Bit Manipulation

Introduction

The technique of applying logical operations on a series of bits to produce the desired outcome is known as bit manipulation. It is the act of altering bits or other units of data shorter than a word using algorithms. Because bit manipulations are executed in parallel, they can eliminate or decrease the requirement to loop through a data structure, thereby increasing your speed to solve a problem. It helps us to make the code shorter and also faster.

The problems answered by these patterns employ a wide range of algorithmic strategies that you will come across daily. This topic is sometimes considered a magical topic because sometimes it magically solves our problem in just a few lines. This is one of the most important topics related to competitive programming. So let's look at how to use this powerful tool to solve problems.

Bitwise Operators

Operators are the most crucial tool for manipulation. This statement holds even in the case of bitwise operators. There are various bitwise operators which we use for bit manipulation. Bit operations are faster than arithmetic operations and can be used to optimize the time complexity. Let’s look at some essential bitwise operations.

NOT (**~**): Bitwise NOT is a unary operator that flips the bits of the number. It converts 1s into 0s and vice-versa.

AND (**&**): Bitwise AND is a binary operator. According to its property, If both the bits on which the operation is operated are set(1), then the resultant bit will also be set; otherwise, it is 0.

Some important properties of AND:

- A&0 = 0
- A&A = A
- A&B <= min(A,B)

OR (**|**): Bitwise OR is also a binary operator, just like AND operator. But the difference is that the resulting bit is set when any of the two-bit is set(1); otherwise, it is0.
Some important properties of OR:

- A|0 = A
- A|A = A
- A|B >= max(A,B)

XOR (**^**): Bitwise XOR is also the binary operator. According to its property, If both the bits are opposite, the resultant bit is set(1), otherwise 0.

Some important properties of XOR:

- A^0 = A
- A^A = 0
- A^B >= abs(A-B).
- If A^B = C then, C^A=B and C^B=A. (They form a triplet)

Table of Bitwise operators.

A	B	A&B (AND)	A|B (OR)	A^B (XOR)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Left Shift (**<<**): Left shift operator is a binary operator which shifts some number of bits, in the given bit pattern, to the left and appends 0 at the end. Left shift is equivalent to multiplying the bit pattern with 2^k
(k is the number of bits on which the operator is operated).

Right Shift (**>>**): Right shift operator is a binary operator which shifts some number of bits, in the given bit pattern, to the right and appends 1 at the end. The right shift is equivalent to dividing the bit pattern with 2^k (k is the number of bits on which the operator is operated).

Some Important Points to remember

- ### Check whether the k^{th} bit is set or not.

  Let's take an integer **N**; we have to check whether the **kth** bit of the number is set. We can solve it using the following condition.
  (N >> (k - 1)) & 1) if this returns 1 it means the bit is set otherwise the bit is unset.
  In this method, we simply right shift **N** by k-1 and now perform **&** operation with **1.** If the last bit(the kth bit before shifting) is set, then this will return **1; otherwise,** it will return **0**.

- ### Check whether the number is odd or even.

  Let's take an integer **N;** we have to determine whether **N** is odd or even. To determine the parity we just have to take the last bit into consideration because if the last bit is set, then the number is odd, otherwise, it's even. This is because if the last bit is set it means that the number is not divisible by **2**.
  To check this, we can simply perform **&** operation between **N** and **1.** If the result is **1,** it means that the last bit is set and the number is odd; otherwise, it's even.

- ## Check whether the number is a power of 2.

  Let's take an integer **N** now we **N** is a power of **2,** then all the bits of **N** is zero except the first one (i.e, $4_{10}=100_2$, $8_{10}=1000_2$ ...) and if we subtract a number which is a power of **2**, by **1** then all unset bits after the only set bit become set (i.e, $3_{10}= 011_2$, $7_{10}= 0111_2$ ...). From this fact, we can conclude that if we perform bitwise **&** between **N** and **N-1** and the result comes out to be **0,** then **N** is the power of **2;** otherwise, it is not.

- ## Count the number of set bits.

  There are mainly 2 methods to count the number of set bits. First is the simple brute-force approach. In this method, we just go through all the bits, and if the bit is set, we increment our answer. The second approach is based on Brian Kernighan's Algorithm, According to which Subtracting **1** from a decimal number flips all the bits after the rightmost set bit (which is 1), including the rightmost set bit. So if we subtract a number by **1** and do it bitwise **&** with itself **(N & (N-1))**, we unset the rightmost set bit. If we do **N & (N-1)** in a loop and count the number of times the loop executes, we get the set bit count. Because after every execution, the rightmost set bit becomes unset eventually. After some executions (i.e., equal to the number of set bits) the **N** becomes zero (when no set bit is left). The number of set bits is equal to the number of times the loop executes.

  The flow of the above algorithm:

```
Initialize count: = 0
If integer n is not zero
      · Do bitwise & with (n-1) and assign the value back to n
        n: = n&(n-1)
      · Increment count by 1
      · go to step 2
        Else return
        count
```

- ## A Useful relation between AND and OR operator.

  A+B = (A&B) + (A|B).

Unique number

Problem statement:
You are given an arbitrary array ‘arr’ consisting of N non-negative integers, where every element appears thrice except one. You need to find the element that occurs only once.

Approach:

- First of all, we use a variable to store the answer and unset all of its bits. Basically, initialize it with **0**.
- We go through all the bits of all the numbers one by one.
- Then we store the frequency of all the bits for all the numbers.
- Then we check whether the frequency of ith bit is divisible by **3** or not.
- If the frequency of the **i^{th}** bit is divisible by **3** then we do nothing because this bit won’t contribute to our answer.
- Else if the frequency of the bit is not divisible by **3** then we consider it and flip the **i^{th}** bit of the answer in order to obtain the number.

Code:

```
int getSingle(int arr[], int n)
{
    int result = 0;
    int x, sum;
    for (int i = 0; i < 64; i++) {

        sum = 0;

        x = (1 << i);
        for (int j = 0; j < n; j++) {
            if (arr[j] & x)
                sum++;
        }

        if ((sum % 3) != 0)
            result |= x;
```

```
    }

    return result;
}
```

Time-complexity: O(n) where n is the size of the array.
As we are going through all the numbers 64 times for every bit, the time complexity is O(64*n), which can be considered O(n).

Space complexity: *O(1)*
As we're just using extra space as variables, therefore, the space-complexity is constant.

Missing and repeating number

Problem statement:
You are given an array 'nums' consisting of first **N** positive integers. But from the **N** integers, one of the integers occurs twice in the array, and one of the integers is missing. You need to determine the repeating and the missing integer.

Approach:

- First of all, we store the bitwise xor of all the elements of the array.
- Then, we take the bitwise xor of all the numbers from **1** to **N.**
- Then, we take the bitwise xor of the above 2 results; this will give us the bitwise xor of the missing and the repeating number as a result. Because all the other numbers cancel out each other **(a^a=0 & a^0=a)** as they occur two times in the whole xor process, but the missing number occurs only once and the repeating number thrice.
- We have bitwise xor of missing and repeating numbers; now we just have to separate them. For this, we consider some set bit of the result. Because if

the **ith** bit is set, it means that the **ith** bit of missing number and the repeating number is opposite.

- Now we separate all the elements of the array and the numbers from **1** to **N** into 2 groups based on their ith bit.
- Then, we take the bitwise xor of both the groups separately, and we are now left with 2 numbers; one of them is missing, and one of them is repeating.
- Now to verify which one is missing and which one is repeating, we just traverse the array and check which one of them is not present in the array and which is present.

Code:

```
import java.util.Scanner;

public class Code{
   public static void main(String[] args) {
      int t , n;

      Scanner sc = new Scanner(System.in);

      t = sc.nextInt();

      while(t-- > 0){
         n = sc.nextInt();

         int[] ar = new int[n+1];

         for(int i=1;i<=n;i++) ar[i] = sc.nextInt();

         int resultXor = 0;

         for(int i=1;i<=n;i++) resultXor = resultXor ^ ar[i];
         for(int i=1;i<=n;i++) resultXor = resultXor ^ i;

         int setBit = resultXor & ~(resultXor - 1);

         int x = 0 , y = 0;

         for(int i=1;i<=n;i++){
            if((ar[i] & setBit) > 0){
               x = x ^ ar[i];
```

```
            }else{
                y = y ^ ar[i];
            }
        }

        for(int i=1;i<=n;i++){
            if((i & setBit) > 0){
                x = x ^ i;
            }else{
                y = y ^ i;
            }
        }

        boolean isXMissing = true;

        for(int i=1;i<=n;i++){
            if(ar[i] == x){
                isXMissing = false;
                break;
            }
        }

        if(isXMissing == true){
            System.out.println(x + " is missing number & " + y + " is repeating number");
        }else{
            System.out.println(y + " is missing number & " + x + " is repeating number");
        }
    }
  }
}
```

Time-complexity: **O(n)**, Where **n** is the total number of elements in the array.
Space-complexity: **O(1).**