BIT MANIPULATION

Doing ordinary mathematic operations such as 132+287 or 564/21 may seem pretty easy. But how exactly does a computer do it? The computer doesn't know what the number 21 is. It recognizes 21 as 00010101. Every operation performed, is done using binary numbers. If you don't already know how to convert binary to decimal and vice versa, I highly recommend you to read this before proceeding any further: http://www.purplemath.com/modules/numbbase.htm

In the binary representation of 21 i.e. 00010101, each of the 1s and 0s are called as bits. A set of 8 bits is called a 'Byte'. But as you know, an int data type occupies a total of 4 bytes. Therefore, the number 21 is stored in the computer as:

00000000 00000000 00000000 00010101

The negative of a number is represented by writing its 2s complement. 2s compliment of a number is the number's 1s compliment plus 1. Finding a number's 1s compliment is easy, just flip all the numbers in the binary representation of that number. Here's an example for a better understanding,

The number 12 is written as

00000000 00000000 00000000 00001100

12's 1s compliment is found by flipping all the bits

11111111 11111111 11111111 11110011

Adding 1 to that will make it

11111111 11111111 11111111 11110100

which is the 2s compliment of 12 and hence the above number is -12.

It's important to note that all negative numbers have their first bit as 1. Always.

From now on, I'll represent all the numbers in just 1 byte. So, -12 is: 11110100

BIT OPERATIONS:

AND **&** : If both the bits are 1, only then the output is 1.

OR **|** : If at least one of the bits 1, then the output is 1.

NOT **~** : The opposite of the input bit.

XOR **^** : If both inputs are same, then output is 0. If they're different, the output is 1.

The computers perform addition and subtraction a lot faster than multiplication and division. And dealing with bits, is a whole lot faster than dealing with numbers.

Now we'll move on to the real part of this task. There are three types of shift operators:

1. Arithmetic Left Shift   2. Arithmetic Right Shift   3. Logical Right Shift

ARITHMETIC LEFT SHIFT (<<)

Say you have a number x = 20. Its binary representation is 00010100. If I perform a left shift by 1,it means that all the bits will be shifted by one position to the left. So our resultant will be 00101000. An additional 0 will be added to the right most bit (The one which is underlined).

If you find the decimal value of the new number, its value is 40. Now I'll left shift it by 1 again. The resultant is now 01010000. Again like before, the new bit that is added is 0. This new number in decimal value is 80.

If I left shift again, the value becomes 10100000. But this number is negative, since the first bit is 1. Of course this won't happen when you're dealing with the number in 4 bytes. The number will be 00000000 00000000 00000000 10100000 which is equal to 160 and its positive.

Summarizing the whole of what I said above, left shifting a number by 1 will double its value.

How is this done in programming?

```
int x = 3;

int y = x<<1;   //This means y is assigned the value of x*2…. y = 6

int z = x<<2;   //This means the value of z is x*4  …… z = 12

int w = y<<x;  //The bits of y are shifted by x positions to the left and assigned to w…… w = 128
```

ARITHMETIC RIGHT SHIFT (>>)

We'll take the same example as above. Let x = 20. x in binary is 00010100. If I shift the bits to the right by 1 bit, then the new number will be 00001010. The new bit that is added to the first bit will be 0 because the bit in the first position was 0 initially itself. If I right shift the number 11011010, the new number will be 11101101. Here, the new bit is 1 because there was a 1 in the first bit before shifting.

The syntax is quite similar to the previous one.

```
int x = 13;

int y = x>>1;  //Pushes all the bits of the number x to the right by 1 position and assigns it to y
```

Right shifting a number to the right by 1 position will half its value. (13/2 = 6 as it performs integer division.)

LOGICAL RIGHT SHIFT (>>>)   (This operator is not there in C or C++)

It is very similar to the right shift, the only difference is that the new incoming bit will always be 0

Eg: if we have the number as x = 11101011

x>>1 will result in $\underline{1}1110101$

x>>>1 will result in $\underline{0}1110101$

This isn't used much. Its main purpose is when using and implement Maps and Sets. But all that has been taken care of by your compiler. Just know the difference between Logical right shift and arithmetic right shift.

Problems:

1. int x = 5, y = 6;
   int z = x<<(y>>1);
   What is the value of z?

2. int p = 1024;
   for(int i=0;i<8;i++)
           p >>=1;
   What is the value of p?
   (Just like how z += 9 is the same as z = z+9,   even x<<=7 is the same as x = x<<7)

3. As I mentioned earlier, multiplication and division is kind of slower than addition and subtraction. Given a value x, how will you find 14*x ?
   (Don't use a for loop to add 'x' 14 times   -__-)

The answers are at the end of the document.

<u>Summary:</u>

a<<b will result in $a*2^b$

a>>b will result in $a/2^b$

Now, the real part of this topic. Dealing with bits with bit operators.

This is not hard to learn and understand:

Say I have x = 89 and y = 187. Look at the next few lines and try to figure out what is happening.

x & y

```
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 1 0 1 1 0 0 1      = 89
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  1 0 1 1 1 0 1 1      = 187
---------------------------------------------------------------------------
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 1 1 0 0 1      = 25
```

x | y

```
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 1 0 1 1 0 0 1      = 89
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  1 0 1 1 1 0 1 1      = 187
---------------------------------------------------------------------------
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  1 1 1 1 1 0 1 1      = 251
```

x ^ y

```
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 1 0 1 1 0 0 1      = 89
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  1 0 1 1 1 0 1 1      = 187
---------------------------------------------------------------------------
 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  1 1 1 0 0 0 1 0      = 226
```

Forget the end result. Don't bother about the number. I want you to notice how each operation, AND, OR and XOR work. If an AND operation is performed, then each bit of one number is ANDed (I don't even know if it's a word) with the corresponding bit of the next number. So the first bit of each of the numbers go through an AND operation, then the second bit of each number goes through an AND operation and so on…you get the idea. Similarly, the same thing happens with OR and XOR operations.

I'll illustrate a few problems with each of AND, OR, XOR and NOT so that you know why they're important.

AND:

Q. Without using the modulo operator (%) determine if the number is odd or even.

A. If the number is x, perform x&1. If the value is 0, then it is even. If the value is 1, then it is odd.

Take x = 78. I'll illustrate it with just two bytes.

```
 00000000 01001110   =  78
 00000000 00000001   =  1
-------------------
 00000000 00000000   =  78 & 1 = 0
```

Take x = 43

```
 00000000 00101011   =  43
 00000000 00000001   =  1
-------------------
 00000000 00000001   =  43 & 1 = 1
```

Q. How will you check if the ith bit of a number (from the right) is 1 or not?

A. Let the number be x. We create another number and set its ith bit to 1. How do we do that?

Shift the number 1 to the left by i positions. AND the original number by the number obtained after the shift. If the value is non zero, then there is a 1 in that position. Else, there is a 0 in that position.

Let x = 63. We'll check if the 3rd bit from the right is a one or not.

```
 00000000 00111111   =  63
 00000000 00001000   =  1<<3
-------------------
 00000000 00001000  != 0      So, the number 63 has its 3rd  bit as 1.
```

NOTE::: The rightmost bit is not the 1st bit. It's the **0th** bit.

The above method can be used to check if the ith is 0 or not as well. If the resultant is 0 then the ith bit is zero, else it is 1.

OR

Q. Given a number, set its ith bit to 1.

A. This is similar to the previous one. Here, we again shift 1 by i positions to the left. We then OR the 2 number. Then the resultant is the original number with its ith bit set to 1.

Let x = 23. We want to set its 5th bit to 1.

```
 00000000 00010111   =  23
 00000000 00100000   =  1<<5
 00000000 00110111   =  55
```

In programming, it is done as follows:

```
x = x|(1<<i);    or   x |= (1<<i);
```

If the numbers ith bit is already 1, then nothing will happen to that number.

NOT

Q. Given a number x, clear its ith bit, i.e. set its ith bit to 0.

A. AND the number with 1, at all positions except the ith position. AND the ith bit with 0.

This is slightly trickier. But same as the previous 2 questions. Shift 1 by i bits to the left. Then NOT that number.

Say I want to clear the 4th bit of 91.

```
 00000000 01011011   =  91
 11111111 11101111   =  ~(1<<4)
 00000000 01001011      91 with its 4th bit cleared to zero is 75
```

Guys, if this is getting even slightly difficult to understand, please be sure to PM me.

The last three questions showed you how to check, set and clear the ith bit. The numbers like 1<<i that we used are called as MASKS. More specifically, they are called BIT Masks.

To summarize the last three problems BIT masks can be used for checking, setting and clearing the required position of a number.

Next page is the last one….. :P

XOR

This is my personal favourite bit operator ^_^

There aren't many problems that I can share….but here is one..

Q. Count the total number of positions where two number differ in their binary representation.

A. XOR both the numbers. The number of 1s that are present in the result is the number of positions they differ in.

To check how many places the number 10010011 00101101 and 00110111 00010111 differ at,

```
10010011 00101101
00110111 00010111
10100100 00111010   This is the result of the two numbers XORed
```

There are some properties that you need to know about XOR.

1. X ^ X = 0
2. X ^ 0 = X
3. XOR is commutative
4. XOR is associative
5. XOR is not distributive
6. If X ^ Y = Z   then X ^ Z = Y and Y ^ Z = X

My friend Zoheb who is also in IECSE told me a few cool things about XOR. You guys know that each character has an ASCII value right? Just like how 'A' is 65 and 'g' is 103….

If you XOR each character with the number 32, then you'll just toggle its case!

That is, if you XOR 'a' with 32, you get 65 which if typecasted to a (char) you get 'A'
Similarly,   (char)('P'^32) will give you 'p'.

This is a lot easier than having to remember the range of ASCII values of lowercase and uppercase characters. You can easily capitalize the first character of a string, or convert the entire string from Uppercase to lowercase or vice versa.

Another thing he told me about is swapping variables. Most of you were asked how to swap the values of two variables a and b without the use of a third variable in your WC interviews… The normal answer is   a = a+b; b = a-b; a = a-b;   You can swap the numbers by doing this as well:

a = a^b;   b = a^b; a = a^b.

I guess that's it..........PM if you have any doubts….

Answers of the pehle waale questions: 1. 40   2. 4   3. (x<<4) – (x<<1);