Bitwise Operators and Bitmasking

Taein Kim

September 2022

1 Bit Representation of Numbers

Often, in competitive programming, it may be useful to represent numbers as base 2 before performing calculations. This is called the "bitwise representation" of a number; these representations can be manipulated faster, and have a lot of useful operators and applications.

1.1 The Basics

If we're going to use bitwise operators in our algorithms, we must first understand how base 2 works. Unlike base 10, the standard base, numbers in each digit are represented as powers of two. For example, the bitwise representation of 2 equals 10, and the representation of 9 equals 1001.

1.2 Other Applications

Integers in base 10 are not the only data structure that can be manipulated into a bitwise representation– we can also do the same with sets and subsets of integers. For example, say we have a subset of the first 8 integers: $\{1, 3, 4, 8\}$. We can represent sum of each number in the set raised to the power of 2 bitwise, and get $2^8 + 2^4 + 2^3 + 2^1 = 100011010$.

2 Bitwise Operators

Bitwise operators are operations performed on one or two binary numbers. These operations are important because they all run in constant-time, and will cut down your program runtime significantly under the right circumstances. Especially with manipulation with sets, which tend to take above O(n) time, turning the set into a binary number and using bitwise operators will be much faster.
Here are the most basic operators:

2.1 Two-number Bitwise Operators

1. AND (&): Returns 1 if both numbers in a corresponding digit are 1, otherwise returns 0.
2. OR (|): Returns 1 if either number in a corresponding digit is 1, otherwise returns 0.
3. XOR ($\wedge$): Returns 1 if one number in a corresponding digit is 1 and the other 0; otherwise returns 0.

2.2 One-number Bitwise Operators

1. NOT ($\sim$): Flips every 1 and 0 in a binary number.
2. Right Shift (>>): Shifts every bit of a number one right, filling voids on the left with 0. Similar to dividing the base-10 representation of the number by 2.
3. Left Shift (<<): Shifts every bit of a number one left, filling voids on the right with 0. Similar to multiplying the base-10 representation of the number by 2.

3 Applications and Problems

3.1 Exercises

Find each of the following in base 10:

1. 5 & 10
2. 25 | 33
3. 50 >> 2
4. ~ 36

3.2 Simple Applications

Find a bitwise solution for these simple programming problems:

1. Determine whether a number is even or odd.
2. Given two sets of integers, find whether or not the two sets share an element.

4 Addition Using Bitwise Operators, Intro to Bitmasking

Through a clever manipulation of multiple different bitwise operators, we can create our typical addition operation.

4.1 Addition

Try to use three of the given six operations to add one binary number to another. Hint: you'll need two separate operations to both determine the sum of two given digits and determine whether anything will carry over to the next. Try manipulating one of the two numbers to keep track of all the numbers being carried over until it reaches zero.

4.2 The Solution

First, we will keep track of all digits that carry over by the AND function. Then, we use the XOR function to get the initial sum without carrying over; lastly, we repeat the same process, adding the results from AND and XOR, until the results from the AND process reaches zero and there is nothing to carry over.

5 Bitmasking

A bitmask is a "mask" that we can apply on a binary number in order to check elements or perform operations on a binary number. A bitmask is given as the same form as the number we are applying the mask to. Some common tasks we perform using a bitmask include checking whether an element is in a set or iterating through all subsets of a set. Note that a bitmask can be inverted using the NOT function, just as a regular binary number can.

5.1 Simple Manipulation

Try to find a bitmask-bitwise operator pair that can add, remove, and check for the ith element in a set. Suppose you're working with a set S and a bitmask B. Note that both the bitmask and the set are zero-indexed from the left.

5.2 Solution

1. Adding: S OR 1<<i
2. Removing: S OR ∼(1<<i)
3. Checking: Check whether S AND 1<<i != 0

6 Bitmask Challenge Problems

Try to find algorithms that involve for loops and bitwise operations for the challenges below!

6.1 Iteration through all subsets of a set

Here, given a bitmask M, we want to iterate through all submasks of M that only include bits that M includes.

6.2 Iteration through all submasks of all subsets of a set

Here, we want to first iterate through all bitmasks, which is a fixed number; however, we also want to iterate through the submasks of each bitmask we visit.

6.3 Solutions

1. Challenge 1:

```
int s = m;
while (s > 0) {
    #add s
    s = (s-1) & m;
}
```

   Here, a placeholder s begins exactly at m. Afterwards, for each iteration, we subtract 1 from s; this will result in every bit to the right of the last 1-bit of the previous mask being converted to 1. Then, we simply get rid of all of the unnecessary 1's by performing an AND with the original bitmask M, resulting in the next smallest bitmask that is a submask of M.

2. Challenge 2:

```
for (int m=0; m<(1<<n); ++m)
    for (int s=m; s; s=(s-1)&m)
        #add m and its submask s
```

   Notice how the same algorithm above is executed for each mask of length n. For each bit, the index can be included in neither m nor s, m but not s, or both m and s. Thus, the algorithm as a whole will execute in $O(3^n)$ time.

7 Math and Bitmasks

As an ending note, bitmasks and bitwise operators can also be used to find whether two numbers are relatively prime. For example, if two numbers AND to zero in base two, we can tell that they are relatively prime. Similarly, the OR function is used as a least common multiple, and iterating through bits and submasks is equivalent to iterating through prime and all divisors.