1 + x^k + x^d \)

\( k=1 \): The polynomials \( p = 1 + x + x^d \) are irreducible for the following \( 2 \leq d \leq 34353 \):

\( 2, 3, 4, 6, 7, 9, 15, 22, 28, 30, 46, 60, 63, 127, 153, 172, 303, 471, 532, 865, 900, 1366, 2380, 3310, 7447, 10198, 11425, 21846, 24369, 27286, 28713, 32767, 34353 \)

This is sequence [A002475] of [245]. The primitive trinomials of the form \( x^n + x + 1 \) for \( n \leq 400 \) are those with \( n \in \),

\( 2, 3, 4, 6, 7, 15, 22, 60, 63, 127, 153 \)

These numbers are the sequence [A073639] of [245], where one finds in addition \( 471, 532, 865, 900, 1366 \) with the next candidate being \( 4495 \).

\( k=2 \): \( p = 1 + x^2 + x^d \) is irreducible for the following \( 3 \leq d \leq 57341 \) (sequence [A057460]):

\( 3, 5, 11, 21, 29, 35, 93, 123, 333, 845, 4125, 10437, 10469, 14211, 20307, 34115, 47283, 50621, 57341 \)

\( k=3 \): \( p = 1 + x^3 + x^d \) is irreducible for the following \( 4 \leq d \leq 1000 \) (sequence [A057461]):

\( 4, 5, 6, 7, 10, 12, 17, 18, 20, 25, 28, 31, 41, 52, 66, 130, 151, 180, 196, 503, 650, 761, 986 \)

\( k=4 \): \( p = 1 + x^4 + x^d \) is irreducible for the following \( 5 \leq d \leq 1000 \) (sequence [A057463]):

\( 7, 9, 15, 39, 57, 81, 105 \)

\( k=5 \): \( p = 1 + x^5 + x^d \) is irreducible for the following \( 6 \leq d \leq 1000 \) (sequence [A057474]):

\( 6, 9, 12, 14, 17, 20, 23, 44, 47, 63, 84, 129, 236, 278, 279, 297, 300, 647, 726, 737, \ldots \)

### 38.9.4 Irreducible trinomials of the form \( 1 + x^d + x^{kd} \)

\( k=2 \): The polynomial \( p = 1 + x^d + x^{2d} \) is irreducible whenever \( d \) is a power of three:

\[ \begin{align*}
1: & \quad x^2 + x + 1 \\
3: & \quad x^6 + x^3 + 1 \\
5: & \quad x^{10} + x^5 + 1 \\
7: & \quad x^{14} + x^7 + 1 \\
11: & \quad x^{22} + x^{11} + 1 \\
13: & \quad x^{24} + x^{12} + 1 \\
& \cdots
\end{align*} \]

\( k=3 \): Similarly, \( p = 1 + x^d + x^{3d} \) is irreducible whenever \( d \) is a power of seven:

\[ \begin{align*}
1: & \quad x^3 + x + 1 \\
7: & \quad x^{21} + x^7 + 1 \\
11: & \quad x^{44} + x^{14} + 1 \\
13: & \quad x^{102} + x^{34} + 1 \\
& \cdots
\end{align*} \]

\( k=4 \): The polynomial \( p = 1 + x^d + x^{4d} \) whenever \( d = 3^i 5^j \), \( i, j \in \mathbb{N} \):

\[ \begin{align*}
4, & \quad 12, 20, 36, 60, 100, 108, 180, 300, 324, 500, 540, 900, 972, 1500, 1620, 2500, 2700, 2916, 4500, 4860, 7500, 8100, 8748, 12500, \ldots
\end{align*} \]

Similar regularities can be observed for other forms, see [46].
38.9.5 Primitive pentanomials

A pentanomial is a polynomial that has exactly five nonzero coefficients. PPs that are pentanomials are given in [FXT: data/pentanomial-primpoly.txt]. No primitive pentanomial exists for degrees \( n < 5 \) but for all higher degrees one seems to exist but this has not been proven so far. Entries with the special form \( x^n + x^3 + x^2 + x + 1 \) are \( n \in \{ 5, 7, 17, 25, 31, 41, 151 \} \).

38.9.6 Primitive minimum-weight and low-bit polynomials

The data in [FXT: data/minweight-primpoly.txt] lists minimal-weight PPs where in addition the coefficients are as close to the low end as possible. The first entries are shown in figure 38.9-C. A list of minimal-weight PPs that fit into a machine word is given in [FXT: bpol/primpoly-minweight.cc]. Choosing those PPs where the highest nonzero coefficient is as low as possible one obtains the list in [FXT: data/lowbit-primpoly.txt]. It starts as shown in figure 38.9-D. The corresponding extract for small degrees is given in [FXT: bpol/primpoly-lowbit.cc]. The index (position) of the second highest nonzero coefficient (the subdegree of the polynomial) grows slowly with \( n \) and is \( \leq 12 \) for all \( n \leq 400 \). Thereby one can store the list compactly as an array of 16-bit words.

38.9.7 All primitive low-bit polynomials for certain degrees

A list of all PPs \( x^n + \sum_{j=0}^{k} c_j x^j \) for degree \( n = 256 \) with the second-highest order \( k \leq 15 \) (and the first few polynomials for \( k = 16 \)) is given in [FXT: data/lowbit256-primpoly.txt]. The list starts as:

\[
\begin{align*}
256, 10, 5, 2, 0 \\
256, 10, 8, 5, 4, 1, 0 \\
256, 10, 9, 8, 7, 4, 2, 1, 0
\end{align*}
\]
Chapter 38: Binary polynomials

Equivalent tables for degrees DEG = 63, 64, 127, 128, 256, 512, 521, 607, 1000, and 1024, can be found in the files data/lowbitDEG-primpoly.txt (where DEG has to be replaced by the number).

38.9.8 Primitive low-block polynomials

A low-block polynomial has the special form \( x^n + \sum_{j=0}^{k} x^j \). Such PPs exist for 218 degrees \( n \leq 400 \). These are especially easy to store in an array (saving the index of the second highest nonzero coefficient in array element \( n \)). A complete list of all low-block PPs with degree \( n \leq 400 \) is given in [FXT: data/all-lowblock-primpoly.txt]. A short form of the list is [FXT: data/all-lowblock-primpoly-short.txt]. Among the low-block PPs are a few where just one bit (the coefficient after the leading coefficient) is not set. For \( n \leq 400 \) this is for the following degrees:

\[
3, 5, 7, 13, 15, 23, 37, 47, 85, 127, 183, 365, 383
\]

The PPs listed in [FXT: data/all-lowblock-primpoly.txt] have the smallest possible block of set bits.

38.9.9 Irreducible all-ones polynomials

Irreducible polynomials of the form \( x^n + x^{n-1} + x^{n-2} + \ldots + x + 1 \) (so-called all-ones polynomials) exist whenever \( n+1 \) is a prime number for which 2 is a primitive root. The list of such primes up to 2000 is shown in figure 39.7-B on page 875. The all-ones polynomials are irreducible for the following \( s < 400 \):

\[
1, 2, 4, 10, 12, 18, 28, 36, 52, 58, 66, 82, 100, 106, 130, 138, 148, 162, 172, 178, 180, 196, 210, 226, 268, 292, 316, 346, 348, 372, 378, 388
\]

The sequence is entry A071642 of [245].

With the exception of \( x^2 + x + 1 \), none of the all-ones polynomials is primitive. In fact, the order of \( x \) equals \( n+1 \), which is immediate when printing the powers of \( x \) (example using \( n+1 = 5 \), \( p = x^5 + x^4 + x^2 + x + 1 \)):

\[
\begin{align*}
x^0 & : 1 \\
x^1 & : 1 \\
x^2 & : 1111 \\
x^3 & : 1111 \\
x^4 & : 1111 \\
x^5 & : 1111 \\
x^6 & : 1111 \\
x^7 & : 1111 \\
\end{align*}
\]

The all-ones polynomials are a special case for the factorization of cyclotomic polynomials, see section 38.11 on page 854. Irreducible polynomials of high weight are considered in [6] where irreducible polynomials of the form \((x^{n+1} + 1)/(x+1) + x^k\) up to degree 340 are given.

38.9.10 Irreducible self-reciprocal polynomials

A list of all irreducible self-reciprocal polynomials (see section 38.8 on page 843) up to degree 22 is given in [FXT: data/all-irred-srp.txt]. These polynomials have even degree and none of them (with the exception of \( x^2 + x + 1 \)) is primitive. The number after the percent sign with each entry in figure 38.9-F equals \((2^{n/2} + 1)/r\) where \( r \) is the order of the polynomial with degree \( n \).

38.9.11 Irreducible normal polynomials

The normal irreducible polynomials are those whose roots are linearly independent (see section 40.6 on page 899). A complete list up to degree \( n = 13 \) is given in [FXT: data/all-normalpoly.txt], figure 38.9-F shows the polynomials up to degree \( n = 8 \). Normal polynomials must have subdegree \( n - 1 \), that is,
### Figure 38.9-E: Irreducible self-reciprocal polynomials up to degree 16.

<table>
<thead>
<tr>
<th>Degree</th>
<th>Polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1,0</td>
<td>14,9,7,5,0</td>
</tr>
<tr>
<td>4,3,2,1,0</td>
<td>14,12,9,8,7,6,5,2,0</td>
</tr>
<tr>
<td>6,3,0</td>
<td>14,13,12,11,10,9,7,5,4,3,2,1,0</td>
</tr>
<tr>
<td>8,5,4,3,0</td>
<td>16,15,14,13,12,9,8,7,6,5,2,1,0</td>
</tr>
<tr>
<td>10,7,5,3,0</td>
<td>16,12,11,8,6,4,3,0</td>
</tr>
<tr>
<td>12,10,7,6,5,2,0</td>
<td>16,12,11,8,6,5,3,1,0</td>
</tr>
<tr>
<td>12,10,9,8,6,4,3,2,0</td>
<td>16,12,11,8,6,5,3,2,1,0</td>
</tr>
<tr>
<td>12,11,9,7,6,5,3,1,0</td>
<td>16,12,11,8,6,5,3,2,1,0</td>
</tr>
<tr>
<td>12,8,7,6,5,4,0</td>
<td>16,12,11,8,6,5,3,2,1,0</td>
</tr>
</tbody>
</table>

### Figure 38.9-F: All normal irreducible polynomials up to degree $n = 9$. Polynomials that are not primitive are marked with a `-'.

<table>
<thead>
<tr>
<th>Degree</th>
<th>Polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1,0</td>
<td>-8,7,3,1,0</td>
</tr>
<tr>
<td>3,2,0</td>
<td>-8,7,3,2,0</td>
</tr>
<tr>
<td>4,3,0</td>
<td>-8,7,4,3,2,1,0</td>
</tr>
<tr>
<td>5,4,2,1,0</td>
<td>-8,7,5,4,3,2,0</td>
</tr>
<tr>
<td>5,4,3,2,0</td>
<td>-8,7,5,4,3,2,1,0</td>
</tr>
<tr>
<td>6,5,0</td>
<td>-8,7,6,5,4,3,2,0</td>
</tr>
<tr>
<td>6,5,2,1,0</td>
<td>-8,7,6,5,4,3,2,1,0</td>
</tr>
<tr>
<td>6,5,4,2,0</td>
<td>-8,7,6,5,4,3,2,1,0</td>
</tr>
<tr>
<td>7,6,0</td>
<td>-8,7,6,5,4,3,5,0</td>
</tr>
<tr>
<td>7,6,3,1,0</td>
<td>-8,7,6,5,4,3,5,1,0</td>
</tr>
<tr>
<td>7,6,4,1,0</td>
<td>-8,7,6,5,4,3,5,1,0</td>
</tr>
<tr>
<td>7,6,4,2,0</td>
<td>-8,7,6,5,4,3,5,1,0</td>
</tr>
<tr>
<td>7,6,5,3,2,1,0</td>
<td>-8,7,6,5,4,3,5,1,0</td>
</tr>
<tr>
<td>7,6,5,4,2,1,0</td>
<td>-8,7,6,5,4,3,5,1,0</td>
</tr>
</tbody>
</table>

[fxbook draft of 2008-August-17]
Chapter 38: Binary polynomials

they are of the form $x^n + x^{n-1} + \ldots$. The condition is necessary but not sufficient: not all irreducible polynomials of subdegree $n-1$ are normal. A list of primitive normal polynomials $x^n + x^{n-1} + \ldots + x^w + 1$ with $w$ as big as possible is given in [FXT: data/highbit-normalpoly.txt]. Primitive normal polynomials $x^n + x^{n-1} + x^w + \ldots + 1$ where $w$ is as small as possible are given in [FXT: data/lowbit-normalprimpoly.txt]. Every all-ones polynomials is normal.

38.9.12 Irreducible alternating polynomials

The ‘alternating’ polynomial $1 + \sum_{k=0}^{d} x^{2k+1} = 1 + x + x^3 + x^5 \ldots + x^{2d+1}$ can be irreducible only if $d$ is odd:

\[
\begin{align*}
1: & \quad x^3 + x + 13: x^7 + x^5 + x^3 + x + 15: x^{11} + x^9 + x^7 + x^5 + x^3 + x + 1 \\
3: & \quad x^{11} + x^9 + x^7 + x^5 + x^3 + x + 1
\end{align*}
\]

The list up to $d = 1000$ (sequence A107220 in [OEIS]) is

\[
1, 3, 5, 7, 9, 13, 23, 27, 31, 37, 63, 69, 117, 119, 173, 219, 223, 247, 307, 363, 383, 495, 695, 987,
\]

can be obtained (within about 10 minutes) via

\[
\text{for}(d=1,1000, p=(1+\text{sum}(t=0,d,x^{2*t+1})); \text{if}(\text{polisirreducible}(\text{Mod}(1,2)*p),\text{print1}(d,", ")))
\]

A computational simplification with modular reduction can be observed by noting that

\[
(1 + x + x^3 + x^5 + \ldots + x^n) \cdot (1 + x^2) = 1 + x + x^2 + x^{n+2}
\]

One does all computations modulo the product (with cheap reductions) and only reduces the final result modulo the alternating polynomial.

38.9.13 Primitive polynomials with uniformly distributed coefficients

Primitive polynomials with (roughly) equally spaced coefficients are given in [224] for degrees from 9 to 660. Polynomials with weight 5 (pentanomials) are given in [FXT: data/eq-primpoly-w5.txt], the polynomials around degree 500 are

\[
\begin{align*}
498 & \quad 372 \quad 247 \quad 124 \quad 0 \\
499 & \quad 380 \quad 253 \quad 125 \quad 0 \\
500 & \quad 378 \quad 250 \quad 127 \quad 0 \\
501 & \quad 375 \quad 255 \quad 125 \quad 0 \\
502 & \quad 370 \quad 240 \quad 121 \quad 0
\end{align*}
\]

The polynomials with weight 7 are given in [FXT: data/eq-primpoly-w7.txt], the list for weight 9 is [FXT: data/eq-primpoly-w9.txt].

38.10 Generating irreducible polynomials from Lyndon words

That the number of length-$n$ Lyndon words (see section 17.2 on page 369) is equal to the number of degree-$n$ irreducible polynomials is not a coincidence. Indeed, [80] gives an algorithm that, given one primitive polynomial, generates an irreducible polynomial from a Lyndon word: Let $b$ be a Lyndon word, $c$ an irreducible polynomial of degree $n$ and $a$ an element of maximal order modulo $c$. Set $e = a^c$ and compute the polynomial $p(x)$ over GF$(2^n)$, defined as

\[
p(x) := (x - e) (x - e^2) (x - e^4) (x - e^8) \ldots (x - e^{2^{n-1}})
\]

Then all coefficients of $p(x)$ are either zero or one and the polynomial is irreducible over GF$(2)$.

An implementation in C++ is given in [FXT: class necklace2bitpol in bpol/necklace2bitpol.h].

[fxtbook draft of 2008-August-17]


![Figure 38.10-A: Characteristic polynomials of the powers \(e = x^b\) of the primitive element \(x\) modulo \(c = x^4 + x^3 + 1\) (left). If only necklaces are used as exponents \(b\) each polynomial is obtained only once (right). Irreducible polynomials are obtained for aperiodic necklaces.

```cpp
class necklace2bitpol
{
public:
ulong p_[BITS_PER_LONG+1]; // polynomial over GF(2**n_)
ulong n_; // degree of c_
ulong c_; // modulus (irreducible polynomial)
ulong h_; // mask used for computation
ulong a_; // generator modulo c
ulong e_; // a^b

necklace2bitpol(ulong n, ulong c=0, ulong a=0) :
    n_(n), c_(c), a_(a)
{
    if ( 0==c ) c_ = lowbit_primpoly[n_];
    if ( 0==a ) a_ = 2UL; // 'x'
    h_ = (highest_one(c_) >> 1);
}
~necklace2bitpol() { ; }

ulong poly(ulong b)
{
    for (ulong k=0; k<n_; ++k) p_[k] = 0;
    p_[0] = 1;
    ulong e = bitpolmod_power(a_, b, c_, h_);
    e_ = e; // for reference
    for (ulong d=1; d<n_; ++d)
    {
        for (ulong k=d; k<n_; ++k) p_[k] = p_[k-1];
        p_[0] = 0;
        for (ulong k=0; k<d; ++k)
        {
            p_[k] ^= bitpolmod_mult( p_[k+1], e, c_, h_);
        }
        e = bitpolmod_square(e, c_, h_);
    }
    ulong p2 = 0;
    for (ulong j=0; j<n_; ++j) p2 |= (p_[j] << j);
    return p2;
}
};
```

Figure 38.10-A (left) shows all polynomials that are generated with \(c = x^4 + x^3 + 1\) and the generator \(a = x\). This is the output of [FXT: `gf2n/necklace2irred-demo.cc`]. The columns are: \(b\) and its cyclic period (an 'm' appended if the word is the cyclic minimum), \(e\) and \(p\) where a 'P' indicates that \(p\) is primitive. Observe that cyclic shifts of the same word give identical polynomials \(p\). Further, if the period is not maximal then \(p\) is reducible. Restricting our attention to the necklaces \(b\) we obtain each polynomial
just once (right of figure 38.10-A). The Lyndon words $b$ give all degree-$n$ irreducible polynomials. The primitive polynomials are exactly those where $\gcd(b, 2^n - 1) = 1$.

<table>
<thead>
<tr>
<th>degree</th>
<th>necklaces</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>49 k/sec</td>
<td>147 k/sec</td>
</tr>
<tr>
<td>35</td>
<td>17 k/sec</td>
<td>64 k/sec</td>
</tr>
<tr>
<td>45</td>
<td>8 k/sec</td>
<td>36 k/sec</td>
</tr>
<tr>
<td>63</td>
<td>3 k/sec</td>
<td>18 k/sec</td>
</tr>
</tbody>
</table>

Figure 38.10-B: Rate of generation of irreducible polynomials via necklaces and with exhaustive search.

To generate all irreducible binary polynomials of fixed degree use [FXT: class all_irredpoly in bpol/all-irredpoly.h]. The usage is shown in [FXT: gf2n/all-irredpoly-demo.cc]. For $n = 24$ one obtains $I_{24} = 698,870$ irreducible polynomials (among them are $P_{24} = 276,480$ primitive polynomials). The computation takes about 15 seconds. It turns out that the generation via exhaustive search [FXT: gf2n/bitpol-search-irred-demo.cc] is actually faster, figure 38.10-B gives the rates of generation for various degrees and both methods.

38.11 Irreducible and cyclotomic polynomials *

The primitive binary polynomials of degree $n$ can be obtained by factoring the cyclotomic polynomial (see section 35.1.1 on page 687) $Y_N$ over GF(2) where $N = 2^n - 1$. For example, with $n = 6$,

```plaintext
? n=6; N=2^n-1; lift( factormod(polcyclo(N),2) )
[x^6 + x + 1 1]
[x^6 + x^4 + x^3 + x + 1 1]
[x^6 + x^5 + 1 1]
[x^6 + x^5 + x^2 + x + 1 1]
[x^6 + x^5 + x^3 + x^2 + 1 1]
[x^6 + x^5 + x^4 + x + 1 1]
```

We use a routine (`pcfprint(N)`) that prints the $N$-th cyclotomic polynomial and its factors in symbolic form. With $n = 6, N = 2^n - 1 = 63$ we obtain

```plaintext
? n=6; N=2^n-1;
? pcfprint(N)
63: [ 36 33 27 24 18 9 3 0 ]
   [ 6 1 0 ]
   [ 6 4 3 1 0 ]
   [ 6 5 0 ]
   [ 6 5 2 1 0 ]
   [ 6 5 3 2 0 ]
   [ 6 5 4 1 0 ]
```

The irreducible but non-primitive binary polynomials are factors of cyclotomic polynomials $Y_d$ where $d | N, d < N$ and the order of 2 modulo $d$ equals $n$:

```plaintext
? fordiv(N,d,if(n==znorder(Mod(2,d)) && (d<N), pcfprint(d) ));
9: [ 6 3 0 ]
[ 6 3 0 ]
21: [ 12 11 9 8 6 4 3 1 0 ]
[ 6 4 2 1 0 ]
[ 6 5 4 2 0 ]
```

The number of factors of $Y_d$ equals $\varphi(d)/n$ so we can count how many degree-$n$ irreducible polynomials correspond to which divisor of $N = 2^n - 1$:

```plaintext
1: [1:1] 1
2: [3:1] 1
3: [7:2] 2
4: [6:1] [15:2] 3
5: [31:6] 6
7: [127:18] 18
8: [17:2] [51:4] [85:8] [255:16] 30
9: [73:8] [511:48] 56
10: [11:1] [33:2] [93:6] [341:30] [1023:60] 99
11: [23:2] [89:8] [2047:176] 186
```
Line 6 tells us that one irreducible polynomial of degree 6 is due to the factor 9, two are due to the factor 21 and the 6 primitive polynomials correspond to \( N = 63 \) itself which we have verified a moment ago. Further, the \( a \) polynomials corresponding to an entry \([d:a]\) all have order \( d \). The list was produced using

```plaintext
1 ( for (n=1, 11,
2  print1(n,"": ");
3  s = 0;
4  N = 2^n-1;
5  fordiv (N, d,
6      if ( n==znorder(Mod(2,d)) ,
7          a = eulerphi(d)/n;
8          print1(" [",d,"":",a," ] ");
9          s += a;
10      );
11  );
12 print(" ",s);
13 ); }
```

### 38.12 Factorization of binary polynomials

We give a method for the factorization of binary polynomials. The first part describes how to factorize polynomials that do not contain a square factor. The second part gives algorithms to detect and remove square factors. Finally, an algorithm to factorize arbitrary binary polynomials is given.

#### 38.12.1 Factorization of squarefree polynomials

- \( c = x^7 + x + 1 \) (irreducible):
  - \( Q= \)
  - \( Q-id= \)
  - \( \text{nullspace}= \)
  - \( 1...... 

- \( c = x^7 + x^3 + x + 1 = (x+1)(x^2 + x + 1)(x^4 + x + 1): \)
  - \( Q= \)
  - \( Q-id= \)
  - \( \text{nullspace}= \)
  - \( 1...... 

- \( c = (1+x)^7 = x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \) (not square-free):
  - \( Q= \)
  - \( Q-id= \)
  - \( \text{nullspace}= \)
  - \( 1...... 

**Figure 38.12-A:** The \( Q \)-matrices for three binary polynomials and the nullspaces of \( Q - \text{id} \).

In order to factorize a polynomial that must not contain a square factor we will use Berlekamp’s \( Q \)-matrix algorithm described in [38]. The algorithm consists of two main steps: the computation of the nullspace of a matrix, and a refinement phase that finds the distinct irreducible factors.

Let \( c \) be a binary polynomial of degree \( d \). The \( Q \)-matrix is a \( d \times d \) matrix whose \( n \)-th column can be computed as the binary polynomial \( x^{2^n} \pmod{c} \). The algorithm will use the nullspace of \( Q - \text{id} \).
The routine to compute the $Q$-matrix is \[\text{[FXT: setup_q_matrix() in } \text{bpol/berlekamp.cc}}\] :

```c
void setup_q_matrix(ulong c, ulong d, ulong *ss)
// Compute the Q-matrix for the degree-d polynomial c.
// Used in Berlekamp's factorization algorithm.
{
    ulong h = 1UL << (d-1);
    {
        ulong x2 = 2UL; // == 'x'
        ulong q = 1UL;
        x2 = bitpolmod_mult(x2, x2, c, h);
        for (ulong k=0; k<d; ++k)
            ss[k] = q;
        q = bitpolmod_mult(q, x2, c, h);
    }
    bitmat_transpose(ss, d, ss);
}
```

For the irreducible binary polynomial $c = x^7 + x + 1$ we obtain what is shown at the top of figure 38.12-A. This is the output of the program \[\text{[FXT: gf2n/qmatrix-demo.cc}}\]. The vector $n_0 = [1, 0, \ldots, 0]$ lies in the nullspace of $Q - \text{id}$ for every polynomial $c$. For $c = x^7 + x^3 + x + 1 = (x+1)(x^3 + x + 1)(x^4 + x + 1)$ we obtain a nullspace of rank three (middle of figure 38.12-A). In fact, the rank $r$ of the nullspace equals the number of distinct irreducible factors of $c$. For polynomials containing a square factor we do not get the total number of factors. For example, with $c = (1 + x)^7 = x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ we obtain what is shown at the bottom of figure 38.12-A.

The vectors spanning the nullspace must be post-processed in order to obtain the irreducible factors of $c$. The algorithm can be described as follows: let $F$ be a set of binary polynomials whose product equals $c$, the refinement step $R_i$ proceeds as follows:

Let $t$ be the $i$-th element of the nullspace. For each element $f \in F$ do the following: if the degree of $f$ equals one, keep it in the set, else delete $f$ from the set and add from the set $X = \{ \text{gcd}(f, t), \text{gcd}(f, t+1) \}$ those elements whose degrees are greater or equal to one.

One starts with $F = \{ c \}$ and does the refinement steps $R_0, R_1, \ldots, R_{r-1}$ corresponding to the vectors of the nullspace. Afterwards the set $F$ will contain exactly the distinct irreducible factors of $c$.

This is implemented in the routine \[\text{[FXT: bitpol_refine_factors() in } \text{bpol/berlekamp.cc}}\] :

```c
ulong bitpol_refine_factors(ulong *f, ulong nf, const ulong *nn, ulong r)
// Given the nullspace nn[0,...,r-1] of (Q-id)
// and nf factors f[0,...,nf-1] whose product equals c
// (typically nf=1 and f[0]==c)
// then get all r irreducible factors of c.
{
    ulong ss[r];
    for (ulong j=0; j<r; ++j) // for all elements t in nullspace
        ss[j] = nn[j];
    // skip trivial elements in nullspace:
    if ( bitpol_deg(t)==0 ) continue;
    ulong sc = 0;
    for (ulong b=0; b<nf; ++b) // for all elements bv in set
        {
            ulong bv = f[b];
            ulong db = bitpol_deg(bv);
            if ( db <= 1 ) // bv cannot be reduced
                ss[sc++] = bv;
        }
    else
        for (ulong s=0; s<2; ++s) // for all elements in GF(2)
        {
            ulong ti = t ^ s;
```

[\text{fxtbook draft of 2008-August-17}]
ulong g = bitpol_gcd(bv, ti);
if ( bitpol_deg(g) >= 1 ) ss[sc++] = g;

for (ulong k=0; k<nf; ++k) f[k] = ss[k];
if ( nf>=r ) break; // done
return nf;

As can be seen, one can skip elements corresponding to constant polynomials. Further, as soon as the set $F$ contains $r$ elements, all factors are found and the algorithm terminates.

Now Berlekamp’s algorithm can be implemented as

```cpp
ulong bitpol_factor_squarefree(ulong c, ulong *f)
// Fill irreducible factors of squarefree polynomial c into f[]
// Return number of factors.
{
    ulong d = bitpol_deg(c);
    if ( d<=1 ) // trivial cases: 0, 1, x, x+1
    {
        f[0] = c;
        if ( 0==c ) d = 1; // 0==0^1
        return d;
    }
    ulong ss[d];
    setup_q_matrix(c, d, ss);
    bitmat_add_unit(ss, d);
    ulong nn[d];
    ulong r = bitmat_nullspace(ss, d, nn);
    f[0] = c;
    ulong nf = 1;
    if ( r>1 ) nf = bitpol_refine_factors(f, nf, nn, r);
    return r;
}
```

The algorithm for the computation of the nullspace was taken from [172], find the implementation in [FXT: bmat/bitmat-nullspace.cc].

Berlekamp’s algorithm is given in [91] in a more general form: In order to factorize a polynomial with coefficients in the finite field GF($q$) set up the Q-matrix with columns $x^q i$ and in the refinement step set $X = \{ \gcd(f, t + 0), \gcd(f, t + 1), \ldots, \gcd(f, t + (q - 1)) \}$. The algorithm is efficient only if $q$ is small.

### 38.12.2 Extracting the squarefree part of a polynomial

To test whether a polynomial $c$ has a square factor one computes $g = \gcd(c, c')$ where $c'$ is the derivative. If $g \neq 1$ then $c$ has the square factor $g$: let $c = a \cdot b^2$, then $c' = a' b^2 + 2 a b b' = a' b^2$, so $\gcd(c, c') = b^2$.

The corresponding routine for binary polynomials is given in [FXT: bpol/bitpol-squarefree.h]:

```cpp
inline ulong bitpol_test_squarefree(ulong c)
// Return 0 if polynomial is squarefree
// else return square factor != 0
{
    ulong d = bitpol_deriv(c);
    if ( 0==d ) return (1==c ? 0 : c);
    ulong g = bitpol_gcd(c, d);
    return (1==g ? 0 : g);
}
```
The derivative of a binary polynomial can be computed easily [FXT: bpol/bitpol-deriv.h] (64-bit version):

```c
inline ulong bitpol_deriv(ulong c)
// Return derived polynomial
{
    c &= 0xaaaaaaaaaaaaaaaaUL;
    return (c>>1);
}
```

The coefficients at the even powers have to be deleted because derivation multiplies them with an even factor which equals zero modulo two.

If the derivative of a binary polynomial is zero, then it is a perfect square or a constant polynomial:

```c
inline ulong bitpol_pure_square_q(ulong c)
// Return whether polynomial is a pure square != 1
{
    if ( 1UL==c ) return 0;
    c &= 0xaaaaaaaaaaaaaaaaUL;
    return (0==c);
}
```

The following routine returns zero if \( c \) is squarefree or equal to one. If \( c \) is has a square factor \( s \neq 1 \) then \( s \) is returned:

```c
inline ulong bitpol_test_squarefree(ulong c)
{
    ulong d = bitpol_deriv(c);
    if ( 0==d ) return (1==c ? 0 : c);
    ulong g = bitpol_gcd(c, d);
    return (1==g ? 0 : g);
}
```

In case a polynomial is a perfect square then its square root can be computed as

```c
inline ulong bitpol_pure_sqrt(ulong c)
{
    ulong t = 0;
    for (ulong mc=1,mt=1; mc; mc<<=2,mt<<=1)
    {
        if ( mc & c ) t |= mt;
    }
    return t;
}
```

A faster way to do the computation is to use the function `bit_unzip0()` from section 1.15 on page 40.

For the factorization algorithm for general polynomials we have to extract the product of all distinct irreducible factors (the squarefree part) from a polynomial. The routine [FXT: bitpol_sreduce()] in `bpol/bitpol-squarefree.cc` returns a polynomial where the even exponents in the factorization are reduced:

```c
ulong bitpol_sreduce(ulong c)
{
    ulong s = bitpol_test_squarefree(c);
    if ( 0==s ) return c; // c is squarefree
    ulong f = bitpol_div(c, s);
    do // here s is a pure square and s>1
    {
        s = bitpol_pure_sqrt(s);
    }
    while ( bitpol_pure_square_q(s) );
    ulong g = bitpol_gcd(s, f);
    s = bitpol_div(s, g);
    f = bitpol_mult(f, s);
    return f;
}
```

With \( c = f \cdot s^k 2^t \) (\( k \) odd, the factors of \( f \) and \( s \) not necessarily distinct) the returned polynomial

[fxtbook draft of 2008-August-17]
equals \( f \cdot s^k \). Some examples:

\[
\begin{align*}
    a^2 & \mapsto a \\
    a^4 & \mapsto a \\
    a^3 = a a^2 & \mapsto a a \\
    a^5 = a a^4 & \mapsto a a \\
    a b^2 & \mapsto a b \\
    a b b^2 & \mapsto a b b = a b^2 \mapsto a b \\
    f \cdot s^k & \mapsto f \cdot s^k
\end{align*}
\]

To extract the squarefree part of a polynomial call the routine repeatedly until the returned polynomial equals the input:

```c
inline ulong bitpol_make_squarefree(ulong c)
{
    ulong z = c, t;
    while ( z!=(t=bitpol_sreduce(z)) ) z = t;
    return z;
}
```

The reduction routine will be called at most \( \log_2(n) \) times for a polynomial of degree \( n \): the worst case is a perfect power \( p = a^{2^k - 1} \) where \( 2^k - 1 \leq n \). Observe that \( (2^k - 1) = 1 + 2(2^k - 1) \), so the reduction routine will split \( p \) as

\[
    p = a s^2 \mapsto a s
\]

where \( s = a^{2^k - 1} - 1 \) is of the same form.

### 38.12.3 Factorization of arbitrary polynomials

The factorization routine for arbitrary binary polynomials [FXT: `bitpol_factor()` in `bpol/bitpol-factor.cc`] extracts the squarefree part \( f \) of its input \( c \), uses Berlekamp's algorithm to factor \( f \) and updates the exponents according to the polynomial \( s = c/f \):

```c
ulong bitpol_factor(ulong c, ulong *f, ulong *e)
{
    // Factorize the binary polynomial c:
    // c = \( \prod_{i=0}^{fct-1}{f[i]^e[i]} \)
    // The number of factors (fct) is returned.
    ulong d = bitpol_deg(c);
    if ( d<=1 ) // trivial cases: 0, 1, x, x+1
    {
        f[0] = c;
        if ( 0==c ) d = 1; // 0==0^1
        return d;
    }
    ulong cf = bitpol_make_squarefree(c);
    // ... and factor it:
    ulong fct = bitpol_factor_squarefree(cf, f);
    // All exponents are one:
    for (ulong j=0; j<fct; ++j) { e[j] = 1; }
    // Here f[],e[] is a valid factorization of the squarefree part cf
    // Update exponents with square part:
    ulong cs = bitpol_div(c, cf);
    for (ulong j=0; j<fct; ++j)
    {
        if ( 1==cs ) break;
        ulong fj = f[j];
        ulong g = bitpol_gcd(cs, fj);
        while ( 1!=g )
        {
            ++e[j];
        }
    }
}
```
cs = bitpol_div(cs, fj);
if ( i==cs ) break;
g = bitpol_gcd(cs, fj);
}
return fct;

As given the algorithm makes just one call to the routine that computes a nullspace.

Figure 38.12-B: Factorizations of the binary polynomials of degree 5.

Figure 38.12-B shows the factorizations of the binary polynomials of degree 5, it was created with the program [FXT: gf2n/bitpolfactor-demo.cc]. Factoring the first million polynomials of degrees 20, 30, 40 and 60 takes about 5, 10, 15 and 30 seconds, respectively.

A variant of the factorization algorithm often given uses the so-called squarefree factorization $c = \prod_i a_i^j$ where the polynomials $a_i$ are squarefree and pairwise coprime. Given the squarefree factorization one has to call the core routine for each non-trivial $a_i$.

As noted, the refinement step becomes expensive if the coefficients are in a field GF($q$) where $q$ is not small because $q$ computations of the polynomial gcd are involved. For an algorithm that is efficient also for large values of $q$ see [91] or [128, ch.14].
Chapter 39

Shift registers

We describe shift register sequences (SRS) and their generation via linear feedback shift registers (LFSR). The underlying mechanism is the modular arithmetic of binary polynomials described in section 38.3. We give an expression for the number of shift registers sequences of maximal length, the so-called m-sequences. Two related mechanisms, feedback carry shift registers (FCSR) and linear hybrid cellular automata (LHCA), are described. Most of the algorithms given can easily be implemented in hardware. Among the many applications for shift registers are random number generators, the computation of CRCs, spectrum spreading with communication protocols, and hardware testing.

39.1 Linear feedback shift registers (LFSR)

![Figure 39.1-A: Linear feedback shift register using the primitive polynomial $C = x^4 + x + 1$.]

Multiplication of a binary polynomial $A$ by $x$ modulo a polynomial $C$ is particularly easy as shown near the beginning of section 38.3 on page 830. Simply shift the input to the left (multiplication) and if the result $A \cdot x$ has the same degree as $C$ then subtract (XOR) the polynomial $C$ (modular reduction).

The underlying mechanism of shifting and conditionally feeding back certain bits is called a linear feedback shift register (LFSR). A shift register sequence (SRS) can be obtained by computing $A_k = x^k$, $k = 0, 1, \ldots, 2^n - 1$ modulo $C$ and setting bit $k$ of the SRS to the least significant bit of $A_k$. In the context of LFSRs the polynomial $C$ is sometimes called the connection polynomial of the shift register.
When the modulus $C$ is a primitive polynomial (see section 38.5 on page 838) of degree $n$ then the SRS is a sequence of zeros and ones that contains all nonzero words of length $n$. Further, if a word $W$ is updated at each step by left shifting and adding the bit of the SRS then this sequence also contains all nonzero words.

This is demonstrated in \[FXT: gf2n/lfsr-demo.cc\], which for $n = 4$ uses the primitive polynomial $C = x^4 + x + 1$ and gives the output shown in figure 39.1-A. Here we pasted the first few lines after the end of the actual output to emphasize the periodicity of the sequences. The corresponding SRS of period 15 is (extra spaces to mark start of period):

```
1 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 0 ...
```

In fact any of the bits of the words $A_k = x^k \mod C$ (or linear combination of two or more bits) could be used, each producing a cyclically shifted version of the SRS.

The cheapest way to generate a SRS is to compute the powers $x^{-k}$ modulo $C$, that is, to repeatedly divide by $x$:

```cpp
1  ulong c = /* a primitive polynomial */;
2  ulong n = /* degree of C */;
3  ulong a = 1;
4  for (ulong k=0; k<n; ++k)
5  {
6    ulong s = a & 1;
7    // Use s here.
8    if (s) a ^= c;
9    a >>= 1;
10  }
```

The routine will work for deg($C$) $< n$ where $n$ is the number of bits in a machine word. A version that also works for deg($C$) = $n$ is given in section 38.3 on page 830.

A C++ implementation of a LFSR is \[FXT: class lfsr in bpol/lfsr.h\]:

```cpp
1  class lfsr
2  // (binary) Linear Feedback Shift Register
3  // Produces a shift register sequence (SRS)
4  // generated by $a_k = x^k \mod C$ where $c$ is a primitive polynomial of degree $n$.
5  // The period of SRS is $2^n - 1$ (non-primitive $c$ lead to smaller periods)
6  {
7  public:
8    ulong a_; // internal state (polynomial modulo c)
9    ulong w_; // word of the shift register sequence (SRS)
10   ulong c_; // (mod 2) poly e.g. $x^4 + x + 1$ == 0x13 == 1111
11   ulong h_; // highest bit in SRS word e.g. (above) == 16 == 1...
12   ulong mask_; // mask e.g. (above) == 15 == 1111
13   ulong n_; // degree of polynomial e.g. (above) == 4
14  }
15  public:
16    lfsr(ulong n, ulong c=0)
17     // n: degree of polynomial c
18     // c: polynomial (defaults to minimum weight primitive polynomial)
19    {
20      [---snip---]
21  }
22
23  public:
24    lfsr()  // (binary) Linear Feedback Shift Register
25    {
26      ulong s = a_ & h_;
27      a_ <<= 1;
28      w_ <<= 1;
29      if ( 0!=s )
30      {
31        a_ ^= c_;
32        w_ |= 1;
33      }
34      w_ &= mask_;  
35      return w_; 
36  }
```

The crucial computation is implemented as
Up to the lines that update the word \( w \), this function is identical to \texttt{bitpolmod\_times\_x()} given in section 38.3 on page 830.

The method \texttt{next\_w()} skips to the next word by calling \texttt{next()} \( n \) times:

```cpp
ulong next\_w()
{
    for (ulong k=0; k<n_; ++k) next();
    return w_;}
```

Let \( a \) and \( w \) a pair of values that correspond to each other. The following two methods directly set one of these two while keeping the pair consistent. The following routine sets \( a \) to a given value:

```cpp
void set\_a(ulong a)
{
    a_ = a;
    w_ = 0;
    ulong b = 1;
    for (ulong j=0; j<n_; ++j)
        { 
            if ( a & 1 )
                { 
                    w_ |= b;
                    a ^= c_; 
                }
            b <<= 1;
            a >>= 1;
        }
}
```

The loop executes \( n \) times where \( n \) is the degree of the modulus. The following routine sets \( w \) to a given value:

```cpp
void set\_w(ulong w)
{
    w_ = w;
    a_ = 0;
    ulong c = c_; 
    while ( w )
        { 
            if ( w & 1 ) a_ ^= c;
            c <<= 1;
            w >>= 1;
        }
    a_ &= mask_;}
```

The supplied value must be nonzero for both methods.

Going back one step is possible via the method \texttt{prev()}:

```cpp
public:
ulong prev()
{
    prev\_a();
    set\_a(a_);
    return w_;}
```

which calls \texttt{prev\_a()}:

```cpp
private:
{
    ulong s = a_ & 1;
    a_ >>= 1;
    if ( s )
        { 
            a_ ^= (c_>>1);
            a_ |= h_; // so it works for n_ == BITS\_PER\_LONG
        }
}
The method `prev_a()` leaves the value of $w$ inconsistent with $a$ and therefore cannot be called directly. Note that stepping back is more expensive than stepping forward because `set_a()` is rather expensive. It is also possible to go backwards word-wise:

```c
ulong prev_w()
{
    for (ulong k=0; k<n_; ++k) prev_a();
    set_a(a_);
    return w_;
}
```

As this routine involves only one call to `set_a()` it is about as expensive as stepping one word forward using `next_w()`.

### 39.2 Galois and Fibonacci setup

The type of shift registers considered so far is the so-called *Galois setup* of a binary shift register. The mechanism is to detect whether a one is being shifted out and, if so, subtract the polynomial modulus. The auxiliary variable $h$ must be the word where only bit $n-1$ is set where $n$ is the degree of the polynomial $c$. The left- and right-shift operations can be implemented as

```c
ulong galois_left(ulong x, ulong c, ulong h)
{
    ulong s = x & h;
    x <<= 1;
    if ( 0!=s ) x ^= c;
    return x;
}
```

and

```c
ulong galois_right(ulong x, ulong c, ulong)
{
    ulong s = ( x & 1UL );
    x >>= 1;
    if ( s ) x ^= (c>>1);
    return x;
}
```

c = .11..1 == 0x19 == 25 (deg = 4)
r = .1..11 == 0x13 == 19 (deg = 4)

<table>
<thead>
<tr>
<th>k</th>
<th>Lc</th>
<th>Lr</th>
<th>Rc</th>
<th>Rr</th>
<th>Lc</th>
<th>Lr</th>
<th>Rc</th>
<th>Rr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k</th>
<th>Lc</th>
<th>Lr</th>
<th>Rc</th>
<th>Rr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 39.2-A: Sequences of words generated with the Galois- and Fibonacci- mechanisms, either with the left- or the right-shift (capital letters ‘L’ and ‘R’ on top of columns) and primitive polynomial ‘c’ or its reciprocal ‘r’. Each track of any sequence is a shift register sequence.

Four sequences of binary words that (starting identically and) are generated with either the left- or right-shift and a primitive polynomial or its reciprocal (reversed word) are shown in figure 39.2-A. A different set of sequences shown in the same figure is obtained via the so-called *Fibonacci setup*. In the Fibonacci setup the sum (modulo 2) of bits determined by the used polynomial is shifted in at each step.
The left- and right- shift operations can be implemented as

```cpp
1 ulong fibonacci_left(ulong x, ulong c, ulong h)
2 {
3     x <<= 1;
4     ulong s = parity( x & c );
5     if ( 0!=s ) x ^= 1;
6     x &=(h<<1); // remove excess bit at high end
7     return x;
8 }
```

and

```cpp
1 ulong fibonacci_right(ulong x, ulong c, ulong h)
2 {
3     ulong s = parity( x & c );
4     x >>= 1;
5     if ( s ) x ^= h;
6     return x;
7 }
```

As the parity computation is expensive on most machines (see section [1.16.1 on page 44](#)) the Galois setup should usually be preferred. The programs [FXT: gf2n/lfsr-fibonacci-demo.cc](#) and [FXT: gf2n/lfsr-galois-demo.cc](#) can be used to create the binary patterns shown in figure 39.2-A. With both programs the polynomial modulus can be specified.

### 39.3 Error detection by hashing: the CRC

A hash value is an element from a set $H$ that is computed via a hash function $f$ that maps any (finite) sequence of input data to $H$.

For the sake of simplicity we now consider the case that the input sequences are of fixed size so they are in a fixed set, say $S$. We further assume that the set $S$ is (much) bigger than $H$.

$$f : S \rightarrow H, \quad s \mapsto h$$

(39.3-1)

where $s \in S$ and $h \in H$.

Two input sequences with different hash values are necessarily different. But, as the hash function maps a larger set to a smaller one, there are different input sequences with identical hash values.

A trivial example is the set $H = \{0, 1\}$ together with a function that count binary digits modulo two, the parity function. Another example is the sum-of-digits test (see section [27.5 on page 564](#)) used to check the multiplication of large numbers. In the test we compute the value of a multi digit number decimal $s \in S$ modulo nine. The crucial additional property of this hash is that with $f(A) = a$, $f(B) = b$, $f(C) = c$ (were $A, B, C$ are decimal numbers), then $A \cdot B = C$ implies $a \cdot b = c$.

A hash function $f$ that is actually useful should have the mixing property: it should map the elements $s \in S$ ‘randomly’ to $H$. With the sum-of-digits test we could have used rather arbitrary moduli for the hash function. With one exception: the value modulo ten as hash would be pretty useless. No change in any digit except for the last could ever be detected.

The so-called cyclic redundancy check (CRC) is a hash where the hash values are binary words of fixed length. The hash function (basically) computes $h = s \mod c$ where $s$ is a binary polynomial build from the input sequence and $c$ is a binary polynomial that is primitive (see chapter [38](#) on page [819](#)). We will use polynomials $c$ of degree 64 so the hash values (CRCs) are 64-bit words.

An C++ implementation is given as [FXT: class crc64 in bits/crc64.h](#):

```cpp
1 class crc64
2 // 64-bit CRC (cyclic redundancy check)
3 {
4 public:
5     uint64 a_; // internal state (polynomial modulo c)
```
Chapter 39: Shift registers

```c
uint64 c_; // a binary primitive polynomial
// (non-primitive c lead to smaller periods)
// The leading coefficient needs not be present.
uint64 h_; // auxiliary

static const uint64 cc[]; // 16 "random" 64-bit primitive polynomials

public:

crc64(uint64 c=0)
{
  if ( 0==c ) c = 0x1bULL; // ^= 64,4,3,1,0 (default)
  init(c);
}
~crc64() {}

void init(uint64 c)
{
  c_ = c;
  c_ >>= 1;
  h_ = 1ULL<<63;
  c_ |= h_; // leading coefficient
  reset();
}

void reset() { set_a(~0ULL); } // all ones
void set_a(uint64 a) { a_=a; }
uint64 get_a() const { return a_; }
```

Note that a nonzero initial state (member variable `a`) is used: starting with zero will only go to a nonzero state with the first nonzero bit in the input sequence. That is, input sequences differing only by initial runs of zeros would get the same CRC.

After an instance is created one can feed in bits via `bit_in(b)` where lowest bit of `b` must contain the bit to be used:

```c
void shift()
{
  bool s = (a_ & 1);
  a_ >>= 1;
  if ( 0!=s ) a_ ^= c_;
}
uint64 bit_in(unsigned char b)
{
  a_ ^= (b&1);
  shift();
  return a_;
}
```

When a byte is to be checksummed we can do better than just feeding in the bits one by one. This is achieved adding the byte followed by eight calls to `shift()`:

```c
uint64 byte_in(unsigned char b)
{
  #if 1
    a_ ^= b;
    shift(); shift(); shift(); shift();
    shift(); shift(); shift(); shift();
  #else // identical but slower:
    bit_in(b); b>>=1; // bit 0
    bit_in(b); b>>=1; // bit 1
    bit_in(b); b>>=1; // bit 2
    bit_in(b); b>>=1; // bit 3
    bit_in(b); b>>=1; // bit 4
    bit_in(b); b>>=1; // bit 5
    bit_in(b); b>>=1; // bit 6
    bit_in(b); b>>=1; // bit 7
  #endif
  return a_;
}
```

[fxtbook draft of 2008-August-17]
The lower block implements the straightforward idea. The program [FXT: bits/crc64-demo.cc] computes the 64-bit CRC of a single byte in both ways.

Binary words are fed in byte by byte, starting from the lower end:

```c
uint64 word_in(uint64 w)
{
    ulong k = BYTES_PER_LONG_LONG;
    while ( k-- ) { byte_in( (uchar)w ); w>>=8; }
    return a_;
}
```

To feed in a given number of bits of a word, use the following method:

```c
uint64 bits_in(uint64 w, uchar k)
// Feed in the k lowest bits of w
{
    if ( k&1 ) { a_ ^= (w&1); w >>= 1; shift(); }
    k >>= 1;
    if ( k&1 ) { a_ ^= (w&3); w >>= 2; shift(); shift(); }
    k >>= 1;
    if ( k&1 ) { a_ ^= (w&15); w >>= 4; shift(); shift(); shift(); shift(); }
    k >>= 1;
    while ( k-- ) { byte_in( (uchar)w ); w>>=8; }
    return a_;
}
```

The operation is the optimized equivalent to

```c
while ( k-- ) { bit_in( (uchar)w ); w>>=1; }
```

If two sequences differ in a single block of up to 64 bits, their CRCs will be different. The probability that two different sequences have the same CRC equals \(2^{-64} \approx 5.42 \cdot 10^{-20}\). If that is not enough (and one does not want to write a CRC with more than 64 bits) then one can simply use two (or more) instances where different polynomials must be used. Sixteen ‘random’ primitive polynomials are given [FXT: bits/crc64.cc] as static class member:

```c
const uint64 crc64::cc[] = {
0x5a0127dd34af1e81ULL, // [0]
0x4ef12e145d0e3ccdULL, // [1]
0x16503f45acce9345ULL, // [2]
0x24e8034491298b3fULL, // [3]
0x9e4a8ad2261db8b1ULL, // [4]
0xb199aecfbb17a13fULL, // [5]
0x3f1fa2cc0dfbbf51ULL, // [6]
0xfb6e45b2f694fb1fULL, // [7]
0xd4597140a01d32edULL, // [8]
0xbd08ba1a2d621bf0ULL, // [9]
0xeae2b680542730db1ULL, // [10]
0x8ec06ec4a8fe8f6dULL, // [11]
0xb89a2ecea2233001ULL, // [12]
0xb899e790b615ad1ULL, // [13]
0x7eae8397265e1f90ULL, // [14]
0xf368ae22deec7c3ull, // [15]
};
```

These are taken from the list [FXT: data/rand64-hex-primpoly.txt]. Initialize multiple CRCs as follows:

```c
crc64 crca( crc64::cc[0] );
crc64 crcb( crc64::cc[1] );
```

A class for 32-bit CRCs is given in [FXT: class crc32 in bits/crc32.h]. Its usage is completely equivalent.

The CRC can easily be implemented in hardware and is, for example, used to detect errors in hard disk blocks. When a block is written its CRC is computed and stored in an extra word. When the block is read, the CRC is computed from the data and compared to the stored CRC. A mismatch indicates an error.

One property that the CRC does not have is cryptographic security. It is possible to intentionally create a data set with a prescribed CRC. With secure hashes (like MD5 and SHA) it is (practically) not possible to do so. Secure hashes can be used to ‘sign’ data. Imagine you distribute a file (for example an binary
executable) over the Internet. You want to make sure that someone downloading the file (from any source) can verify that it is not an altered version (like, in the case of an executable, a malicious program). To do so you create a (secure!) hash value which you publish on your web site. Any person can verify the authenticity of the file by computing the hash and comparing it to the published version.

The cryptographic security of hash functions like MD5 and SHA is the object of ongoing research, see [268], [42], and [43].

### 39.3.1 Optimization via lookup tables

One can feed in an \( n \)-bit word \( w \) into the CRC in one step (instead of \( n \) steps) as follows: add \( w \) to (the CRC word) \( a \). Save the lowest \( n \) bits of the result to a variable \( x \). Right shift \( a \) by \( n \) bits. Add to \( a \) the entry \( x \) of an auxiliary table \( t \). For \( n = 8 \) the operation can be implemented as [FXT: class tcrc64 in bits/tcrc64.h]:

```cpp
1 uint64 byte_in(uchar b)
2 {
3     a_ ^= b;
4     uint64 x = t_[a_ & 255];
5     a_ >>= 8;
6     a_ ^= x;
7     return a_;
8 }
```

The size of the table \( t \) is \( 2^n = 256 \) words. For \( n = 1 \) the table would have only two entries, zero and \( c \), the polynomial used. Then the implementation reduces to

```cpp
1 uint64 bit_in(uchar b)
2 {
3     a_ ^= (b&1);
4     bool s = (a_ & 1);
5     a_ >>= 1;
6     if ( 0!=s ) a_ ^= c_; // t[0]=0; t[1]=c_
7     return a_;
8 }
```

which is equivalent to the \textit{bit_in()} routine of the unoptimized CRC.

The lookup table is computed upon class initialization as follows:

```cpp
1 for (ulong w=0; w<256; ++w)
2 {
3     set_a(0);
4     for (ulong k = 0; k<8; ++k) bit_in( (uchar)w>>k );
5     t_[w] = a_;  // CRC data
6 }
```

The class can use tables of either 16 or 256 words. When a table of size 16 is used, the computation is about 6 times faster than with the non-optimized routine. A table of size 256 gives a speedup by a factor of 12. Optimization techniques based on lookup tables are often used in practical applications, both in hardware and in software, see [63].

### 39.3.2 Parallel CRCs

A very fast method for checksumming is to compute the CRCs for each bit of the fed-in words in parallel. An array of 64 words is used [FXT: class pcrc64 in bits/pcrc64.h]:

```cpp
1 template <typename Type>
2 class pcrc64
3 // Parallel computation of 64-bit CRCs for each bit of the input words.
4 // Primitive polynomial used is \( x^{64} + x^4 + x^3 + x^2 + 1 \)
5 {
6     public:
7         Type x_[64]; // CRC data
8         // bit(i) of x_[0], x_[1], ..., x_[63] is a 64-bit CRC
9         // of bit(i) of all input words
10         uint pos_; // position of constant polynomial term
```
Upon initialization all words are set to all ones:

```cpp
public:
    pcrc64()
        : m_(63)
    {
        reset();
    }
    ~pcrc64() {}

    void reset()
    {
        pos_ = 0;
        Type ff = 0; ff = ~ff;
        for (uint k=0; k<64; ++k) x_[k] = ff;
    }
```

The cyclic shift of the array is avoided by working modulo 64 when feeding in words:

```cpp
void word_in(Type w)
{
    uint p = pos_;
    uint h = (p+1) & m_;
    Type a = x_[p & m_]; // 0
    p += 2;
    a ^= x_[p & m_]; // 2
    ++p;
    a ^= x_[p & m_]; // 3
    ++p;
    a ^= x_[p & m_]; // 4
    x_[h] = a ^ w;
}
```

The algorithm corresponds to the Fibonacci setup of the linear feedback shift registers (see section 39.2 on page 864). There is no primitive trinomial with degree a multiple of eight so we use the pentanomial

\[ x^{64} + x^4 + x^3 + x^2 + 1 \]

With an array size where a primitive trinomial exists the modulo computations would be more expensive. A unrolled routine can be used to feed in multiple words:

```cpp
void words_in(Type *w, ulong n)
{
    if ( n&1 ) { word_in(w[0]); ++w; }
    n >>= 1;
    if ( n&1 ) { word_in(w[0]); word_in(w[1]); w+=2; }
    n >>= 1;
    for (ulong k=0; k<n; ++k)
    {
        word_in(w[0]);
        word_in(w[1]);
        word_in(w[2]);
        word_in(w[3]);
        w += 4;
    }
}
```

The program [FXT: bits/pcrc64-demo.cc] feeds the numbers up to a given value into a pcrc64<

\[ \text{int main()} \]

\[ \{ \text{Type n = 32768;} \]

\[ \text{pcrc64<Type> P;} \]

\[ \text{for (Type k=0; k<n; ++k) P.word_in(k);} \]

\[ \text{// print array P.x[]} \]

This rather untypical type of input data illustrates the independence of the bits in the array x_[]:

```
0: ........................1.1.1.1.1.1.
48: ........................1.1.1.1.1.1.
```
The implementation can process about 2 GB of data per second when 64-bit types are used, 1 GB/s with 32-bit types, 500 MB/sec with 16-bit types, and about 230 MB/sec with 8-bit types.

### 39.4 Generating all revbin pairs

Using a primitive polynomial of degree $n$ and its reverse one can generate all nonzero pairs $x$ and $\text{revbin}(x,n)$ as follows [FXT: gf2n/lfsr-revbin-demo.cc]:

```cpp
cr
inline void revbin_next(ulong &x, ulong c, ulong &xr, ulong cr)
// if x and xr are (nonzero) n-bit words that are a revbin pair
// compute the next revbin pair.
// c must be a primitive polynomial, cr its reverse (the reciprocal polynomial).
{
ulong s = ( x & 1UL );
x >>= 1;
 xr <<= 1;
if ( s )
    x ^= (c>>1);
    xr ^= (cr);
}
```

An equivalent technique for computing the revbin permutation (see section 2.1 on page 91) has been proposed in [214]. Figure 39.4-A shows all nonzero 5-bit revbin pairs generated with the primitive polynomial $c = x^5 + x^3 + x^2 + x + 1$ and its reverse.

### 39.5 The number of m-sequences and De Bruijn sequences

The shift register sequences generated with a polynomial of degree $n$ is of maximal length if the polynomial is primitive. The corresponding shift register sequences are called $m$-sequences.
39.5: The number of m-sequences and De Bruijn sequences

We now consider all sequences that are cyclic shifts of each other as the same sequence. For given \( n \) there are as many m-sequences as primitive polynomials (\( P_n = \varphi(2^n - 1)/n \), see section 38.6 on page 840). These can be generated using the linear feedback shift registers described in section 39.1 on page 861.

One might suspect that the using the powers of other elements than \( x \) might lead to additional m-sequences, but this is not the case. Further, the powers of elements of maximal order modulo irreducible non-primitive polynomials do not give additional m-sequences.

The program \([\text{FXT: gf2n/all-primpoly-srs-demo.cc}]\) computes all m-sequences for a given \( n \). The output for \( n = 2, 3, 4, 5, 6 \) is shown in figure 39.5-A.

When a zero is inserted to the (unique) run of \( n-1 \) zeros in an m-sequence then a De Bruijn sequence (DBS) is obtained. A DBS contains all binary words including the all-zero word.

For all \( n \geq 4 \) given there exist more DBSs than m-sequences. For example, for \( n = 6 \) there are 6 m-sequences and 67,108,864 DBSs. An exhaustive search for all DBSs of given length \( L = 2^n \) is possible only for tiny \( n \). The program \([\text{FXT: bits/all-dbs-demo.cc}]\) finds all DBSs for \( n = 3, 4, 5 \). Its output with \( n = 4 \) and \( n = 5 \) (partly) is shown in figure 39.5-B.
Chapter 39: Shift registers

The total number of DBSs equals

\[ S_n = 2^x \quad \text{where} \quad x = 2^{n-1} - n \] (39.5-1)

The two DBSs for \( n = 3 \) are \([\ldots 111.1]\) and \([\ldots 1.111]= [1.111 \ldots]\), reversed sequences are considered different by the formula. The first few values of \( S_n \) are shown in figure 39.5-C. The sequence is entry A016031 of [245]. One has \( S_{n+1} = S_n^2 L_{n-1} \), equivalently \( x_{n+1} = 2x_n + n - 1 \).

The general formula for the number of length-\( n \) base-\( m \) DBSs is \( S_n = m!^{m^{n-1}} / m^n \), as given in [174]. A graph theoretical proof of the formula for the case \( m = 2 \) can be found in [190, p.56], see also [280] entry: De Bruijn graph. For a more efficient approach to generate all DBSs of given length see section 19.2.2 on page 385.

We note that there are several ways to generalize the idea of the De Bruijn cycles to universal cycles of combinatorial objects as described in [86]. The problem of finding a rectangular pattern such that all different patterns of given size appear is discussed in [153].

### 39.6 Auto correlation of m-sequences

We have seen that a De Bruijn sequence (DBS) can be obtained from an m-sequence by inserting a single zero at the longest run of zeros. In the other direction, if we take a DBS and delete a zero from the longest run of zeros then we obtain a sequence of length \( N = 2^n - 1 \) that contains every \( n \)-bit nonzero word. But these sequences are not m-sequences in general: most of them cannot be obtained with an \( n \)-bit LFSR and miss an important property of m-sequences.
For a sequence $M$ of $N - 1$ zeros and ones $M_k$ define the sequence $S$ of elements $S_k$ via

$$ S_k := \begin{cases} +1 & \text{if } M_k = 1 \\ -1 & \text{else} \end{cases} $$

Then, if $M$ is an length-$L$ m-sequence, we have for the cyclic auto correlation (or \textit{auto correlation function}, ACF) of $S$

$$ C_\tau := \sum_{k=0}^{L-1} S_k S_{k+\tau \mod L} = \begin{cases} L & \text{if } \tau = 0 \\ -1 & \text{else} \end{cases} $$

where $L = N - 1$ and $N = 2^n$. That is, $C_0$ equals the length of the sequence, all other entries are of minimal absolute value: they cannot be zero because $L$ is odd.

This property does not hold for most of the ‘truncated’ DBS (where one zero in the single run of $n$ consecutive zeros is removed). Figure \ref{fig:truncatedDBS} shows all (signed) truncated DBS for $n = 4$ and their auto correlations. Only 2 out of the 16 truncated DBS have an auto correlation satisfying relation \ref{eq:acf-truncatedDBS}, these are exactly the m-sequences for $n = 4$.

For odd primes $q$ one can obtain sequences of length $L = q$ whose ACF satisfies

$$ \sum_{k=0}^{L-1} S_k S_{k+\tau \mod L} = \begin{cases} L - 1 & \text{if } \tau = 0 \\ -1 & \text{else} \end{cases} $$

However, the sequences start with a single zero: set $S_0 = 0$ and for $1 \leq k < q$ set $S_k = 1$ if $k$ is a square modulo $q$, else $S_k = -1$. A method to determine whether a number is a square modulo a prime is given in section \ref{sec:squares}.

The first three m-sequences for primes of the form $4k + 1$ and their ACFs are:

- 5: $S$: [0, +1, -1, -1, +1]  
  $C$: [4, -1, -1, -1, -1]

- 13: $S$: [0, +1, -1, +1, -1, -1, -1, +1, +1, -1, +1]  
  $C$: [12, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

- 17: $S$: [0, +1, +1, -1, +1, +1, -1, -1, -1, +1, +1]  
  $C$: [16, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

With primes of the form $q = 4k + 3$ one can construct a sequence of length $L = q$ satisfying relation \ref{eq:acf-truncatedDBS} by simply setting $S_0 = 1$ ($S_0 = -1$ also works) in the sequence just constructed. The sequences for the first three primes $q = 4k + 3$ and their ACFs are:

- 3: $S$: [+1, +1, -1]  
  $C$: [3, -1, -1]

- 7: $S$: [+1, +1, +1, -1, +1, -1, -1]  
  $C$: [7, -1, -1, -1, -1, -1, -1]

- 11: $S$: [+1, +1, -1, +1, +1, -1, -1, +1, -1, -1, +1]  
  $C$: [11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

These sequences can be used for the construction of Hadamard matrices, see section \ref{sec:hadamard}.

### 39.7 Feedback carry shift register (FCSR)

There is an analogue of the LFSR in the modulo world, the \textit{feedback carry shift register} (FCSR). With the LFSR we needed an irreducible (‘prime’) polynomial $C$ where $x$ has maximal order. The powers of $x$ modulo $C$ did run through all different (nonzero) words. Now take a prime $c$ where 2 has maximal order (that is, 2 is primitive root modulo $c$, see section \ref{sec:primes} on page \pageref{sec:primes}). Then the powers of 2 modulo $c$ run through all nonzero values less than $c$.

An implementation of a FCSR is [FXT: \texttt{class fcsr} in \texttt{bpol/fcsr.h}]:

[fxtbook draft of 2008-August-17]
c = 1..1.1 = 37

0 : a= .....1 = 1 w= .1..11 = 19
1 : a= .....1. = 2 w= 1..11. = 38
2 : a= .....1.1 = 3 w= 11..1 = 24
3 : a= ...1..1 = 16 w= 11...1 = 48
4 : a= ...1..1. = 32 w= 1....1. = 32
5 : a= ...1..1.1 = 64 w= ...1..1. = 3
6 : a= ...1..1.1. = 128 w= ...1..1.1 = 3
7 : a= ...1..1.1.1 = 256 w= ...1..1.1 = 27
8 : a= ...1..1.1.1. = 512 w= ...1..1.1. = 6
9 : a= ...1..1.1.1.1 = 1024 w= ...1..1.1. = 13
10 : a= ...1..1.1.1.1. = 2048 w= ...1..1.1.1 = 27
11 : a= ...1..1.1.1.1.1 = 4096 w= ...1..1.1.1 = 2
12 : a= ...1..1.1.1.1.1. = 8192 w= ...1..1.1.1 = 12
13 : a= ...1..1.1.1.1.1.1 = 16384 w= ...1..1.1.1 = 2
14 : a= ...1..1.1.1.1.1.1. = 32768 w= ...1..1.1.1 = 12
15 : a= ...1..1.1.1.1.1.1.1 = 65536 w= ...1..1.1.1 = 2
16 : a= ...1..1.1.1.1.1.1.1. = 131072 w= ...1..1.1.1 = 2
17 : a= ...1..1.1.1.1.1.1.1.1 = 262144 w= ...1..1.1.1 = 2
18 : a= ...1..1.1.1.1.1.1.1.1. = 524288 w= ...1..1.1.1 = 2
19 : a= ...1..1.1.1.1.1.1.1.1.1 = 1048576 w= ...1..1.1.1 = 2
20 : a= ...1..1.1.1.1.1.1.1.1.1. = 2097152 w= ...1..1.1.1 = 2
21 : a= ...1..1.1.1.1.1.1.1.1.1.1 = 4194304 w= ...1..1.1.1 = 2
22 : a= ...1..1.1.1.1.1.1.1.1.1.1. = 8388608 w= ...1..1.1.1 = 2
23 : a= ...1..1.1.1.1.1.1.1.1.1.1.1 = 16777216 w= ...1..1.1.1 = 2
24 : a= ...1..1.1.1.1.1.1.1.1.1.1.1. = 33554432 w= ...1..1.1.1 = 2
25 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1 = 67108864 w= ...1..1.1.1 = 2
26 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1. = 134217728 w= ...1..1.1.1 = 2
27 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1 = 268435456 w= ...1..1.1.1 = 2
28 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1. = 536870912 w= ...1..1.1.1.1 = 2
29 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1. = 1073741824 w= ...1..1.1.1.1 = 2
30 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1. = 2147483648 w= ...1..1.1.1.1.1 = 2
31 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1.1 = 4294967296 w= ...1..1.1.1.1.1 = 2
32 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1.1. = 8589934592 w= ...1..1.1.1.1.1 = 2
33 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1.1. = 17179869184 w= ...1..1.1.1.1.1 = 2
34 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 = 34359738368 w= ...1..1.1.1.1.1 = 2
35 : a= ...1..1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. = 68719476736 w= ...1..1.1.1.1.1 = 2
36 : a= .....1. = 1 w= .1..11 = 19 <-- period restarts
37 : a= .....1. = 2 w= 1..11. = 38
38 : a= .....1. = 4 w= 11..1 = 12

Figure 39.7-A: Successive states of a FCSR with modulus c = 37.

```cpp
class fcsr
{
public:
    ulong a_; // internal state (a_0*2**k modulo c), 1 <= a < c
    ulong w_; // word of the SRS, 1 <= w <= mask
    ulong c_; // a prime with primitive root 2, e.g. 37 = 1..1.1
    ulong mask_; // mask e.g. (with above) mask == 63 == 111111

public:
    fcsr(ulong c)
    {
        c_ = c;
        const ulong h = highest_one(c_);
        mask_ = ( h | (h-1) );
        a_ = 1;
    }

    ~fcsr() { ; }

    ulong next()
    {
        a_ <<= 1; // a *= 2
        if ( a_ > c_ ) a_ -= c_; // reduce mod c

        // update w:
        w_ <<= 1;
        w_ |= a_ & 1;
        w_ &= mask_;

        return w_;
    }

    --snip--
}
```

[fxtbook draft of 2008-August-17]
void set_a(ulong a)
{
    w_ = 0;
    ulong t = c_;
    while ( (t >>= 1) )
    {
        if ( 0==(a & 1) ) a >>= 1;
        else
        {
            a = (a & c_) + ((a ^ c_) >> 1);
        }
    a_ = a;
    next_w();
    } } 
ulong get_a() const  { return a_; }
ulong get_w() const  { return w_; }
}

The routine corresponds to the Galois setup described (for the LFSR) in section 39.2 on page 864, see also [132]. Figure 39.7-A shows the successive states of a FCSR with modulus $c = 37$, it was created with the program [FXT: gf2n/fcsr-demo.cc]. Note that $w$ does not run through all values smaller than $c$ but through a subset of $c − 1$ distinct values smaller than $2^6$.

**Figure 39.7-B:** List of primes $p < 2048$ where 2 is a primitive root.

A list of all primes less than 2048 for which 2 is a primitive root is shown in figure 39.7-B. The shown sequence is entry A001122 of [245]. For further information on the correspondence between LFSR and FCSR see [168].

### 39.8 Linear hybrid cellular automata (LHCA)

**Linear hybrid cellular automata** (LHCA) are 1-dimensional cellular automata (with 0 and 1 the only possible states for each cell) where two different rules are applied dependent of the position, therefore the 'hybrid' in the name. An implementation for LHCA is [FXT: lhca_next() in bpol/lhca.h]:

```c
inline ulong lhca_next(ulong x, ulong r, ulong m)
// LHCA := (1-dim) Linear Hybrid Cellular Automaton.
// Return next state (after x) of the LHCA with
// Rule (defined by) r:
// Rule 150 is applied for cells where r is one, rule 90 else.
// Rule 150 := next(x) = x + leftbit(x) + rightbit(x)
// Rule 90 := next(x) = leftbit(x) + rightbit(x)
// Length defined by m:
// m has to be a burst of the n lowest bits (n: length of automaton)
{  
    r &= x;
    ulong t = (x>>1) ^ (x<<1);
    t ^= r;
    t &= m;
    return t;
}
```

[fxtbook draft of 2008-August-17]
Note that the routine is branch free and implementation in hardware is trivial.

The naming convention for the rules is as follows: draw a table of the eight possible states of a cell together with its neighbors then draw the new states below:

\[
\begin{array}{cccccccc}
XXX & XX0 & XO0 & X00 & XX0 & OX0 & O0X & O00 \\
0 & X & 0 & X & 0 & X & 0 & 0
\end{array}
\]

Now read the lower row as a binary number, the result equals 01011010₂ = 90, so this is rule 90. Rule 150 corresponds to 10010110₂ = 150:

\[
\begin{array}{cccccccc}
XXX & XX0 & XO0 & X00 & XX0 & OX0 & O0X & O00 \\
X & 0 & 0 & X & 0 & X & 0 & 0
\end{array}
\]

---snip---

**Figure 39.8-A:** Partial run of a 16-bit LHCA (left) and complete runs of two 5-bit LHCAs (middle and right). The LHCAs shown have maximal periods.

A run of successive values for the length-16 weight-2 rule vector \( r = 4001₁₆ \) starting with 1 is shown on the left side of figure [39.8-A]. For certain rule vectors \( r \) all \( m = 2^n - 1 \) nonzero values occur, the period is maximal. This is demonstrated in [FXT: `gf2n/lhca-demo.cc`], which for \( n = 5 \) and rule \( r = 1 \) gives the output shown in the middle of figure [39.8-A]. Rule vectors with minimal weight that lead to maximal period are given in [79]. The list [FXT: `minweight_lhca_rule()` in `bpol/lhcarule-minweight.cc`] is taken from that source:

```c
#define R1(n,s1) (1UL<<s1)
#define R2(n,s1,s2) (1UL<<s1) | (1UL<<s2)
extern const ulong minweight_lhca_rule[] =
{ 0, // (empty)  
  R1( 1, 0),  
  R1( 2, 0),  
  R1( 3, 0),  
  R2( 4, 0, 2),  
  R1( 5, 0),  
  R1( 6, 0),  
  R1( 7, 2),  
  R2( 8, 1, 2),  
  R1( 9, 0),  
  R2(10, 1, 6),  
  R1(11, 0),  
  R2(12, 2, 6),  
  R1(13, 4),
```

---snip---

[fxtbook draft of 2008-August-17]
Up to $n = 500$ there is always a rule with weight at most 2 that leads to the maximal period. The full list of these rules is given in [FXT: data/minweight-lhca-rules.txt].

### 39.8.1 Conversion of LHCAs to binary polynomials

To convert a length-$n$ LHCA to a binary polynomial proceed as follows: initialize $p_{-1} := 0$, $p_0 := 1$, and iterate for $k = 1, 2, \ldots, n$:

$$ p_k := (x + r_{k-1}) p_{k-1} + p_{k-2} $$

where $r_i$ denotes bit $i$ of rule $r$. The degree of the returned polynomial $p_n$ is $n$. An implementation of the algorithm is [FXT: lhca2poly() in bpol/lhca.h]:

```c
1 inline ulong lhca2poly(ulong r, ulong n)
2 // Return binary polynomial p that corresponds to the length-n LHCA rule r.
3 {
4   ulong p2 = 0, p1 = 1;
5   while ( n-- )
6     {
7       ulong m = r & 1;
8       r >>= 1;
9       ulong p = (p1<<1) ^ p2;
10      if ( m ) p ^= p1;
11     p2 = p1; p1 = p;
12   }
13   return p1;
14 }
```

The lexicographically first minimum-weight LHCA rules and their binary polynomials are shown in figure 39.8-B. The table was created by the program [FXT: gf2n/lhca2poly-demo.cc]. For rules of maximal period the polynomials are primitive.
An LHCA rule and its reverse give the identical polynomial. The polynomials corresponding to a LHCA rule and its complement are either both reducible or irreducible: if \( p(x) \) corresponds to the LHCA rule \( r \) then \( p(x + 1) \) corresponds to the rule that is the complement of \( r \).

<table>
<thead>
<tr>
<th>LHCA rule</th>
<th>polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( r=...............................1 ) c=...............................11 ( r= [0] )</td>
<td></td>
</tr>
<tr>
<td>2: ( r=...............................1 ) c=..............................111 ( r= [0] )</td>
<td></td>
</tr>
<tr>
<td>3: ( r=...............................1 ) c=............................11.1 ( r= [0, 2] )</td>
<td></td>
</tr>
<tr>
<td>4: ( r=...............................1 ) c=..........................111.11 ( r= [0] )</td>
<td></td>
</tr>
<tr>
<td>5: ( r=...............................1 ) c=.........................11111 ( r= [0] )</td>
<td></td>
</tr>
<tr>
<td>6: ( r=...............................1 ) c=.............................11111 ( r= [0] )</td>
<td></td>
</tr>
<tr>
<td>7: ( r=...............................1 ) c=............................11111 ( r= [0, 2] )</td>
<td></td>
</tr>
<tr>
<td>8: ( r=...............................1 ) c=..........................11111 ( r= [0, 2] )</td>
<td></td>
</tr>
<tr>
<td>9: ( r=...............................1 ) c=..............................11111 ( r= [0] )</td>
<td></td>
</tr>
<tr>
<td>10: ( r=............................1111 ) c=..............................11111111 ( r= [0, 1, 2, 3] )</td>
<td></td>
</tr>
<tr>
<td>11: ( r=...............................1 ) c=..............................11111111 ( r= [0] )</td>
<td></td>
</tr>
<tr>
<td>12: ( r=............................1111 ) c=..............................11111111 ( r= [0, 2, 4] )</td>
<td></td>
</tr>
<tr>
<td>13: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>14: ( r=............................1111 ) c=..............................11111111 ( r= [0, 2, 4] )</td>
<td></td>
</tr>
<tr>
<td>15: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>16: ( r=............................1111 ) c=..............................11111111 ( r= [0, 2, 4] )</td>
<td></td>
</tr>
<tr>
<td>17: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>18: ( r=............................1111 ) c=..............................11111111 ( r= [0, 2, 4] )</td>
<td></td>
</tr>
<tr>
<td>19: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>20: ( r=............................1111 ) c=..............................11111111 ( r= [0, 2, 4] )</td>
<td></td>
</tr>
<tr>
<td>21: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>22: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>23: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>24: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>25: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>26: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>27: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>28: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>29: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>30: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
<tr>
<td>31: ( r=............................1111 ) c=..............................11111111 ( r= [0, 3] )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 39.8-C: Low-bit LHCA rules and the corresponding binary polynomials.

Figure 39.8-C shows a list of LHCA rules with maximal period where the highest bit lies in the lowest possible position. The list can be produced with [FXT: gf2n/lowbit-lhca-demo.cc]. For software implementation of LHCA that need more than one machine word low-bit rules are advantageous. A table of low-bit LHCA rules corresponding to primitive polynomials up to \( n = 400 \) is given in [FXT: data/lowbit-lhca-rules.txt]. The maximal value of a rule that occurs is \( r = 1293 \) for \( n = 380 \) so the table can be stored compactly, for example, in an array of 16-bit integers.

Figure 39.8-D shows a list of LHCA rules that have minimal weight and the highest bit in the lowest possible position. The list can be produced with [FXT: gf2n/minweight-lowbit-lhca-demo.cc].

### 39.8.2 Conversion of irreducible binary polynomials to LHCA

Computing the LHCA corresponding to an irreducible binary polynomial \( p(x) \) proceeds in two steps. Firstly, the following quadratic equation over GF\((2^n)\) must be solved for \( Z \):

\[
Z^2 + (x^2 + x)p'(x) Z + 1 = 0 \mod p(x) \tag{39.8-2}
\]
39.8: Linear hybrid cellular automata (LHCA)

The algorithm is given in section 40.4 on page 895. The second step is a GCD computation. Set $Z_{n-1} := p$, $Z_{n-2} := Z$, and compute successively $Z_{n-3}$, $Z_{n-4}$, ..., $Z_0$ such that

$$Z_{n-1} = (x + r_0) Z_{n-2} + Z_{n-3}$$

$$Z_{n-2} = (x + r_1) Z_{n-3} + Z_{n-4}$$

$$\vdots$$

$$Z_2 = (x + r_{n-3}) Z_1 + 1$$

$$Z_1 = (x + r_{n-2}) Z_0 + 1$$

$$Z_0 = (x + r_{n-1}) 1 + 0$$

Each step consists of a computation of polynomial quotient and remainder (see section 38.1 on page 819). The vector $[r_{n-1}, r_{n-2}, \ldots, r_0]$ then gives the LHCA rule. An implementation of the method is [FXT: bpol/bpol2lhca.cc]:

```c
ulong poly2lhca(ulong p) // Return LHCA rule corresponding to the binary polynomial P.
// Must have: P irreducible.
{
    ulong dp = bitpol_deriv(p);
    const ulong h = bitpol_h(p);
    ulong b = dp;
    b ^= bitpolmod_times_x(b, p, h); // p' * (x+1)
    b = bitpolmod_times_x(b, p, h); // p' * (x^2+x)
    ulong r0, r1; // solutions of 1 + (p'*(x*x+x))z + z*z == 0 modulo p
    bool q = bitpolmod_solve_quadratic(1, b, 1, r0, r1, p);
    if (q == eq) return 0;
    // GCD steps:
    ulong r = 0; // rule vector
    ulong x = p, y = r0; // same result with r1
```
while ( y )

{  
    ulong tq, tr;
    bitpol_divrem(x, y, tq, tr);
    r <<= 1;
    r |= (tq & 1);
    x = y;
    y = tr;
}

return r;

The described algorithm is given in the very readable paper [78] which is recommended for further studies. Note that the paper uses the reversed rule which can be obtained by inserting the line

\[ r = \text{revbin}(r, \text{bitpol}_\text{deg}(p)) \]

just before the final return statement. The program [FXT: \texttt{gf2n/poly2lhca-demo.cc}] converts a given polynomial into the corresponding LHCA rule.

### 39.9 Additive linear hybrid cellular automata

The algorithm for the conversion of LHCA rules to binary polynomials is a special case of a general method for additive cellular automata. An automaton is called additive if, for all words \( a \) and \( b \),

\[
N(a) + N(b) = N(a + b) \tag{39.9-1}
\]

where \( N(x) \) is the next state after the state \( x \) and addition is bit-wise (XOR).

#### 39.9.1 Conversion into binary polynomials

For additive automata the action of \( N \) on a binary word can be described by a matrix over GF(2): Let \( e_k \) be the word where only bit \( k \) is set. The matrix whose \( k \)-th row is \( N(e_k) \) is the matrix we seek. For example, for the LHCA with cyclic boundary condition (CLHCA) whose action \( N \) can be implemented as [FXT: \texttt{bpol/clhca.h}]

\[
\text{ulong clhca_next(ulong x, ulong r, ulong n)}
\]

\{
  r ^= x;
  ulong t = x ^ \text{bit_rotate_right}(x, 1, n);
  t ^= r;
  return t;
\}

we obtain for the rule \( r := [r_0, r_1, \ldots, r_{n-1}] \) the matrix

\[
M_r := \begin{bmatrix}
  s_0 & 1 & s_1 & 1 & s_2 & 1 & \cdots & s_{n-3} & 1 & s_{n-2} & 1 & s_{n-1} \\
  1 & 1 & s_2 & 1 & s_3 & 1 & \cdots & s_{n-3} & 1 & s_{n-2} & 1 & s_{n-1} \\
  s_0 & 1 & s_1 & 1 & s_2 & 1 & \cdots & s_{n-3} & 1 & s_{n-2} & 1 & s_{n-1} \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
  1 & 1 & s_2 & 1 & s_3 & 1 & \cdots & s_{n-3} & 1 & s_{n-2} & 1 & s_{n-1} \\
\end{bmatrix} \tag{39.9-2}
\]

where \( s_k \) is the complement of \( r_k \) and blank entries denote zero. The binary polynomial corresponding to the automaton is the characteristic polynomial of the matrix \( M_r \). We use the routine \texttt{bitmat_charpoly()} given in [FXT: \texttt{bmat/bitmat-charpoly.cc}] (which uses a reduction to the Hessenberg form as given in [91, p.55]).
inline ulong clhca2poly(ulong r, ulong n)
// Compute the binary polynomial corresponding
// to the length-n CLHCA with rule r.
{
    ALLOC(ulong, M, n);
    for (ulong k=0; k<n; ++k) M[k] = clhca_next( 1UL<<k, r, n );
    ulong c = bitmat_charpoly(M, n);
    return c;
}

The routine computes the polynomial for any additive automaton if the call to clhca_next() is replaced by the update for the automaton. Note that with our particular example the matrix is already in Hessenberg form so we could directly call bitmat_hessenberg2charpoly().

### 39.9.2 Properties of the CLHCA

<table>
<thead>
<tr>
<th>k</th>
<th>r = CLHCA rule</th>
<th>c = binary polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.111</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>2</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>3</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>4</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>5</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>6</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>7</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>8</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>9</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>10</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>11</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>12</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>13</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>14</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
<tr>
<td>15</td>
<td>.1111111111111</td>
<td>1111111111111111111111</td>
</tr>
</tbody>
</table>

Figure 39.9-A: Rules of identical weight lead to essentially identical CLHCA: all tracks in the successive states of all weight-3 (length-4) automata are cyclic shifts of each other.

The polynomials for the given automata depend only on the number of bits set in the rule r. Figure 39.9-A shows the successive states for all length-4 CLHCA with rules of weight 3. As there are essentially only n + 1 different automata of length n we only need to investigate the rules with the lowest k bits set for k = 0, 1, ..., n. The polynomials for the length-17 automata are shown in figure 39.9-B. If we write $C_n(x)$ for the polynomial for the length-n rule with k set bits then $C_{n,k}(x) = C_n(x+1)$. Indeed, they can be given in the closed form $C_{n,k}(x) = 1 + x^k (1 + x)^{n-k}$. Thereby the polynomials can be computed with the simple routine [FXT: bpol/clhca.h]

```c
inline ulong clhca2poly(ulong r, ulong n)
```
Chapter 39: Shift registers

Figure 39.9-C: The length-5, weight-\( w \) CLHCA has maximal period for \( w = 2 \) and \( w = 3 \). The successive states are shown for both automata (left and middle). The weight-1 rule leads to a period of 21 (right).

Figure 39.9-D: The polynomials (\( c \), right) corresponding to rules of lowest weight (\( w \)) such that the length-\( n \) (\( n \leq 30 \)) CLHCA has maximal period.
Additive linear hybrid cellular automata

```c
{ 
    ulong c = 1UL << n;
    for (ulong k=0; k<n; ++k) if ( 0==(r & (1UL<<k)) ) c ^= (c>>1);
    c ^= 1;
    return c;
}
```

With $n = 5$ there are just two rules that lead to maximal periods, $r = [0,0,0,1,1]$ (weight 2), and $r = [0,0,1,1,1]$ (weight 3). The successive states for both rules are shown in figure 39.9-C. The polynomials corresponding to the rules of minimal weight for all length-$n$ automata where $n \leq 30$ are given in figure 39.9-D. The sequence of values $n$ where a primitive length-$n$ CLHCA exists starts as:

$$2, 3, 4, 5, 6, 7, 9, 10, 11, 15, 17, 18, 20, 21, 22, 23, 25, 28, 29, 31, 33, 35, 36, 39, 41, 47, 49, 52, 55, 57, 58, 60, 63, \ldots$$

It coincides with entry A073726 of [245], values $n$ such that there is a primitive trinomial of degree $n$. The sequence was computed with the program [FXT: gf2n/clhca-demo.cc]. A list of CLHCA rules with maximal period is given in [FXT: data/clhca-rules.txt].
Chapter 40

Binary finite fields: $\text{GF}(2^n)$

We introduce the binary finite fields $\text{GF}(2^n)$. The polynomial representation is used for stating some basic properties. The underlying arithmetical algorithms are given in chapter \[38\] An introduction of the representation by normal bases follows. Certain normal bases are advantageous for hardware implementations of the arithmetical algorithms. Finally, several ways of computing the number of irreducible binary normal polynomials are given.

Binary finite fields are important for applications like error correcting codes and cryptography.

40.1 Arithmetic and basic properties

In chapter \[25\] we have met the finite fields $\mathbb{Z}/p\mathbb{Z} = \text{GF}(p)$ for $p$ a prime. The ‘GF’ stands for Galois Field, another symbol often used is $\mathbb{F}_p$. The arithmetic in $\text{GF}(p)$ is the simply the arithmetic modulo $p$.

There are more finite fields: for every prime $p$ there are fields with $Q = p^n$ elements for all $n \geq 1$. All elements in a finite field $\text{GF}(p^n)$ can be represented as polynomials modulo a degree-$n$ irreducible polynomial $C$ with coefficients over the field $\text{GF}(p)$. The arithmetic to be used is polynomial arithmetic modulo $C$. As in general there is more than one irreducible polynomial of degree $n$ it might seem that there is more than one field $\text{GF}(Q)$ for given $Q = p^n$. There isn’t. Using different polynomials as modulus leads to isomorphic representations of the same field. The field $\text{GF}(p^n)$ is called an extension field of $\text{GF}(p)$. The field $\text{GF}(p)$ is called the ground field (or bas field) of $\text{GF}(p^n)$.

When speaking about an element of $\text{GF}(Q)$ one can think of a polynomial modulo some fixed irreducible polynomial $C$ (modulus). For example, the product of two elements can be computed as the polynomial product modulo $C$. For the equivalent construction using the polynomial $x^2 + 1$ with real coefficients that leads to the complex numbers see section \[37.12\] on page 803.

The elements zero, the neutral element of addition, and one, the neutral element of multiplication, are the constant polynomials with constant zero and one, respectively. This does not depend on the choice of the modulus.

Multiplication with an element of the ground field is called scalar multiplication. In this section an element of the ground field is denoted by $u$. Scalar multiplication corresponds to the multiplication of every coefficient of (the polynomial representing) the field element $a$ by $u$.

We restrict our attention mostly to $Q = 2^n$, that is, the binary finite fields $\text{GF}(2^n)$ as we have seen the algorithms for the underlying arithmetic in section \[38\] on page 819.
40.1.1 Characteristic and linear functions

If we add any element of the field GF($p^n$) $p$ times to zero the result will be zero. One calls $p$ the characteristic of the field. For infinite fields such as $\mathbb{C}$ the characteristic is said to be zero (while it should really be $\infty$). Note that the notion “multiplication is repeated addition” is meaningless in extension fields GF($p^n$).

For GF($p^n$) we have

\[ (u + v)^p = u^p + v^p \]  (40.1-1)

because the binomial coefficients \( \binom{p}{k} \) are divisible by $p$ for $k = 1, 2, \ldots, p - 1$. For GF($2^n$):

\[ (u + v)^2 = u^2 + v^2 \]  (40.1-2)

We call a function $f$ linear if the relation

\[ f(u_1 \cdot a + u_2 \cdot b) = u_1 \cdot f(a) + u_2 \cdot f(b) \]  (40.1-3)

holds for $u_1$ and $u_2$ from the ground field. The linear functions in GF($p^n$) are of the form

\[ f(x) = \sum_{k=0}^{n-1} u_k \cdot x^{b^k} \]  (40.1-4)

where the $u_k$ are again in the ground field. Linear functions can be computed using lookup tables. In GF($2^n$) these are all functions of the form

\[ f(x) = \sum_{k=0}^{n-1} u_k \cdot x^{2^k} \]  (40.1-5)

40.1.2 Squaring

Squaring (and raising to any power $2^k$) is a linear operation in GF($2^n$). The linearity can be used to accelerate the computation of squares. Write

\[
\begin{align*}
(u_0 + u_1 x + u_2 x^2 + \ldots + u_{n-1} x^{n-1})^2 &= u_0 + u_1 x^2 + u_2 x^4 + \ldots + u_{n-1} x^{2(n-1)} \\
&=: u_0 s_0 + u_1 s_1 + u_2 s_2 + \ldots + u_{n-1} s_{n-1}
\end{align*}
\]  (40.1-6)

One has to precompute and store the values $s_i = x^{2^i} \mod C$ for $i = 0, 1, 2, \ldots, n - 1$. For successive square computations it is only necessary to add (that is, XOR) those $s_i$ corresponding to nonzero $u_i$. For example, with $n = 13$ and the polynomial modulus $C = x^{13} + x^4 + x^3 + x^1 + 1$ one obtains the table

<table>
<thead>
<tr>
<th>i</th>
<th>s_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
</tr>
<tr>
<td>4</td>
<td>1111111</td>
</tr>
<tr>
<td>5</td>
<td>1111111</td>
</tr>
<tr>
<td>6</td>
<td>1111111</td>
</tr>
<tr>
<td>7</td>
<td>1111111</td>
</tr>
<tr>
<td>8</td>
<td>1111111</td>
</tr>
<tr>
<td>9</td>
<td>1111111</td>
</tr>
<tr>
<td>10</td>
<td>1111111</td>
</tr>
<tr>
<td>11</td>
<td>1111111</td>
</tr>
<tr>
<td>12</td>
<td>1111111</td>
</tr>
<tr>
<td>13</td>
<td>1111111</td>
</tr>
</tbody>
</table>

The squares $s_0, s_1 \ldots s_{\lfloor (n-1)/2 \rfloor}$ have the trivial form $s_i = x^{2^i}$ which can be used to further accelerate the computation.
40.1.3 Computation of the trace

The trace of an element \( u \) in \( \text{GF}(2^n) \) is defined as

\[
\text{Tr}(u) := a + a^2 + a^4 + a^8 + \ldots + a^{2^n-1} = \sum_{j=0}^{n-1} a^{2^j}
\]  

The trace of any element is either zero or one. The trace of zero is always zero. The trace of one in \( \text{GF}(2^n) \) equals one for odd \( n \), else zero, that is, \( \text{Tr}(1) = n \mod 2 \). Exactly half of the elements have trace one. The trace of the sum of two elements is the sum of the traces of the elements:

\[
\text{Tr}(a + b) = \text{Tr}(a) + \text{Tr}(b)
\]  

The trace of the square of an element is the trace of that element:

\[
\text{Tr}(a^2) = \text{Tr}(a)
\]  

With \( u \) zero or one (an element of the ground field \( \text{GF}(2) \)) one has

\[
\text{Tr}(u \cdot a) = u \cdot \text{Tr}(a)
\]  

Thereby, for \( u_1 \) and \( u_2 \) from the ground field,

\[
\text{Tr}(u_1 \cdot a + u_2 \cdot b) = u_1 \cdot a + u_2 \cdot b
\]

That is, the trace function is linear.

A fast algorithm to compute the trace uses the trace vector, a precomputed table \( t_i = \text{Tr}(x^i) \) for \( i = 0, 1, 2, \ldots, n - 1 \), and the linearity of the trace

\[
\text{Tr}(u) = \sum_{i=0}^{n-1} u_i t_i
\]

where the \( u_i \) are zero or one. Precompute the trace vector \( tv \) whose bits are the traces of the powers of \( x \), and later compute the trace of an element via \( \text{FXT: gf2n_fast_trace()} \) in \file{bpol/gf2n-trace.h}:

```c
inline ulong gf2n_fast_trace(ulong a, ulong tv)
// Fast trace computation of the trace of a
// using the pre-calculated table tv.
{ return parity( a & tv ); } // scalar product over GF(2)
```

Given the trace vector it is also easy to find elements of trace zero or one by simply taking the lowest unset or set bit of the vector, respectively. There are polynomials such that the trace vector contains just one nonzero bit, see section 40.3 on page 893.

40.1.4 Inverse and square root

The number of elements in \( \text{GF}(Q) \) equals \( Q \). For any element \( a \in \text{GF}(Q) \) one has

\[
a^Q = a
\]

and thereby \( a^{Q-1} = 1 \). So we can compute the inverse \( a^{-1} \) of a nonzero element \( a \) as

\[
a^{-1} = a^{Q-2}
\]
All elements except zero are invertible in a field. That is, the number of invertible elements (units) in \( GF(Q) \) equals \(|GF(Q)^*| = Q - 1 = p^n - 1 \).

Every element \( a \) of \( GF(2^n) \) has a unique square root \( s \) which can be computed as

\[
s = a^{Q/2} = a^{2^{n-1}}
\]

(40.1-15)

It can be computed by squaring the element \( n - 1 \) times. But the square root is a linear function, so we can again apply table lookup methods. A method that uses the precomputed value \( \sqrt{x} \) is described in [119]: for an element \( a = \sum k a_k x^k \) we have

\[
\sqrt{a} = \sum \text{even } k \ a_k x^{k/2} + \sqrt{x} \sum \text{odd } k \ a_k x^{(k-1)/2}
\]

(40.1-16)

The only nontrivial operation is the multiplication with \( \sqrt{x} \). The method is easily generalized for higher roots: for example, for the third root define \( S(m) := \sum_{k \equiv m \mod 3} a_k x^{(k-m)/3} \), then

\[
\sqrt[3]{a} = S(0) + \sqrt{x} S(1) + \left( \sqrt{x} \right)^2 S(2)
\]

(40.1-17a)

\[
= S(0) + \sqrt{x} \left[ S(1) + \sqrt{x} S(2) \right]
\]

(40.1-17b)

### 40.1.5 Order and primitive roots

The order of an element \( a \) is the least positive exponent \( r \) such that \( a^r = 1 \). The maximal order of an element in \( GF(2^n) \) equals \( 2^n - 1 = Q - 1 \). An element of maximal order is called a generator (or primitive root) as its powers 'generate' all nonzero elements in the field. The order of a given element \( a \) in \( GF(2^n) \) can be computed like

```
function order(a, n)
{
    if a==0 then return 0 // a not a unit
    h := 2**n - 1 // number of units
    e := h
    {np, p[], k[]} := factorization(h) // h==product(i=0..np-1, p[i]**k[i])
    for i:=0 to np-1
    {
        f := p[i]**k[i]
        e := e / f
        g1 := a**e // modulo polynomial
        while g1!=1
        {
            g1 := g1**p[i] // modulo polynomial
            e := e * p[i]
            p[i] := p[i] - 1
        }
    }
    return e
}
```

The C++ implementation is given in [FXT: \( \text{gf2n\_order()} \) in \( \text{bpol/gf2n\_order.cc} \):
Let $a$ be an element of order $r$, then the order of $a^k$ is
\[
\text{ord}(a^k) = \frac{r}{\gcd(k,r)} \quad (40.1\text{-}18)
\]
The order remains unchanged if $\gcd(k,r) = 1$. Let $N = 2^n - 1$ and $g$ a generator of $GF(2^n)$. Then all $\varphi(N)$ generators can obtained as $g^k$ where $\gcd(k,N) = 1$. For $N$ a (Mersenne-) prime the order of all invertible elements except 1 is $N$.

40.1.6 Implementation

A C++ class for computations in the fields $GF(2^n)$ with $n$ not greater than BITS_PER_LONG is [FXT: class GF2n in [bpol/gf2n.h]]:

```cpp
class GF2n {
    public:
        ulong v_; // the 'n' in GF(2**n)
        static ulong n_; // the 'n' in GF(2**n)
        static ulong c_; // polynomial modulus
        static ulong h_; // auxiliary bitmask for computations
        static ulong mm_; // 2**n - 1 == max order (a Mersenne number)
        static ulong g_; // a generator (element of maximal order)
        static ulong tv_; // trace vector
        static ulong sqr_tab[BITS_PER_LONG]; // table for fast squaring
        static factorization mfact_; // factorization of max order
        static char* pc_; // chars to print zero and one: e.g. "01" or ".1"
        static GF2n zero; // zero (neutral element wrt. addition) in GF(2**n)
        static GF2n one; // one (neutral element wrt. multiplication) in GF(2**n)
        static GF2n tr1e; // an element with trace == 1

        public:
            explicit GF2n() { ; }
            explicit GF2n(const ulong i) : v_(i & mm_) { ; }
            GF2n(const GF2n &g) : v_(g.v_) { ; }
            ~GF2n() { ; }

        public:
            explicit GF2n() { ; }
            explicit GF2n(const ulong i) : v_(i & mm_) { ; }
            GF2n(const GF2n &g) : v_(g.v_) { ; }
            ~GF2n() { ; }

    public:
        explicit GF2n() { ; }
        explicit GF2n(const ulong i) : v_(i & mm_) { ; }
        GF2n(const GF2n &g) : v_(g.v_) { ; }
        ~GF2n() { ; }

Before doing anything you have to call the initializing function [FXT: GF2n::init() in [bpol/gf2n.cc]]:
```
Chapter 40: Binary finite fields: GF(2^n)

1 // if INIT_ASSERT is defined, asserts are C asserts,
2 // else init() returns false if one of the tests fail:
3 #define INIT_ASSERT
4
5 bool // static
6 GF2n::init(ulong n, ulong c/*=0*/, bool normalq/*=0*/, bool trustme/*=0*/)
7 // Initialize class GF(2**n) for 0<n<=BITS_PER_LONG.
8 // If an irreducible polynomial c is supplied it is used as modulus,
9 // else a primitive polynomial of degree n is used.
10 // Irreducibility of c is asserted.
11 // When normalq is set a primitive normal polynomial is used,
12 // if in addition c is supplied, normality of c is asserted.
13 // When trustme is set the asserts are omitted.
14 {
15      [--snip--]
16      if ( n_ < BITS_PER_LONG ) // test only works for polynomials that fit into words
17      {
18          if ( ! trustme )
19              {
20                  #ifdef INIT_ASSERT
21                      jjassert( bitpol_irreducible_q(c_, h_) );
22                  #else
23                      if ( ! bitpol_irreducible_q(c_, h_) ) return false;
24                  #endif
25              }
26      }[-snip--]
27  
28 }

n = 4  GF(2^n)
   c = 1..11 == x^4 + x + 1 (polynomial modulus)
   mm = .1111 == 15 = 3 * 5 (maximal order)
   h = .1... (aux. bitmask)
   g = ...1. (element of maximal order)
   v = ...1. (traces of x^i)
   trie = .1... (element with trace=1)

k : f:=g**k  Tr(f)  ord(f)  f*f  sqrt(f)
.... : ....  1 0 15 111 111
   ...: ...1 0 15 .1... .1...
   ...1: ...1 0 15 111 .1...
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111
   ...1: ...1 0 15 111 111

Figure 40.1-A: Powers of the generator g = x in GF(2^4) with a primitive polynomial modulus.

The class defines all the standard operators like the binary operators ‘+’ and ‘-’ (which are the same operation in GF(2^n)), ‘*’ and ‘/’, the comparison operators ‘==’ and ‘!=’, also the computation of inverse, powering, order, and trace. The algorithms used for the arithmetic operations are described in section 38.3 on page 830. We give the method for the inverse and the arithmetic shortcut operators as examples:

1 GF2n inv() const
2 {
3     GF2n z;
4     z.v_ = bitpolmod_inverse(v_, GF2n::c_);
5     return z;
6 }
7
8 friend inline GF2n & operator += (GF2n &z, const GF2n &f)
9 {
10     z.v_ ^= f.v_; return z;
11 }
12
13 friend inline GF2n & operator -= (GF2n &z, const GF2n &f)
14 {
15     z.v_ ^= f.v_; return z;
16 }

[fxtbook draft of 2008-August-17]
40.2 Minimal polynomials

The minimal polynomial of an element \( a \) in \( \mathbb{GF}(2^n) \) is defined as the polynomial of least degree which has \( a \) as a root. The minimal polynomial can be computed as the product

\[
p_a(x) := \prod_{k=0}^{r-1} (x - a^{2^k})
\]  

(40.2-1)

where \( r \) is the smallest positive integer such that \( a^{2^r} = a \). The minimal polynomial of any element is irreducible and its degree is a divisor of \( n \).

The zeros of the polynomial are \( a, a^2, a^4, a^8, \ldots, a^{2^{r-1}} \). Note that \( (a^{2^{r-1}})^2 = a \), that is, \( p_a = p_{a^2} = p_{a^4} = \ldots = p_{a^{2^{r-1}}} \). The elements \( a^2, a^4, \ldots \) are called the conjugates of \( a \). For the field \( \mathbb{GF}(2^n) \) the minimal polynomial has the form \( \prod_{k=0}^{r-1} (x - a^{2^k}) \).

From the definition it can be seen that the coefficients of the minimal polynomial lie in \( \mathbb{GF}(2^n) \), however, all of them are either zero or one so the lie in \( \mathbb{GF}(2) \). Thereby the computation has to be carried out using the arithmetic in \( \mathbb{GF}(2^n) \) but the final result is a binary polynomial [FXT: bpol/gf2n-minpoly.cc].
\( n = 6 \) \( \text{GF}(2^n) \)

\( c = 1 \ldots 11 = x^6 + x + 1 \) \( \text{(polynomial modulus)} \)

\( \text{mm} = .111111 = 63 = 3^2 \cdot 7 \) \( \text{(maximal order)} \)

<table>
<thead>
<tr>
<th>( k )</th>
<th>( f := g^k )</th>
<th>( \text{ord}(f) )</th>
<th>( \text{Tr}(f) )</th>
<th>( p := \text{minpoly}(f) )</th>
<th>( \text{deg}(p) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>.11</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>1.11</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.111</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>1.11</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>1.111</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>1.11</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>.111</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>1.111</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>1.111</td>
<td>6</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>.111</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>.111</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>.111</td>
<td>3</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>.111</td>
<td>3</td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>52</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>53</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>54</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>.111</td>
<td>3</td>
</tr>
<tr>
<td>55</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>57</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1.1111</td>
<td>6</td>
</tr>
<tr>
<td>58</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>59</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
<tr>
<td>62</td>
<td>1</td>
<td>63</td>
<td>1</td>
<td>11.11</td>
<td>6</td>
</tr>
</tbody>
</table>

**Figure 40.2-A:** Minimal polynomials of the powers of a generator in \( \text{GF}(2^6) \). Polynomials of degree \( n = 6 \) that are non-primitive are marked with an ‘\( \mathcal{N} \)’. The trace of an element equals the coefficient of \( x^n - 1 \) of its minimal polynomial.
A version of the routine that does not depend on the class \texttt{GF2n} is given in [FXT: bpol/bitpolmod-minpoly.h]. The program [FXT: gf2n/gf2n-minpoly-demo.cc] prints the minimal polynomials for the powers of a primitive element \(g\), see figure 40.2-A. The polynomials (of maximal degree) that are non-primitive are marked by an ‘\(^N\)’. The minimal polynomials for \(g^k\) are non-primitive or of degree smaller than \(n\) whenever \(\gcd(k, 2^n - 1) \neq 1\).

Let \(C\) be a irreducible polynomial of degree \(n\) and \(a\) an element such that \(r = \text{ord}_C(a)\) is the order of \(a\) modulo \(C\). Then the order of \(x\) modulo the minimal polynomial of \(a\) is also \(r\). Thereby a primitive polynomial can be determined from an irreducible polynomial \(C\) and a generator \(g\) modulo \(C\) by computing the minimal polynomial of \(g\).

With a primitive polynomial (and the generator \(g = x\)) the minimal polynomial of the element \(x^k\) is primitive if \(k\) is a Lyndon word and \(\gcd(k, n) = 1\). With a fast algorithm for the generation of Lyndon words one can therefore generate all primitive polynomials as shown in section 38.10 on page 852.

An alternative technique for the computation of the minimal polynomial is given in [243].

### 40.3 Fast computation of the trace vector

We give two methods for the computation of the trace vector and some properties of the trace vector for certain field polynomials.

#### 40.3.1 Computation via Newton’s formula

Let \(C(x)\) be a polynomial with \(n\) roots \(a_0, a_1, \ldots, a_{n-1}\)

\[
C(x) = \prod_{k=0}^{n-1} (x - a_k) = \sum_{k=0}^{n} c_k x^k
\]

(40.3-1)

Following [262 sec.32] we define

\[
s_k := a_0^k + a_1^k + \ldots + a_{n-1}^k
\]

(40.3-2)
Then, for \( m = 1, \ldots, n \), we have Newton’s formula:

\[
m c_{n-m} = - \sum_{j=0}^{m-1} s_{m-j} c_{n-j}
\]  

(40.3-3)

Now let \( C = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n \) be an irreducible polynomial with coefficients in \( \mathbb{GF}(p) \). Its roots are \( a_i \) (and the conjugates) \( a_i^p, a_i^{p^2}, \ldots, a_i^{p^{n-1}} \). Let \( t_0 = n \) and \( t_i = \text{Tr}(a_i^j) \) (computationally \( x \) is a root of \( C \), so \( t_i = \text{Tr}(x^j) \)). Note that \( t_0, \ldots, t_{n-1} \) are the elements of the trace vector, see relation 40.1-12 on page 887. Using \( c_n = 1 \) (monic polynomial \( C \)) and \( s_j \) we rewrite Newton’s formula as

\[
t_1 = -c_{n-1} \tag{40.3-4a}
\]

\[
t_2 = -c_{n-1} t_1 - 2 c_{n-2} \tag{40.3-4b}
\]

\[
t_3 = -c_{n-1} t_2 - c_{n-2} t_1 - 3 c_{n-3} \tag{40.3-4c}
\]

\[
t_4 = -c_{n-1} t_3 - c_{n-2} t_2 - c_{n-3} t_1 - 4 c_{n-4} \tag{40.3-4d}
\]

\[
t_5 = -c_{n-1} t_4 - c_{n-2} t_3 - c_{n-3} t_2 - c_{n-4} t_1 - 5 c_{n-5} \tag{40.3-4e}
\]

\[\vdots\]

\[
t_k = -c_{n-1} t_{k-1} - c_{n-2} t_{k-2} - \ldots - c_{n-k} t_1 - k c_{n-k} \tag{40.3-4f}
\]

To compute the trace vector for the field \( \mathbb{GF}(p^n) \), make the assignments in the given order, and finally compute \( t_0 = n \mod p \). The computation does not involve any polynomial modular reduction so the method can be worthwhile even for the determination of the trace of just one element.

With binary finite fields, the components with even subscripts can be computed as \( t_{2k} = t_k \). During the computation we set \( t_0 = 0 \) and correct the value at the end of the routine. An implementation of the implied algorithm is [FXT: `bpol/gf2n-trace.cc`]:

```c
ulong gf2n_trace_vector_x(ulong c, ulong n)
// Return vector of traces of powers of x, where
// x is a root of the irreducible polynomial C.
// Must have: n == degree(C)
{
    c &= ~(2UL<<(n-1)); // remove coefficient c[n]
    ulong t = 1; // set t[0]=1, will be corrected at the end
    for (ulong k=1; k<n; ++k)
    {
        if (k & 1) // k odd: use recursion
            {
                ulong cv = c >> (n-k); // polynomials coefficients [n-1]..[n-k]
                cv &= t; // vector {j=1, k, c[n-j]*t[k-j]}
                cv = parity(cv); // sum {j=1, k, c[n-j]*t[k-j]}
                t |= (cv<<k);
            }
        else // k even: copy t[k/2] to t[k]
            {
                t |= (t>>(k/2)) & 1) << k;
            }
    }
    // correct t[0]:
    t ^= ((n+1)&1); // change low bit if n is even
    return t;
}
```

The routine involves \( n \) computations of the parity. The complexity of the equivalent routine for large \( n \) has complexity \( O(n^2) \) (\( n \) computations of sums with \( \sim n \) summands).
40.3.2 Computation via division of power series

The following variant of the algorithm, suggested by Richard Brent [priv.comm.], shows that the computation is equivalent to a division of power series. Let $R$ be the reciprocal polynomial of $C$, then (see [60, p.135])

$$\log (R(x)) = - \sum_{j=1}^{\infty} t_j x^j / j$$

(40.3-5)

Differentiating both sides gives

$$\frac{R'(x)}{R(x)} = - \sum_{j=1}^{\infty} t_j x^{j-1}$$

(40.3-6)

Using Newton’s method for the inversion we obtain a computational cost of $\gamma M(n)$ where $M(n)$ is the cost for the multiplication of two power series up to order $x^n$ and $\gamma$ is a constant (the method is also given in [127, p.24]). The constant $\gamma$ equals three if the division is performed by one inversion, which is two multiplications with the second order Newton iteration, and one final multiplication with $R'(x)$, see section 28.1.1 on page 569. For large $n$ the multiplications should be done by either one of the splitting schemes suggested in [50] or by FFT methods such as given in [238].

40.3.3 Some properties of the trace vector

For a binary polynomial $C$ of odd degree $n$ and all nonzero coefficients $c_i$ at odd indices $i$ we obtain $t_0 = 1$ and $t_i = 0$ for all $i \neq 0$, thereby the trace of any element is just the value of its lowest bit. In [48] it is shown that for $n \equiv \pm 3 \mod 8$ the first nonzero coefficient $c_k$ (with $k < n$) must appear at a position $k \geq n/3$.

With even degree and all nonzero odd coefficients $c_i$, $c_j$, $c_k$, ... at positions $i, j, k, \ldots < n/2$ the only nonzero components of the trace vector are $t_{n-i}$, $t_{n-j}$, $t_{n-k}$, .... Thereby polynomials of even degree with just one nonzero coefficient $c_k$ where $k < n/2$ lead to only $t_{n-k}$ being nonzero. A special case are trinomials $C = x^n + x^k + 1$ of even degree $n$ and $k < n/2$ ($k$ must be odd else $C$ is reducible). In the trace vector for all-ones polynomials ($C = \sum_{k=0}^{n} x^k$, see section 38.9.9 on page 850) the only zero component is $t_0$. A detailed discussion of the properties of the trace vector is given in [3].

40.4 Solving quadratic equations

We want to solve, in $\text{GF}(2^n)$, the equation

$$a x^2 + b x + c = 0$$

(40.4-1)

The fact that extracting a square root of an arbitrary element in $\text{GF}(2^n)$ is easy does not enable us to solve the given equation. The formula $r_{0,1} = \left(-b \pm \sqrt{b^2 - 4ac}\right) / (2a)$ that works fine for real and complex numbers is of no help here: how should we divide by two?

Instead we transform the equation into a special form: divide by $a$: $x^2 + \left(b/a\right) x + \left(c/a\right) = 0$, substitute $x = z \left(b/a\right)$ to get $z^2 \left(b/a\right)^2 + \left(b/a\right)^2 z + \left(c/a\right) = 0$, and divide by $\left(b/a\right)^2$ to obtain

$$z^2 + z + C = 0 \quad \text{where} \quad C = \frac{a c}{b^2}$$

(40.4-2)

If $r_0$ is one solution of this equation then $r_1 = r_0 + 1$ is the other one: $z (z + 1) = C$. The equation does not necessarily have a solution at all, the trace of $C$ must be zero because we have $\text{Tr}(C) = \text{Tr}(z^2 + z) = \text{Tr}(z^2) + \text{Tr}(z) = \text{Tr}(z) + \text{Tr}(z) = 0$ for all $z \in \text{GF}(2^n)$. 

[fxtbook draft of 2008-August-17]
Chapter 40: Binary finite fields: GF(2ⁿ)

n = 5  GF(2ⁿ)
c = 1..1.1 == x⁵ + x² + 1 (polynomial modulus)
mm= .11111 == 31 (prime) (maximal order)
h = .1... (aux. bitmask)
g = ...1. (element of maximal order)
tv= ..1..1 (traces of x^i)
tire= ......1 (element with trace=1)

k:  f:=g**k  Tr(f)  RootOf(z^2+z=f)
0:  ..1.  0    .1..1
1:  ...1  1    ...1
2:  .1.... 0    .1.11
3:  i.... 1    .1111
5:  .1..i  1    .11
9:  1.1.1  1    111i
11:  .1i..  1    i1i
13:  .1i1.  1    i1i1
15:  .i11.  1    i1i11
16:  i111. 0    i11i
17:  ...i1  1    ...i1
20:  1...i  1    1...i
21:  i...1  1    i...1
22:  1.i... 0    1....i
23:  .i...1  1    .i...1
24:  i....1 0    i....1
25:  1....1 0    1....1
26:  .1i... 0    .1i...1
27:  1.i... 0    1.i...1
28:  .i1... 0    .i1...1
29:  1...i  0    1...i
30:  i...1  0    i...1

Figure 40.4-A: Solutions of the equation z² + z = f for all elements f ∈ GF(2⁵) with trace zero.

--------- k=1: ---------
  u=t²*2 + t
  z=c*t²
  z²z=c²*t^4 + c*t²

--------- k=2: ---------
  u=t^4 + t² + t
  z=(c^2 + c)*t^4 + c*t²
  z²z=(c^4 + c²)*t^8 + c²*t⁴

--------- k=3: ---------
  u=t⁸ + t⁴ + t² + t
  z=(c^4 + c² + c)*t⁸ + (c^² + c)*t⁴ + c*t²
  z²z=(c^8 + c⁴ + c²)*t⁸ + c⁴*t⁸ + c*t⁴ + c²+t⁴

Figure 40.4-B: Solving the reduced quadratic equation z² + z = c in GF(2⁴).
The following function checks whether the reduced equation has a solution and if so, returns true and writes one solution to the variable \( r \):

```cpp
bool gf2n_solve_reduced_quadratic(GF2n c, GF2n& r)
// Solve \( z^2 + z = c \)
// Return whether solutions exist.
// If so, one solutions is written to r.
// The other solution is \( r+1 \).
{
  if ( 1==c.trace() ) return false;
  GF2n t( GF2n::tr1e );
  GF2n z( GF2n::zero );
  GF2n u( t );
  for (ulong j=1; j<GF2n::n_; ++j)
  {
    GF2n u2 = u.sqr();
    z = z.sqr(); z += u2*c;  // \( z = z^2 + c \)
    u = u2 + t;  // \( u = u^2 + t \)
  }
  r = z;
  return true;
}
```

Figure 40.4-A shows the solutions to the reduced equations \( x^2 + x = f \) for all elements \( f \) with trace zero [FXT: gf2n/gf2n-solvequadratic-demo.cc].

The implementation of the algorithm takes advantage of a precomputed element with trace one. At the end of step \( k \geq 1 \) we have

\[
\begin{align*}
  u_k & = \sum_{j=0}^{k} t^{2^j} \\
  z_k & = \sum_{j=0}^{k-1} \left[ t^{2^j} \sum_{i=0}^{k-1} c^{2^i} \right]
\end{align*}
\]

Figure 40.4-B shows (for \( GF(2^4) \)) that this expression is the solution sought.

For \( GF(2^n) \) with \( n \) odd the solution of the reduced quadratic equation \( z^2 + z = A \) can be computed via the half-trace of \( A \) which is defined as

\[
H(A) = A + A^4 + A^{16} + \ldots + A^{4^{(n-1)/2}}
\]

We have \( H(A)^2 + H(A) = \text{Tr}(A) + A \), so \( H(A) \) is a solution of the reduced quadratic if \( \text{Tr}(A) = 0 \). The half-trace of an element \( A \) in the field with field polynomial \( C \) can be computed via [FXT: gf2n_half_trace() in bpol/gf2n-trace.cc]:

```cpp
ulong gf2n_half_trace(ulong a, ulong c, ulong h)
{
  ulong t = a;
  ulong d = h;
  while ( d>>=2 )
  {
    t = bitpolmod_square(t, c, h);
    t = bitpolmod_square(t, c, h);
    t ^= a;
  }
  return t;
}
```

The function [FXT: gf2n_solve_quadratic() in bpol/gf2n-solvequadratic.cc] transforms the equation into the reduced form, solves it, and transforms back to obtain both solutions:

```cpp
bool gf2n_solve_quadratic(GF2n a, GF2n b, GF2n c, GF2n& r0, GF2n& r1)
// Solve \( a*x^2 + b*x + c = 0 \)
// Return whether solutions exist.
// If so, the solutions are written to r0 and r1.
```
Routines for the solution of quadratic equations that do not depend on the class GF2n are given in [FXT: bpol/bitpolmod-solvequadratic.cc].

40.5 Representation by matrices *

With a primitive polynomial modulus $C(x)$ a representation of the elements of GF($2^n$) as matrices can be obtained from the powers of the primitive element (‘$x$’) in a surprisingly simple way: in the table of powers of a generator take row $k$ through $k + n - 1$ to obtain the columns of matrices $M_k$, see figure 40.5-A. Now one has $M_k = M_1^k$ so one can use the matrices to represent the elements of GF($2^n$).

The matrix $M := M_1$ is the companion matrix of the polynomial modulus $g$. The companion matrix of a polynomial $p(x) = x^n - \sum_{i=0}^{n-1} c_i x^i$ of degree $n$ is defined as the $n \times n$ matrix

$$ M = \begin{bmatrix} 0 & 0 & \cdots & 0 & c_0 \\ 1 & 0 & \cdots & 0 & c_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & c_{n-2} \\ 0 & 0 & \cdots & 1 & c_{n-1} \end{bmatrix} \tag{40.5-1} $$

For polynomials $p(x) = \sum_{i=0}^{n} a_i x^i$ set $c_i := -a_i/a_n$.

The characteristic polynomial $c(x)$ of a $n \times n$ matrix $M$ is defined as

$$ c(x) := \det(x E_n - M) \tag{40.5-2} $$

---

---

---

---
where $E_n$ is the $n \times n$ unit matrix. The roots of characteristic polynomial are the eigenvalues of the matrix. The characteristic polynomial of the companion matrix of a polynomial $p(x)$ equals $p(x)$. If $p(x)$ is the characteristic polynomial of a matrix $M$ then $p(M) = 0$ (non-proof: set $x = M$ in relation 40.5-2 see \[84] for a proof).

<table>
<thead>
<tr>
<th>k</th>
<th>$[ p_k(x) ]^d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$[ 11 ]^4$</td>
</tr>
<tr>
<td>1</td>
<td>$[ 11 ]^1$</td>
</tr>
<tr>
<td>2</td>
<td>$[ 11 ]^1$</td>
</tr>
<tr>
<td>3</td>
<td>$[ 111 ]^1$</td>
</tr>
<tr>
<td>4</td>
<td>$[ 11 ]^1$</td>
</tr>
<tr>
<td>5</td>
<td>$[ 111 ]^1$</td>
</tr>
<tr>
<td>6</td>
<td>$[ 11 ]^1$</td>
</tr>
<tr>
<td>7</td>
<td>$[ 1..11 ]^1$</td>
</tr>
<tr>
<td>8</td>
<td>$[ 11..1 ]^1$</td>
</tr>
<tr>
<td>9</td>
<td>$[ 111 ]^1$</td>
</tr>
<tr>
<td>10</td>
<td>$[ 11111 ]^1$</td>
</tr>
<tr>
<td>11</td>
<td>$[ 1111 ]^1$</td>
</tr>
<tr>
<td>12</td>
<td>$[ 11111 ]^1$</td>
</tr>
<tr>
<td>13</td>
<td>$[ 11111 ]^1$</td>
</tr>
<tr>
<td>14</td>
<td>$[ 11111 ]^1$</td>
</tr>
</tbody>
</table>

**Figure 40.5-B:** Characteristic polynomials of the powers of the generator $x$ with the field GF(2$^4$) and the polynomial $x^4 + x + 1$.

Let $c_k(x)$ be the characteristic polynomial of the matrix $M_k = M^k$ and $p_k(x)$ the minimal polynomial of the element $g^k \in$ GF($2^n$). Then

$$c_k(x) = [p_k(x)]^d \text{ where } d = n/r$$

where $r$ is the smallest positive integer such that $M^r_k = M_k$. For example, for the primitive modulus $C(x) = x^4 + x + 1$ the sequence of characteristic polynomials of the powers of the generator ‘$x$’ are shown in figure 40.5-B

The trace of the matrix $M^k$ is the $d$-th power of the *polynomial trace* of the minimal polynomial of $g^k$. The polynomial trace of $p(x) = x^n - (c_{n-1}x^{n-1} + \cdots + c_1 x + c_0)$ equals $c_{n-1}$ as can be seen from relation 40.5-1.

By construction, picking the first column of $M_k$ gives the vector of the coefficients of the polynomial $x^k$ modulo $C(x)$:

$$M^k[1, 0, 0, \ldots, 0]^T \equiv x^k \text{ (mod } C(x))$$

Finally, the characteristic polynomial of an element $a \in$ GF($2^n$) in polynomial representation can be written as

$$p_a(x) := \prod_{k=0}^{n-1} \left(x - a^{2^k}\right)$$

Compare to relation 40.2-1 on page 891 for minimal polynomials.

### 40.6 Representation by normal bases

So far we used the $n$ basis vectors $x^0, x^1, x^2, x^3, \ldots, x^{n-1}$ to represent an element $a \in$ GF($2^n$) (as a vector space over GF(2)):

$$a = \sum_{k=0}^{n-1} a_k x^k$$

The arithmetic operations were the polynomial operations modulo an irreducible polynomial modulus $C$. 

[fxtbook draft of 2008-August-17]
For certain irreducible polynomials (which are called normal polynomials or $N$-polynomials) it is possible to use the normal basis $x^1, x^2, x^4, x^8, \ldots, x^{2^{n-1}}$ to represent elements of $\text{GF}(2^n)$:

$$a = \sum_{k=0}^{n-1} a_k x^{2^k}$$  \hspace{1cm} (40.6-2)

To check whether a polynomial $C$ is normal, compute $r_k = x^{2^k} \mod C$ for $1 \leq k \leq n$, compute the nullspace of the matrix $M$ whose $k$-th row is $r_k$. If the nullspace is empty (that is, $M \cdot v = 0$ implies $v = 0$) then the polynomial is normal.

The normality of a polynomial is equivalent to the fact that its roots are linearly independent (see section [38.5.1 on page 839 for the equivalence of computations modulo a polynomial and computations with linear combinations of its roots).

An element $f \in \text{GF}(2^n)$ where $f^1, f^2, f^4, f^8, \ldots, f^{2^{n-1}}$ are linearly independent is called a normal element (or free element). The minimal polynomial of a normal element $f$ is normal.

Addition and subtraction with a normal basis is again a simple XOR. Squaring an element can be achieved by a cyclic shift by one position. Note that $(x^{2^{n-1}})^2 = x^1$. Taking the square root is a cyclic shift in the other direction.

In normal basis representation the element one is the all-ones word. Thereby adding one is equivalent to complementing the binary word.

The trace can be computed easily with normal bases, it equals the parity of the binary word.

### 40.6.1 Multiplication and test for normality

Multiplication of two elements is achieved via a multiplication matrix $M$. Given two elements $a, b \in \text{GF}(2^n)$ in normal basis representation

$$a = \sum_{k=0}^{n-1} a_k x^{2^k}, \quad b = \sum_{k=0}^{n-1} b_k x^{2^k}$$  \hspace{1cm} (40.6-3)

their product $p = a \cdot b$ can be computed as follows: for the first component $p_0$ of the product we have

$$p_0 = a^T \cdot M \cdot b$$  \hspace{1cm} (40.6-4)

and in general

$$p_k = \left(a^{-2^{k-1}}\right)^T \cdot M \cdot b^{-2^{k-1}}$$  \hspace{1cm} (40.6-5)

That is, all components of the product are computed like the first, but with $a$ and $b$ cyclically shifted.

A routine that checks whether a given polynomial $c$ is normal and, if so, computes the multiplication matrix $M$, is [FXT: bitpol_normal_q()] in [bpol/bitpol-normal.cc] which proceeds as follows:

1. If the polynomial $c$ is reducible, return false.
2. Compute the matrix $A$ whose $k$-th row equals $x^{2^k} \mod c$. If $A$ is not invertible then (the nullspace is not empty and) $c$ is not normal, so return false.
3. Set $D := A \cdot C^T \cdot A^{-1}$ where $C$ is the companion matrix of $c$.
4. Compute the multiplication matrix $M$ where $M_{i,j} := D_{j',i'}$, $i' := -i \mod n$ and $j' := j - i \mod n$. Return (true and) the matrix $M$. 

[fxtbook draft of 2008-August-17]
40.6: Representation by normal bases

Table 40.6-A: Matrices that occur with the computation of the multiplication matrix for the field polynomials \( c = 1 + x + x^2 + x^3 + x^4 \) (left) and \( c = 1 + x + x^3 + x^4 + x^5 \) (right).

Examples of the intermediate results for two different field polynomials are given in figure 40.6-A.

A C++ function implementing the multiplication algorithm is [FXT: `bpol/normal-mult.cc`]:

```cpp
ulong normal_mult(ulong a, ulong b, const ulong *M, ulong n)
{
    ulong p = 0;
    for (ulong k=0; k<n; ++k)
    {
        ulong v = bitmat_mult_Mv(M, n, b);
        v = parity(v & a); // dot product
        p ^= (v << k);
        a = bit_rotate_right(a, 1, n);
        b = bit_rotate_right(b, 1, n);
    }
    return p;
}
```

We note that the algorithm is much more attractive for hardware implementations than for software, see [100].

### 40.6.2 Solving the reduced quadratic equation

The reduced quadratic equation \( x^2 + x = f \) has two solutions if \( \text{Tr}(f) = 0 \). If so, one solution \( x = [x_0, x_1, \ldots, x_{n-1}] \) can be obtained as \( x_k = \sum_{j=0}^{k} f_k \) where \( f = [f_0, f_1, \ldots, f_{n-1}] \). This follows from observing that

\[
x^2 + x = [x_0 + x_{n-1}, x_0 + x_1, x_1 + x_2, \ldots, x_{n-2} + x_{n-1}, x_{n-1} + x_0]
\]  

(40.6-6)

Now equate \( x^2 + x = f \) and set \( x_{n-1} = 0 \) (setting \( x_{n-1} = 1 \) gives the complement which is also a solution). In C++ this translates to (see section 1.13.5 on page 34) [FXT: `normal_solve_reduced_quadratic.cc`]:

```cpp
ulong normal_solve_reduced_quadratic(ulong c)
{
    // Solve \( x^2 + x = c \)
    // Must have: \( \text{trace}(c) = 0 \)
    return x;
    // that is, the complement of \( x \).
}
```
Normal poly: c=1111111 = x^5 + x^4 + x^3 + x^2 + 1

k = 0: f=g**k Tr(f) x^2+x==f
   0: 11111  1
   2: ...11  1
   3: ...11  1
   4: ...11  1
   5: ...11  x=.111
   6: ...11  x=111
   7: ...11  x=111
   9: ...11  x=111
  10: ...11  x=111
  11: ...11  x=111
  12: ...11  x=111
  13: ...11  x=111
  14: ...11  x=111
  15: ...11  x=111
  16: ...11  x=111
  17: ...11  x=111
  18: ...11  x=111
  19: ...11  x=111
  20: ...11  x=111
  21: ...11  x=111
  22: ...11  x=111
  23: ...11  x=111
  24: ...11  x=111
  25: ...11  x=111
  26: ...11  x=111
  27: ...11  x=111
  28: ...11  x=111
  29: ...11  x=111
  30: ...11  x=111

Figure 40.6-B: Solving the reduced quadratic equation $x^2 + x = f$ for powers $f = g^k$ of the generator $g = x$. The equation is solvable if the trace is zero, that is, the number of bits in the normal representation is even. The (primitive) field polynomial is $1 + x^2 + x^3 + x^4 + x^5$.

The highest bit of the result is zero exactly if the equation is solvable. The reversed Gray code is given in section 1.16.6 on page 17.

The program [FXT: gf2n/normalbasis-demo.cc] prints the powers of $x$ in normal basis representation, see figure 40.6-B. By default a primitive normal polynomial from [FXT: bpol/normal-primpoly.cc] is used.

40.6.3 The number of binary normal bases *

<table>
<thead>
<tr>
<th>$n$</th>
<th>$A_n$</th>
<th>$n$</th>
<th>$A_n$</th>
<th>$n$</th>
<th>$A_n$</th>
<th>$n$</th>
<th>$A_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>93</td>
<td>21</td>
<td>27783</td>
<td>31</td>
<td>28629151</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>128</td>
<td>22</td>
<td>95232</td>
<td>32</td>
<td>67108864</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>13</td>
<td>315</td>
<td>23</td>
<td>182183</td>
<td>33</td>
<td>97327197</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>14</td>
<td>448</td>
<td>24</td>
<td>262144</td>
<td>34</td>
<td>250675200</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>15</td>
<td>675</td>
<td>25</td>
<td>629145</td>
<td>35</td>
<td>352149525</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>16</td>
<td>2048</td>
<td>26</td>
<td>1290240</td>
<td>36</td>
<td>704643072</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>17</td>
<td>3825</td>
<td>27</td>
<td>1835001</td>
<td>37</td>
<td>1857283155</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>18</td>
<td>13797</td>
<td>28</td>
<td>3670016</td>
<td>38</td>
<td>3616800768</td>
</tr>
<tr>
<td>9</td>
<td>21</td>
<td>19</td>
<td>13797</td>
<td>29</td>
<td>9256395</td>
<td>39</td>
<td>5282242875</td>
</tr>
<tr>
<td>10</td>
<td>48</td>
<td>20</td>
<td>24576</td>
<td>30</td>
<td>11059200</td>
<td>40</td>
<td>12884901888</td>
</tr>
</tbody>
</table>

Figure 40.6-C: The number $A_n$ of degree-$n$ binary normal polynomials up to $n = 40$.

The number $A_n$ of degree-$n$ binary normal polynomials up to $n = 40$ is given in figure 40.6-C. A table of
40.6: Representation by normal bases

<table>
<thead>
<tr>
<th>$n$</th>
<th>$B_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>87</td>
</tr>
<tr>
<td>12</td>
<td>52</td>
</tr>
<tr>
<td>13</td>
<td>315</td>
</tr>
<tr>
<td>14</td>
<td>291</td>
</tr>
<tr>
<td>15</td>
<td>562</td>
</tr>
<tr>
<td>16</td>
<td>1017</td>
</tr>
<tr>
<td>17</td>
<td>3825</td>
</tr>
<tr>
<td>18</td>
<td>2870</td>
</tr>
<tr>
<td>19</td>
<td>13797</td>
</tr>
<tr>
<td>20</td>
<td>11255</td>
</tr>
<tr>
<td>21</td>
<td>23579</td>
</tr>
<tr>
<td>22</td>
<td>178259</td>
</tr>
<tr>
<td>23</td>
<td>78899078</td>
</tr>
</tbody>
</table>

Figure 40.6-D: The number $B_n$ of degree-$n$ binary primitive normal polynomials up to $n = 33$.

the values $A_n$ for $1 \leq n \leq 130$ and their factorizations is given in [FXT: data/num-normalpoly.txt]. The sequence $A_n$ is entry A027362 of [245]. The number $B_n$ of degree-$n$ binary primitive normal polynomials up to $n = 33$ is given in figure 40.6-D. This is sequence A107222 of [245].

40.6.3.1 Computation via exhaustive search

For small degrees all normal polynomials can be generated by selecting from the irreducible polynomials those that are normal. Using the mechanism that generates all irreducible polynomials via Lyndon words that is described in section 38.10 on page 852, the computation is a matter of minutes for $n < 25$. The program [FXT: gf2n/all-normalpoly-demo.cc] prints all normal polynomials of a given degree $n$, its output for $n = 9$ is shown in figure 40.6-E. We can compute the number of normal ($A_n$) and primitive normal ($B_n$) binary polynomials for small degrees $n$ using that program. The table of the values $B_n$ in figure 40.6-D was produced with the mentioned program, the computation up to $n = 30$ takes about 90 minutes. As noted in [133], no formula for the number of primitive normal polynomials is presently known. The proof that primitive normal bases exist for all finite fields is given in [187].

40.6.3.2 Cycles in the De Bruijn graph

Quite surprisingly, it turns out that $A_n$ equals the number of cycles in the De Bruijn graph (see section 19.2.2 on page 389 and section 39.5 on page 870). Thereby for $n$ a power of two the number $A_n$ equals the number of binary De Bruijn sequences of length $2n$. No isomorphism between both objects (paths and polynomials) is presently known.
40.6.3.3 Invertible circulant matrices

```
arg 1: 6 == n  [ n x n - matrices] default=6
arg 2: 2 == wh [What to do: 0==>just count 1==>print words
                  2==>also print matrix] default=2

v0 = 1..... [I]  v0 = 1,11.. [I]
M = M = M = M =
 1..... 1,11..
 1..... 1,111
   1    1,11
   1    11,1
   1    11,1
v0 = 111... [S] v0 = 11111. [I] v0 = 11.1.. [I]
M = M = M = M =
 111... 11111
 111... 11111
   1    111
   1    111
   1    111
n=6   #invertible=4 #singular=1
```

Figure 40.6-F: The length-6 Lyndon words of odd weight and the corresponding circulant matrices. Singular matrices are marked with ‘[S]’, invertible matrices with ‘[I]’.

The number $A_n$ of binary normal bases also equals the number of invertible circulant $n \times n$ matrices over GF(2). This is demonstrated in [FXT: gf2n/bitmat-circulant-demo.cc] whose output for $n = 6$ is shown in figure 40.6-F. The search uses the Lyndon words as periodic words would trivially lead to singular matrices. Further, Lyndon words with an even number of bits can be skipped as the vector $[1, 1, 1, \ldots, 1]$ is in the nullspace of the corresponding matrices.

If the set $\{\alpha, \alpha^2, \alpha^4, \alpha^8, \ldots, \alpha^{2^{n-1}}\}$ is a normal basis of GF(2) we say that $\alpha$ generates the normal basis. Considering the rows of a circulant matrix as some element $\beta$ in a normal basis representation then the following rows are $\beta^2, \beta^4, \beta^8, \ldots, \beta^{2^{n-2}}$ and the matrix is invertible if $\beta$ generates a normal basis. If $\alpha$ generates a normal basis then an element $\beta = \sum_{i=0}^{n-1} a_i \alpha^i$ generates a normal basis exactly if the polynomial $\sum_{i=0}^{n-1} a_i x^i$ is relatively prime to $x^n - 1$. Thereby, with a fast algorithm to generate Lyndon words, determine all elements that generate normal bases if one such element is known as follows: select the Lyndon words with an odd number of ones and test whether gcd($L(x), x^n - 1$) = 1 where $L(x)$ is the binary polynomial corresponding to the Lyndon word. If $n$ is a power of two, then $x^n - 1 = (x - 1)^n$ and all Lyndon words with an odd number of ones are coprime to $x^n - 1$.

```
L(x) = 111.11.  W(x) = 1.....11   L(x) = 1.1..1.  W(x) = 1..1111
M = M = M = M =
 1111. 11111
 11111 11111
 1111. 11111
 1111. 11111
n=7   #invertible=4 #singular=1
```

Figure 40.6-G: The inverse of a $n \times n$ circulant matrix over GF(2) can be found by computing the inverse $W(x)$ of its first row as a polynomial $L(x)$ modulo $x^n - 1$.

If the Lyndon word under consideration is taken as a polynomial $L(x)$ over GF(2) then the corresponding matrix is invertible exactly if gcd($L(x), x^n - 1$) = 1. The first row inverse of a circulant matrix over GF(2) can be found by computing $W(x) = L(x)^{-1} \mod x^n - 1$ where $L(x)$ is the binary polynomial with coefficients one where the Lyndon word has a one. As the inverse of a circulant matrix is also circulant, the remaining rows are cyclic shifts of $W(x)$. Two examples with $n = 7$ are shown in figure 40.6-G.

The equality of the number of invertible circulants and normal bases can also be seen as follows: choose a normal basis and test for each element $f$ whether the set $f^1, f^2, f^4, f^8, \ldots, f^{2^{n-1}}$ is linearly independent. As squaring is a cyclic shift the matrices to be tested are the circulants we considered.
40.6.3.4 Factorization of $x^n - 1$

The factorization of the polynomial $x^n - 1$ over GF(2) can be used for the computation of $A_n$. The file [FXT: data/polfactdeg.txt] supplies the necessary information:

```
# Structure of the factorization of x^n-1 over GF(2):
1: [1] [1*1]
2: [2] [1*1]
3: [1] [1*1 + 1*2]
4: [4] [1*1]
5: [1] [1*1 + 1*4]
6: [2] [1*1 + 1*2]
7: [1] [1*1 + 2*3]
8: [8] [1*1]
9: [1] [1*1 + 1*2 + 1*6]
10: [2] [1*1 + 1*4]
11: [1] [1*1 + 1*10]
12: [4] [1*1 + 1*2]
13: [1] [1*1 + 1*12]
14: [2] [1*1 + 2*3]
15: [1] [1*1 + 1*2 + 3*4]
16: [16] [1*1]
17: [1] [1*1 + 2*8]
```

An entry: $n$: [e] [m1*d1 + m2*d2 + ... ] says that $(x^n - 1) = P(x)^e$ and $P(x)$ factors into $m_1$ different irreducible polynomials of degree $d_1$, $m_2$ different irreducible polynomials of degree $d_2$ and so on. As an example, for $n = 6$ we have

$$x^6 - 1 = [x^3 - 1]^2 = [(x + 1)(x^2 + x + 1)]^2$$ (40.6-7)

$x^6$ is the square ($e = 2$) of a product of one irreducible polynomial of degree one and one of degree two. Therefore we have the entry: 6: [2] [1*1 + 1*2]. Another example, $n = 15$,

$$x^{15} - 1 = [(x + 1)(x^2 + x + 1)(x^4 + x + 1)(x^4 + x^3 + 1)(x^4 + x^3 + x^2 + x + 1)]^1$$ (40.6-8)

corresponding to the entry 15: [1] [1*1 + 1*2 + 3*4].

Now one has for the number of normal polynomials:

$$A_n = \frac{2^n}{n} \prod_{d|n} \left(1 - \frac{1}{2^d}\right)^{m_i}$$ (40.6-9)

Note that the quantity $e$ does not appear in the formula. For example, with $n = 6$ and $n = 15$ we obtain

$$A_6 = \frac{2^6}{6} \cdot \left(1 - \frac{1}{2^1}\right)^1 \cdot \left(1 - \frac{1}{2^2}\right)^1 = \frac{64}{6} \cdot \frac{1}{2} \cdot \frac{3}{4} = 4$$ (40.6-10a)

$$A_{15} = \frac{2^{15}}{15} \cdot \left(1 - \frac{1}{2^1}\right)^1 \cdot \left(1 - \frac{1}{2^2}\right)^1 \cdot \left(1 - \frac{1}{2^4}\right)^3 = 675$$ (40.6-10b)

40.6.3.5 Efficient computation

It is actually possible to compute the number of degree-$n$ normal binary polynomials without explicitly factorizing $x^n - 1$. We have

$$x^n - 1 = \prod_{d|n} Y_d(x)$$ (40.6-11)

where $Y_d(x)$ is the $d$-th cyclotomic polynomial (see section 38.11 on page 854). We further know that $Y_d(x)$ factors into $\varphi(d)/r$ polynomials of degree $r$ where $r = \text{ord}_d(2)$ is the order of 2 modulo $d$. Let $a_n := A_n / (2^n/n)$, then $a_n$ can for odd $n$ be computed as

$$a_n = \prod_{d|n} \left(1 - \frac{1}{2^r}\right)^{\varphi(d)/r}$$ (40.6-12)
The following pari/gp code works all odd \( n \):

```gp
1 p=2 /* global */
2 num_normal_p(n)=
3 { local( r, i, pp );
4 pp = 1;
5 fordiv (n, d,
6 r = znorder(Mod(p,d));
7 i = eulerphi(d)/r;
8 pp *= (1 - 1/p^r)^i;
9 );
10 return( pp );
11 }
```

The number \( A_n \) can be computed (for arbitrary \( n \) as \( A_n = a_q \left( \frac{2^n}{n} \right) \) where \( q \) odd, and \( n = q^{2^t} \):

```gp
1 num_normal(n)=
2 { local( t, q, pp );
3 t = 1; q = n;
4 while ( 0==(q%p), q/=p; t+=1; );
5 /* here: n==q*p^t */
6 pp = num_normal_p(q);
7 pp *= p^n/n;
8 return( pp );
9 }
```

The quantity \( t \) is not used in the computation. The implementation is quite efficient: the computation of \( A_n \) for all \( n \leq 10,000 \) takes less than three seconds. The computation of \( A_n \) for \( n = 1234567 = 127 \cdot 9721 \) (\( A_n \) is a number with 371,636 decimal digits) takes about 200 milliseconds.

### 40.6.4 Dual normal bases

Let \( A = \{a_0, a_1, a_2, \cdots, a_{n-1}\} \) be a basis of \( \text{GF}(2^n) \). A basis \( B = \{b_0, b_1, b_2, \cdots, b_{n-1}\} \) is said to be the dual basis (or complementary basis, or trace-orthonormal basis) of \( A \) if

\[
\text{Tr}(a_k b_j) = \delta_{k,j} \quad \text{for} \quad 0 \leq k, j < n \quad (40.6-13)
\]

A basis that is its own dual is called self-dual. We consider only normal basis here. If \( \alpha \) is a root of a normal polynomial \( C \) then \( A = \{\alpha, \alpha^2, \alpha^4, \cdots, \alpha^{2^n-1}\} \) is a normal basis.

<table>
<thead>
<tr>
<th>( C )</th>
<th>( T )</th>
<th>( C^* )</th>
<th>( D = T^{-1} \pmod{x^n-1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 )</td>
<td>.11...11</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 2 )</td>
<td>.11...11</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 3 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 4 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 5 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 6 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 7 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 8 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 9 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 10 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 11 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 12 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 13 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 14 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 15 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 16 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 17 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 18 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 19 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 20 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
<tr>
<td>( 21 )</td>
<td>.1111...111</td>
<td>.1111...111</td>
<td>( .11111.11 )</td>
</tr>
</tbody>
</table>

**Figure 40.6-H:** All normal polynomials \( C \) of degree 9 and their polynomials \( T \) (left), their duals \( C^* \) and \( D = T^{-1} \pmod{x^n-1} \) (right). Primitive polynomials are marked with ‘P’, self-dual \( C = C^* \) are marked with ‘S’.
A necessary condition for $C$ to be normal is that \( \gcd(T, x^n - 1) = 1 \) where

$$T = t_0 + t_1 x + t_2 x^2 + \ldots + t_{n-1} x^{n-1} \quad (40.6-14)$$

with $t_k = \text{Tr}(\alpha \alpha^k)$. The polynomial $T$ can be computed via [FXT: bpol/normalpoly-dual.cc]

Now if $C$ is normal then $T$ has an inverse $D \equiv T^{-1} \mod x^n - 1$. Write

$$D = d_0 + d_1 x + d_2 x^2 + \ldots + d_{n-1} x^{n-1} \quad (40.6-15a)$$

then

$$\beta = d_0 \alpha + d_1 \alpha^2 + d_2 \alpha^4 + \ldots + d_{n-1} \alpha^{2^{n-1}} \quad (40.6-15b)$$

is a root of a normal polynomial and $B = \{\beta, \beta^2, \beta^4, \ldots, \beta^{2^{n-1}}\}$ is the dual (normal) basis of $A$. The following routine computes $T$, $D$, $B$, and finally the minimal polynomial $C^*$ of $\beta$:

Figure 40.6-H shows the normal bases of degree 9 and their duals, it was created with the program [FXT: gf2n/normalpoly-dual-demo.cc].
Chapter 40: Binary finite fields: GF(2^n)

40.6.5 The number of self-dual normal basis

<table>
<thead>
<tr>
<th>n</th>
<th>S_n</th>
<th>n</th>
<th>S_n</th>
<th>n</th>
<th>S_n</th>
<th>n</th>
<th>S_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1</td>
<td>9:</td>
<td>3</td>
<td>17:</td>
<td>17</td>
<td>25:</td>
<td>205</td>
</tr>
<tr>
<td>2:</td>
<td>1</td>
<td>10:</td>
<td>4</td>
<td>18:</td>
<td>48</td>
<td>26:</td>
<td>320</td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
<td>11:</td>
<td>3</td>
<td>19:</td>
<td>27</td>
<td>27:</td>
<td>513</td>
</tr>
<tr>
<td>4:</td>
<td>0</td>
<td>12:</td>
<td>0</td>
<td>20:</td>
<td>0</td>
<td>28:</td>
<td>0</td>
</tr>
<tr>
<td>5:</td>
<td>1</td>
<td>13:</td>
<td>5</td>
<td>21:</td>
<td>63</td>
<td>29:</td>
<td>565</td>
</tr>
<tr>
<td>6:</td>
<td>2</td>
<td>14:</td>
<td>8</td>
<td>22:</td>
<td>96</td>
<td>30:</td>
<td>1920</td>
</tr>
<tr>
<td>7:</td>
<td>1</td>
<td>15:</td>
<td>15</td>
<td>23:</td>
<td>89</td>
<td>31:</td>
<td>961</td>
</tr>
<tr>
<td>8:</td>
<td>0</td>
<td>16:</td>
<td>0</td>
<td>24:</td>
<td>0</td>
<td>32:</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>Z_n</th>
<th>n</th>
<th>Z_n</th>
<th>n</th>
<th>Z_n</th>
<th>n</th>
<th>Z_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>0</td>
<td>9:</td>
<td>2</td>
<td>17:</td>
<td>17</td>
<td>25:</td>
<td>200</td>
</tr>
<tr>
<td>2:</td>
<td>1</td>
<td>10:</td>
<td>3</td>
<td>18:</td>
<td>25</td>
<td>26:</td>
<td>215</td>
</tr>
<tr>
<td>3:</td>
<td>1</td>
<td>11:</td>
<td>3</td>
<td>19:</td>
<td>27</td>
<td>27:</td>
<td>428</td>
</tr>
<tr>
<td>4:</td>
<td>0</td>
<td>12:</td>
<td>0</td>
<td>20:</td>
<td>0</td>
<td>28:</td>
<td>0</td>
</tr>
<tr>
<td>5:</td>
<td>1</td>
<td>13:</td>
<td>5</td>
<td>21:</td>
<td>57</td>
<td>29:</td>
<td>562</td>
</tr>
<tr>
<td>6:</td>
<td>1</td>
<td>14:</td>
<td>4</td>
<td>22:</td>
<td>60</td>
<td>30:</td>
<td>997</td>
</tr>
<tr>
<td>7:</td>
<td>1</td>
<td>15:</td>
<td>11</td>
<td>23:</td>
<td>87</td>
<td>31:</td>
<td>961</td>
</tr>
<tr>
<td>8:</td>
<td>0</td>
<td>16:</td>
<td>0</td>
<td>24:</td>
<td>0</td>
<td>32:</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 40.6-I: Number of self-dual normal basis (S_n, top) and self-dual normal basis where the field polynomial is primitive (Z_n, bottom).

Figure 40.6-I gives the number S_n of self-dual normal basis (top), and the number Z_n of such basis where the field polynomial is primitive (bottom). The field polynomial is the minimal polynomial of any of the basis elements. The sequence of values S_n is entry A135488 of [245], the values Z_n are entry A135498. No formula for the numbers Z_n is known, the values were computed with the program [FXT: gf2n/normalpoly-dual-demo.cc]. An expression for S_n is given in [158, theorem 5]. The following routine for computing the values S_n (for p = 2) is given by Max Alekseyev [priv.comm.]:

```cpp
1  sdn(m,p) =
2  \ Number of distinct self-dual normal bases of GF(p^m) over GF(p) where p is prime
3  { local(F, f, g, s, c, d);
4    if ( p==2 && m%4==0, return(0) );
5    if ( !(m%p), /* p divides m */
6        s = m\p;
7        return( p^((p-1)*(s+(s*(p+1))%2)/2-1) * sdn(s,p) );
8    , /* else */
9      F = factormod( (x^m - 1)/(x - 1), p );
10     c = d = []; for (i=1, matsize(F)[1],
11          f = lift(F[i,1]);
12          if ( f==g, c = concat( c, vector(F[i,2],j,poldegree(f)/2) ));
13          if ( lex(Vec(f), Vec(g))=1 ,
14              d = concat( d, vector(F[i,2],j,poldegree(f)) ));
15        );
16    return( 2^(p%2) * prod(i=1,\c, p^c[i] + 1) * prod(j=1,\d, p^d[j] - 1) / m );
23  }
```

We note that duality is defined for any basis, but no self-dual polynomial basis exists. See [129] and [158] for more information.
40.7 Conversion between normal and polynomial representation

If the field polynomial $C$ is normal then conversion between the representations in polynomial and normal basis can be achieved as follows: Let $Z$ be the $n \times n$ matrix whose $k$-th column equals $x^k \mod C$ where $n$ is the degree of $C$. If $a$ is the polynomial representation then the normal representation is $b = Z^{-1} \cdot a$.

**Figure 40.7-A:** Conversion between normal- and polynomial representation with the (primitive) polynomial $c = 1 + x^2 + x^3 + x^4 + x^5$. The conversion matrices are given as $P2N = Z^{-1}$ and $N2P = Z$.

The implementation [FXT: class GF2n in bpol/gf2n.h] allows the conversion to the normal representation if the field polynomial is normal. In the initializer [FXT: GF2n::init() in bpol/gf2n.cc] the matrices $Z(n2p_tab[])$ and $Z^{-1}(p2n_tab[])$ are computed with the lines

```cpp
1 // conversion to and from normal representation:
2 for (ulong k=0, s=2; k<n_; ++k)
3 {
4     n2p_tab[k] = s;
5     s = bitpolmod_square(s, c_, h_);
6 }
7 bitmat_transpose(n2p_tab, n_, n2p_tab);
8 is_normal_ = bitmat_inverse(n2p_tab, n_, p2n_tab);
```

The last line records whether the field polynomial is normal which is the case exactly if $Z$ is invertible.

The functions [FXT: bpol/gf2n.cc]

```cpp
1 ulong // static
2 GF2n::p2n(ulong f)
3 { return bitmat_mult_Mv(p2n_tab, n_, f); }
4 ulong // static
5 GF2n::n2p(ulong f)
6 { return bitmat_mult_Mv(n2p_tab, n_, f); }
```

allow conversions between the normal and polynomial representations. The method

```cpp
ulong get_normal() const { return p2n(v_); }
```

provides a convenient way to obtain the normal representation of a given element.

[fxtbook draft of 2008-August-17]
This is demonstrated in [FXT: \texttt{gf2n/gf2n-normal-demo.cc}] where both the polynomial and the normal representation are given, see figure 40.7-A.

If the last argument of the initialization routine of the C++ class \texttt{GF2n}, \texttt{init(n, c, normalq)}, is set then a (primitive) normal polynomial will be used as field polynomial. A list of primitive normal polynomials is given in [FXT: \texttt{bpol/normal-primpoly.cc}].

### 40.8 Optimal normal bases (ONB)

The number of nonzero terms in the multiplication matrix determine the complexity (operation count) for the multiplication with normal bases. It turns out that for certain values of \( n \) there are normal bases of \( \text{GF}(2^n) \) whose multiplication matrices have at most two nonzero entries in each row (and column). Such bases are called \textit{optimal normal bases} (ONB).

Optimal normal bases are especially interesting for hardware implementations because of both the highly regular structure of the multiplication algorithm and the minimal complexity with ONBs.

#### 40.8.1 Type-1 optimal normal bases

A so-called \textit{type-1 optimal normal basis} exists for \( n \) when \( p:=n+1 \) is prime and 2 is a primitive root modulo \( p \) (and for \( n = 0 \) and \( n = 1 \)). The sequence of such \( n \) starts

\[
0, 1, 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82, 100, 106, 130, 138, 148, 162, 172, 178, 196, 210, 226, 242, 268, 292, 316, 346, 348, 372, 378, 388, 418, 420, 442, 460, 466, 490, 508, \ldots
\]

This is entry [A071642] of [215]. One has always \( n \equiv 2 \) or \( n \equiv 4 \) modulo 8. A list of the corresponding primes is given in figure 39.7-B on page 875. The field polynomial corresponding to a type-1 ONB is the all-ones polynomial

\[
c = \frac{x^p - 1}{x - 1} = 1 + x + x^2 + x^3 + \ldots + x^n \quad (40.8-1)
\]

The order of these polynomials is \( n + 1 \) (for \( n > 1 \)) so they are non-primitive for all \( n \geq 3 \).

<table>
<thead>
<tr>
<th>Normal poly:</th>
<th>( c = 111111111111 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A ):</td>
<td>111111111111</td>
</tr>
<tr>
<td>( A^{-1} ):</td>
<td>11111111111111</td>
</tr>
<tr>
<td>( C^T ):</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>( D=A<em>C^T</em>A^{-1} ):</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>Mult. matrix M:</td>
<td>1111111111111111</td>
</tr>
</tbody>
</table>

\[\text{Figure 40.8-A: Matrices that occur with the computation of the multiplication matrix for the field polynomial } c = 1 + x + \ldots + x^{10}\.

The multiplication matrices are sparse: there is exactly one entry in the first row and column and exactly two entries in the other rows and columns. That is, the multiplication matrices for \( \text{GF}(n) \) with optimal normal basis have exactly \( 2n - 1 \) nonzero entries. For example, with \( n = 10 \) we obtain the matrix shown (together with the intermediate results) in figure 40.8-A. The equivalent data for \( n = 4 \) is shown in figure 40.6-A on page 901. 

[fxtbook draft of 2008-August-17]
40.8.2 Type-2 optimal normal bases

A type-2 optimal normal basis exists for \( n \) if \( p := 2^n + 1 \) is prime and either

- \( n \equiv 1 \) or \( n \equiv 2 \) modulo 4 and the order of 2 modulo \( p \) equals \( 2^n \).
- \( n \equiv 3 \) mod 4 and the order of 2 modulo \( p \) equals \( n \).

A type-2 basis exists for the following \( n \leq 200 \):

\[
1, 2, 3, 5, 6, 9, 11, 14, 18, 23, 26, 29, 30, 33, 35, 39, 41, 50, 51, 53, 65, 69, 74, 81, 83, 86, 89, 90, 95, 98, 99, 105, 113, 119, 131, 134, 135, 146, 155, 169, 173, 174, 179, 183, 186, 189, 191, 194
\]

The sequence is entry A054639 of [244]. The corresponding normal polynomials \( p_n \) of degree \( n \) can be computed via the recurrence

\[
p_0 := 1, \quad p_1 := x + 1 \quad (40.8-2a)
\]
\[
p_k := x p_{k-1} + p_{k-2} \quad (40.8-2b)
\]

Compare to the recursion that transforms a linear hybrid cellular automaton into a binary polynomial relation [39.8-1] on page 877: the type-2 ONBs correspond to the most trivial LHCA defined by the rule having a single one as the lowest bit of the rule word.

Expressions for the polynomials \( p_n \) are

\[
p_n(x) = \sum_{j=0}^{n} \left( n - \left\lfloor (j + 1)/2 \right\rfloor \right) x^{n-j} = \sum_{j=0}^{n} \left( \left\lfloor n/2 + j/2 \right\rfloor \right) x^j \quad (40.8-3)
\]
Chapter 40: Binary finite fields: $\text{GF}(2^n)$

$p_7 = 1 \cdot x^7 + 1 \cdot x^6 + 6 \cdot x^5 + 5 \cdot x^4 + 10 \cdot x^3 + 6 \cdot x^2 + 4 \cdot x + 1$

<table>
<thead>
<tr>
<th>n \ k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>15</td>
<td>20</td>
<td>15</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>7</td>
<td>21</td>
<td>35</td>
<td>35</td>
<td>21</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8</td>
<td>28</td>
<td>56</td>
<td>70</td>
<td>56</td>
<td>28</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 40.8-C: Locations (starred entries) of the coefficients of the polynomial $p_7$ in Pascal’s triangle.

The locations of the coefficients of the polynomials $p_n$ in Pascal’s triangle (see figure 6.0-A on page 175) lie on a rising diagonal. For $p_7(x) = 1 \cdot x^7 + 1 \cdot x^6 + 6 \cdot x^5 + 5 \cdot x^4 + 10 \cdot x^3 + 6 \cdot x^2 + 4 \cdot x + 1$ they are shown in figure 40.8-C. The following relations hold over $\text{GF}(2)$ (but not over $\mathbb{Z}$):

$$p_n(x) = \sum_{j=0}^{n} \binom{2n-j}{j} x^{n-j} = \sum_{j=0}^{n} \binom{n+j}{n-j} x^j$$

(40.8-4)

The binomial coefficient $\binom{n}{k}$ modulo 2 equals one if the binary expansion of $k$ is a subset of the expansion of $n$. Using the trick from section 1.10.1 on page 24 we obtain a fast method for the computation of the polynomials $p_n$ (using the first equality in 40.8-4):

```plaintext
1 t2poly(n) = sum(j=0,n, (bitand(2*n-j, j)==j)*x^(n-j));
```

The value of binomial coefficients modulo a prime $p$ can be computed via the relation

$$\binom{n}{k} \equiv \prod_{j=0}^{\infty} \left( \binom{n}{j} \binom{k}{j} \right) \mod p$$

(40.8-5)

where $n = \sum n_j p^j$ and $k = \sum k_j p^j$ are the radix-$p$ expansions (see the entry “Lucas Correspondence Theorem” in [273], and also [117]). Moreover, the highest power of $p$ that divides $\binom{n}{k}$ equals the number of carries when subtracting $k$ from $n$ in base $p$, see [203]. Especially if $k_j > n_j$ for any $j$ then $\binom{n}{k} \equiv 0 \mod p$.

The computation above is obtained by setting $p = 2$.

We note a relation of the polynomials $p_k$ to the Fibonacci polynomials defined by

$$f_0 := 0, \quad f_1 := 1$$

(40.8-6a)

$$f_k := x f_{k-1} + f_{k-2}$$

(40.8-6b)

We have

$$p_k^2 = f_{2k+1}$$

(40.8-7)

As with type-1 ONBs, the multiplication matrices are sparse. The polynomials and multiplication matrices for $n = 6$ and $n = 9$ are

1 6 5 4 1 0 9 8 6 5 4 1 0

\ldots 1 1 1 \ldots 1 1 1 \ldots

\ldots 1 1 1 \ldots 1 1 1 \ldots

\ldots 1 1 1 \ldots 1 1 1 \ldots

\ldots 1 1 1 \ldots 1 1 1 \ldots

The intermediate values with the computation of the multiplication matrix for $n = 5$ are shown in figure 40.6-A on page 901.

The sequence of values $n$ such that a optimal normal basis (either type-1 or type-2) over $\text{GF}(2^n)$ exists is entry A136250 of [245]. The values up to 100 are

1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 14, 18, 23, 26, 28, 29, 30, 33, 35, 36, 39, 41, 50, 51, 52, 53, 58, 60, 65, 66, 69, 74, 81, 82, 83, 86, 89, 90, 95, 98, 99, 100
40.9 Gaussian normal bases

The type-$t$ Gaussian normal basis (GNB) generalize the optimal normal basis. The type-1 and type-2 GNBs are the corresponding ONBs. The multiplication matrices for type-$t$ GNBs for $t > 2$ have more nonzero entries than the ONBs.

A type-$t$ GNB exists for $n$ when $p := tn + 1$ is prime and $\operatorname{gcd}(n, \frac{tn}{r_2}) = 1$ where $r_2$ is the order of 2 modulo $p$. For $n$ divisible by 8 no GNB exist. Figure 40.9-A shows, for $t \leq 10$, the first values $n$ such that a type-$t$ GNB exists. The sequences for $1 \leq t \leq 7$ are the following entries of [245]: A071642 (type-1), A054639 (type-2), A136415 (type-3), A137310 (type-4), A137311 (type-5), A137313 (type-6), A137314 (type-7), and A101284 (type-8). We implement the test using pari/gp:

``` pari/gp
1 gauss_test(n, t)=
2 { /* test whether a type-t Gaussian normal basis exists for GF(2^n) */
3   local( p, r2, F, w, x, nh, m, ir, ic);
4   p = t*n + 1;
5   if ( isprime(p), return( 0 ) );
6   if ( p<2, return( 0 ) );
7   r2 = znorder( Mod(2, p) );
8   d = (t*n)/r2;
9   g = gcd(d, n);
10  return ( if ( 1==g, 1, 0) );
}
```

40.9.1 Computation of the multiplication matrix

An algorithm that computes the multiplication matrix for a type-$t$ GNB proceeds as follows (The algorithm uses a vector $F[1, 2, \ldots, p – 1]$):

1. Set $p = tn + 1$ (this is a prime), and compute an element $r$ of order $t$ modulo $p$.
2. For $k = 0, 1, \ldots, t – 1$ do the following: set $j = r^k$ and for $i = 0, 1, \ldots, n – 1$ set $F[j2^i] = i$.
3. Set the multiplication matrix $M$ to zero.
4. For $i = 1, 2, \ldots, p – 2$ add one to $M_F[p-i,F_i+1]$.
5. If $t$ is odd set $h = n/2$ and do the following: for $i = 0, 1, \ldots, h – 1$ increment $M_{i,h+i}$ and $M_{h+i,i}$.

Implementation in pari/gp:

``` pari/gp
1 gauss_nb(n, t)=
2 { /* return multiplier matrix for type-t Gaussian normal basis */
3   /* returned matrix is over Z and has to be multiplied by Mod(1,2) */
4   local(p, r, F, w, x, nh, m, ir, ic);
5   p = t*n + 1;
6   r = znprimroot(p); \ r = r^*(n); /* r has order t */
```
Chapter 40: Binary finite fields: GF(2^n)

8 F = vector(p-1);
9 w = Mod(1, p);
10 for (i=0, t-1,
11 j = lift(w);
12 for (i=0, n-1,
13 F[j] = i;
14 j+=j; if (j>=p, j-=p); /* 2*j mod p */
15 );
16 w *= r;
17 );
18 m = matrix(n, n);
19 for (i=1, p-2,
20 ir = F[p-i]; ic = F[i+1];
21 m[ ir+1, ic+1 ] += 1;
22 );
23 if ( 1==(t%2),
24 nh = n/2; /* odd t ==> even n */
25 for (i=0, nh-1,
26 ir = i; ic = nh + i;
27 ir += 1; ic += 1;
28 m[ir, ic] += 1;
29 m[ic, ir] += 1;
30 );
31 );
32 return ( m );
33 }

```
n=7, t=4                   n=12, t=3

<table>
<thead>
<tr>
<th>M</th>
<th>M mod 2=</th>
</tr>
</thead>
<tbody>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>M mod 2=</th>
</tr>
</thead>
<tbody>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
<tr>
<td>1...1</td>
<td>1...1</td>
</tr>
</tbody>
</table>
```

Figure 40.9-B: Multiplication matrices over $\mathbb{Z}$ and GF(2) for Gaussian normal bases with $n = 7$, $t = 4$ (left), and $n = 12$, $t = 3$ (right). Dots denote zeros.

The implementation computes $M$ with entries in $\mathbb{Z}$ and has to be reduced modulo 2 before usage. Figure 40.9-B gives two examples.

The file [FXT: data/gauss-normalbasis.txt](FXT: data/gauss-normalbasis.txt) lists for each $2 \leq n \leq 1032$ the smallest ten values of $t$ such that there is a type-$t$ GNB of GF(2^n). Note that different values of $t$ do not necessary lead to different multiplication matrices, especially for small values of $n$. For example, the modulo 2 reduced multiplication matrices for $n = 6$ and the 10 smallest values of $t$ are:

```
t=2:   t=3:   t=6:   t=10:   t=11:   t=23:   t=27:   t=30:   t=35:   t=55:
 1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1
 1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1
 1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1
 1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1
 1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1
 1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1  1...1
```

40.9.2 Determination of the field polynomial

We give algorithms to compute the field polynomial corresponding to a given pair $(n, t)$ such that a type-$t$ GNB exists over GF(2^n).
40.9.2.1 Algorithm with complex numbers

The normal polynomial corresponding to a type-\( t \) Gaussian basis can be computed as follows:

1. Set \( p = t \cdot n + 1 \) and determine \( r \) such that the order of \( r \) modulo \( p \) equals \( t \).
2. For \( 1 \leq k \leq n \) compute \( w_k = \sum_{j=0}^{t-1} \exp(a_k 2 \pi i/p) \) where \( a_k = 2^k r^j \mod p \).
3. Let \( z(x) = \prod_{k=1}^{n} (x - w_k) \), this is a polynomial with real integer coefficients.
4. Return the polynomial with coefficients reduced modulo 2.

The computation of the polynomial \( z(x) \) uses complex (inexact) arithmetics. That its coefficients should be close to real integers can be used as a check.

The following pari/gp routine computes the complex polynomial. In order to keep the arguments for the exponential function small the values \( a_k \) are computed modulo \( p \), the periodicity of \( f(a) := \exp(a 2 \pi i/p) \).

```
1 gauss_zpoly(n, t)=
2 { /* return field polynomial for type-t Gaussian normal basis */
3   local(p, r, wk, tk1, tk, a, zp);
4   p = n*t + 1;
5   r = znprimroot(p)**n; \r has order t (mod p)
6   zp = 1;
7   tk1 = Mod(2,p); tk = Mod(1,p);
8   for (k=1, n, \n9       tk *= tk1; \n10       wk = 0;
11       a = tk;
12       for (j=0, t-1, \n13           wk += exp(2.0*I*Pi*lift(a)/p);
14           a *= r;
15         );
16       zp *= (x-wk);
17   );
18   return ( zp );
}
```

The final step uses pari/gp’s function `round()` which rounds all coefficients of its polynomial argument:

```
1 gauss_poly(n, t)=
2 { /* return field polynomial for type-t Gaussian normal basis */
```
Chapter 40: Binary finite fields: GF($2^n$)

n=11 t=2: p=23

\begin{align*}
a(1)=2 & \ w(1)=\left(+1.7088388090929771051 - 1.175494350822287508 E-38 i\right) \\
a(2)=4 & \ w(2)=\left(+0.9201300754623042520 + 5.877471754111437540 E-39 i\right) \\
a(3)=8 & \ w(3)=\left(-1.15363606442297342825 - 5.877471754111437540 E-39 i\right) \\
a(4)=16 & \ w(4)=\left(-0.6697592243419723039 - 2.938735877055718770 E-39 i\right) \\
a(5)=9 & \ w(5)=\left(+0.4069120261052675797 + 1.193861450053885750 E-39 i\right) \\
a(6)=18 & \ w(6)=\left(+1.981371892072665047 + 2.77520304718182046 E-38 i\right) \\
a(7)=13 & \ w(7)=\left(-1.364948267293419618 - 1.205340887073634651 E-39 i\right) \\
a(8)=3 & \ w(8)=\left(+1.3651062864373081657 + 1.175494350822287508 E-38 i\right) \\
a(9)=6 & \ w(9)=\left(-0.1364848267293419518 - 1.205340887073634651 E-39 i\right) \\
a(10)=12 & \ w(10)=\left(-1.981371892072665047 + 2.77520304718182046 E-38 i\right) \\
a(11)=1 & \ w(11)=\left(+1.9258345746955985900 - 2.938735877055718770 E-38 i\right) \\
\end{align*}

\begin{align*}
z(x)=x^{11} + x^{10} - 10*x^9 - 9*x^8 + 36*x^7 + 28*x^6 - 56*x^5 \\
- 35*x^4 + 35*x^3 + 15*x^2 - 6*x - 1 \\
p(x)=x^{11} + x^{10} + x^8 + x^4 + x^3 + x^2 + x + 1 \\
n=11 t=6: p=67 \\
\begin{split}
\text{Figure 40.9-D: Numerical values with the computation of the field polynomial for } n=11 \text{ and types } t \in \{2, 6, 8, 18\}. \text{ The final results are identical for } t=6 \text{ and } t=8. \\
\end{split}
\end{align*}

\begin{verbatim}
3 local(pp, zp);
4  zp = gauss_zpoly(n, t);
5  pp = round(real(zp)); /* rounds all coefficients */
6  pp *= Mod(1,2); /* coefficients modulo 2 */
7  return( pp );
8 }
\end{verbatim}

The results for type-1 bases can be verified using relation 40.8-1 on page 910, results with type-2 bases with relations 40.8-2a...40.8-2b on page 911. The intermediate values occurring with the computation for $n=4$ and the types $t \in \{1, 3, 7\}$ are shown in figure 40.9-C. The values for $n=11$ and the types $t \in \{2, 6, 8, 18\}$ are shown in figure 40.9-D.

The computation can be optimized by using a trigonometric recursion as described in section 20.4.2 on page 412. We further exploit symmetry, and use real values if the type $t$ is even:

\begin{verbatim}
1  vexp(p, t)=
2  { local(ve, ph, c, s, al, be, cp, sp, tt);
3    tt = 2.0*Pi/p; \ angle increment
4    c = 1.0; s = 0.0; ga = ph; al = 2.0*(sin(0.5*tt))^2; be = sin(tt);
5    ve = vector(p); ve[1] = 1.0;
6    if ( t&1, /* odd t, need complex values */
7        for (j=1, (p-1)>>1, 
8            tt = c;
9            c -= (al*tt+be*s);
10           s -= (al*s -be*tt);
11           ve[j+1] = c + I*s;
12           ve[p-j+1] = c - I*s;
13        );
14    , /* even t: can use real values */
15        for (j=1, (p-1)>>1, 
16            tt = c;
17            c -= (al*tt+be*s);
18            s -= (al*s -be*tt);
19            ve[j+1] = c + s;
20            ve[p-j+1] = c - s;
21    )
\end{verbatim}
The computation of the field polynomial needs two changes for even \( t \):

1. Set \( p = t \cdot n + 1 \) and determine \( r \) such that the order of \( r \) modulo \( p \) equals \( t \).
2. Set \( M(x) = \sum_{k=0}^{p-1} x^k \). All computations are done modulo \( M \).
3. If \( t \) equals 1 return \( M \).
4. Set \( F_0 = 1 \) (modulo \( M \)).
5. For \( 1 \leq k \leq n \):
   (a) Set \( Z_k = \sum_{j=0}^{t-1} x^{a(k,j)} \) (modulo \( M \)) where \( a(k,j) = 2^k r^j \mod p \).
   (b) Set \( F_k = \( x + Z_k \) \cdot F_{k-1} \) (modulo \( M \)).
6. Return \( F_n \) with coefficients reduced modulo 2.

The intermediate quantities in the computation for \( n = 4 \) and \( t = 3 \) are shown at the top of figure 40.9-E. The result is a polynomial over the integers identical to the one computed with the algorithm that uses complex numbers. When all polynomials are taken over GF(2) the computation proceeds as shown at the bottom of figure 40.9-E. Implementation in pari/gp:

```plaintext
40.9.2.2 Algorithm working in GF(2)

The following algorithm is a variation of what is given in \[265\].

1. Set \( p = t \cdot n + 1 \) and determine \( r \) such that the order of \( r \) modulo \( p \) equals \( t \).
2. Set \( M(x) = \sum_{k=0}^{p-1} x^k \). All computations are done modulo \( M \).
3. If \( t \) equals 1 return \( M \).
4. Set \( F_0 = 1 \) (modulo \( M \)).
5. For \( 1 \leq k \leq n \):
   (a) Set \( Z_k = \sum_{j=0}^{t-1} x^{a(k,j)} \) (modulo \( M \)) where \( a(k,j) = 2^k r^j \mod p \).
   (b) Set \( F_k = \( x + Z_k \) \cdot F_{k-1} \) (modulo \( M \)).
6. Return \( F_n \) with coefficients reduced modulo 2.

The intermediate quantities in the computation for \( n = 4 \) and \( t = 3 \) are shown at the top of figure 40.9-E. The result is a polynomial over the integers identical to the one computed with the algorithm that uses complex numbers. When all polynomials are taken over GF(2) the computation proceeds as shown at the bottom of figure 40.9-E. Implementation in pari/gp:

```
Chapter 40: Binary finite fields: GF($2^n$)

n=4  t=3:  p=13  \ integer computation

\begin{verbatim}
F = ('x+Z)*F;
t2 *= t21;
return ( lift(F) );
\end{verbatim}

While the algorithm avoids inexact arithmetic the polynomial modulus $M$ is of degree $p - 1 = nt$ which is large for large $t$. In practice the computation with complex numbers is much faster. It finishes in less than a second for $n = 620$ and $t = 3$ (and a working precision of 150 decimal digits) while the exact method needs about two minutes.

Using the redundant modulus $x^p - 1 = (x - 1) \cdot M$ gives a significant speedup:

\begin{verbatim}
F = ('x+Z)*F;
t2 *= t21;
return ( lift(F) );
\end{verbatim}

Now computation of the polynomial for $n = 620$ and $t = 3$ takes less than 9 seconds. The final reduction can be simplified by observing that no reduction is needed if the constant coefficient is one, and else all

\[ F = \ \{ \ \text{snip} \ \} \]
coefficients just have to be negated. So the end of the routine can be changed to

\begin{verbatim}
\\ final reduction for redundant modulus (simplified):
F = lift(F);
if ( G==polcoeff(F,0), F=sum(k=0, n, (1-polcoeff(F,k))*'x^k) );
return ( F );
\end{verbatim}
Appendix A

The electronic version of the book

How to make the hyperlinks work

The hyperlink showing as [FXT: bits/revbin.h] points to file://.fxtdir/bits/revbin.h. To make this work on your machine you may want to create (in your home directory) a soft-link to the directory of the FXTSources. For example, assuming that the package is located at ~/work/fxt, execute the following statement:

```
ln -sv ~/work/fxt ~/.fxtdir
```

Similarly, for hfloat, do

```
ln -sv ~/work/hfloat ~/.hfloatdir
```

Test with the hyperlink [hfloat: src/hf/funcsrt.cc] which points to file://.hfloatdir/src/hf/funcsrt.cc

You may want to add the following lines to your file://.mailcap:

```
text/plain:/usr/bin/emacs -no-site-file %s &
text/x-csrc:/usr/bin/emacs -no-site-file %s &
text/x-chdr:/usr/bin/emacs -no-site-file %s &
text/x-c++src:/usr/bin/emacs -no-site-file %s &
text/x-c++hdr:/usr/bin/emacs -no-site-file %s &
```

Here the editor emacs is used for viewing plain text, C and C++ sources and headers.
Appendix B

Machine used for benchmarking

The machine used for performance measurements is an AMD64 (Athlon64) clocked at 2.2 GHz with dual channel double data rate (DDR) clocked at 200 MHz (‘800 MHz’). It has 512 kB (16-way associative) second level cache and separate first level caches for data and instructions, each 64 kB (and 2-way associative). Cache lines are 64 bytes (8 words, 512 bits). The memory controller is integrated in the CPU.

The CPU has 16 general purpose (64 bit) registers that are addressable as byte, 16 bit word, 32 bit word, or 64 bit (full) word. These are used for integer operations and for passing integer function arguments. There are 16 (128 bit, SSE) registers that are used for floating point operations and for passing floating point function arguments. The SSE registers are SIMD registers. Additionally, there are 8 (legacy, x87) FPU registers.

The performance-wise interesting information reported by the CPUID instruction is:

Vendor: AuthenticAMD
Name: AMD Athlon(tm) 64 Processor 3400+
  Family: 15, Model: 15, Stepping: 0
Level 1 cache (data): 64 kB, 2-way associative.
  64 bytes per line, lines per tag: 1.
Level 1 cache (instr): 64 kB, 2-way associative.
  64 bytes per line, lines per tag: 1.
Level 2 cache: 512 kB, 16-way associative
  64 bytes per line, lines per tag: 1.
Max virtual addr width: 48
Max physical addr width: 40
Features:
  lm: Long Mode (64-bit mode)
mtrr: Memory Type Range Registers
tsc: Time Stamp Counter
fpu: x87 FPU
3dnow: AMD 3DNow! instructions
3dnowext: AMD Extensions to 3DNow!
mmx: Multimedia Extensions
mmxext: AMD Extensions to MMX
sse: Streaming SIMD Extensions
sse2: Streaming SIMD Extensions-2
cmov: CMOV instruction (plus FPU FCMOVCC and FCOMI)
cx8: CMPXCHG8 instruction
clflush: CLFLUSH instruction
fxsr: FXSAVE and FXRSTOR instructions

Special instructions as SIMD, prefetch and non-temporal moves are not used unless explicitly noted.

See [136] for a comparison of instruction latencies and throughput for various x86 CPU cores. You do want to study the cited document before buying an x86-based system.

The compiler used was the GNU (C and C++) compiler [120].
Appendix C

The pseudo language Sprache

Many algorithms in this book are given in a pseudo language called Sprache. Sprache is meant to be immediately understandable for everyone who ever had contact with programming languages like C, FORTRAN, Pascal or Algol. Sprache is hopefully self explanatory. The intention of using Sprache instead of completely relying an mathematical formulas (like tensor formalism) or algorithm description by words is to minimize the work it takes to translate the given algorithm to one’s favorite programming language. It should be mere syntax adaptation.

By the way, ‘Sprache’ is the German word for language.

```plaintext
// a comment:
// comments are useful.

// assignment:
t := 2.71

// parallel assignment:
{s, t, u} := {5, 6, 7}

// same as:
s := 5
t := 6
u := 7

{s, t} := {s+t, s-t}

// same as (avoiding the temporary):
temp := s + t
s := temp
t := s - t

// if conditional:
if a==b then a:=3

// with block
if a>=3 then
{
    // do something ...
}

// a function returns a value:
function plus_three(x)
{
    return x + 3
}

// a procedure works on data:
procedure increment_copy(f[],g[],n)
// real f[0..n-1] input
// real g[0..n-1] result
{
    for k:=0 to n-1
    {
        g[k] := f[k] + 1
    }
}

// for loop with stepsize:
```

[fxtbook draft of 2008-August-17]
Chapter C: The pseudo language Sprache

for i:=0 to n step 2 // i:=0,2,4,6,...
{ // do something
}
// for loop with multiplication:
for i:=1 to 32 mul_step 2
{ print i,", ",
}
will print 1, 2, 4, 8, 16, 32,
// for loop with division:
for i:=32 to 8 div_step 2
{ print i, ", ",
}
will print 32, 16, 8,
// while loop:
i:=5
while i>0
{ // do something 5 times...
i := i - 1
}
The usage of foreach emphasizes that no particular order is needed in the array access (so parallelization is possible):

procedure has_element(f[],x)
{
foreach t in f[]
{ if t==x then return TRUE
}
return FALSE
}

Emphasize type and range of arrays:

real a[0..n-1], // has n elements (floating point reals)
complex b[0..2**n-1] // has 2**n elements (floating point complex)
mod_type m[729..1728] // has 1000 elements (modular integers)
type i[] // has ? elements (integers)

Arithmetical operators: +, -, *, /, % and ** for powering. Arithmetical functions: min(), max(),
gcd(), lcm(), ...

Mathematical functions: sqr(), sqrt(), pow(), exp(), log(), sin(), cos(), tan(), asin(),
acos(), atan(), ...

Bitwise operators: ~, & , |, ^ for bit-wise complement, AND, OR, XOR, respectively. Bit shift operators: A<<3 shifts (the integer) A 3 bits to the left A>>1 shifts A 1 bits to the right.

Comparison operators: ==, !=, <, > ,<=, >=

There is no operator '=' in Sprache, only '==' (for testing equality) and ':=' (assignment operator).

A well-known constant: PI = 3.14159265...

The complex square root of minus one in the upper half plane: I = √−1

Boolean values: TRUE and FALSE

Logical operators: NOT, AND, OR, XOR

Modular arithmetic: x := a * b mod m shall do what it says, i := a**(-1) mod m shall set i to the modular inverse of a.
Appendix D

The pari/gp language

We give a short introduction to the pari/gp language.

From the manual page of pari \[215\] (slightly edited):

**NAME**

gp - PARI calculator

**SYNOPSIS**

**DESCRIPTION**

Invokes the PARI-GP calculator. This is an advanced programmable calculator, which computes symbolically as long as possible, numerically where needed, and contains a wealth of number-theoretic functions (elliptic curves, class field theory...). Its basic data types are integers, real numbers, exact rational numbers, algebraic numbers, p-adic numbers, complex numbers, modular integers, polynomials and rational functions, power series, binary quadratic forms, matrices, vectors, lists, character strings, and recursive combinations of these.

**Interactive usage**

To use pari/gp interactively, just type `gp` at your command line prompt. A startup message like the following will appear:

```
GP/PARI CALCULATOR Version 2.3.3 (released)
amd64 running linux (x86-64 kernel) 64-bit version
compiled: Jan 15 2008, gcc-4.2.1 (SUSE Linux)
(readline v5.2 enabled, extended help available)
Copyright (C) 2000-2006 The PARI Group
PARI/GP is free software, covered by the GNU General Public License, and comes WITHOUT ANY WARRANTY WHATSOEVER.
Type ? for help, \q to quit.
Type ?12 for how to get moral (and possibly technical) support.
parisize = 8000000, primelimit = 500000
? 
```

The question mark in the last line is a prompt, the program is waiting for your input.

```
? 1+1
%1 = 2

Here we successfully computed one plus one. Next we compute a factorial:

? 44!
%2 = 265827157478844876804362581101461589031963852800000000
```

[fxtbook draft of 2008-August-17]
Integers are of unlimited precision, the practical limit is the amount of physical RAM. For floating point numbers, the precision (number of decimal digits) can be set as follows:

```
? default(realprecision,55)
%3 = 55
? sin(1.5)
%4 = 0.9974949866040544309417233711414873227066514259221158219
```

The history numbers `%N` (where `N` is a number) can be used to recall the result of a prior computation:

```
? %4
%5 = 0.9974949866040544309417233711414873227066514259221158219
```

The output of the result of a calculation can suppressed using a semicolon at the end of the command. This can be useful for timing purposes:

```
? default(realprecision,10000)
%5 = 10000
? sin(2.5);
```

The command `##` gives the time used for the last computation.

The printing format can be set independently of the precision used:

```
? default(realprecision,10000);
? default(format,"g.15");
? sin(2.5)
%6 = 0.598472144103956
```

Command line completion is available, typing `si`, then the tab-key, gives a list of builtin functions whose names start with `si`:

```
? si
  sigma sign simplify sin sinh sizebyte sizedigit
```

You can get the help text by using the question mark, followed by the help topic:

```
? ?sinh
  sinh(x): hyperbolic sine of x.
```

A help overview is invoked by a single question mark

```
? ?
  Help topics: for a list of relevant subtopics, type ?n for n in
  0: user-defined identifiers (variable, alias, function)
  1: Standard monadic or dyadic OPERATORS
  2: CONVERSIONS and similar elementary functions
  3: TRANSCENDENTAL functions
  4: NUMBER THEORETICAL functions
  5: Functions related to ELLIPTIC CURVES
  6: Functions related to general NUMBER FIELDS
  7: POLYNOMIALS and power series
  8: Vectors, matrices, LINEAR ALGEBRA and sets
  9: SUMS, products, integrals and similar functions
 10: GRAPHIC functions
 11: PROGRAMMING under GP
 12: The PARI community

Select a section by its number:

```
? 7
  deriv eval factorpadic
  intformal padicappr polcoeff polycyclo
  poldegree poldiscreduced polhensellift
  polinterpolate polisirreducible pollead pollegendre
  polrecip polresultant polroots polrootsmod
  polrootspadic polsturm polsubcyclo polsylvestermatrix
  polsym poltchebi polzagier serconvol
  serlaplace serreverse subst substpol
  substvec taylor thue thueinit
```

You should try both of the following

```
? ??tutorial
  displaying 'tutorial.dvi'.
? ??displaying 'users.dvi'.
```

A short overview (which you may want to print) of most functions can be obtained via
A session can be ended by either entering `quit` or just hitting control-d.

**Builtin operators and basic functions**

There are the ‘usual’ operators `+`, `-`, `*`, `/`, `^` (powering), and `%` (modulo). The operator `\` gives the integer quotient without remainder. The assignment operator is `=`. C-style shortcuts are available, for example `t+3` is the same as `t=t+3`.

The increment by one can be abbreviated as `t++`, the decrement as `t--`. [Technical note: these behave as the C-language pre-increment (and pre-decrement), that is, the expression evaluates to `t+1`, not `t`. There is no post-increment or post-decrement in pari/gp.]

Comparison operators are `==`, `!=` (alternatively `<>`), `>`, `>=`, `<`, and `<=`. Logical operators are `& &` (and), `(or), and `!` (not)

Bit-wise operations for integers are

`bitand` `bitneg` `bitnegimply` `bitor` `bittest` `bitxor`

and

`shift(x,n)`: shift x left n bits if n>=0, right -n bits if n<0.
`shiftmul(x,n)`: multiply x by 2^n (n>=0 or n<0)

One can also use the operators `>>` and `<<`, as in the C-language, and the shortcuts `>>=` and `<<=`.

An overview of basic functions is obtained as

```
? ?
Col     List     Mat     Mod     Pol     Polrev     Qfb
Ser     Set      Str     Strchr   Strexpand Strtex   Vec
Vecsmall binary bitand bitneg bitnegimply bitor bittest
bitxor   ceil     centerlift changevar component conj conjvec
denominator floor frac imag length lift norm
norml2   numerator numtoperm padicprec permtonum precision random
real     round    simplify sizebyte sizedigit truncate valuation
variable
```

Here are a few:

`sign(x)`: sign of x, of type integer, real or fraction.
`max(x,y)`: maximum of x and y.
`min(x,y)`: minimum of x and y.
`abs(x)`: absolute value (or modulus) of x.
`floor(x)`: floor of x = largest integer <= x.
`ceil(x)`: ceiling of x = smallest integer >= x.
`frac(x)`: fractional part of x = x-floor(x)

An overview of sums, products, and some numerical functions:

```
? ??
intcirc   intfouriercos intfourierexp intfouriersin
intfuncinit intlaplaceinv intmellininv intmellininvshort
intnum     intnuminit  intnuminitgen intnumromb
intnumstep  prod      prodeuler    prodinf
solve      sum        sumalt      sumdiv
suminf     sumnum     sumnumalt   sumnuminit
sumpos
```

For example:

`sum(X=a,b,expr,{x=0})`: x plus the sum (X goes from a to b) of expression expr.
`prod(X=a,b,expr,{x=1})`: x times the product (X runs from a to b) of expression.

**Basic data types**

Strings:
Chapter D: The pari/gp language

Integers, floating-point numbers (real or complex), and complex integers:

? factor(239+5*I)
[-I 1]
[1 + I 1]
[117 + 122*I 1]

Exact rationals:

? 2/3+4/5
22/15

Modular integers:

? Mod(3,239)^77
Mod(128, 239)

Vectors and matrices:

? v=vector(5,j,j^2)
[1, 4, 9, 16, 25]
? m=matrix(5,5,r,c,r+c)
[2 3 4 5 6]
[3 4 5 6 7]
[4 5 6 7 8]
[5 6 7 8 9]
[6 7 8 9 10]

The vector is a row vector, trying to right-multiply it with the matrix fails:

? t=m*v *** impossible multiplication t_MAT * t_VEC.

The operator \( \sim \) transposes vectors (and matrices), we multiply with the column vector:

? t=m*v~
\%14 = \[280, 335, 390, 445, 500\]~

The result is a column vector, note the tilde at the end of the line.

Vector indices start with one:

? t[1]
\%15 = 280

Symbolic computations

Univariate polynomials:

? (1+x)^7
x^7 + 7*x^6 + 21*x^5 + 35*x^4 + 35*x^3 + 21*x^2 + 7*x + 1

? factor((1+x)^6+1)
[x^2 + 2*x + 2 1]
[x^4 + 4*x^3 + 5*x^2 + 2*x + 1 1]

Power series:

? (1+x+O(x^4))^7
1 + 7*x + 21*x^2 + 35*x^3 + O(x^4)

? log((1+x+O(x^4))^7)
7*x - 7/2*x^2 + 7/3*x^3 + O(x^4)

Types can be nested, here we compute modulo the polynomial \( 1 + x + x^7 \) with coefficients over GF(2):

? t=Mod(1+x, Mod(1, 2)*{1+x*x^7})^77
Mod(Mod(1, 2)*x^3 + Mod(1, 2)*x + Mod(1, 2), Mod(1, 2)*x^7 + Mod(1, 2)*x + Mod(1, 2))

? lift(t) \( \backslash \) discard modulo polynomial
Mod(1, 2)*x^3 + Mod(1, 2)*x + Mod(1, 2)

? lift(lift(t)) \( \backslash \) discard modulo polynomial, then modulus 2 with coefficient
\( x^3 + x + 1 \)

Symbolic computations are limited when compared to a computer algebra system: for example, multivariate polynomials cannot (yet) be factored, and there is no symbolic solver for polynomials.

An uninitialized variable evaluates to itself, as a symbol:
To create a symbol, prepend a tick:

? \w=3
3
? \hello='\w /* the symbol \w, not the value of \w */
\w

Here is a method to create symbols:

? sym(k)=eval(Str("A", k))
? t=vector(5, j, sym(j-1))
[\A0, \A1, \A2, \A3, \A4]

The ingredients are \texttt{eval()} and \texttt{Str()}:

\texttt{eval(x)}: evaluation of \texttt{x}, replacing variables by their value.
\texttt{Str({str}*): concatenates its (string) argument into a single string.

Some more trickery to think about:

\texttt{sym(k)=eval(Str("A", k))}
\texttt{t=vector(5, j, sym(j-1)); print("1: t=", t);}
\{ for (k=1, 5,
  sy = sym(k-1);
  \/* assign to the symbol that \texttt{sy} evaluates to, the value of \texttt{v}: */
  eval( Str( "\texttt{sy}", "=" , Str( \texttt{v} ) ) ) ;
  \}
print("2: t=", t); /* no lazy evaluation with pari/gp */
\texttt{t=eval(t); print("3: t=", t);}

The output of this script is

1: t=[\A0, \A1, \A2, \A3, \A4]
2: t=[\A0, \A1, \A2, \A3, \A4]
3: t=[1, 1/4, 1/9, 1/16, 1/25]

More builtin functions

The following constants and transcendental functions are known by pari:

\texttt{? \?3}
\begin{verbatim}
? Euler I Pi abs acos acosh agm arg
asin asinh atan atanh bernfrac bernreal bernvec besselh1 besselh2
cotan dilog eint1 erfc eta exp gamma gammah
hyperu incgam incgami lngamma log polylog psi sin
sinh sqrt sqrtl sqrtn tan tanh teichmuller theta
cotan weber zeta
\end{verbatim}

To obtain information about a particular function, use a question mark:

? \?sinh
\texttt{sinh(x): hyperbolic sine of x.}

Transcendental functions will also work with complex arguments and symbolically, returning a power series:

? sinh(x)
%9 = x + 1/6*x^-3 + 1/120*x^-5 + 1/5040*x^-7 + 1/362880*x^-9 \n + 1/39916800*x^-11 + 1/6227020800*x^-13 + 1/1307674368000*x^-15 + O(x^-17)

The line break (and the backslash indicating it) was manually entered for layout reasons. The ‘precision’
(that is default order) of power series can be set by the user:

? default(seriesprecision,9);
? sinh(x)
%11 = x + 1/6*x^-3 + 1/120*x^-5 + 1/5040*x^-7 + 1/362880*x^-9 + O(x^-10)

One can also manually give the \texttt{O(x^N)} term:

? sinh(x+O(x^-23))
%12 = x + 1/6*x^-3 + 1/120*x^-5 + 1/5040*x^-7 + \\n\[--snip--] \n + 1/121645100408832000*x^-19 + 1/51090942171709440000*x^-21 + O(x^-23)
Chapter D: The pari/gp language

Functions operating on matrices are (type `mat`, then hit the tab-key)

- `matadjoint`
- `matalgobasis`
- `matbasistoalg`
- `matcompanion`
- `matdet`
- `matdetint`
- `matdiagonal`
- `matformlement`
- `matindexrank`
- `matintersect`
- `matinverseimage`
- `matsize`
- `matsolve`
- `matsolvenmod`
- `matsupplement`
- `mattranspose`

Builtin number theoretical functions are

- `? addprimes` `bestappr` `bezout` `bezoutres` `bigomega` `binomial`
- `chinese` `content` `contrfrac` `contrfracpqn` `core` `coredisc`
- `dirdiv` `dirdiver` `dirnmul` `divisors` `eulerphi` `factor`
- `ffactorback` `ffactorford` `factorial` `factorint` `factormod`
- `ffinit` `ffibonacci` `gcd` `hilbert` `ifundamental` `ispower`
- `isprime` `ipseudoprime` `issquare` `issquarefree` `kroenecker` `lcm`
- `moebius` `nextprime` `numpart` `numdiv` `omega` `precpriem`
- `prime` `primedep` `primes` `qfblas` `qfcompr` `qfhclassno`
- `qfbnucomp` `qfbnupov` `qfprimf` `qfprimes` `qf qfolve` `qfbsolve`
- `quadclassunit` `quaddisc` `quadgen` `quadhilbert` `quadpoly` `quadvray`
- `quadregulator` `quadunit` `removeprimes` `sigma` `sqtint` `zncoppersmith`
- `znlog` `znorder` `znprimroot` `znstar`

Functions related to polynomials and power series are

- `? deriv` `eval` `factorpadic`
- `intformal` `padicappr` `polcoeff` `polcyclo` `polcyclom`
- `poldegree` `poldisc` `poldiscreduced` `polhensellift`
- `polinterpolate` `polisirreducible` `pollead` `pollegendre`
- `полrecip` `polresultant` `polroots` `polrootsmod`
- `полrootspadic` `полстор` `polsubcy` `polsylvestermatrix`
- `polynom` `poltchebi` `polzagier` `serconvol`
- `servalp` `serlaplace` `serreverse` `subst` `subtfrac`
- `substvec` `taylor` `thue` `thueinit`

Plenty to explore!

Control structures for programming

Some loop constructs available are

- `while(a,seq):` while a is nonzero evaluate the expression sequence seq. Otherwise 0.
- `until(a,seq):` evaluate the expression sequence seq until a is nonzero.
- `for(X=a,b,seq):` the sequence is evaluated, X going from a up to b.
- `forstep(X=a,b,s,seq):` the sequence is evaluated, X going from a to b in steps of s (can be a vector of steps)
- `forprime(X=a,b,seq):` the sequence is evaluated, X running over the primes between a and b.
- `fordiv(n,X,seq):` the sequence is evaluated, X running over the divisors of n.

The expression seq is a list of statements:

- `for ( k=1, 10, stat1; stat2; stat3; ) /* last semicolon optional */`
- `for ( k=1, 10, stat1; )`
- `for ( k=1, 10, ; ) /* zero statements (do nothing, ten times) */`

(The comments enclosed in /* */ were manually added.)

The loop-variable is local to the loop:

- `? k /* not initialized in global scope => returned as symbol */`

A global variable of the same name is not changed:
For the sake of clarity, avoid using global and loop-local variables of the same name.

A loop can be aborted with the statement `break()`. The \( n \) enclosing loops are aborted by `break(n)`. With `next()`, the next iteration of a loop is started (and the statements until the end of the loop are skipped). With `break(n)` the same is done for the \( n \)-th enclosing loop.

And yes, there is an if statement:

```plaintext
if(a,seq1,seq2): if a is nonzero, seq1 is evaluated, otherwise seq2. seq1 and seq2 are optional, and if seq2 is omitted, the preceding comma can be omitted also.
```

To have more than one statement in the branches use semicolons between the statements:

```plaintext
if ( a==3, /* then */
   b=b+1;
   c=7;
   , /* else */
   b=b-1;
   c=0;
);  
```

### Non-interactive usage (scripts)

Usually one will create scripts that are fed into gp (at the command line):

```plaintext
gp -q < myscript.gp
```

The option `-q` suppresses the startup message and the history numbers \( %N \).

If the script contains just the line

```plaintext
exp(2.0)
```

the output would be

```plaintext
7.3890560989306502272304274605750078131
```

To also see the commands in the output, add a `default(echo,1);` to the top of the file. The output will then be

```plaintext
? exp(2.0)
7.3890560989306502272304274605750078131
```

You should use comments in your scripts, there are two types of them:

```plaintext
\ a line comment, started with backslashes
/* a block comment
 can stretch over several lines, as in the C-language */
```

Comments are not visible in the output. With the script

```plaintext
default(echo, 1);
\ \ sum of square numbers:
s=0; for (k=1, 10, s=s+k*k); s
```

the output would be

```plaintext
? default(echo,1);
? s=0;for(k=1,10,s=s+k*k);s
385
```

Note that all whitespaces are suppressed in the output.

A command can be broken into several lines if it is inclosed into a pair of braces:
{ for (k=1, 10,
        s=s+k*k;
        print(k,"": s="", s);}

This is equivalent to the one-liner
for (k=1, 10, s=s+k*k; print(k,"": s="", s); );

User-defined Functions

Now we define a function:

defsum(n, p)=
{ /* return the sum 1^p+2^p+3^p+...+n^p */
    local(t);
    t = 0;
    for (k=1, n,
        t = t+k^p); \ '^' is the powering operator
    return( t );
}

The statement local(t); makes sure that no global variable named t (if it exists) would be changed by the function. It must be the first statement in the function. The variable k in the for()-loop is automatically local and should not be listed with the locals. Note that each statement is terminated with a semicolon. The output would be

? defsum(n,p)=local(t);t=0;for(k=1,n,t=t+k^p);return(t);
? defsum(10,2)
385

Note how the function definition is changed to a one-liner in the output.

If you have to use global variables, list them at the beginning of your script as follows:

global(var1, var2, var3);

Any attempt to use the listed names as names of function arguments or local variables in functions will trigger an error.

Arguments are passed by value. There is no mechanism for passing by reference, global variables can be a workaround for this.

Arguments can have defaults, as in
defsum(n, p=2)= /* etc */

Calling the function as either defsum(9) or defsum(9,) would compute the number of the first 9 squares. Defaults can appear anywhere in the argument list, as in
defsum(a, b=3, c)= return( a+b+c );

So defsum(1,,1) would return 5.

All arguments are implicitly given the default zero, so the sequence of statements
def(a, b, c)= print(a,"":"",b,"":"",c);
def(,,)
def()
def

will print three times 0:0:0. This feature is rarely useful and does lead to obscure errors. It will hopefully be removed in future versions of pari/gp.
Bibliography


[58] J. M. Borwein, P. B. Borwein: On the Mean Iteration \((a, b) \rightarrow (\frac{a+b}{2}, \sqrt{ab}+\frac{b}{2})\), Mathematics of Computation, vol.53, no.187, pp.311-326, (July-1989).


940

BIBLIOGRAPHY


[170] Donald E. Knuth: Structured programming with go to statements, ACM Computing Surveys, vol.6, no.4, (December-1974). 144


[200] Keith Matthews: *Solving $Ax^2 - By^2 = N$ in integers, where $A > 0$, $B > 0$ and $D = AB$ is not a perfect square and $\gcd(A, B) = \gcd(A, N) = 1$*, web note, (13-September-2007). Online at [http://www.numbertheory.org/notes.html](http://www.numbertheory.org/notes.html).


[248] Hong-Yeop Song: Examples and Constructions of Hadamard Matrices, Dept. of Electrical and Electronics Engineering, Yonsei University, Korea, (June-2002). Online at \url{http://calliope.uwaterloo.ca/~ggong/710T4/Song-lecture.ps}


[253] Carl Stormer: Sur l’application de la théorie des nombres entiers complexes a la solution en nombres rationnels \( x_1, x_2 \ldots, x_n \) de l’équation: \( c_1 \arctg x_1 + c_2 \arctg x_2 + \ldots + c_n \arctg x_n = k \frac{\pi}{4} \), Archiv for Mathematik og Naturvidenskab, B.XIX, Nr.3, (vol.19, no.3), pp.1-96, (1896).


Index

Symbols ........................................
−2, representations with radix −2 63
999, computation 150
E, elliptic function 606
K, elliptic function 605
O(1) algorithms 552
Θ2, Θ3, and Θ4 (theta functions) 627
⊙, cyclic convolution 437
⊙(v), weighted convolution 446
⊙lin, linear convolution 440
π computation
– AGM vs. binary splitting, 645
– iterative algorithms, 612
φ(n), Euler’s totient function 773
i, computation 633
z, series for 710
η-product (eta-product) 694
Wv[,], weighted transform 445
2-adic, inverse and square root 61
2D Hilbert curve 37, 164, 359, 744
3D Hilbert curve 359

A ..................................................
AC (adjacent changes), Gray code 389
ac_gray_delta() 391
acceleration of convergence, sumalt algorithm 651
ACF (auto correlation function) 442, 873
acyclic (linear) convolution 440
acyclic (linear) correlation 442
addition, modulo m 763
additive group, with a ring 772
additive inverse, modulo m 766
adjacency matrix of a graph 381
adjacent changes (AC), Gray code 389
adjacent nodes in a graph 381
AGM
– (arithmetic-geometric mean), 603
– 4-th order variant, 604
– and hypergeometric functions, 608
– vs. binary splitting, 645
algebra 814
all-ones polynomial 850, 910
all-ones polynomials, trace vector 895
all_irredpoly (C++ class) 854
alternating permutations 286
alternating series, and continued fractions 720
alternating series, sumalt algorithm 651
AND-convolution 490
apply_permutation() 109, 117
applying a permutation to data, in-place routine 116
approximations, initial, for iterations 578
arbitrary length FFT 451
arc (edge) of a digraph 381
arctan relations for π 620
arctan, computation by rectangular scheme 647
argument reduction
– for arctan, 631
– for cos, 635
– for exp, 635
– for log, 631
arithmetic transform 483
arithmetic transform, convolution property 489
arithmetic, modular 763
arithmetic-geometric mean (AGM) 603
array notation 171
array, of bits 158
asm trick, with GCC 4, 531
asymptotics, of an algorithm 551
auto correlation function (ACF) 442, 873
automaton, finite 163
average, of two integers, without overflow 26

B .................................
backtracking 381
bag (multiset) 291
base (radix) conversion 646
base field 803, 885
basis functions, Reed-Muller transform 486
Beatty sequence with Φ 753
Bell numbers 137, 345
Bell polynomials 346
Ben-Or test for irreducibility 835
Berlekamp’s Q-matrix algorithm 855
Bhaskara equation 811
big endian machine 3
binary
- exponentiation, 565
- finite field, 804, 885
- GCD algorithm, 766
- heap, 154
- powering, 565
- relation, 133
- search, 123
binary polynomial 819
binary splitting
- for rational series, 641
- vs. AGM, 645
- with continued fractions, 718
binary_necklace (C++ class) 304
Binet form, of a recurrence 675
binomial coefficient
- and type-2 ONB, 911
- modulo a prime, 912
- number of combinations, 175
bit combinations 67
bit counting 19
bit subsets, via sparse counting 71
bit-array 158
bit-array, fitting in a word 25
bit-block boundaries, determination 13
bit-reversal 36
bit-reversal permutation 91
bit-subset, testing 24
bit-wise
- reversal, 35
- rotation, 29
- zip, 40
bit_fibgray (C++ class) 82
bit_necklace (C++ class) 32
bit_rotate_sgn() 56
bit_subset (C++ class) 71
bitarray (C++ class) 158
bitpol_factor() 859
bitpol_mult() 89
bitpol_normal_q() 900
bitpol_refine_factors() 856
bitpol_sreduce() 858
bitset permutation 91
BIT_S_PER_LONG 3
blocks of bits, counting 21
blocks of bits, creation 12
blocks, swapping via quadruple reversion 98
blue code 51, 368
blue code, fixed points 53
bracelets, as equivalence classes 135
branches, avoiding them 24
bsearch 123
bsearch() 123
bsearch_approx() 124
bsearch_ge() 124
bswap instruction 36
bubble sort 27
builtins, GCC 22
butterfly diagram, for radix-2 transforms 459
byte-wise Gray code and parity 44
BYTES_PER_LONG 3

C

C++ class XYZ see XYZ (C++ class)
C2RFT see real FFT
C2RFT (complex to real FT) 426
canonical sequence 109
carries with mixed radix counting 222
carry, in multiplication 561
CAT, constant amortized time 172
catalan (C++ class) 319
Catalan constant 692
Catalan numbers 324
Cayley numbers 815
Cayley-Dickson construction 815
characteristic polynomial
- of a matrix, 898
- of a recurrence relation, 667
- with Fourier transform, 533
characteristic, of a field 886
Chase’s sequence, for combinations 190
Chebyshev polynomials
- and Pell’s equation, 814
- and products for the square root, 685
- and recurrence for subsequences, 673
- and square root approximants, 684
- as hypergeometric functions, 706
- definition, 677
- fast computation, 682
- with accelerated summation, 652
Chinese Remainder Theorem (CRT) 778
Chinese Remainder Theorem, for convolution 542
chirp z-transform 451
circuit in a graph 581
circulant matrix 904
Clausen’s products 702
CLHCA (a class of cellular automata) 880
c1z (Count Leading Zeros), GCC builtin 22
c-co-lexicographic order
- (definition), 171
- for combinations, 176
- for compositions, 193
- for permutations, 235
- for subsets of a multiset, 291
– with bit combinations, 67
colex (co-lexicographic) order 171
comb_rec (C++ class) 191
combination_chase (C++ class) 191
combination_colex (C++ class) 178
combination_emk (C++ class) 184
combination_enup (C++ class) 188
combination_lex (C++ class) 176
combination_mod (C++ class) 185
combination_pref (C++ class) 180
combination_revdoor (C++ class) 182
combinations, Gray code, with binary words 78
combinations, of k bits 67
combinatorial Gray code 171
companion matrix 608, 898
comparison function, for sorting 127
compiler, smarter than you thought 28
complement, of a permutation 110
complement-shift sequences 387
complementary basis 906
complementing the sequency 49
complete graph 383
complex numbers, construction 803
complex numbers, mult. via 3 real mult. 805
complex numbers, sorting 128
composite modulus 773
compositeness of an integer, test for 785
composition, of permutations 113
composition_colex (C++ class) 193
composition_colex2 (C++ class) 195
composition_ex_colex (C++ class) 195
compositions 193
computation of π, AGM vs. binary splitting 645
concave search, for paths in a graph 388
conditional swap 26
conference matrix 375
conjugates of an element in GF(2^n) 891
connected permutation 286
connection polynomial 861
constant
– Catalan, 652
– CORDIC scaling, 663, 664
– Fibonacci parity, 731
– Gray code, 739
– Komornik-Loreti, 726
– parity number, 724
– Pell, 755
– Pell Gray code, 758
– Pell palindromic, 765
– period-doubling, 732
– rabbit, 750
– revbin, 738
– Roth’s, 728
– ruler, 731
– sum of Gray code digits, 741
– sum-of-digits, 737
– Thue, 728
– weighted sum of Gray code digits, 743
– weighted sum-of-digits, 738
constant amortized time (CAT) 172
continued fraction 714
continued fractions, as matrix products 718
convergent, of a continued fraction 714
conversion, float to int 7
conversion, of the radix (base) 646
convex sequence 138
convolution
– acyclic (linear), 440
– and Chinese Remainder Theorem, 542
– and multiplication, 560
– AND-convolution, 490
– by FFT, without revbin permutations, 439
– by FHT, 521
– cyclic, 137
– cyclic, by FHT, 521
– dyadic, 473
– exact, 542
– linear, 440
– mass storage, 449
– negacyclic, 446, 525
– OR-convolution, 489
– property, of the Fourier transform, 438
– right-angle, 446
– skew circular, 446
– weighted, 446
– XOR-convolution, 473
cool-lex, order for combinations 180
Cooley-Tukey FFT algorithm 406
copy, reverse 426
copying one bit 8
CORDIC algorithms 660
coroutine (C++ class) 166
coroutines 165
correlation 142
cos_rot() 528
cosine transform (DCT) 528
cosine, by rectangular scheme 650
cosine, CORDIC algorithm 660
cosine, in a finite field 808
counting bits of a sparse word 21
counting bits of a word 19
counting sort 140
coupled iteration, for the square root 571
CPU instructions, often missed 88
CRC (cyclic redundancy check) 865
crc32 (C++ class) 867
crc64 (C++ class) 865

[fxtbook draft of 2008-August-17]
Creutzburg-Tasche primitive root 807
CRT (Chinese Remainder Theorem) 778
cross correlation 442
cubic root extraction 571
cubic convergence 691
cycle in a graph 831
cycle type, of a permutation 284
cycle-leaders, for the Gray code permutation 367
cycle-leaders, for the Gray permutation 104
cycles (C++ class) 112
cycles, of a permutation 111
cyclic – convolution, 437
cyclic – convolution, by FFT, 439
cyclic – correlation, 442
cyclic – distance, with binary words, 34
cyclic – group, 774
cyclic – period, of a binary word, 31
cyclic – permutation (definition), 112
cyclic – permutation, random, 119
cyclic – permutations, and factorial numbers, 215
cyclic – permutations, recursive generation, 280
cyclic – redundancy check (CRC), 865
cyclic – XOR, 374
cyclic_perm (C++ class) 281
cyclotomic polynomials
  – (definition), 687
  – and primes, 801
  – and primitive binary polynomials, 854

deadbeats wavelets 548
de Bruijn sequence, to compute bit position 15
de Bruijn graph 385
de Bruijn sequence 207, 871
de Bruijn sequence, as path in a graph 385
de Bruijn sequences, number of 872
debruijn (C++ class) 207
decimation in frequency (DIF) 409
decimation in frequency FFT algorithm 409
decimation in time (DIT) 406
decimation in time FFT algorithm 406
delta sequence 444
delta set 171
delta squared process 601
demo-programs, and timing 174
deque (C++ class) 152
deque (double-ended queue) 152
derangement 110, 287
derangement order, for permutations 277
dFT (discrete Fourier transform) 403, 406
dIF (decimation in frequency) 409
difference sets, and correlation 444
digraph 381
digraph (C++ class) 382
digraph::sort_edges() 392
digraph_paths (C++ class) 383
digraph_paths::print_turns() 393
directed graph 381
discrete cosine transform (DCT) 528
discrete Fourier transform (DFT) 403, 406
discrete sine transform (DST) 529
DIT (decimation in time) 406
division
  – algorithm using only multiplication 569
  – CORDIC algorithm, 602
  – exact, by \( C = 2^k \pm 1 \), 62
  – exact, with polynomials over GF(2) 823
divisionless iterations for polynomial roots 589
divisors (C++ class) 291
Dobinski’s formula, for Bell numbers 346
dragon curve sequence 741
dST (discrete sine transform) 529
dual basis 906
dyadic convolution 473
dyadic_convolution() 473

e ....................
e, elliptic function 606
Eades-McKay sequence, for combinations 182
easy case, with combinatorial generation 173
edge of a graph 381
edge sorting, with graph search 392
element of order \( n \) 535
elementary functions, as hypergeometric f. 705
elementary functions, as hypergeometric functions 711
elementary functions, of a computer 3
diendian-ness, of a computer 3
endo (Even Numbers DOwn) order 186
endo order, for mixed radix numbers 229
emp (Even Numbers UP) order 187
emp order for combinations 188
emp order, with permutations 271
equivalence classes 133
equivalence relation 133
equivalence relations, number of 137
equivalence classes() 134
Eratosthenes, prime sieve 769
eta-product 694
Euclidean algorithm 766
Euler numbers 250
Euler’s formula 250
Euler’s identity, for hypergeometric functions 700
Euler’s totient function 773
exact convolution 542

[fxtbook draft of 2008-August-17]
Index

exact division 61
exact division, by \( C = 2^k \pm 1 \) 62
exact division, with polynomials over \( \text{GF}(2) \) 823
exponent, of a group 772
exponential convergence 591
exponential function
  - bit-wise computation, 657
  - by rectangular scheme, 650
  - computation via \( q = \exp(-\pi K'/K) \), 633
  - iteration for, 633
  - of power series, 637
exponentiation
  - algorithms, 565
  - modulo \( m \), 766
extension field 803, 885
external algorithms 449
extraneous fixed point, of an iteration 597

F

factorial number system 236
factorial, binsplit algorithm for 641
factorization of binary polynomials 855
falling factorial basis 236
fast Fourier transform (FFT) 404
fast Hartley transform (FHT) 511
fcsr (C++ class) 873
FCSR (feedback carry shift register) 873
feedback carry shift register (FCSR) 873
Fermat numbers 794
Ferrat primes 781
ffact2cyclic() 283
ffs (Find First Set), GCC builtins 22
FFT
  - as polynomial evaluation, 562
  - radix-2 DIF, 411
  - radix-2 DIT, 409
  - radix-4 DIF, 418
  - radix-4 DIT, 416
  - split-radix algorithm, 421
FFT (fast Fourier transform) 404
FFT caching 566
FFT, for multiplication 560
FFT-primes 536
fft_complex_convoluition() 440
fft_dif4l() 419
fft_dit4_core_p1() 417
FHT
  - convolution by, 521
  - DIF step, 515
  - DIF, recursive, 515
  - DIT, recursive, 512
  - radix-2 DIF, 510
  - radix-2 DIT, 513
  - radix-2 DIT step, 512
  - shift operator, 512
FHT (fast Hartley transform) 511
fht_dif2() 516
fht_dif_core() 413
fht_real_complex_ffp() 519
Fibonacci numbers 67, 751
Fibonacci
  - \( k \)-step sequence, 304
  - numbers, 304, 306, 750
  - parity, 751
  - parity constant, 751
  - polynomials, 912
  - representation, 751
  - setup, of a shift register, 864
  - words, 300
  - words, Gray code, 81, 301
  - words, shifts-order, 210
Fibonacci-Haar transform 506
Fibonacci-Walsh transform 508
FIFO (first-in, first-out), queue 150
filter, for wavelet transforms 544
finite field 804
finite state machine (FSM) 163
fixed point, of a function 591
fixed point, of an iteration, extraneous 597
fixed points, of the blue code 53
FKM algorithm 362
FKM algorithm, for binary words 32
four step FFT 434
Fourier shift operator 408
Fourier transform (FT) 403
Fourier transform, convolution property 438
fractional (order) Fourier transform 532
fractional Fourier transform 452
free element (normal element) 900
FSM, finite state machine 163
FT (Fourier transform) 403
full path in a graph 381

G

Galois Field 885
Galois setup, of a shift register 864
Gauss’ transformation 700
Gaussian normal basis 913
GCC, builtins 22
GCD, computation 766
generalized subset-lex (gslex) order 226
generator in \( \text{GF}(2^n) \) 888
generator of a group 772
generator, modulo \( p \) 453
generator, program producing programs 530
\( \text{GF}(2^n) \) (binary finite field) 885
\( \text{GF}2n \) (C++ class) 889, 909
\( \text{GF}2n::\text{init}() \) 889, 909

[fxtbook draft of 2008-August-17]
Index

gf2n_fast_trace() 887
gf2n_half_trace() 897
gf2n_order() 888
gf2n_solve_quadratic() 897
GNB (Gaussian normal basis) 913
Golay-Rudin-Shapiro sequence 46, 728
Goldschmidt algorithm 584
Gray code
  – and radix −2 representations, 64
  – binary, reversed, 47
  – combinatorial (minimal-change order), 171
  – constant, 739
  – for bit-subsets, 72
  – for combinations, 181
  – for combinations of a binary word, 78
  – for Fibonacci words, 81, 301
  – for Lyndon words, 394
  – for mixed radix numbers, 222
  – for multiset permutations, 295
  – for Pell words, 306, 756
  – for sparse signed binary words, 308
  – for subsets, with shifts-order, 208
  – of a binary word, 42
  – permutation, 103
  – powers of, 50
  – single track, 394
Gray permutation 103

gray_cycle_leaders (C++ class) 105
green code 52
grep 145
ground field 803, 885

Golay-Rudin-Shapiro sequence 46, 728

grs_negate() 464
gslex order, for mixed radix numbers 226

H
  .................................................................

haar() 394
Haar transform 493
haar_inplace() 495
haar_rev_n() 401
Hadamard matrix 873, 817
Hadamard transform 457
half-trace, in GF(2^n) with n odd 897
Halley’s formula 593, 596
Hamiltonian cycle 381
Hanoi, towers of, puzzle 733
Hartley shift 512
Hartley transform (HT) 511
hashing, via CRC 865
heap 154
Heap’s algorithm for permutations 249
heapsort 140
hexanacci numbers 304, 306

hidden constant, with asymptotics 552
high bits of a word, operations on 16
Hilbert curve
  – 3D, 359
  – by finite state machine, 164
  – by string substitution, 359
  – function encoding it, 744
  – moves, 57
  – turns, 59, 740
homogeneous moves, with combinations 182
homogenous moves, with k-subsets 216
Householder’s iteration 596
Householder’s method 592
HT (Hartley transform) 511
hybrid linear cellular automaton (LHCA) 875
hyperbolic sine and cosine, by CORDIC 663
hypercomplex numbers 815
hypergeometric function
  – (definition), 696
  – AGM algorithms, 608
  – conversion to continued fraction, 721

I
  ........................................................................

identical permutation 109
idsth() 530
i^l, computation 633
indecomposable permutation 289
index of an element modulo m 772
index of the single set bit 14
index sort 124
infinite products, from series 692
inhomogeneous recurrence 671
initial approximations, for iterations 578
integer partitions 331
integer sequence
  – Beatty seq. with Φ, 753
  – Carmichael numbers, 785
  – Catalan numbers, 324, 593
  – Euler function ϕ(n), 773
  – Euler numbers, 286
  – F-increment RGS, 355
  – Feigenbaum symbolic seq., 732
  – Fibbinary numbers, 671, 751
  – Fibonacci numbers, 304, 306, 750
  – fixed points in lex-rev seq., 76
  – Golay-Rudin-Shapiro seq., 46, 47, 728
  – Gray codes, 739
  – GRS (Golay-Rudin-Shapiro) seq., 46, 728
  – hexanacci numbers, 304, 306
  – hypercomplex multiplication, 817
  – indecomposable permutations, 287
  – integer partitions, 336
  – involutions, 285
  – irreducible polynomials, 840

[fxtbook draft of 2008-August-17]
- irreducible self-reciprocal polynomial, 843
- irreducible trinomials, 846
- K-increment RGS, 556
- Kronecker symbols \( \left( \frac{-1}{n} \right) \), 742
- Lyndon words, 340
- max-increment RGS, 355
- Mephisto Waltz seq., 727
- Moser – De Bruijn sequence, 64
- necklaces, 369
- non-generous primes, 777
- number of XYZ, see number of, XYZ
- optimal normal bases, type-1, 910
- optimal normal bases, type-2, 911
- paper-folding seq., 741
- paper-folding seq., signed, 742
- paren words, 84
- partitions into distinct parts, 338
- partitions, of an integer, 336
- Pell equation not solvable, 813
- pentanacci numbers, 304, 306
- period-doubling seq., 11, 732
- primes with primitive root 2, 850, 875
- primitive roots of Mersenne primes, 365
- primitive trinomials, 846
- quadratic residues all non-prime, 783
- rabbit seq., 507, 750
- radix \(-2\) representations, 65
- restricted growth strings, 326
- ruler function, 730
- sparse signed binary words, 309
- Stirling numbers of the second kind, 345
- subfactorial numbers, 287
- subset-lex words, 74
- sum of binary digits, 736
- sum of digits of binary Gray code, 741
- swaps with revbin permutation, 92
- tetranacci numbers, 304, 306
- Thue-Morse seq., 46, 460, 723, 817
- tribonacci numbers, 304, 306
- type-1 optimal normal bases, 910
- type-2 optimal normal bases, 911
- values of the Möbius function, 690
- Wieferich primes, 777

integer sequence, by OEIS number
- A000005, 691
- A000009, 338
- A000010, 773
- A000011, 136
- A000013, 136, 399
- A000029, 136
- A000031, 136, 369
- A000041, 304
- A000043, 796
- A000045, 304, 306, 308, 311, 313, 750
- A000048, 399, 845
- A000073, 304
- A000078, 304
- A000085, 285
- A000108, 324, 326
- A000110, 137, 343, 355
- A000111, 286
- A000123, 725
- A000129, 754
- A000166, 287
- A000201, 753
- A000213, 306
- A000288, 306
- A000296, 347
- A000322, 306
- A000383, 306
- A000695, 64
- A000700, 338
- A001037, 370, 841
- A001045, 309, 311
- A001122, 875
- A001200, 777
- A001262, 787
- A001333, 308, 754
- A001511, 730
- A001591, 304
- A001592, 304
- A001764, 326
- A002293, 326
- A002294, 326
- A002450, 64
- A002475, 848
- A002812, 796
- A002997, 189
- A003010, 796
- A003188, 739
- A003319, 287
- A003462, 664
- A003688, 308
- A003714, 671, 751, 752
- A003849, 761
- A004211, 355
- A004212, 355
- A004213, 355
- A005351, 65
- A005352, 65
- A005418, 136, 729
- A005578, 309
- A005614, 750
- A005727, 741
- A006130, 311
- A006131, 311
- A006206, 693
interior bit blocks, determination
interleaving process
for set partitions, 341
for Trotter’s permutations, 254
interpolation binary search 124
interpolation, linear 124
introsort 123
inverse
2-adic, 61
additive, modulo \( m \), 766
by exponentiation, 887
cube root, iteration for, 571
in \( \text{GF}(Q) \), 887
iteration for, 569
modulo \( m \), by exponentiation, 778
multiplicative, modulo \( m \), 766
of a circulant matrix, 904
permutation, 113
permutation, in-place computation, 113
power series over \( \text{GF}(2) \), 823
root, iteration for, 575
square root, iteration for, 570
XYZ transform, see XYZ transform
inverse_haar() 495
inversehaar_inplace() 496
inversion formula, Lagrange 594
inversion principle, Möbius 689
inversion table, of a permutation 237
invertible modulo m 766
involutions 113, 285
irreducible
  – polynomial, 834
  – trinomial, 846
is_cyclic() 112
is_quadratic_residue_2ex() 782
is_small_prime() 771
isolated ones or zeros in a word 12
iteration
  – and multiple roots, 597
  – divisionless, for polynomial roots, 589
  – for exp, 633
  – for inverse, 569
  – for inverse cube root, 571
  – for inverse root, 579
  – for inverse square root, 570
  – for logarithm, 629
  – for roots, p-adic, 571
  – for the zero of a function, 591
    – Goldschmidt, 584
    – Householder’s, 596
    – Schröder’s, 592
    – synthetic, 728
    – to compute π, 612
Ives’ algorithm for permutation generation 268

J
Jacobi matrix 548

K
Karatsuba multiplication
  – for integers, 552
  – for polynomials, 824
Komornik-Loreti constant 726
König iteration functions 596
kronecker() 781
Kronecker product
  – (definition), 461
    – of Hadamard matrices, 374
Kronecker symbol 781
ksubset_gray (C++ class) 213
ksubset_rec (C++ class) 210
ksubset_twoclose (C++ class) 210
Kummer’s transformation 703

L
Lagrange inversion formula 594
Lambert series 690, 735
LCM, least common multiple 767
left inversion, of a permutation 237
left-right array 160
left-right array, with Lehmer code 239
left-to-right powering 566
left_right_array (C++ class) 160
Legendre symbol 780
Legendre’s relation 607, 713
Lehmer code, of a permutation 236
lex (lexicographic) order 171
lexicographic order
  – (definition), 171
    – for bit combinations, 609
    – for combinations, 176
    – for multiset permutations, 292
    – for subsets, 201
    – for subsets of a binary word, 73
      – generalized, for mixed radix numbers, 226
lfsr (C++ class) 802
LFSR (linear feedback shift register) 861
LFSR, and Hadamard matrices 373
LHCA (linear hybrid cellular automaton) 875
lhca2poly() 877
lhca_next() 879
LIFO (last-in, first-out), stack 147
lin2hilbert() 164
linear convolution 140
linear correlation 142
linear feedback shift register (LFSR) 861
linear hybrid cellular automaton (LHCA) 875
linear interpolation 124
linear, function in a finite field 886
Lipski’s Gray codes for permutations 251
list recursions, for Gray codes 299
little endian machine 3
localized Hartley transform algorithm 725
localized Walsh transform algorithm 469
logarithm
  – bit-wise computation, 655, 658
  – computation by rectangular scheme, 648
  – computation via AGM, 627
  – computation via \pi/\log(q), 627
  – curious series for, 632
  – iteration using exp, 629
  – of power series, 636
loop in a graph 381
loopless algorithm 172
low bits, operations on 9
LR-array (left-right array) 160
Lucas test, for primality 798
Lucas-Lehmer test, for Mersenne numbers 796
lucky path, in a graph 393

[fxtbook draft of 2008-August-17]
Lunnon’s Gray code for multiset permutations 295
Lyndon words
  – (definition), 362
  – and irreducible binary polynomials, 852
  – and Mersenne primes, 365
  – binary, number of, 369
  – number of, 369
  – with fixed content, 372
  – with fixed density, 370
Lyndon words, as binary word 32
lyndon_gray (C++ class) 396
M
                        373
                        870
MAC (modular adjacent changes), Gray code 389
make_complement_shift_digraph() 387
make_complete_digraph() 382
make_oddprime_bitarray() 771
mass storage convolution 449
matrix Fourier algorithm (MFA) 434
matrix square root, applications 579
matrix transposition, and zip permutation 100
matrix transposition, in-place 96
maximal order modulo m 772
maxorder_element_mod() 770
mean, arithmetic-geometric 603
median of three elements 122
Mephisto Waltz sequence 727
Mersenne numbers, Lucas-Lehmer test 796
Mersenne primes
  – 2^j-th roots, 807
  – and Lyndon words, 365
  – generalized, 769
  – Lucas-Lehmer test, 796
Mersenne-Walsh transform 460
MFA (matrix Fourier algorithm) 434
minimal polynomial, in GF(2^n) 891
minimal-change order see GF(2^n) 891
minweight_lhca_rule() 876
missing, CPU instructions 88
mixed radix numbers 219
mixedradix_endo (C++ class) 229
mixedradix_endo_gray (C++ class) 230
mixedradix_gray (C++ class) 222
mixedradix_gslex (C++ class) 226
mixedradix_gslex_alt (C++ class) 228
mixedradix_lex (C++ class) 219
mixedradix_modular_gray (C++ class) 225
mixedradix_modular_gray2 (C++ class) 225
Möbius function 689
Möbius inversion principle 689
mod (C++ class) 537
mod m FFTs 535
modular adjacent changes (MAC), Gray code 389
modular arithmetic 763
modular multiplication 764
modular reduction, with structured primes 769
modular square root 783
modulo, as equivalence classes 134
modulus
  – composite, 773
  – prime, 772
  – prime, with NTTs, 536
moment conditions, for wavelet filters 546
monotone sequence 137
Moser – De Bruijn sequence 64
moves, of the Hilbert curve 57
mpartition (C++ class) 335
mset_perm_gray (C++ class) 295
mset_perm_lex (C++ class) 291
mset_perm_lex_rec (C++ class) 293
multi-dimensional Walsh transform 460
multi-point iteration 601
multigraph 381
multinomial coefficient 175
multiplication
  – by FFT, 560
  – carry, 561
  – integer vs. float, 6
  – is convolution, 560
  – Karatsuba, 552
  – modulo m, 764
  – of complex numbers via 3 real mult., 805
  – of hypercomplex numbers, 815
  – of octonions, 817
  – of polynomials, 441
  – of quaternions, 817
  – sum-of-digits test, 564
multiplication matrix, for normal bases 900
multiplication table, of an algebra 814
multiplicative function 689
multiplicative group 774
multiplicative group, with a ring 772
multiplicative inverse, modulo m 766
multiset 291
N
                        175
                        66
                        308
N-polynomial (normal polynomial) 899
n-set, a set with n elements 175
NAF (nonadjacent form) 66
NAF, Gray code 308
necklace (C++ class) 208
necklace2bitpol (C++ class) 852
necklaces
  – as equivalence classes, 135
  – binary, 81
Index

- binary, number of, 369
- definition, 361
- with fixed content, 372
- with fixed density, 370
necklaces, as binary words 32
negacyclic convolution 446, 525
neighbors of a node in a graph 381
Newton’s formula 894
Newton’s iteration, for vector-valued functions 548
node (vertex) of a graph 381
non-generous primes 777
nonadjacent form (NAF) 66
nonadjacent form (NAF), Gray code 308
normal bases, for GF(2^n) 899
normal basis, optimal 910
normal element (free element) 900
normal polynomial 899
normal_solve_reduced_quadratic() 901
NTT
- (number theoretic transforms), 535
- radix-2 DIF, 538
- radix-2 DIT, 537
- radix-4, 540
number of
- alternating permutation, 286
- alternating permutations, 286
- aperiodic necklaces, 369
- binary necklaces, 369
- binary partitions of even numbers, 725
- binary reversible strings, 136
- binary words at most r successive ones, 303
- bracelets, 136
- carries, 222
- combinations, 175
- cycles in De Bruijn graph, 387
- De Bruijn sequences, 872
- derangements, 287
- divisors, 691
- equivalence relations, 137
- F-increment RGS, 855
- fixed density Lyndon words, 370
- fixed density necklaces, 370
- generators modulo n, 770
- increment-i RGS, 326
- indecomposable permutations, 287
- integer partitions, 336
- integers coprime to n, 773
- invertible circulant matrices, 904
- involutions, 285
- irreducible polynomials, 840
- irreducible SRPs, 845
- K-increment RGS, 536
- Lyndon words, 369, 840
- m-sequences, 870
- max-increment RGS, 353
- necklaces, 136, 369
- normal polynomials, 902, 905
- ones in binary Gray code, 741
- parenthesis pairs, 324
- partitions into distinct parts, 338
- partitions of an integer, 336
- permutations of a multiset, 292
- permutations with m cycles, 283
- primitive normal polynomials, 903
- primitive polynomials, 841
- primitive SRPs, 846
- self-dual normal bases, 908
- shift register sequences, 870
- sparse signed binary words, 309
- strings with fixed content, 372
- swaps with revbin permutation, 92
- units in GF(Q), 887
- units modulo m, 774
- unlabeled bracelets, 136
- unlabeled necklaces, 136
number theoretic transforms (NTT) 535

O .........................................................
O(1) algorithm 172
octonions 815
ONB (optimal normal basis) 910
one-point iteration 691
optimal normal basis (ONB) 910
optimization, with combinatorial generation 172
OR-convolution 489
order
- of a polynomial, 838
- of an element modulo m, 771
- of an iteration, 591
out of core algorithms 449

P .........................................................
p-adic roots, iterations for 571
Padé approximants
- for arctan, 630
- for exp, 634
- for the logarithm, 629
paper-folding sequence 741
paren (C++ class) 317
paren_gray (C++ class) 323
parentheses, and binary words 83
parity
- number, 724, 736
- of a binary word, 44
- of a permutation, 112
parity (parity of a word), GCC builtin 22
Parseval’s equation 404

[fxtbook draft of 2008-August-17]
– of a set, 341
– of an integer, 331
partition (C++ class) 333
partition_rec (C++ class) 331
partitioning, for quicksort 122
Pascal’s triangle 176
path in a graph 381
pcrc64 (C++ class) 868
Pell
– constant, 756
– equation, 811
– Gray code constant, 758
– palindromic constant, 755
– ruler function, 756
Pell words, Gray code 306, 756
pentagonal number theorem 337, 661
pentanacci numbers 304, 306
pentanomial 849
Pepin’s test, for Fermat numbers 794
period of a polynomial 838
period-doubling constant 732
period-doubling sequence 11, 732
perm_colex (C++ class) 235
perm_derange (C++ class) 277
perm_gray_ffact (C++ class) 260
perm_gray_ffact2 (C++ class) 259
perm_gray_lipski (C++ class) 253
perm_gray_rfact (C++ class) 261
perm_gray_rot1 (C++ class) 264
perm_gray_wells (C++ class) 253
perm_heap (C++ class) 250
perm_heap2 (C++ class) 250
perm_heap2_swaps (C++ class) 251
perm_ives (C++ class) 265
perm_lex (C++ class) 234
perm_mvo (C++ class) 261
perm_rec (C++ class) 279
perm_restrpref (C++ class) 285
perm_rev (C++ class) 246
perm_rev2 (C++ class) 248
perm_rot (C++ class) 277
perm_st (C++ class) 271
perm_st_gray (C++ class) 273
perm_star (C++ class) 275
perm_star_swaps (C++ class) 275
perm_trotter (C++ class) 254
perm_trotter_lg (C++ class) 258
permutation
– alternating, 286
– as path in the complete graph, 385
– composition, 113
– connected, 286
– cycle type, 284
– cycles, 111
– cyclic, random, 119
– derangement, 287
– indecomposable, 286
– inverse of, 113
– involution, 113, 285
– of a multiset, 292
– random, 118
– with m cycles, number of, 283
Pfaff’s reflection law 700
phi function, number theoretic 773
π, computation 612
pitfall, shifts in C 5
pitfall, two’s complement 5
Pocklington-Lehmer test, for primality 793
pointer sort 126
pointer, size of 4
polar decomposition, of a matrix 580
polynomial
– binary, weight, 846
– irreducible, 834
– multiplication, 441
– multiplication, splitting schemes, 824
– primitive, 838
– roots, divisionless iterations for, 589
polynomial, binary 819
popcount (bit-count), GCC builtin 22
power series
– computation of exponential function, 637
– computation of logarithm, 636
– reversion, 635
powering
– algorithms, 565
– modulo m, 736
– of permutations, 115
– of the binary Gray code, 50
Pratt’s certificate of primality 791
prefix shifts, order for combinations 180
prev_lexrev() 473
prime length FFT, Rader’s algorithm 453
primes
– and cyclotomic polynomials, 801
– as modulus, 772
– as modulus, with NTTs, 536
– non-generous, 777
– sieve of Eratosthenes, 769
– structured, 769
– Wieferich, 777
– with primitive root 2, 875
primitive
– n-th root, modulo m, 535
– r-th root of unity, 772
– elements of a group, 772
– pentanomial, 849
– polynomial, 838
- root, 453, 772
- root in GF(2^n), 888
- root of Mersenne primes, 365
- trinomial, 846, 883
primitive root, Creutzburg-Tasche 807
print_cycles() 111
priority queue 155
priority_queue (C++ class) 156
product form
- for a-th root, 586
- for continued fractions, 718
- for elliptic K, 606
- for power series of exp, 637
- for square root, 685
products of k out of n factors 178
products, infinite, from series 692
Proth's theorem 794
Prouhet-Thue-Morse constant 724
pseudo graph 381
pseudo-inverse, of a matrix 582
pseudoprime 785
pseudoprime, strong (SPP) 785
Q-matrix 855
quadratic convergence 591
quadratic reciprocity 780
quadratic residue (square) modulo p 780
quadratic residues, and Hadamard matrices 375
quadruple reversion trick 98
quantization 132
quantize() 132
quartic convergence 591
quaternions 815
queue (C++ class) 150
quick sort 122
R
R2CFT see real FFT
R2CFT (real to complex FT) 426
rabbit constant 750
rabbit sequence 507, 750
Rabin's test for irreducibility 836
Rabin-Miller test, for compositeness 786
Rader's algorithm, for prime length FFT 453
radix –2 representations 63
radix (base) conversion 646
radix permutation 95
radix sort 142
radix_sort() 142
random permutation 118
random selection 119
ranking, with combinatorial objects 172
rational, square root iterations 572
re-orthogonalization, of a matrix 579
real FFT
- by FHT, 519
- split-radix algorithm, 430
- with wrap routines, 428
reciprocal polynomial 842
reciprocity, quadratic 780
rectangular scheme
- for arctan and log, 647
- for exp, sin, and cos, 649
recurrence
- (definition), 667
- inhomogeneous, 671
- relation, 687
- relation, for subsequences, 673
red code 52
reduction
- modular, with structured primes, 769
- modulo x^2 + x + 1 etc., 805
Reed-Muller transform
- (definition), 486
- and necklaces, 367
- convolution property, 490
relation, binary 133
relex order 171
representations, radix –2 63
representatives, with equivalence classes 134
restricted growth strings
- (definition), 319
- for k-ary trees, 325
- for parenthesis strings, 319
- for set partitions, 344
revbin
- (bit-wise reversal), 35
- constant, 738
- pairs, via shift registers, 870
- permutation, 91
- permutation, and convolution by FFT, 439
revbin_permute() 93
revbin_permute0() 94
reversal, of a permutation 110
reverse_0() 120
reversed Gray code 47
reversed Gray code permutation 107
reversing the bits of a word 35, 36
reversion of power series 593
reversion, of series, for Schröder's formula 595
RGS (restricted growth string) 319
rgs_binomial (C++ class) 329
rgs_fincr (C++ class) 353
rgs_maxincr (C++ class) 351
right inversion, of a permutation 237
right-angle convolution 446
right-to-left powering 565
ring buffer 149
ringbuffer (C++ class) 149
rising factorial basis 237

root
– p-adic, iterations for 571
– extraction 575
– inverse, iteration for 575
– of a polynomial, divisionless iterations 589
– primitive 772
– primitive, in GF(2^n) 888
– primitive, modulo m 535
– primitive, of Mersenne primes 365

rotate_left() 97
rotate_sgn() 488
rotation, bit-wise 29
rotation, by triple reversion 97
row-column algorithm 434
ruler constant 731
ruler function 206, 730
ruler_func (C++ class) 206, 289

S
Sande-Tukey FFT algorithm 409
scalar multiplication 885
Schröder’s formula 592
search, binary 123
searching, with unsorted arrays 143
secant method 591
sedenions 815
selection sort 121
self correlation 442
self-dual (basis over GF(2^n)) 906
self-reciprocal polynomial 843
semi-continuous Fourier transform 405
sentinel element 173
sequence see integer sequence
sequence 476
sequence, of a binary word 48
series reversion 593
series reversion, for Schröder’s formula 595
set partition 341
setpart (C++ class) 343
setpart_p_rgs_lex (C++ class) 348
setpart_rgs_gray (C++ class) 350
setpart_rgs_lex (C++ class) 347
setup_q_matrix() 856
shift operator, for Fourier transform 408
shift operator, for Hartley transform 512
shift register sequence (SRS) 861
shift-and-add algorithms 655
shifts in C, pitfall 6
shifts, and division 4
shifts-order
– for bit combinations 70
– for subsets 208
sieve of Eratosthenes 769
sign decomposition, of a matrix 581
sign of a permutation 112
sign of the Fourier transform 404
signed binary representation 66
signed binary words, sparse, Gray code 308
simple continued fraction 714
simple path in a graph 381
sine transform (DST) 529
sine, CORDIC algorithm 660
sine, in a finite field 808
single track
– binary Gray code 394
– order for permutations 271
– order for subsets 207
singular value decomposition (SVD) 580
skew circular convolution 446
slant transform 482
slant transform, sequence ordered 482
slow_convolution() 437
slow_fft() 404
smart, your compiler 28
sorting, edges in a graph 392
sorting, using a heap 140
space-filling Hilbert curve 359
sparse counting, and bit subsets 71
sparse signed binary representation 66
sparse signed binary words, Gray code 308
sparse words, bit counting 21
SPI (strong pseudo-irreducible) 837
split-radix FFT algorithm 421
splitting schemes for multiplication
– for integers 552
– for polynomials over GF(2) 825
splitting, binary, for rational series 641
SPP (strong pseudoprime) 785
sqrt_modf() 785
sqrt_modp() 783
sqrt_modpp() 784
square modulo p 780
square of a permutation 114
square root
– 2-adic 62
– in GF(2^n) 888
– iteration for 570
– modulo p 783
– of a matrix, applications 570
squarefree factorization, with polynomials 860
SRS (shift register sequence) 861
stable sort 141
stack (C++ class) 147

[fxtbook draft of 2008-August-17]
stack (LIFO) 147
stack, for coroutine emulation 166
star-transposition order, for permutations 275
state engine (finite state machine) 163
state engine, for coroutine emulation 166
state machine (finite state machine) 163
Stirling numbers
   – of the first kind (cycle numbers), 283, 707
   – of the second kind (set numbers), 345
strictly convex sequence 139
strictly monotone sequence 138
string substitution engine 357
strings with fixed content 372
strong minimal-change order 171, 254, 323
strong minimal-change order for combinations 182
strong pseudo-irreducible (SPI) 837
strong pseudoprime (SPP) 785
structured primes 769
subdegree of a polynomial 849
subfactorial numbers 287
subsequences, recurrence relations for 673
subset of bit-set, testing 24
subset_debruijn (C++ class) 207
subset_deltalex (C++ class) 202
subset_gray (C++ class) 205
subset_gray_delta (C++ class) 174, 204
subset_lex (C++ class) 202
subsets
   – of k bits (combinations), 67
   – of a binary word, 71, 73
   – of a multiset, 291
subtraction, modulo m 763
sum of Gray code digits constant 741
sum of two squares 809
sum-of-digits constant 737
sum-of-digits test, with multiplication 564
sumalt algorithm 651
sumdiff() 430, 465
super-linear iteration 591
SVD (singular value decomposition) 580
Swan’s theorem 847
swap without temporary 7
swap, conditional 26
swapping blocks via quadruple reversion 98
swapping two bits 9
symmetries, of revbin permutation 92
synthetic iterations 723

T
   tcrc64 (C++ class) 868
tensor product 461
tetranacci numbers 304, 306
theta functions: \( \Theta_2, \Theta_3 \), and \( \Theta_4 \) 627
Thue constant 728
Thue-Morse sequence 46, 460, 723, 817
thue_morse (C++ class) 46
timing, with demo-programs 174
TMFA (transposed matrix Fourier algorithm) 435
toggle between two values 6
Toom-Cook algorithm 553
Toom-Cook algorithm for binary polynomials 828
totient function 773
towers of Hanoi 733
trace
   – of a polynomial, 899
   – of an element in GF\((2^n)\), 887
   – vector, fast computation, 893
   – vector, in finite field, 887
trace-orthonormal basis 900
transformations, for elliptic K and E 711
transformations, of hypergeometric functions 699
transforms, on binary words 51
transition count, for a Gray code 394
transpose() 96
transposed matrix Fourier algorithm (TMFA) 435
transposed Reed-Muller transform 487
transposition of a matrix, and zip permutation 100
transposition of a matrix, in-place 90
transposition, with permutation 112
tree_gray (C++ class) 327
tribonacci numbers 304, 306
trigonometric recursion 412
trinomial
   – primitive, 846, 883
   – trace vector, 895
triple reversion trick 97
Trotter’s algorithm for permutations 254
two’s complement, pitfall 5
two-close order for k-subsets 216
two-close order for combinations 188
type, of a set partition 346
type-1 optimal normal basis 910
type-2 optimal normal basis 911
type-t Gaussian normal basis 913

U
   unique 130
   units (invertible elements) 766
   unlabeled bracelets 136
   unranking, with combinatorial objects 172
   unsorted arrays, searching 143
   unzip permutation 100
   unzip_rev() 102

V
   vertex, of a graph 381
   vertical addition 22
Index

**W**  ..............................................................
Walsh transform 457
Walsh transform, multi-dimensional 460
walsh_pal_basefunc() 476
walsh_q1() 480
walsh_q2() 480
walsh_wak_basefunc() 459
walsh_wak_dif2() 459, 486
walsh_wak_dit2() 458, 487
walsh_wal_basefunc() 477
walsh_wal_rev() 478
walsh_wal_rev_basefunc() 479
wavelet conditions 544
wavelet filter 544
wavelet transform 543
wavelet_filter (C++ class) 545, 547
weight, of binary polynomial 846
weighted convolution 446
weighted sum of Gray code digits constant 743
weighted sum-of-digits constant 738
weighted transform 445
Wells' Gray code for permutations 253
Whipple's identity 701
Wieferich primes 777

**X**  ..............................................................
XOR permutation 102
XOR, cyclic 34
XOR-convolution 473
xor_permute() 103
xrevbin() 57
x^n, series for 710

**Y**  ..............................................................
yellow code 51, 367

**Z**  ..............................................................
Z-order 59
z-transform 451
Zeckendorf representation 751
zero bytes, finding 60
zero divisors, of an algebra 815
zero padding, for linear convolution 440
zero-one transitions, determination 13
zip() 100
zip permutation 99
zip, bit-wise 40
zip_rev() 101
z^n, series for 710