Bitwise Operator Introduction for Programming Team

(Arup Guha, University of Central Florida)

Integers and longs (long long in C++) are internally stored in binary, but we typically don't access that binary storage directly. Doing so however, can help us write faster code and helps us express ideas more concisely. Bitwise operators allow you to access the individual bits of ints and longs.

Both Java and C++ use two's complement, so it's typically best not to use the most significant bit. For programming team questions, it's rarely the case that the most significant bit needs to be used. Thus, if one needs 31 or fewer bits of storage, an int will suffice, and a long can handle up to 63 bits of storage.

The most common use of bitwise operators is to allow a single integer to store multiple pieces of information. Consider the following example that doesn't use bitwise operators:

To store a single location in a grid with *r* rows and *c* columns, we might store the location in row *i*, column *j* as the integer $c*i + j$. If this value were stored in an integer code, we could recover the row and column it represents with the expressions *code/c* and *code%c*, respectively.

Using the same general idea, we can store multiple pieces of information in a single integer.

The most simple idea is simply to store a boolean array. The following boolean array

Index	0	1	2	3	4	5
Value	false	true	true	false	true	false

could be stored in binary as 010110, which has the decimal equivalent of $2^4 + 2^2 + 2^1 = 22$.

To be able to store this array in a single integer and use it effectively, we need the following tools:

1) Ability to change a single entry (bit) from true to false or false to true.
2) Ability to access the value of a single entry (bit).

In addition, we'll learn some other features that allow us to do some really nice operations, really fast with boolean arrays that are stored in single integers.

We will interpret 0 as false and 1 as true. The other nice features we'll be able to do are as follows:

Take two boolean arrays and calculate the *"and"*, *"or"*, and *"xor"* at each corresponding index, each in one simple operation.

Here is a list of the bitwise operators and what they do:

Operator	Symbol	What it does
Bitwise *and*	&	Calculates the *and* for each corresponding bit
Bitwise *or*	|	Calculates the *or* for each corresponding bit
Bitwise *xor*	^	Calculates the *xor* for each corresponding bit
Bitwise *left shift*	<<	Moves all bits to the *left* a certain number of bits, filling in the empty space with 0s
Bitwise *right shift*	>>	Moves all bits to the *right* a certain number of bits, getting rid of the right most bits.

Here is an example of each:

```
   71         →     1000111
&  45         →     0101101
--------           -----------
    5               0000101

   71         →     1000111
|  45         →     0101101
--------           -----------
  111               1101111

   71         →     1000111
^  45         →     0101101
--------           -----------
  106               1101010

71 >> 2       →     10001 (remove the last two bits)
---------
17

71 << 3       →     1000111000 (pad with 3 zeros, multiplying by 8)
---------
568
```

Skill #1 - Seeing if a particular bit is on

Given an integer mask, the way we determine if the *k*th bit (The bit storing the value 2^k) is on or not is as follows:

```
if ((mask & (1 << k)) != 0)
    // kth bit is on
else
    // kth bit is off
```

The reason this works is that `1<<k`, is the number 1 left-shifted by *k* bits, which, in binary, is simply a 1 followed by *k* zeros. Numerically it's equal to 2^k (unless *k* represents the most significant bit). If we do a bitwise *and* with a binary number with only 1 bit on, all of the other bits in the answer are guaranteed to be 0. The 1 shines a "flashlight" on the *k*th bit only. If that *k*th bit is 1, then the bit will be on in the answer of the *and*, and if it's off, it will be off. Consider this example with mask = 71 and *k* = 6, with the flashlight in yellow:

```
   71          →      1000111
&  (1<<6)      →      1000000
--------              -----------
   64                 1000000
```

Notice that the bits in green are guaranteed to be 0 by the way we did the bitwise *and*. Thus the final answer will always either be 0, or a power of 2.

Now that we know this, we can write a simple function which counts how many of the *n* least significant bits are on:

```
int numBitsOn(int mask, int n) {
    int res = 0;
    for (int i = 0; i < n; i++)
        if ((mask & (1 << i)) != 0)
            res++;
    return res;
}
```

Skill #2 - Turning a bit on

To turn the *k*th bit of mask on, we can do: `mask |= (1 << k);`

When we do the *or*, we are guaranteed to potentially be turning a bit on and since we are *or*ing with 1 left shifted by *k* bits, the only bit we could be turning on in mask is the *k*th one.

Skill #3 - Flipping a bit

To flip/toggle the value of the k^{th} bit in mask, do: `mask ^= (1 << k);`

Since we are *xor*ing with 1 *left shifted k* bits, we are guaranteed to be flipping only one bit (from 1 to 0 or from 0 to 1), i.e., only the k^{th} bit in mask changes and the remaining bits keep their values.

Example #1 - Collecting a set of items

Let's say we are collecting a set of items and in each bag we get a subset of those items.
Let's say we want to keep on collecting until we get each item at least once.

In this sample problem, let the first line of input have two integers, *n* and *k*, representing the number of bags of items, and the number of items in the collection. Assume the items are labeled from 0 to *k*-1, with bounds $n < 1000$, $k < 21$.

This is followed by *n* input lines. The i^{th} line contains information about the i^{th} bag of items. The first value on i^{th} of these lines is how many items are in the i^{th} bag. This is followed by that many distinct integers in the range 0 to *k*-1.

The goal is to determine the minimum integer *x* such that the first *x* bags contain at least one of each item.

Here is a short sample input:

```
10 6
2 0 1
3 0 1 4
2 1 5
3 0 2 5
2 3 5
1 0
2 1 2
3 3 4 5
4 1 2 3 4
2 0 4
```

In this sample input, the first bag has items 0 and 1. The second bag has items 0, 1 and 4. The third bag has items 1 and 5, etc.

Here is a solution to the problem (in a C code segment):

```
int n, k;
scanf("%d%d", &n, &k);
int res = 0, have = 0;
for (int i = 0; i < n; i++) {
    int count, item, bagMask = 0;
    scanf("%d", &count);
    for (int j = 0; j < count; j++) {
        scanf("%d", &item);
        bagMask |= (1 << item);
    }
    have |= bagMask;
    if (res == 0 && have == ((1 << k) - 1))
        res = i + 1;
}
printf("%d", res);
```

We can check that we have all items by seeing if our "boolean array" has all trues in the first *k* entries. The corresponding binary value where the first *k* bits are 1 is $2^k - 1$, which, using bitwise operators is represented as ((1<<k) - 1).

Then, to keep track of everything we already have, we just do a bitwise *or* with our set of current items and the new bag.

Example #2 - Grading a True/False Exam

Let's say we have a T/F exam with *q* questions ($q < 32$), then we can store the answer key in a single integer. Let's say our problem is that we're given an answer key and *n* students to grade. Let the first line of input be the answer key, a string of T and F characters. Let the second line of input be a positive integer *n*, the number of exams to grade. Let the following *n* lines simply be the answers of each of the *n* students. The problem is to output the number of questions each student got correct, one by one. Here is a sample input file:

```
FFTFTFTTTT
5
FTFTTTTTTT
TTTTTTTTTT
FFFFFFFFFF
FTFTFTFTFT
TTFTTFTTFT
```

Here is a Java main method (and two supporting methods) that solve the problem:

```
public static void main(String[] args) {
    Scanner stdin = new Scanner(System.in);
    String key = stdin.next();
    int numQ = key.length();
    int quizAns = convert(key);
    int n = stdin.nextInt();
    for (int loop = 0; loop < n; loop++) {
        int student = convert(stdin.next());
        int wrong = numBitsOn(student ^ quizAns, numQ);
        System.out.println(numQ - wrong);
    }
}

public static int convert(String key) {
    int res = 0;
    for (int i = 0; i < key.length(); i++)
        if (key.charAt(i) == 'T')
            res |= (1 << i);
    return res;
}

public static int numBitsOn(int x, int len) {
    int res = 0;
    for (int i = 0; i < len; i++)
       if ( (x & (1 << i)) != 0)
            res++;
    return res;
}
```

The bitwise *xor* does the whole grading! Each bit on in the *xor* represents a bit location where the answer key and student disagreed, which means that the student got that question wrong. We can count how many bits are on in this *xor* and that is how many questions the student answered wrong, so we can score their exam!

The next two examples will discuss keeping track of a "state" in a search space involving subsets of a breadth first search.

<u>Example #3 - 2010 MCPC Problem: Quick Search</u>

Consider the problem Quick Search from the 2010 Mid-Central Programming Contest (full text on Google Drive). Here is a quick summary:

A graph is given with up to 10 vertices.
An integer, *k*, is given, up to 4, representing the # of police officers

The goal of the problem is to determine the minimum amount of time it would take all the police officers, working collectively, to search each vertex. At time $t = 0$, all the police officers start at vertex 0. In a single unit of time, each police officer can either (a) stay put, or (b) move to another location via an existing edge.

Rather than code this recursively, we should keep track of the shortest time we can arrive at a particular "state." In this case, all we care about is (a) which subset of vertices is visited, and (b) which vertex each cop is at right now. We need 10 bits to store the subset of visited vertices. For each of the 4 cops, we can store their location using 4 bits. We should always store the cops in numeric order to avoid redundant states. Thus, 26 bits total can be used to store the complete relevant state for this problem. In our solution, we would create a HashMap, where each state is mapped to the minimum amount of time it takes to achieve that state. Consider the following 26 bit number:

1001 0110 0101 0011 1001101001

This indicates that one cop is at vertex 3 (red), another cop is at vertex 5 (magenta), a third cop is at vertex 6 (blue) and the last cop is at vertex 9 (yellow). The subset of vertices visited by all the cops is 0, 3, 5, 6 and 9. (I envisioned a situation where all 4 cops started at vertex 0 and in 1 time step moved to 3, 5, 6 and 9, respectively.)

Once you decide how to store a state in bits, then you just need to use the bitwise operators to (a) know how to extract the relevant pieces from a bitmask, and (b) create the appropriate integer based on the problem.

In this example, here is how we can extract the information.
Let *mask* equal the state we are looking at, *n* is the number of vertices and *k* is the # of cops.

```
int subset = (mask & ((1 << n) - 1));
ArrayList<Integer> copLocs = new ArrayList<Integer>();
int tmp = (mask >> 10);
for (int i = 0; i < k; i++) {
     copLocs.add(tmp & 15);
     tmp >>= 4;
}
```

At first, we extract the subset of vertices visited by *and*ing the mask with the mask of the least *n* significant digits filled in. The value with *n* ones in the least significant bits is always 2^n - 1, and this will isolate the desired value. Then to get each cop, *right shift* away the mask bits, and then you can grab each cop in backwards order. When looking at *tmp*, *and*ing with 15 reveals the last four bits.

Now, to build a mask given a subset and an array list of cop locations:

```
Collections.sort(copLocs);
int newmask = subset;
for (int i = 0;  i < (4 * k);  i += 4)
    newmask |= (copLocs.get(i) << (10 + i));
```

The basic idea here is that we start with the subset in our mask since that will occupy the correct set of bits. Then, we must add in the contribution of each cop location. First sort, so the cops are always in sorted order. Then, one by one, add them by doing a bitwise *or* and then place the bits in the right place by *left-shift*ing, first by 10 bits, then 14, then 18 and finally 22 (if *k* is 4).

So, basically the *left* and *right shift* move the bits to a different location. It's almost like you are at the end of an assembly line, but if you have to place something in location *L*, you can place it at location 0, and then *left-shift* by *L* bits:

I place stuff:	0011
Then it gets left shifted	00110000000000 (🡐 conveyor belt goes left 10 spots)
Then I *"or"* it with the temp bitmask:	1001101001
Here is the bitmask with one cop:	00111001101001
Etc.	

Example #4: Block Game (2009 SER)

This problem is essentially the Rush Hour problem, of cars stuck on a grid that is up to 6 x 6 with cars which are of sizes 1 x 2 or 1 x 3, each of which moves either horizontally or vertically. The goal is to find the fewest number of moves necessary to the designated car free from the grid.

Here, a move is sliding a car as many squares as you would like.

In the input the board can be up to 6 by 6 and there can be up to 18 cars, though this is pretty unlikely since the cars wouldn't be able to move at all if we had 18 cars of size 2!

First, let's consider how many bits it might take to store a car.

A car of length 2 can really occupy only 5 spaces, because it's doomed to stay in its designated row or column. Consider the car A:

```
......     ......     ......     ......     ......
......     ......     ......     ......     ......
......     ......     ......     ......     ......
......     ......     ......     ......     ......
AA....     .AA...     ..AA..     ...AA.     ....AA
......     ......     ......     ......     ......
```

Thus, it's good enough to use 3 bits to store the location of any car. The bits can simply store the car's current starting row or column, in between 0 and 4, inclusive. Globally, you can store for each car, which direction it's facing (H/V) and what its fixed row or column value is.

So, let's say we use the 3 least significant bits to store where Car A currently is. Then the five positions shown above would have values 0, 1, 2, 3, and 4, respectively.

In this way, we can use a maximum of 54 bits (18 cars * 3 bits per car) to signify the locations of every car in the diagram, where the first letter in alphabetic order uses the 3 least significant bits, and the next letter the next 3 bits and so forth.

Consider the first sample input:

```
C
..AB..
..AB..
CCAB..
......
.DDEE.
......
```

Car A is in row 0, Car B is also in row 0, Car C is in column 0, Car D is in column 1 and Car E is in column 3. Thus the corresponding bitmask for this position (the starting position) is:

011 001 000 000 000

Let's move D to the left by 1 and E to the right by 1:

```
..AB..
..AB..
CCAB..
......
DD..EE
......
```

Now, D is in column 0 and E in column 4, the corresponding mask, which we can achieve in 2 moves is:

100 000 000 000 000

Next, move down both cars A and B as much as possible, so in four moves we get to this board position and mask:

```
......
......
CC....
..AB..
DDABEE
..AB..
```

100 000 000 011 011

On the next move, we can get car C out!

Portions of Nadeem's Handout from First Lecture in Fall

```
// Check if the bit at index pos is on in mask.
boolean on(int mask, int pos) {
   return (mask & (1 << pos)) > 0;
}

// Set the bit at index pos in mask.
int set(int mask, int pos) {
   return mask | (1 << pos);
}
```

Generating subsets reduces to a single loop:

```
void subsets() {
   // mask will iterate through all 2^N subsets.
   for (int mask = 0; mask < (1 << N); mask++) {
      // Do something problem-specific with the subset.
      // Here I'll just print it.
      for (int k = 0; k < N; k++) {
         if (on(mask, k)) {
            System.out.printf("%d ", L[k]);
         }
      }
      System.out.println();
   }
}
```

As they say on the internet (my native land), bit magic is best magic. Won't be the last example of bit-twiddling you'll see in this business.