filename: c-bitwise-operations-hacks-multif-20250813.txt
https://graphics.stanford.edu/⍰seander/bithacks.html

Bit Twiddling Hacks

About the operation counting methodology

When totaling the number of operations for algorithms here, any C operator is counted as one operation. Intermediate assignments, which need not be written to RAM, are not counted. Of course, this operation counting approach only serves as an approximation of the actual number of machine instructions and CPU time. All operations are assumed to take the same amount of time, which is not true in reality, but CPUs have been heading increasingly in this direction over time. There are many nuances that determine how fast a system will run a given sample of code, such as cache sizes, memory bandwidths, instruction sets, etc. In the end, benchmarking is the best way to determine whether one method is really faster than another, so consider the techniques below as possibilities to test on your target architecture.

Compute the sign of an integer

```
int v;      // we want to find the sign of v
int sign;   // the result goes here

// CHAR_BIT is the number of bits per byte (normally 8).
sign = -(v < 0);  // if v < 0 then -1, else 0.

// or, to avoid branching on CPUs with flag registers (IA32):
sign = -(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT - 1));

// or, for one less instruction (but not portable):
sign = v >> (sizeof(int) * CHAR_BIT - 1);
```

The last expression above evaluates to sign = v >> 31 for 32-bit integers. This is one operation faster than the obvious way, sign = -(v < 0). This trick works because when signed integers are shifted right, the value of the far left bit is copied to the other bits. The far left bit is 1 when the value is negative and 0 otherwise; all 1 bits gives -1. Unfortunately, this behavior is architecture-specific.

Alternatively, if you prefer the result be either -1 or +1, then use:

```
sign = +1 | (v >> (sizeof(int) * CHAR_BIT - 1));  // if v < 0 then -1, else +1
```

On the other hand, if you prefer the result be either -1, 0, or +1, then use:

```
sign = (v != 0) | -(int)((unsigned int)((int)v) >> (sizeof(int) * CHAR_BIT - 1));

// Or, for more speed but less portability:
sign = (v != 0) | (v >> (sizeof(int) * CHAR_BIT - 1));  // -1, 0, or +1

// Or, for portability, brevity, and (perhaps) speed:
sign = (v > 0) - (v < 0); // -1, 0, or +1
```

If instead you want to know if something is non-negative, resulting in +1 or else 0, then use:

```
sign = 1 ^ ((unsigned int)v >> (sizeof(int) * CHAR_BIT - 1)); // if v < 0 then 0, else 1
```

Caveat: On March 7, 2003, Angus Duggan pointed out that the 1989 ANSI C specification leaves the result of signed right-shift implementation-defined, so on some systems this hack might not work. For greater portability, Toby Speight suggested on September 28, 2005 that CHAR_BIT be used here and throughout rather than assuming bytes were 8 bits long. Angus recommended the more portable versions above, involving casting on March 4, 2006. Rohit Garg suggested the version for non-negative integers on September 12, 2009.

Detect if two integers have opposite signs

```
int x, y;               // input values to compare signs

bool f = ((x ^ y) < 0); // true iff x and y have opposite signs
```

Manfred Weis suggested I add this entry on November 26, 2009.

Compute the integer absolute value (abs) without branching

```
int v;           // we want to find the absolute value of v
unsigned int r;  // the result goes here
int const mask = v >> sizeof(int) * CHAR_BIT - 1;

r = (v + mask) ^ mask;
```

Patented variation:

```
r = (v ^ mask) - mask;
```

Some CPUs don't have an integer absolute value instruction (or the compiler fails to use them). On machines where branching is expensive, the above expression can be faster than the obvious approach, r = (v < 0) ? -(unsigned)v : v, even though the number of operations is the same.

On March 7, 2003, Angus Duggan pointed out that the 1989 ANSI C specification leaves the result of signed right-shift implementation-defined, so on some systems this hack might not work. I've read that ANSI C does not require values to be represented as two's complement, so it may not work for that reason as well (on a diminishingly small number of old machines that still use one's complement). On March 14, 2004, Keith H. Duggar sent me the patented variation above; it is superior to the one I initially came up with, r=(+1|(v>>(sizeof(int)*CHAR_BIT-1)))*v, because a multiply is not used. Unfortunately, this method has been patented in the USA on June 6, 2000 by Vladimir Yu Volkonsky and assigned to Sun Microsystems. On August 13, 2006, Yuriy Kaminskiy told me that the patent is likely invalid because the method was published well before the patent was even filed, such as in How to Optimize for the Pentium Processor by Agner Fog, dated November, 9, 1996. Yuriy also mentioned that this document was translated to Russian in 1997, which Vladimir could have read. Moreover, the Internet Archive also has an old link to it. On January 30, 2007, Peter Kankowski shared with me an abs version he discovered that was inspired by Microsoft's Visual C++ compiler output. It is featured here as the primary solution. On December 6, 2007, Hai Jin complained that the result was signed, so when computing the abs of the most negative value, it was still negative. On April 15, 2008 Andrew Shapira pointed out that the obvious approach could overflow, as it lacked an (unsigned) cast then; for maximum portability he suggested (v < 0) ? (1 + ((unsigned)(-1-v))) : (unsigned)v. But citing the ISO C99 spec on July 9, 2008, Vincent LefÃ�vre convinced me to remove it becasue even on non-2s-complement machines -(unsigned)v will do the right thing. The evaluation of -(unsigned)v first converts the negative value of v to an unsigned by adding 2**N, yielding a 2s complement representation of v's value that I'll call U. Then, U is negated, giving the desired result, -U = 0 - U = 2**N - U = 2**N - (v+2**N) = -v = abs(v).

Compute the minimum (min) or maximum (max) of two integers without branching

```
int x;  // we want to find the minimum of x and y
int y;
int r;  // the result goes here

r = y ^ ((x ^ y) & -(x < y)); // min(x, y)
```

On some rare machines where branching is very expensive and no condition move instructions exist, the above expression might be faster than the obvious approach, r = (x < y) ? x : y, even though it involves two more instructions. (Typically, the obvious approach is best, though.) It works because if x < y, then -(x < y) will be all ones, so r = y ^ (x ^ y) & �0 = y ^ x ^ y = x. Otherwise, if x >= y, then -(x < y) will be all zeros, so r = y ^ ((x ^ y) & 0) = y. On some machines, evaluating (x < y) as 0 or 1 requires a branch instruction, so there may be no advantage.

To find the maximum, use:

```
r = x ^ ((x ^ y) & -(x < y)); // max(x, y)
```

Quick and dirty versions:

If you know that INT_MIN <= x - y <= INT_MAX, then you can use the following, which are faster because (x - y) only needs to be evaluated once.

```
r = y + ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1))); // min(x, y)
r = x - ((x - y) & ((x - y) >> (sizeof(int) * CHAR_BIT - 1))); // max(x, y)
```

Note that the 1989 ANSI C specification doesn't specify the result of signed right-shift, so these aren't portable. If exceptions are thrown on overflows, then the values of x and y should be unsigned or cast to unsigned for the subtractions to avoid unnecessarily throwing an exception, however the right-shift needs a signed operand to produce all one bits when negative, so cast to signed there.

On March 7, 2003, Angus Duggan pointed out the right-shift portability issue. On May 3, 2005, Randal E. Bryant alerted me to the need for the precondition, INT_MIN <= x - y <= INT_MAX, and suggested the non-quick and dirty version as a fix. Both of these issues concern only the quick and dirty version. Nigel Horspoon observed on July 6, 2005 that gcc produced the same code on a Pentium as the obvious solution because of how it evaluates (x < y). On July 9, 2008 Vincent LefÃ�vre pointed out the potential for overflow exceptions with subtractions in r = y + ((x - y) & -(x < y)), which was the previous version. Timothy B. Terriberry suggested using xor rather than add and subract to avoid casting and the risk of overflows on June 2, 2009.

Determining if an integer is a power of 2

```
unsigned int v; // we want to see if v is a power of 2
bool f;         // the result goes here

f = (v & (v - 1)) == 0;
```

Note that 0 is incorrectly considered a power of 2 here. To remedy this, use:

```
f = v && !(v & (v - 1));
```

Sign extending from a constant bit-width

Sign extension is automatic for built-in types, such as chars and ints. But suppose you have a signed two's complement number, x, that is stored using only b bits. Moreover, suppose you want to convert x to an int, which has more than b bits. A simple copy will work if x is positive, but if negative, the sign must be extended. For example, if we have only 4 bits to store a number, then -3 is represented as 1101 in binary. If we have 8 bits, then -3 is 11111101. The most-significant bit of the 4-bit representation is replicated sinistrally to fill in the destination when we convert to a representation with more bits; this is sign extending. In C, sign extension from a constant bit-width is trivial, since bit fields may be specified in structs or unions. For example, to convert from 5 bits to an full integer:

```
int x; // convert this from using 5 bits to a full int
int r; // resulting sign extended number goes here
struct {signed int x:5;} s;
r = s.x = x;
```

The following is a C++ template function that uses the same language feature to convert from B bits in one operation (though the compiler is generating more, of course).

```
template <typename T, unsigned B>
```

```
inline T signextend(const T x) {
    struct {T x:B;} s;
    return s.x = x;
}

int r = signextend<signed int,5>(x);  // sign extend 5 bit number x to r
```

John Byrd caught a typo in the code (attributed to html formatting) on May 2, 2005. On March 4, 2006, Pat Wood pointed out that the ANSI C standard requires that the bitfield have the keyword "signed" to be signed; otherwise, the sign is undefined.

Sign extending from a variable bit-width

Sometimes we need to extend the sign of a number but we don't know a priori the number of bits, b, in which it is represented. (Or we could be programming in a language like Java, which lacks bitfields.)

```
unsigned b; // number of bits representing the number in x
int x;      // sign extend this b-bit number to r
int r;      // resulting sign-extended number
int const m = 1U << (b - 1); // mask can be pre-computed if b is fixed

x = x & ((1U << b) - 1);  // (Skip this if bits in x above position b are already zero.)
r = (x ^ m) - m;
```

The code above requires four operations, but when the bitwidth is a constant rather than variable, it requires only two fast operations, assuming the upper bits are already zeroes.

A slightly faster but less portable method that doesn't depend on the bits in x above position b being zero is:

```
int const m = CHAR_BIT * sizeof(x) - b;
r = (x << m) >> m;
```

Sean A. Irvine suggested that I add sign extension methods to this page on June 13, 2004, and he provided m = (1 << (b - 1)) - 1; r = -(x & �m) | x; as a starting point from which I optimized to get m = 1U << (b - 1); r = -(x & m) | x. But then on May 11, 2007, Shay Green suggested the version above, which requires one less operation than mine. Vipin Sharma suggested I add a step to deal with situations where x had possible ones in bits other than the b bits we wanted to sign-extend on Oct. 15, 2008. On December 31, 2009 Chris Pirazzi suggested I add the faster version, which requires two operations for constant bit-widths and three for variable widths.

Sign extending from a variable bit-width in 3 operations

The following may be slow on some machines, due to the effort required for multiplication and division. This version is 4 operations. If you know that your initial bit-width, b, is greater than 1, you might do this type of sign extension in 3 operations by using r = (x * multipliers[b]) / multipliers[b], which requires only one array lookup.

```
unsigned b; // number of bits representing the number in x
int x;      // sign extend this b-bit number to r
int r;      // resulting sign-extended number
#define M(B) (1U << ((sizeof(x) * CHAR_BIT) - B)) // CHAR_BIT=bits/byte

static int const multipliers[] =
{
    0,     M(1),  M(2),  M(3),  M(4),  M(5),  M(6),  M(7),
    M(8),  M(9),  M(10), M(11), M(12), M(13), M(14), M(15),
    M(16), M(17), M(18), M(19), M(20), M(21), M(22), M(23),
    M(24), M(25), M(26), M(27), M(28), M(29), M(30), M(31),
    M(32)
}; // (add more if using more than 64 bits)

static int const divisors[] =
{
    1,    �M(1),  M(2),  M(3),  M(4),  M(5),  M(6),  M(7),
    M(8),  M(9),  M(10), M(11), M(12), M(13), M(14), M(15),
    M(16), M(17), M(18), M(19), M(20), M(21), M(22), M(23),
    M(24), M(25), M(26), M(27), M(28), M(29), M(30), M(31),
    M(32)
}; // (add more for 64 bits)
#undef M
r = (x * multipliers[b]) / divisors[b];
```

The following variation is not portable, but on architectures that employ an arithmetic right-shift, maintaining the sign, it should be fast.

```
const int s = -b; // OR:  sizeof(x) * CHAR_BIT - b;
r = (x << s) >> s;
```

Randal E. Bryant pointed out a bug on May 3, 2005 in an earlier version (that used multipliers[] for divisors[]), where it failed on the case of x=1 and b=1.

Conditionally set or clear bits without branching

```
bool f;         // conditional flag
unsigned int m; // the bit mask
unsigned int w; // the word to modify:  if (f) w |= m; else w &= �m;

w ^= (-f ^ w) & m;

// OR, for superscalar CPUs:
```

```
w = (w & ⍰m) | (-f & m);
```

On some architectures, the lack of branching can more than make up for what appears to be twice as many operations. For instance, informal speed tests on an AMD Athlon XP 2100+ indicated it was 5-10% faster. An Intel Core 2 Duo ran the superscalar version about 16% faster than the first. Glenn Slayden informed me of the first expression on December 11, 2003. Marco Yu shared the superscalar version with me on April 3, 2007 and alerted me to a typo 2 days later.

Conditionally negate a value without branching

If you need to negate only when a flag is false, then use the following to avoid branching:

```
bool fDontNegate;  // Flag indicating we should not negate v.
int v;             // Input value to negate if fDontNegate is false.
int r;             // result = fDontNegate ? v : -v;

r = (fDontNegate ^ (fDontNegate - 1)) * v;
```

If you need to negate only when a flag is true, then use this:

```
bool fNegate;  // Flag indicating if we should negate v.
int v;         // Input value to negate if fNegate is true.
int r;         // result = fNegate ? -v : v;

r = (v ^ -fNegate) + fNegate;
```

Avraham Plotnitzky suggested I add the first version on June 2, 2009. Motivated to avoid the multiply, I came up with the second version on June 8, 2009. Alfonso De Gregorio pointed out that some parens were missing on November 26, 2009, and received a bug bounty.

Merge bits from two values according to a mask

```
unsigned int a;    // value to merge in non-masked bits
unsigned int b;    // value to merge in masked bits
unsigned int mask; // 1 where bits from b should be selected; 0 where from a.
unsigned int r;    // result of (a & ⍰mask) | (b & mask) goes here

r = a ^ ((a ^ b) & mask);
```

This shaves one operation from the obvious way of combining two sets of bits according to a bit mask. If the mask is a constant, then there may be no advantage.

Ron Jeffery sent this to me on February 9, 2006.

Counting bits set (naive way)

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v

for (c = 0; v; v >>= 1) {
    c += v & 1;
}
```

The naive approach requires one iteration per bit, until no more bits are set. So on a 32-bit word with only the high set, it will go through 32 iterations.

Counting bits set by lookup table

```
static const unsigned char BitsSetTable256[256] =
{
#   define B2(n) n,     n+1,     n+1,     n+2
#   define B4(n) B2(n), B2(n+1), B2(n+1), B2(n+2)
#   define B6(n) B4(n), B4(n+1), B4(n+1), B4(n+2)
    B6(0), B6(1), B6(1), B6(2)
};

unsigned int v; // count the number of bits set in 32-bit value v
unsigned int c; // c is the total bits set in v

// Option 1:
c = BitsSetTable256[v & 0xff] +
    BitsSetTable256[(v >> 8) & 0xff] +
    BitsSetTable256[(v >> 16) & 0xff] +
    BitsSetTable256[v >> 24];

// Option 2:
unsigned char * p = (unsigned char *) &v;
c = BitsSetTable256[p[0]] +
    BitsSetTable256[p[1]] +
    BitsSetTable256[p[2]] +
    BitsSetTable256[p[3]];


// To initially generate the table algorithmically:
BitsSetTable256[0] = 0;

for (int i = 0; i < 256; i++) {
    BitsSetTable256[i] = (i & 1) + BitsSetTable256[i / 2];
}
```

On July 14, 2009 Hallvard Furuseth suggested the macro compacted table.

Counting bits set, Brian Kernighan's way

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v
for (c = 0; v; c++) {
    v &= v - 1; // clear the least significant bit set
}
```

Brian Kernighan's method goes through as many iterations as there are set bits. So if we have a 32-bit word with only the high bit set, then it will only go once through the loop.

Published in 1988, the C Programming Language 2nd Ed. (by Brian W. Kernighan and Dennis M. Ritchie) mentions this in exercise 2-9. On April 19, 2006 Don Knuth pointed out to me that this method "was first published by Peter Wegner in CACM 3 (1960), 322. (Also discovered independently by Derrick Lehmer and published in 1964 in a book edited by Beckenbach.)"

Counting bits set in 14, 24, or 32-bit words using 64-bit instructions

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v

// option 1, for at most 14-bit values in v:
c = (v * 0x200040008001ULL & 0x111111111111111ULL) % 0xf;

// option 2, for at most 24-bit values in v:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;

// option 3, for at most 32-bit values in v:
c =  ((v & 0xfff) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += (((v & 0xfff000) >> 12) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
c += ((v >> 24) * 0x1001001001001ULL & 0x84210842108421ULL) % 0x1f;
```

This method requires a 64-bit CPU with fast modulus division to be efficient. The first option takes only 3 operations; the second option takes 10; and the third option takes 15.

Rich Schroeppel originally created a 9-bit version, similiar to option 1; see the Programming Hacks section of Beeler, M., Gosper, R. W., and Schroeppel, R. HAKMEM. MIT AI Memo 239, Feb. 29, 1972. His method was the inspiration for the variants above, devised by Sean Anderson. Randal E. Bryant offered a couple bug fixes on May 3, 2005. Bruce Dawson tweaked what had been a 12-bit version and made it suitable for 14 bits using the same number of operations on Feburary 1, 2007.

Counting bits set, in parallel

```
unsigned int v; // count bits set in this (32-bit value)
unsigned int c; // store the total here
static const int S[] = {1, 2, 4, 8, 16}; // Magic Binary Numbers
static const int B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF, 0x0000FFFF};

c = v - ((v >> 1) & B[0]);
c = ((c >> S[1]) & B[1]) + (c & B[1]);
c = ((c >> S[2]) + c) & B[2];
c = ((c >> S[3]) + c) & B[3];
c = ((c >> S[4]) + c) & B[4];
```

The B array, expressed as binary, is:

```
B[0] = 0x55555555 = 01010101 01010101 01010101 01010101
B[1] = 0x33333333 = 00110011 00110011 00110011 00110011
B[2] = 0x0F0F0F0F = 00001111 00001111 00001111 00001111
B[3] = 0x00FF00FF = 00000000 11111111 00000000 11111111
B[4] = 0x0000FFFF = 00000000 00000000 11111111 11111111
```

We can adjust the method for larger integer sizes by continuing with the patterns for the Binary Magic Numbers, B and S. If there are k bits, then we need the arrays S and B to be ceil(lg(k)) elements long, and we must compute the same number of expressions for c as S or B are long. For a 32-bit v, 16 operations are used.

The best method for counting bits in a 32-bit integer v is the following:

```
v = v - ((v >> 1) & 0x55555555);                    // reuse input as temporary
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);     // temp
c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24; // count
```

The best bit counting method takes only 12 operations, which is the same as the lookup-table method, but avoids the memory and potential cache misses of a table. It is a hybrid between the purely parallel method above and the earlier methods using multiplies (in the section on counting bits with 64-bit instructions), though it doesn't use 64-bit instructions. The counts of bits set in the bytes is done in parallel, and the sum total of the bits set in the bytes is computed by multiplying by 0x1010101 and shifting right 24 bits.

A generalization of the best bit counting method to integers of bit-widths upto 128 (parameterized by type T) is this:

```
v = v - ((v >> 1) & (T)�(T)0/3);                           // temp
v = (v & (T)�(T)0/15*3) + ((v >> 2) & (T)�(T)0/15*3);      // temp
v = (v + (v >> 4)) & (T)�(T)0/255*15;                      // temp
c = (T)(v * ((T)�(T)0/255)) >> (sizeof(T) - 1) * CHAR_BIT; // count
```

See Ian Ashdown's nice newsgroup post for more information on counting the number of bits set (also known as sideways addition). The best bit counting method was brought to my attention on October 5, 2005 by Andrew Shapira; he found it in pages 187-188 of Software Optimization Guide for AMD Athlon 64 and Opteron Processors. Charlie Gordon suggested a way to shave off one operation from the purely parallel version on December 14, 2005, and Don Clugston trimmed three more from it on December 30, 2005. I made a typo with Don's suggestion that Eric Cole spotted on January 8, 2006. Eric later suggested the arbitrary bit-width generalization to the best method on November 17, 2006. On April 5, 2007, Al Williams observed that I had a line of dead code at the top of the first method.

Count bits set (rank) from the most-significant bit upto a given position

The following finds the the rank of a bit, meaning it returns the sum of bits that are set to 1 from the most-signficant bit downto the bit at the given position.

```
  uint64_t v;       // Compute the rank (bits set) in v from the MSB to pos.
  unsigned int pos; // Bit position to count bits upto.
  uint64_t r;       // Resulting rank of bit at pos goes here.

  // Shift out bits after given position.
  r = v >> (sizeof(v) * CHAR_BIT - pos);

  // Count set bits in parallel.
  // r = (r & 0x5555...) + ((r >> 1) & 0x5555...);
  r = r - ((r >> 1) & �0UL/3);

  // r = (r & 0x3333...) + ((r >> 2) & 0x3333...);
  r = (r & �0UL/5) + ((r >> 2) & �0UL/5);

  // r = (r & 0x0f0f...) + ((r >> 4) & 0x0f0f...);
  r = (r + (r >> 4)) & �0UL/17;

  // r = r % 255;
  r = (r * (�0UL/255)) >> ((sizeof(v) - 1) * CHAR_BIT);
```

Juha JÃ¤rvi sent this to me on November 21, 2009 as an inverse operation to the computing the bit position with the given rank, which follows.

Select the bit position (from the most-significant bit) with the given count (rank)

The following 64-bit code selects the position of the r^th 1 bit when counting from the left. In other words if we start at the most significant bit and proceed to the right, counting the number of bits set to 1 until we reach the desired rank, r, then the position where we stop is returned. If the rank requested exceeds the count of bits set, then 64 is returned. The code may be modified for 32-bit or counting from the right.

```
  uint64_t v;          // Input value to find position with rank r.
  unsigned int r;      // Input: bit's desired rank [1-64].
  unsigned int s;      // Output: Resulting position of bit with rank r [1-64]
  uint64_t a, b, c, d; // Intermediate temporaries for bit count.
  unsigned int t;      // Bit count temporary.

  // Do a normal parallel bit count for a 64-bit integer,
  // but store all intermediate steps.
  // a = (v & 0x5555...) + ((v >> 1) & 0x5555...);
  a =  v - ((v >> 1) & �0UL/3);

  // b = (a & 0x3333...) + ((a >> 2) & 0x3333...);
  b = (a & �0UL/5) + ((a >> 2) & �0UL/5);

  // c = (b & 0x0f0f...) + ((b >> 4) & 0x0f0f...);
  c = (b + (b >> 4)) & �0UL/0x11;

  // d = (c & 0x00ff...) + ((c >> 8) & 0x00ff...);
  d = (c + (c >> 8)) & �0UL/0x101;
  t = (d >> 32) + (d >> 48);

  // Now do branchless select!
  s  = 64;

  // if (r > t) {s -= 32; r -= t;}
  s -= ((t - r) & 256) >> 3; r -= (t & ((t - r) >> 8));
  t  = (d >> (s - 16)) & 0xff;

  // if (r > t) {s -= 16; r -= t;}
  s -= ((t - r) & 256) >> 4; r -= (t & ((t - r) >> 8));
  t  = (c >> (s - 8)) & 0xf;

  // if (r > t) {s -= 8; r -= t;}
  s -= ((t - r) & 256) >> 5; r -= (t & ((t - r) >> 8));
  t  = (b >> (s - 4)) & 0x7;
  // if (r > t) {s -= 4; r -= t;}
  s -= ((t - r) & 256) >> 6; r -= (t & ((t - r) >> 8));
  t  = (a >> (s - 2)) & 0x3;

  // if (r > t) {s -= 2; r -= t;}
  s -= ((t - r) & 256) >> 7; r -= (t & ((t - r) >> 8));
  t  = (v >> (s - 1)) & 0x1;

  // if (r > t) s--;
  s -= ((t - r) & 256) >> 8;
  s = 65 - s;
```

If branching is fast on your target CPU, consider uncommenting the if-statements and commenting the lines that follow them.

Juha JÃ¤rvi sent this to me on November 21, 2009.

Computing parity the naive way

```
unsigned int v;       // word value to compute the parity of
bool parity = false;  // parity will be the parity of v

while (v) {
    parity = !parity;
    v = v & (v - 1);
}
```

The above code uses an approach like Brian Kernigan's bit counting, above. The time it takes is proportional to the number of bits set.

Compute parity by lookup table

```
static const bool ParityTable256[256] =
{
#   define P2(n) n, n^1, n^1, n
#   define P4(n) P2(n), P2(n^1), P2(n^1), P2(n)
#   define P6(n) P4(n), P4(n^1), P4(n^1), P4(n)
    P6(0), P6(1), P6(1), P6(0)
};

unsigned char b;  // byte value to compute the parity of
bool parity = ParityTable256[b];

// OR, for 32-bit words:
unsigned int v;
v ^= v >> 16;
v ^= v >> 8;
bool parity = ParityTable256[v & 0xff];

// Variation:
unsigned char * p = (unsigned char *) &v;
parity = ParityTable256[p[0] ^ p[1] ^ p[2] ^ p[3]];
```

Randal E. Bryant encouraged the addition of the (admittedly) obvious last variation with variable p on May 3, 2005. Bruce Rawles found a typo in an instance of the table variable's name on September 27, 2005, and he received a $10 bug bounty. On October 9, 2006, Fabrice Bellard suggested the 32-bit variations above, which require only one table lookup; the previous version had four lookups (one per byte) and were slower. On July 14, 2009 Hallvard Furuseth suggested the macro compacted table.

Compute parity of a byte using 64-bit multiply and modulus division

```
unsigned char b;  // byte value to compute the parity of
bool parity =
  (((b * 0x0101010101010101ULL) & 0x8040201008040201ULL) % 0x1FF) & 1;
```

The method above takes around 4 operations, but only works on bytes.

Compute parity of word with a multiply

The following method computes the parity of the 32-bit value in only 8 operations using a multiply.

```
    unsigned int v; // 32-bit word

    v ^= v >> 1;
    v ^= v >> 2;
    v = (v & 0x11111111U) * 0x11111111U;
    return (v >> 28) & 1;
```

Also for 64-bits, 8 operations are still enough.

```
    unsigned long long v; // 64-bit word
    v ^= v >> 1;
    v ^= v >> 2;
    v = (v & 0x1111111111111111UL) * 0x1111111111111111UL;
    return (v >> 60) & 1;
```

Andrew Shapira came up with this and sent it to me on Sept. 2, 2007.

Compute parity in parallel

```
unsigned int v;  // word value to compute the parity of
v ^= v >> 16;
v ^= v >> 8;
v ^= v >> 4;
v &= 0xf;
return (0x6996 >> v) & 1;
```

The method above takes around 9 operations, and works for 32-bit words. It may be optimized to work just on bytes in 5 operations by removing the two lines immediately following "unsigned int v;". The method first shifts and XORs the eight nibbles of the 32-bit value together, leaving the result in the lowest nibble of v. Next, the binary number 0110 1001 1001 0110 (0x6996 in hex) is shifted to the right by the value represented in the lowest nibble of v. This number is like a miniature 16-bit parity-table indexed by the low four bits in v. The result has the parity of v in bit 1, which is masked and returned.

Thanks to Mathew Hendry for pointing out the shift-lookup idea at the end on Dec. 15, 2002. That optimization shaves two operations off using only shifting and XORing to find the parity.

Swapping values with subtraction and addition

```
#define SWAP(a, b) ((&(a) == &(b)) || \
                    (((a) -= (b)), ((b) += (a)), ((a) = (b) - (a))))
```

This swaps the values of a and b without using a temporary variable. The initial check for a and b being the same location in memory may be omitted when you know this can't happen. (The compiler may omit it anyway as an optimization.) If you enable overflows exceptions, then pass unsigned values so an exception isn't thrown. The XOR method that follows may be slightly faster on some machines. Don't use this with floating-point numbers (unless you operate on their raw integer representations).

Sanjeev Sivasankaran suggested I add this on June 12, 2007. Vincent LefÃ vre pointed out the potential for overflow exceptions on July 9, 2008

Swapping values with XOR

```
#define SWAP(a, b) (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))
```

This is an old trick to exchange the values of the variables a and b without using extra space for a temporary variable.

On January 20, 2005, Iain A. Fleming pointed out that the macro above doesn't work when you swap with the same memory location, such as SWAP(a[i], a[j]) with i == j. So if that may occur, consider defining the macro as (((a) == (b)) || (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))). On July 14, 2009, Hallvard Furuseth suggested that on some machines, (((a) ^ (b)) && ((b) ^= (a) ^= (b), (a) ^= (b))) might be faster, since the (a) ^ (b) expression is reused.

Swapping individual bits with XOR

```
unsigned int i, j; // positions of bit sequences to swap
unsigned int n;    // number of consecutive bits in each sequence
unsigned int b;    // bits to swap reside in b
unsigned int r;    // bit-swapped result goes here

unsigned int x = ((b >> i) ^ (b >> j)) & ((1U << n) - 1); // XOR temporary
r = b ^ ((x << i) | (x << j));
```

As an example of swapping ranges of bits suppose we have have b = 00101111 (expressed in binary) and we want to swap the n = 3 consecutive bits starting at i = 1 (the second bit from the right) with the 3 consecutive bits starting at j = 5; the result would be r = 11100011 (binary).

This method of swapping is similar to the general purpose XOR swap trick, but intended for operating on individual bits. The variable x stores the result of XORing the pairs of bit values we want to swap, and then the bits are set to the result of themselves XORed with x. Of course, the result is undefined if the sequences overlap.

On July 14, 2009 Hallvard Furuseth suggested that I change the 1 << n to 1U << n because the value was being assigned to an unsigned and to avoid shifting into a sign bit.

Reverse bits the obvious way

```
unsigned int v;     // input bits to be reversed
unsigned int r = v; // r will be reversed bits of v; first get LSB of v
int s = sizeof(v) * CHAR_BIT - 1; // extra shift needed at end

for (v >>= 1; v; v >>= 1) {
    r <<= 1;
    r |= v & 1;
    s--;
}
r <<= s; // shift when v's highest bits are zero
```

On October 15, 2004, Michael Hoisie pointed out a bug in the original version. Randal E. Bryant suggested removing an extra operation on May 3, 2005. Behdad Esfabod suggested a slight change that eliminated one iteration of the loop on May 18, 2005. Then, on February 6, 2007, Liyong Zhou suggested a better version that loops while v is not 0, so rather than iterating over all bits it stops early.

Reverse bits in word by lookup table

```
static const unsigned char BitReverseTable256[256] =
{
#   define R2(n)     n,     n + 2*64,     n + 1*64,     n + 3*64
#   define R4(n) R2(n), R2(n + 2*16), R2(n + 1*16), R2(n + 3*16)
#   define R6(n) R4(n), R4(n + 2*4 ), R4(n + 1*4 ), R4(n + 3*4 )
    R6(0), R6(2), R6(1), R6(3)
};

unsigned int v; // reverse 32-bit value, 8 bits at time
unsigned int c; // c will get v reversed

// Option 1:
c = (BitReverseTable256[v & 0xff] << 24) |
    (BitReverseTable256[(v >> 8) & 0xff] << 16) |
    (BitReverseTable256[(v >> 16) & 0xff] << 8) |
    (BitReverseTable256[(v >> 24) & 0xff]);

// Option 2:
unsigned char * p = (unsigned char *) &v;
unsigned char * q = (unsigned char *) &c;
q[3] = BitReverseTable256[p[0]];
q[2] = BitReverseTable256[p[1]];
q[1] = BitReverseTable256[p[2]];
q[0] = BitReverseTable256[p[3]];
```

The first method takes about 17 operations, and the second takes about 12, assuming your CPU can load and store bytes easily.

On July 14, 2009 Hallvard Furuseth suggested the macro compacted table.

Reverse the bits in a byte with 3 operations (64-bit multiply and modulus division):

```
unsigned char b; // reverse this (8-bit) byte
b = (b * 0x0202020202ULL & 0x010884422010ULL) % 1023;
```

The multiply operation creates five separate copies of the 8-bit byte pattern to fan-out into a 64-bit value. The AND operation selects the bits that are in the correct (reversed) positions, relative to each 10-bit groups of bits. The multiply and the AND operations copy the bits from the original byte so they each appear in only one of the 10-bit sets. The reversed positions of the bits from the original byte coincide with their relative positions within any 10-bit set. The last step, which involves modulus division by 2^10 - 1, has the effect of merging together each set of 10 bits (from positions 0-9, 10-19, 20-29, ...) in the 64-bit value. They do not overlap, so the addition steps underlying the modulus division behave like or operations.

This method was attributed to Rich Schroeppel in the Programming Hacks section of Beeler, M., Gosper, R. W., and Schroeppel, R. HAKMEM. MIT AI Memo 239, Feb. 29, 1972.

Reverse the bits in a byte with 4 operations (64-bit multiply, no division):

```
unsigned char b; // reverse this byte
b = ((b * 0x80200802ULL) & 0x0884422110ULL) * 0x0101010101ULL >> 32;
```

The following shows the flow of the bit values with the boolean variables a, b, c, d, e, f, g, and h, which comprise an 8-bit byte. Notice how the first multiply fans out the bit pattern to multiple copies, while the last multiply combines them in the fifth byte from the right.

```
                                                                                        abcd efgh (-> hgfe dcba
)
*                                                      1000 0000  0010 0000  0000 1000  0000 0010 (0x80200802)
-------------------------------------------------------------------------------------------------
                                            0abc defg  h00a bcde  fgh0 0abc  defg h00a  bcde fgh0
&                                           0000 1000  1000 0100  0100 0010  0010 0001  0001 0000 (0x0884422110
)
-------------------------------------------------------------------------------------------------
                                            0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
*                                           0000 0001  0000 0001  0000 0001  0000 0001  0000 0001 (0x0101010101
)
-------------------------------------------------------------------------------------------------
                                            0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
                                 0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
                      0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
           0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
0000 d000  h000 0c00  0g00 00b0  00f0 000a  000e 0000
-------------------------------------------------------------------------------------------------
0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba  hgfe 0cba  0gfe 00ba  00fe 000a  000e 0000
>> 32
-------------------------------------------------------------------------------------------------
                                            0000 d000  h000 dc00  hg00 dcb0  hgf0 dcba  hgfe dcba
&                                                                                       1111 1111
-------------------------------------------------------------------------------------------------
                                                                                        hgfe dcba
```

Note that the last two steps can be combined on some processors because the registers can be accessed as bytes; just multiply so that a register stores the upper 32 bits of the result and the take the low byte. Thus, it may take only 6 operations.

Devised by Sean Anderson, July 13, 2001.

Reverse the bits in a byte with 7 operations (no 64-bit):

```
b = ((b * 0x0802LU & 0x22110LU) | (b * 0x8020LU & 0x88440LU)) * 0x10101LU >> 16;
```

Make sure you assign or cast the result to an unsigned char to remove garbage in the higher bits. Devised by Sean Anderson, July 13, 2001. Typo spotted and correction supplied by Mike Keith, January 3, 2002.

Reverse an N-bit quantity in parallel in 5 * lg(N) operations:

```
unsigned int v; // 32-bit word to reverse bit order

// swap odd and even bits
v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);

// swap consecutive pairs
v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);

// swap nibbles ...
v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);

// swap bytes
v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);

// swap 2-byte long pairs
v = ( v >> 16             ) | ( v               << 16);
```

The following variation is also O(lg(N)), however it requires more operations to reverse v. Its virtue is in taking less slightly memory by computing the constants on the fly.

```
unsigned int s = sizeof(v) * CHAR_BIT; // bit size; must be power of 2
unsigned int mask = ⍰0;
while ((s >>= 1) > 0) {
    mask ^= (mask << s);
    v = ((v >> s) & mask) | ((v << s) & ⍰mask);
}
```

These methods above are best suited to situations where N is large. If you use the above with 64-bit ints (or larger), then you need to add more lines (following the pattern); otherwise only the lower 32 bits will be reversed and the result will be in the lower 32 bits.

See Dr. Dobb's Journal 1983, Edwin Freed's article on Binary Magic Numbers for more information. The second variation was suggested by Ken Raeburn on September 13, 2005. Veldmeijer mentioned that the first version could do without ANDS in the last line on March 19, 2006.

Compute modulus division by 1 << s without a division operator

```
const unsigned int n;          // numerator
const unsigned int s;
const unsigned int d = 1U << s; // So d will be one of: 1, 2, 4, 8, 16, 32, ...
unsigned int m;                // m will be n % d
m = n & (d - 1);
```

Most programmers learn this trick early, but it was included for the sake of completeness.

Compute modulus division by (1 << s) - 1 without a division operator

```
unsigned int n;                      // numerator
const unsigned int s;                // s > 0
const unsigned int d = (1 << s) - 1; // so d is either 1, 3, 7, 15, 31, ...).
unsigned int m;                      // n % d goes here.

for (m = n; n > d; n = m) {
    for (m = 0; n; n >>= s) {
        m += n & d;
    }
}
// Now m is a value from 0 to d, but since with modulus division
// we want m to be 0 when it is d.
m = m == d ? 0 : m;
```

This method of modulus division by an integer that is one less than a power of 2 takes at most 5 + (4 + 5 * ceil(N / s)) * ceil(lg(N / s)) operations, where N is the number of bits in the numerator. In other words, it takes at most O(N * lg(N)) time.

Devised by Sean Anderson, August 15, 2001. Before Sean A. Irvine corrected me on June 17, 2004, I mistakenly commented that we could alternatively assign m = ((m + 1) & d) - 1; at the end. Michael Miller spotted a typo in the code April 25, 2005.

Compute modulus division by (1 << s) - 1 in parallel without a division operator

```
// The following is for a word size of 32 bits!
```

```
static const unsigned int M[] =
{
  0x00000000, 0x55555555, 0x33333333, 0xc71c71c7,
  0x0f0f0f0f, 0xc1f07c1f, 0x3f03f03f, 0xf01fc07f,
  0x00ff00ff, 0x07fc01ff, 0x3ff003ff, 0xffc007ff,
  0xff000fff, 0xfc001fff, 0xf0003fff, 0xc0007fff,
  0x0000ffff, 0x0001ffff, 0x0003ffff, 0x0007ffff,
  0x000fffff, 0x001fffff, 0x003fffff, 0x007fffff,
  0x00ffffff, 0x01ffffff, 0x03ffffff, 0x07ffffff,
  0x0fffffff, 0x1fffffff, 0x3fffffff, 0x7fffffff
};

static const unsigned int Q[][6] =
{
  { 0,  0,  0,  0,  0,  0}, {16,  8,  4,  2,  1,  1}, {16,  8,  4,  2,  2,  2},
  {15,  6,  3,  3,  3,  3}, {16,  8,  4,  4,  4,  4}, {15,  5,  5,  5,  5,  5},
  {12,  6,  6,  6 , 6,  6}, {14,  7,  7,  7,  7,  7}, {16,  8,  8,  8,  8,  8},
  { 9,  9,  9,  9,  9,  9}, {10, 10, 10, 10, 10, 10}, {11, 11, 11, 11, 11, 11},
  {12, 12, 12, 12, 12, 12}, {13, 13, 13, 13, 13, 13}, {14, 14, 14, 14, 14, 14},
  {15, 15, 15, 15, 15, 15}, {16, 16, 16, 16, 16, 16}, {17, 17, 17, 17, 17, 17},
  {18, 18, 18, 18, 18, 18}, {19, 19, 19, 19, 19, 19}, {20, 20, 20, 20, 20, 20},
  {21, 21, 21, 21, 21, 21}, {22, 22, 22, 22, 22, 22}, {23, 23, 23, 23, 23, 23},
  {24, 24, 24, 24, 24, 24}, {25, 25, 25, 25, 25, 25}, {26, 26, 26, 26, 26, 26},
  {27, 27, 27, 27, 27, 27}, {28, 28, 28, 28, 28, 28}, {29, 29, 29, 29, 29, 29},
  {30, 30, 30, 30, 30, 30}, {31, 31, 31, 31, 31, 31}
};

static const unsigned int R[][6] =
{
  {0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000},
  {0x0000ffff, 0x000000ff, 0x0000000f, 0x00000003, 0x00000001, 0x00000001},
  {0x0000ffff, 0x000000ff, 0x0000000f, 0x00000003, 0x00000003, 0x00000003},
  {0x00007fff, 0x0000003f, 0x00000007, 0x00000007, 0x00000007, 0x00000007},
  {0x0000ffff, 0x000000ff, 0x0000000f, 0x0000000f, 0x0000000f, 0x0000000f},
  {0x00007fff, 0x0000001f, 0x0000001f, 0x0000001f, 0x0000001f, 0x0000001f},
  {0x00000fff, 0x0000003f, 0x0000003f, 0x0000003f, 0x0000003f, 0x0000003f},
  {0x00003fff, 0x0000007f, 0x0000007f, 0x0000007f, 0x0000007f, 0x0000007f},
  {0x0000ffff, 0x000000ff, 0x000000ff, 0x000000ff, 0x000000ff, 0x000000ff},
  {0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff, 0x000001ff},
  {0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff, 0x000003ff},
  {0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff, 0x000007ff},
  {0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff, 0x00000fff},
  {0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff, 0x00001fff},
  {0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff, 0x00003fff},
  {0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff, 0x00007fff},
  {0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff},
  {0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff, 0x0001ffff},
  {0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff, 0x0003ffff},
  {0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff, 0x0007ffff},
  {0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff, 0x000fffff},
  {0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff, 0x001fffff},
  {0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff, 0x003fffff},
  {0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff, 0x007fffff},
  {0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff, 0x00ffffff},
  {0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff, 0x01ffffff},
  {0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff, 0x03ffffff},
  {0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff, 0x07ffffff},
  {0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff},
  {0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff, 0x1fffffff},
  {0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff, 0x3fffffff},
  {0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff}
};

unsigned int n;       // numerator
const unsigned int s; // s > 0
const unsigned int d = (1 << s) - 1; // so d is either 1, 3, 7, 15, 31, ...).
unsigned int m;       // n % d goes here.

m = (n & M[s]) + ((n >> s) & M[s]);

for (const unsigned int * q = &Q[s][0], * r = &R[s][0]; m > d; q++, r++) {
    m = (m >> *q) + (m & *r);
}
m = m == d ? 0 : m; // OR, less portably: m = m & -((signed)(m - d) >> s);
```

This method of finding modulus division by an integer that is one less than a power of 2 takes at most O(lg(N)) time, where N is the number of bits in the numerator (32 bits, for the code above). The number of operations is at most 12 + 9 * ceil(lg(N)). The tables may be removed if you know the denominator at compile time; just extract the few relevent entries and unroll the loop. It may be easily extended to more bits.

It finds the result by summing the values in base (1 << s) in parallel. First every other base (1 << s) value is added to the previous one. Imagine that the result is written on a piece of paper. Cut the paper in half, so that half the values are on each cut piece. Align the values and sum them onto a new piece of paper. Repeat by cutting this paper in half (which will be a quarter of the size of

   the previous one) and summing, until you cannot cut further. After performing lg(N/s/2) cuts, we cut
   no more; just continue to add the values and put the result onto a new piece of paper as before,
   while there are at least two s-bit values.

   Devised by Sean Anderson, August 20, 2001. A typo was spotted by Randy E. Bryant on May 3, 2005
   (after pasting the code, I had later added "unsinged" to a variable declaration). As in the previous
   hack, I mistakenly commented that we could alternatively assign m = ((m + 1) & d) - 1; at the end,
   and Don Knuth corrected me on April 19, 2006 and suggested m = m & -((signed)(m - d) >> s). On June
   18, 2009 Sean Irvine proposed a change that used ((n >> s) & M[s]) instead of ((n & M[s]) >> s),
   which typically requires fewer operations because the M[s] constant is already loaded.

Find the log base 2 of an integer with the MSB N set in O(N) operations (the obvious way)

```
unsigned int v; // 32-bit word to find the log base 2 of
unsigned int r = 0; // r will be lg(v)

while (v >>= 1) { // unroll for more speed...
    r++;
}
```

   The log base 2 of an integer is the same as the position of the highest bit set (or most significant
   bit set, MSB). The following log base 2 methods are faster than this one.

Find the integer log base 2 of an integer with an 64-bit IEEE float

```
int v; // 32-bit integer to find the log base 2 of
int r; // result of log_2(v) goes here
union { unsigned int u[2]; double d; } t; // temp

t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] = 0x43300000;
t.u[__FLOAT_WORD_ORDER!=LITTLE_ENDIAN] = v;
t.d -= 4503599627370496.0;
r = (t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] >> 20) - 0x3FF;
```

   The code above loads a 64-bit (IEEE-754 floating-point) double with a 32-bit integer (with no
   paddding bits) by storing the integer in the mantissa while the exponent is set to 2^52. From this
   newly minted double, 2^52 (expressed as a double) is subtracted, which sets the resulting exponent to
   the log base 2 of the input value, v. All that is left is shifting the exponent bits into position
   (20 bits right) and subtracting the bias, 0x3FF (which is 1023 decimal). This technique only takes 5
   operations, but many CPUs are slow at manipulating doubles, and the endianess of the architecture
   must be accommodated.

   Eric Cole sent me this on January 15, 2006. Evan Felix pointed out a typo on April 4, 2006. Vincent
   LefÃ vre told me on July 9, 2008 to change the endian check to use the float's endian, which could
   differ from the integer's endian.

Find the log base 2 of an integer with a lookup table

```
static const char LogTable256[256] =
{
#define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
    -1, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
    LT(4), LT(5), LT(5), LT(6), LT(6), LT(6), LT(6),
    LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
};

unsigned int v; // 32-bit word to find the log of
unsigned r;     // r will be lg(v)
register unsigned int t, tt; // temporaries

if (tt = v >> 16) {
    r = (t = tt >> 8) ? 24 + LogTable256[t] : 16 + LogTable256[tt];
} else {
    r = (t = v >> 8) ? 8 + LogTable256[t] : LogTable256[v];
}
```

   The lookup table method takes only about 7 operations to find the log of a 32-bit value. If extended
   for 64-bit quantities, it would take roughly 9 operations. Another operation can be trimmed off by
   using four tables, with the possible additions incorporated into each. Using int table elements may
   be faster, depending on your architecture.

   The code above is tuned to uniformly distributed output values. If your inputs are evenly distributed
   across all 32-bit values, then consider using the following:

```
if (tt = v >> 24) {
    r = 24 + LogTable256[tt];
} else if (tt = v >> 16) {
    r = 16 + LogTable256[tt];
} else if (tt = v >> 8) {
    r = 8 + LogTable256[tt];
} else {
    r = LogTable256[v];
}
```

   To initially generate the log table algorithmically:

```
LogTable256[0] = LogTable256[1] = 0;
for (int i = 2; i < 256; i++) {
    LogTable256[i] = 1 + LogTable256[i / 2];
}
LogTable256[0] = -1; // if you want log(0) to return -1
```

Behdad Esfahbod and I shaved off a fraction of an operation (on average) on May 18, 2005. Yet another fraction of an operation was removed on November 14, 2006 by Emanuel Hoogeveen. The variation that is tuned to evenly distributed input values was suggested by David A. Butterfield on September 19, 2008. Venkat Reddy told me on January 5, 2009 that log(0) should return -1 to indicate an error, so I changed the first entry in the table to that.

Find the log base 2 of an N-bit integer in O(lg(N)) operations

```
unsigned int v;  // 32-bit value to find the log2 of
const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000};
const unsigned int S[] = {1, 2, 4, 8, 16};
int i;

register unsigned int r = 0; // result of log2(v) will go here
for (i = 4; i >= 0; i--) { // unroll for speed...
    if (v & b[i]) {
        v >>= S[i];
        r |= S[i];
    }
}


// OR (IF YOUR CPU BRANCHES SLOWLY):

unsigned int v;          // 32-bit value to find the log2 of
register unsigned int r; // result of log2(v) will go here
register unsigned int shift;

r =     (v > 0xFFFF) << 4; v >>= r;
shift = (v > 0xFF  ) << 3; v >>= shift; r |= shift;
shift = (v > 0xF   ) << 2; v >>= shift; r |= shift;
shift = (v > 0x3   ) << 1; v >>= shift; r |= shift;
                                        r |= (v >> 1);


// OR (IF YOU KNOW v IS A POWER OF 2):

unsigned int v;  // 32-bit value to find the log2 of
static const unsigned int b[] = {0xAAAAAAAA, 0xCCCCCCCC, 0xF0F0F0F0,
                                 0xFF00FF00, 0xFFFF0000};
register unsigned int r = (v & b[0]) != 0;
for (i = 4; i > 0; i--) { // unroll for speed...
    r |= ((v & b[i]) != 0) << i;
}
```

Of course, to extend the code to find the log of a 33- to 64-bit number, we would append another element, 0xFFFFFFFF00000000, to b, append 32 to S, and loop from 5 to 0. This method is much slower than the earlier table-lookup version, but if you don't want big table or your architecture is slow to access memory, it's a good choice. The second variation involves slightly more operations, but it may be faster on machines with high branch costs (e.g. PowerPC).

The second version was sent to me by Eric Cole on January 7, 2006. Andrew Shapira subsequently trimmed a few operations off of it and sent me his variation (above) on Sept. 1, 2007. The third variation was suggested to me by John Owens on April 24, 2002; it's faster, but it is only suitable when the input is known to be a power of 2. On May 25, 2003, Ken Raeburn suggested improving the general case by using smaller numbers for b[], which load faster on some architectures (for instance if the word size is 16 bits, then only one load instruction may be needed). These values work for the general version, but not for the special-case version below it, where v is a power of 2; Glenn Slayden brought this oversight to my attention on December 12, 2003.

Find the log base 2 of an N-bit integer in O(lg(N)) operations with multiply and lookup

```
uint32_t v; // find the log base 2 of 32-bit v
int r;      // result goes here

static const int MultiplyDeBruijnBitPosition[32] =
{
  0, 9, 1, 10, 13, 21, 2, 29, 11, 14, 16, 18, 22, 25, 3, 30,
  8, 12, 20, 28, 15, 17, 24, 7, 19, 27, 23, 6, 26, 5, 4, 31
};

v |= v >> 1; // first round down to one less than a power of 2
v |= v >> 2;
v |= v >> 4;
v |= v >> 8;
v |= v >> 16;

r = MultiplyDeBruijnBitPosition[(uint32_t)(v * 0x07C4ACDDU) >> 27];
```

The code above computes the log base 2 of a 32-bit integer with a small table lookup and multiply. It requires only 13 operations, compared to (up to) 20 for the previous method. The purely table-based method requires the fewest operations, but this offers a reasonable compromise between table size and speed.

If you know that v is a power of 2, then you only need the following:

```
static const int MultiplyDeBruijnBitPosition2[32] =
{
  0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
  31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9
};
r = MultiplyDeBruijnBitPosition2[(uint32_t)(v * 0x077CB531U) >> 27];
```

Eric Cole devised this January 8, 2006 after reading about the entry below to round up to a power of 2 and the method below for computing the number of trailing bits with a multiply and lookup using a DeBruijn sequence. On December 10, 2009, Mark Dickinson shaved off a couple operations by requiring v be rounded up to one less than the next power of 2 rather than the power of 2.

Find integer log base 10 of an integer

```
unsigned int v; // non-zero 32-bit integer value to compute the log base 10 of
int r;          // result goes here
int t;          // temporary

static unsigned int const PowersOf10[] =
    {1, 10, 100, 1000, 10000, 100000,
     1000000, 10000000, 100000000, 1000000000};

t = (IntegerLogBase2(v) + 1) * 1233 >> 12; // (use a lg2 method from above)
r = t - (v < PowersOf10[t]);
```

The integer log base 10 is computed by first using one of the techniques above for finding the log base 2. By the relationship log[10](v) = log[2](v) / log[2](10), we need to multiply it by 1/log[2](10), which is approximately 1233/4096, or 1233 followed by a right shift of 12. Adding one is needed because the IntegerLogBase2 rounds down. Finally, since the value t is only an approximation that may be off by one, the exact value is found by subtracting the result of v < PowersOf10[t
].

This method takes 6 more operations than IntegerLogBase2. It may be sped up (on machines with fast memory access) by modifying the log base 2 table-lookup method above so that the entries hold what is computed for t (that is, pre-add, -mulitply, and -shift). Doing so would require a total of only 9 operations to find the log base 10, assuming 4 tables were used (one for each byte of v).

Eric Cole suggested I add a version of this on January 7, 2006.

Find integer log base 10 of an integer the obvious way

```
unsigned int v; // non-zero 32-bit integer value to compute the log base 10 of
int r;          // result goes here

r = (v >= 1000000000) ? 9 : (v >= 100000000) ? 8 : (v >= 10000000) ? 7 :
    (v >= 1000000) ? 6 : (v >= 100000) ? 5 : (v >= 10000) ? 4 :
    (v >= 1000) ? 3 : (v >= 100) ? 2 : (v >= 10) ? 1 : 0;
```

This method works well when the input is uniformly distributed over 32-bit values because 76% of the inputs are caught by the first compare, 21% are caught by the second compare, 2% are caught by the third, and so on (chopping the remaining down by 90% with each comparision). As a result, less than 2.6 operations are needed on average.

On April 18, 2007, Emanuel Hoogeveen suggested a variation on this where the conditions used divisions, which were not as fast as simple comparisons.

Find integer log base 2 of a 32-bit IEEE float

```
const float v; // find int(log2(v)), where v > 0.0 && finite(v) && isnormal(v)
int c;         // 32-bit int c gets the result;

c = *(const int *) &v;  // OR, for portability:  memcpy(&c, &v, sizeof c);
c = (c >> 23) - 127;
```

The above is fast, but IEEE 754-compliant architectures utilize subnormal (also called denormal) floating point numbers. These have the exponent bits set to zero (signifying pow(2,-127)), and the mantissa is not normalized, so it contains leading zeros and thus the log2 must be computed from the mantissa. To accomodate for subnormal numbers, use the following:

```
const float v;              // find int(log2(v)), where v > 0.0 && finite(v)
int c;                      // 32-bit int c gets the result;
int x = *(const int *) &v;  // OR, for portability:  memcpy(&x, &v, sizeof x);

c = x >> 23;

if (c) {
    c -= 127;
} else { // subnormal, so recompute using mantissa: c = intlog2(x) - 149;
    register unsigned int t; // temporary
                             // Note that LogTable256 was defined earlier
    if (t = x >> 16) {
        c = LogTable256[t] - 133;
    } else {
        c = (t = x >> 8) ? LogTable256[t] - 141 : LogTable256[x] - 149;
    }
}
```

   On June 20, 2004, Sean A. Irvine suggested that I include code to handle subnormal numbers. On June 11, 2005, Falk HÃ¼ffner pointed out that ISO C99 6.5/7 specified undefined behavior for the common type punning idiom *(int *)&, though it has worked on 99.9% of C compilers. He proposed using memcpy for maximum portability or a union with a float and an int for better code generation than memcpy on some compilers.

Find integer log base 2 of the pow(2, r)-root of a 32-bit IEEE float (for unsigned integer r)

```
const int r;
const float v; // find int(log2(pow((double) v, 1. / pow(2, r)))),
               // where isnormal(v) and v > 0
int c;         // 32-bit int c gets the result;

c = *(const int *) &v;  // OR, for portability:  memcpy(&c, &v, sizeof c);
c = ((((c - 0x3f800000) >> r) + 0x3f800000) >> 23) - 127;
```

   So, if r is 0, for example, we have c = int(log2((double) v)). If r is 1, then we have c = int(log2(sqrt((double) v))). If r is 2, then we have c = int(log2(pow((double) v, 1./4))).

   On June 11, 2005, Falk HÃ¼ffner pointed out that ISO C99 6.5/7 left the type punning idiom *(int *)& undefined, and he suggested using memcpy.

Count the consecutive zero bits (trailing) on the right linearly

```
unsigned int v;  // input to count trailing zero bits
int c;  // output: c will count v's trailing zero bits,
        // so if v is 1101000 (base 2), then c will be 3
if (v) {
    v = (v ^ (v - 1)) >> 1;  // Set v's trailing 0s to 1s and zero rest
    for (c = 0; v; c++) {
        v >>= 1;
    }
} else {
    c = CHAR_BIT * sizeof(v);
}
```

   The average number of trailing zero bits in a (uniformly distributed) random binary number is one, so this O(trailing zeros) solution isn't that bad compared to the faster methods below.

   Jim Cole suggested I add a linear-time method for counting the trailing zeros on August 15, 2007. On October 22, 2007, Jason Cunningham pointed out that I had neglected to paste the unsigned modifier for v.

Count the consecutive zero bits (trailing) on the right in parallel

```
unsigned int v;      // 32-bit word input to count zero bits on right
unsigned int c = 32; // c will be the number of zero bits on the right
v &= -signed(v);
if (v) c--;
if (v & 0x0000FFFF) c -= 16;
if (v & 0x00FF00FF) c -= 8;
if (v & 0x0F0F0F0F) c -= 4;
if (v & 0x33333333) c -= 2;
if (v & 0x55555555) c -= 1;
```

   Here, we are basically doing the same operations as finding the log base 2 in parallel, but we first isolate the lowest 1 bit, and then proceed with c starting at the maximum and decreasing. The number of operations is at most 3 * lg(N) + 4, roughly, for N bit words.

   Bill Burdick suggested an optimization, reducing the time from 4 * lg(N) on February 4, 2011.

Count the consecutive zero bits (trailing) on the right by binary search

```
unsigned int v;     // 32-bit word input to count zero bits on right
unsigned int c;     // c will be the number of zero bits on the right,
                    // so if v is 1101000 (base 2), then c will be 3
// NOTE: if 0 == v, then c = 31.
if (v & 0x1) {
    // special case for odd v (assumed to happen half of the time)
    c = 0;
} else {
    c = 1;
    if ((v & 0xffff) == 0) {
        v >>= 16;
        c += 16;
    }

    if ((v & 0xff) == 0) {
        v >>= 8;
        c += 8;
    }

    if ((v & 0xf) == 0) {
        v >>= 4;
        c += 4;
    }

    if ((v & 0x3) == 0) {
        v >>= 2;
        c += 2;
    }

    c -= v & 0x1;
}
```

   The code above is similar to the previous method, but it computes the number of trailing zeros by accumulating c in a manner akin to binary search. In the first step, it checks if the bottom 16 bits of v are zeros, and if so, shifts v right 16 bits and adds 16 to c, which reduces the number of bits in v to consider by half. Each of the subsequent conditional steps likewise halves the number of bits until there is only 1. This method is faster than the last one (by about 33%) because the bodies of the if statements are executed less often.

   Matt Whitlock suggested this on January 25, 2006. Andrew Shapira shaved a couple operations off on Sept. 5, 2007 (by setting c=1 and unconditionally subtracting at the end).

Count the consecutive zero bits (trailing) on the right by casting to a float

```
unsigned int v;            // find the number of trailing zeros in v
int r;                     // the result goes here
float f = (float)(v & -v); // cast the least significant bit in v to a float
r = (*(uint32_t *)&f >> 23) - 0x7f;
```

   Although this only takes about 6 operations, the time to convert an integer to a float can be high on some machines. The exponent of the 32-bit IEEE floating point representation is shifted down, and the bias is subtracted to give the position of the least significant 1 bit set in v. If v is zero, then the result is -127.

Count the consecutive zero bits (trailing) on the right with modulus division and lookup

```
unsigned int v;  // find the number of trailing zeros in v
int r;           // put the result in r
static const int Mod37BitPosition[] = // map a bit value mod 37 to its position
{
  32, 0, 1, 26, 2, 23, 27, 0, 3, 16, 24, 30, 28, 11, 0, 13, 4,
  7, 17, 0, 25, 22, 31, 15, 29, 10, 12, 6, 0, 21, 14, 9, 5,
  20, 8, 19, 18
};
r = Mod37BitPosition[(-v & v) % 37];
```

   The code above finds the number of zeros that are trailing on the right, so binary 0100 would produce 2. It makes use of the fact that the first 32 bit position values are relatively prime with 37, so performing a modulus division with 37 gives a unique number from 0 to 36 for each. These numbers may then be mapped to the number of zeros using a small lookup table. It uses only 4 operations, however indexing into a table and performing modulus division may make it unsuitable for some situations. I came up with this independently and then searched for a subsequence of the table values, and found it was invented earlier by Reiser, according to Hacker's Delight.

Count the consecutive zero bits (trailing) on the right with multiply and lookup

```
unsigned int v;  // find the number of trailing zeros in 32-bit v
int r;           // result goes here
static const int MultiplyDeBruijnBitPosition[32] =
{
  0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
  31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9
};
r = MultiplyDeBruijnBitPosition[((uint32_t)((v & -v) * 0x077CB531U)) >> 27];
```

   Converting bit vectors to indices of set bits is an example use for this. It requires one more operation than the earlier one involving modulus division, but the multiply may be faster. The expression (v & -v) extracts the least significant 1 bit from v. The constant 0x077CB531UL is a de Bruijn sequence, which produces a unique pattern of bits into the high 5 bits for each possible bit position that it is multiplied against. When there are no bits set, it returns 0. More information can be found by reading the paper Using de Bruijn Sequences to Index 1 in a Computer Word by Charles E. Leiserson, Harald Prokof, and Keith H. Randall.

   On October 8, 2005 Andrew Shapira suggested I add this. Dustin Spicuzza asked me on April 14, 2009 to cast the result of the multiply to a 32-bit type so it would work when compiled with 64-bit ints.

Round up to the next highest power of 2 by float casting

```
unsigned int const v; // Round this 32-bit value to the next highest power of 2
unsigned int r;       // Put the result here. (So v=3 -> r=4; v=8 -> r=8)

if (v > 1) {
    float f = (float)v;
    unsigned int const t = 1U << ((*(unsigned int *)&f >> 23) - 0x7f);
    r = t << (t < v);
} else {
    r = 1;
}
```

The code above uses 8 operations, but works on all v <= (1<<31).

Quick and dirty version, for domain of 1 < v < (1<<25):

```
float f = (float)(v - 1);
r = 1U << ((*(unsigned int*)(&f) >> 23) - 126);
```

Although the quick and dirty version only uses around 6 operations, it is roughly three times slower than the technique below (which involves 12 operations) when benchmarked on an Athlon XP 2100+ CPU. Some CPUs will fare better with it, though.

On September 27, 2005 Andi Smithers suggested I include a technique for casting to floats to find the lg of a number for rounding up to a power of 2. Similar to the quick and dirty version here, his version worked with values less than (1<<25), due to mantissa rounding, but it used one more operation.

Round up to the next highest power of 2

```
unsigned int v; // compute the next highest power of 2 of 32-bit v

v--;
v |= v >> 1;
v |= v >> 2;
v |= v >> 4;
v |= v >> 8;
v |= v >> 16;
v++;
```

In 12 operations, this code computes the next highest power of 2 for a 32-bit integer. The result may be expressed by the formula 1U << (lg(v - 1) + 1). Note that in the edge case where v is 0, it returns 0, which isn't a power of 2; you might append the expression v += (v == 0) to remedy this if it matters. It would be faster by 2 operations to use the formula and the log base 2 method that uses a lookup table, but in some situations, lookup tables are not suitable, so the above code may be best. (On a Athlon XP 2100+ I've found the above shift-left and then OR code is as fast as using a single BSR assembly language instruction, which scans in reverse to find the highest set bit.) It works by copying the highest set bit to all of the lower bits, and then adding one, which results in carries that set all of the lower bits to 0 and one bit beyond the highest set bit to 1. If the original number was a power of 2, then the decrement will reduce it to one less, so that we round up to the same original value.

You might alternatively compute the next higher power of 2 in only 8 or 9 operations using a lookup table for floor(lg(v)) and then evaluating 1<<(1+floor(lg(v))); Atul Divekar suggested I mention this on September 5, 2010.

Devised by Sean Anderson, Sepember 14, 2001. Pete Hart pointed me to a couple newsgroup posts by him and William Lewis in February of 1997, where they arrive at the same algorithm.

Interleave bits the obvious way

```
unsigned short x;   // Interleave bits of x and y, so that all of the
unsigned short y;   // bits of x are in the even positions and y in the odd;
unsigned int z = 0; // z gets the resulting Morton Number.

for (int i = 0; i < sizeof(x) * CHAR_BIT; i++) { // unroll for more speed...
    z |= (x & 1U << i) << i | (y & 1U << i) << (i + 1);
}
```

Interleaved bits (aka Morton numbers) are useful for linearizing 2D integer coordinates, so x and y are combined into a single number that can be compared easily and has the property that a number is usually close to another if their x and y values are close.

Interleave bits by table lookup

```
static const unsigned short MortonTable256[256] =
{
  0x0000, 0x0001, 0x0004, 0x0005, 0x0010, 0x0011, 0x0014, 0x0015,
  0x0040, 0x0041, 0x0044, 0x0045, 0x0050, 0x0051, 0x0054, 0x0055,
  0x0100, 0x0101, 0x0104, 0x0105, 0x0110, 0x0111, 0x0114, 0x0115,
  0x0140, 0x0141, 0x0144, 0x0145, 0x0150, 0x0151, 0x0154, 0x0155,
  0x0400, 0x0401, 0x0404, 0x0405, 0x0410, 0x0411, 0x0414, 0x0415,
  0x0440, 0x0441, 0x0444, 0x0445, 0x0450, 0x0451, 0x0454, 0x0455,
  0x0500, 0x0501, 0x0504, 0x0505, 0x0510, 0x0511, 0x0514, 0x0515,
  0x0540, 0x0541, 0x0544, 0x0545, 0x0550, 0x0551, 0x0554, 0x0555,
  0x1000, 0x1001, 0x1004, 0x1005, 0x1010, 0x1011, 0x1014, 0x1015,
  0x1040, 0x1041, 0x1044, 0x1045, 0x1050, 0x1051, 0x1054, 0x1055,
  0x1100, 0x1101, 0x1104, 0x1105, 0x1110, 0x1111, 0x1114, 0x1115,
  0x1140, 0x1141, 0x1144, 0x1145, 0x1150, 0x1151, 0x1154, 0x1155,
  0x1400, 0x1401, 0x1404, 0x1405, 0x1410, 0x1411, 0x1414, 0x1415,
  0x1440, 0x1441, 0x1444, 0x1445, 0x1450, 0x1451, 0x1454, 0x1455,
  0x1500, 0x1501, 0x1504, 0x1505, 0x1510, 0x1511, 0x1514, 0x1515,
  0x1540, 0x1541, 0x1544, 0x1545, 0x1550, 0x1551, 0x1554, 0x1555,
  0x4000, 0x4001, 0x4004, 0x4005, 0x4010, 0x4011, 0x4014, 0x4015,
  0x4040, 0x4041, 0x4044, 0x4045, 0x4050, 0x4051, 0x4054, 0x4055,
  0x4100, 0x4101, 0x4104, 0x4105, 0x4110, 0x4111, 0x4114, 0x4115,
  0x4140, 0x4141, 0x4144, 0x4145, 0x4150, 0x4151, 0x4154, 0x4155,
  0x4400, 0x4401, 0x4404, 0x4405, 0x4410, 0x4411, 0x4414, 0x4415,
  0x4440, 0x4441, 0x4444, 0x4445, 0x4450, 0x4451, 0x4454, 0x4455,
  0x4500, 0x4501, 0x4504, 0x4505, 0x4510, 0x4511, 0x4514, 0x4515,
  0x4540, 0x4541, 0x4544, 0x4545, 0x4550, 0x4551, 0x4554, 0x4555,
  0x5000, 0x5001, 0x5004, 0x5005, 0x5010, 0x5011, 0x5014, 0x5015,
  0x5040, 0x5041, 0x5044, 0x5045, 0x5050, 0x5051, 0x5054, 0x5055,
  0x5100, 0x5101, 0x5104, 0x5105, 0x5110, 0x5111, 0x5114, 0x5115,
  0x5140, 0x5141, 0x5144, 0x5145, 0x5150, 0x5151, 0x5154, 0x5155,
  0x5400, 0x5401, 0x5404, 0x5405, 0x5410, 0x5411, 0x5414, 0x5415,
  0x5440, 0x5441, 0x5444, 0x5445, 0x5450, 0x5451, 0x5454, 0x5455,
  0x5500, 0x5501, 0x5504, 0x5505, 0x5510, 0x5511, 0x5514, 0x5515,
  0x5540, 0x5541, 0x5544, 0x5545, 0x5550, 0x5551, 0x5554, 0x5555
};

unsigned short x; // Interleave bits of x and y, so that all of the
unsigned short y; // bits of x are in the even positions and y in the odd;
unsigned int z;   // z gets the resulting 32-bit Morton Number.

z = MortonTable256[y >> 8]   << 17 |
    MortonTable256[x >> 8]   << 16 |
    MortonTable256[y & 0xFF] <<  1 |
    MortonTable256[x & 0xFF];
```

For more speed, use an additional table with values that are MortonTable256 pre-shifted one bit to the left. This second table could then be used for the y lookups, thus reducing the operations by two, but almost doubling the memory required. Extending this same idea, four tables could be used, with two of them pre-shifted by 16 to the left of the previous two, so that we would only need 11 operations total.

Interleave bits with 64-bit multiply

In 11 operations, this version interleaves bits of two bytes (rather than shorts, as in the other versions), but many of the operations are 64-bit multiplies so it isn't appropriate for all machines. The input parameters, x and y, should be less than 256.

```
unsigned char x;  // Interleave bits of (8-bit) x and y, so that all of the
unsigned char y;  // bits of x are in the even positions and y in the odd;
unsigned short z; // z gets the resulting 16-bit Morton Number.

z = ((x * 0x0101010101010101ULL & 0x8040201008040201ULL) *
     0x0102040810204081ULL >> 49) & 0x5555 |
    ((y * 0x0101010101010101ULL & 0x8040201008040201ULL) *
     0x0102040810204081ULL >> 48) & 0xAAAA;
```

Holger Bettag was inspired to suggest this technique on October 10, 2004 after reading the multiply-based bit reversals here.

Interleave bits by Binary Magic Numbers

```
static const unsigned int B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF};
static const unsigned int S[] = {1, 2, 4, 8};

unsigned int x; // Interleave lower 16 bits of x and y, so the bits of x
unsigned int y; // are in the even positions and bits from y in the odd;
unsigned int z; // z gets the resulting 32-bit Morton Number.
                // x and y must initially be less than 65536.

x = (x | (x << S[3])) & B[3];
x = (x | (x << S[2])) & B[2];
x = (x | (x << S[1])) & B[1];
x = (x | (x << S[0])) & B[0];

y = (y | (y << S[3])) & B[3];
y = (y | (y << S[2])) & B[2];
y = (y | (y << S[1])) & B[1];
y = (y | (y << S[0])) & B[0];

z = x | (y << 1);
```

Determine if a word has a zero byte

```
// Fewer operations:
unsigned int v; // 32-bit word to check if any 8-bit byte in it is 0
bool hasZeroByte = ⍰((((v & 0x7F7F7F7F) + 0x7F7F7F7F) | v) | 0x7F7F7F7F);
```

The code above may be useful when doing a fast string copy in which a word is copied at a time; it uses 5 operations. On the other hand, testing for a null byte in the obvious ways (which follow) have at least 7 operations (when counted in the most sparing way), and at most 12.

```
// More operations:
bool hasNoZeroByte = ((v & 0xff) && (v & 0xff00) && (v & 0xff0000) && (v & 0xff000000))
```

```
// OR:
unsigned char * p = (unsigned char *) &v;
bool hasNoZeroByte = *p && *(p + 1) && *(p + 2) && *(p + 3);
```

The code at the beginning of this section (labeled "Fewer operations") works by first zeroing the high bits of the 4 bytes in the word. Subsequently, it adds a number that will result in an overflow to the high bit of a byte if any of the low bits were initialy set. Next the high bits of the original word are ORed with these values; thus, the high bit of a byte is set iff any bit in the byte was set. Finally, we determine if any of these high bits are zero by ORing with ones everywhere except the high bits and inverting the result. Extending to 64 bits is trivial; simply increase the constants to be 0x7F7F7F7F7F7F7F7F.

For an additional improvement, a fast pretest that requires only 4 operations may be performed to determine if the word may have a zero byte. The test also returns true if the high byte is 0x80, so there are occasional false positives, but the slower and more reliable version above may then be used on candidates for an overall increase in speed with correct output.

```
bool hasZeroByte = ((v + 0x7efefeff) ^ �v) & 0x81010100;
if (hasZeroByte) { // or may just have 0x80 in the high byte
    hasZeroByte = �((((v & 0x7F7F7F7F) + 0x7F7F7F7F) | v) | 0x7F7F7F7F);
}
```

There is yet a faster method - use hasless(v, 1), which is defined below; it works in 4 operations and requires no subsquent verification. It simplifies to

```
#define haszero(v) (((v) - 0x01010101UL) & �(v) & 0x80808080UL)
```

The subexpression (v - 0x01010101UL), evaluates to a high bit set in any byte whenever the corresponding byte in v is zero or greater than 0x80. The sub-expression �v & 0x80808080UL evaluates to high bits set in bytes where the byte of v doesn't have its high bit set (so the byte was less than 0x80). Finally, by ANDing these two sub-expressions the result is the high bits set where the bytes in v were zero, since the high bits set due to a value greater than 0x80 in the first sub-expression are masked off by the second.

Paul Messmer suggested the fast pretest improvement on October 2, 2004. Juha JÃ¤rvi later suggested hasless(v, 1) on April 6, 2005, which he found on Paul Hsieh's Assembly Lab; previously it was written in a newsgroup post on April 27, 1987 by Alan Mycroft.

Determine if a word has a byte equal to n

We may want to know if any byte in a word has a specific value. To do so, we can XOR the value to test with a word that has been filled with the byte values in which we're interested. Because XORing a value with itself results in a zero byte and nonzero otherwise, we can pass the result to haszero.

```
#define hasvalue(x,n) \
(haszero((x) ^ (�0UL/255 * (n))))
```

Stephen M Bennet suggested this on December 13, 2009 after reading the entry for haszero.

Determine if a word has a byte less than n

Test if a word x contains an unsigned byte with value < n. Specifically for n=1, it can be used to find a 0-byte by examining one long at a time, or any byte by XORing x with a mask first. Uses 4 arithmetic/logical operations when n is constant.

Requirements: x>=0; 0<=n<=128

```
#define hasless(x,n) (((x)-�0UL/255*(n))&�(x)&�0UL/255*128)
```

To count the number of bytes in x that are less than n in 7 operations, use

```
#define countless(x,n) \
(((�0UL/255*(127+(n))-((x)&�0UL/255*127))&�(x)&�0UL/255*128)/128%255)
```

Juha JÃ¤rvi sent this clever technique to me on April 6, 2005. The countless macro was added by Sean Anderson on April 10, 2005, inspired by Juha's countmore, below.

Determine if a word has a byte greater than n

Test if a word x contains an unsigned byte with value > n. Uses 3 arithmetic/logical operations when n is constant.

Requirements: x>=0; 0<=n<=127

```
#define hasmore(x,n) (((x)+�0UL/255*(127-(n))|(x))&�0UL/255*128)
```

To count the number of bytes in x that are more than n in 6 operations, use:

```
#define countmore(x,n) \
(((((x)&�0UL/255*127)+�0UL/255*(127-(n))|(x))&�0UL/255*128)/128%255)
```

The macro hasmore was suggested by Juha JÃ¤rvi on April 6, 2005, and he added countmore on April 8, 2005.

Determine if a word has a byte between m and n

When m < n, this technique tests if a word x contains an unsigned byte value, such that m < value < n. It uses 7 arithmetic/logical operations when n and m are constant.

Note: Bytes that equal n can be reported by likelyhasbetween as false positives, so this should be checked by character if a certain result is needed.

Requirements: x>=0; 0<=m<=127; 0<=n<=128

```
#define likelyhasbetween(x,m,n) \
((((x)-⍰0UL/255*(n))&⍰(x)&((x)&⍰0UL/255*127)+⍰0UL/255*(127-(m)))&⍰0UL/255*128)
```

This technique would be suitable for a fast pretest. A variation that takes one more operation (8 total for constant m and n) but provides the exact answer is:

```
#define hasbetween(x,m,n) \
((⍰0UL/255*(127+(n))-((x)&⍰0UL/255*127)&⍰(x)&((x)&⍰0UL/255*127)+⍰0UL/255*(127-(m)))&⍰0UL/255*128)
```

To count the number of bytes in x that are between m and n (exclusive) in 10 operations, use:

```
#define countbetween(x,m,n) (hasbetween(x,m,n)/128%255)
```

Juha JÃ¤rvi suggested likelyhasbetween on April 6, 2005. From there, Sean Anderson created hasbetween and countbetween on April 10, 2005.

Compute the lexicographically next bit permutation

Suppose we have a pattern of N bits set to 1 in an integer and we want the next permutation of N 1 bits in a lexicographical sense. For example, if N is 3 and the bit pattern is 00010011, the next patterns would be 00010101, 00010110, 00011001,00011010, 00011100, 00100011, and so forth. The following is a fast way to compute the next permutation.

```
unsigned int v; // current permutation of bits
unsigned int w; // next permutation of bits

unsigned int t = v | (v - 1); // t gets v's least significant 0 bits set to 1
// Next set to 1 the most significant bit to change,
// set to 0 the least significant ones, and add the necessary 1 bits.
w = (t + 1) | (((⍰t & -⍰t) - 1) >> (__builtin_ctz(v) + 1));
```

The __builtin_ctz(v) GNU C compiler intrinsic for x86 CPUs returns the number of trailing zeros. If you are using Microsoft compilers for x86, the intrinsic is _BitScanForward. These both emit a bsf instruction, but equivalents may be available for other architectures. If not, then consider using one of the methods for counting the consecutive zero bits mentioned earlier.

Here is another version that tends to be slower because of its division operator, but it does not require counting the trailing zeros.

```
unsigned int t = (v | (v - 1)) + 1;
w = t | ((((t & -t) / (v & -v)) >> 1) - 1);
```

Thanks to Dario Sneidermanis of Argentina, who provided this on November 28, 2009.

---
https://developerinsider.co/awesome-bitwise-operations-and-tricks-with-examples/

Awesome Bitwise Operations and Tricks with Examples

1. Set n^th bit of integer x

```
x | (1<<n)
```

Example

```
#include<stdio.h>

int main() {
    int x = 10; //1010
    int n = 2;
    int result = x | (1<<n);  //1110
    printf("%d\n", result); //14
    return 0;
}
```

2. Unset n^th bit of integer x

```
x & ⍰(1<<n)
```

Example

```
#include<stdio.h>

int main() {
    int x = 10; //1010
    int n = 1;
    int result = x & ⍰(1<<n); //1000
    printf("%d", result2);  //8
    return 0;
}
```

3. Toggle n^th bit of x

```
x ^ (1<<n)
```

```
   Example
#include<stdio.h>

int main() {
    int x = 10; //1010
    int n = 0;
    int result = x ^ (1<<n); //1011
    n = 3;
    result = result ^ (1<<n); //0011
    printf("%d\n", result); //3
    return 0;
}
```

4. Multiply integer x by the n^th power of 2

```
x << n

   Example
#include<stdio.h>

int main() {
    int x = 10;
    int n = 3;
    int result = x << n;  // 10 * (2^3)
    printf("%d\n", result); //80
    return 0;
}
```

5. Divide integer x by the n^th power of 2

```
x >> n;

   Example
#include<stdio.h>

int main() {
    int x = 80;
    int n = 3;
    int result = x >> n;    // 80 / (2^3)
    printf("%d\n", result); //10
    return 0;
}
```

6. Check equality of two integer

```
(num1 ^ num2) == 0; // num1 == num2

   Example
#include<stdio.h>

int main() {
    int num1 = 10;
    int num2 = 10;
    if ((num1 ^ num2) == 0)
        printf("Equal");
    else
        printf("Not Equal");
    return 0;
}
```

7. Check if an integer number is odd

```
(num & 1) == 1

   Example
#include<stdio.h>

int main() {
    int num = 13;
    if ((num & 1) == 1)
        printf("Odd");
    else
        printf("Even");
    return 0;
}
```

8. Swap two integer values

```
//version 1
a ^= b;
b ^= a;
a ^= b;

//version 2
a = a ^ b ^ (b = a)
```

   Example
```
#include<stdio.h>

int main() {
    int a = 5;
    int b = 7;

    //Version 1
    a ^= b;
    b ^= a;
    a ^= b;

    printf("a = %d & b = %d\n", a, b); // a = 7 & b = 5

    //Version 2
    a = a ^ b ^ (b = a);

    printf("a = %d & b = %d", a, b); // a = 5 & b = 7
    return 0;
}
```

9. Get the max of two integer values

```
b & ((a-b) >> 31) | a & (⍰(a-b) >> 31);
```

   Example
```
#include<stdio.h>

int main() {
    int a = 5;
    int b = 7;

    int max = b & ((a-b) >> 31) | a & (⍰(a-b) >> 31);
    printf("%d", max);

    return 0;
}
```

10. Get the min of two integer values

```
a & ((a-b) >> 31) | b & (⍰(a-b) >> 31);
```

   Example
```
#include<stdio.h>

int main() {
    int a = 5;
    int b = 7;

    int min = a & ((a-b) >> 31) | b & (⍰(a-b) >> 31);
    printf("%d", min);

    return 0;
}
```

11. Check whether both integer numbers have the same sign

```
(num1 ^ num2) >= 0;
```

   Example
```
#include<stdio.h>

int main() {
    int num1 = 5;
    int num2 = -7;

    if ((num1 ^ num2) >= 0)
        printf("Same Sign");
    else
        printf("Different Sign");

    return 0;
}
```

12. Flip the sign of an integer number

```
num = ⍰num + 1;
```

   Example
```
#include<stdio.h>

int main() {
    int num = 5;

    num = ⍰num + 1;
    printf("num = %d", num);

    return 0;
}
```

13. Check whether a integer number is power of 2

```
num > 0 && (num & (num - 1)) == 0;
```

   Example

```
#include<stdio.h>

int main() {
    int num = 16;

    if (num > 0 && (num & (num - 1)) == 0)
        printf("Number is power of 2");
    else
        printf("Number is not power of 2");
    return 0;
}
```

14. Increment by 1 (num + 1)

```
-�num
```

   Example

```
#include<stdio.h>

int main() {
    int num = 16;

    num = -�num;
    printf("num = %d", num);

    return 0;
}
```

15. Decrement by 1 (num - 1)

```
�-num
```

   Example

```
#include<stdio.h>

int main() {
    int num = 16;

    num = �-num;
    printf("num = %d", num);

    return 0;
}
```

---
https://github.com/keon/awesome-bits

awesome-bits Awesome

     A curated list of awesome bitwise operations and tricks

Integers

   Set n^th bit

```
x | (1<<n)
```

   Unset n^th bit

```
x & �(1<<n)
```

   Toggle n^th bit

```
x ^ (1<<n)
```

   Round up to the next power of two

```
unsigned int v; //only works if v is 32 bit
v--;
v |= v >> 1;
v |= v >> 2;
v |= v >> 4;
v |= v >> 8;
v |= v >> 16;
v++;
```

   Round down / floor a number

```
n >> 0
```

```
5.7812 >> 0 // 5


   Check if even
(n & 1) == 0

   Check if odd
(n & 1) != 0

   Get the maximum integer
int maxInt = ⍰(1 << 31);
int maxInt = (1 << 31) - 1;
int maxInt = (1 << -1) - 1;
int maxInt = -1u >> 1;

   Get the minimum integer
int minInt = 1 << 31;
int minInt = 1 << -1;

   Get the maximum long
long maxLong = ((long)1 << 127) - 1;

   Multiply by 2
n << 1; // n*2

   Divide by 2
n >> 1; // n/2

   Multiply by the m^th power of 2
n << m;

   Divide by the m^th power of 2
n >> m;

   Check Equality

   [This is 35% faster in Javascript]
(a^b) == 0; // a == b
!(a^b) // use in an if

   Check if a number is odd
(n & 1) == 1;

   Exchange (swap) two values
//version 1
a ^= b;
b ^= a;
a ^= b;

//version 2
a = a ^ b ^ (b = a)

   Get the absolute value
//version 1
x < 0 ? -x : x;

//version 2
(x ^ (x >> 31)) - (x >> 31);

   Get the max of two values
b & ((a-b) >> 31) | a & (⍰(a-b) >> 31);

   Get the min of two values
a & ((a-b) >> 31) | b & (⍰(a-b) >> 31);

   Check whether both numbers have the same sign
(x ^ y) >= 0;

   Flip the sign
i = ⍰i + 1; // or
i = (i ^ -1) + 1; // i = -i

   Calculate 2^n
1 << n;

   Whether a number is power of 2
n > 0 && (n & (n - 1)) == 0;

   Modulo 2^n against m
m & ((1 << n) - 1);

   Get the average
(x + y) >> 1;
((x ^ y) >> 1) + (x & y);

   Get the m^th bit of n (from low to high)
```

```
(n >> (m-1)) & 1;

   Set the m^th bit of n to 0 (from low to high)
n & �(1 << (m-1));

   Check if n^th bit is set
if (x & (1<<n)) {
    n-th bit is set
} else {
    n-th bit is not set
}

   Isolate (extract) the right-most 1 bit
x & (-x)

   Isolate (extract) the right-most 0 bit
�x & (x+1)

   Set the right-most 0 bit to 1
x | (x+1)

   Set the right-most 1 bit to 0
x & (x-1)

   n + 1
-�n

   n - 1
�-n

   Get the negative value of a number
�n + 1;
(n ^ -1) + 1;

if (x == a) x = b; if (x == b) x = a;
x = a ^ b ^ x;

   Swap Adjacent bits
((n & 10101010) >> 1) | ((n & 01010101) << 1)

   Different rightmost bit of numbers m & n
(n^m)&-(n^m) // returns 2^x where x is the position of the different bit (0 based)

   Common rightmost bit of numbers m & n
�(n^m)&(n^m)+1 // returns 2^x where x is the position of the common bit (0 based)

Floats
   These are techniques inspired by the fast inverse square root method. Most of these are original.

   Turn a float into a bit-array (unsigned uint32_t)

#include <stdint.h>
typedef union {float flt; uint32_t bits} lens_t;
uint32_t f2i(float x) {
    return ((lens_t) {.flt = x}).bits;
}

   [Caveat: Type pruning via unions is undefined in C++; use std::memcpy instead.]

   Turn a bit-array back into a float
float i2f(uint32_t x) {
    return ((lens_t) {.bits = x}).flt;
}

   Approximate the bit-array of a positive float using frexp
   frexp gives the 2^n decomposition of a number, so that man, exp = frexp(x) means that man * 2^exp = x
   and 0.5 <= man < 1.
man, exp = frexp(x);
return (uint32_t)((2 * man + exp + 125) * 0x800000);

   [Caveat: This will have at most 2^-16 relative error, since man + 125 clobbers the last 8 bits,
   saving the first 16 bits of your mantissa.]

   Fast Inverse Square Root
return i2f(0x5f3759df - f2i(x) / 2);

   [Caveat: We're using the i2f and the f2i functions from above instead.]

   See [https://en.wikipedia.org/wiki/Fast_inverse_square_root#A_worked_example]this Wikipedia article for
   reference.

   Fast n^th Root of positive numbers via Infinite Series
float root(float x, int n) {
    #DEFINE MAN_MASK 0x7fffff
    #DEFINE EXP_MASK 0x7f800000
    #DEFINE EXP_BIAS 0x3f800000
    uint32_t bits = f2i(x);
    uint32_t man = bits & MAN_MASK;
    uint32_t exp = (bits & EXP_MASK) - EXP_BIAS;
    return i2f((man + man / n) | ((EXP_BIAS + exp / n) & EXP_MASK));
}
```

See [http://www.phailed.me/2012/08/somewhat-fast-square-root/]this blog post regarding the derivation.

Fast Arbitrary Power

```
return i2f((1 - exp) * (0x3f800000 - 0x5c416) + f2i(x) * exp)
```

[Caveat: The 0x5c416 bias is given to center the method. If you plug in exp = -0.5, this gives the 0x5f3759df magic constant of the fast inverse root method.]

See [http://www.bullshitmath.lol/FastRoot.slides.html]these set of slides for a derivation of this method.

Fast Geometric Mean

The geometric mean of a set of n numbers is the n^th root of their product.

```
#include <stddef.h>
float geometric_mean(float* list, size_t length) {
    // Effectively, find the average of map(f2i, list)
    uint32_t accumulator = 0;
    for (size_t i = 0; i < length; i++) {
        accumulator += f2i(list[i]);
    }
    return i2f(accumulator / n);
}
```

See [https://github.com/leegao/float-hacks#geometric-mean-1]here for its derivation.

Fast Natural Logarithm

```
#DEFINE EPSILON 1.1920928955078125e-07
#DEFINE LOG2 0.6931471805599453
return (f2i(x) - (0x3f800000 - 0x66774)) * EPSILON * LOG2
```

[Caveat: The bias term of 0x66774 is meant to center the method. We multiply by ln(2) at the end because the rest of the method computes the log2(x) function.]

See [https://github.com/leegao/float-hacks#log-1]here for its derivation.

Fast Natural Exp

```
return i2f(0x3f800000 + (uint32_t)(x * (0x800000 + 0x38aa22)))
```

[Caveat: The bias term of 0x38aa22 here corresponds to a multiplicative scaling of the base. In particular, it corresponds to z such that 2^z = e]

See [https://github.com/leegao/float-hacks#exp-1]here for its derivation.

Strings

Convert letter to lowercase:

```
OR by space => (x | ' ')
Result is always lowercase even if letter is already lowercase
eg. ('a' | ' ') => 'a' ; ('A' | ' ') => 'a'
```

Convert letter to uppercase:

```
AND by underline => (x & '_')
Result is always uppercase even if letter is already uppercase
eg. ('a' & '_') => 'A' ; ('A' & '_') => 'A'
```

Invert letter's case:

```
XOR by space => (x ^ ' ')
eg. ('a' ^ ' ') => 'A' ; ('A' ^ ' ') => 'a'
```

Letter's position in alphabet:

```
AND by chr(31)/binary('11111')/(hex('1F') => (x & "\x1F")
Result is in 1..26 range, letter case is not important
eg. ('a' & "\x1F") => 1 ; ('B' & "\x1F") => 2
```

Get letter's position in alphabet (for Uppercase letters only):

```
AND by ? => (x & '?') or XOR by @ => (x ^ '@')
eg. ('C' & '?') => 3 ; ('Z' ^ '@') => 26
```

Get letter's position in alphabet (for lowercase letters only):

```
XOR by backtick/chr(96)/binary('1100000')/hex('60') => (x ^ '`')
eg. ('d' ^ '`') => 4 ; ('x' ^ '`') => 24
```

Miscellaneous

Fast color conversion from R5G5B5 to R8G8B8 pixel format using shifts

```
R8 = (R5 << 3) | (R5 >> 2)
G8 = (G5 << 3) | (G5 >> 2)
B8 = (B5 << 3) | (B5 >> 2)
```

Note: using anything other than the English letters will produce garbage results

---
https://catonmat.net/low-level-bit-hacks

Introduction to Low Level Bit Hacks

I decided to write an article about a thing that is second nature to embedded systems programmers - low level bit hacks. Bit hacks are ingenious little programming tricks that manipulate integers in a smart and efficient manner. Instead of performing operations (such as counting the number of 1 bits in an integer) by looping over individual bits, these programming tricks do the same with one or two carefully chosen bitwise operations.

To get things going, I'll assume that you know what the two's complement binary representation of an integer is and also that you know all the the bitwise operations. I'll use the following notation for bitwise operations in the article:

```
&   -  bitwise and
|   -  bitwise or
^   -  bitwise xor
�   -  bitwise not
<< -  bitwise shift left
>>  -  bitwise shift right
```

Numbers in this article are 8 bit signed integers (though the operations work on arbitrary length signed integers) that are represented as two's complement and they are usually named 'x'. The result is usually 'y'. The individual bits of 'x' are named b7, b6, b5, b4, b3, b3, b2, b1 and b0. The bit b7 is the most significant bit (or in signed arithmetic - sign bit), and b0 is the least significant.

I'll start with the most basic bit hacks and gradually progress to more difficult ones. I'll use examples to explain how each bithack works.

If you like this topic, you can subscribe to my blog, or you can just read along. There is also going to be the second part of this article where I'll cover more advanced bit hacks, and I'll also release a cheat sheet with all these bit tricks.

Here we go!

Bit Hack #1. Check if the integer is even or odd.

```
if ((x & 1) == 0) {
    x is even
}
else {
    x is odd
}
```

I am pretty sure everyone has seen this trick. The idea here is that an integer is odd if and only if the least significant bit b0 is 1. It follows from the binary representation of 'x', where bit b0 contributes to either 1 or 0. By AND-ing 'x' with 1 we eliminate all the other bits than b0. If the result after this operation is 0, then 'x' was even because bit b0 was 0. Otherwise 'x' was odd.

Let's look at some examples. Let's take integer 43, which is odd. In binary 43 is 00101011. Notice that the least significant bit b0 is 1 (in bold). Now let's AND it with 1:

```
    00101011
&   00000001   (note: 1 is the same as 00000001)
    --------
    00000001
```

See how AND-ing erased all the higher order bits b1-b7 but left bit b0 the same it was? The result is thus 1 which tells us that the integer was odd.

Now let's look at -43. Just as a reminder, a quick way to find negative of a given number in two's complement representation is to invert all bits and add one. So -43 is 11010101 in binary. Again notice that the last bit is 1, and the integer is odd. (Note that if we used one's complement it wouldn't be true!)

Now let's take a look at an even integer 98. In binary 98 is 1100010.

```
    01100010
&   00000001
    --------
    00000000
```

After AND-ing the result is 0. It means that the bit b0 of original integer 98 was 0. Thus the given integer is even.

Now the negative -98. It's 10011110. Again, bit b0 is 0, after AND-ing, the result is 0, meaning -98 is even, which indeed is true.

Bit Hack #2. Test if the n-th bit is set.

```
if (x & (1<<n)) {
    n-th bit is set
}
else {
    n-th bit is not set
}
```

In the previous bit hack we saw that (x & 1) tests if the first bit is set. This bit hack improves this result and tests if n-th bit is set. It does it by shifting that first 1-bit n positions to the left and then doing the same AND operation, which eliminates all bits but n-th.

Here is what happens if you shift 1 several positions to the left:

```
1         00000001    (same as 1<<0)
1<<1      00000010
1<<2      00000100
1<<3      00001000
1<<4      00010000
1<<5      00100000
1<<6      01000000
1<<7      10000000
```

Now if we AND 'x' with 1 shifted n positions to the left we effectively eliminate all the bits but n-th bit in 'x'. If the result after AND-ing is 0, then that bit must have been 0, otherwise that bit was set.

Let's look at some examples.

Does 122 have 3rd bit set? The operation we do to find it out is:

```
122 & (1<<3)
```

Now, 122 is 01111010 in binary. And (1<<3) is 00001000.

```
    01111010
&   00001000
    --------
    00001000
```

We see that the result is not 0, so yes, 122 has the 3rd bit set.

Note: In my article bit numeration starts with 0. So it's 0th bit, 1st bit, ..., 7th bit.

What about -33? Does it have the 5th bit set?

```
    11011111      (-33 in binary)
&   00100000     (1<<5)
    --------
    00000000
```

Result is 0, so the 5th bit is not set.

Bit Hack #3. Set the n-th bit.

```
y = x | (1<<n)
```

This bit hack combines the same (1<<n) trick of setting n-th bit by shifting with OR operation. The result of OR-ing a variable with a value that has n-th bit set is turning that n-th bit on. It's because OR-ing any value with 0 leaves the value the same; but OR-ing it with 1 changes it to 1 (if it wasn't already). Let's see how that works in action:

Suppose we have value 120, and we want to turn on the 2nd bit.

```
    01111000    (120 in binary)
|   00000100    (1<<2)
    --------
    01111100
```

What about -120 and 6th bit?

```
    10001000   (-120 in binary)
|   01000000   (1<<6)
    --------
    11001000
```

Bit Hack #4. Unset the n-th bit.

```
y = x & ⍰(1<<n)
```

The important part of this bithack is the ⍰(1<<n) trick. It turns on all the bits except n-th.

Here is how it looks:

```
⍰1        11111110  (same as ⍰(1<<0))
⍰(1<<1)   11111101
⍰(1<<2)   11111011
⍰(1<<3)   11110111
⍰(1<<4)   11101111
⍰(1<<5)   11011111
⍰(1<<6)   10111111
⍰(1<<7)   01111111
```

The effect of AND-ing variable 'x' with this quantity is eliminating n-th bit. It does not matter if the n-th bit was 0 or 1, AND-ing it with 0 sets it to 0.

Here is an example. Let's unset 4th bit in 127:

```
    01111111    (127 in binary)
&   11101111    (⍰(1<<4))
    --------
    01101111
```

```
   Bit Hack #5. Toggle the n-th bit.
y = x ^ (1<<n)

   This bit hack also uses the wonderful "set n-th bit shift hack" but this time it XOR's it with the
   variable 'x'. The result of XOR-ing something with something else is that if both bits are the same,
   the result is 0, otherwise it's 1. How does it toggle n-th bit? Well, if n-th bit was 1, then XOR-ing
   it with 1 changes it to 0; conversely, if it was 0, then XOR-ing with with 1 changes it to 1. See,
   the bit got flipped.

   Here is an example. Suppose you want to toggle 5th bit in value 01110101:
    01110101
^   00100000
    --------
    01010101

   What about the same value but 5th bit originally 0?
    01010101
^   00100000
    --------
    01110101

   Notice something? XOR-ing the same bit twice returned it to the same value. This nifty XOR property
   is used in calculating parity in RAID arrays and used in simple cryptography cyphers, but more about
   that in some other article.

   Bit Hack #6. Turn off the rightmost 1-bit.
y = x & (x-1)

   Now it finally gets more interesting!!! Bit hacks #1 - #5 were kind of boring to be honest.

   This bit hack turns off the rightmost one-bit. For example, given an integer 00101010 (the rightmost
   1-bit in bold) it turns it into 00101000. Or given 00010000 it turns it into 0, as there is just a
   single 1-bit.

   Here are more examples:
    01010111    (x)
&   01010110    (x-1)
    --------
    01010110

    01011000    (x)
&   01010111    (x-1)
    --------
    01010000

    10000000    (x = -128)
&   01111111    (x-1 = 127 (with overflow))
    --------
    00000000

    11111111    (x = all bits 1)
&   11111110    (x-1)
    --------
    11111110

    00000000    (x = no rightmost 1-bits)
&   11111111    (x-1)
    --------
    00000000

   Why does it work?

   If you look at the examples and think for a while, you'll realize that there are two possible
   scenarios:
    1. The value has the rightmost 1 bit. In this case subtracting one from it sets all the lower bits
       to one and changes that rightmost bit to 0 (so that if you add one now, you get the original
       value back). This step has masked out the rightmost 1-bit and now AND-ing it with the original
       value zeroes that rightmost 1-bit out.
    2. The value has no rightmost 1 bit (all 0). In this case subtracting one underflows the value (as
       it's signed) and sets all bits to 1. AND-ing all zeroes with all ones produces 0.

   Bit Hack #7. Isolate the rightmost 1-bit.
y = x & (-x)

   This bit hack finds the rightmost 1-bit and sets all the other bits to 0. The end result has only
   that one rightmost 1-bit set. For example, 01010100 (rightmost bit in bold) gets turned into
   00000100.

   Here are some more examples:
    10111100  (x)
&   01000100  (-x)
    --------
    00000100

    01110000  (x)
&   10010000  (-x)
```

```
    --------
    00010000

    00000001  (x)
&   11111111  (-x)
    --------
    00000001

    10000000  (x = -128)
&   10000000  (-x = -128)
    --------
    10000000

    11111111  (x = all bits one)
&   00000001  (-x)
    --------
    00000001

    00000000  (x = all bits 0, no rightmost 1-bit)
&   00000000  (-x)
    --------
    00000000
```

This bit hack works because of two's complement. In two's complement system -x is the same as ~x+1.
Now let's examine the two possible cases:

1. There is a rightmost 1-bit b[i]. In this case let's pivot on this bit and divide all other bits into two flanks - bits to the right and bits to the left. Remember that all the bits to the right b[i-1], b[i-2] ... b[0] are 0's (because b[i] was the rightmost 1-bit). And bits to the left are the way they are. Let's call them b[i+1], ..., b[n].

Now, when we calculate -x, we first do ~x which turns bit b[i] into 0, bits b[i-1] ... b[0] into 1s, and inverts bits b[i+1], ..., b[n], and then we add 1 to this result.

Since bits b[i-1] ... b[0] are all 1's, adding one makes them carry this one all the way to bit b[i], which is the first zero bit.

If we put it all together, the result of calculating -x is that bits b[i+1], ..., b[n] get inverted, bit b[i] stays the same, and bits b[i-1], ..., b[0] are all 0's.

Now, AND-ing x with -x makes bits b[i+1], ..., b[n] all 0, leaves bit b[i] as is, and sets bits b[i-1], ..., b[0] to 0. Only one bit is left, it's the bit b[i] - the rightmost 1-bit.

2. There is no rightmost 1-bit. The value is 0. The negative of 0 in two's complement is also 0. 0&0 = 0. No bits get turned on.

We have proved rigorously that this bithack is correct.

Bit Hack #8. Right propagate the rightmost 1-bit.

```
y = x | (x-1)
```

This is best understood by an example. Given a value 01010000 it turns it into 01011111. All the 0-bits right to the rightmost 1-bit got turned into ones.

This is not a clean hack, tho, as it produces all 1's if x = 0.

Let's look at more examples:

```
    10111100  (x)
|   10111011  (x-1)
    --------
    10111111

    01110111  (x)
|   01110110  (x-1)
    --------
    01110111

    00000001  (x)
|   00000000  (x-1)
    --------
    00000001

    10000000  (x = -128)
|   01111111  (x-1 = 127)
    --------
    11111111

    11111111  (x = -1)
|   11111110  (x-1 = -2)
    --------
    11111111

    00000000  (x)
|   11111111  (x-1)
    --------
    11111111
```

Let's prove it, though not as rigorously as in the previous bithack (as it's too time consuming and this is not a scientific publication). There are two cases again. Let's start with easiest first.

1. There is no rightmost 1-bit. In that case x = 0 and x-1 is -1. -1 in two's complement is 11111111. OR-ing 0 with 11111111 produces the same 11111111. (Not the desired result, but that's the way it is.)
2. There is the rightmost 1-bit b[i]. Let's divide all the bits in two groups again (like in the previous example). Calculating x-1 modifies only bits to the right, turning b[i] into 0, and all the lower bits to 1's. Now OR-ing x with x-1 leaves all the higher bits (to the left) the same, leaves bit b[i] as it was 1, and since lower bits are all low 1's it also turns them on. The result is that the rightmost 1-bit got propagated to lower order bits.

Bit Hack #9. Isolate the rightmost 0-bit.

```
y = �x & (x+1)
```

This bithack does the opposite of #7. It finds the rightmost 0-bit, turns off all bits, and sets this bit to 1 in the result. For example, it finds the zero in bold in this number 10101011, producing 00000100.

More examples:

```
    10111100  (x)
    --------
    01000011  (�x)
&   10111101  (x+1)
    --------
    00000001

    01110111  (x)
    --------
    10001000  (�x)
&   01111000  (x+1)
    --------
    00001000

    00000001  (x)
    --------
    11111110  (�x)
&   00000010  (x+1)
    --------
    00000010

    10000000  (x = -128)
    --------
    01111111  (�x)
&   10000001  (x+1)
    --------
    00000001

    11111111  (x = no rightmost 0-bit)
    --------
    00000000  (�x)
&   00000000  (x+1)
    --------
    00000000

    00000000  (x)
    --------
    11111111  (�x)
&   00000001  (x+1)
    --------
    00000001
```

Proof: Suppose there is a rightmost 0-bit. Then �x turns this rightmost 0 bit into 1 bit. And so does x+1 (because bits more right to the rightmost 0 bit are 1's). Now AND-ing �x with x+1 evaporates all the bits up to this rightmost 0 bit. This is the highest order bit set in the result. Now what about lower order bits to the right of rightmost 0 bit? They also got evaporated because because x+1 turned them into 0's (they were 1's) and �x turned them into 0's. They got AND-ed with 0 and evaporated.

Bit Hack #10. Turn on the rightmost 0-bit.

```
y = x | (x+1)
```

This hack changes the rightmost 0-bit into 1. For example, given an integer 10100011 it turns it into 10100111.

More examples:

```
    10111100  (x)
|   10111101  (x+1)
    --------
    10111101

    01110111  (x)
|   01111000  (x+1)
    --------
    01111111

    00000001  (x)
|   00000010  (x+1)
    --------
    00000011

    10000000  (x = -128)
|   10000001  (x+1)
    --------
    10000001

    11111111  (x = no rightmost 0-bit)
|   00000000  (x+1)
    --------
    11111111

    00000000  (x)
|   00000001  (x+1)
    --------
    00000001
```

Here is the proof as a bunch of true statements. OR-ing x with x+1 does not lose any information. Adding 1 to x fills the first rightmost 0. The result is max{x, x+1}. If x+1 overflows it's x and there were no 0 bits. If it doesn't, it's x+1 which just got rightmost bit filled with 1.

Hacker's Delight

There's a book 300 pages long entirely about bit hacks like these. It's called [http://www.amazon.com/gp/product/0201914654]Hacker's Delight.
Take a look. If you liked the contents of my post, then you'll love this book.

Bonus stuff

If you decide to play more with these hacks, here are a few utility functions to print binary values of 8 bit signed integers in Perl, Python and C.

Print binary representation in Perl:

```
sub int_to_bin {
    my $num = shift;
    print unpack "B8", pack "c", $num;
}
```

Or you can print it from command line right away:

```
perl -wle 'print unpack "B8", pack "c", shift' <integer>

# For example:
perl -wle &#39;print unpack &#34;B8&#34;, pack &#34;c&#34;, shift&#39; 113
01110001

perl -wle &#39;print unpack &#34;B8&#34;, pack &#34;c&#34;, shift&#39; &#45;&#45; -128
10000000
```

Print binary number in Python:

```
def int_to_bin(num, bits=8):
 r = ''
 while bits:
    r = ('1' if num&1 else '0') + r
    bits = bits - 1
    num = num >> 1
 print r
```

Print binary representation in C:

```
void int_to_bin(int num) {
    char str[9] = {0};
    int i;
    for (i=7; i>=0; i--) {
        str[i] = (num&1)?'1':'0';
        num >>= 1;
    }
    printf("%s\n", str);
}
```

---
https://codeforwin.org/c-programming/10-cool-bitwise-operator-hacks-and-tricks

10 cool bitwise operator hacks and tricks every programmer must know
July 20, 2025

Bitwise operators are used to manipulate data at its lowest level (bit level). Data in memory (RAM) is organized as a sequence of bytes. Each byte is a group of eight consecutive bits. We use bitwise operators whenever we need to manipulate bits directly. In this post I will show you some cool bitwise operator hacks and tricks. These hacks will boost your programming skill.

Bitwise operator hacks and tricks

* [https://codeforwin.org/2017/08/bitwise-operators-c.html]Bitwise operators in C programming.
* [https://codeforwin.org/2016/01/bitwise-operator-programming-exercises-and-solutions-in-c.html]Bitwise operator programming exercises.

Quick overview of Bitwise operators

1. Bitwise AND (&) operator compare two bits and return 1 if both bits are set (1), otherwise

```
       return 0.
    2. Bitwise OR (|) operator compare two bits and return 1 if any of them or both bits are set (1),
       otherwise return 0.
    3. Bitwise XOR (^) operator compare two bits and return 1 if either of the bits are set (1),
       otherwise return 0.
    4. Bitwise complement (￼) operator takes single operand and invert all the bits of the operand.
    5. Bitwise right shift (>>) operator insert 0 bit at most significant bit and shift subsequent
       bits to right.
    6. Bitwise Left shift (<<) operator insert 0 bit at least significant bit and shift subsequent
       bits to left.

   Let us get started and learn some cool bitwise operator hacks and tricks.

Bitwise operator hacks and tricks
    1. Right shift (>>) operator is equivalent to division by 2
       Want to divide a number by 2 quicky. Here you go, use bitwise right shift operator to divide
       an integer by 2. Each right shift operation reduces the number (operand) to its half.
       Example:
#include <stdio.h>

int main() {
    int a = 24;

    // Use bitwise right shift to divide
    // number by power of 2
    printf("24 / (2^1) => %d\n", (a >> 1));
    printf("24 / (2^2) => %d\n", (a >> 2));
    printf("24 / (2^3) => %d\n", (a >> 3));

    return 0;
}

       Output:
24 / (2^1) => 12
24 / (2^2) => 6
24 / (2^3) => 3

    2. Left shift (<<) operator is equivalent to multiplication by 2
       Similar to division, you can use bitwise left shift operator to quickly multiply a number by the
       power of 2. Each left shift makes doubles the number (operand).
       Example:

#include <stdio.h>

int main() {
    int a = 12;

    // Use bitwise left shift to multiply
    // number by power of 2
    printf("12 * (2^1) => %d\n", (a << 1));
    printf("12 * (2^2) => %d\n", (a << 2));
    printf("12 * (2^3) => %d\n", (a << 3));

    return 0;
}

       Output:
12 * (2^1) => 24
12 * (2^2) => 48
12 * (2^3) => 96

    3. Use bitwise AND (&) operator to check even or odd number
       To check even or odd number we generally use modulo division operator. You can use
       bitwise AND & operator to check whether a number is even or odd.
       You can also use this trick to check if a number is divisible by two or not.

     Read more how to use bitwise AND operator to check even or odd.
       Example:
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 21;

    // Check even odd
    if (num1 & 1)
        printf("%d is an ODD number.\n", num1);
    else
        printf("%d is an EVEN number.\n", num1);

    if(num2 & 1)
        printf("%d is an ODD number.\n", num2);
    else
        printf("%d is an EVEN number.\n", num2);

    return 0;
}
```

```
       Output:
10 is an EVEN number.
21 is an ODD number.
```

    4. Store multiple flags in single variable
       We often use variable to store boolean flag values e.g. isEven, isMarried, isPrime etc.
       Instead of wasting 4 byte to store single flag. You can use bit masking to store multiple flag
       values in single variable. A 4 byte unsigned integer can store 32 flags.
       We use bitwise OR | operator to set flag. To unset or check flag status we use bitwise AND &
       operator. At a high level it is known as bit masking, but you can think it as set, unset and
       check a bit status.

       Example:
       In below example I will set, check and reset three flag values. Flag for marital status at 0th
       bit, voting status at 1st bit, VISA status at 2nd bit.

```
#include <stdio.h>

int main() {
    // Make all bits off.
    unsigned char flag = 0;

    // Set marital status YES, i.e. 0th bit 1
    // (flag => 0000 0001 = 1)
    flag = flag | 1;

    // Set voting status YES, i.e. 1st bit 1
    // (flag => 0000 0011 = 3)
    flag = flag | 2;

    // Set VISA eligibility status YES, i.e. 2nd bit 1
    // (flag => 0000 0111 = 7)
    flag = flag | 4;

    // Print flag value
    printf("flag, DECIMAL = %d, HEX = %x\n\n", flag, flag);

    // Check if married
    if(flag & 1)
        printf("You are married.\n");
    else
        printf("You are not married.\n");

    // Check voting eligibility
    if(flag & 2)
        printf("You are eligible for voting.\n");
    else
        printf("You are not eligible for voting.\n");

    // Check VISA status
    if(flag & 4)
        printf("You are eligible to get VISA.\n");
    else
        printf("You are not eligible to get VISA.\n");


    // Unset or set all flags to false.
    flag = flag & (⍰(1 << 0));
    flag = flag & (⍰(1 << 1));
    flag = flag & (⍰(1 << 2));

    // Print flag value
    printf("\nflag, DECIMAL = %d, HEX = %x\n", flag, flag);

    return 0;
}
```

```
       Output:
flag, DECIMAL = 7, HEX = 7

You are married.
You are eligible for voting.
You are eligible to get VISA.

flag, DECIMAL = 0, HEX = 0
```

    5. Quickly find 1s and 2s complement of a number
       One's complement of a binary number is defined as value obtained after inverting all bits of
       the number. We use bitwise complement operator ⍰ operator, to find 1s complement of a number.
       You can get two's complement of a binary number by adding 1 to its one's complement.
       Example:

```
#include <stdio.h>

int main() {
    int num = 8;

    // ⍰num yields 1s complement of num
    printf("1s complement of %d = %d\n", num, (⍰num));

    // (⍰num + 1) yields 2s complement of num
    printf("2s complement of %d = %d\n", num, (⍰num + 1));

    return 0;
}
```

Output:

```
1s complement of 8 = -9
2s complement of 8 = -8
```

6. Quickly convert character to lowercase and uppercase
   This is my favourite hack. You can use bitwise OR and AND operator to convert a character to lowercase and uppercase respectively.
   + [https://codeforwin.org/2015/04/c-program-convert-upper-case-string-to-lower.html]How to convert string to lowercase.
   + [https://codeforwin.org/2015/04/c-program-convert-lower-case-string-to-upper.html]How to convert string to uppercase.

   To convert a character ch to lowercase use ch = ch | ' '. Whether ch is uppercase or lowercase. Result of this is always a lowercase character.
   To convert a character ch to uppercase use ch = ch & '_'. It always return uppercase character, doesn't matter whether ch is uppercase or lowercase.

   Example:

```
#include <stdio.h>

int main() {
    // Convert to lowercase
    printf("'a' => '%c'\n", ('a' | ' '));
    printf("'A' => '%c'\n", ('A' | ' '));

    // Convert to uppercase
    printf("'a' => '%c'\n", ('a' & '_'));
    printf("'A' => '%c'\n", ('a' & '_'));

    return 0;
}
```

   Output:

```
'a' => 'a'
'A' => 'a'
'a' => 'A'
'A' => 'A'
```

7. Quick conditional assignment hack
   This is one of my favourite bitwise XOR ^ hack. In programming you may require some conditional assignment such as,

```
if (x == a)
    x = b;
if (x == b)
    x = a;
```

   You can use bitwise XOR operator for these type of assignment.

   Example:

```
#include <stdio.h>

int main() {
    int a = 10, b = 20, x;

    // Original value
    x = a;
    printf("x = %d\n", x);

    // if (x == a) x = b;
    x = a ^ b ^ x;
    printf("x = %d\n", x);

    // if (x == b) x = a;
    x = a ^ b ^ x;
    printf("x = %d\n", x);

    // x = 0
    x = x ^ x;
    printf("x = %d\n", x);

    return 0;
}
```

   Output:

```
x = 10
x = 20
x = 10
x = 0
```

8. Find maximum or minimum without if...else
   Another hack frequently asked in interviews. We all know to find maximum or minimum using if else. Let's do it bitwise way.

   Example:

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;

    int min = (y ^ (x ^ y) & -(x < y));
    int max = (x ^ (x ^ y) & -(x < y));

    printf("Minimum(10, 20) => %d\n", min);
    printf("Maximum(10, 20) => %d\n", max);

    return 0;
}
```

   Output:

```
Maximum(10, 20) => 20
Minimum(10, 20) => 10
```

9. Use bitwise XOR (^) operator to quickly swap two number without third variable
   Frequently asked in interviews, how to swap two numbers without using third variable. You can use bitwise XOR ^ operator to swap two variables without using third variable.

   Example:

```
#include <stdio.h>


int main() {
    int a, b;

    // Input two numbers
    printf("Enter two numbers to swap: ");
    scanf("%d%d", &a, &b);

    // Print original values.
    printf("Original value: a=%d, b=%d\n", a, b);

    // Swap a with b
    a ^= b;
    b ^= a;
    a ^= b;

    // Swapped values.
    printf("Swapped value: a=%d, b=%d\n", a, b);

    return 0;
}
```

   Output:

```
Enter two numbers to swap: 10 20
Original value: a=10, b=20
Swapped value: a=20, b=10
```

10. Use bitwise XOR (^) operator for basic encryption and decryption
    Bitwise XOR operator is one of the magical operators in C. It has a special property, suppose a and b two integers and c = a ^ b. Then, the result of a ^ b i.e. c, when XORed with a return b and vice versa.

    For example:

```
int a, b, c;
a = 10, b=20;

c = a ^ b; // c = 30
printf("%d", (c ^ a)); // 20
printf("%d", (c ^ b)); // 10
```

    We can use this feature of XOR operator for basic encryption/decryption.

    Example:

```
#include <stdio.h>

#define KEY 22

int main() {
    char text[100];
    int i;
    // Input text
    printf("Enter text to encrypt: ");
    fgets(text, 100, stdin);

    // Encrypt text
    for (i=0; text[i] != '\0'; i++)
    {
        text[i] = text[i] ^ KEY;
    }

    printf("Encrypted text: %s\n", text);

    // Decrypt text
    for (i = 0; text[i] != '\0'; i++)
    {
        text[i] = text[i] ^ KEY;
    }

    printf("Original text: %s\n", text);

    return 0;
}
```

Output:

```
Enter text to encrypt: I love C programming.
Encrypted text: _6zy`s6U6fdyqdw{{xq8
Original text: I love C programming.
```

---
https://stackoverflow.com/questions/1533131/what-useful-bitwise-operator-code-tricks-should-a-developer-know-ab
out

What USEFUL bitwise operator code tricks should a developer know about? [closed]

I must say I have never had cause to use bitwise operators, but I am sure there are some operations that I have performed that would have been more efficiently done with them. How have "shifting" and "OR-ing" helped you solve a problem more efficiently?

***
Using bitwise operations on strings (characters)

Convert letter to lowercase:
* OR by space => (x | ' ')
* Result is always lowercase even if letter is already lowercase
* eg. ('a' | ' ') => 'a' ; ('A' | ' ') => 'a'

Convert letter to uppercase:
* AND by underline => (x & '_')
* Result is always uppercase even if letter is already uppercase
* eg. ('a' & '_') => 'A' ; ('A' & '_') => 'A'

Invert letter's case:
* XOR by space => (x ^ ' ')
* eg. ('a' ^ ' ') => 'A' ; ('A' ^ ' ') => 'a'

Letter's position in alphabet:
* AND by chr(31)/binary('11111')/(hex('1F') => (x & "\x1F")
* Result is in 1..26 range, letter case is not important
* eg. ('a' & "\x1F") => 1 ; ('B' & "\x1F") => 2

Get letter's position in alphabet (for Uppercase letters only):
* AND by ? => (x & '?') or XOR by @ => (x ^ '@')
* eg. ('C' & '?') => 3 ; ('Z' ^ '@') => 26

Get letter's position in alphabet (for lowercase letters only):
* XOR by backtick/chr(96)/binary('1100000')/hex('60') => (x ^ '`')
* eg. ('d' ^ '`') => 4 ; ('x' ^ '`') => 25

Note: using anything other than the english letters will produce garbage results

***
* Bitwise operations on integers(int)

Get the maximum integer

```
int maxInt = ⍰(1 << 31);
int maxInt = (1 << 31) - 1;
int maxInt = (1 << -1) - 1;
```

Get the minimum integer

```
int minInt = 1 << 31;
int minInt = 1 << -1;
```

Get the maximum long

```
long maxLong = ((long)1 << 127) - 1;
```

```
   Multiplied by 2
n << 1; // n*2

   Divided by 2
n >> 1; // n/2

   Multiplied by the m-th power of 2
n << m; // (3<<5) ==>3 * 2^5 ==> 96

   Divided by the m-th power of 2
n >> m; // (20>>2) ==>(20/( 2^2) ==> 5

   Check odd number
(n & 1) == 1;

   Exchange two values
a ^= b;
b ^= a;
a ^= b;

   Get absolute value
(n ^ (n >> 31)) - (n >> 31);

   Get the max of two values
b & ((a-b) >> 31) | a & (⍰(a-b) >> 31);

   Get the min of two values
a & ((a-b) >> 31) | b & (⍰(a-b) >> 31);

   Check whether both have the same sign
(x ^ y) >= 0;

   Calculate 2^n
2 << (n-1);

   Whether is factorial of 2
n > 0 ? (n & (n - 1)) == 0 : false;

   Modulo 2^n against m
m & (n - 1);

   Get the average
(x + y) >> 1;
((x ^ y) >> 1) + (x & y);

   Get the m-th bit of n (from low to high)
(n >> (m-1)) & 1;

   Set the m-th bit of n to 0 (from low to high)
n & ⍰(1 << (m-1));

   n + 1
-⍰n

   n - 1
⍰-n

   Get the contrast number
⍰n + 1;
(n ^ -1) + 1;

   if (x==a) x=b; if (x==b) x=a;
x = a ^ b ^ x;

***
   See the famous [http://graphics.stanford.edu/⍰seander/bithacks.html]Bit Twiddling Hacks
   Most of the multiply/divide ones are unnecessary - the compiler will do that automatically and you
   will just confuse people.

   But there are a bunch of, 'check/set/toggle bit N' type hacks that are very useful if you work with
   hardware or communications protocols.

***
   There's only three that I've ever used with any frequency:
    1. Set a bit: a |= 1 << bit;
    2. Clear a bit: a &= ⍰(1 << bit);
    3. Test that a bit is set: a & (1 << bit);

***
     Average without overflow
     A routine for the computation of the average (x + y)/2 of two arguments x and y is

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);
}
```

***

   1) Divide/Multiply by a power of 2
   foo >>= x; (divide by power of 2)
   foo <<= x; (multiply by power of 2)

   2) Swap

```
x ^= y;
y = x ^ y;
x ^= y;
```

***

   You can compress data, e.g. a collection of integers:
     * See which integer values appear more frequently in the collection
     * [http://en.wikipedia.org/wiki/Huffman_coding]Use short bit-sequences to represent the values which appear more frequently (and longer bit-sequences to represent the values which appear less frequently)
     * Concatenate the bits-sequences: so for example, the first 3 bits in the resulting bit stream might represent one integer, then the next 9 bits another integer, etc.

***

   I wanted a function to round numbers to the next highest power of two, so I visited the Bit Twiddling website that's been brought up several times and came up with this:

```
i--;
i |= i >> 1;
i |= i >> 2;
i |= i >> 4;
i |= i >> 8;
i |= i >> 16;
i++;
```

   I use it on a size_t type. It probably won't play well on signed types. If you're worried about portability to platforms with different sized types, sprinkle your code with #if SIZE_MAX >= (number) directives in appropriate places.

---
https://medium.com/@r.siddhesh96/10-bitwise-tricks-you-should-know-with-examples-in-c-4b04820dfbc0

10 Bitwise Tricks You Should Know (with examples in C++)
Mar 7, 2025

   Bitwise operations are one of those things in programming that sound super fancy but are actually pretty cool once you get the hang of them. They let you mess around with numbers at the binary level (you know, 0s and 1s), and sometimes they can make your code way faster or just plain clever. Honestly, I used to ignore them because they seemed complicated, but turns out, they're not that bad. So, here's a lazy little collection of some bitwise tricks I've come across - nothing too serious, just some fun and useful stuff to know!!
   Don't have Medium account? - Use this link
   https://medium.com/@r.siddhesh96/10-bitwise-tricks-you-should-know-with-examples-in-c-4b04820dfbc0?sk=c66c754e5fee726d095d6fbb5d6a9ee6

1. Turning Uppercase Letters to Lowercase (or Vice Versa)
   One of the coolest tricks with bitwise operations is converting uppercase letters to lowercase (or vice versa). This works because of how characters are represented in ASCII.
     * Uppercase letters ('A'-'Z') have ASCII values from 65 to 90.
     * Lowercase letters ('a'-'z') have ASCII values from 97 to 122.
     * The difference between uppercase and lowercase is a single bit: the 6th bit (counting from 0).

   To convert an uppercase letter to lowercase, you can set the 6th bit using the bitwise OR (|) operator:

```
char toLower(char c) {
    return c | (1 << 5);
}
```

---
https://www.hackerearth.com/practice/basic-programming/bit-manipulation/basics-of-bit-manipulation/tutorial/

Basics of Bit Manipulation

   Working on bytes, or data types comprising of bytes like ints, floats, doubles or even data structures which stores large amount of bytes is normal for a programmer. In some cases, a programmer needs to go beyond this - that is to say that in a deeper level where the importance of bits is realized.

   Operations with bits are used in Data compression (data is compressed by converting it from one representation to another, to reduce the space) ,Exclusive-Or Encryption (an algorithm to encrypt the data for safety issues). In order to encode, decode or compress files we have to extract the data at bit level. Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

We all know that 1 byte comprises of 8 bits and any integer or character can be represented using bits in computers, which we call its binary form(contains only 1 or 0) or in its base 2 form.

```
Example:
1) 14 = {1110 }[2]
= 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0
= 14.

2) 20 = {10100 }[2]
= 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0
= 20.
```

For characters, we use ASCII representation, which are in the form of integers which again can be represented using bits as explained above.

Bitwise Operators:
There are different bitwise operations used in the bit manipulation. These bit operations operate on the individual bits of the bit patterns. Bit operations are fast and can be used in optimizing time complexity. Some common bit operators are:

NOT (⍰): Bitwise NOT is an unary operator that flips the bits of the number i.e., if the ith bit is 0, it will change it to 1 and vice versa. Bitwise NOT is nothing but simply the one's complement of a number. Lets take an example.

```
N = 5 = (101)[2]
⍰N = ⍰5 = ⍰(101)[2] = (010)[2] = 2
```

AND (&): Bitwise AND is a binary operator that operates on two equal-length bit patterns. If both bits in the compared position of the bit patterns are 1, the bit in the resulting bit pattern is 1, otherwise 0.

```
A = 5 = (101)[2] , B = 3 = (011)[2] A & B = (101)[2] & (011)[2]= (001)[2] = 1
```

OR (|): Bitwise OR is also a binary operator that operates on two equal-length bit patterns, similar to bitwise AND. If both bits in the compared position of the bit patterns are 0, the bit in the resulting bit pattern is 0, otherwise 1.

```
A = 5 = (101)[2] , B = 3 = (011)[2]
A | B = (101)[2] | (011)[2] = (111)[2] = 7
```

XOR (^): Bitwise XOR also takes two equal-length bit patterns. If both bits in the compared position of the bit patterns are 0 or 1, the bit in the resulting bit pattern is 0, otherwise 1.

```
A = 5 = (101)[2] , B = 3 = (011)[2]
A ^ B = (101)[2] ^ (011)[2] = (110)[2] = 6
```

Left Shift (<<): Left shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the left and append 0 at the end. Left shift is equivalent to multiplying the bit pattern with $$2^k$$ (if we are shifting k bits).

```
1 << 1 = 2 = 2^1
1 << 2 = 4 = 2^2 1 << 3 = 8 = 2^3
1 << 4 = 16 = 2^4
...
1 << n = 2^n
```

Right Shift (>>): Right shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the right and append 1 at the end. Right shift is equivalent to dividing the bit pattern with 2k (if we are shifting k bits).

```
4 >> 1 = 2
6 >> 1 = 3
5 >> 1 = 2
16 >> 4 = 1
```

Bitwise operators are good for saving space and sometimes to cleverly remove dependencies.

Note: All left and right side taken in this article, are taken with reference to the reader.

Lets discuss some algorithms based on bitwise operations:
1) How to check if a given number is a power of 2 ?
Consider a number N and you need to find if N is a power of 2. Simple solution to this problem is to repeated divide N by 2 if N is even. If we end up with a 1 then N is power of 2, otherwise not. There are a special case also. If N = 0 then it is not a power of 2. Let's code it.

Implementation:

```
bool isPowerOfTwo(int x) {
    if(x == 0) {
        return false;
    } else {
        while(x % 2 == 0) x /= 2;
        return (x == 1);
    }
}
```

Above function will return true if x is a power of 2, otherwise false.
Time complexity of the above code is O(logN).

The same problem can be solved using bit manipulation. Consider a number x that we need to check for being a power for 2. Now think about the binary representation of (x-1). (x-1) will have all the bits same as x, except for the rightmost 1 in x and all the bits to the right of the rightmost 1.
Let, x = 4 = (100)[2]

```
   x - 1 = 3 = (011)[2]
   Let, x = 6 = (110)[2]
   x - 1 = 5 = (101)[2]
```

It might not seem obvious with these examples, but binary representation of (x-1) can be obtained by simply flipping all the bits to the right of rightmost 1 in x and also including the rightmost 1.

Now think about x & (x-1). x & (x-1) will have all the bits equal to the x except for the rightmost 1 in x.

```
   Let, x = 4 = (100)[2]
   x - 1 = 3 = (011)[2]
   x & (x-1) = 4 & 3 = (100)[2] & (011)[2] = (000)[2]
   Let, x = 6 = (110)[2]
   x - 1 = 5 = (101)[2]
   x & (x-1) = 6 & 5 = (110)[2] & (101)[2] = (100)[2]
```

Properties for numbers which are powers of 2, is that they have one and only one bit set in their binary representation. If the number is neither zero nor a power of two, it will have 1 in more than one place. So if x is a power of 2 then x & (x-1) will be 0.

Implementation:

```
bool isPowerOfTwo(int x) {
    // x will check if x == 0 and !(x & (x - 1)) will check if x is a power of 2 or not
    return (x && !(x & (x - 1)));
}
```

2) Count the number of ones in the binary representation of the given number.
The basic approach to evaluate the binary form of a number is to traverse on it and count the number of ones. But this approach takes log[2]N of time in every case.

Why log[2]N ?
As to get a number in its binary form, we have to divide it by 2, until it gets 0, which will take log[2]N of time.

With bitwise operations, we can use an algorithm whose running time depends on the number of ones present in the binary form of the given number. This algorithm is much better, as it will reach to logN, only in its worst case.

```
int count_one (int n) {
    while( n ) {
        n = n&(n-1);
        count++;
    }
    return count;
}
```

Why this algorithm works ?
As explained in the previous algorithm, the relationship between the bits of x and x-1. So as in x-1, the rightmost 1 and bits right to it are flipped, then by performing x&(x-1), and storing it in x, will reduce x to a number containing number of ones(in its binary form) less than the previous state of x, thus increasing the value of count in each iteration.

Example:
n = 23 = {10111}[2] .
1. Initially, count = 0.

2. Now, n will change to n&(n-1). As n-1 = 22 = {10110}[2] , then n&(n-1) will be {10111[2] & {10110}[2], which will be {10110}[2] which is equal to 22. Therefore n will change to 22 and count to 1.

3. As n-1 = 21 = {10101}[2] , then n&(n-1) will be {10110}[2] & {10101}[2], which will be {10100}[2] which is equal to 20. Therefore n will change to 20 and count to 2.

4. As n-1 = 19 = {10011}[2] , then n&(n-1) will be {10100}[2] & {10011}[2], which will be {10000}[2] which is equal to 16. Therefore n will change to 16 and count to 3.

5. As n-1 = 15 = {01111}[2] , then n&(n-1) will be {10000}[2] & {01111}[2], which will be {00000}[2] which is equal to 0. Therefore n will change to 0 and count to 4.

6. As n = 0, the the loop will terminate and gives the result as 4.

Complexity: O(K), where K is the number of ones present in the binary form of the given number.

3) Check if the i^th bit is set in the binary form of the given number.
To check if the i^th bit is set or not (1 or not), we can use AND operator. How?

Let's say we have a number N, and to check whether it's i^th bit is set or not, we can AND it with the number 2^i . The binary form of 2^i contains only i^th bit as set (or 1), else every bit is 0 there. When we will AND it with N, and if the i^th bit of N is set, then it will return a non zero number (2^i to be specific), else 0 will be returned.

Using Left shift operator, we can write 2^i as 1 << i . Therefore:

```
bool check (int N) {
    if( N & (1 << i) )
        return true;
    else
```

```
        return false;
}
```

Example:
Let's say N = 20 = {10100}[2]. Now let's check if it's 2nd bit is set or not(starting from 0). For that, we have to AND it with 2^2 = 1<<2 = {100}[2] .
{10100} & {100} = {100} = 2^2 = 4(non-zero number), which means it's 2nd bit is set.

4) How to generate all the possible subsets of a set ?
A big advantage of bit manipulation is that it can help to iterate over all the subsets of an N-element set. As we all know there are 2^N possible subsets of any given set with N elements. What if we represent each element in a subset with a bit. A bit can be either 0 or 1, thus we can use this to denote whether the corresponding element belongs to this given subset or not. So each bit pattern will represent a subset.

Consider a set A of 3 elements.
A = {a, b, c}

Now, we need 3 bits, one bit for each element. 1 represent that the corresponding element is present in the subset, whereas 0 represent the corresponding element is not in the subset. Let's write all the possible combination of these 3 bits.

```
0 = (000)[2] = {}
1 = (001)[2] = {c}
2 = (010)[2] = {b}
3 = (011)[2] = {b, c}
4 = (100)[2] = {a}
5 = (101)[2] = {a, c}
6 = (110)[2] = {a, b}
7 = (111)[2] = {a, b, c}
```

Pseudo Code:

```
possibleSubsets(A, N):
        for i = 0 to 2^N:
            for j = 0 to N:
                if jth bit is set in i:
                    print A[j]
            print '\n'
```

Implementation:

```
void possibleSubsets(char A[], int N) {
    for(int i = 0;i < (1 << N); ++i) {
        for(int j = 0;j < N;++j) {
            if(i & (1 << j)) {
                cout << A[j] << ' ';
            }
        }
        cout << endl;
    }
}
```

5) Find the largest power of 2 (most significant bit in binary form), which is less than or equal to the given number N.

Idea: Change all the bits which are at the right side of the most significant digit, to 1.

Property: As we know that when all the bits of a number N are 1, then N must be equal to the 2^i -1 , where i is the number of bits in N.

Example:
Let's say binary form of a N is {1111}[2] which is equal to 15.
15 = 2^4-1, where 4 is the number of bits in N.

This property can be used to find the largest power of 2 less than or equal to N. How?
If we somehow, change all the bits which are at right side of the most significant bit of N to 1, then the number will become x + (x-1) = 2 * x -1 , where x is the required answer.

Example:
Let's say N = 21 = {10101}, here most significant bit is the 4th one. (counting from 0th digit) and so the answer should be 16.

So lets change all the right side bits of the most significant bit to 1. Now the number changes to {11111} = 31 = 2 * 16 -1 = Y (let's say).
Now the required answer is (Y+1)>>1 or (Y+1)/2.

Now the question arises here is how can we change all right side bits of most significant bit to 1?

Let's take the N as 16 bit integer and binary form of N is {1000000000000000}.
Here we have to change all the right side bits to 1.

Initially we will copy that most significant bit to its adjacent right side by:

N = N | (N>>1).

As you can see, in above diagram, after performing the operation, rightmost bit has been copied to

```
   its adjacent place.

   Now we will copy the 2 rightmost set bits to their adjacent right side.
   N = N | (N>>2).

   Now we will copy the 4 rightmost set bit to their adjacent right side.

   N = N | (N>>4)

   Now we will copy these 8 rightmost set bits to their adjacent right side.

   N = N| (N>>8)

   Now all the right side bits of the most significant set bit has been changed to 1 .This is how we can
   change right side bits. This explanation is for 16 bit integer, and it can be extended for 32 or 64
   bit integer too.

   Implementation:
long largest_power(long N) {
    //changing all right side bits to 1.
    N = N| (N>>1);
    N = N| (N>>2);
    N = N| (N>>4);
    N = N| (N>>8);

    //as now the number is 2 * x-1, where x is required answer, so adding 1 and dividing it by 2.
    return (N+1)>>1;
}

   Tricks with Bits:
   1) x ^ ( x & (x-1)) : Returns the rightmost 1 in binary representation of x.

   As explained above, (x & (x - 1)) will have all the bits equal to the x except for the rightmost 1 in
   x. So if we do bitwise XOR of x and (x & (x-1)), it will simply return the rightmost 1. Let's see an
   example.
   x = 10 = (1010)[2] ` x & (x-1) = (1010)[2] & (1001)[2] = (1000)[2]
   x ^ (x & (x-1)) = (1010)[2] ^ (1000)[2] = (0010)[2]

   2) x & (-x) : Returns the rightmost 1 in binary representation of x

   (-x) is the two's complement of x. (-x) will be equal to one's complement of x plus 1.
   Therefore (-x) will have all the bits flipped that are on the left of the rightmost 1 in x. So x &
   (-x) will return rightmost 1.

   x = 10 = (1010)[2]
   (-x) = -10 = (0110)[2]
   x & (-x) = (1010)[2] & (0110)[2] = (0010)[2]

   3) x | (1 << n) : Returns the number x with the nth bit set.
   (1 << n) will return a number with only nth bit set. So if we OR it with x it will set the nth bit of
   x.
   x = 10 = (1010)[2] n = 2
   1 << n = (0100)[2]
   x | (1 << n) = (1010)[2] | (0100)[2] = (1110)[2]

   Applications of bit operations:
   1) They are widely used in areas of graphics ,specially XOR(Exclusive OR) operations.
   2) They are widely used in the embedded systems, in situations, where we need to set/clear/toggle
   just one single bit of a specific register without modifying the other contents. We can do OR/AND/XOR
   operations with the appropriate mask for the bit position.
   3) Data structure like n-bit map can be used to allocate n-size resource pool to represent the
   current status.
   4) Bits are used in networking, framing the packets of numerous bits which is sent to another system
   generally through any type of serial interface.

   To see a creative use of Bitwise operators, you can refer this amazing article , where the bit
   wise operations are smartly used while developing an online calendar for events.


---
https://github.com/eashish93/C-Notes/blob/master/Bitwise%20Twiddling%20Tricks.c
C-Notes/Bitwise Twiddling Tricks.c

/**
 * BIT TWIDDLING TRICKS
 * =============================================
 */

/**
 10             - 2
 100            - 4
 1000           - 8
 10000          - 16
 100000         - 32
 1000000        - 64
 10000000       - 128
 etc...
**/


/** Extract/Test the first bit from the integer **/
    // Example:
    int k = 13;
    printf("%d", k & 1);    // This will extract the 0th bit from the k
    // Explaination
    /**
     * 13 is 1101
     *       0001
     *      ------
     *       0001       // This will extract the 0th bit
     */


/** Determine if number is even or odd **/
    // Example:
     if((n & 1) == 0)
        printf("even");
     else
        printf("odd");
    // Explanation
    /**
     * As before, it will extract the first bit, and by this bit we can know that if number is even or odd
     * Because only 0th bit will determine (if it is 1 then odd, else even). Think?
     */


/** Extract/Test the nth bit from the integer **/
    // Example:
     if(x & (1 << n))
        printf("nth-bit is 1");
     else
        printf("nth-bit is 0");
    // Explanation:
    /**
     * 13 is 1101. And we have to check 2nd bit, so 13 & (1 << 2)
     *    1101
     *     100
     *   ------
     *    0100
     *   ------
     *   So, 2nd bit is 1 or we can say 2nd bit is set
     */



/** Set the nth-bit i.e., make it 1 **/
    // Example:
    int y = x | (1 << n)
    // Explanation:
    /**
     * 13 is 1101. And we have to set 2nd bit, so 13 | (1 << 2)
     *   1101
     *    100
     *   -----
     *   1101       // No change, bet 2nd bit already on (or 1)
     *
     *  Now, try for 1st bit, so 13 & (1 << 1)
     *      1101
     *        10
     *      ----
     *      1111    // Now 1st bit is ON
     */
    // If otherwise, we want to set it to zero if it already set to 1. then we XOR(^) operator. eg: 13 ^ (1 <<
n).
    // This is also called toggling of bit (x ^ (1 << n))




/** Unset the nth-bit. i.e., Turns on all the bits except nth-bit **/
    // Example:
     int y = x & ⍰(1<<n)
    // Explanation:
    /**
        01111111    (127 in decimal)
    &   11101111    (⍰(1<<4))
        --------
        01101111
    **/
```

```
/** Turn off the rightmost (i.e., first 1 bit from the right) 1-bit in a word, producing 0 if none **/
    // Formula:
    int y = x & (x-1)
    /** Example
        1011000  (88 in decimal)
    &  (1011000
    -   0000001)
        -------
        1010000
    **/


/** Turn off the rightmost 0-bit in a word **/
    // Formula
    int y = x | (x + 1)
    /** Example
        1101
     +     1
        ----
        1110
     &  1101
        ----
        1100
        ----
        Adding `1` to any word, transform righmost 1's in a zero and first rightmost 0 in a 1.
    **/



/**
 * Basic Theorems
 */

// De-morgan's law (first two)
(X & Y & Z ...)` == X` | Y` | Z` ...   // Eg: �(x & y) = �x | �y
(X | Y | Z ...)` == X` & Y` & Z` ...  // Eg: �(x | y) = �x & �y


�(x + 1) = �x - 1
�(x - 1) = �x + 1
�(x^y) = �x^y
�-x = x - 1
-�x = x + 1
�(x + y) = �x - y
�(x - y) = �x + y
-x = �x + 1 = �(x - 1)
�x = -x - 1
x + y = x - �y - 1
      = (x ^ y) + 2(x & y)
      = (x | y) + (x & y)
      = 2(x | y) - (x ^ y)
x - y = x + �y + 1
      = (x ^ y) - 2(�x & y)
      = (x & �y) - (�x & y)
      = 2(x & �y) - (x ^ y)
x^y = (x | y) - (x & y)



---
https://github.com/eashish93/C-Notes/blob/master/Bitwise%20operators.c
C-Notes/Bitwise operators.c

// Bitwise Operations
// For More: 1) http://graphics.stanford.edu/�seander/bithacks.html
//           2) http://www.catonmat.net/blog/low-level-bit-hacks-you-absolutely-must-know/

    <<          left shift  (equivalent to multiplication by 2)
    >>          right shift (equivalent to division by 2)

    /*
        Example:
        unsigned short i, j;
        i = 13          (binary 0000000000001101)
        j = i << 2      (binary 0000000000110100)  (equal to 52) (i.e. multiply by 2 two times)
        j = i >> 2      (binary 0000000000000011)  (equal to 3) (i.e. divide by 2 two times 13/4 == 3)
        // Note: Shifting is much faster than actual multiplication or division.
        Note: i >>= 2 is same as i = i >> 2,
              i <<= 2 is same as i = i << 2
     */
    /** WARNING ** -- Beware of shift like this: **/
              a << -5
        // What does it mean? It is undefined. (on some machine it shift left 27 bits). So avoid this

    �       unary operator (bitwise complement)
    &       bitwise and
    ^       bitwise exclusive or (XOR)
    |       bitwise inclusive or (OR) (PIPE)


// `�` operator produces the complement of its operand, with zeros replaced by ones and ones replaced by zeros.
// `&` operator performs boolean `and` operation on all corresponding bits in its two operands.
// `^` and `|` operators are similar (both performs a Boolean `or` operation on the bits in their operands);
// however, `^` produces 0 whenever both operands have a `1` bit, whereas `|` produces 1.

    /*
        Example:
        0 ^ 0 = 0;      0 | 0 = 0;
        0 ^ 1 = 1;      0 | 1 = 1;
        1 ^ 0 = 1;      1 | 0 = 1;
        1 ^ 1 = 0;      1 | 1 = 1;
     */
// Also note that these boolean bitwise operators checks on both operands not on single operands.
// (i.e. in the case of a || b, it checks for one of a or b, but in case of a | b, it checks for both a and b)


/* example */
    unsigned short i, j, k;
    i = 21;     // (binary : 0000000000010101)
    j = 56;     // (binary : 0000000000111000)
    k = �i;     // k is now 65514 (binary : 1111111111101010)
    k = i & j;  // k is now 16    (binary : 0000000000010000)
    k = i ^ j;  // k is now 45    (binary : 0000000000101101)
    k = i | j;  // k is now 61    (binary : 0000000000111101)

Precedence
    Highest   �
              &
              ^
    Lowest    |


/**
 * -------------------- Important tricks ----------------------
 */

/**
 * Sometimes, you see the constant define as
 * 0100, 0200, 0400 -- these are 64, 128, 256 in decimal (by converting from octal to decimal)
 */


---
https://news.ycombinator.com/item?id=35010447

Demystifying bitwise operations, a gentle C tutorial (andreinc.net)

   I suppose the online list of hacks at the 1st reference in the OP  makes more sense, but I've got
   a fondness for the book "Hacker's Delight", which I've owned and browsed for many years now . And
   checking on that, I see there was a 2nd edition issued 10 years ago. Hmm, might need to pick that up.

    https://graphics.stanford.edu/�seander/bithacks.html
   https://en.wikipedia.org/wiki/Hacker's_Delight


   There are quite a lot of bit hacks on the web, but Hackers Delight is where it took off for me.It was
   a massive eye-opener, what was possible and even better doing it the old-fashioned way.

   The second book is very, very heavy on division and as such it's not really as much 'fun' as the
   original, however I'd still recommend it!

   I shared the original Hackers delight with a work colleague, who had a mathematical bent and he
   contacted Henry Warren with a possible addition for the forthcoming second edition, Morton curves
   (https://en.wikipedia.org/wiki/Z-order_curve) which are extremely simple to calculate, much more
   so than the space filling curve given in Hackers Delight, but which despite the author's interested
   response to us, did not go into the second edition. My colleague was very disappointed. Me too.
   Morton curves are just interlace-the-bits so would have slotted in so well.

   Sadly there won't be a third edition as the author died.I found out when I contacted him to let him
   know his website was down (again!). His family let me know he had gone.

***
   The second book is very, very heavy on division and as such it's not really as much 'fun' as the
   original, however I'd still recommend it!

   What do you mean by this? Did they merely add a ton of information on division? Or did they take out
   the 'fun' bits from the original and replace them with division stuff?

***
   It seems to be addition of info. I can't say why but 2nd Ed just feels more like work than 'fun'
   somehow IYSWIM.
```

***
The last time I reached for Hacker's Delight for "productive" use it was specifically for this chapter (I was aware of space-filling curves from some years in the games industry but had never implemented one), and I remember being disappointed at the lack of depth and options compared to the rest of the book.

***
I have the second edition, back around 2010 I made a bit of a splurge on books. Super handy when you need it; which in my case has only been one time. Don't spend much time on low-level code nowadays.

***
Funnily enough just spent a day tracking down a weird problem in an embedded system - some event timestamps were getting corrupted in a weird way. The pattern wasn't obvious until in desperation I dumped them in hex and found the second MSB was toggling between 0 and 1 (whereas it should have been been part of a count sequence). That told me exactly where to look - where the count was re-assembled from four bytes and I found this upstream (paraphrased):-

```
 uint8_t tx_data[64];

 ....
 tx_data[43] = utime>>24;
 tx_data[44] = utime>16;
 tx_data[45] = utime>>8;
 tx_data[46] = utime;
```

***
This is one case where lining up values (and surrounding operators with spaces) can be good practice:

```
 tx_data[43] = utime >> 24;
 tx_data[44] = utime >> 16;
 tx_data[45] = utime >>  8;
 tx_data[46] = utime >>  0;
```

Or even:

```
 tx_data[43] = utime >> (3 * 8);
 tx_data[44] = utime >> (2 * 8);
 tx_data[45] = utime >> (1 * 8);
 tx_data[46] = utime >> (0 * 8);
```

***
Yep. Bugs like this just pop out at you this way.

Having spent a lot of time digging into C code and doing microcontroller programming professionally, I have learned a certain disdain for the practice of using minimal whitespace in expressions, and the related practice of minimising LOC using C's very dense expressions syntax(unary dec/inc operators, the fact that assignment is itself an expression, etc).

The dense expressions syntax is pretty much a holdover from the time C was invented by Dennis Ritchie for the purpose of porting Unix; when it had to be typed on a painfully slow electromechanical teletype. This is also why most Unix commands are 2-4 characters long.

But it serves very little meaningful purpose today. It's important to remember that you might be able to turn 3 lines of code into 1, but it'll still be the same amount of machine code. And it will probably be harder to read, harder to change, and harder to reason about. And it can be a lot easier to miss some edgecase or even a typo like in this case. I have seen such a silly amount of off-by-one errors in C code due to some subtle misuse of unary increment/decrement that I stopped using those operators altogether. They don't improve your software in any meaningful way.

***
I've done the >> 0 thing to make it very clear what's going on, but I hadn't considered the * 8 construction. That's actually easier to comprehend at a glance because the numbers become less "magic". (8 and 16 are obvious to me, but 24 always has me second-guessing myself somewhere in the back of my mind)

***
I use octal for this.

```
x << (3 * 8);
```

vs.

```
x << 030;
```

***
```
    tx_data[44] = utime>16;
```

Good catch!

I wonder if this would have been flagged with -Wall?

***
Nope. There's no warning for bool->int conversion:
https://stackoverflow.com/questions/28716391/gcc-forbid-impl...

***

Somewhat oddly even in C++, where bool is a distinct type, g++ doesn't have any warning for implicit bool to int conversion, although clang's clang-tidy "linter" tool does.

***

Nope, code was clean with -Wall on arm-9 compiler (i.e. gcc).

Interesting thought now you say that (no warning) - only found it because of seeing the pattern (apropos the article) and from that having a good idea what was causing it (knowing the int was assembled a byte at a time).

If I had a criticism of modern compilers, it's the blizzard of uninteresting warnings ("strncmp takes const char star, did you really mean to pass it unsigned char star") that make people want to not use -Wall.

***

All warnings are uninteresting until they're not.
> ("strncmp takes const char star, did you really mean to pass it unsigned char star")

This warning (with a different function) actually saved my bacon once, pointing me to a very obscure bug in the code.

My practice is to use -Wall and make sure that the code compiles without any warnings at all. Then I don't have a deluge of warnings to wade through.

***

Remember -Wall isn't all. -Wall -Wextra is the new -Wall.

I recommend asan too, where possible.

***

Why do compilers do stupid things like this? Seriously don't get why -Wall isn't all warnings.

***

Oh, you're right. I need to update my build system.

***

clang-tidy has an open issue to catch this. https://github.com/llvm/llvm-project/issues/56009

***

This is a good example

Why C's implicit conversions are trash. Why big endian is trash. Why you should do a reverse copy instead of stuff like this.

***

While it's based on C, a large chunk of the content is simply a great introduction to number representations in other bases such as binary and hexadecimal etc. Fundamental knowledge and very well explained, and a few insights I've not seen before.

***

Thanks, I wanted to post it after it was finished and revised. I also wanted to add more sections, but somebody was faster than I expected (thanks for posting it tod!). I suppose Getting to hn will add some valuable feedback in the first place.

I will probably add some more content in the coming weeks.

***

>Fundamental knowledge

yes? no? hard to say

The number of times I've needed this knowledge during all years of formal education and then years of work would be probably around 3.

Then I started working with C and close to hardware and it became something that I need everyday.

It's feels like bit proficiency is only useful in some very specific domains.

Thought: is HTTP foundational knowledge nowadays? after all whole world is built on it

if not, when will it become?

***

If you're a web developer and your product relies on HTTP, then HTTP is absolutely fundamental knowledge.

Similarly, if you're a web developer and your language doesn't even have integers, then understanding twos compliment is probably not fundamental.

That said, the people who proudly know the very least amount possible to perform their day job usually are not top performers.

***

Over years I've found that "top performer" is not purely about skill, often it is *just* about motivation, desire to move stuff ahead, to get things done, to improve current state of affair.

Like, rarely stuff is so hard that it requires some outstanding skills.

***

Like you say it really depends on what your job is. For me those bit algo's for a few years were my jam. I was moving data between 4 different CPU types and across 3 different network transports depending on the project and 4 different OS's. You need to know your bits when moving data between diff arch types and across different transport layers. These days most of that same sort of thing I did then? I would drop it into a json text stream and call it a day.

***

I think it's fundamental because understanding it goes hand-in-hand with understanding how computers actually work. When I interview applicants, I usually ask one or two questions about bitwise operations for this reason -- an engineer who knows how machines work at a very low level is valuable even if their work is not low level.

***

I think if you work with UUIDs, it's worth learning about bitwise stuff. So backend developers in general. It's not a day-to-day skill, but it is one I've used at just about every job in the last decade to elegantly and efficiently solve hard problems regarding idempotency and randomness.

***

When would you want to perform bitwise operations on a UUID?

***

A UUID is a bunch of bit fields packed together: the version/variant, and depending on the version, fields for the datetime, MAC address, node ID, etc. You'd use bit ops to both construct and extract the fields.

***

Say you need 5 UUIDs derived from one (say a primary key and some set of idempotency keys). You can mask them.

***

Or if you work directly with hardware, as I do. I use bitwise stuff constantly. Or if you're doing certain obscure kinds of highly optimized mathematical operations.

***

Bitwise NOT (⍰) gives you a bijection between negative and nonnegative integers in two's complement representation (which negation doesn't).

This is for example exploited by the return value of Java's binarySearch() function, which returns the (nonnegative) index of the search key when found, or else the (negative) bitwise NOT of the index where the key would have to be inserted [0]. In other words, it combines a nonnegative int value plus a flag into one int, while making the flag easily testable (< 0) and the value easily flippable (⍰). Strangely, the API doc doesn't mention bitwise NOT, but instead expresses it as numeric negation minus one (which is equivalent, as TFA explains).

[0] As opposed to C's bsearch(), which only returns a position when the key was found.

***

Great article. One potential improvement is not to call Two's complement MSB a 'sign bit'. It implies that it only stores a sign which is how 'Signed magnitude' works. With all its downsides like having two representations for 0 etc. The beauty of Two's complement is that everything works exactly like unsigned representation, with the exception of MSB. In Two's complement MSB contributes either 0 or negative 2^N-1. So for 8 bit signed numbers, most significant bit contributes either 0 or -128. Everything else, subtraction, hardware adders etc work the same. Signed numbers are basically numbers with a flexible number line, and the first bit represents where this number line starts. I personally remember MSB for signed integers as 'origin bit'.

***

I've found significant code in c/c++ where bitwise operations are done for things like division etc by shifting a certain way.

I Can imagine in the past, this was "faster", yet clang/gcc can emit the same by just writing a basic A/B function.

Seems the win goes to readability by reducing some of these old school hacks.

What say you, greybeards ?

***

Oh definitely. Some of this goes back to my 6502 assembly days when there was no hardware multiply instruction. So to multiply by 40, for example. I would shift right 3 bits, store the result, shift right 2 more bits and add the stored result.

Similarly, a fast divisibility test (we'll assume we're dividing n by some odd prime p):

1. Shift the bits of p right so that there is a 1 in the last position.
2. If n = p then p|n, if n < p then p!|n, otherwise continue.
3. Subtract p and go back to step 1.

(One of my ADD habits during meetings is to find the prime factors phone numbers or anything else very long. I do something similar with the numbers in decimal, but for this, I'll subtract multiples of the potential divisor to get 0s at the end of the number in decimal. I remember co-workers puzzling over a piece of paper with my notes I left behind in a conference room trying to figure out what the numbers represented and how the process worked.)

***
   Left, you would (obviously, this is a typo) shift left. And 3 followed by 2 since 1<<3 is 8, and 1<<5
   is 32 and 8+32 is 40.

***
   Depends on the situation. The compiler is smart, but in a way it's also dumb. It's very good at
   recognizing certain patterns and optimizing them, but not all patterns are recognized, and thus not
   all optimizations are applied, let alone consistently applied.

   For example, see the article and discussion on Bitwise Division from two days ago:
   https://news.ycombinator.com/item?id=34981027

***
   One thing to consider is that the compiler can't simply replace a division by just a right shift for
   signed variables (it will round towards -inf for negative numbers), so even today there's a tiny bit
   of benefit of explicitly using shifts if you know that the number can't be negative (and the compiler
   can't prove it) or you don't care about the rounding (https://godbolt.org/z/vTzYYxqz9).

   Of course that tiny bit of extra work is usually negligible, but might explain why the idiom has
   stuck around longer than you might otherwise expect.

***
   If the number can't be negative surely you should be using unsigned ints?

***
   I only have a little grey in my beard so far, but like all optimizations it heavily depends on
   context. My broad rule of thumb is that if you only care what the code does, you should let the
   compiler figure it out. If you care how the compiler accomplishes that goal, you should specify that
   rather than hoping things don't silently break in the future. This is a fairly common thing in crypto
   and systems code.

   But yes, fast inverse sqrt is obsolete.

***
   Nowaways you just write 'divout = divin/8192' and assume the compiler is going to do the right thing
   (and very possibly do something deeper than "divin>>12" at the assembler level).

   Makes me wonder who pays attention to this sort of thing these days :)

***
   I do! When optimizing code that must run obscenely fast, I look at the assembly the compiler spits
   out to make sure that it can't be improved on, speed-wise.

   Usually, it can't -- but sometimes...

***
   > I've found significant code in c/c++ where bitwise operations are done for things like division etc
   by shifting a certain way.

   Oh, yes. I used to do that sort of thing frequently because the time savings was significant enough.
   As you say, though, compilers have improved a great deal since then, so it's not generally needed
   anymore.

   If stupid bit tricks like that aren't necessary, they shouldn't be used. They do bring a
   readability/mental load cost with them.

***
   If x is signed and happens to be negative, x/16 will round one way, and x>>4 another. x/16 can still
   be implemented more performantly than a general division with unknown (or even known but non power of
   two) denominator, but it will be marginally slower than a plain shift. It depends on which semantics
   you desire.

***
   https://www.andreinc.net/2023/02/01/demystifying-bitwise-ops...

   With -O1 it performed the optimisation.

***
   Heads up: the gray_code function does not return the results shown in the table of correspondence
   above it.

   I suspect that this is because both the table and the code used were sourced from Wikipedia, and they
   correspond to different Gray codes. The table is for the BRGC, but the implementation isn't.

***
   I see several people sharing the Stanford bithacks link, so I'll throw in a slightly-less well-known
   resource that I found particularly instructive. Basically, a collection of the lemmata we can prove
   about fixed-length sequences of bits and the fun algorithms that can be built atop those results.

   https://www.jjj.de/fxt/

   And for the non-pdf-phobic: https://www.jjj.de/fxt/fxtbook.pdf

***
   For many years I have recommended "Code" by Charles Petzold and published by microsoft. It's a great

bathroom reader, reading on a road trip, or over a spaghetti dinner. It covers bottom up how computers work, starting out with - well what the heck are numbers and works up from there.

***

A related puzzle: https://www.quaxio.com/know_your_bits/

More such puzzles: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f02/www/L1... and http://csapp.cs.cmu.edu/public/datalab.pdf

Bit Twiddling Hacks: http://graphics.stanford.edu/�seander/bithacks.html

***

For the first puzzle, code that determines if a value has one bit high, without any if/while/for, here's what I got:

```
!(x & (x-1)) && x
```

Was there a "best" solution somewhere? I couldnt get the code window to work.

***

-1 is not an allowed operator in my puzzle.

***

The bitwise XOR operator (^) is a binary operator that compares the corresponding bits of two operands and returns a new value where each bit is set to 1 if the corresponding bits of the operand are different, and 0 if they are the same.

You may also think about XOR in a following way:

Any "1" in B flips (inverts) the corresponding bit in A.

It's like a vectorized NOT operation for single bits. Also works the other way (xor is commutative, A^B === B^A). This way of thinking is helpful when you see expressions like:

```
X ^ (1 << 3)
X ^ 8
X ^ 0b1000
```

Which means "X with bit 3 flipped". (3 means 4th from the right, as in ...3210).

***

Xor is also equivalent to adding without propagating the carry.

```
Ob010 + 0b011 = 0b101 (carry is propagated to the 3rd bit)
0b010 ^ 0b011 = 0b001 (same result with no carry)
```

***

Some very interesting advent of code contributions written in Rust, available on github, use bitwise operations. Shout out to Tim Visee!
https://github.com/timvisee/advent-of-code-2021/blob/master/...

***

I'm trying to find some good resources or some book that covers modern C.

For everything else from Python to Golangz Rust and everything in between - there are tons of quality books and even their own documentation in some cases is pretty decent. But not so with C.

If you read this and have something, please share. Just C (not much interested in C++)

***

https://en.cppreference.com/w/c

***

https://www.youtube.com/watch?v=QpAhX-gsHMs

***

Nice!

***

Just do a search on HN itself for "Modern C" and you will find a lot of discussions and recommendations.

That said, you can get started with any C book (The K&R Ansi C book is a good one to start with) since it is not a "huge" language and you can get to "newer" features incrementally.

***

I wish I have a use case for this as a web dev. I just haven't found the need for this so Ill just forget this again

***

Sounds like you need a hobby project to exercise those new muscles.

***

I totally intuitively understand bitwise operators in C because I grew up with assembler and all this was my second nature.

I'm afraid this article is unnecessarily complicated. The things that are eventually illuminated are really trivially simple, it's just that they are explained in a complicated way.

***

I thought this was going to be ridiculous but it's actually a really good. It's clear enough that you could extend it down to showing how logic gates work.

The only thing missing is a little endian discussion; it assumes big endian (network byte order), and that may be confusing for all those x86 users out there.

***

Endianness isn't relevant to any of the stuff discussed in the article. It is equally applicable to both big endian and little endian architectures.

***

> that may be confusing for all those x86 users out there.

It's pretty much every user these days - even most MIPS based network oriented devices run LE. BE lost many years ago.

***

The Network is big-endian (network byte order).

***

That's just a data serialization format, there are zillions of those. The important thing is the endianness of the CPU.

***

You are the one that brought up CPU architectures. Except for the now aging S390/z that will probably last in a few installations forever and the 5 people that pay to run POWER workstations the world is little endian.

***

That's a nicely written piece, I wish it had been around when I was first struggling with bit-mangling.

***

FTA: "Based on the above picture, another important observation is that to represent the number 1078 in binary, we need at least ten memory cells (bits) for it (look at the most significant power of 2 used, which is 10)"

As the picture shows, we need 11 bits, with powers ranging from zero to ten, both included (under the implicit assumption that we want to represent all integers between 0 and 1078. If all we want to represent is 1078, we can do with one or, pedantically, zero bits)

***

This is great. However, he omitted one I encountered a need for pretty recently and have so far not found a good solution:

How do you count the number of bits which have been set in a bitfield of type uint32_t?

I couldn't find any x64_64 intrinsics for this, which would probably be incredibly efficient.

***

Like bluesnowmonkey says, the concept you're looking for is called popcount, for population count. It's also called the Hamming weight. Wikipedia has 5 different example implementations under that latter name. Many C/C++ compilers have it available as an intrinsic as well, like __builtin_popcount or __popcnt. It's also std::popcount in C++20.

***

Wow! Just 5 instructions. This is almost 4 times faster than my previous best solution:

```
// 19 instructions, does not use intrinsics
int countbits(unsigned x) {
  unsigned n;
  n = (x >> 1) & 0x77777777;
  x = x - n;
  n = (n >> 1) & 0x77777777;
  x = x - n;
  n = (n >> 1) & 0x77777777;
  x = x - n;
  x = (x + (x >> 4)) & 0x0F0F0F0F;
  x = x\*0x01010101;
  return x >> 24;
}
```

Amazing tip, thanks.

***

popcount?

***

The graphics are impressive. Specifically, I've never seen someone fold a "bit surface" in half like that. I've also never seen a ladder view. Assuming Andrei Ciobanu himself posted this, how did you come up with those views? Bored on a Friday night?

***
I am not sure if this is all that original, I am sure what other have seen that before. For me it clicked when I was redoing the graphics.

What it's more interesting is that symmetry is specific to numbers in general, and the way we represent them. As an exercise if you use base 3, and plot more numbers you will also see hidden patterns.

Thanks for noticing that section.

***
Articles like these seen today should make us value more the love and effort that requires to train people. Because there is no shortage of "love" and effort (and money) for training AI models.

***
I don't understand why bit masking and manipulation is so popular when it makes the code impossible to read. I like using Ruby, string, and pack and unpack.

***
Well one reason is that bit operations are significantly faster than other methods, it's one reason for example, why compilers will turn a divide by a constant into bit manipulation.

Using a scripting language's string packing is hundreds of times slower than doing a couple of bit operations, and if there's a memory allocation involved, it can be thousands of times slower. If you're curious about this, I recommend doing some profiling to find out how fast your favorite algorithms are when using C bitwise operators compared to Ruby string packing.

FWIW, having the code be readable is mostly a familiarity problem that you can resolve by practicing more bitwise operations, if you want. If you spend your days in Ruby, then there might not be strong reasons to, but if you're curious and want to improve, you might have fun playing in C or C++. I spend my days mostly in CUDA, and using bit manipulation is par for the course, failure to use the best tricks can result in much lower compute throughput and much higher power consumption.

***
C23 finally introduces binary literals, e.g. you can write now "0b11110000" instead of "0xf0" for situations where binaries are more readable (but with a bit of 'training', hexadecimal works just as well).

Also, bitwise operations are essentially "SIMD for bits" and important down on the machine code level, it makes a lot of sense to have that same functionality also in higher level languages (unfortunately not all common bitwise instructions - like rotations - made it into high level languages).

(edited)

***
Do you ever wonder how Ruby's pack and unpack are implemented?

https://github.com/ruby/ruby/blob/4ce642620f10ae18171b41e166...

***
It takes some getting used to but the operations by themselves are readable, it is just that what you do with them can be complex. It is like arithmetic. People usually don't have a problem with addition, subtraction, multiplication and division, but it doesn't mean they won't have a hard time dealing with complex equations.

Packing and unpacking are just some of the things you can do with bitwise operations. They are a common problem, and they are often tricky, so having an API for that makes a lot of sense. But if what you need to do is not packing and unpacking, I find it harder to understand the unpack -> string manipulation -> pack workflow than bitwise operations, it also tends to be more verbose and slower.

***
Do Ruby's pack and unpack routines let you access a field smaller than a byte? What about a field that is split across a byte boundary?

I agree that bit masking and manipulation is hard to read and can be misused by ego-driven programmers. But the "popularity" because because bit masking and manipulation are both fundamental to computer science and an absolute necessity in certain situations.

***
I just do not understand why this document is so large - there just is not enough complexity in bitwise operators to warrant this giant multi chapter document. If anything I feel it will be confusing because it's so much text being used to describe so little actual logic.

***
integer promotion yuminess...

***
This long article is a great example of why I believe GPT has a strong future.

The article spends several pages to explain hexadecimal, bases, etc. Probably some fraction of the audience already knows that. In that case, the article should automatically adapt and skip that section. Some people will react negatively to the mathematical formulation, preferring the intuitive section that follows instead.

But this is a static blog post, so it doesn't know how to adapt. Imagine the same post, but with some toggles/sliders (you can come up with even more sophisticated mechanisms): I indicate my level of competency, and the article re-writes itself to match my knowledge. Skip the boring math, show a lot of examples, etc. Or the opposite. The point is that it adapts to you. GPT (or some future variant) is really good at doing this.

***

GPT isn't needed. Just hyperlinks, like exist at the top of the document. Paired with the ability to expand sections either in place or in an adjacent view. State the prerequisite knowledge, then give a way to skip past the explanations or have the explanations hidden by default. Permit further expansions up to the limit of what the author cares to provide, can be added to later.

***

"Hey chatGPT, paraphrase my comment without the passive aggressive dismissal of its contents."

In all seriousness though, this blogpost does exactly what it intends to do. Demistify bitwise operations and you can't skip the math.

***

Sure, let's just skip on the minor details that modern computation rests on. Who needed that anyway.

Although I do agree that repeating the basics on every post is cumbersome, it seems adequate for this post.

***

The problem with GPT is there's a good chance that it introduces information which is plain wrong while doing that transformation.

***

> Some people will react negatively to the mathematical formulation, preferring the intuitive section that follows instead.

They have no place in software engineering then.
