5

Bit Manipulation

Bit manipulation is used in a variety of problems. Sometimes, the question explicitly calls for bit manipulation. Other times, it's simply a useful technique to optimize your code. You should be comfortable doing bit manipulation by hand, as well as with code. Be careful; it's easy to make little mistakes.

▸ Bit Manipulation By Hand

If you're rusty on bit manipulation, try the following exercises by hand. The items in the third column can be solved manually or with "tricks" (described below). For simplicity, assume that these are four-bit numbers.

If you get confused, work them through as a base 10 number. You can then apply the same process to a binary number. Remember that ^ indicates an XOR, and ~ is a NOT (negation).

`0110 + 0010`	`0011 * 0101`	`0110 + 0110`
`0011 + 0010`	`0011 * 0011`	`0100 * 0011`
`0110 - 0011`	`1101 >> 2`	`1101 ^ (~1101)`
`1000 - 0110`	`1101 ^ 0101`	`1011 & (~0 << 2)`

Solutions: line 1 (1000, 1111, 1100); line 2 (0101, 1001, 1100); line 3 (0011, 0011, 1111); line 4 (0010, 1000, 1000).

The tricks in Column 3 are as follows:

1. `0110 + 0110` is equivalent to `0110 * 2`, which is equivalent to shifting `0110` left by 1.
2. `0100` equals 4, and multiplying by 4 is just left shifting by 2. So we shift `0011` left by 2 to get `1100`.
3. Think about this operation bit by bit. If you XOR a bit with its own negated value, you will always get 1. Therefore, the solution to `a^(~a)` will be a sequence of 1s.
4. `~0` is a sequence of 1s, so `~0 << 2` is 1s followed by two 0s. ANDing that with another value will clear the last two bits of the value.

If you didn't see these tricks immediately, think about them logically.

▸ Bit Facts and Tricks

The following expressions are useful in bit manipulation. Don't just memorize them, though; think deeply about why each of these is true. We use "1s" and "0s" to indicate a sequence of 1s or 0s, respectively.

```
x ^ 0s = x        x & 0s = 0       x | 0s = x
x ^ 1s = ~x       x & 1s = x       x | 1s = 1s
 x ^ x = 0         x & x = x        x | x = x
```

To understand these expressions, recall that these operations occur bit-by-bit, with what's happening on one bit never impacting the other bits. This means that if one of the above statements is true for a single bit, then it's true for a sequence of bits.

▸ Two's Complement and Negative Numbers

Computers typically store integers in two's complement representation. A positive number is represented as itself while a negative number is represented as the two's complement of its absolute value (with a 1 in its sign bit to indicate that a negative value). The two's complement of an N-bit number (where N is the number of bits used for the number, *excluding* the sign bit) is the complement of the number with respect to 2^N.

Let's look at the 4-bit integer -3 as an example. If it's a 4-bit number, we have one bit for the sign and three bits for the value. We want the complement with respect to 2^3, which is 8. The complement of 3 (the absolute value of -3) with respect to 8 is 5. 5 in binary is `101`. Therefore, -3 in binary as a 4-bit number is 1101, with the first bit being the sign bit.

In other words, the binary representation of -K (negative K) as a N-bit number is `concat(1,` 2^{N-1} `- K)`.

Another way to look at this is that we invert the bits in the positive representation and then add 1. 3 is `011` in binary. Flip the bits to get `100`, add 1 to get `101`, then prepend the sign bit (1) to get `1101`.

In a four-bit integer, this would look like the following.

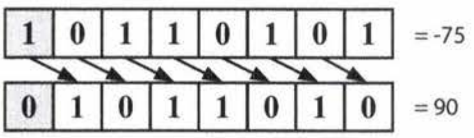
Positive Values		Negative Values	
7	0 111	-1	1 111
6	0 110	-2	1 110
5	0 101	-3	1 101
4	0 100	-4	1 100
3	0 011	-5	1 011
2	0 010	-6	1 010
1	0 001	-7	1 001
0	0 000		

Observe that the absolute values of the integers on the left and right always sum to 2^3, and that the binary values on the left and right sides are identical, other than the sign bit. Why is that?

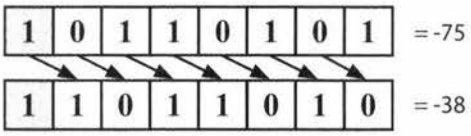
▸ Arithmetic vs. Logical Right Shift

There are two types of right shift operators. The arithmetic right shift essentially divides by two. The logical right shift does what we would visually see as shifting the bits. This is best seen on a negative number.

In a logical right shift, we shift the bits and put a `0` in the most significant bit. It is indicated with a >>> operator. On an 8-bit integer (where the sign bit is the most significant bit), this would look like the image below. The sign bit is indicated with a gray background.



In an arithmetic right shift, we shift values to the right but fill in the new bits with the value of the sign bit. This has the effect of (roughly) dividing by two. It is indicated by a >> operator.



What do you think these functions would do on parameters `x = -93242` and `count = 40`?

```
int repeatedArithmeticShift(int x, int count) {
  for (int i = 0; i < count; i++) {
    x >>= 1; // Arithmetic shift by 1
  }
  return x;
}

int repeatedLogicalShift(int x, int count) {
  for (int i = 0; i < count; i++) {
    x >>>= 1; // Logical shift by 1
  }
  return x;
}
```

With the logical shift, we would get `0` because we are shifting a zero into the most significant bit repeatedly.

With the arithmetic shift, we would get `-1` because we are shifting a one into the most significant bit repeatedly. A sequence of all `1`s in a (signed) integer represents `-1`.

▸ Common Bit Tasks: Getting and Setting

The following operations are very important to know, but do not simply memorize them. Memorizing leads to mistakes that are impossible to recover from. Rather, understand *how* to implement these methods, so that you can implement these, and other, bit problems.

Get Bit

This method shifts `1` over by `i` bits, creating a value that looks like `00010000`. By performing an AND with `num`, we clear all bits other than the bit at bit `i`. Finally, we compare that to `0`. If that new value is not zero, then bit `i` must have a `1`. Otherwise, bit `i` is a 0.

```
boolean getBit(int num, int i) {
  return ((num & (1 << i)) != 0);
}
```

Set Bit

`SetBit` shifts `1` over by `i` bits, creating a value like `00010000`. By performing an OR with `num`, only the value at bit `i` will change. All other bits of the mask are zero and will not affect `num`.

```
int setBit(int num, int i) {
  return num | (1 << i);
}
```

Clear Bit

This method operates in almost the reverse of `setBit`. First, we create a number like `11101111` by creating the reverse of it (`00010000`) and negating it. Then, we perform an AND with `num`. This will clear the `i`th bit and leave the remainder unchanged.

```
int clearBit(int num, int i) {
  int mask = ~(1 << i);
  return num & mask;
}
```

To clear all bits from the most significant bit through `i` (inclusive), we create a mask with a `1` at the `i`th bit (`1 << i`). Then, we subtract `1` from it, giving us a sequence of `0`s followed by `i` `1`s. We then AND our number with this mask to leave just the last `i` bits.

```
int clearBitsMSBthroughI(int num, int i) {
  int mask = (1 << i) - 1;
  return num & mask;
}
```

To clear all bits from `i` through `0` (inclusive), we take a sequence of all `1`s (which is `-1`) and shift it left by `i + 1` bits. This gives us a sequence of 1s (in the most significant bits) followed by `i` `0` bits.

```
int clearBitsIthrough0(int num, int i) {
  int mask = (-1 << (i + 1));
  return num & mask;
}
```

Update Bit

To set the `i`th bit to a value v, we first clear the bit at position `i` by using a mask that looks like `11101111`. Then, we shift the intended value, v, left by `i` bits. This will create a number with bit `i` equal to v and all other bits equal to `0`. Finally, we OR these two numbers, updating the `i`th bit if v is `1` and leaving it as `0` otherwise.

```
int updateBit(int num, int i, boolean bitIs1) {
  int value = bitIs1 ? 1 : 0;
  int mask = ~(1 << i);
  return (num & mask) | (value << i);
}
```

Interview Questions

5.1 **Insertion:** You are given two 32-bit numbers, N and M, and two bit positions, `i` and `j`. Write a method to insert M into N such that M starts at bit `j` and ends at bit `i`. You can assume that the bits `j` through `i` have enough space to fit all of M. That is, if M = `10011`, you can assume that there are at least 5 bits between `j` and `i`. You would not, for example, have `j` = 3 and `i` = 2, because M could not fully fit between bit 3 and bit 2.

EXAMPLE

Input: `N = 10000000000, M = 10011, i = 2, j = 6`

Output: `N = 10001001100`

Hints: #137, #169, #215

pg 276

5.2 **Binary to String:** Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."

Hints: #143, #167, #173, #269, #297

pg 277

5.3 **Flip Bit to Win:** You have an integer and you can flip exactly one bit from a `0` to a `1`. Write code to find the length of the longest sequence of 1s you could create.

EXAMPLE

Input: `1775 (or: 11011101111)`

Output: `8`

Hints: #159, #226, #314, #352

pg 278

5.4 **Next Number:** Given a positive integer, print the next smallest and the next largest number that have the same number of 1 bits in their binary representation.

Hints: #147, #175, #242, #312, #339, #358, #375, #390

pg 280

5.5 **Debugger:** Explain what the following code does: `((n & (n-1)) == 0)`.

Hints: #151, #202, #261, #302, #346, #372, #383, #398

pg 285

5.6 **Conversion:** Write a function to determine the number of bits you would need to flip to convert integer A to integer B.

EXAMPLE

Input: `29 (or: 11101), 15 (or: 01111)`

Output: `2`

Hints: #336, #369

pg 286

5.7 **Pairwise Swap:** Write a program to swap odd and even bits in an integer with as few instructions as possible (e.g., bit `0` and bit `1` are swapped, bit `2` and bit `3` are swapped, and so on).

Hints: #145, #248, #328, #355

pg 286

5.8 **Draw Line:** A monochrome screen is stored as a single array of bytes, allowing eight consecutive pixels to be stored in one byte. The screen has width `w`, where `w` is divisible by 8 (that is, no byte will be split across rows). The height of the screen, of course, can be derived from the length of the array and the width. Implement a function that draws a horizontal line from `(x1, y)` to `(x2, y)`.

The method signature should look something like:

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

Hints: #366, #381, #384, #391

pg 287

Additional Questions: Arrays and Strings (#1.1, #1.4, #1.8), Math and Logic Puzzles (#6.10), Recursion (#8.4, #8.14), Sorting and Searching (#10.7, #10.8), C++ (#12.10), Moderate Problems (#16.1, #16.7), Hard Problems (#17.1).

Hints start on page 662.