Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 5 - Binary Mathematics

Binary Representation Alright, today we're going to talk about binary numbers, binary operations, and how to use them effectively. These are useful not only because they'll give you a better understanding of how integer operations are handled, but because the binary representation of integers can be useful as a way to concisely represent system states or subsets of items.

The binary representation of unsigned integers is very simple, and you've likely seen it before:

Place Value

```
16 8 4 2 1
```

Digits

```
 1 0 1 1 0
```

Each digit corresponds to a power of two, and contains either a 0 or a 1, and the integer is the appropriate sum of the powers of two; in this case, $16 + 4 + 2 = 22$. Most programming languages use integers with a fixed number of digits (generally 32), in which case only numbers up to $2^N - 1$ are storable, and trying to manipulate integers larger than this will cause overflow problems.

Signed (that is, marked as positive or negative) integers are more complicated. Historically there have been several different ways to represent negative integers in binary, but we're only going to talk about the current most common way, which is found in nearly every modern programming language, including C, C++, Java, and Python, the two's complement method. Let's start with an eight bit number:

Place Value

```
128 64 32 16 8 4 2 1
```

Digits

```
  0  0  0  1 0 1 1 0
```

The largest unsigned number we can represent is $2^8 - 1$, or 255. The two's complement representation of a signed number replaces the place value of the highest order bit (the leftmost) with its negative:

Place Value

```
-128 64 32 16 8 4 2 1
```

Digits

```
0  0  0  1 0 1 1 0
```

If the bit in that position is zero, then the number is identical as under the unsigned representation. If it is 1, then the number is negative, because the highest order place value has a large absolute value than the sum of all the other place values, and the remaining digits count up from that maximal negative value. This then limits the range of representable values from -2^{N-1} to $2^{N-1} - 1$.

Binary (Bitwise) Operations There are some special bitwise binary operations that are meant to act on integers not as numbers but as strings of binary digits. These operations are usually very fast, because they compile to very concise machine code, and are often even faster than basic arithmatic operation. Notation will vary from language to language, but the tokens I'll show are fairly standard. Be warned that due to the ubiquitous two's complement system, binary operations on *negative* numbers don't always give you the answer you expect!

Binary AND (&) Binary AND operations take two fixed-length integers and return the number whose representation is such that for each bit in the output, that bit is 1 if it is 1 in both inputs and 0 otherwise. (I.e., if both the bit in the first input AND the second input is on, then it is on in the output.) Usually notated by an ampersand, '&'. For example:

```
  0 1 0 1 (=5)
& 0 0 1 1 (=3)

  0 0 0 1 (=1)
```

Binary OR (|) Similarly, binary ON operations take two fixed-length integers and returns the number where if either the bit in the first input OR the second input is on, then it is on in the output. Usually notated by a pipe, '|':

```
  0 1 0 1 (=5)
| 0 0 1 1 (=3)

  0 1 1 1 (=7)
```

Binary XOR (exclusive-OR) (^) Binary XOR operations similarly take two integers, but now return the number where a bit in the output is on only if it is on in one but not the other of the two input numbers. Usually notated by a caret, '^':

```
  0 1 0 1 (=5)
^ 0 0 1 1 (=3)

  0 1 1 0 (=6)
```

Left and Right Bitshifts (<< and >>) Left and right bitshifts shift every bit in the input number to the right or left in the output number. A right shift shifts every bit to the next lowest order bit:

```
1 0 1 0 (=10)
>>

0 1 0 1 (=5)
```

While the left shift shifts every bit to the next highest order bit:

```
0 0 1 1 (=3)
<<

0 1 1 0 (=6)
```

Bits that are right-shifted from the lowest order bit are simply lost entirely. The new highest order bit is always 0 for positive numbers, but for negative numbers it depends on the implementation whether it is a 0, as for positive numbers (called a logical shift) or a 1 for the sign bit (called an arithmetic shift). Similarly for left-shifts the bits shifted off the register are discarded and a new 0 bit enters the lowest-order position.

You should note that left shifting is equivalent to multiplying by 2 and right shifting equivalent to dividing by 2 and rounding down (towards negative infinity, not towards zero), and so can be useful substitutes for multiplication and division by powers of 2 (as these operations are much faster).

Binary NOT (~) Finally, the binary NOT operation simply reverses the value of every bit in the number (including the sign bit!), and is denoted by a tilde:

```
~ (sign 0) 1 1 0 0 (=12)

  (sign 1) 0 0 1 1 (=-13)
```

For signed two's complement integers, this is equal to -(N + 1).

Bitmasks Basic knowledge of these tools will be useful for particular problems dealing directly with binary arithmetic, but by far the most common way you can use binary representation of integers in the context of a programming challenge is as a bitmask. A bitmask is the use of the binary representation of an integer to represent a (small) vector of booleans, with each bit in the representation standing in for a particular element. Bitmasks can represent, for example, a particular subset of elements in a set of 10 total elements: the first 10 digits of the number correspond to whether or not each element is present in the subset:

Place Value

```
512 256 128 64 32 16 8 4 2 1
```

Element ID

```
  9   8   7  6  5  4 3 2 1 0
```

Digits

```
  1   0   1  1  0  0 1 1 0 0 (=716 and subset{2,3,6,7,9})
```

This representation makes several things quite easy. For one, we can quickly iterate through all subsets. Each subset is represented by an integer from 0 (no elements selected) to $2^{10} - 1$ (all elements selected), so we can simply loop through these integers and ensure we have covered each possible subset. Secondly, there are a number of useful tricks using binary operations to quickly manipulate bitmasks that can be much faster than operating on an array of booleans.

Manipulation of Bitmasks Let our bitmask be S. We can modify the bitmask using binary operations as follows:

Turn on element j

```
S = S | (1 << j)
```

Left shifting 1 by j places gives us the j^{th} power of 2, equivalent to a bitmask with only the j^{th} element is set. By OR-ing this with our initial bitmask, we ensure that this element is set.

```
          S = 100010 (=34)
OR (1 << 3) = 001000 (=8)

          S = 101010 (=42)
```

Turn off element j Similarly, we can turn off element j by using the bitwise AND with the NOT of a left-shifted 1:

```
S = S & ~(1 << j)

            S = 101010 (=42)
AND ~(1 << 3) = 110111 (=55)

            S = 101010 (=34)
```

Toggle element j We can toggle an element by XOR-ing the bitmask with the j^{th} element:

```
S = S ^ (1 << j)

           S = 101010 (=42)
XOR (1 << 3) = 001000 (=8)

           S = 100010 (=34)
```

Check if element j is set We can check if element j is set by AND-ing our bitmask with the j^{th} element:

```
T = S & (1 << j)

           S = 101010 (=42)
AND (1 << 3) = 001000 (=8)

           T = 001000 (=8)
```

We'll get some positive number if it is set, and zero if it isn't. Note that this is also a quick way to check if an integer is even or odd by AND-ing with 1, which checks whether the lowest order bit is set or not.

Identify the least significant bit (smallest element) that is set

```
T = S & (-S)

     S = 000...000101010 (=42, two's complement)
AND -S = 111...111010110 (=-42, two's complement)

     T = 000...000000010 (=2)
```

Remember that NOT flips all the bits and (in two's complement) is equal to $-(N+1)$. If we take this complement and add 1 to it, we carry over until we hit the first on bit in the original number, meaning that $-N$ has all the bits to the right of and including the least significant on bit equal to N and all the bits to the left of it opposite of N.

Turn off the least significant on bit

```
S = S & (S - 1)

        S = 101010 (=42)
AND S - 1 = 101001 (=41)

        S = 101000 (=40)
```

Turn on the least significant off bit

```
S = S | (S + 1)

       S = 101010 (=42)
OR S + 1 = 101011 (=43)

       S = 101011 (=43)
```

Sample Problem Let's discuss a real problem where application of binary mathematics will simplify things: Snapper Chain, from the 2010 GCJ Qualification Round.

Here's the problem statement (simplified for clarity):

> We have a *chain* of snappers. Each snapper is a device with two states: it can be either on or off and either powered or unpowered. It is plugged into a power source and another device can be plugged into it. It passes power on to the device plugged into it only when it is both on and powered. When you snap your fingers, any snapper receiving power (whether on or off) toggles between on or off states, all at once.
>
> We have a *chain* of N snappers serially plugged into each other, with the first snapper plugged into a wall socket that is constantly powered, and a light bulb plugged into the last snapper that is lit only when powered. All snappers start in the OFF state, with only the first snapper powered (directly from the wall socket). The first snap toggles the first (powered) snapper to ON, which then starts passing power to the second snapper. The next snap toggles the first two (powered) snappers, toggling the first to OFF (which stops passing power) and the second to ON.
>
> After K snaps, is the light at the end of the chain lit?

How can we solve this problem using the binary tools we've learned about? Well, let's start by representing each of the powered and on/off states of the snapper chain as a bitmask, with the lowest order bit representing the snapper plugged into the wall, and so on down the chain. Our on/off bitmask starts off as 0 (all the snappers are OFF) while the power bitmask starts off as 1 (only the first snapper is powered):

```
PWR = 000...000001 (initial)
SWT = 000...000000 (initial)
```

When we snap our fingers, we want to take all the powered snappers and toggle their on/off state. We can do that with an XOR operation between the two bitmasks:

```
    PWR = 000...000001
XOR SWT = 000...000000

    SWT = 000...000010 (after 2 snaps)
```

And now the powered bitmask is only 1 (as the first snapper is not ON anymore). The next snap updates SWT again:

```
    PWR = 000...000001
XOR SWT = 000...000010

    SWT = 000...000100 (after 4 snaps)
```

With some inspection of the outputs, you should be able to convince yourself that this series of operations is equivalent to incrementing the on/off bitmask by 1 with each snap! That means that after K snaps, the on/off bitmask will be equal to K. As our light bulb, plugged into N snappers, is lit if and only if all N snappers are on, we can inspect the rightmost N bits of K and if and only if they are all on will the bulb be lit.

Lab Section Again, we're going to work on this week's homework problem during lab section:

- GCJ Qualification 2011 - Candy Splitting