Chapter 1

Bit wizardry

We give low-level functions for binary words, such as isolation of the lowest set bit or counting all set bits. Sometimes the term 'one' is used for a set bit and 'zero' for an unset bit. Where it cannot cause confusion, the term 'bit' is used for a set bit (as in "counting the bits of a word").

The C-type `unsigned long` is abbreviated as `ulong` as defined in [FXT: fxttypes.h]. It is assumed that `BITS_PER_LONG` reflects the size of an `unsigned long`. It is defined in [FXT: bits/bitsperlong.h] and usually equals the machine word size: 32 on 32-bit architectures, and 64 on 64-bit machines. Further, the quantity `BYTES_PER_LONG` reflects the number of bytes in a machine word: it equals `BITS_PER_LONG` divided by eight. For some functions it is assumed that `long` and `ulong` have the same number of bits.

Many functions will only work on machines that use two's complement, which is used by all of the current general purpose computers (the only machines using one's complement appear to be some successors of the UNIVAC system, see [358, entry "UNIVAC 1100/2200 series"]).

The examples of assembler code are for the x86 and the AMD64 architecture. They should be simple enough to be understood by 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 (*little endian* machine) or with the most significant byte (*big endian* machine). The hexadecimal number `0x0D0C0B0A` will be stored in the following manner if memory addresses grow from left to right:

```
adr:   z   z+1   z+2  z+3
mem:   0D   0C   0B   0A    // big endian
mem:   0A   0B   0C   0D    // little endian
```

The difference becomes visible when you cast pointers. Let `V` be the 32-bit integer with the value above. Then the result of `char c = *(char *)(&V);` will be `0x0A` (value modulo 256) on a little endian machine but `0x0D` (value divided by 2^{24}) on a big endian machine. Though friends of big endian 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, as with transfer over a network or to a disk, one will need two code versions, one for big endian and one for little endian machines. The C-type `union` (with words and bytes) may also require separate treatment for big and little endian architectures.

1.1.2 Size of pointer is not size of int

If programming for a 32-bit architecture (where the size of `int` and `long` coincide), casting pointers to integers (and back) will usually work. The same code *will* fail on 64-bit machines. If you have to cast pointers to an integer type, cast them to a sufficiently big type. For portable code it is better to avoid casting pointers to integer types.

DOI 10.1007/978-3-642-14764-7_1, © Springer-Verlag Berlin Heidelberg 2011

1.1.3 Shifts and division

With two's complement arithmetic division and multiplication by a power of 2 is a 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 2) that rounds to $-\infty$:

```
    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:

```
   9:test.cc   @ int foo(int a)
  10:test.cc   @ {
 285 0003 8B442410              movl 16(%esp),%eax  // move argument to %eax
  11:test.cc   @     int s = a >> 1;
 289 0007 89C1                  movl %eax,%ecx
 290 0009 D1F9                  sarl $1,%ecx
  12:test.cc   @     int d = a / 2;
 293 000b 89C2                  movl %eax,%edx
 294 000d C1EA1F                shrl $31,%edx  // fix: %edx=(%edx<0?1:0)
 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:

```
# 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 *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 2 with unsigned types is equivalent to a bit-and:

```
    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 compiler does it for you. A division by the constant 10 is compiled to:

```
   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 == 0xcccccccd
 292 000a 89D0                  movl %edx,%eax
 293 000c C1E803                shrl $3,%eax
```

Therefore 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:

```
   8:test.cc   @ ulong foo(ulong a)
   9:test.cc   @ {
  53 0000 8B4C2404              movl 4(%esp),%ecx
  10:test.cc   @     ulong b = a % 10000;
  57 0004 89C8                  movl %ecx,%eax
  58 0006 F7250000              mull .LC0   // value == 0xd1b71759
  59 000c 89D0                  movl %edx,%eax
  60 000e C1E80D                shrl $13,%eax
  61 0011 69C01027              imull $10000,%eax,%eax
  62 0017 29C1                  subl %eax,%ecx
  63 0019 89C8                  movl %ecx,%eax
```

Algorithms to replace divisions by a constant with multiplications and shifts are given in [168], see also [346].

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 default behavior of many commonly used C compilers.

1.1.4 A pitfall (two’s complement)

```
 c=................  -c=................   c=     0  -c=     0  <--=
 c=...............1  -c=1111111111111111   c=     1  -c=    -1
 c=..............1.  -c=111111111111111.   c=     2  -c=    -2
 c=..............11  -c=11111111111111.1   c=     3  -c=    -3
 c=.............1..  -c=11111111111111..   c=     4  -c=    -4
 c=.............1.1  -c=1111111111111.11   c=     5  -c=    -5
 c=.............11.  -c=1111111111111.1.   c=     6  -c=    -6
[--snip--]
 c=.1111111111111.1  -c=1.............11   c= 32765  -c=-32765
 c=.11111111111111.  -c=1.............1.   c= 32766  -c=-32766
 c=.111111111111111  -c=1..............1   c= 32767  -c=-32767
 c=1...............  -c=1...............   c=-32768  -c=-32768  <--=
 c=1..............1  -c=.111111111111111   c=-32767  -c= 32767
 c=1.............1.  -c=.11111111111111.   c=-32766  -c= 32766
 c=1.............11  -c=.1111111111111.1   c=-32765  -c= 32765
 c=1............1..  -c=.1111111111111..   c=-32764  -c= 32764
 c=1............1.1  -c=.111111111111.11   c=-32763  -c= 32763
 c=1............11.  -c=.111111111111.1.   c=-32762  -c= 32762
[--snip--]
 c=1111111111111..1  -c=.............111   c=    -7  -c=     7
 c=1111111111111.1.  -c=.............11.   c=    -6  -c=     6
 c=1111111111111.11  -c=.............1.1   c=    -5  -c=     5
 c=11111111111111..  -c=.............1..   c=    -4  -c=     4
 c=11111111111111.1  -c=..............11   c=    -3  -c=     3
 c=111111111111111.  -c=..............1.   c=    -2  -c=     2
 c=1111111111111111  -c=...............1   c=    -1  -c=     1
```

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 16 bits. This is why innocent looking code like the following can simply fail:

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

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:

```
static 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

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

has to be inserted just before the return statement.

1.1.6 Shortcuts

Test whether at least one of a and b equals zero with

```
if ( !(a && b) )
```

This works for both signed and unsigned integers. Check whether both are zero with

```
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 Average without overflow

A routine for the computation of the average $(x+y)/2$ of two arguments x and y is [FXT: bits/average.h]

```
static inline ulong average(ulong x, ulong y)
// Return floor( (x+y)/2 )
// Use:      x+y == ((x&y)<<1) + (x^y)
// that is:  sum ==  carries   + sum_without_carries
{
    return  (x & y) + ((x ^ y) >> 1);
}
```

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 we can use the simpler statement `return y+(x-y)/2`. The following version rounds to infinity:

```
static inline ulong ceil_average(ulong x, ulong y)
// Use:      x+y == ((x|y)<<1) - (x^y)
// ceil_average(x,y) == average(x,y) + ((x^y)&1))
{
    return  (x | y) - ((x ^ y) >> 1);
}
```

1.1.8 Toggling between values

To toggle an integer `x` between two values `a` and `b`, use:

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

The equivalent trick for floating-point types is

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

Here an overflow could occur with `a` and `b` in the allowed range if both are close to overflow.

1.1.9 Next or previous even or odd value

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

```
static inline ulong next_even(ulong x)  { return x+2-(x&1); }
static inline ulong prev_even(ulong x)  { return x-2+(x&1); }

static inline ulong next_odd(ulong x)  { return x+1+(x&1); }
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:

```
static inline ulong next0_even(ulong x)  { return x+(x&1); }
static inline ulong prev0_even(ulong x)  { return x-(x&1); }

static inline ulong next0_odd(ulong x)  { return x+1-(x&1); }
static inline ulong prev0_odd(ulong x)  { return x-1+(x&1); }
```

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

```
static inline ulong next_even(ulong x)  { return (x|1)+1; }
static inline ulong prev_even(ulong x)  { return (x-1)&~1; }

static inline ulong next_odd(ulong x)  { return (x+1)|1; }
static inline ulong prev_odd(ulong x)  { return (x&~1)-1; }
```

```
static inline ulong next0_even(ulong x)  { return (x+1)&~1; }
static inline ulong prev0_even(ulong x)  { return x&~1; }

static inline ulong next0_odd(ulong x)  { return x|1; }
static inline ulong prev0_odd(ulong x)  { return (x-1)|1; }
```

1.1.10 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 111111111 using a 32-bit integer type and floating-point types with 24-bit and 53-bit mantissa (significand):

```
  a =  111111111         // assignment
a*a == 12345678987654321 // true result

a*a == 1653732529            // result with 32-bit integer multiplication
(a*a)%(2**32) == 1653732529  // ... which is modulo (2**bits_per_int)

a*a == 1.2345679481405440e+16 // result with float multiplication (24 bit mantissa)
a*a == 1.2345678987654320e+16 // result with float multiplication (53 bit mantissa)
```

1.1.11 Double precision float to signed integer conversion

Conversion of double precision floats that have a 53-bit mantissa to signed integers via [11, p.52-53]

```
#define DOUBLE2INT(i, d)  { double t = ((d) + 6755399441055744.0); i = *((int *)(&t)); }
double x = 123.0;
int i;
DOUBLE2INT(i, x);
```

can be a faster alternative to

```
double x = 123.0;
int 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, the floating-point number has a 53-bit mantissa with the most significant bit (that is always one with normalized numbers) omitted, and that the address of the number points to the mantissa.

1.1.12 Optimization considerations

Never assume that some code is the 'fastest possible'. There is always another trick that can still improve performance. Many factors can have an influence on performance, like the 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 is pretty much out of fashion today:

```
          //   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:  tmp = a;  a = b;  b = tmp;
```

However, under some conditions (like extreme register pressure) it may be the way to go. Note that if both operands are identical (memory locations) then the result is zero.

The only way to find out which version of a function is faster is to actually do benchmarking (timing). The performance does depend on the sequence of instructions surrounding the machine code, assuming that all of these low-level functions get inlined. Studying the generated CPU instructions helps to understand what happens, but can never replace benchmarking. This means that benchmarks for just the isolated routine can at best give a rough indication. Test your application using different versions of the routine in question.

Never ever delete the unoptimized version of some code fragment when introducing a streamlined one. Keep the original in the source. If something nasty happens (think of low level software failures when porting to a different platform), you will be *very* grateful for the chance to temporarily resort to the slow but correct version.

Study the optimization recommendations for your CPU (like [11] and [12] for the AMD64, see also [144]). You can also learn a lot from the documentation for other architectures.

Proper documentation is an absolute must for optimized code. Always assume that nobody will understand the code without comments. You may not be able to understand uncommented code written by yourself after enough time has passed.

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 than or equal to `BITS_PER_LONG`, the result is undefined [FXT: bits/bittest.h]:

```
static 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:

```
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:

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

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

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

1.2.2 Copying a bit

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

```
static 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
    return  a;
}
```

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

```
static 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.
{
    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, or 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]:

```
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;
}
```

If it is known that the bits do have different values, the following routine should be used:

```
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 operating 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 functions are given in [FXT: bits/bitlow.h].

1.3.1 Isolating, setting, and deleting the lowest one

The lowest one (set bit) is isolated via

```
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) is isolated using the equivalent of `lowest_one( ~x )`:

```
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, we can use either of

```
return  (x ^ (x+1)) & ~x;
return  ((x ^ (x+1)) >> 1 ) + 1;
```

The sequence of returned values for $x = 0, 1, \ldots$ is the highest power of 2 that divides $x + 1$, entry A006519 in [312] (see also entry A001511):

```
  x:  ==   x        lowest_zero(x)
  0:  == ........    .......1
  1:  == .......1    ......1.
  2:  == ......1.    .......1
  3:  == ......11    .....1..
  4:  == .....1..    .......1
  5:  == .....1.1    ......1.
  6:  == .....11.    .......1
  7:  == .....111    ....1...
  8:  == ....1...    .......1
  9:  == ....1..1    ......1.
 10:  == ....1.1.    .......1
```

The lowest set bit in a word can be cleared by

```
static inline ulong clear_lowest_one(ulong x)
// Return word where the lowest bit set in x is cleared.
// Return 0 for input == 0.
{
    return  x & (x-1);
}
```

The lowest unset bit can be set by

```
static inline ulong set_lowest_zero(ulong x)
// Return word where the lowest unset bit in x is set.
// Return ~0 for input == ~0.
{
    return  x | (x+1);
}
```

1.3.2 Computing the index of the lowest one

We compute the index (position) of the lowest bit with an assembler instruction if available [FXT: bits/bitasm-amd64.h]:

```
static inline ulong asm_bsf(ulong x)
// Bit Scan Forward
{
    asm ("bsfq %0, %0" : "=r" (x) : "0" (x));
    return x;
}
```

Without the assembler instruction an algorithm that involves $O\left(\log_2 \texttt{BITS\_PER\_LONG}\right)$ operations can be used. The function can be implemented as follows (suggested by Nathan Bullock [priv. comm.], 64-bit version) [FXT: bits/bitlow.h]:

```
static inline ulong lowest_one_idx(ulong x)
// Return index of lowest bit set.
// Examples:
//    ***1 --> 0
//    **10 --> 1
//    *100 --> 2
// Return 0 (also) if no bit is set.
{
    ulong r = 0;
    x &= -x;  // isolate lowest bit
    if ( x & 0xffffffff00000000UL )  r += 32;
    if ( x & 0xffff0000ffff0000UL )  r += 16;
    if ( x & 0xff00ff00ff00ff00UL )  r += 8;
    if ( x & 0xf0f0f0f0f0f0f0f0UL )  r += 4;
    if ( x & 0xccccccccccccccccUL )  r += 2;
    if ( x & 0xaaaaaaaaaaaaaaaaUL )  r += 1;
    return r;
}
```

The function returns zero for two inputs, one and zero. If a special value for the input zero is needed, a statement as the following should be added as the first line of the function:

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

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

```
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

```
001000101010001000100010101000101010001010100010001000101010001 0...
```

This is the complement of the *period-doubling sequence*, entry A035263 in [312]. See section 38.5.1 on page 735 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 [FXT: bits/bitlow.h]:

```
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:

```
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;
}
```

The lowest block of ones (which may have zeros to the right of it) can be isolated by

```
static inline ulong lowest_block(ulong x)
// Isolate lowest block of ones.
// e.g.:
// x   = *****011100
// l   = 00000000100
// y   = *****100000
// x^y = 00000111100
// ret = 00000011100
{
    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

Use the following routines to set a rising or falling edge at the position of the lowest set bit [FXT: bits/bitlow-edge.h]:

```
static inline ulong lowest_one_10edge(ulong x)
// Return word where all bits from (including) the
//   lowest set bit to most significant bit are set.
// Return 0 if no bit is set.
// Example:  00110100 --> 11111100
{
    return ( x | -x );
}
```

```
static inline ulong lowest_one_01edge(ulong x)
// Return word where all bits from  (including) the
//   lowest set bit to the least significant are set.
// Return 0 if no bit is set.
// Example:  00110100 --> 00000111
```

```
{
    if ( 0==x )  return 0;
    return  x^(x-1);
}
```

1.3.5 Isolating the lowest run of matching bits

Let $x = *0W$ and $y = *1W$, the following function computes W:

```
static inline ulong low_match(ulong x, ulong y)
{
    x ^= y;   // bit-wise difference
    x &= -x;  // lowest bit that differs in both words
    x -= 1;   // mask that covers equal bits at low end
    x &= y;   // isolate matching bits
    return x;
}
```

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. 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].

```
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

```
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

The following functions are given in [FXT: bits/bit-isolate.h]:

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

We can assume a word is embedded in zeros or ignore the bits outside the word:

```
static inline ulong single_zeros_xi(ulong x)
// Return word with only the isolated zeros of x set.
{
    return  single_ones( ~x );  // ignore outside values
}
```

```
static inline ulong single_zeros(ulong x)
// Return word with only the isolated zeros of x set.
{
    return  ~x & ( (x<<1) & (x>>1) );  // assume outside values == 0
}
```

```
static inline ulong single_values(ulong x)
// Return word where only the isolated ones and zeros of x are set.
{
    return  (x ^ (x<<1)) & (x ^ (x>>1));  // assume outside values == 0
}
```

```
static inline ulong single_values_xi(ulong x)
// Return word where only the isolated ones and zeros of x are set.
{
    return  single_ones(x) | single_zeros_xi(x);  // ignore outside values
}
```

1.4.3 Isolating single ones or zeros at the word boundary

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

```
static inline ulong border_values(ulong x)
// Return word where those bits of x are set that lie on a transition.
{
    return  (x ^ (x<<1)) | (x ^ (x>>1));
}
```

1.4.4 Isolating transitions

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

```
static inline ulong low_border_ones(ulong x)
// Return word where only those ones of x are set
//   that lie left to (i.e. in the next higher bin of) a zero.
{
    return  x & ( x ^ (x<<1) );
}
```

1.4.5 Isolating ones or zeros at block boundaries

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

```
static inline ulong low_block_border_ones(ulong x)
// 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.
{
    ulong t = x & ( (x<<1) ^ (x>>1) );  // block_border_ones()
    return  t & (x>>1);
}
```

```
static inline ulong high_block_border_ones(ulong x)
// 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.
{
    ulong t = x & ( (x<<1) ^ (x>>1) );  // block_border_ones()
    return  t & (x<<1);
}
```

```
static inline ulong block_ones(ulong x)
// Return word where only those bits of x are set
//   that are part of a block of at least 2 bits.
{
    return  x & ( (x<<1) | (x>>1) );
}
```

1.5 Computing the index of a single set bit

In the function `lowest_one_idx()` given in section 1.3.2 on page 9 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 compute 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

```
modulus m=11
    k =  0  1  2  3  4  5  6  7
mt[k]=  0  0  1  8  2  4  9  7

Lowest bit == 0:       x= .......1 =   1    x % m=  1 ==> lookup = 0
Lowest bit == 1:       x= ......1. =   2    x % m=  2 ==> lookup = 1
Lowest bit == 2:       x= .....1.. =   4    x % m=  4 ==> lookup = 2
Lowest bit == 3:       x= ....1... =   8    x % m=  8 ==> lookup = 3
Lowest bit == 4:       x= ...1.... =  16    x % m=  5 ==> lookup = 4
Lowest bit == 5:       x= ..1..... =  32    x % m= 10 ==> lookup = 5
Lowest bit == 6:       x= .1...... =  64    x % m=  9 ==> lookup = 6
Lowest bit == 7:       x= 1....... = 128    x % m=  7 ==> lookup = 7
```

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

A nice trick is presented in [110]: for N-bit words find a number m such that all powers of 2 are different modulo m. That is, the (multiplicative) order of 2 modulo m must be greater than or equal to N. We use a table `mt[]` of size m that contains the power of 2: `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.

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

```
const ulong m = 11; // the modulus
ulong mt[m+1];
static void mt_setup()
{
    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. We can use any value to be returned for input zero. Here we simply use zero to always have the same return value as with `lowest_one_idx()`. The index can be computed by

```
static inline ulong m_lowest_one_idx(ulong x)
{
    x &= -x;  // isolate lowest bit
    x %= m;   // power of 2 modulo m
    return  mt[x];  // lookup
}
```

The code is given in the program [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 $m(N)$.

```
db=...1.111  (De Bruijn sequence)
    k  =  0  1  2  3  4  5  6  7
dbt[k] =  0  1  2  4  7  3  6  5

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 = .....1.1 == 5 ==> lookup = 3
Lowest bit == 4:    x = ...1....   db * x = .111....   shifted = ......11 == 3 ==> lookup = 4
Lowest bit == 5:    x = ..1.....   db * x = 111.....   shifted = .....111 == 7 ==> lookup = 5
Lowest bit == 6:    x = .1......   db * x = 11......   shifted = .....11. == 6 ==> lookup = 6
Lowest bit == 7:    x = 1.......   db * x = 1.......   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.

1.5.2 Using De Bruijn sequences

The following method (given in [228]) 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 41.1 on page 864. These are the sequences for 32 and 64 bit, as binary words:

```
#if BITS_PER_LONG == 32
const ulong db = 0x4653ADFUL;
// == 00000100011001010011101011011111
const ulong s = 32-5;
#else
const ulong db = 0x218A392CD3D5DBFUL;
// == 0000001000011000101000111001001011001101001111010101110110111111
const ulong s = 64-6;
#endif
```

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 dbt[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:

```
static inline ulong db_lowest_one_idx(ulong x)
{
    x &= -x;  // isolate lowest bit
    x *= db;  // multiplication by a power of 2 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 set. Therefore if we convert an integer into a float, 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 with 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 functions operating on the highest bit there is no method as trivial as shown for 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.

1.6.1 Isolating the highest one and finding its index

```
................1111....1111.111 = 0xf0f7  == word
................1............... = highest_one
................1111111111111111 = highest_one_01edge
11111111111111111............... = highest_one_10edge
                              15 = highest_one_idx
................................ = low_zeros
.............................111 = low_ones
...............................1 = lowest_one
...............................1 = lowest_one_01edge
11111111111111111111111111111111 = lowest_one_10edge
                               0 = lowest_one_idx
.............................111 = lowest_block
................1111....1111.11. = clear_lowest_one
............................1... = lowest_zero
................1111....11111111 = set_lowest_zero
................................ = high_ones
1111111111111111................ = high_zeros
1............................... = highest_zero
1...............1111....1111.111 = set_highest_zero
```

```
1111111111111111....1111....1... = 0xffff0f08  == word
1............................... = highest_one
11111111111111111111111111111111 = highest_one_01edge
1............................... = highest_one_10edge
                              31 = highest_one_idx
.............................111 = low_zeros
................................ = low_ones
............................1... = lowest_one
............................1111 = lowest_one_01edge
11111111111111111111111111111... = lowest_one_10edge
                               3 = lowest_one_idx
............................1... = lowest_block
1111111111111111....1111........ = clear_lowest_one
...............................1 = lowest_zero
1111111111111111....1111....1..1 = set_lowest_zero
1111111111111111................ = high_ones
................................ = high_zeros
................1............... = highest_zero
11111111111111111...1111....1... = set_highest_zero
```

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.

Isolation of the highest set bit is easy if a bit-scan instruction is available [FXT: bits/bitasm-i386.h]:

```
static inline ulong asm_bsr(ulong x)
// Bit Scan Reverse
{
    asm ("bsrl %0, %0" : "=r" (x) : "0" (x));
    return x;
}
```

Without a bit-scan instruction, we use the auxiliary function [FXT: bits/bithigh-edge.h]

```
static inline ulong highest_one_01edge(ulong x)
// Return word where all bits from (including) the
//   highest set bit to bit 0 are set.
// Return 0 if no bit is set.
{
    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 resulting code is [FXT: bits/bithigh.h]

```
static inline ulong highest_one(ulong x)
// Return word where only the highest bit in x is set.
// Return 0 if no bit is set.
{
#if defined  BITS_USE_ASM
    if ( 0==x )  return 0;
    x = asm_bsr(x);
    return  1UL<<x;
#else
    x = highest_one_01edge(x);
    return  x ^ (x>>1);
#endif // BITS_USE_ASM
}
```

To determine the index of the highest set bit, use

```
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

    if ( 0==x )  return  0;

    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
}
```

The branches in the non-assembler part of the routine can be avoided by a technique given in [215, rel.96, sect.7.1.3] (version for 64-bit words):

```
static inline ulong highest_one_idx(ulong x)
{
#define MU0 0x5555555555555555UL  // MU0 == ((-1UL)/3UL) == ...01010101_2
#define MU1 0x3333333333333333UL  // MU1 == ((-1UL)/5UL)   == ...00110011_2
#define MU2 0x0f0f0f0f0f0f0f0fUL  // MU2 == ((-1UL)/17UL)  == ...00001111_2
#define MU3 0x00ff00ff00ff00ffUL  // MU3 == ((-1UL)/257UL)  == (8 ones)
#define MU4 0x0000ffff0000ffffUL  // MU4 == ((-1UL)/65537UL) == (16 ones)
#define MU5 0x00000000ffffffffUL  // MU5 == ((-1UL)/4294967297UL) == (32 ones)
    ulong r = ld_neq(x, x & MU0)
        + (ld_neq(x, x & MU1) << 1)
        + (ld_neq(x, x & MU2) << 2)
        + (ld_neq(x, x & MU3) << 3)
        + (ld_neq(x, x & MU4) << 4)
        + (ld_neq(x, x & MU5) << 5);
    return r;
}
```

The auxiliary function `ld_neq()` is given in [FXT: bits/bitldeq.h]:

```
static inline bool ld_neq(ulong x, ulong y)
// Return whether floor(log2(x))!=floor(log2(y))
{ return  ( (x^y) > (x&y) ); }
```

The following version for 64-bit words provided by Sebastiano Vigna [priv. comm.] is an implementation of Brodal's algorithm [215, alg.B, sect.7.1.3]:

```
static inline ulong highest_one_idx(ulong x)
{
    if ( x == 0 )  return 0;
    ulong r = 0;
    if ( x & 0xffffffff00000000UL )  { x >>= 32;  r += 32; }
    if ( x & 0xffff0000UL )          { x >>= 16;  r += 16; }
    x |= (x << 16);
    x |= (x << 32);
    const ulong y = x & 0xff00f0f0ccccaaaaUL;
    const ulong z = 0x8000800080008000UL;
    ulong t = z & ( y | (( y | z ) - ( x ^ y )));
    t |= (t << 15);
    t |= (t << 30);
    t |= (t << 60);
    return  r + ( t >> 60 );
}
```

1.6.2 Isolating the highest block of ones or zeros

Isolate the left block of zeros with the function

```
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 left block of ones can be isolated using arithmetical right shifts:

```
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, use

```
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]. Part of its 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:

```
static inline ulong ld(ulong x)
// Return floor(log2(x)),
// i.e. 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()`:

```
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.

Use the function `one_bit_q()` to determine whether its argument is a power of 2:

```
static inline bool one_bit_q(ulong x)
// Return whether x \in {1,2,4,8,16,...}
{
    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:

```
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 power of 2 the following functions are useful:

```
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;
}
```

```
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()`:

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

The following routine for comparison of base-2 logarithms without actually computing them is suggested by [215, rel.58, sect.7.1.3] [FXT: bits/bitldeq.h]:

```
static inline bool ld_eq(ulong x, ulong y)
// Return whether floor(log2(x))==floor(log2(y))
{ return  ( (x^y) <= (x&y) ); }
```

1.8 Counting the bits and blocks of a word

The following functions count the ones in a binary word. They need $O\left(\log_2(\texttt{BITS\_PER\_LONG})\right)$ operations. We give mostly the 64-bit versions [FXT: bits/bitcount.h]:

```
static inline ulong bit_count(ulong x)
// Return number of bits set
{
    x = (0x5555555555555555UL & x) + (0x5555555555555555UL & (x>> 1));  // 0-2 in 2 bits
    x = (0x3333333333333333UL & x) + (0x3333333333333333UL & (x>> 2));  // 0-4 in 4 bits
    x = (0x0f0f0f0f0f0f0f0fUL & x) + (0x0f0f0f0f0f0f0f0fUL & (x>> 4));  // 0-8 in 8 bits
    x = (0x00ff00ff00ff00ffUL & x) + (0x00ff00ff00ff00ffUL & (x>> 8));  // 0-16 in 16 bits
    x = (0x0000ffff0000ffffUL & x) + (0x0000ffff0000ffffUL & (x>>16));  // 0-32 in 32 bits
    x = (0x00000000ffffffffUL & x) + (0x00000000ffffffffUL & (x>>32));  // 0-64 in 64 bits
    return x;
}
```

The underlying idea is to do a search via bit masks. The code can be improved to either

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

or (taken from [10])

```
    x -=  (x>>1) & 0x5555555555555555UL;                                 // 0-2 in 2 bits
    x  = ((x>>2) & 0x3333333333333333UL) + (x & 0x3333333333333333UL);  // 0-4 in 4 bits
    x  = ((x>>4) + x) & 0x0f0f0f0f0f0f0f0fUL;                           // 0-8 in 8 bits
    x *= 0x0101010101010101UL;
    return  x>>56;
```

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

The following code for 32-bit words (given by Johan Rönnblom [priv. comm.]) may be advantageous if loading constants is expensive. Note some constants are in octal notation:

```
static inline uint CountBits32(uint a)
{
    uint mask = 011111111111UL;
    a = (a - ((a&~mask)>>1)) - ((a>>2)&mask);
    a += a>>3;
    a = (a & 070707) + ((a>>18) & 070707);
    a *= 010101;
    return  ((a>>12) & 0x3f);
}
```

If the table holds the bit-counts of the numbers 0...255, then the bits can be counted as follows:

```
ulong bit_count(ulong x)
{
    unsigned char ct = 0;
    ct += tab[ x & 0xff ];   x >>= 8;
    ct += tab[ x & 0xff ];   x >>= 8;
  [--snip--]  /* BYTES_PER_LONG times */
    ct += tab[ x & 0xff ];
    return ct;
}
```

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

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

```
static inline ulong bit_count_01(ulong x)
// Return number of bits in a word
// for words of the special form 00...0001...11
{
    ulong ct = 0;
    ulong a;
#if  BITS_PER_LONG == 64
    a = (x & (1UL<<32)) >> (32-5);  // test bit 32
    x >>= a;  ct += a;
#endif
    a = (x & (1UL<<16)) >> (16-4);  // test bit 16
    x >>= a;  ct += a;

    a = (x & (1UL<<8)) >> (8-3);  // test bit 8
    x >>= a;  ct += a;

    a = (x & (1UL<<4)) >> (4-2);  // test bit 4
    x >>= a;  ct += a;

    a = (x & (1UL<<2)) >> (2-1);  // test bit 2
    x >>= a;  ct += a;

    a = (x & (1UL<<1)) >> (1-0);  // test bit 1
    x >>= a;  ct += a;

    ct += x & 1; // test bit 0

    return ct;
}
```

All branches are avoided, thereby the code may be useful on a planet with pink air, for further details see [301].

1.8.1 Sparse counting

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

```
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:

```
    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, use the given routine with the complement:

```
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 );
}
```

If the number of ones is guaranteed to be less than 16, then the following routine (suggested by Gunther Piez [priv. comm.]) can be used:

```
static inline ulong bit_count_15(ulong x)
// Return number of set bits, must have at most 15 set bits.
{
    x -=  (x>>1) & 0x5555555555555555UL;                              // 0-2 in 2 bits
    x  = ((x>>2) & 0x3333333333333333UL) + (x & 0x3333333333333333UL);  // 0-4 in 4 bits
    x *= 0x1111111111111111UL;
    return  x>>60;
}
```

A routine for words with no more than 3 set bits is

```
static inline ulong bit_count_3(ulong x)
{
    x -=  (x>>1) & 0x5555555555555555UL;  // 0-2 in 2 bits
    x *= 0x5555555555555555UL;
    return  x>>62;
}
```

1.8.2 Counting blocks

Compute the number of bit-blocks in a binary word with the following function:

```
static inline ulong bit_block_count(ulong x)
// Return number of bit blocks.
// E.g.:
// ..1..11111...111.  -> 3
// ...1..11111...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:

```
static inline ulong bit_block_ge2_count(ulong x)
// Return number of bit blocks with at least 2 bits.
// E.g.:
// ..1..11111...111.  -> 2
// ...1..11111...111  -> 2
// ......1.....1.1..  -> 0
// .........111.1111  -> 2
{
    return  bit_block_count( x & ( (x<<1) & (x>>1) ) );
}
```

1.8.3 GCC built-in functions ‡

Newer versions of the C compiler of the GNU Compiler Collection (GCC [146], starting with version 3.4) include a function `__builtin_popcountl(ulong)` that counts the bits of an unsigned long integer. The following list is taken from [147]:

```
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 the corresponding versions for arguments of type unsigned long are obtained by adding '`l`' (ell) to the names, for the type unsigned long long append '`ll`'. Two more useful built-ins are:

```
void __builtin_prefetch (const void *addr, ...)
    Prefetch memory location addr

long __builtin_expect (long exp, long c)
    Function to provide the compiler with branch prediction information.
```

1.8.4 Counting the 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=11111111  a2=........  a3=11111111  a4=........
x[10]=11111111  a0=11111111  a1=11111111  a2=........  a3=11111111  a4=........
x[11]=11111111  a0=........  a1=........  a2=11111111  a3=11111111  a4=........
x[12]=11111111  a0=11111111  a1=........  a2=11111111  a3=11111111  a4=........
x[13]=11111111  a0=........  a1=11111111  a2=11111111  a3=11111111  a4=........
x[14]=11111111  a0=11111111  a1=11111111  a2=11111111  a3=11111111  a4=........
x[15]=11111111  a0=........  a1=........  a2=........  a3=........  a4=11111111
x[16]=11111111  a0=11111111  a1=........  a2=........  a3=........  a4=11111111
```

Figure 1.8-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 now replace this 'horizontal' propagation by a 'vertical' one, that is, propagation into another word. An implementation of this idea is [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,    3,    7,   15,   31,  <--= max n
    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;
}
```

Figure 1.8-A shows the intermediate values with the computation of a length-17 array of all-ones words. After the loop the values of the variables `a0`, ..., `a4` are

```
a4=11111111
a3=........
a2=........
a1=........
a0=11111111
```

The columns, read as binary numbers, tell 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 [FXT: bits/bitcount-v.cc]:

```
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;
}
```

Each of the macros `VV` gives three machine instructions, one AND, XOR, and MOVE. The routine for the user is

```
ulong
bit_count_v(const ulong *x, ulong n)
// Return sum(j=0, n-1, bit_count(x[j]) )
{
    ulong b = 0;
    const ulong *xe = x + n + 1;
    while ( x+15 < xe )  // process blocks of 15 elements
    {
        b += bit_count_v15(x);
        x += 15;
    }

    // process remaining elements:
    const ulong r = (ulong)(xe-x-1);
    for (ulong k=0; k<r; ++k)  b+=bit_count(x[k]);

    return  b;
}
```

Compared to the obvious method of bit-counting

```
ulong bit_count_v2(const ulong *x, ulong n)
{
    ulong b = 0;
    for (ulong k=0; k<n; ++k)  b += bit_count(x[k]);
    return  b;
}
```

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.

1.9 Words as bitsets

1.9.1 Testing whether subset of given bitset

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

```
static inline bool is_subset(ulong u, ulong e)
// Return whether the set bits of u are a subset of the set bits of e.
// That is, as bitsets, test whether u is a subset of e.
{
    return  ( (u & e)==u );
//    return  ( (u & ~e)==0 );
//    return  ( (~u | e)!=0 );
}
```

If u contains any bits not set in e, then these bits are cleared in the AND-operation and the test for equality will fail. The second version tests whether no element of u lies outside of e, the third is obtained by complementing the equality. A proper subset of e is a subset $\neq e$:

```
static inline bool is_proper_subset(ulong u, ulong e)
// Return whether u (as bitset) is a proper subset of e.
{
    return  ( (u<e) && ((u & e)==u) );
}
```

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    andl    %edx, %eax      # tmp66, tmp63
108    movzbl  %al, %eax       # tmp63, tmp61
```

1.9.2 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 $\{0, 1, 2, \ldots, \texttt{BITS\_PER\_LONG}-1\}$). For example, to determine whether `x` is a prime less than 32, use the function

```
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]:

```
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]:

```
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: .1..111.)
                  0x82UL,  // x = 7:  1 7      ( bits: 1.....1.)
                 0x116UL,  // x = 8:  1 2 4 8
                 0x20aUL,  // x = 9:  1 3 9
[--snip--]
            0x20000002UL,  // 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 3 11 33
[--snip--]
    0x2000000000000002UL,  // x = 61:  1 61
    0x4000000080000006UL,  // x = 62:  1 2 31 62
    0x800000000020028aUL   // x = 63:  1 3 7 9 21 63
#endif // ( BITS_PER_LONG > 32 )
};
```

Bit-arrays of arbitrary size are discussed in section 4.6 on page 164.

1.10 Index of the i-th set bit

To determine the index of the i-th set bit, we use a technique similar to the method for counting the bits of a word. Only the 64-bit version is shown [FXT: bits/ith-one-idx.h]:

```
static inline ulong ith_one_idx(ulong x, ulong i)
// Return index of the i-th set bit of x where 0 <= i < bit_count(x).
{
    ulong x2 = x - ((x>>1) & 0x5555555555555555UL);    // 0-2 in 2 bits
    ulong x4 = ((x2>>2) & 0x3333333333333333UL) +
                (x2 & 0x3333333333333333UL);        // 0-4 in 4 bits
    ulong x8 = ((x4>>4) + x4) & 0x0f0f0f0f0f0f0f0fUL;  // 0-8 in 8 bits
    ulong ct = (x8 * 0x0101010101010101UL) >> 56;      // bit count

    ++i;
    if ( ct < i )  return ~0UL;  // less than i bits set

    ulong x16 = (0x00ff00ff00ff00ffUL & x8) + (0x00ff00ff00ff00ffUL & (x8>>8));     // 0-16
    ulong x32 = (0x0000ffff0000ffffUL & x16) + (0x0000ffff0000ffffUL & (x16>>16));  // 0-32

    ulong w, s = 0;

    w = x32 & 0xffffffffUL;
    if ( w < i )  { s += 32;  i -= w; }

    x16 >>= s;
    w = x16 & 0xffff;
    if ( w < i )  { s += 16;  i -= w; }

    x8 >>= s;
    w = x8 & 0xff;
    if ( w < i )  { s += 8;  i -= w; }

    x4 >>= s;
    w = x4 & 0xf;
    if ( w < i )  { s += 4;  i -= w; }

    x2 >>= s;
    w = x2 & 3;
    if ( w < i )  { s += 2;  i -= w; }

    x >>= s;
    s += ( (x&1) != i );

    return s;
}
```

1.11 Avoiding branches

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

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

where `x` might be a signed integer, by

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

The obvious code to test whether a point (x, y) lies outside a square box of size m is

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

If m is a power of 2, it is better to use

```
if ( ( (ulong)x | (ulong)y ) > (unsigned)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:

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

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 clears all bits. Note this function will only work if the compiler emits an arithmetic right shift, see section 1.1.3 on page 3. The following routine computes $\min(0, x)$:

```
static inline long min0(long x)
// Return min(0, x), i.e. return zero for positive input
{
    return  x & (x >> (BITS_PER_LONG-1));
}
```

The following `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 $|a - b|$:

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

The following routine sorts two values:

```
static inline void upos_sort2(ulong &a, ulong &b)
// Set {a, b} := {min(a, b), max(a,b)}
// Both a and b must not have the most significant bit set
{
    long d = b - a;
    d &= (d>>(BITS_PER_LONG-1));
    a += d;
    b -= d;
}
```

Johan Rönnblom gives [priv. comm.] the following versions for signed integer minimum, maximum, and absolute value, that can be advantageous for CPUs where immediates are expensive:

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

Your compiler may be smarter than you thought

The machine code generated for

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

is

```
35:   48 99                   cqto
37:   48 83 c4 08             add    $0x8,%rsp  // stack adjustment
3b:   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 [13]:

> 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

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

is compiled to:

```
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, the 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]:

```
static inline long clip_range(long x, long mi, long ma)
// Code equivalent to (for mi<=ma):
//   if ( x<mi )  x = mi;
//   else if ( x>ma )  x = ma;
{
    x -= mi;
    x = clip_range0(x, ma-mi);
    x += mi;
    return  x;
}
```

The auxiliary function used involves one branch:

```
static inline long clip_range0(long x, long m)
// Code equivalent (for m>0) to:
//   if ( x<0 )  x = 0;
//   else if ( x>m )  x = m;
//   return  x;
{
    if ( (ulong)x > (ulong)m )  x = m & ~(x >> (BITS_PER_LONG-1));
    return  x;
}
```

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
  10:   48 89 d1                mov    %rdx,%rcx
  13:   48 0f 4e c8             cmovle %rax,%rcx
  17:   48 8d 04 0e             lea    (%rsi,%rcx,1),%rax
```

Now we replace the code by

```
static inline long clip_range(long x, long mi, long ma)
{
    x -= mi;
    if ( x<0 )  x = 0;
//    else  // commented out to make (compiled) function really branchless
    {
        ma -= mi;
        if ( x>ma )  x = ma;
    }
    x += mi;
}
```

Then the compiler generates branchless code:

```
   0:   48 89 f8                mov    %rdi,%rax
   3:   b9 00 00 00 00          mov    $0x0,%ecx
   8:   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]:

```
static inline ulong bit_rotate_left(ulong x, ulong r)
// Return word rotated r bits to the left
// (i.e. toward the most significant bit)
```

```
{
    return  (x<<r) | (x>>(BITS_PER_LONG-r));
}
```

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

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

Here we use an assembler instruction when available [FXT: bits/bitasm-amd64.h]:

```
static inline ulong asm_ror(ulong x, ulong r)
{
    asm ("rorq   %%cl, %0" : "=r" (x) : "0" (x), "c" (r));
    return x;
}
```

Rotation using only a part of the word length can be implemented as

```
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 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:
//   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:

```
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]:

```
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 18 on page 370. For example, with 6-bit words we find:

```
  word   period           word   period
......   1               ..11.1  6
.....1   6               ..1111  6
....11   6               .1.1.1  2
...1.1   6               .1.111  6
...111   6               .11.11  3
..1..1   3               .11111  6
..1.11   6               111111  1
```

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

```
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 computed as `x^(x>>d)` are zero if and only if the word has period d. So we can use the following function body:

```
    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:

```
    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 1.9.2 on page 24.

The version for `ldn==BITS_PER_LONG` can be optimized similarly:

```
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 18.1.1 on page 371. Here we specialize the method for binary words. The words generated are the cyclic maxima [FXT: `class bit_necklace` in bits/bit-necklace.h]:

```
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;

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

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

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

The method for computing the successor is

```
    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]:

```
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:

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

    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:

```
static 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

```
static 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]

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

and

```
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 41):

```
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, 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 45):

```
static inline ulong bit_cyclic_inv_lxor(ulong x)
// Return v so that bit_cyclic_lxor(v) == x.
{
    return  inverse_rev_gray_code(x);
}
```

We do not need to mask out the lowest bit because for valid arguments (that have an even number of bits) the high bits of the result are zero. This function can be used to solve the quadratic equation $v^2 + v = x$ in the finite field $\mathrm{GF}(2^n)$ when normal bases are used, see section 42.6.2 on page 903.

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]:

```
[ 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 ]  <--= pairs swapped
[ 3 2 1 0 7 6 5 4 b a 9 8 f e d c ]  <--= groups of 2 swapped
[ 7 6 5 4 3 2 1 0 f e d c b a 9 8 ]  <--= groups of 4 swapped
[ f e d c b a 9 8 7 6 5 4 3 2 1 0 ]  <--= groups of 8 swapped
```

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

```
static inline ulong bit_swap_1(ulong x)
// Return x with neighbor bits swapped.
{
#if  BITS_PER_LONG == 32
    ulong m = 0x55555555UL;
#else
#if  BITS_PER_LONG == 64
    ulong m = 0x5555555555555555UL;
#endif
#endif
    return  ((x & m) << 1) | ((x & (~m)) >> 1);
}
```

The 64-bit branch is omitted in the following examples. Adjacent groups of 2 bits are swapped by

```
static inline ulong bit_swap_2(ulong x)
// Return x with groups of 2 bits swapped.
{
    ulong m = 0x33333333UL;
    return  ((x & m) << 2) | ((x & (~m)) >> 2);
}
```

Equivalently,

```
static inline ulong bit_swap_4(ulong x)
// Return x with groups of 4 bits swapped.
{
    ulong m = 0x0f0f0f0fUL;
    return  ((x & m) << 4) | ((x & (~m)) >> 4);
}
```

and

```
static inline ulong bit_swap_8(ulong x)
// Return x with groups of 8 bits swapped.
{
    ulong m = 0x00ff00ffUL;
    return  ((x & m) << 8) | ((x & (~m)) >> 8);
}
```

When swapping half-words (here for 32-bit architectures)

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

we could also use the bit-rotate function from section 1.12 on page 27, 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]:

```
static inline ulong revbin(ulong x)
// Return x with reversed bit order.
{
    x = bit_swap_1(x);
    x = bit_swap_2(x);
    x = bit_swap_4(x);
    x = bit_swap_8(x);
    x = bit_swap_16(x);
#if  BITS_PER_LONG >= 64
    x = bit_swap_32(x);
#endif
    return x;
}
```

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

```
static inline ulong asm_bswap(ulong x)
{
    asm ("bswap %0" : "=r" (x) : "0" (x));
    return x;
}
```

We use it for byte reversal if available:

```
static inline ulong bswap(ulong x)
// Return word with reversed byte order.
{
#ifdef BITS_USE_ASM
    x = asm_bswap(x);
#else
    x = bit_swap_8(x);
    x = bit_swap_16(x);
#if  BITS_PER_LONG >= 64
    x = bit_swap_32(x);
#endif
#endif // def BITS_USE_ASM
    return x;
}
```

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

```
static inline ulong revbin(ulong x)
{
    x = bit_swap_1(x);
    x = bit_swap_2(x);
    x = bit_swap_4(x);
    x = bswap(x);
    return x;
}
```

The masks can be generated in the process:

```
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;
}
```

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

```
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

```
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 if `ldn` is not too small, else be replaced by the trivial algorithm.

We can use table lookups so that, for example, eight bits are reversed at a time using a 256-byte table. The routine for full words is

```
unsigned char revbin_tab[256]; // reversed 8-bit words
ulong revbin_t(ulong x)
{
    ulong r = 0;
    for (ulong k=0; k<BYTES_PER_LONG; ++k)
    {
        r <<= 8;
        r |= revbin_tab[ x & 255 ];
        x >>= 8;
    }
    return r;
}
```

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

```
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;
    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 `bit_swap_NN()` calls, see [FXT: bits/revbin-tab-demo.cc].

1.14.3 Generating the bit-reversed words in order

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

```
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 $N = 2^n - 1$ words less 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, two bits change for a quarter, three bits change for one eighth of all updates, and so on. So the loop executes less than $2\,N$ times:

$$N\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\dots+\frac{\log_2(N)}{N}\right) \quad = \quad N\sum_{j=1}^{\log_2(N)}\frac{j}{2^j} \;<\; 2\,N \qquad (1.14\text{-}1)$$

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:

```
static 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

```
pattern  reversed pattern
  ....1     1....
  ...11     11...
  ....1     1....
  ..111     111..
  ....1     1....
  ...11     11...
  ....1     1....
  .1111     1111.
  ....1     1....
  ...11     11...
```

The pattern with x set bits is used for the update of k to $k+1$ when the lowest zero of k is at position $x - 1$:

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

The update routine can now be implemented as

```
static 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 2.6.

	30 bits	16 bits	8 bits	
Update, bit-wise	1.00	1.00	1.00	`revbin_upd()`
Update, table	0.99	1.08	1.15	`revbin_tupd()`
Full, masks	0.74	0.81	0.86	`revbin()`
Full, 8-bit table	1.77	1.94	2.06	`revbin_t()`
Full32, 8-bit table	0.83	0.90	0.96	`revbin_t_le32()`
Full16, 8-bit table	—	0.54	0.58	`revbin_t_le16()`
Full, generated masks	2.97	3.25	3.45	[page 35]
Full, bit-wise	8.76	5.77	2.50	[page 35]

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 1. 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, mainly because the machine used (see appendix B on page 922) 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:

```
    ulong n = 32;  // a power of 2
    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

```
    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]:

```
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);
}
```

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 41.4 on page 873.

1.15 Bit-wise zip

The bit-wise zip (bit-zip) operation moves the bits in the lower half to even indices and the bits in the upper half to odd indices. For example, with 8-bit words the permutation of bits is

```
[ a b c d A B C D ]  |-->  [ a A b B c C d D ]
```

A straightforward implementation is

```
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 upper half-word:

```
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;
        m <<= 1;
    }
}
```

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

```
static inline ulong butterfly_4(ulong x)
// Swap in each block of 16 bits the two central blocks of 4 bits.
{
    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:

```
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]:

```
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);
    return  x;
}
```

and

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

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

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

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

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

Its inverse is

```
static inline ulong bit_unzip0(ulong x)
// Bits at odd positions must be zero.
{
    x = (x | (x>>1))  & 0x3333333333333333UL;
    x = (x | (x>>2))  & 0x0f0f0f0f0f0f0f0fUL;
    x = (x | (x>>4))  & 0x00ff00ff00ff00ffUL;
    x = (x | (x>>8))  & 0x0000ffff0000ffffUL;
    x = (x | (x>>16)) & 0x00000000ffffffffUL;
    return  x;
}
```

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

```
static inline ulong bit_zip(ulong x)
{
    ulong y =  (x >> 32);
    x &= 0xffffffffUL;
    x = (x | (x<<16)) & 0x0000ffff0000ffffUL;
    y = (y | (y<<16)) & 0x0000ffff0000ffffUL;
    x = (x | (x<<8))  & 0x00ff00ff00ff00ffUL;
    y = (y | (y<<8))  & 0x00ff00ff00ff00ffUL;
    x = (x | (x<<4))  & 0x0f0f0f0f0f0f0f0fUL;
    y = (y | (y<<4))  & 0x0f0f0f0f0f0f0f0fUL;
    x = (x | (x<<2))  & 0x3333333333333333UL;
    y = (y | (y<<2))  & 0x3333333333333333UL;
    x = (x | (x<<1))  & 0x5555555555555555UL;
    y = (y | (y<<1))  & 0x5555555555555555UL;
    x |= (y<<1);
    return  x;
}
```

```
static inline ulong bit_unzip(ulong x)
{
    ulong y = (x >> 1) & 0x5555555555555555UL;
    x &= 0x5555555555555555UL;
    x = (x | (x>>1))  & 0x3333333333333333UL;
    y = (y | (y>>1))  & 0x3333333333333333UL;
    x = (x | (x>>2))  & 0x0f0f0f0f0f0f0f0fUL;
    y = (y | (y>>2))  & 0x0f0f0f0f0f0f0f0fUL;
    x = (x | (x>>4))  & 0x00ff00ff00ff00ffUL;
    y = (y | (y>>4))  & 0x00ff00ff00ff00ffUL;
    x = (x | (x>>8))  & 0x0000ffff0000ffffUL;
    y = (y | (y>>8))  & 0x0000ffff0000ffffUL;
    x = (x | (x>>16)) & 0x00000000ffffffffUL;
    y = (y | (y>>16)) & 0x00000000ffffffffUL;
    x |= (y<<32);
    return  x;
}
```

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 those 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 upper 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)
// 2-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)
// 2-word version:
// only the lower half of x and y are filled
{
    t = bit_unzip(t);
    y = t >> BPLH;
    x = t & 0x00000000ffffffffUL;
}
```

1.16 Gray code and parity

```
  k:    bin(k)        g(k)      g^-1(k)       g(2*k)     g(2*k+1)
  0:    .......     .......     .......     .......     ......1
  1:    ......1     ......1     ......1     .....11     .....1.
  2:    .....1.     .....11     .....11     ....11.     ....111
  3:    .....11     .....1.     .....1.     ....1.1     ....1..
  4:    ....1..     ....11.     ....111     ...11..     ...11.1
  5:    ....1.1     ....111     ....11.     ...1111     ...111.
  6:    ....11.     ....1.1     ....1..     ...1.1.     ...1.11
  7:    ....111     ....1..     ....1.1     ...1..1     ...1...
  8:    ...1...     ...11..     ...1111     ..11...     ..11..1
  9:    ...1..1     ...11.1     ...111.     ..11.11     ..11.1.
 10:    ...1.1.     ...1111     ...11..     ..1111.     ..11111
 11:    ...1.11     ...111.     ...11.1     ..111.1     ..111..
 12:    ...11..     ...1.1.     ...1...     ..1.1..     ..1.1.1
 13:    ...11.1     ...1.11     ...1..1     ..1.111     ..1.11.
 14:    ...111.     ...1..1     ...1.11     ..1..1.     ..1..11
 15:    ...1111     ...1...     ...1.1.     ..1...1     ..1....
 16:    ..1....     ..11...     ..11111     .11....     .11...1
 17:    ..1...1     ..11..1     ..1111.     .11..11     .11..1.
 18:    ..1..1.     ..11.11     ..111..     .11.11.     .11.111
 19:    ..1..11     ..11.1.     ..111.1     .11.1.1     .11.1..
 20:    ..1.1..     ..1111.     ..11...     .1111..     .1111.1
 21:    ..1.1.1     ..11111     ..11..1     .111111     .11111.
 22:    ..1.11.     ..111.1     ..11.11     .111.1.     .111.11
 23:    ..1.111     ..111..     ..11.1.     .111..1     .111...
 24:    ..11...     ..1.1..     ..1....     .1.1...     .1.1..1
 25:    ..11..1     ..1.1.1     ..1...1     .1.1.11     .1.1.1.
 26:    ..11.1.     ..1.111     ..1..11     .1.111.     .1.1111
 27:    ..11.11     ..1.11.     ..1..1.     .1.11.1     .1.11..
 28:    ..111..     ..1..1.     ..1.111     .1..1..     .1..1.1
 29:    ..111.1     ..1..11     ..1.11.     .1..111     .1..11.
 30:    ..1111.     ..1...1     ..1.1..     .1...1.     .1...11
 31:    ..11111     ..1....     ..1.1.1     .1....1     .1.....
```

Figure 1.16-A: Binary words, their Gray code, inverse Gray code, and Gray codes of even and odd values (from left to right).

The *Gray code* of a binary word can easily be computed by [FXT: bits/graycode.h]

```
static inline ulong gray_code(ulong x)  { return  x ^ (x>>1); }
```

Gray codes of consecutive values differ in one bit. Gray codes of values that differ by a power of 2 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/gray-demo.cc], whose output is given in figure 1.16-A.

To produce a random value with an even/odd number of bits set, set the lowest bit of a random number to 0/1, respectively, and return its Gray code.

Computing the inverse Gray code is slightly more expensive. As the Gray code is the bit-wise difference modulo 2, we can compute 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;
}
```

For n-bit words, n-fold application of the Gray code gives back the original word. Using the symbol G for the Gray code (operator), we have $G^n = \mathrm{id}$, so $G^{n-1} \circ G = \mathrm{id} = G^{-1} \circ G$. That is, applying the Gray code computation $n-1$ times gives the inverse Gray code. Thus we can simplify to

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

Applying the Gray code twice is identical to `x^=x>>2;`, applying it four times is `x^=x>>4;`, and the idea holds for all power of 2. This leads to the most efficient way to compute the inverse Gray code:

```
    // VERSION 3 (use: gray ** BITSPERLONG == id):
    x ^= x>>1;  // gray ** 1
    x ^= x>>2;  // gray ** 2
    x ^= x>>4;  // gray ** 4
    x ^= x>>8;  // gray ** 8
    x ^= x>>16; // gray ** 16
    // here: x = gray**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x>>32;  // for 64bit words
#endif
    return  x;
```

1.16.1 The parity of a binary word

The *parity* of a word is its bit-count modulo 2. The lowest bit of the inverse Gray code of a word contains the parity of the word. So we can compute the parity as [FXT: bits/parity.h]:

```
static inline ulong parity(ulong x)
// Return 0 if the number of set bits is even, else 1
{
    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 into account the lowest byte. The following routine computes the parity of a full word [FXT: bits/bitasm-i386.h]:

```
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>16);
    x ^= (x>>8);
    asm ("addl  $0, %0  \n"
         "setnp %%al    \n"
         "movzx %%al, %0"
         : "=r" (x) : "0" (x) : "eax");
    return x;
}
```

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

```
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>32);
    x ^= (x>>16);
    x ^= (x>>8);
    asm ("addq  $0, %0  \n"
         "setnp %%al    \n"
         "movzx %%al, %0"
         : "=r" (x) : "0" (x) : "rax");
    return x;
}
```

1.16.2 Byte-wise Gray code and parity

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

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

Its inverse is

```
static inline ulong byte_inverse_gray_code(ulong x)
// Return the inverse Gray code of bytes in parallel
{
    x ^= ((x & 0xfefefefeUL)>>1);
    x ^= ((x & 0xfcfcfcfcUL)>>2);
    x ^= ((x & 0xf0f0f0f0UL)>>4);
    return  x;
}
```

And the parities of all bytes can be computed as

```
static inline ulong byte_parity(ulong x)
// Return the parities of bytes in parallel
{
    return  byte_inverse_gray_code(x) & 0x01010101UL;
}
```

1.16.3 Incrementing (counting) in Gray code

```
 k:     g(k)      g(2*k)       g(k) p     diff p     set
 0:    .......    .......     ...... .    ...... .    {}
 1:    ......1    .....11     .....1 1    .....+ 1    {0}
 2:    .....11    ....11.     ....11 .    ....+1 .    {0, 1}
 3:    .....1.    ....1.1     ....1. 1    ....1- 1    {1}
 4:    ....11.    ...11..     ...11. .    ...+1. .    {1, 2}
 5:    ....111    ...1111     ...111 1    ...11+ 1    {0, 1, 2}
 6:    ....1.1    ...1.1.     ...1.1 .    ...1-1 .    {0, 2}
 7:    ....1..    ...1..1     ...1.. 1    ...1.- 1    {2}
 8:    ...11..    ..11...     ..11.. .    ..+1.. .    {2, 3}
 9:    ...11.1    ..11.11     ..11.1 1    ..11.+ 1    {0, 2, 3}
10:    ...1111    ..1111.     ..1111 .    ..11+1 .    {0, 1, 2, 3}
11:    ...111.    ..111.1     ..111. 1    ..111- 1    {1, 2, 3}
12:    ...1.1.    ..1.1..     ..1.1. .    ..1-1. .    {1, 3}
13:    ...1.11    ..1.111     ..1.11 1    ..1.1+ 1    {0, 1, 3}
14:    ...1..1    ..1..1.     ..1..1 .    ..1.-1 .    {0, 3}
15:    ...1...    ..1...1     ..1... 1    ..1..- 1    {3}
16:    ..11...    .11....     .11... .    .+1... .    {3, 4}
17:    ..11..1    .11..11     .11..1 1    .11..+ 1    {0, 3, 4}
```

Figure 1.16-B: The Gray code equals the Gray code of doubled value shifted to the right once. Equivalently, 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 $g(k+1)$. We can implement a fast Gray counter if we use a spare bit to keep track of the parity of the Gray code word, see figure 1.16-B The following routine does this [FXT: bits/nextgray.h]:

```
static 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);
    }

```

This is shown in [FXT: bits/bit-nextgray-demo.cc]. To start at an arbitrary (Gray code) value `g`, compute

```
x = (g<<1) ^ parity(g)
```

Then use the statement `x=next_gray2(x)` for later increments.

If working with a set whose elements are the set bits in the Gray code, the parity is the set size k modulo 2. Compute the increment as follows:

1. If k is even, then goto step 2, else goto step 3.
2. If the first element is zero, then remove it, else prepend the element zero.
3. 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.

A method to decrement is obtained by simply swapping the actions for even and odd parity.

When working with an array that contains the elements of the set, it is more convenient to do the described operations at the end of the array. This leads to the (loopless) algorithm for subsets in minimal-change order given in section 8.2.2 on page 206. Properties of the Gray code are discussed in [127].

1.16.4 The Thue-Morse sequence

The sequence of parities of the binary words

```
0110100110010110100101100110100110010110011010011...
```

is called the *Thue-Morse sequence* (entry A010060 in [312]). It appears in various seemingly unrelated contexts, see [8] and section 38.1 on page 726. The sequence can be generated with [FXT: `class thue_morse` in bits/thue-morse.h]:

```
class thue_morse
// Thue-Morse sequence
{
public:
    ulong k_;
    ulong tm_;

public:
    thue_morse(ulong k=0)  { init(k); }
    ~thue_morse()  { ; }

    ulong init(ulong k=0)
    {
        k_ = k;
        tm_ = parity(k_);
        return tm_;
    }

    ulong data()  { return tm_; }

    ulong next()
    {
        ulong x = k_ ^ (k_ + 1);
        ++k_;
        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 366 M/s (6 cycles per update) [FXT: bits/thue-morse-demo.cc].

1.16.5 The Golay-Rudin-Shapiro sequence ‡

The function [FXT: bits/grsnegative.h]

```
static inline ulong grs_negative_q(ulong x)  { return  parity( x & (x>>1) ); }
```

returns +1 for indices where the *Golay-Rudin-Shapiro sequence* (or *GRS sequence*, entry A020985 in [312]) has the value −1. The algorithm is to count the bit-pairs modulo 2. The pairs may overlap: the

```
++
+++-
+++- ++-+
+++- ++-+  +++- --+-
+++- ++-+  +++- --+-  +++- ++-+  ---+ ++-+
+++- ++-+  +++- --+-  +++- ++-+  ---+ ++-+  +++- ++-+  +++- --+- ...
   ^   ^      ^ ^^ ^     ^   ^
                                ...
   3,  6,  11,12,13,15, 19, 22, ...
```

Figure 1.16-C: A construction for the Golay-Rudin-Shapiro (GRS) sequence.

sequence `[1111]` contains the three bit-pairs `[11..]`, `[.11.]`, and `[..11]`. The function returns +1 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, ...
```

This is entry A022155 in [312], see also section 38.3 on page 731. The sequence can be computed by starting with two ones, and appending the left half and the negated right half of the values so far in each step, see figure 1.16-C. To compute the successor in the GRS sequence, use

```
static inline ulong grs_next(ulong k, ulong g)
// With g == grs_negative_q(k), compute grs_negative_q(k+1).
{
    const ulong cm = 0x5555555555555554UL;  // 64-bit version
    ulong h = ~k;  h &= -h;  // == lowest_zero(k);
    g ^= ( ((h&cm) ^ ((k>>1)&h)) !=0 );
    return  g;
}
```

With incrementing k, the lowest run of ones of k is replaced by a one at the lowest zero of k. If the length of the lowest run is odd and ≥ 2 then a change of parity happens. This is the case if the lowest zero of k is at one of the positions

```
bin  0101 0101 0101 0100  ==  hex 5 5 5 4  ==  cm
```

If the position of the lowest zero is adjacent to the next block of ones, another change of parity will occur. The element of the GRS sequence changes if exactly one of the parity changes takes place.

The update function can be used as shown in [FXT: bits/grs-next-demo.cc]:

```
    ulong n = 65;  // Generate this many values of the sequence.
    ulong k0 = 0;  // Start point of the sequence.
    ulong g = grs_negative_q(k0);
    for (ulong k=k0;  k<k0+n;  ++k)
    {
        // Do something with g here.
        g = grs_next(k, g);
    }
```

The rate of generation is about 347 M/s, direct computation gives a rate of 313 M/s.

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,

```
rev_gray_code(x) := revbin( gray_code( 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 '<<'). So computing the reversed Gray code is as easy as [FXT: bits/revgraycode.h]:

```
static inline ulong rev_gray_code(ulong x)  { return  x ^ (x<<1); }
```

Its inverse is

```
static inline ulong inverse_rev_gray_code(ulong x)
{
    // use: rev_gray ** BITSPERLONG == id:
    x ^= x<<1;  // rev_gray ** 1
    x ^= x<<2;  // rev_gray ** 2
    x ^= x<<4;  // rev_gray ** 4
```

```
----------------------------------------------------------
111.1111....1111................  = 0xef0f0000  == word
1..11...1...1...1............... = gray_code
..11...1...1...1................ = rev_gray_code
1.11.1.11111.1.11111111111111111 = inverse_gray_code
1.1..1.1....1.1................. = inverse_rev_gray_code
----------------------------------------------------------
...1....1111....1111111111111111  = 0x10f0ffff  == word
...11...1...1...1............... = gray_code
..11...1...1...1...............1 = rev_gray_code
...11111.1.11111.1.1.1.1.1.1.1.1 = inverse_gray_code
1111.....1.1.....1.1.1.1.1.1.1.1 = inverse_rev_gray_code
----------------------------------------------------------
......1.........................  = 0x2000000  == word
......11........................ = gray_code
.....11......................... = rev_gray_code
......11111111111111111111111111 = inverse_gray_code
1111111......................... = inverse_rev_gray_code
----------------------------------------------------------
111111.1111111111111111111111111  = 0xfdffffff  == word
1.....11........................ = gray_code
.....11........................1 = rev_gray_code
1.1.1..1.1.1.1.1.1.1.1.1.1.1.1.1 = inverse_gray_code
1.1.1.11.1.1.1.1.1.1.1.1.1.1.1.1 = inverse_rev_gray_code
----------------------------------------------------------
```

Figure 1.16-D: Examples of the Gray code, reversed Gray code, and their inverses with 32-bit words.

```
    x ^= x<<8;  // rev_gray ** 8
    x ^= x<<16; // rev_gray ** 16
    // here: x = rev_gray**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x<<32;  // for 64bit words
#endif
    return  x;
}
```

Some examples with 32-bit words are shown in figure 1.16-D.

Let G and E denote be the Gray code and reversed Gray code of a word X, respectively. Write G^{-1} and E^{-1} for their inverses. Then E preserves the lowest bit of X, while E preserves the highest. Also E preserves the lowest *set* bit of X, while E preserves the highest. Further, E^{-1} contains at each bit the parity of all bits of X right from it, including the bit itself. Especially, the word parity can be found in the highest bit of E^{-1}.

Let $\overline{X}$ denote the complement of X, p its parity, and let S the right shift by one of G^{-1}. Then we have

$$G^{-1} \text{ XOR } E^{-1} = \begin{cases} X & \text{if } p = 0 \\ \overline{X} & \text{otherwise} \end{cases} \tag{1.16-1a}$$

$$S \text{ XOR } E^{-1} = \begin{cases} 0 & \text{if } p = 0 \\ \overline{0} & \text{otherwise} \end{cases} \tag{1.16-1b}$$

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 40.1.6 on page 826. The inverse reversed Gray code can be used to solve the reduced quadratic equation for binary normal bases, see section 42.6.2 on page 903.

1.17 Bit sequency ‡

The *sequency* of a binary word is the number of zero-one transitions in the word. A function to determine the sequency is [FXT: bits/bitsequency.h]:

```
static inline ulong bit_sequency(ulong x)  { return bit_count( gray_code(x) ); }
```

```
    seq=    0         1         2         3         4         5         6
          ......    .....1    ....1.    ...1.1    ..1.1.    .1.1.1    1.1.1.
                    ....11    ...11.    ..11.1    .11.1.    11.1.1
                    ...111    ...1..    ..1..1    .1..1.    1..1.1
                    ..1111    ..111.    ..1.11    .1.11.    1.11.1
                    .11111    ..11..    .111.1    .1.1..    1.1..1
                    111111    ..1...    .11..1    111.1.    1.1.11
                              .1111.    .11.11    11..1.
                              .111..    .1...1    11.11.
                              .11...    .1..11    11.1..
                              .1....    .1.111    1...1.
                              11111.    1111.1    1..11.
                              1111..    111..1    1..1..
                              111...    111.11    1.111.
                              11....    11...1    1.11..
                              1.....    11..11    1.1...
                                        11.111
                                        1....1
                                        1...11
                                        1..111
                                        1.1111
```

Figure 1.17-A: 6-bit words of prescribed sequency as generated by `next_sequency()`.

The function assumes that all bits to the left of the word are zero and all bits to the right are equal to the lowest bit, see figure 1.17-A. For example, the sequency of the 8-bit word `[00011111]` 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:

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

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

```
    if ( k==0 )  return 0;
    const ulong m = 0xaaaaaaaaUL;
    return  m >> (BITS_PER_LONG-k);
```

The maximal binary word with given sequency can be computed via

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

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 1.24 on page 62).

For the generation of all words with a given sequency, starting with the smallest, we use a function that computes the next word with the same sequency:

```
static inline ulong next_sequency(ulong x)
{
    x = gray_code(x);
    x = next_colex_comb(x);
    x = inverse_gray_code(x);
    return x;
}
```

The inverse function, returning the previous word with the same sequency, is

```
static inline ulong prev_sequency(ulong x)
{
    x = gray_code(x);
    x = prev_colex_comb(x);
    x = inverse_gray_code(x);
    return x;
7 }
```

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].

The sequency of a word can be complemented as follows (32-bit version):

```
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 ‡

```
 1.......    11......    1.1.....    1111....    1...1...    11..11..    1.1.1.1.    11111111
 .1......    .11.....    .1.1....    .1111...    .1...1..    .11..11.    .1.1.1.1    .1111111
 ..1.....    ..11....    ..1.1...    ..1111..    ..1...1.    ..11..11    ..1.1.1.    ..111111
 ...1....    ...11...    ...1.1..    ...1111.    ...1...1    ...11..1    ...1.1.1    ...11111
 ....1...    ....11..    ....1.1.    ....1111    ....1...    ....11..    ....1.1.    ....1111
 .....1..    .....11.    .....1.1    .....111    .....1..    .....11.    .....1.1    .....111
 ......1.    ......11    ......1.    ......11    ......1.    ......11    ......1.    ......11
 .......1    .......1    .......1    .......1    .......1    .......1    .......1    .......1
  G^0=id      G^1=G       G^2         G^3         G^4         G^5         G^6       G^7=G^(-1)
---------------------------------------------------------------------------------------------
 1.......    1.......    1.......    1.......    1.......    1.......    1.......    1.......
 .1......    11......    .1......    11......    .1......    11......    .1......    11......
 ..1.....    .11.....    1.1.....    111.....    ..1.....    .11.....    1.1.....    111.....
 ...1....    ..11....    .1.1....    1111....    ...1....    ..11....    .1.1....    1111....
 ....1...    ...11...    ..1.1...    .1111...    1...1...    1..11...    1.1.1...    11111...
 .....1..    ....11..    ...1.1..    ..1111..    .1...1..    11..11..    .1.1.1..    111111..
 ......1.    .....11.    ....1.1.    ...1111.    ..1...1.    .11..11.    1.1.1.1.    1111111.
 .......1    ......11    .....1.1    ....1111    ...1...1    ..11..11    .1.1.1.1    11111111
  E^0=id      E^1=E       E^2         E^3         E^4         E^5         E^6       E^7=E^(-1)
```

Figure 1.18-A: Powers of the matrices for the Gray code (top) and the reversed Gray code (bottom).

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]:

```
static 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 $g = [g_0, g_1, \ldots, g_7]$ of a 8-bit binary word $x = [x_0, x_1, \ldots, x_7]$ can be expressed as a matrix multiplication over GF(2) (dots for zeros):

```
 g    =        G          x

[g0]      [ 11...... ]  [x0]
[g1]      [ .11..... ]  [x1]
[g2]      [ ..11.... ]  [x2]
[g3]  =   [ ...11... ]  [x3]
[g4]      [ ....11.. ]  [x4]
[g5]      [ .....11. ]  [x5]
[g6]      [ ......11 ]  [x6]
[g7]      [ .......1 ]  [x7]
```

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 2) can be computed by the relation $G^e\, G^{N-e} = G^N = \text{id}$.

```
static 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)
{
    return  gray_pow(x, -e);
}
```

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:

```
static inline ulong rev_gray_pow(ulong x, ulong e)
// Return (rev_gray_code**e)(x)
{
    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

```
static inline ulong inverse_rev_gray_pow(ulong x, ulong e)
// Return (inverse_rev_gray_code**(e))(x)
{
    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*)

```
static 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)

```
    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.
```

The same masks are used in the *yellow code*

```
static 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 need $O(\log_2 \texttt{BITS\_PER\_LONG})$ operations. The `blue_code` can be used as a fast implementation for the composition of a binary polynomial with $x + 1$, see section 40.7.2 on page 845. The yellow code can also be computed by the statement

```
revbin( blue_code( revbin(x) ) );
```

So we could have called it *reversed blue code*. Note the names 'blue code' etc. are ad hoc terminology and not standard. See section 23.11 on page 486 for the closely related Reed-Muller transform.

```
              blue                          yellow
     0:     ......  0*        ................................  0
     1:     .....1  1*        11111111111111111111111111111111 32
     2:     ....11  2         1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. 16
     3:     ....1.  1         .1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 16
     4:     ...1.1  2         11..11..11..11..11..11..11..11.. 16
     5:     ...1..  1         ..11..11..11..11..11..11..11..11 16
     6:     ...11.  2*        .11..11..11..11..11..11..11..11. 16
     7:     ...111  3*        1..11..11..11..11..11..11..11..1 16
     8:     ..1111  4         1...1...1...1...1...1...1...1...  8
     9:     ..111.  3         .111.111.111.111.111.111.111.111 24
    10:     ..11..  2         ..1...1...1...1...1...1...1...1.  8
    11:     ..11.1  3         11.111.111.111.111.111.111.111.1 24
    12:     ..1.1.  2         .1...1...1...1...1...1...1...1..  8
    13:     ..1.11  3         1.111.111.111.111.111.111.111.11 24
    14:     ..1..1  2         111.111.111.111.111.111.111.111. 24
    15:     ..1...  1         ...1...1...1...1...1...1...1...1  8
    16:     .1...1  2         1111....1111....1111....1111.... 16
    17:     .1....  1         ....1111....1111....1111....1111 16
    18:     .1..1.  2*        .1.11.1..1.11.1..1.11.1..1.11.1. 16
    19:     .1..11  3*        1.1..1.11.1..1.11.1..1.11.1..1.1 16
    20:     .1.1..  2*        ..1111....1111....1111....1111.. 16
    21:     .1.1.1  3*        11....1111....1111....1111....11 16
    22:     .1.111  4         1..1.11.1..1.11.1..1.11.1..1.11. 16
    23:     .1.11.  3         .11.1..1.11.1..1.11.1..1.11.1..1 16
    24:     .1111.  4         .1111....1111....1111....1111... 16
    25:     .11111  5         1....1111....1111....1111....111 16
    26:     .111.1  4         11.1..1.11.1..1.11.1..1.11.1..1. 16
    27:     .111..  3         ..1.11.1..1.11.1..1.11.1..1.11.1 16
    28:     .11.11  4         1.11.1..1.11.1..1.11.1..1.11.1.. 16
    29:     .11.1.  3         .1..1.11.1..1.11.1..1.11.1..1.11 16
    30:     .11...  2         ...1111....1111....1111....1111. 16
    31:     .11..1  3         111....1111....1111....1111....1 16
```

Figure 1.19-A: Blue and yellow transforms of the binary words 0, 1, ..., 31. 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 ± 1 between successive values of $B(a)$, the transition $B(47) \to B(48)$ changes the bit-count by 3. The sequence of the indices a where the bit-count changes by more than one is

```
47, 51, 59, 67, 75, 79, 175, 179, 187, 195, 203, 207, 291, 299, 339, 347, 419, 427, ...
```

The yellow code might be a good candidate for 'randomization' of binary words. The blue code maps any range $[0 \ldots 2^k - 1]$ onto itself. Both the blue code and the yellow code are involutions (self-inverse).

The transforms (*red code*)

```
static 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;
```

```
                 red                                                  green
  0:    ................................  0     ................................  0
  1:    1...............................  1     11111111111111111111111111111111 32
  2:    11..............................  2     .1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 16
  3:    .1..............................  1     1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. 16
  4:    1.1.............................  2     ..11..11..11..11..11..11..11..11 16
  5:    ..1.............................  1     11..11..11..11..11..11..11..11.. 16
  6:    .11.............................  2     .11..11..11..11..11..11..11..11. 16
  7:    111.............................  3     1..11..11..11..11..11..11..11..1 16
  8:    1111............................  4     ...1...1...1...1...1...1...1...1  8
  9:    .111............................  3     111.111.111.111.111.111.111.111. 24
 10:    ..11............................  2     .1...1...1...1...1...1...1...1..  8
 11:    1.11............................  3     1.111.111.111.111.111.111.111.11 24
 12:    .1.1............................  2     ..1...1...1...1...1...1...1...1.  8
 13:    11.1............................  3     11.111.111.111.111.111.111.111.1 24
 14:    1..1............................  2     .111.111.111.111.111.111.111.111 24
 15:    ...1............................  1     1...1...1...1...1...1...1...1...  8
 16:    1...1...........................  2     ....1111....1111....1111....1111 16
 17:    ....1...........................  1     1111....1111....1111....1111.... 16
 18:    .1..1...........................  2     .1.11.1..1.11.1..1.11.1..1.11.1. 16
 19:    11..1...........................  3     1.1..1.11.1..1.11.1..1.11.1..1.1 16
 20:    ..1.1...........................  2     ..1111....1111....1111....1111.. 16
 21:    1.1.1...........................  3     11....1111....1111....1111....11 16
 22:    111.1...........................  4     .11.1..1.11.1..1.11.1..1.11.1..1 16
 23:    .11.1...........................  3     1..1.11.1..1.11.1..1.11.1..1.11. 16
 24:    .1111...........................  4     ...1111....1111....1111....1111. 16
 25:    11111...........................  5     111....1111....1111....1111....1 16
 26:    1.111...........................  4     .1..1.11.1..1.11.1..1.11.1..1.11 16
 27:    ..111...........................  3     1.11.1..1.11.1..1.11.1..1.11.1.. 16
 28:    11.11...........................  4     ..1.11.1..1.11.1..1.11.1..1.11.1 16
 29:    .1.11...........................  3     11.1..1.11.1..1.11.1..1.11.1..1. 16
 30:    ...11...........................  2     .1111....1111....1111....1111... 16
 31:    1..11...........................  3     1....1111....1111....1111....111 16
```

Figure 1.19-B: Red and green transforms of the binary words 0, 1, . . . , 31.

```
            m ^= (m<<s);
        }
        while ( s );
        return  a;
    }
```

and (*green code*)

```
    static inline ulong green_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;
    }
```

use the masks

```
    ................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
```

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]. The red code can also be computed by the statement

```
  revbin( blue_code( x ) );
```

and the green code by

```
  blue_code( revbin( x ) );
```

	i	r	B	Y	R	E
i	i	r	B	Y	R	E
r	r	i	R*	E*	B*	Y*
B	B	E*	i	R*	Y*	r*
Y	Y	R*	E*	i	r*	B*
R	R	Y*	r*	B*	E	i
E	E	B*	Y*	r*	i	R

Figure 1.19-C: Multiplication table for the transforms.

1.19.1 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:

$$B = Y\,r\,Y = r\,Y\,r \tag{1.19-1a}$$
$$Y = B\,r\,B = r\,B\,r \tag{1.19-1b}$$
$$r = Y\,B\,Y = B\,Y\,B \tag{1.19-1c}$$

As said, B and Y are self-inverse:

$$B^{-1} = B, \qquad B\,B = \mathrm{id} \tag{1.19-2a}$$
$$Y^{-1} = Y, \qquad Y\,Y = \mathrm{id} \tag{1.19-2b}$$

We write R for the red code, and E for the green code. The red code and the green code are not involutions (square roots of identity) but third roots of identity:

$$R\,R\,R = \mathrm{id}, \qquad R^{-1} = R\,R = E \tag{1.19-3a}$$
$$E\,E\,E = \mathrm{id}, \qquad E^{-1} = E\,E = R \tag{1.19-3b}$$
$$R\,E = E\,R = \mathrm{id} \tag{1.19-3c}$$

Figure 1.19-C shows the multiplication table. The R in the third column of the second row says that $r\,B = R$. The letter i is used for identity (id). An asterisk says that $x\,y \neq y\,x$.

By construction we have

$$R = r\,B \tag{1.19-4a}$$
$$E = r\,Y \tag{1.19-4b}$$

Relations between R and E are:

$$R = E\,r\,E = r\,E\,r \tag{1.19-5a}$$
$$E = R\,r\,R = r\,R\,r \tag{1.19-5b}$$
$$R = R\,E\,R \tag{1.19-5c}$$
$$E = E\,R\,E \tag{1.19-5d}$$

For the bit-reversal we have

$$r = Y\,R = R\,B = B\,E = E\,Y \tag{1.19-6}$$

Some products for the transforms are

$$B = R\,Y = Y\,E = R\,B\,R = E\,B\,E \tag{1.19-7a}$$
$$Y = E\,B = B\,R = R\,Y\,R = E\,Y\,E \tag{1.19-7b}$$
$$R = B\,Y = B\,E\,B = Y\,E\,Y \tag{1.19-7c}$$
$$E = Y\,B = B\,R\,B = Y\,R\,Y \tag{1.19-7d}$$

Some triple products that give the identical transform are

$$\mathrm{id} = B\,Y\,E = R\,Y\,B \tag{1.19-8a}$$
$$\mathrm{id} = E\,B\,Y = B\,R\,Y \tag{1.19-8b}$$
$$\mathrm{id} = Y\,E\,B = Y\,B\,R \tag{1.19-8c}$$

1.19.2 Relations to Gray code and reversed Gray code

Write g for the Gray code, then:

$$g\,B\,g\,B = \mathrm{id} \tag{1.19-9a}$$
$$g\,B\,g = B \tag{1.19-9b}$$
$$g^{-1}\,B\,g^{-1} = B \tag{1.19-9c}$$
$$g\,B = B\,g^{-1} \tag{1.19-9d}$$

Let S_k be the operator that rotates a word by k bits (bit 0 is moved to position k), then

$$Y\,S_{+1}\,Y = g \tag{1.19-10a}$$
$$Y\,S_{-1}\,Y = g^{-1} \tag{1.19-10b}$$
$$Y\,S_k\,Y = g^k \tag{1.19-10c}$$

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 2.12.1 on page 128) gives a curious method to generate the binary necklaces whose length is a power of 2, described in section 18.1.6 on page 376. Let e be the operator for the reversed Gray code, then

$$B\,S_{+1}\,B = e^{-1} \tag{1.19-11a}$$
$$B\,S_{-1}\,B = e \tag{1.19-11b}$$
$$B\,S_k\,B = e^{-k} \tag{1.19-11c}$$

1.19.3 Fixed points of the blue code ‡

```
 0 = ...... :    .......... =    0        16 = .1.... :    .1...1.... =   272
 1 = .....1 :    .........1 =    1        17 = .1...1 :    .1.11.1... =   360
 2 = ....1. :    .......11. =    6        18 = .1..1. :    .1.....1.. =   260
 3 = ....11 :    .......111 =    7        19 = .1..11 :    .1.11111.. =   380
 4 = ...1.. :    .....1.1.. =   20        20 = .1.1.. :    .1...1.11. =   278
 5 = ...1.1 :    .....1..1. =   18        21 = .1.1.1 :    .1.11.111. =   366
 6 = ...11. :    .....1.1.1 =   21        22 = .1.11. :    .1......1. =   258
 7 = ...111 :    .....1..11 =   19        23 = .1.111 :    .1.1111.1. =   378
 8 = ..1... :    ...1111... =  120        24 = .11... :    .1...1...1 =   273
 9 = ..1..1 :    ...11.11.. =  108        25 = .11..1 :    .1.11.1..1 =   361
10 = ..1.1. :    ...111111. =  126        26 = .11.1. :    .1.....1.1 =   261
11 = ..1.11 :    ...11.1.1. =  106        27 = .11.11 :    .1.11111.1 =   381
12 = ..11.. :    ...1111..1 =  121        28 = .111.. :    .1...1.111 =   279
13 = ..11.1 :    ...11.11.1 =  109        29 = .111.1 :    .1.11.1111 =   367
14 = ..111. :    ...1111111 =  127        30 = .1111. :    .1......11 =   259
15 = ..1111 :    ...11.1.11 =  107        31 = .11111 :    .1.1111.11 =   379
```

Figure 1.19-D: 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.

The sequence of fixed points of the blue code is (entry A118666 in [312])

```
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, $2\,(f \text{ XOR } (2\,f))$ 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]:

```
static inline ulong blue_fixed_point(ulong s)
{
    if ( 0==s )  return 0;
    ulong f = 1;
    while ( s>1 )
    {
        f ^= (f<<1);
        f <<= 1;
        f |= (s&1);
        s >>= 1;
    }
    return f;
}
```

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

Now write $f(x)$ for the binary polynomial corresponding to f (see chapter 40 on page 822), if $f(x)$ is a fixed point (that is, $B\,f(x) = f(x+1) = f(x)$), 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.

For the inverse function, we exploit that polynomial division by $x+1$ can be done with the inverse reversed Gray code (see section 1.16.6 on page 45) if the polynomial is divisible by $x+1$:

```
static 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;
}
```

1.19.4 More transforms by symbolic powering

The idea of powering a transform (as with the Gray code, see section 1.18 on page 48) can be applied to the 'color'-transforms as exemplified for the blue code:

```
static 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 $B\,B = \mathrm{id}$. The transforms (and the equivalents for Y, R and E, see [FXT: bits/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 $B\,B = \mathrm{id}$, now $B^x\,B^x = \mathrm{id}$ is true for all x. Similarly, $E\,E = R$ now has to be $E^x\,E^x = R^x$. That is, we have `BITS_PER_LONG` different versions of our four transforms that share their properties with the 'simple' versions. Among them are `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: $E^x\,E^x\,E^x = R^x\,R^x\,R^x = \mathrm{id}$.

While not powers of the simple versions, we still have $B^0 = Y^0 = R^0 = E^0 = \mathrm{id}$. 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, we have $Z^x Z^y = Z^{x+y}$ and so $Z^x Z^y = Z$ whenever $x + y = e$.

1.19.5 The building blocks of the transforms

Consider the following transforms on 2-bit words where addition is bit-wise (that is, XOR):

$$\mathrm{id}_2\, v = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{1.19-12a}$$

$$r_2\, v = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix} \tag{1.19-12b}$$

$$B_2\, v = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ b \end{bmatrix} \tag{1.19-12c}$$

$$Y_2\, v = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \tag{1.19-12d}$$

$$R_2\, v = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a+b \end{bmatrix} \tag{1.19-12e}$$

$$E_2\, v = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ a \end{bmatrix} \tag{1.19-12f}$$

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-reversal, and identity are the transforms obtained as repeated Kronecker-products of the matrices (see section 23.3 on page 462). The transforms are linear over GF(2):

$$Z(\alpha\, a + \beta\, b) = \alpha\, Z(a) + \beta\, Z(b) \tag{1.19-13}$$

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

```
static 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, $R^x = r^x\, B^x$ (see relation 1.19-4a on page 52). The yellow code is the bit-wise Reed-Muller transform (described in section 23.11 on page 486) of a binary word. The symbolic powering is equivalent to selecting individual levels of the transform.

1.20 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]:

```
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. A simplified version is given in [215, sect.7.1.3, rel.90]:

```
    return  0x80808080UL & ( x - 0x01010101UL ) & ~x;
```

To scan for other values than zero (e.g. 0xa5), we can use

```
contains_zero_byte( x ^ 0xa5a5a5a5UL )
```

For very long strings and word sizes of 64 or more bits the following function may be a win [FXT: aux1/bytescan.cc]:

```
ulong long_strlen(const char *str)
// Return length of string starting at str.
{
    ulong x;
    const char *p = str;

    // Alignment: scan bytes up to word boundary:
    while ( (ulong)p % BYTES_PER_LONG )
    {
        if ( 0 == *p )  return  (ulong)(p-str);
        ++p;
    }

    x = *(ulong *)p;
    while ( ! contains_zero_byte(x) )
    {
        p += BYTES_PER_LONG;
        x = *(ulong *)p;
    }

    // now a zero byte is somewhere in x:
    while ( 0 != *p )  { ++p; }

    return  (ulong)(p-str);
}
```

1.21 Inverse and square root modulo 2^n

1.21.1 Computation of the inverse

The inverse modulo 2^n where n is the number of bits in a word can be computed using an iteration (see section 29.1.5 on page 569) with quadratic convergence. The number to be inverted has to be odd [FXT: bits/bit2adic.h]:

```
static 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 2), then the computed value i is the inverse of x modulo m: $i \equiv x^{-1} \bmod m$. It can be used for the *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 $a = b\,x$ (a is divisible by x), so $a\,i \equiv b\,x\,i \equiv b \bmod m$.

1.21.2 Exact division by $C = 2^k \pm 1$

We use the following relation where $Y = 1 - C$:

$$\frac{A}{C} = \frac{A}{1-Y} = A\,(1+Y)\,(1+Y^2)\,(1+Y^4)\,(1+Y^8)\,\ldots\,(1+Y^{2^n}) \quad \bmod Y^{2^{n+1}} \tag{1.21-1}$$

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

$$\frac{A}{C} = A\,(1-2^k)\,(1+2^{k\,2})\,(1+2^{k\,4})\,(1+2^{k\,8})\cdots(1+2^{k\,2^u}) \quad \bmod 2^N \tag{1.21-2}$$

where $k\,2^u \geq N$. For $C = 2^k - 1$ use $(A/C = -A/-C)$

$$\frac{A}{C} = -A\,(1+2^k)\,(1+2^{k\,2})\,(1+2^{k\,4})\,(1+2^{k\,8})\cdots(1+2^{k\,2^u}) \quad \bmod 2^N \tag{1.21-3}$$

The equivalent method for exact division by polynomials (over GF(2)) is given in section 40.1.6 on page 826.

1.21.3 Computation of the square root

```
  x  = ...............................1 =  1        x  = .............................1.1 =  5
 inv = ...............................1             inv = 11..11..11..11..11..11..11..11.1
sqrt = ...............................1
                                                     x  = 11111111111111111111111111111.11 = -5
  x  = 11111111111111111111111111111111 = -1        inv = ..11..11..11..11..11..11..11..11
 inv = 11111111111111111111111111111111
                                                     x  = .............................11. =  6
  x  = ..............................1. =  2
                                                     x  = 11111111111111111111111111111.1. = -6
  x  = 1111111111111111111111111111111. = -2
                                                     x  = .............................111 =  7
  x  = ..............................11 =  3        inv = 1.11.11.11.11.11.11.11.11.11.111
 inv = 1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.11
                                                     x  = 11111111111111111111111111111..1 = -7
  x  = 111111111111111111111111111111.1 = -3        inv = .1..1..1..1..1..1..1..1..1..1..1
 inv = .1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1            sqrt = 1..111..1..11...11......1.11.1.1

  x  = .............................1.. =  4        x  = ............................1... =  8
sqrt = ..............................1.
                                                     x  = 11111111111111111111111111111... = -8
  x  = 111111111111111111111111111111.. = -4
                                                     x  = ............................1..1 =  9
                                                    inv = ..111...111...111...111...111..1
                                                   sqrt = 111111111111111111111111111111.1
```

Figure 1.21-A: Examples of the inverse and square root modulo 2^n 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 control which of the two possible values of the inverse square root is computed. The argument modulo 8 has to be equal to 1.

```
static 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 ( x!=y );
    return  x;
}
```

The square root is computed as $d \cdot 1/\sqrt{d}$:

```
static 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 square root modulo 2^n is something completely different from the integer square root in general. If the argument d is a perfect square, then the result is $\pm\sqrt{d}$. The output of the program [FXT: bits/bit2adic-demo.cc] is shown in figure 1.21-A. For further information see [213, ex.31, p.213], [135, chap.6, p.126], and also [208].

1.22 Radix −2 (minus two) representation

The radix −2 representation of a number n is

$$n \quad = \quad \sum_{k=0}^{\infty} t_k \, (-2)^k \tag{1.22-1}$$

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).

1.22.1 Conversion from binary

```
 k:    bin(k)   m=bin2neg(k)  g=gray(m)   dec(g)
 0:   .......     .......     .......      0 <= 0
 1:   ......1     ......1     ......1      1 <= 1
 2:   .....1.     ....11.     ....1.1      5
 3:   .....11     ....111     ....1..      4
 4:   ....1..     ....1..     ....11.      2
 5:   ....1.1     ....1.1     ....111      3 <= 5
 6:   ....11.     ..11.1.     ..1.111     19
 7:   ....111     ..11.11     ..1.11.     18
 8:   ...1...     ..11...     ..1.1..     20
 9:   ...1..1     ..11..1     ..1.1.1     21
10:   ...1.1.     ..1111.     ..1...1     17
11:   ...1.11     ..11111     ..1....     16
12:   ...11..     ..111..     ..1..1.     14
13:   ...11.1     ..111.1     ..1..11     15
14:   ...111.     ..1..1.     ..11.11      7
15:   ...1111     ..1..11     ..11.1.      6
16:   ..1....     ..1....     ..11...      8
17:   ..1...1     ..1...1     ..11..1      9
18:   ..1..1.     ..1.11.     ..111.1     13
19:   ..1..11     ..1.111     ..111..     12
20:   ..1.1..     ..1.1..     ..1111.     10
21:   ..1.1.1     ..1.1.1     ..11111     11 <= 21
22:   ..1.11.     11.1.1.     1.11111     75
23:   ..1.111     11.1.11     1.1111.     74
24:   ..11...     11.1...     1.111..     76
25:   ..11..1     11.1..1     1.111.1     77
26:   ..11.1.     11.111.     1.11..1     73
27:   ..11.11     11.1111     1.11...     72
28:   ..111..     11.11..     1.11.1.     70
29:   ..111.1     11.11.1     1.11.11     71
30:   ..1111.     11...1.     1.1..11     79
31:   ..11111     11...11     1.1..1.     78
```

Figure 1.22-A: Radix −2 representations and their Gray codes. Lines ending in '<=N' indicate that all values $\leq N$ occur in the last column up to that point.

A surprisingly simple algorithm to compute the coefficients t_k of the radix -2 representation of a binary number is [39, item 128] [FXT: bits/negbin.h]:

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

An example:

```
    14 -->  ..1..1. == 16 - 2 == (-2)^4 + (-2)^1
```

The inverse routine executes the inverse of the two steps in reversed order:

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

Figure 1.22-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 $0, \ldots, k$ for the following values of k (entry A002450 in [312]):

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

1.22.2 Fixed points of the conversion ‡

```
   0: ...........        64: ....1......       256: ..1........       320: ..1.1......
   1: ..........1        65: ....1.....1       257: ..1.......1       321: ..1.1.....1
   4: ........1..        68: ....1...1..       260: ..1.....1..       324: ..1.1...1..
   5: ........1.1        69: ....1...1.1       261: ..1.....1.1       325: ..1.1...1.1
  16: ......1....        80: ....1.1....       272: ..1...1....       336: ..1.1.1....
  17: ......1...1        81: ....1.1...1       273: ..1...1...1       337: ..1.1.1...1
  20: ......1.1..        84: ....1.1.1..       276: ..1...1.1..       340: ..1.1.1.1..
  21: ......1.1.1        85: ....1.1.1.1       277: ..1...1.1.1       341: ..1.1.1.1.1
```

Figure 1.22-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 1.22-B). This is the *Moser – De Bruijn sequence*, entry A000695 in [312]. The generating function of the sequence is

$$\frac{1}{1-x}\sum_{j=0}^{\infty}\frac{4^j\,x^{2^j}}{1+x^{2^j}} \quad = \quad x + 4\,x^2 + 5\,x^3 + 16\,x^4 + 17\,x^5 + 20\,x^6 + 21\,x^7 + 64\,x^8 + 65\,x^9 + \ldots \qquad (1.22\text{-}2)$$

The sequence also appears as exponents in the power series (see also section 38.10.1 on page 750)

$$\prod_{k=0}^{\infty}\left(1+x^{4^k}\right) \quad = \quad 1 + x + x^4 + x^5 + x^{16} + x^{17} + x^{20} + x^{21} + x^{64} + x^{65} + x^{68} + \ \ldots \qquad (1.22\text{-}3)$$

The k-th fixed point is computed by moving all bits of the binary representation of k to position $2\,x$ where $x \geq 0$ is the index of the bit under consideration:

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

The bit-zip function is given in section 1.15 on page 39. The sequence of radix −2 representations of 0, 1, 2, . . ., interpreted as binary numbers, is entry A005351 in [312]:

```
0,1,6,7,4,5,26,27,24,25,30,31,28,29,18,19,16,17,22,23,20,21,106,107,104,105,110,111, ...
```

The corresponding sequence for the negative numbers −1, −2, −3, . . . 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, ...
```

More information about 'non-standard' representations of numbers can be found in [213].

1.22.3 Generating negbin words in order

```
......................::::::::::::::::::::::::::::::::::::::::::
......................111111111111111111111111111111111111111111
......................11111111111111111111111111111111..........
......1111111111111111................1111111111111111..........
......11111111........11111111........11111111........11111111..
..1111....1111....1111....1111....1111....1111....1111....1111..
..11..11..11..11..11..11..11..11..11..11..11..11..11..11..11..11
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1
----------------------------------------------------------------
...........................................:::::::::::::::::::::
...........................................111111111111111111111
...........................................111111111111111111111
...........11111111111111111111111111111111.....................
...........1111111111111111................1111111111111111.....
...11111111........11111111........11111111........11111111.....
...1111....1111....1111....1111....1111....1111....1111....1111.
.11..11..11..11..11..11..11..11..11..11..11..11..11..11..11..11.
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1
```

Figure 1.22-C: Radix −2 representations of the numbers 0 . . . + 63 (top) and 0 . . . − 63 (bottom).

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

```
static 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

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

Decrementing can be done via

```
static 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;
}
```

or via

```
    const ulong m = 0x55555555UL;
    x ^= m;
    ++x;
    x ^= m;
    return x;
```

The functions are quite fast, about 730 million words per second are generated (3 cycles per increment or decrement). Figure 1.22-C shows the generated words in forward (top) and backward (bottom) order. It was created with the program [FXT: bits/negbin2-demo.cc].

1.23 A sparse signed binary representation

```
 0:    .......    .......      0 =
 1:    ......1    ......P      1 =  +1
 2:    .....1.    .....P.      2 =  +2
 3:    .....11    ....P.M      3 =  +4 -1
 4:    ....1..    ....P..      4 =  +4
 5:    ....1.1    ....P.P      5 =  +4 +1
 6:    ....11.    ...P.M.      6 =  +8 -2
 7:    ....111    ...P..M      7 =  +8 -1
 8:    ...1...    ...P...      8 =  +8
 9:    ...1..1    ...P..P      9 =  +8 +1
10:    ...1.1.    ...P.P.     10 =  +8 +2
11:    ...1.11    ..P.M.M     11 =  +16 -4 -1
12:    ...11..    ..P.M..     12 =  +16 -4
13:    ...11.1    ..P.M.P     13 =  +16 -4 +1
14:    ...111.    ..P..M.     14 =  +16 -2
15:    ...1111    ..P...M     15 =  +16 -1
16:    ..1....    ..P....     16 =  +16
17:    ..1...1    ..P...P     17 =  +16 +1
18:    ..1..1.    ..P..P.     18 =  +16 +2
19:    ..1..11    ..P.P.M     19 =  +16 +4 -1
20:    ..1.1..    ..P.P..     20 =  +16 +4
21:    ..1.1.1    ..P.P.P     21 =  +16 +4 +1
22:    ..1.11.    .P.M.M.     22 =  +32 -8 -2
23:    ..1.111    .P.M..M     23 =  +32 -8 -1
24:    ..11...    .P.M...     24 =  +32 -8
25:    ..11..1    .P.M..P     25 =  +32 -8 +1
26:    ..11.1.    .P.M.P.     26 =  +32 -8 +2
27:    ..11.11    .P..M.M     27 =  +32 -4 -1
28:    ..111..    .P..M..     28 =  +32 -4
29:    ..111.1    .P..M.P     29 =  +32 -4 +1
30:    ..1111.    .P...M.     30 =  +32 -2
31:    ..11111    .P....M     31 =  +32 -1
32:    .1.....    .P.....     32 =  +32
```

Figure 1.23-A: Sparse signed binary representations (nonadjacent form, NAF). The symbols '`P`' and '`M`' are respectively used for +1 and −1, dots denote zeros.

```
 0:    ........    ........      0 =
 1:    .......1    .......P      1 =  +1
 2:    ......1.    ......P.      2 =  +2
 4:    .....1..    .....P..      4 =  +4
 5:    .....1.1    .....P.P      5 =  +4 +1
 8:    ....1...    ....P...      8 =  +8
 9:    ....1..1    ....P..P      9 =  +8 +1
10:    ....1.1.    ....P.P.     10 =  +8 +2
16:    ...1....    ...P....     16 =  +16
17:    ...1...1    ...P...P     17 =  +16 +1
18:    ...1..1.    ...P..P.     18 =  +16 +2
20:    ...1.1..    ...P.P..     20 =  +16 +4
21:    ...1.1.1    ...P.P.P     21 =  +16 +4 +1
32:    ..1.....    ..P.....     32 =  +32
33:    ..1....1    ..P....P     33 =  +32 +1
34:    ..1...1.    ..P...P.     34 =  +32 +2
36:    ..1..1..    ..P..P..     36 =  +32 +4
37:    ..1..1.1    ..P..P.P     37 =  +32 +4 +1
40:    ..1.1...    ..P.P...     40 =  +32 +8
41:    ..1.1..1    ..P.P..P     41 =  +32 +8 +1
42:    ..1.1.1.    ..P.P.P.     42 =  +32 +8 +2
64:    .1......    .P......     64 =  +64
```

Figure 1.23-B: The numbers whose negative part in the NAF representation is zero.

An algorithm to compute a representation of a number x as

$$x \quad = \quad \sum_{k=0}^{\infty} s_k \cdot 2^k \quad \text{where} \quad s_k \in \{-1,\, 0,\, +1\} \tag{1.23-1}$$

such that two consecutive digits s_k, s_{k+1} are never simultaneously nonzero is given in [275]. Figure 1.23-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]:

```
static 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*2^k} where d_j \in {-1,0,+1}
//   and no two adjacent digits d_j, d_{j+1} are both nonzero.
// np has bits j set where d_j==+1
// nm has bits j set where d_j==-1
// We have:  x = np - nm
{
    ulong xh = x >> 1;  // x/2
    ulong x3 = x + xh;  // 3*x/2
    ulong c = xh ^ x3;
    np = x3 & c;
    nm = xh & c;
}
```

Converting back to binary is trivial:

```
static inline ulong naf2bin(ulong np, ulong nm)  { return ( np - nm ); }
```

The representation is one example of a *nonadjacent form* (NAF). A method for the computation of certain nonadjacent forms (w-NAF) is given in [255]. A Gray code for the signed binary words is described in section 14.7 on page 315.

If 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, ...], entry A003714 in [312], see figure 1.23-B. The numbers are called the *Fibbinary numbers*.

1.24 Generating bit combinations

1.24.1 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 1.24-A. The method to determine the successor is to determine the lowest block of ones and move its highest bit one position up. Then the rest of the block is moved to the low end of the word [FXT: bits/bitcombcolex.h]:

```
static inline ulong next_colex_comb(ulong x)
{
    ulong r = x & -x;       // lowest set bit
    x += r;                 // replace lowest block by a one left to it

    if ( 0==x )  return 0;  // input was last combination

    ulong z = x & -x;       // first zero beyond lowest block
    z -= r;                 // lowest block  (cf. lowest_block())

    while ( 0==(z&1) )  { z >>= 1; }  // move block to low end of word
    return  x | (z>>1);     // need one bit less of low block
}
```

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.

```
         word   =     set         =  set (reversed)
   1:   ...111  =  { 0, 1, 2 }    =  { 2, 1, 0 }
   2:   ..1.11  =  { 0, 1, 3 }    =  { 3, 1, 0 }
   3:   ..11.1  =  { 0, 2, 3 }    =  { 3, 2, 0 }
   4:   ..111.  =  { 1, 2, 3 }    =  { 3, 2, 1 }
   5:   .1..11  =  { 0, 1, 4 }    =  { 4, 1, 0 }
   6:   .1.1.1  =  { 0, 2, 4 }    =  { 4, 2, 0 }
   7:   .1.11.  =  { 1, 2, 4 }    =  { 4, 2, 1 }
   8:   .11..1  =  { 0, 3, 4 }    =  { 4, 3, 0 }
   9:   .11.1.  =  { 1, 3, 4 }    =  { 4, 3, 1 }
  10:   .111..  =  { 2, 3, 4 }    =  { 4, 3, 2 }
  11:   1...11  =  { 0, 1, 5 }    =  { 5, 1, 0 }
  12:   1..1.1  =  { 0, 2, 5 }    =  { 5, 2, 0 }
  13:   1..11.  =  { 1, 2, 5 }    =  { 5, 2, 1 }
  14:   1.1..1  =  { 0, 3, 5 }    =  { 5, 3, 0 }
  15:   1.1.1.  =  { 1, 3, 5 }    =  { 5, 3, 1 }
  16:   1.11..  =  { 2, 3, 5 }    =  { 5, 3, 2 }
  17:   11...1  =  { 0, 4, 5 }    =  { 5, 4, 0 }
  18:   11..1.  =  { 1, 4, 5 }    =  { 5, 4, 1 }
  19:   11.1..  =  { 2, 4, 5 }    =  { 5, 4, 2 }
  20:   111...  =  { 3, 4, 5 }    =  { 5, 4, 3 }
```

Figure 1.24-A: Combinations $\binom{6}{3}$ in co-lexicographic order. The reversed sets are sorted.

A variant of the method which involves a division appears in [39, item 175]. The routine given here is due to Doug Moore and Glenn Rhoads.

The following routine computes the predecessor of a combination:

```
static inline ulong prev_colex_comb(ulong x)
// Inverse of next_colex_comb()
{
    x = next_colex_comb( ~x );
    if ( 0!=x )  x = ~x;
    return  x;
}
```

The first and last combination can be computed via

```
static 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)
// Must have:  0 <= k <= BITS_PER_LONG
{
    ulong t = ~0UL >> ( BITS_PER_LONG - k );
    if ( k==0 )  t = 0;  // shift with BITS_PER_LONG is undefined
    return t;
}
```

and

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

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 4.

The listing in figure 1.24-A can be created with the program [FXT: bits/bitcombcolex-demo.cc]:

```
    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 );
```

1.24.2 Lexicographic (lex) order

```
          lex (5, 3)                    colex (5, 2)
         word   =   set                 word   =   set
   1:   ..111   =  { 0, 1, 2 }         ...11   =  { 0, 1 }
   2:   .1.11   =  { 0, 1, 3 }         ..1.1   =  { 0, 2 }
   3:   1..11   =  { 0, 1, 4 }         ..11.   =  { 1, 2 }
   4:   .11.1   =  { 0, 2, 3 }         .1..1   =  { 0, 3 }
   5:   1.1.1   =  { 0, 2, 4 }         .1.1.   =  { 1, 3 }
   6:   11..1   =  { 0, 3, 4 }         .11..   =  { 2, 3 }
   7:   .111.   =  { 1, 2, 3 }         1...1   =  { 0, 4 }
   8:   1.11.   =  { 1, 2, 4 }         1..1.   =  { 1, 4 }
   9:   11.1.   =  { 1, 3, 4 }         1.1..   =  { 2, 4 }
  10:   111..   =  { 2, 3, 4 }         11...   =  { 3, 4 }
```

Figure 1.24-B: Combinations $\binom{5}{3}$ in lexicographic order (left). The sets are sorted. The binary words with lex order are the bit-reversed complements of the words with colex order (right).

The binary words corresponding to combinations $\binom{n}{k}$ in lexicographic order are the bit-reversed complements of the words for the combinations $\binom{n}{n-k}$ in co-lexicographic order, see figure 1.24-B. 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:

```
    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 != l );
```

The bit-reversal routine `revbin()` is shown in section 1.14 on page 33. Sections 6.2.1 on page 177 and section 6.2.2 give iterative algorithms for combinations (represented by arrays) in lex and colex order, respectively.

1.24.3 Shifts-order

```
   1:  1....      1:  11...      1:  111..      1:  1111.
   2:  .1...      2:  .11..      2:  .111.      2:  .1111
   3:  ..1..      3:  ..11.      3:  ..111      3:  111.1
   4:  ...1.      4:  ...11      4:  11.1.      4:  11.11
   5:  ....1      5:  1.1..      5:  .11.1      5:  1.111
                  6:  .1.1.      6:  11..1
                  7:  ..1.1      7:  1.11.
                  8:  1..1.      8:  .1.11
                  9:  .1..1      9:  1.1.1
                 10:  1...1     10:  1..11
```

Figure 1.24-C: Combinations $\binom{5}{k}$, for $k = 1, 2, 3, 4$ in shifts-order.

Figure 1.24-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 $[1^k 0^{n-k}]$ and the successor is computed as follows (see figure 1.24-D):

```
 1:  1111...             18:  .11..11
 2:  .1111..             19:  11..1.1  < S
 3:  ..1111.             20:  11...11  < S-2
 4:  ...1111             21:  1.111..  < S-2
 5:  111.1..  < S        22:  .1.111.
 6:  .111.1.             23:  ..1.111
 7:  ..111.1             24:  1.11.1.  < S
 8:  111..1.  < S        25:  .1.11.1
 9:  .111..1             26:  1.11..1  < S
10:  111...1  < S        27:  1.1.11.  < S-2
11:  11.11..  < S-2      28:  .1.1.11
12:  .11.11.             29:  1.1.1.1  < S
13:  ..11.11             30:  1.1..11  < S-2
14:  11.1.1.  < S        31:  1..111.  < S-2
15:  .11.1.1             32:  .1..111
16:  11.1..1  < S        33:  1..11.1  < S
17:  11..11.  < S-2      34:  1..1.11  < S-2
18:  .11..11             35:  1...111  < S-2
```

Figure 1.24-D: Updates with combinations $\binom{7}{4}$: simple split 'S', split second 'S-2', easy case unmarked.

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 ($[0^n]$, $[0^{n-1}1]$, $[10^{n-k}1^{k-1}]$), 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]:

```
class bit_comb_shifts
{
public:
    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();
    }

    ulong first(ulong n, ulong k)
    {
        s_ = 0;
        x_ =  last_comb(k, n);

        if ( k>1 )  last_ = first_comb(k-1) | (1UL<<(n_-1));  // [10000111]
        else        last_ = k;  //  [000001] or [000000]

        return x_;
    }

    ulong first()  { return first(n_, k_); }

    ulong next()
    {
        if ( 0==(x_&1) )  // easy case:
        {
            ++s_;
            x_ >>= 1;
            return  x_;
        }
        else  // splitting cases:
```

```
        {
            if ( x_ == last_ )  return 0;  // combination was last

            x_ <<= s_;  s_ = 0;  // shift back to the left
            ulong b = x_ & -x_;  // lowest bit


            if ( b!=1UL )  // simple split
            {
                x_ -= (b>>1);  // move rightmost bit to the right
                return x_;
            }
            else  // split second block and attach first
            {
                ulong t = low_ones(x_);  //  block of ones at lower end
                x_ ^= t;  // remove block
                ulong b2 = x_ & -x_;  // (second) lowest bit

                b2 >>= 1;
                x_ -= b2;  // move bit to the right

                // attach block:
                do  { t<<=1; }  while ( 0==(t&x_) );
                x_ |= (t>>1);
                return x_;
            }
        }
    }
};
```

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.

1.24.4 Minimal-change order ‡

The following routine is due to Doug Moore [FXT: bits/bitcombminchange.h]:

```
static inline ulong igc_next_minchange_comb(ulong x)
// Return the inverse Gray code of the next combination in minimal-change order.
// Input must be the inverse Gray code of the current combination.
{
    ulong g = rev_gray_code(x);
    ulong i = 2;
    ulong cb; // ==candidate bits;
    do
    {
        ulong y = (x & ~(i-1)) + i;
        ulong j = lowest_one(y) << 1;
        ulong h = !!(y & j);
        cb = ((j-h) ^ g) & (j-i);
        i = j;
    }
    while ( 0==cb );

    return  x + lowest_one(cb);
}
```

It can be used as suggested by the routine

```
static inline ulong next_minchange_comb(ulong x, ulong last)
// Not efficient, just to explain the usage of igc_next_minchange_comb()
// Must have: last==igc_last_comb(k, n)
{
    x = inverse_gray_code(x);
    if ( x==last )  return 0;
    x = igc_next_minchange_comb(x);
    return  gray_code(x);
}
```

The auxiliary function `igc_last_comb()` is (32-bit version only)

```
static inline ulong igc_last_comb(ulong k, ulong n)
// Return the (inverse Gray code of the) last combination
// as in igc_next_minchange_comb()
{
    if ( 0==k )  return 0;

    const ulong f = 0xaaaaaaaaUL >> (BITS_PER_LONG-k);  // == first_sequency(k);
    const ulong c =  ~0UL >> (BITS_PER_LONG-n);  // == first_comb(n);
    return c ^ (f>>1);
    // =^=  (by Doug Moore)
//    return  ((1UL<<n) - 1) ^ (((1UL<<k) - 1) / 3);
}
```

Successive combinations differ in exactly two positions. For example, with $n = 5$ and $k = 3$:

```
   x       inverse_gray_code(x)
 ..111       ..1.1 == first_sequency(k)
 .11.1       .1..1
 .111.       .1.11
 .1.11       .11.1
 11..1       1...1
 11.1.       1..11
 111..       1.111
 1.1.1       11..1
 1.11.       11.11
 1..11       111.1 == igc_last_comb(k, n)
```

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

For greater efficiency one may prefer code which avoids the repeated computation of the inverse Gray code, for example:

```
    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 );
```

```
      n = 6  k = 2              n = 6  k = 3              n = 6  k = 4
  ....11  ....1.  ....1.     ...111  ...1.1  ...1..     ..1111  ..1.1.  ..1...
  ...11.  ...1..  ....1.     ..11.1  ..1..1  ....1.     .11.11  .1..1.  ....1.
  ...1.1  ...11.  ....1.     ..111.  ..1.11  ....1.     .1111.  .1.1..  ....1.
  ..11..  ..1...  ...1..     ..1.11  ..11.1  ...1..     .111.1  .1.11.  ...1..
  ..1.1.  ..11..  ....1.     .11..1  .1...1  ....1.     .1.111  .11.1.  ..1...
  ..1..1  ..111.  ....1.     .11.1.  .1..11  ...1..     11..11  1...1.  ....1.
  .11...  .1....  ..1...     .111..  .1.111  ....1.     11.11.  1..1..  ....1.
  .1.1..  .11...  ...1..     .1.1.1  .11..1  ....1.     11.1.1  1..11.  ....1.
  .1..1.  .111..  ....1.     .1.11.  .11.11  ....1.     1111..  1.1...  ...1..
  .1...1  .1111.  ....1.     .1..11  .111.1  ...1..     111.1.  1.11..  ....1.
  11....  1.....  .1....     11...1  1....1  ....1.     111..1  1.111.  ...1..
  1.1...  11....  ..1...     11..1.  1...11  ...1..     1.1.11  11..1.  ....1.
  1..1..  111...  ...1..     11.1..  1..111  ..1...     1.111.  11.1..  ....1.
  1...1.  1111..  ....1.     111...  1.1111  ....1.     1.11.1  11.11.  ...1..
  1....1  11111.  ....1.     1.1..1  11...1  ....1.     1..111  111.1.  ..1...
                             1.1.1.  11..11  ...1..
                             1.11..  11.111  ....1.
                             1..1.1  111..1  ....1.
                             1..11.  111.11  ....1.
                             1...11  1111.1  ...1..
```

Figure 1.24-E: Minimal-change combinations, their inverse Gray codes, and the differences of the inverse Gray codes. The differences are powers of 2.

The difference of the inverse Gray codes of two successive combinations is always a power of 2, see figure 1.24-E (the listings were created with the program [FXT: bits/bitcombminchange-demo.cc]). With this observation we can derive a different version that checks the pattern of the change:

```
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
        // of the new and one bit of the old pattern:
        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 has a bit-count 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;
}
```

1.25 Generating bit subsets of a given word

1.25.1 Counting order

To generate all subsets of the set of ones of a binary word we use the sparse counting idea shown in section 1.8.1 on page 20. The implementation is [FXT: `class bit_subset` in bits/bitsubset.h]:

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

public:
    bit_subset(ulong v) : u_(0), v_(v)  { ; }
    ~bit_subset() { ; }
    ulong current()  const  { return u_; }
    ulong next()  { u_ = (u_ - v_) & v_;  return u_; }
    ulong prev()  { u_ = (u_ - 1 ) & 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 `[...11.1.]` the following sequence of words is produced by subsequent `next()`-calls:

```
......1.
....1...
....1.1.
...1....
...1..1.
...11...
...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].

1.25.2 Minimal-change 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 set bits are separated by any amount of zeros. In fact, we want to find the single bit that changed from 0 to 1. The bits that switched from 0 to 1 in the transition from the word A to B can also be isolated via `X=B&~A`. The implementation is [FXT: `class bit_subset_gray` in bits/bitsubset-gray.h]:

```
class bit_subset_gray
{
public:
    bit_subset S_;
    ulong g_;  // subsets in Gray code order
    ulong h_;  // 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);  h_=highest_one(v);  g_=0;  return g_; }
    ulong first()  { S_.first();  g_=0;  return g_; }
  [--snip--]
```

With the word `[...11.1.]` the following sequence of words is produced by subsequent `next()`-calls:

```
......1.
....1.1.
....1...
...11...
...11.1.
...1..1.
...1....
........
```

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 `u0` and `u1`:

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

    ulong last(ulong v)  { S_.last(v);  h_=highest_one(v);  g_=h_;  return g_; }
    ulong last()  { S_.last();  g_=h_;  return g_; }
};
```

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

1.26 Binary words in lexicographic order for subsets

1.26.1 Next and previous word in lexicographic order

```
 1:  1...  =  8    {0}
 2:  11..  = 12    {0, 1}
 3:  111.  = 14    {0, 1, 2}
 4:  1111  = 15    {0, 1, 2, 3}
 5:  11.1  = 13    {0, 1, 3}
 6:  1.1.  = 10    {0, 2}
 7:  1.11  = 11    {0, 2, 3}
 8:  1..1  =  9    {0, 3}
 9:  .1..  =  4    {1}
10:  .11.  =  6    {1, 2}
11:  .111  =  7    {1, 2, 3}
12:  .1.1  =  5    {1, 3}
13:  ..1.  =  2    {2}
14:  ..11  =  3    {2, 3}
15:  ...1  =  1    {3}
```

Figure 1.26-A: Binary words corresponding to nonempty 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.

```
[0:  ......   =  0 *]   16:  .1...1   = 17      32:  1....1   = 33      48:  11..11   = 51
 1:  .....1   =  1 *    17:  .1..11   = 19      33:  1...11   = 35      49:  11..1.   = 50
 2:  ....11   =  3      18:  .1..1.   = 18 *    34:  1...1.   = 34 *    50:  11.1.1   = 53
 3:  ....1.   =  2      19:  .1.1.1   = 21      35:  1..1.1   = 37      51:  11.111   = 55
 4:  ...1.1   =  5      20:  .1.111   = 23      36:  1..111   = 39      52:  11.11.   = 54
 5:  ...111   =  7      21:  .1.11.   = 22      37:  1..11.   = 38      53:  11.1..   = 52
 6:  ...11.   =  6 *    22:  .1.1..   = 20      38:  1..1..   = 36      54:  111..1   = 57
 7:  ...1..   =  4      23:  .11..1   = 25      39:  1.1..1   = 41      55:  111.11   = 59
 8:  ..1..1   =  9      24:  .11.11   = 27      40:  1.1.11   = 43      56:  111.1.   = 58
 9:  ..1.11   = 11      25:  .11.1.   = 26      41:  1.1.1.   = 42      57:  1111.1   = 61
10:  ..1.1.   = 10 *    26:  .111.1   = 29      42:  1.11.1   = 45      58:  111111   = 63
11:  ..11.1   = 13      27:  .11111   = 31      43:  1.1111   = 47      59:  11111.   = 62
12:  ..1111   = 15      28:  .1111.   = 30      44:  1.111.   = 46      60:  1111..   = 60 *
13:  ..111.   = 14      29:  .111..   = 28      45:  1.11..   = 44      61:  111...   = 56
14:  ..11..   = 12      30:  .11...   = 24      46:  1.1...   = 40      62:  11....   = 48
15:  ..1...   =  8      31:  .1....   = 16      47:  11...1   = 49      63:  1.....   = 32
```

Figure 1.26-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) binary words in lexicographic order with respect to the subsets shown in figure 1.26-A can be generated by successive calls to the following function [FXT: bits/bitlex.h]:

```
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;
        return  x;
    }
    else  // lowest bit at word end
    {
        x ^= x0;  // clear lowest bit
        x0 = x & -x;  // new lowest bit ...
        x0 >>= 1;  x -= x0;  // ... is moved one to the right
        return  x;
    }
}
```

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 $n-1$, we generate the 2^n subsets of the word of n ones. The function is used as follows [FXT: bits/bitlex-demo.cc]:

```
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:

```
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;  // clear lowest bit
        return x;
    }
    else
    {
        x += x0;  // move lowest bit to the left
        x |= 1;   // set rightmost bit
        return x;
    }
}
```

The sequence of all n-bit words is generated by 2^n calls to `prev_lexrev()`, starting with zero. The words corresponding to subsets of the 6-element set are shown in figure 1.26-B. The sequence [1, 3, 2, 5, 7, 6, 4, 9, ...] in the right column is entry A108918 in [312].

The rate of generation using `next()` is about 274 million per second and about 253 million per second with `prev()`. An equivalent routine for arrays is given in section 8.1.2 on page 203. The routines are useful for a special version of fast Walsh transforms described in section 23.5.3 on page 472.

1.26.2 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]:

```
static inline ulong negidx2lexrev(ulong k)
{
    ulong z = 0;
    ulong h = highest_one(k);
    while ( k )
    {
        while ( 0==(h&k) )  h >>= 1;
        z ^= h;
        ++k;
        k &= h - 1;
    }
    return  z;
}
```

Let the inverse function be $T(x)$, then we have $T(0)=0$ and, with $h(x)$ being the highest power of 2 not greater than x,

$$T(x) \quad = \quad h(x) - 1 + \begin{cases} T\left(x - h(x)\right) & \text{if } x - h(x) \neq 0 \\ h(x) & \text{otherwise} \end{cases} \tag{1.26-1}$$

The ranking algorithm starts with the lowest bit:

```
static 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;
}
```

1.26.3 Minimal decompositions into terms $2^k - 1$ ‡

```
....1 1      ....1 =  1    =   1
...11 2      ...1. =  2    =   1 + 1
...1. 1      ...11 =  3    =   3
..1.1 2      ..1.. =  4    =   3 + 1
..111 3      ..1.1 =  5    =   3 + 1 + 1
..11. 2      ..11. =  6    =   3 + 3
..1.. 1      ..111 =  7    =   7
.1..1 2      .1... =  8    =   7 + 1
.1.11 3      .1..1 =  9    =   7 + 1 + 1
.1.1. 2      .1.1. = 10    =   7 + 3
.11.1 3      .1.11 = 11    =   7 + 3 + 1
.1111 4      .11.. = 12    =   7 + 3 + 1 + 1
.111. 3      .11.1 = 13    =   7 + 3 + 3
.11.. 2      .111. = 14    =   7 + 7
.1... 1      .1111 = 15    =  15
1...1 2      1.... = 16    =  15 + 1
1..11 3      1...1 = 17    =  15 + 1 + 1
1..1. 2      1..1. = 18    =  15 + 3
1.1.1 3      1..11 = 19    =  15 + 3 + 1
1.111 4      1.1.. = 20    =  15 + 3 + 1 + 1
1.11. 3      1.1.1 = 21    =  15 + 3 + 3
1.1.. 2      1.11. = 22    =  15 + 7
11..1 3      1.111 = 23    =  15 + 7 + 1
11.11 4      11... = 24    =  15 + 7 + 1 + 1
11.1. 3      11..1 = 25    =  15 + 7 + 3
111.1 4      11.1. = 26    =  15 + 7 + 3 + 1
11111 5      11.11 = 27    =  15 + 7 + 3 + 1 + 1
1111. 4      111.. = 28    =  15 + 7 + 3 + 3
111.. 3      111.1 = 29    =  15 + 7 + 7
11... 2      1111. = 30    =  15 + 15
1.... 1      11111 = 31    =  31
```

Figure 1.26-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 $x = \sum_k \left(2^k - 1\right)$ (right columns) equals the bit count.

The least number of terms needed in the sum $x = \sum_k \left(2^k - 1\right)$ equals the number of bits of the lex-word as shown in figure 1.26-C. The number can be computed as

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

Alternatively, we 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 in [312]:

```
1,2,1,2,3,2,1,2,3,2,3,4,3,2,1,2,3,2,3,4,3,2,3,4,3,4,5,4,3,2,1,2,3,2,3,...
```

The following function can be used to compute the sequence:

```
void S(ulong f, ulong n) // A100661
{
    static int s = 0;
    ++s;
    cout << s << ",";
    for (ulong m=1; m<n; m<<=1)  S(f+m, m);
    --s;
    cout << s << ",";
}
```

If 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

$$Z(x) \quad := \quad \frac{-1 + 2\,(1-x)\,\prod_{n=1}^{\infty}\left(1 + x^{2^n-1}\right)}{(1-x)^2} \quad = \tag{1.26-2}$$

$$1 + 2x + x^2 + 2x^3 + 3x^4 + 2x^5 + x^6 + 2x^7 + 3x^8 + 2x^9 + 3x^{10} + 4x^{11} + 3x^{12} + 2x^{13} \quad + \ldots$$

1.26.4 The sequence of fixed points ‡

```
   0:   ...........         514:    .1.......1.
   1:   ..........1         540:    .1....111..
   6:   ........11.         556:    .1...1.11..
  10:   .......1.1.       [--snip--]
  18:   ......1..1.        1556:   .11....1.1..
  34:   .....1...1.        1572:   .11...1..1..
  60:   .....1111..        1604:   .11..1...1..
  66:   ....1....1.        1668:   .11.1....1..
  92:   ....1.111..        1796:   .111.....1..
 108:   ....11.11..        2040:   .11111111...
 116:   ....111.1..        2050:   1.........1.
 130:   ...1.....1.        2076:   1......111..
 156:   ...1..111..        2092:   1.....1.11..
 172:   ...1.1.11..        2100:   1.....11.1..
 180:   ...1.11.1..        2124:   1....1..11..
 204:   ...11..11..        2132:   1....1.1.1..
 212:   ...11.1.1..        2148:   1....11..1..
 228:   ...111..1..       [--snip--]
 258:   ..1......1.        4644:  1..1...1..1..
 284:   ..1...111..        4676:  1..1..1...1..
 300:   ..1..1.11..        4740:  1..1.1....1..
 308:   ..1..11.1..        4868:  1..11.....1..
 332:   ..1.1..11..        5112:  1..1111111...
 340:   ..1.1.1.1..        5132:  1.1......11..
 356:   ..1.11..1..        5140:  1.1.....1.1..
 396:   ..11...11..        5156:  1.1....1..1..
 404:   ..11..1.1..        5188:  1.1...1...1..
 420:   ..11.1..1..        5252:  1.1..1....1..
 452:   ..111...1..        5380:  1.1.1.....1..
```

Figure 1.26-D: Fixed points of the binary to lex-rev conversion.

The sequence of fixed points of the conversion to and from indices starts as

```
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, ...
```

This sequence is entry A079471 in [312]. The values as bit patterns are shown in figure 1.26-D. The crucial observation is that a word is a fixed point 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:

```
static inline bool is_lexrev_fixed_point(ulong x)
// Return whether x is a fixed point in the prev_lexrev() - sequence
{
    if ( x & 1 )
    {
        if ( 1==x )  return  true;
        else         return  false;
    }
    else
    {
        ulong w = bit_count(x);
        if ( w != (w & -w) )  return  false;
        if ( 0==x )  return  true;
        return  0 != ( (x & -x) & w );
    }
}
```

Alternatively, use either of the following tests:

```
x == negidx2lexrev(x)
x == lexrev2negidx(x)
```

1.26.5 Recursive generation and relation to a power series ‡

```
    Start: 1
    Rules:
      0 --> 0
      1 --> 110
    -------------
    0:   (#=2)
      1
    1:   (#=4)
      110
    2:   (#=8)
      1101100
    3:   (#=16)
      110110011011000
    4:   (#=32)
      1101100110110001101100110110000
    5:   (#=64)
      110110011011000110110011011000011011001101100011011001101100000
```

Figure 1.26-E: String substitution with rules $\{0 \to 0, 1 \mapsto 110\}$.

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

```
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
}
```

By calling `C(0, 64, 0)` we generate the list of words shown in figure 1.26-B with the all-zeros word moved to the last position. A slight modification of the function

```
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 in [312])

$$\prod_{n=1}^{\infty}\left(1+x^{2^n-1}\right) \quad = \quad 1+x+x^3+x^4+x^7+x^8+x^{10}+x^{11}+x^{15}+x^{16}+\ldots \qquad (1.26\text{-}3)$$

By calling `A(0, 32)` we generate the sequence

```
    1,1,0,1,1,0,0,1,1,0,1,1,0,0,0,1,1,0,1,1,0,0,1,1,0,1,1,0,0,0,0, ...
```

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 by the string substitution shown in figure 1.26-E.

The sequence of sums, prepended by 1,

$$1+x\,\frac{\prod_{n=1}^{\infty}\left(1+x^{2^n-1}\right)}{1-x} \quad = \quad 1+1\,x+2\,x^2+2\,x^3+3\,x^4+4\,x^5+4\,x^6+\ldots \qquad (1.26\text{-}4)$$

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, ...
```

This sequence is entry A046699 in [312]. We have $a(1) = a(2) = 1$ and the sequence satisfies the peculiar recurrence

$$a(n) \quad = \quad a(n-a(n-1)) + a(n-1-a(n-2)) \qquad \text{for} \quad n > 2 \qquad (1.26\text{-}5)$$

1.27 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: bits/fibrep.h]

```
static inline bool is_fibrep(ulong f)
{
    return  ( 0==(f&(f>>1)) );
}
```

The following functions convert between the binary and the Fibonacci representation:

```
static inline ulong bin2fibrep(ulong b)
// Return Fibonacci representation of b
// Limitation: the first Fibonacci number greater
//  than b must be representable as ulong.
// 32 bit:  b < 2971215073=F(47) [F(48)=4807526976 > 2^32]
// 64 bit:  b < 12200160415121876738=F(93) [F(94) > 2^64]
{
    ulong f0=1, f1=1, s=1;
    while ( f1<=b )  { ulong t = f0+f1;  f0=f1;  f1=t;  s<<=1; }
    ulong f = 0;
    while ( b )
    {
        s >>= 1;
        if ( b>=f0 )  { b -= f0;  f^=s; }
        { ulong t = f1-f0;  f1=f0;  f0=t; }
    }
    return f;
}
```

```
static inline ulong fibrep2bin(ulong f)
// Return binary representation of f
// Inverse of bin2fibrep().
{
    ulong f0=1, f1=1;
    ulong b = 0;
    while ( f )
    {
        if ( f&1 )   b += f1;
        { ulong t=f0+f1;  f0=f1;  f1=t; }
        f >>= 1;
    }
    return b;
}
```

1.27.1 Lexicographic order

```
 0: ........     11: ...1.1..     22: .1.....1     33: .1.1.1.1     44: 1..1..1.
 1: .......1     12: ...1.1.1     23: .1....1.     34: 1.......     45: 1..1.1..
 2: ......1.     13: ..1.....     24: .1...1..     35: 1......1     46: 1..1.1.1
 3: .....1..     14: ..1....1     25: .1...1.1     36: 1.....1.     47: 1.1.....
 4: .....1.1     15: ..1...1.     26: .1..1...     37: 1....1..     48: 1.1....1
 5: ....1...     16: ..1..1..     27: .1..1..1     38: 1....1.1     49: 1.1...1.
 6: ....1..1     17: ..1..1.1     28: .1..1.1.     39: 1...1...     50: 1.1..1..
 7: ....1.1.     18: ..1.1...     29: .1.1....     40: 1...1..1     51: 1.1..1.1
 8: ...1....     19: ..1.1..1     30: .1.1...1     41: 1...1.1.     52: 1.1.1...
 9: ...1...1     20: ..1.1.1.     31: .1.1..1.     42: 1..1....     53: 1.1.1..1
10: ...1..1.     21: .1......     32: .1.1.1..     43: 1..1...1     54: 1.1.1.1.
```

Figure 1.27-A: All 55 Fibonacci words with 8 bits in lexicographic order.

The 8-bit Fibonacci words are shown in figure 1.27-A. To generate all Fibonacci words in lexicographic order, use the function [FXT: bits/fibrep.h]

```
static 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 == [*]0 010101   // x == [*]0 01010
    ulong y = x | (x>>1);  // y == [*]? 011111   // y == [*]? 01111
    ulong z = y + 1;       // z == [*]? 100000   // z == [*]? 10000
    z = z & -z;            // z == [0]0 100000   // z == [0]0 10000
    x ^= z;                // x == [*]0 110101   // x == [*]0 11010
    x &= ~(z-1);           // x == [*]0 100000   // x == [*]0 10000

    return x;
}
```

The routine can be used to generate all n-bit words as shown in [FXT: bits/fibrep2-demo.cc]:

```
const ulong f = 1UL << n;
ulong t = 0;
do
{
    // visit(t)
    t = next_fibrep(t);
}
while ( t!=f );
```

The reversed order can be generated via

```
ulong f = 1UL << n;
do
{
    f = prev_fibrep(f);
    // visit(f)
}
while ( f );
```

which uses the function (64-bit version)

```
static inline ulong prev_fibrep(ulong x)
// With x the Fibonacci representation of n
// return Fibonacci representation of n-1.
{
    // 2 examples:         //  ex. 1             //  ex.2
    //                     // x == [*]0 100000   // x == [*]0 10000
    ulong y = x & -x;      // y == [0]0 100000   // y == [0]0 10000
    x ^= y;                // x == [*]0 000000   // x == [*]0 00000
    ulong m = 0x5555555555555555UL;  // m == ...01010101
    if ( m & y )  m >>= 1; // m == ...01010101   // m == ...0101010
    m &= (y-1);            // m == [0]0 010101   // m == [0]0 01010
    x ^= m;                // x == [*]0 010101   // x == [*]0 01010
    return x;
}
```

The forward version generates about 180 million words per second, the backward version about 170 million words per second.

1.27.2 Gray code order ‡

A Gray code for the binary Fibonacci words (shown in figure 1.27-B) can be derived from the Gray code of the radix −2 representations (see section 1.22 on page 58) of binary words whose difference is of the form

```
  1    ...............1
  3    ..............11
  5    .............1.1
  9    ............1..1
 19    ...........1..11
 37    ..........1..1.1
 73    .........1..1..1
147    ........1..1..11
293    .......1..1..1.1
```

The algorithm is to try these values as increments starting from the least, same as for the minimal-change combination described in section 1.24.4 on page 66. 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]:

```
class bit_fibgray
// Fibonacci Gray code with binary words.
{
public:
    ulong x_;  // current Fibonacci word
    ulong k_;  // aux
    ulong fw_, lw_;  // first and last Fibonacci word in Gray code
    ulong mw_;  // max(fw_, lw_)
    ulong n_;   // Number of bits

public:
    bit_fibgray(ulong n)
    {
        n_ = n;
        fw_ = 0;
```

```
  j:       k(j)      k(j)-k(j-1)   x=bin2neg(k)    gray(x)
  1:    ....11...1    ..........    ...111...1    ...1..1..1 =  27
  2:    ....11....    .........1    ...111....    ...1..1... =  26
  3:    ....1.1111    .........1    ...111..11    ...1..1.1. =  28
  4:    ....1.11..    ........11    ...11111..    ...1....1. =  23
  5:    ....1.1.11    .........1    ...1111111    ...1...... =  21
  6:    ....1.1.1.    .........1    ...111111.    ...1.....1 =  22
  7:    ....1.1..1    .........1    ...1111..1    ...1...1.1 =  25
  8:    ....1.1...    .........1    ...1111...    ...1...1.. =  24
  9:    ....1...11    .......1.1    ...11..111    ...1.1.1.. =  32
 10:    ....1...1.    .........1    ...11..11.    ...1.1.1.1 =  33
 11:    ....1....1    .........1    ...11....1    ...1.1...1 =  30
 12:    ....1.....    .........1    ...11.....    ...1.1.... =  29
 13:    .....11111    .........1    ...11...11    ...1.1..1. =  31
 14:    ......11..    .....1..11    .....111..    .....1..1. =  10
 15:    ......1.11    .........1    .....11111    .....1.... =   8
 16:    ......1.1.    .........1    .....1111.    .....1...1 =   9
 17:    ......1..1    .........1    .....11..1    .....1.1.1 =  12
 18:    ......1...    .........1    .....11...    .....1.1.. =  11
 19:    ........11    .......1.1    .......111    .......1.. =   3
 20:    ........1.    .........1    .......11.    .......1.1 =   4
 21:    .........1    .........1    .........1    .........1 =   1
 22:    ..........    .........1    ..........    .......... =   0
 23:    1111111111    .........1    ........11    ........1. =   2
 24:    11111111..    ........11    ......11..    ......1.1. =   7
 25:    1111111.11    .........1    ......1111    ......1... =   5
 26:    1111111.1.    .........1    ......111.    ......1..1 =   6
 27:    111111...1    ......1..1    ....11...1    ....1.1..1 =  19
 28:    111111....    .........1    ....11....    ....1.1... =  18
 29:    11111.1111    .........1    ....11..11    ....1.1.1. =  20
 30:    11111.11..    ........11    ....1111..    ....1...1. =  15
 31:    11111.1.11    .........1    ....111111    ....1..... =  13
 32:    11111.1.1.    .........1    ....11111.    ....1....1 =  14
 33:    11111.1..1    .........1    ....111..1    ....1..1.1 =  17
 34:    11111.1...    .........1    ....111...    ....1..1.. =  16
```

Figure 1.27-B: Gray code for the binary Fibonacci words (rightmost column).

```
        for (ulong m=(1UL<<(n-1)); m!=0; m>>=3)  fw_ |= m;
        lw_ = fw_ >> 1;
        if ( 0==(n&1) )  { ulong t=fw_; fw_=lw_; lw_=t; }  // swap first/last
        mw_ = ( lw_>fw_ ? lw_ : fw_ );
        x_ = fw_;
        k_ = inverse_gray_code(fw_);
        k_ = neg2bin(k_);
    }

    ~bit_fibgray()  {;}

    ulong next()
    // Return next word in Gray code.
    // Return ~0 if current word is the last one.
    {
        if ( x_ == lw_ )  return ~0UL;
        ulong s = n_;  // shift
        while ( 1 )
        {
            --s;
            ulong c = 1 | (mw_ >> s);  // possible difference for negbin word
            ulong i = k_ - c;
            ulong x = bin2neg(i);
            x ^= (x>>1);

            if ( 0==(x&(x>>1)) )  // is_fibrep(x)
            {
                k_ = i;
                x_ = x;
                return x;
            }
        }
    }
};
```

About 130 million words per second are generated. The program [FXT: bits/bitfibgray-demo.cc] shows how to use the class, figure 1.27-B was created with it. Section 14.2 on page 305 gives a recursive algorithm for Fibonacci words in Gray code order.

1.28 Binary words and parentheses strings ‡

```
 0   .... P   [empty string]      .....   [empty string]
 1   ...1 P   ()                  ....1   ()
 2   ..1.                         ...11   (())
 3   ..11 P   (())                ..1.1   ()()
 4   .1..                         ..111   ((()))
 5   .1.1 P   ()()                .1.11   (()())
 6   .11.                         .11.1   ()(())
 7   .111 P   ((()))              .1111   (((())))
 8   1...                         1..11   (())()
 9   1..1                         1.1.1   ()()()
10   1.1.                         1.111   ((()()))
11   1.11 P   (()())              11.11   (()(()))
12   11..                         111.1   ()((()))
13   11.1 P   ()(())              11111   ((((()))))
14   111.
15   1111 P   (((())))
```

Figure 1.28-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.

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 a 'P' in figure 1.28-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. To find out when to terminate, one adds up +1 for each opening parenthesis and −1 for a closing parenthesis. After the ones in the binary word have been scanned, the s closing parentheses have to be added where s is the value of the sum [FXT: bits/parenwords.h]:

```
static 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
}
```

The 5-bit binary words that are valid 'paren words' together with the corresponding strings are shown in figure 1.28-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, the word is not a paren word. A function to determine whether a word is a paren word is

```
static 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, 47, 51, 53, 55, 59, 61, 63, ...`

of nonzero integers x so that `is_parenword(x)` returns `true` is entry A036991 in [312]. If we fix the number of paren pairs, then the following functions generate the least and biggest valid paren words. The first paren word is a block of n ones at the low end:

```
static 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:

```
static 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);
}
```

```
......11111 = ((((()))))    ...1...1111 = (((()))())    ..1....1111 = (((())))()
.....1.1111 = ((()()))))    ...1..1.111 = ((()())())    ..1...1.111 = ((()()))()
.....11.111 = ((()(())))    ...1..11.11 = (()(())())    ..1...11.11 = (()(()))()
.....111.11 = (()((())))    ...1..111.1 = ()((())())    ..1...111.1 = ()((()))()
.....1111.1 = ()(((())))    ...1.1..111 = ((())()())    ..1..1..111 = ((())())()
....1..1111 = (((())()))    ...1.1.1.11 = (()()()())    ..1..1.1.11 = (()()())()
....1.1.111 = ((()()()))    ...1.1.11.1 = ()(()()())    ..1..1.11.1 = ()(()())()
....1.11.11 = (()(()()))    ...1.11..11 = (())(()())    ..1..11..11 = (())(())()
....1.111.1 = ()((()()))    ...1.11.1.1 = ()()(()())    ..1..11.1.1 = ()()(())()
....11..111 = ((())(()))    ...11...111 = ((()))(())    ..1.1...111 = ((()))()()
....11.1.11 = (()()(()))    ...11..1.11 = (()())(())    ..1.1..1.11 = (()())()()
....11.11.1 = ()(()(()))    ...11..11.1 = ()(())(())    ..1.1..11.1 = ()(())()()
....111..11 = (())((()))    ...11.1..11 = (())()(())    ..1.1.1..11 = (())()()()
....111.1.1 = ()()((()))    ...11.1.1.1 = ()()()(())    ..1.1.1.1.1 = ()()()()()
```

Figure 1.28-B: The 42 binary words corresponding to all valid pairings of 5 parentheses, in colex order.

The sequence of all binary words corresponding to n pairs of parens in colex order can be generated with the following (slightly cryptic) function:

```
static 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:
    {
        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;
        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):

```
    ulong n = 4;  // Number of paren pairs
    ulong pn = 2*n+1;
    char *str = new char[n+1];  str[n] = 0;
    ulong x = first_parenword(n);
    while ( x )
    {
        print_bin("  ", x, pn);
        parenword2str(x, str);
        cout << " = " << str << endl;

        x = next_parenword(x);
    }
```

Its output with $n = 5$ is shown in figure 1.28-B. The 1,767,263,190 paren words for $n = 19$ are generated at a rate of about 169 million words per second. Chapter 15 on page 323 gives a different formulation of the algorithm.

Knuth [215, ex.23, sect.7.1.3] gives a very elegant routine for generating the next paren word, the comments are MMIX instructions:

```
static inline ulong next_parenword(ulong x)
{
    const ulong m0 = -1UL/3;
    ulong t = x ^ m0;                // XOR t, x, m0;
    if ( (t&x)==0 )  return 0;       // current is last
    ulong u = (t-1) ^ t;             // SUBU u, t, 1;  XOR u, t, u;
    ulong v = x | u;                 // OR v, x, u;
    ulong y = bit_count( u & m0 );   // SADD y, u, m0;
    ulong w = v + 1;                 // ADDU w, v, 1;
    t = v & ~w;                      // ANDN t, v, w;
    y = t >> y;                      // SRU y, t, y;
    y += w;                          // ADDU y, w, y;
    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. Treating the case of easy update separately as in the other version, we get a rate of about 137 million words per second.

1.29 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.

1.29.1 A restricted method

```
    ................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...............    bits  15 ...
    ........1...............1.......    bits   7 ...
    ....1.......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 ...
```

Figure 1.29-A: Mask with primitives for permuting bits with 32-bit words (top), and words with ones at the highest bit of each block (bottom).

We can specify a subset of all permutations by selecting bit-blocks of the masks as shown for 32-bit words in figure 1.29-A (top). Subsets of the blocks of the masks can be determined with the bits of a word by considering the highest bit of each block (bottom of the figure). We use all bits of a word (except for the highest bit) to select the blocks where the bits defined by the block and those left to it should be swapped. An implementation of the implied algorithm is given in [FXT: bits/bitperm1-demo.cc]. Arrays are used to give more readable code:

```
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);
            }
        }
    }
}
```

The routine for the inverse permutation differs in a single line:

```
for (long s=1; s<n; 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:    7
   4:    3  11x
   2:    1   5x  9  13x
   1:    0   2x  4   6x  8  10x 12  14x
a=01326754CDFEAB98   result
```

This control word leads to the Gray permutation (see 2.12 on page 128). Assume we use words with N bits. We cannot (for $N > 2$) specify 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 $j \le n$ and $0 \le k < 2^{n-j}$, choosing p for each interval arbitrarily ($0 \le p < 2^{n-j}$). 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_{2y}, a_{2y+1} must contain two elements $2z$, $2z+1$ ($j = 1$). The bit-reversal is computed with a control word where all bits are set. Alas, the (important!) zip permutation (bit-zip, see section 1.15 on page 38) is unreachable.

A machine instruction could choose between the two routines via the highest bit in the control word.

1.29.2 A general method

All permutations of $N = 2^n$ elements can be specified with n control words of N bits. Assume we have a machine instruction that collects bits according to a control word. An eight bit example:

```
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
    abdgcefh   result, bits separated according to x
```

We need n such instructions that work on all length-2^k sub-words for $1 \le k \le n$. For example, the instruction working on half words of a 16-bit word would work as

```
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
    abdgcefh 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.    k= 1  (2-bit sub-words)
```

while this

```
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.    k= 5 (32 bit sub-words)
```

results in gathering the even and odd indexed bits in the halfwords.

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

```
                                0123 4567 89ab cdef
11111111........    k= 4  ==> 89ab cdef 0123 4567
1111....1111....    k= 3  ==> cdef 89ab 4567 0123
11..11..11..11..    k= 2  ==> efcd ab89 6745 2301
1.1.1.1.1.1.1.1.    k= 1  ==> fedc ba98 7654 3210
```

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 computed 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. When starting with full words we get:

```
                                 0123 4567 89ab cdef
1.1. 1.1. 1.1. 1.1.  k= 4  ==>   1357 9bdf 0246 8ace
1.1. 1.1. 1.1. 1.1.  k= 3  ==>   37bf 159d 26ae 048c
1.1. 1.1. 1.1. 1.1.  k= 2  ==>   7f3b 5d19 6e2a 4c08
1.1. 1.1. 1.1. 1.1.  k= 1  ==>   f7b3 d591 e6a2 c480
```

The result is different when starting with 2-bit sub-words:

```
                                 0123 4567 89ab cdef
1.1. 1.1. 1.1. 1.1.  k= 1  ==>   1032 5476 98ba dcfe
1.1. 1.1. 1.1. 1.1.  k= 2  ==>   0213 4657 8a9b cedf
1.1. 1.1. 1.1. 1.1.  k= 3  ==>   2367 0145 abef 89cd
1.1. 1.1. 1.1. 1.1.  k= 4  ==>   3715 bf9d 2604 ae8c
```

There are $\binom{2z}{z}$ possibilities to have z bits set in a $2z$-bit word. There are 2^{n-k} length-2^k sub-words in a 2^n-bit word so the number of valid control words for that step is

$$\left[\binom{2^k}{2^{k-1}}\right]^{2^{n-k}}$$

The product of the number of valid words in all steps gives the number of permutations:

$$(2^n)! \quad = \quad \prod_{k=1}^{n} \left[\binom{2^k}{2^{k-1}}\right]^{2^{n-k}} \tag{1.29-1}$$

1.30 CPU instructions often missed

1.30.1 Essential

- Bit-shift and bit-rotate instructions that work properly for shifts greater than 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 33.
- Instructions that return the index of highest or lowest set bit in a word. They must execute fast.
- 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, computing $A \cdot C - B \cdot D$ and $A \cdot D + B \cdot C$ from A, B, C, and D.
- A sum-diff instruction, computing $A + B$ and $A - B$ from A and B. This can serve as a primitive for fast orthogonal transforms.
- An instruction to swap registers. Even better, a conditional version of that.

1.30.2 Nice to have

- A parity bit for the complete machine word. The parity of a word is the number of bits modulo 2, not the complement of it. Even better, an instruction for the inverse Gray code, see section 1.16 on page 41.
- A bit-count instruction, see section 1.8 on page 18. This would also give the parity at bit zero.
- An instruction for computing the index of the i-th set bit of a word, see section 1.10 on page 25. This would be useful even if execution takes a dozen cycles.
- A random number generator, LHCAs (see section 41.8 on page 878) may be candidates. At the very least: a decent entropy source.
- A conditional version of more than just the move instruction, possibly as an instruction prefix.
- A bit-zip and a bit-unzip instruction, see section 1.15 on page 38. Note this is polynomial squaring over GF(2).
- Primitives for permutations of bits, see section 1.29.2 on page 81. A bit-gather and a bit-scatter instruction for sub-words of all sizes a power of 2 would allow for arbitrary permutations (see [FXT: bits/bitgather.h] and [FXT: bits/bitseparate.h] for versions working on complete words).
- Multiplication corresponding to XOR as addition. This is the multiplication without carries used for polynomials over GF(2), see section 40.1 on page 822.

1.31 Some space filling curves ‡

1.31.1 The Hilbert curve

A rendering of the Hilbert curve (named after David Hilbert [182]) is shown in figure 1.31-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 38.9.1 on page 748).

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. So 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]:

```
static inline ulong hilbert_p(ulong t)
// Let dx,dy be the horizontal,vertical move
// with step t of the Hilbert curve.
// Return  zero if (dx+dy)==-1, else one (then: (dx+dy)==+1).
// Algorithm: count number of threes in radix 4
{
    ulong d = (t & 0x5555555555555555UL) & ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
    return  parity( d );
}
```

If 1 is returned the step is to the right or upwards. The function can be slightly optimized as follows (64-bit version only):

```
static inline ulong hilbert_p(ulong t)
{
    t &= ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
```

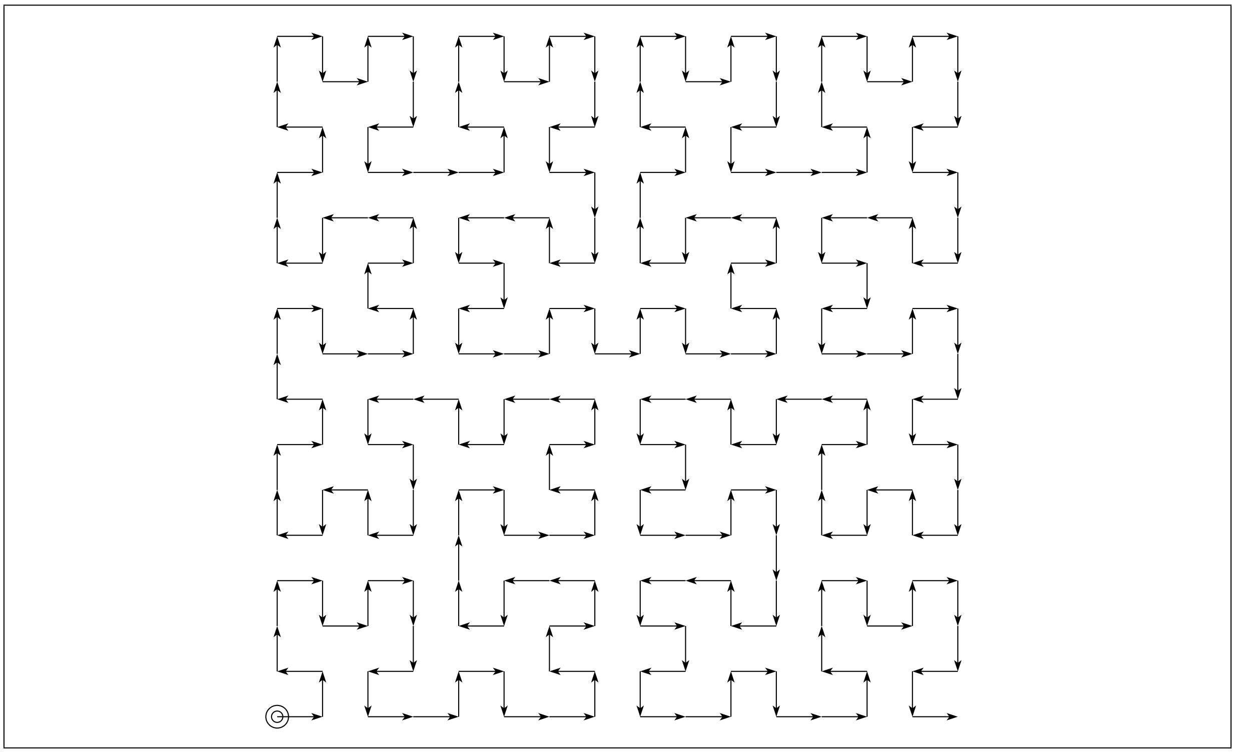

Figure 1.31-A: The first 255 segments of the Hilbert curve.

```
 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 1.31-B: Moves and turns of the Hilbert curve.

```
    t ^= t>>2;
    t ^= t>>4;
    t ^= t>>8;
    t ^= t>>16;
    t ^= t>>32;
    return  t & 1;
}
```

The corresponding value for m can be computed as:

```
static inline ulong hilbert_m(ulong t)
// Let dx,dy be the horizontal,vertical move
// with step t of the Hilbert curve.
// Return  zero if (dx-dy)==-1, else one (then: (dx-dy)==+1).
{
    return  hilbert_p( -t );
}
```

If the values for p and m are equal the step is in horizontal direction. It remains to merge the values of p and m into a 2-bit value d that encodes the direction of the move:

```
static inline ulong hilbert_dir(ulong t)
// Return d encoding the following move with the Hilbert curve.
//
// d \in {0,1,2,3} as follows:
//   d : direction
//   0 : right (+x:  dx=+1, dy= 0)
//   1 : down  (-y:  dx= 0, dy=-1)
//   2 : up    (+y:  dx= 0, dy=+1)
//   3 : left  (-x:  dx=-1, dy= 0)
{
    ulong p = hilbert_p(t);
    ulong m = hilbert_m(t);
    ulong d = p ^ (m<<1);
    return  d;
}
```

To print the value of d symbolically, we can print the value of `(">v^<")[d]`. The sequence of moves can also be generated by the string substitution process shown in figure 1.31-C.

```
Start: A
Rules:
  A --> D>A^A<C
  B --> C<BvB>D
  C --> BvC<C^A
  D --> A^D>DvB
  > --> >
  < --> <
  ^ --> ^
  v --> v
-------------
0:   (#=1)
  A
1:   (#=7)
  D>A^A<C
2:   (#=31)
  A^D>DvB>D>A^A<C^D>A^A<C<BvC<C^A
3:   (#=127)
  D>A^A<C^A^D>DvB>A^D>DvBvC<BvB>D>A^D>DvB>D>A^A<C^D>A^A<C<BvC<C^A^A^D>DvB>D>A^A<C^D> ...
```

Figure 1.31-C: Moves of the Hilbert curve by a string substitution process, the symbols '`A`', '`B`', '`C`', and '`D`', are ignored when drawing the curve.

The turn u between steps can be computed as

```
static inline int hilbert_turn(ulong t)
// Return the turn (left or right) with the steps
//   t and t-1 of the Hilbert curve.
// Returned value is
//   0 for no turn
//  +1 for right turn
//  -1 for left turn
{
    ulong d1 = hilbert_dir(t);
    ulong d2 = hilbert_dir(t-1);
    d1 ^= (d1>>1);
    d2 ^= (d2>>1);
    ulong u = d1 - d2;
    // at this point, symbolically:  cout << ("+.-0+.-")[ u + 3 ];
    if ( 0==u )  return 0;
    if ( (long)u<0 )  u += 4;
    return  (1==u ? +1 : -1);
}
```

To print the value of u symbolically, we can print `("-0+")[u+1];`.

The values of p and m, followed by the direction and turn of the Hilbert curve are shown in figure 1.31-B. The list was created with the program [FXT: bits/hilbert-moves-demo.cc]. Figure 1.31-A was created with the program [FXT: bits/hilbert-texpic-demo.cc]. The computation of a function whose series coefficients are ± 1 and $\pm i$ according to the Hilbert curve is described in section 38.9 on page 747.

A finite state machine (FSM) for the conversion from a 1-dimensional coordinate (linear coordinate of the curve) to the pair of coordinates x and y of the Hilbert curve is described in [39, item 115]. At each step two bits of input are processed. The array `htab[]` serves as lookup table for the next state and two bits of the result. The FSM has an internal state of two bits [FXT: bits/lin2hilbert.cc]:

```
void
lin2hilbert(ulong t, ulong &x, ulong &y)
// Transform linear coordinate t to Hilbert x and 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;
}
```

OLD				NEW			
C_0	C_1	A_I	B_I	X_I	Y_I	C_0	C_1
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	1	1	0	0
0	0	1	1	1	0	0	1
0	1	0	0	1	1	1	1
0	1	0	1	0	1	0	1
0	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	1	0
1	0	1	0	1	1	1	0
1	0	1	1	0	1	1	1
1	1	0	0	1	1	0	1
1	1	0	1	1	0	1	1
1	1	1	0	0	0	1	1
1	1	1	1	0	1	1	0

NEW				OLD			
C_0	C_1	X_I	Y_I	A_I	B_I	C_0	C_1
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	1	1	0	1
0	0	1	1	1	0	0	0
0	1	0	0	1	0	0	1
0	1	0	1	0	1	0	1
0	1	1	0	1	1	0	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
1	0	1	0	0	1	1	0
1	0	1	1	1	0	1	0
1	1	0	0	1	0	1	1
1	1	0	1	1	1	1	0
1	1	1	0	0	1	1	1
1	1	1	1	0	0	0	1

Figure 1.31-D: The original table from [39] 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.

The table used is defined (see figure 1.31-D) as

```
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 1.31-D) is

```
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:

```
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;
}
```

1.31.2 The Z-order

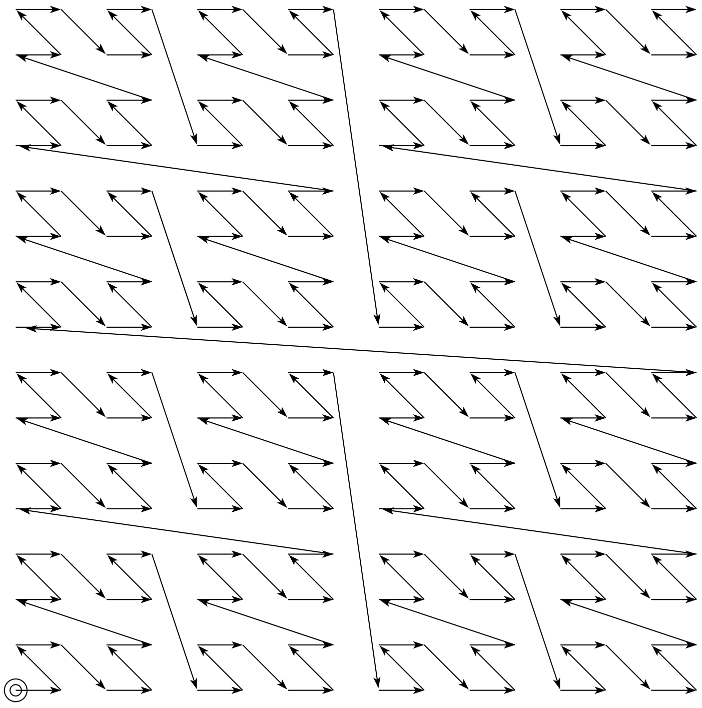
Figure 1.31-E: The first 255 segments of the Z-order curve.

A 2-dimensional space-filling curve in *Z-order* traverses all points in each quadrant before it enters the next. Figure 1.31-E shows a rendering of the Z-order curve, created with the program [FXT: bits/zorder-texpic-demo.cc]. The conversion between a linear parameter to a pair of coordinates is done by separating the bits at the even and odd indices [FXT: bits/zorder.h]:

```
static inline void lin2zorder(ulong t, ulong &x, ulong &y)  { bit_unzip2(t, x, y); }
```

The routine `bit_unzip2()` is described in section 1.15 on page 38. The inverse is

```
static inline ulong zorder2lin(ulong x, ulong y)  { return bit_zip2(x, y); }
```

The next pair can be computed with the following (constant amortized time) routine:

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

The previous pair is computed similarly:

```
static 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 easily to more dimensions:

```
static 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 );
}
```

```
static inline void zorder3d_prev(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 with the Hilbert curve there are steps where the curve advances more than one unit.

1.31.3 Curves via paper-folding sequences

The *paper-folding sequence*, entry A014577 in [312], starts as [FXT: bits/bit-paper-fold-demo.cc]:

```
11011001110010011101100011001001110110011100100011011000110010011 ...
```

The k-th element ($k > 0$) is one if $k = 2^t \cdot (4u + 1)$, entry A091072 in [312]:

```
1, 2, 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 21, 25, 26, 29, 32, 33, ...
```

The k-th element of the paper-folding sequence can be computed by testing the value of the bit left to the lowest (that is, rightmost) one in the binary expansion of k [FXT: bits/bit-paper-fold.h]:

```
static inline bool bit_paper_fold(ulong k)
{
    ulong h = k & -k;  // == lowest_one(k)
    k &= (h<<1);
    return  ( k==0 );
}
```

About 550 million values per second are generated. We use `bool` as return type to indicate that only zero or one is returned. The value can be used as an integer of arbitrary type, there is no need for a cast.

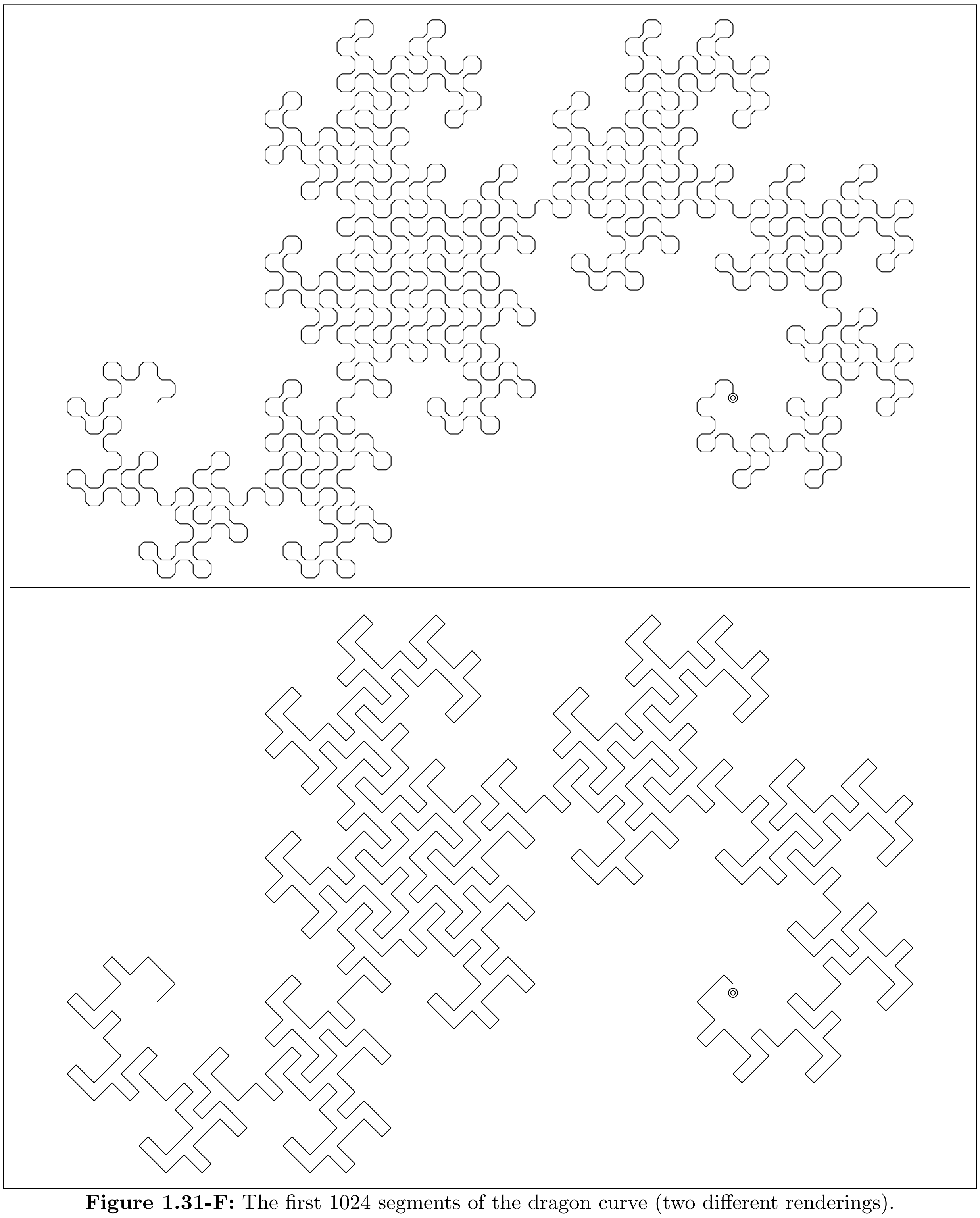
Figure 1.31-F: The first 1024 segments of the dragon curve (two different renderings).

1.31.3.1 The dragon curve

Another name for the sequence is *dragon curve sequence*, because a space filling curve known as dragon curve (or *Heighway dragon*) can be generated if we interpret a one as 'turn left' and a zero as 'turn right'. The top of figure 1.31-F shows the first 1024 segments of the curve (created with [FXT: bits/dragon-curve-texpic-demo.cc]). As some points are visited twice we draw the turns with cut off corners, for the (left) turn $A \to B \to C$:

```
          C                  C
          |                  |
          |    drawn as      |
          |                  /
      A --- B           A --/B
```

The code is given in [FXT: aux0/tex-line.cc]. The first few moves of the curve can be found by repeatedly folding a strip of paper. Always pick up the right side and fold to the left. Unfold the paper and adjust all corners to be 90 degrees. This gives the first few segments of the dragon curve.

When all angles are replaced by diagonals between the midpoints of the lines

```
          C                  C
          |
          |    drawn as      /
          |                 /
      A --- B           A  / B
```

then the curve appears as shown at the bottom of figure 1.31-F.

```
Start: 0
Rules:
  0 --> 01
  1 --> 21
  2 --> 23
  3 --> 03
-------------
0: 0
1: 01
2: 0121
3: 01212321
4: 0121232123032321
5: 01212321230323212303010323032321
6: 0121232123032321230301032303232123030103012101032303010323032321
   +^-^-v-^-v+v-v-^-v+v+^+v-v+v-v-^-v+v+^+v+^-^+^+v-v+v+^+v-v+v-v-^
```

Figure 1.31-G: Moves of the dragon curve generated by a string substitution process.

The net rotation of the dragon-curve after k steps, as multiple of the right angle, can be computed by counting the ones in the Gray code of k. Take the result modulo 4 to ignore multiples of 360 degree [FXT: bits/bit-paper-fold.h]:

```
static inline bool bit_dragon_rot(ulong k)  { return  bit_count( k ^ (k>>1) ) & 3; }
```

The sequence of rotations is entry A005811 in [312]:

```
seq   = 0 1 2 1 2 3 2 1 2 3 4 3 2 3 2 1 2 3 4 3 4 5 4 3 2 3 4 3 2 3 2 1 2 3 ...
mod 4 = 0 1 2 1 2 3 2 1 2 3 0 3 2 3 2 1 2 3 0 3 0 1 0 3 2 3 0 3 2 3 2 1 2 3 ...
move  = + ^ - ^ - v - ^ - v + v - v - ^ - v + v + ^ + v - v + v - v - ^ - v ...
```

The sequence of moves (as symbols, last row) can be computed with [FXT: bits/dragon-curve-moves-demo.cc]. A function related to the paper-folding sequence is described in section 38.8.3 on page 744.

1.31.3.2 The alternate paper-folding sequence

If the strip of paper is folded alternately from the left and right, then another paper-folding sequence is obtained. It is entry A106665 in [312] and it starts as [FXT: bits/bit-paper-fold-alt-demo.cc]:

```
1001110010001101100111011000110010011100100011001001110110001101 1 ...
```

Compute the sequence via [FXT: bits/bit-paper-fold.h]

```
static inline bool bit_paper_fold_alt(ulong k)
{
    ulong h = k & -k;  // == lowest_one(k)
```

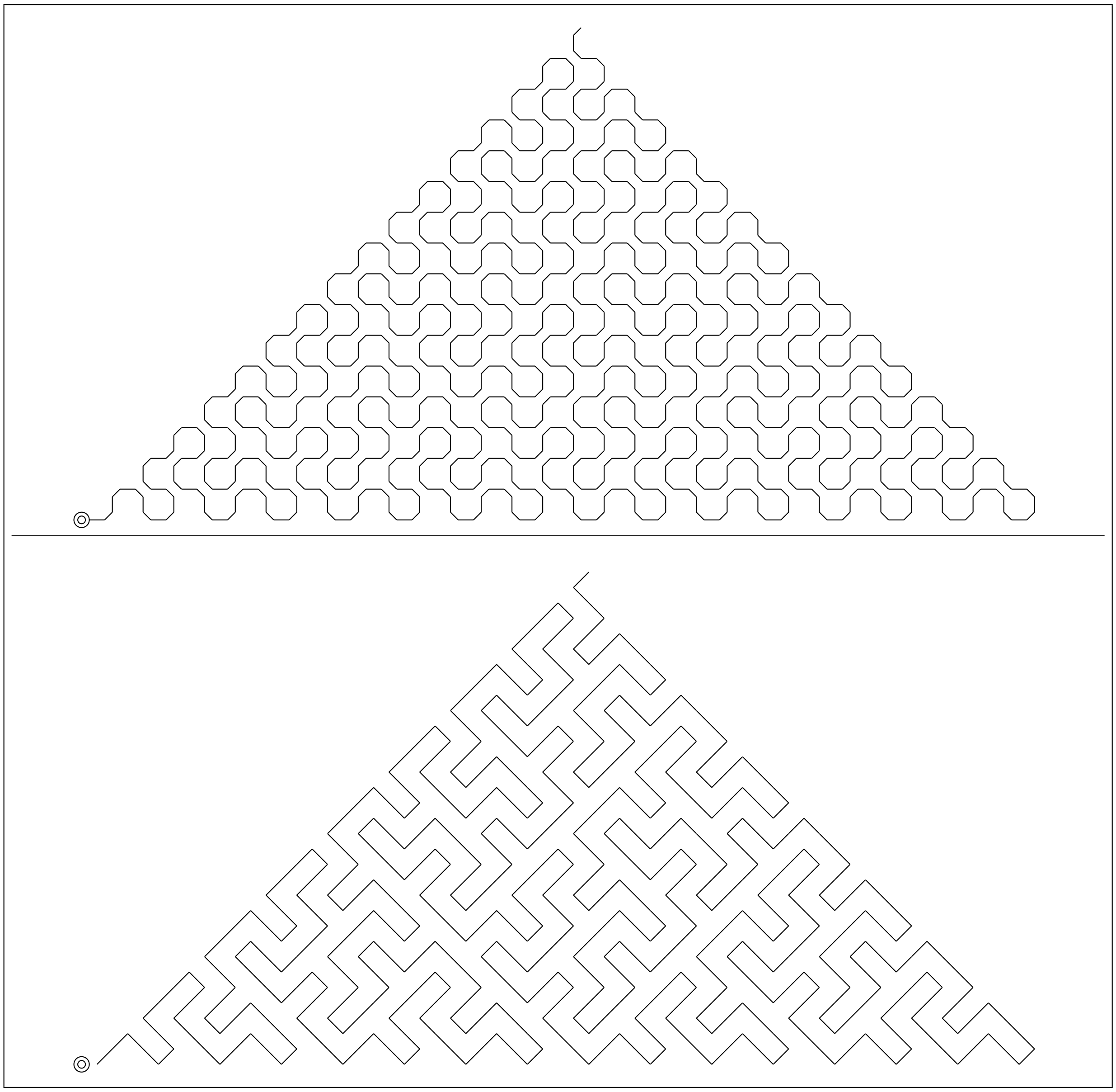
Figure 1.31-H: The first 512 segments of the curve from the alternate paper-folding sequence.

```
Start: 0
Rules:
  0 --> 01
  1 --> 03
  2 --> 23
  3 --> 21
-------------
0: 0
1: 01
2: 0103
3: 01030121
4: 0103012101032303
5: 01030121010323030103012123210121
6: 0103012101032303010301212321012101030121010323032321230301032303
   +^+v+^-^+^+v-v+v+^+v+^-^-v-^+^-^+^+v+^-^+^+v-v+v-v-^-v+v+^+v-v+v
```

Figure 1.31-I: Moves of the alternate curve generated by a string substitution process.

```
Start: L
Rules:
  L --> L+R+L-R
  R --> L+R-L-R
  + --> +
  - --> -
-------------
0:   (#=1)
  L
1:   (#=7)
  L+R+L-R
2:   (#=31)
  L+R+L-R+L+R-L-R+L+R+L-R-L+R-L-R
3:   (#=127)
  L+R+L-R+L+R-L-R+L+R+L-R-L+R-L-R+L+R+L-R+L+R-L-R-L+R+L-R-L+R-L-R+L+R+L-R+L+R-L-R+L+ ...
```

```
Start: L
Rules:
  L --> R+L+R-L
  R --> R+L-R-L
  + --> +
  - --> -
-------------
0:   (#=1)
  L
1:   (#=7)
  R+L+R-L
2:   (#=31)
  R+L-R-L+R+L+R-L+R+L-R-L-R+L+R-L
3:   (#=127)
  R+L-R-L+R+L+R-L-R+L-R-L-R+L+R-L+R+L-R-L+R+L+R-L+R+L-R-L-R+L+R-L+R+L-R-L+R+L+R-L-R+ ...
```

Figure 1.31-J: Moves and turns of the dragon curve (top) and alternate dragon curve (bottom).

```
    h <<= 1;
    ulong t = h & (k ^ 0xaaaaaaaaUL);  // 32-bit version
    return  ( t!=0 );
}
```

About 413 million values per second are generated. By interpreting the sequence of zeros and ones as turns we again obtain triangular space-filling curves shown in figure 1.31-H. The orientations can be computed as

```
static inline ulong bit_paper_fold_alt_rot(ulong k)
// Return total rotation (as multiple of the right angle)
//   after k steps in the alternate paper-folding curve.
// k=      0, 1, 2, 3, 4, 5, ...
// seq(k)= 0, 1, 0, 3, 0, 1, 2, 1, 0, 1, 0, 3, 2, 3, 0, ...
// move =  +  ^  +  v  +  ^  -  ^  +  ^  +  v  -  v  +
// (+==right, -==left, ^==up, v==down).
// Algorithm: count the ones in  (w ^ gray_code(k)).
{
    const ulong w = 0xaaaaaaaaUL;  // 32-bit version
    return  bit_count( w ^ (k ^ (k>>1)) ) & 3;  // modulo 4
}
```

Figure 1.31-J shows a different string substitution process for the generation of the rotations (symbols '`+`' and '`-`') for the paper-folding sequences, both symbols '`L`' and '`R`' are interpreted as a unit move in the current direction.

If the constant in the routine is replaced by a parameter w, then its bits determine whether a left or a right fold was made at each step:

```
static inline bool bit_paper_fold_general(ulong k, ulong w)
{
    ulong h = k & -k;  // == lowest_one(k)
    h <<= 1;
    ulong t = h & (k^w);
    return  ( t!=0 );
}
```

1.31.4 Terdragon and hexdragon

The *terdragon* curve turns to the left or right by 120 degrees depending to the sequence

```
0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, ...
```

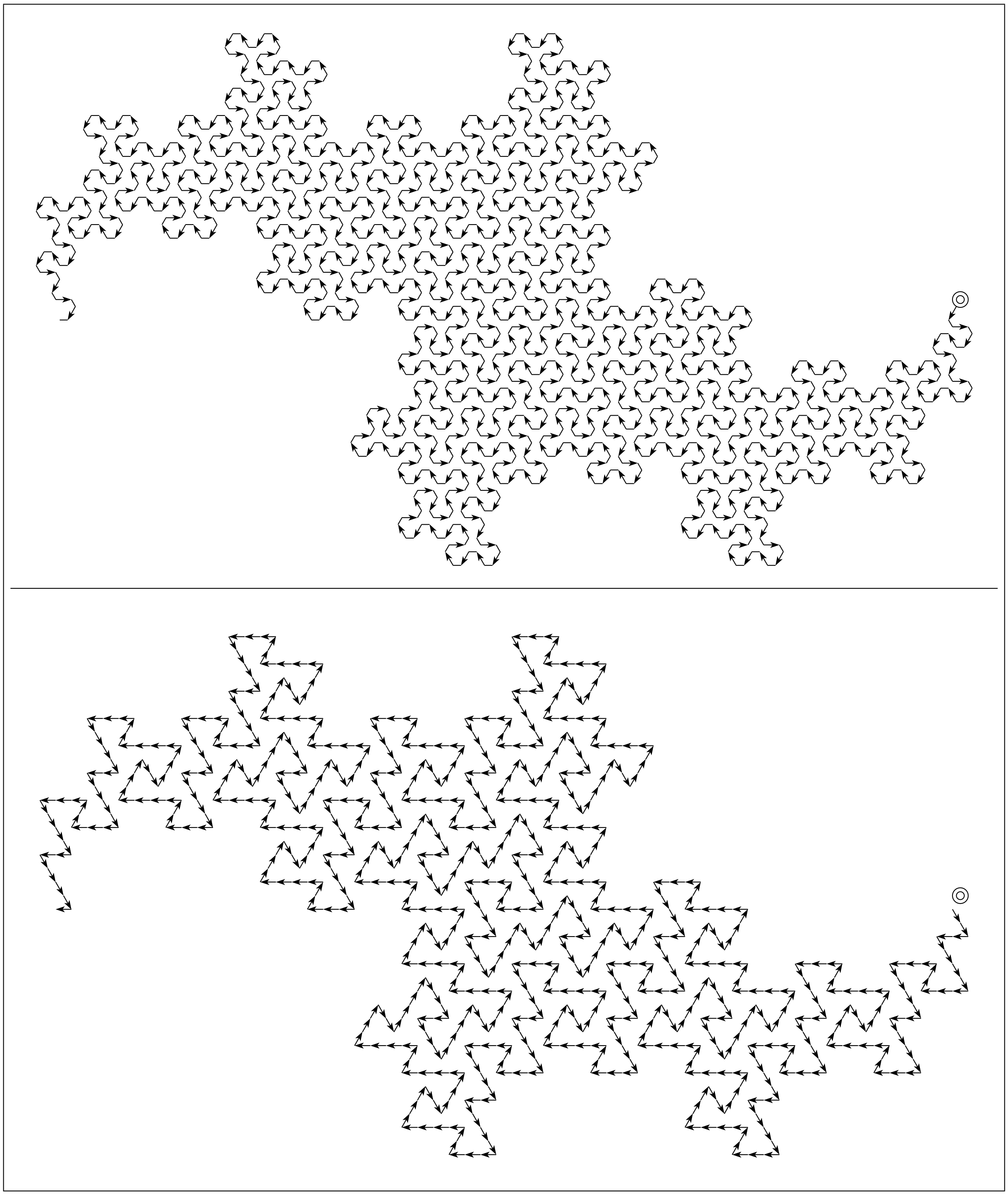

Figure 1.31-K: The first 729 segments of the terdragon (two different renderings).

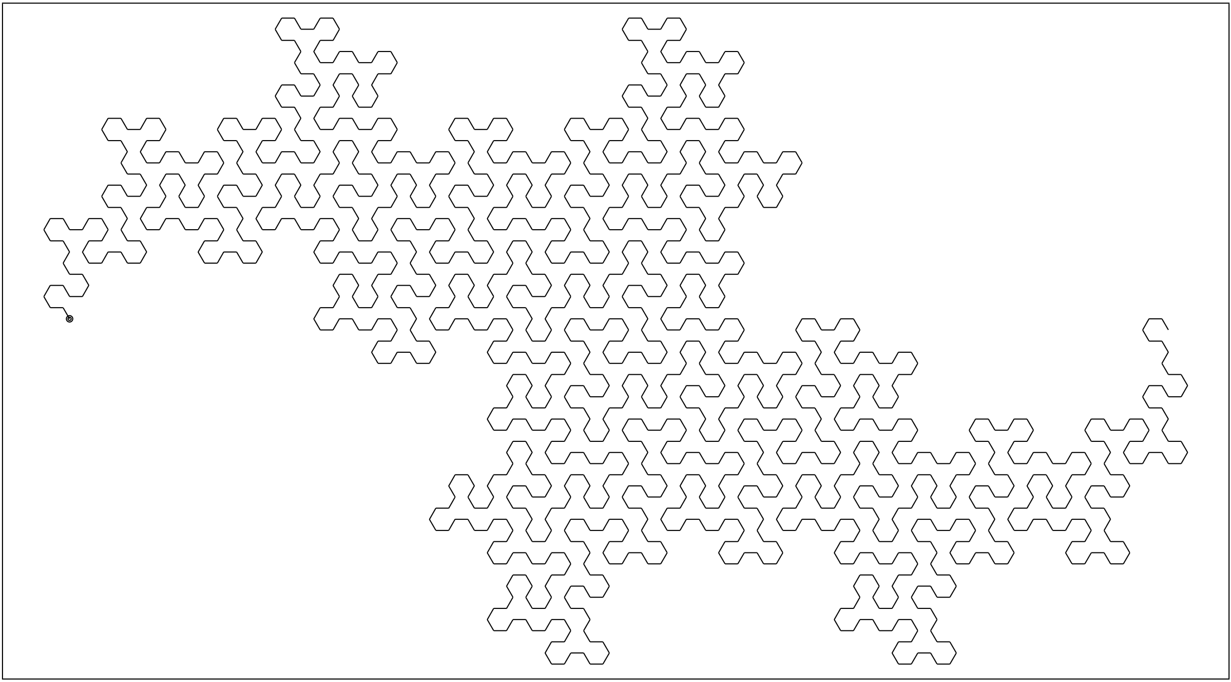

Figure 1.31-L: The first 729 segments of the hexdragon.

```
 Start: 0
 Rules:
   0 --> 010
   1 --> 011
 -------------
 0:   (#=1)
   0
 1:   (#=3)
   010
 2:   (#=9)
   010011010
 3:   (#=27)
   010011010010011011010011010
 4:   (#=81)
   010011010010011011010011010010011010010011011010011011010011010010011011010011010
```

```
 Start: F
 Rules:
   F --> F+F-F
   + --> +
   - --> -
 -------------
 0:   (#=1)
   F
 1:   (#=5)
   F+F-F
 2:   (#=17)
   F+F-F+F+F-F-F+F-F
 3:   (#=53)
   F+F-F+F+F-F-F+F-F+F+F-F+F+F-F-F+F-F-F+F-F+F+F-F-F+F-F
 4:   (#=161)
   F+F-F+F+F-F-F+F-F+F+F-F+F+F-F-F+F-F-F+F-F+F+F-F-F+F-F+F+F-F+F+F-F-F+F-F+F+F-F+F+F-  ...
```

Figure 1.31-M: Turns of the terdragon curve, generated by string substitution (top), alternative process for the moves and turns (bottom, identify ‘+’ with ‘0’ and ‘-’ with ‘1’).

```
Start: F
Rules:
  F --> F+L+F-L-F
  + --> +
  - --> -
  L --> L
-------------
0:   (#=1)
  F
1:   (#=9)
  F+L+F-L-F
2:   (#=33)
  F+L+F-L-F+L+F+L+F-L-F-L-F+L+F-L-F
3:   (#=105)
  F+L+F-L-F+L+F+L+F-L-F-L-F+L+F-L-F+L+F+L+F-L-F+L+F+L+F-L-F-L-F+L+F-L-F-L-F+L+F-L-F+ ...
```

Figure 1.31-N: String substitution process for the hexdragon.

The sequence is entry A080846 in [312], it can be generated via the string substitution with rules $0 \mapsto 101$ and $1 \mapsto 011$, see figure 1.31-M. A fast method to compute the sequence is based on radix-3 counting: let $C_1(k)$ be the number of ones in the radix-3 expansion of k, the sequence is one if $C_1(k+1) < C_1(k)$ [FXT: bits/bit-dragon3.h]:

```
static inline bool bit_dragon3_turn(ulong &x)
// Increment the radix-3 word x and
// return whether the number of ones in x is decreased.
{
    ulong s = 0;
    while ( (x & 3) == 2 )  { x >>= 2;  ++s; }  // scan over nines
//    if ( (x & 3) == 0 )  ==> incremented word will have one more 1
//    if ( (x & 3) == 1 )  ==> incremented word will have one less 1
    bool tr = ( (x & 3) != 0 );  // incremented word will have one less 1
    ++x;  // increment next digit
    x <<= (s<<1);  // shift back
    return  tr;
}
```

About 220 million values per second are generated. Two renderings of the first 729 segments of the curve are shown in figure 1.31-K (created with [FXT: bits/dragon3-texpic-demo.cc]).

If we replace each turn by 120 degrees (followed by a line) by two turns by 60 degrees (each followed by a line) we obtain what may be called a *hexdragon*, shown in figure 1.31-L (created with [FXT: bits/dragon-hex-texpic-demo.cc]). A string substitution process for the hexdragon is shown in figure 1.31-N.

1.31.5 Dragon curves based on radix-R counting

Another dragon curve can be generated on radix-5 counting (we will call the curve *R5-dragon*) [FXT: bits/bit-dragon-r5.h]:

```
static inline bool bit_dragon_r5_turn(ulong &x)
// Increment the radix-5 word x and
// return (tr) whether the lowest nonzero digit
// of the incremented word is > 2.
{
    ulong s = 0;
    while ( (x & 7) == 4 )  { x >>= 3;  ++s; }  // scan over nines
    bool tr = ( (x & 7) >= 2 );  // whether digit will be > 2
    ++x;  // increment next digit
    x <<= (3*s);  // shift back
    return  tr;
}
```

About 310 million values per second are generated. The turns are by 90 degrees. Two renderings of the R5-dragon are shown in figure 1.31-O (created with [FXT: bits/dragon-r5-texpic-demo.cc]). The sequence of returned values (entry A175337 in [312]) can be computed via the string substitution shown in figure 1.31-R (top).

Based on radix-7 counting we can generate a curve that will be called the *R7-dragon*, the turns are be 120 degrees [FXT: bits/bit-dragon-r7.h]:

```
static inline bool bit_dragon_r7_turn(ulong &x)
// Increment the radix-7 word x and
```

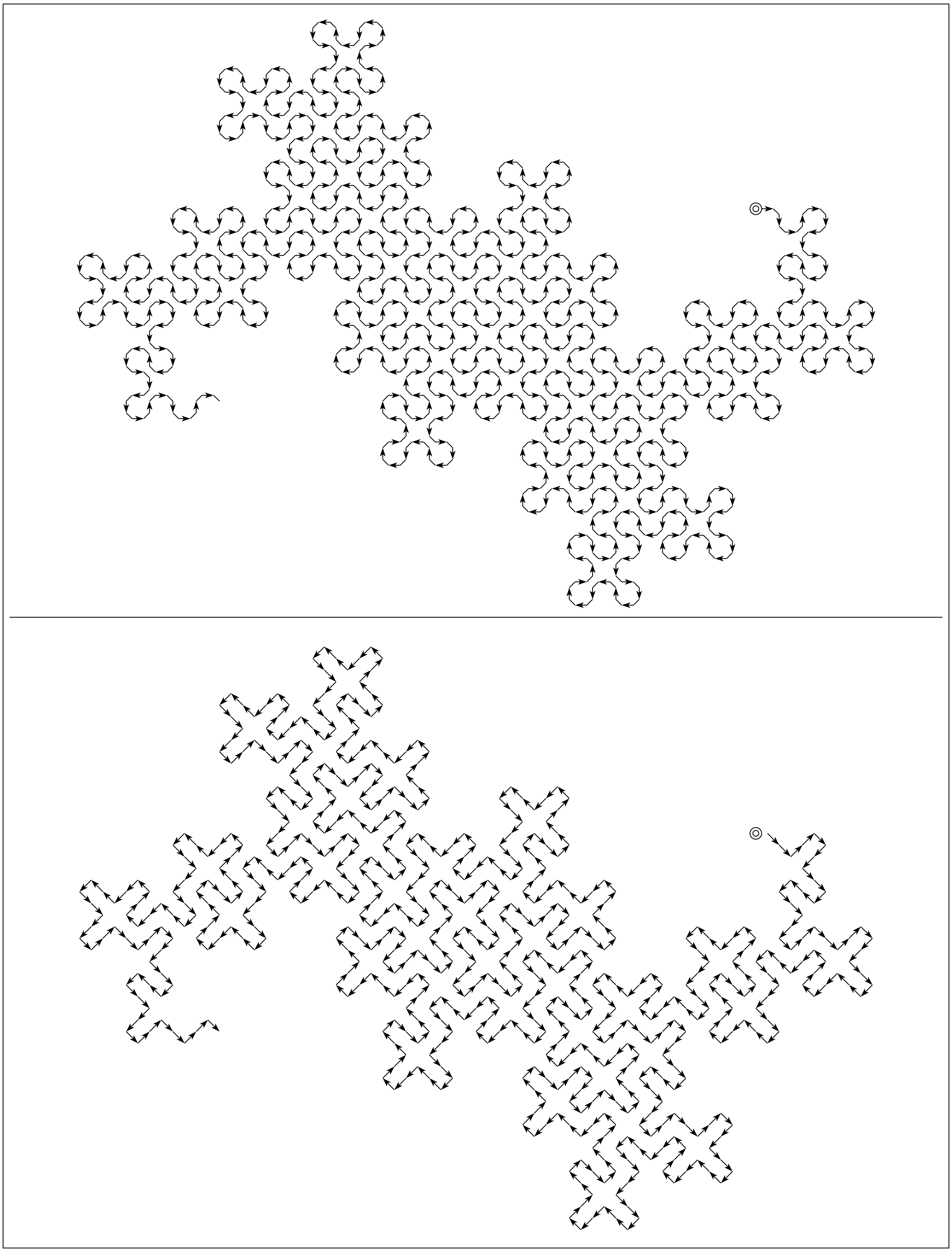

Figure 1.31-O: The first 625 segments of the R5-dragon (two different renderings).

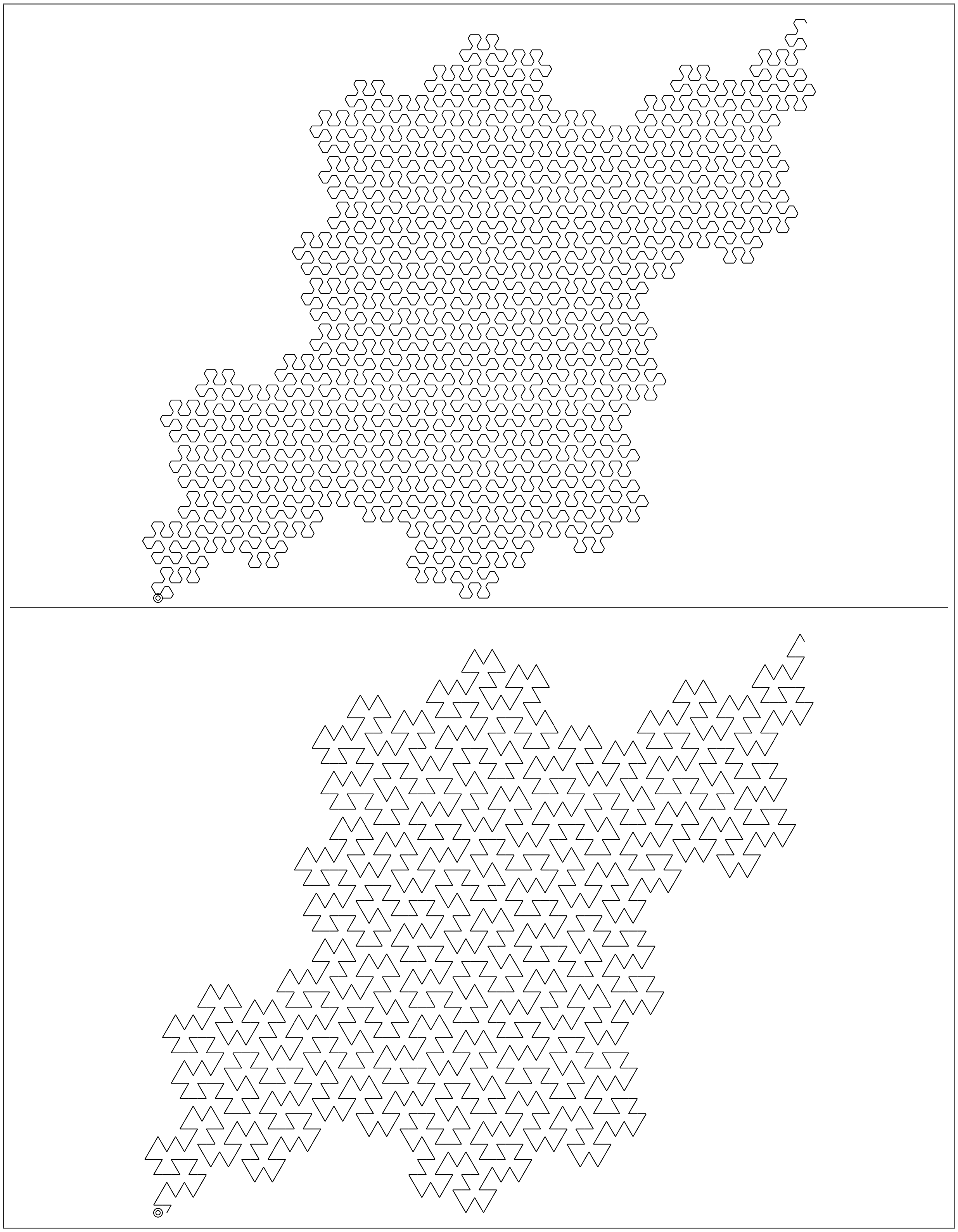

Figure 1.31-P: The first 2401 segments of the R7-dragon (two different renderings).

```
// return (tr) whether the lowest nonzero digit
// of the incremented word is either 2, 3, or 6.
```

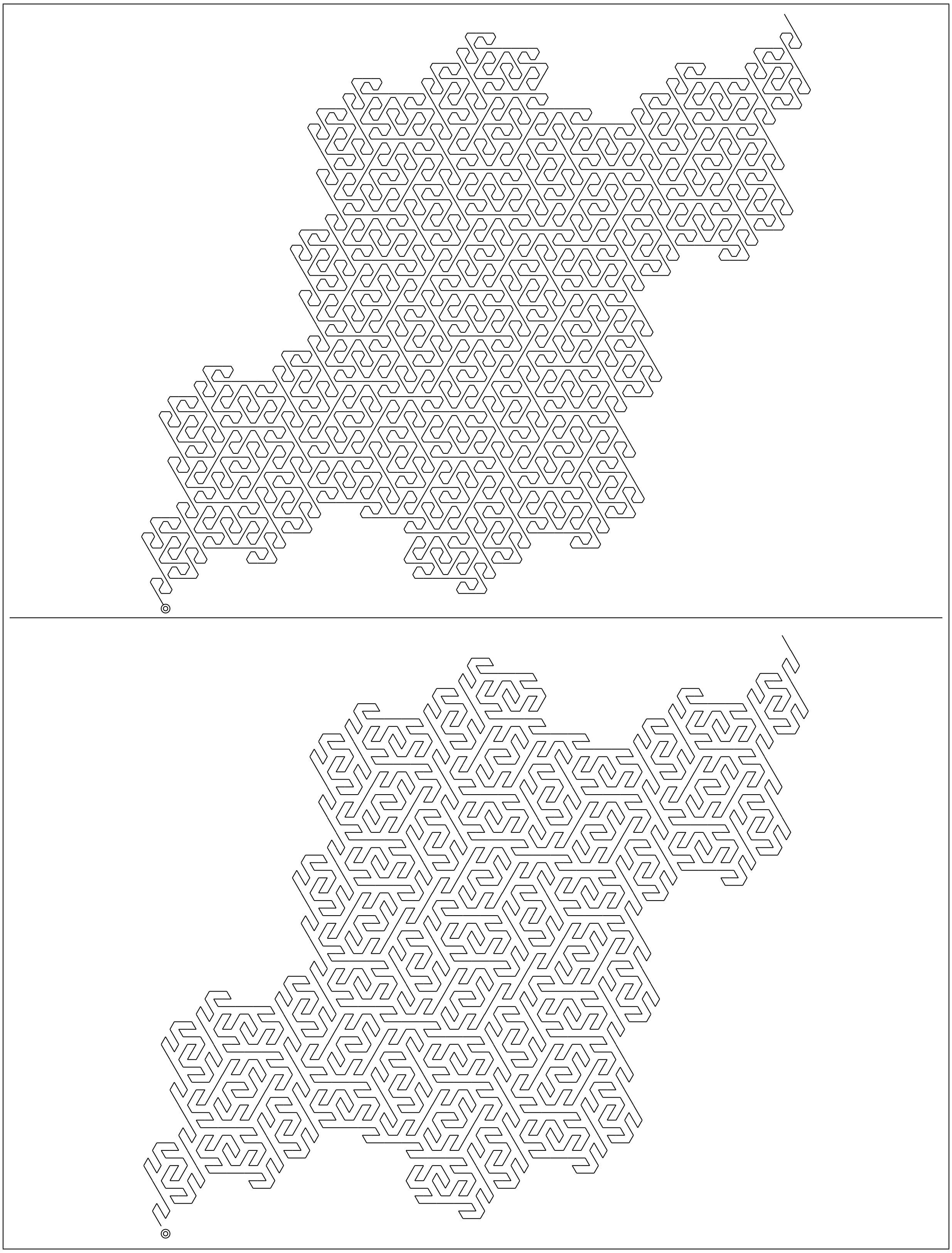

Figure 1.31-Q: The first 2401 segments of the second R7-dragon (two different renderings).

```
Start: 0
Rules:
  0 --> 00110
  1 --> 00111
-------------
0:   (#=1)
  0
1:   (#=5)
  00110
2:   (#=25)
  0011000110001110011100110
3:   (#=125)
  00110001100011100111001100011000110001110011100110001100011000111001110011100 \
  110001100011100111001110011000110001110011100110
```

```
Start: 0
Rules:
  0 --> 0100110
  1 --> 0110110
-------------
0:   (#=1)
  0
1:   (#=7)
  0100110
2:   (#=49)
  0100110011011001001100100110011011001101100100110
3:   (#=343)
  0100110011011001001100100110011011001101100100110010011001101100110110010 01 ...
```

```
Start: 0
Rules:
  0 --> 0++--00
  + --> 0++--0+
  - --> 0++--0-
-------------
0:   (#=1)
  0
1:   (#=7)
  0++--00
2:   (#=49)
  0++--000++--0+0++--0+0++--0-0++--0-0++--000++--00
3:   (#=343)
  0++--000++--0+0++--0+0++--0-0++--0-0++--000++--000++--000++--0+0++--0+0++-- ...
```

Figure 1.31-R: Turns of the R5-dragon (top), the R7-dragon (middle), and the second R7-dragon (bottom), generated by string substitution.

```
{
    ulong s = 0;
    while ( (x & 7) == 6 )  { x >>= 3;  ++s; }  // scan over nines
    ++x;  // increment next digit
    bool tr = ( x & 2 );  // whether digit is either 2, 3, or 6
    x <<= (3*s);  // shift back
    return  tr;
}

```

Two renderings of the R7-dragon are shown in figure 1.31-P (created with [FXT: bits/dragon-r7-texpic-demo.cc]). The sequence of returned values (entry A176405 in [312]) can be computed via the string substitution shown in figure 1.31-R (middle). Turns for another curve based on radix-7 counting (shown in figure 1.31-Q, created with [FXT: bits/dragon-r7-2-texpic-demo.cc]) can be computed as follows:

```
static inline int bit_dragon_r7_2_turn(ulong &x)
// Increment the radix-7 word x and
// return (tr) according to the lowest nonzero digit d
// of the incremented word:
//  d==[1,2,3,4,5,6]  ==>  rt:=[0,+1,+1,-1,-1,0]
// (tr * 120deg) is the turn with the second R7-dragon.
{
    ulong s = 0;
    while ( (x & 7) == 6 )  { x >>= 3;  ++s; }  // scan over nines
    ++x;  // increment next digit
    int tr = 2 - ( (0x2f58 >> (2*(x&7)) ) & 3 );
    x <<= (3*s);  // shift back
    return  tr;
}
```

The sequence of turns can be generated by the string substitution shown in figure 1.31-R (bottom), it is

```
Start: F
Rules: F --> F+F+F-F-F    + --> +    - --> -
-------------
0:   (#=1)
  F
1:   (#=9)
  F+F+F-F-F
2:   (#=49)
  F+F+F-F-F+F+F+F-F-F+F+F+F-F-F-F+F+F-F-F-F+F+F-F-F
3:   (#=249)
  F+F+F-F-F+F+F+F-F-F+F+F+F-F-F-F+F+F-F-F-F+F+F-F-F+F+F+F-F-F+F+F+F-F-F+F+F+F-F-F-F+ ...
```

```
Start: F
Rules: F --> F+F-F-F+F+F-F    + --> +    - --> -
-------------
0:   (#=1)
  F
1:   (#=13)
  F+F-F-F+F+F-F
2:   (#=97)
  F+F-F-F+F+F-F+F+F-F-F+F+F-F-F+F-F-F+F+F-F-F+F-F-F+F+F-F+F+F-F-F+F+F-F+F+F-F-F+F+F- ...
```

```
Start: F
Rules: F --> F0F+F+F-F-F0F    + --> +    - --> -    0 --> 0
-------------
0:   (#=1)
  F
1:   (#=13)
  F0F+F+F-F-F0F
2:   (#=97)
  F0F+F+F-F-F0F0F0F+F+F-F-F0F+F0F+F+F-F-F0F+F0F+F+F-F-F0F-F0F+F+F-F-F0F-F0F+F+F-F-F0 ...
```

Figure 1.31-S: String substitution processes for the turns (symbols '+' and '-') and moves (symbol 'F' is a unit move in the current direction) of the R5-dragon (top), the R7-dragon (middle), and the second R7-dragon (bottom).

entry A176416 in [312].

Two curves respectively based on radix-9 and radix-13 counting are shown in figure 1.31-T. The corresponding routines are given in [FXT: bits/bit-dragon-r9.h]

```
static inline bool bit_dragon_r9_turn(ulong &x)
// Increment the radix-9 word x and
// return (tr) whether the lowest nonzero digit
// of the incremented word is either 2, 3, 5, or 8.
// tr determines whether to turn left or right (by 120 degrees)
// with the R9-dragon fractal.
// The sequence tr is the fixed point
// of the morphism  0 |--> 011010010,  1 |--> 011010011.
// Also fixed point of morphism (identify + with 0 and - with 1)
//  F |--> F+F-F-F+F-F+F+F-F,  + |--> +,  - |--> -
// Also fixed point of morphism
//  F |--> G+G-G,  G |--> F-F+F,  + |--> +,  - |--> -
{
    ulong s = 0;
    while ( (x & 15) == 8 )  { x >>= 4;  ++s; }  // scan over nines
    ++x;  // increment next digit
    bool tr = ( (0x12c >> (x&15)) & 1 );  // whether digit is either 2, 3, 5, or 8
    x <<= (4*s);  // shift back
    return  tr;
}
```

and [FXT: bits/bit-dragon-r13.h]

```
static inline bool bit_dragon_r13_turn(ulong &x)
// Increment the radix-13 word x and
// return (tr) whether the lowest nonzero digit
// of the incremented word is either 3, 6, 8, 9, 11, or 12.
// tr determines whether to turn left or right (by 90 degrees)
// with the R13-dragon fractal.
// The sequence tr is the fixed point
// of the morphism  0 |--> 0010010110110,  1 |--> 0010010110111.
// Also fixed point of morphism (identify + with 0 and - with 1)
//  F |--> F+F+F-F+F+F-F+F-F-F+F-F-F,  + |--> +,  - |--> -
{
    ulong s = 0;
```

```
        while ( (x & 15) == 12 )  { x >>= 4;  ++s; }  // scan over nines
        ++x;  // increment next digit
        bool tr = ( (0x1b48 >> (x&15)) & 1 );  // whether digit is either 3, 6, 8, 9, 11, or 12
        x <<= (4*s);  // shift back
        return  tr;
    }
```

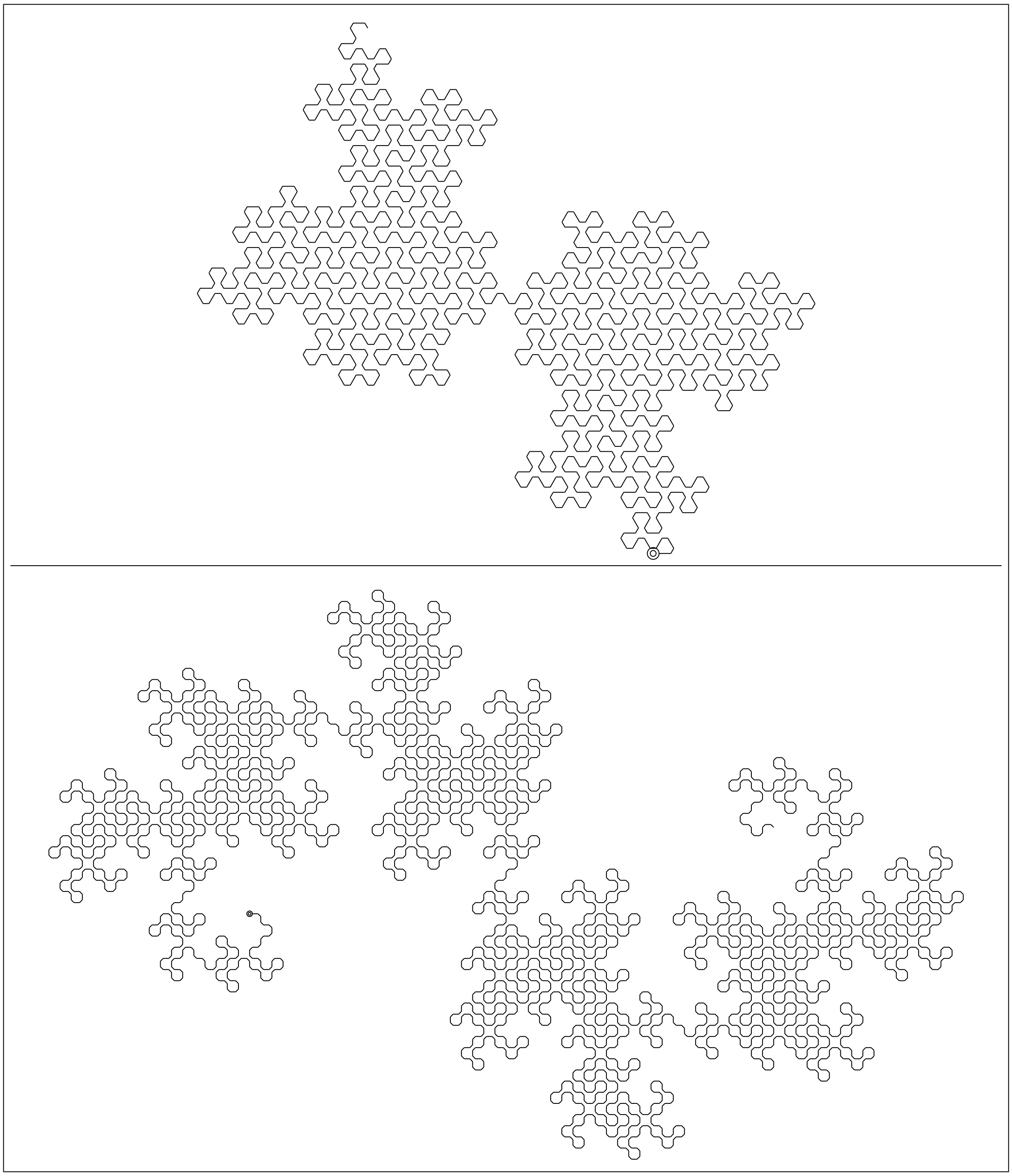

Figure 1.31-T: The R9-dragon (top) and the R13-dragon (bottom).