Algorithm Tutorials

A bit of fun: fun with bits

By bmerry
TopCoder Member

Archive
Printable view
Discuss this article
Write for TopCoder

Introduction

Most of the optimizations that go into TopCoder contests are high-level; that is, they affect the algorithm rather than the implementation. However, one of the most useful and effective low-level optimizations is bit manipulation, or using the bits of an integer to represent a set. Not only does it produce an order-of-magnitude improvement in both speed and size, it can often simplify code at the same time.

I'll start by briefly recapping the basics, before going on to cover more advanced techniques.

The basics

At the heart of bit manipulation are the bit-wise operators `&` (and), `|` (or), `~` (not) and `^` (xor). The first three you should already be familiar with in their boolean forms (`&&`, `||` and `!`). As a reminder, here are the truth tables:

A	B	!A	A && B	A || B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

The bit-wise versions of the operations are the same, except that instead of interpreting their arguments as true or false, they operate on each bit of the arguments. Thus, if A is 1010 and B is 1100, then

- A & B = 1000
- A | B = 1110
- A ^ B = 0110
- ~A = 11110101 (the number of 1's depends on the type of A).

The other two operators we will need are the shift operators `a << b` and `a >> b`. The former shifts all the bits in `a` to the left by `b` positions; the latter does the same but shifts right. For non-negative values (which are the only ones we're interested in), the newly exposed bits are filled with zeros. You can think of left-shifting by `b` as multiplication by 2^b and right-shifting as integer division by 2^b. The most common use for shifting is to access a particular bit, for example, `1 << x` is a binary number with bit `x` set and the others clear (bits are almost always counted from the right-most/least-significant bit, which is numbered 0).

In general, we will use an integer to represent a set on a domain of up to 32 values (or 64, using a 64-bit integer), with a 1 bit representing a member that is present and a 0 bit one that is absent. Then the following operations are quite straightforward, where `ALL_BITS` is a number with 1's for all bits corresponding to the elements of the domain:

Set union
```
A | B
```
Set intersection
```
A & B
```
Set subtraction
```
A & ~B
```
Set negation
```
ALL_BITS ^ A
```
Set bit
```
A |= 1 << bit
```
Clear bit
```
A &= ~(1 << bit)
```
Test bit
```
(A & 1 << bit) != 0
```

Extracting every last bit

In this section I'll consider the problems of finding the highest and lowest 1 bit in a number. These are basic operations for splitting a set into its elements.

Finding the lowest set bit turns out to be surprisingly easy, with the right combination of bitwise and arithmetic operators. Suppose we wish to find the lowest set bit of `x` (which is known to be non-zero). If we subtract 1 from `x` then this bit is cleared, but all the other one bits in `x` remain set. Thus, `x & ~(x - 1)` consists of only the lowest set bit of `x`. However, this only tells us the bit value, not the index of the bit.

If we want the index of the highest or lowest bit, the obvious approach is simply to loop through the bits (upwards or downwards) until we find one that is set. At first glance this sounds slow, since it does not take advantage of the bit-packing at all. However, if all 2^N subsets of the N-element domain are equally likely, then the loop will take only two iterations on average, and this is actually the fastest method.

The 386 introduced CPU instructions for bit scanning: BSF (bit scan forward) and BSR (bit scan reverse). GCC exposes these instructions through the built-in functions `__builtin_ctz` (count trailing zeros) and `__builtin_clz` (count leading zeros). These are the most convenient way to find bit indices for C++ programmers in TopCoder. Be warned though: the return value is *undefined* for an argument of zero.

Finally, there is a portable method that performs well in cases where the looping solution would require many iterations. Use each byte of the 4- or 8-byte integer to index a precomputed 256-entry table that stores the index of the highest (lowest) set bit in that byte. The highest (lowest) bit of the integer is then the maximum (minimum) of the table entries. This method is only mentioned for completeness, and the performance gain is unlikely to justify its use in a TopCoder

match.

Counting out the bits

One can easily check if a number is a power of 2: clear the lowest 1 bit (see above) and check if the result is 0. However, sometimes it is necessary to know how many bits are set, and this is more difficult.

GCC has a function called `__builtin_popcount` which does precisely this. However, unlike `__builtin_ctz`, it does not translate into a hardware instruction (at least on x86). Instead, it uses a table-based method similar to the one described above for bit searches. It is nevertheless quite efficient and also extremely convenient.

Users of other languages do not have this option (although they could re-implement it). If a number is expected to have very few 1 bits, an alternative is to repeatedly extract the lowest 1 bit and clear it.

All the subsets

A big advantage of bit manipulation is that it is trivial to iterate over all the subsets of an N-element set: every N-bit value represents some subset. Even better, if A is a subset of B then the number representing A is less than that representing B, which is convenient for some dynamic programming solutions.

It is also possible to iterate over all the subsets of a particular subset (represented by a bit pattern), provided that you don't mind visiting them in reverse order (if this is problematic, put them in a list as they're generated, then walk the list backwards). The trick is similar to that for finding the lowest bit in a number. If we subtract 1 from a subset, then the lowest set element is cleared, and every lower element is set. However, we only want to set those lower elements that are in the superset. So the iteration step is just `i = (i - 1) & superset`.

Even a bit wrong scores zero

There are a few mistakes that are very easy to make when performing bit manipulations. Watch out for them in your code.

1. When executing shift instructions for `a << b`, the x86 architecture uses only the bottom 5 bits of `b` (6 for 64-bit integers). This means that shifting left (or right) by 32 does nothing, rather than clearing all the bits. This behaviour is also specified by the Java and C# language standards; C99 says that shifting by at least the size of the value gives an undefined result. Historical trivia: the 8086 used the full shift register, and the change in behaviour was often used to detect newer processors.
2. The `&` and `|` operators have lower precedence than comparison operators. That means that `x & 3 == 1` is interpreted as `x & (3 == 1)`, which is probably not what you want.
3. If you want to write completely portable C/C++ code, be sure to use unsigned types, particularly if you plan to use the top-most bit. C99 says that shift operations on negative values are undefined. Java only has signed types: `>>` will sign-extend values (which is probably *not* what you want), but the Java-specific operator `>>>` will shift in zeros.

Cute tricks

There are a few other tricks that can be done with bit manipulation. They're good for amazing your friends, but generally not worth the effect to use in practice.

Reversing the bits in an integer

```
x = ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);
x = ((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2);
x = ((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);
x = ((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8);
x = ((x & 0xffff0000) >> 16) | ((x & 0x0000ffff) << 16);
```

As an exercise, see if you can adapt this to count the number of bits in a word.

Iterate through all k-element subsets of {0, 1, ... N-1}

```
int s = (1 << k) - 1;
while (!(s & 1 << N))
{
    // do stuff with s
    int lo = s & ~(s - 1);        // lowest one bit
    int lz = (s + lo) & ~s;       // lowest zero bit above lo
    s |= lz;                      // add lz to the set
    s &= ~(lz - 1);               // reset bits below lz
    s |= (lz / lo / 2) - 1;       // put back right number of bits at end
}
```

In C, the last line can be written as `s |= (lz >> ffs(lo)) - 1` to avoid the division.

Evaluate `x ? y : -y`, where `x` is 0 or 1

```
(-x ^ y) + x
```

This works on a twos-complement architecture (which is almost any machine you find today), where negation is done by inverting all the bits then adding 1. Note that on i686 and above, the original expression can be evaluated just as efficiently (i.e., without branches) due to the `CMOVE` (conditional move) instruction.

Sample problems

TCCC 2006, Round 1B Medium

For each city, keep a bit-set of the neighbouring cities. Once the part-building factories have been chosen (recursively), ANDing together these bit-sets will give a bit-set which describes the possible locations of the part-assembly factories. If this bit-set has k bits, then there are ${}^{k}C_{m}$ ways to allocate the part-assembly factories.

TCO 2006, Round 1 Easy

The small number of nodes strongly suggests that this is done by considering all possible subsets. For every possible subset we consider two possibilities: either the smallest-numbered node does not communicate at all, in which case we refer back to the subset that excludes it, or it communicates with some node, in which case we refer back to the subset that excludes both of these nodes. The resulting code is extremely short:

```
static int dp[1 << 18];

int SeparateConnections::howMany(vector <string> mat)
{
    int N = mat.size();
    int N2 = 1 << N;
    dp[0] = 0;
    for (int i = 1; i < N2; i++)
    {
        int bot = i & ~(i - 1);
        int use = __builtin_ctz(bot);
        dp[i] = dp[i ^ bot];
        for (int j = use + 1; j < N; j++)
            if ((i & (1 << j)) && mat[use][j] == 'Y')
                dp[i] = max(dp[i], dp[i ^ bot ^ (1 << j)] + 2);
    }
    return dp[N2 - 1];
}
```

SRM 308, Division 1 Medium

The board contains 36 squares and the draughts are indistinguishable, so the possible positions can be encoded into 64-bit integers. The first step is to enumerate all the legal moves. Any legal move can be encoded using three bit-fields: a *before* state, an *after* state and a *mask*, which defines which parts of the before state are significant. The move can be made from the current state if `(current & mask) == before`; if it is made, the new state is `(current & ~mask) | after`.

SRM 320, Division 1 Hard

The constraints tell us that there are at most 8 columns (if there are more, we can swap rows and columns), so it is feasible to consider every possible way to lay out a row. Once we have this information, we can solve the remainder of the problem (refer to the match editorial for details). We thus need a list of all n-bit integers which do not have two adjacent 1 bits, and we also need to know how many 1 bits there are in each such row. Here is my code for this:

```
for (int i = 0; i < (1 << n); i++)
{
    if (i & (i << 1)) continue;
    pg.push_back(i);
    pgb.push_back(__builtin_popcount(i));
}
```