Bit Manipulation - I

Prerequisites: knowledge of binary number system

Get i^{th} bit:

Mask: Right shift n 'i' times, and check the first bit.

```
int getBit(int n, int pos){
    return (n >> pos) & 1;
}
```

Set i^{th} bit:

Mask: 1 << i

Bitwise OR operation between n and mask sets the i^{th} bit to one.

```
int setBit(int n, int pos) {
    return (n | (1 << pos));
}
```

Clear i^{th} bit

Mask: ~ (1 << i)

In the mask, all the bits would be one, except the i^{th} bit. Taking bitwise AND with n would clear the i^{th} bit.

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

Toggle i^{th} bit

Mask: 1 << i

Bitwise XOR operation between n and mask toggle the i^{th} bit.

```
int toggleBit(int n, int pos) {
    return (n xor (1 << pos));
}
```

Update i'th bit to the given value

Mask: mask has all the bits set to one except i^{th} bit.

n = n & mask, i^{th} bit is cleared.

Now, to set i^{th} bit to value, we take value << pos as the mask.

```
int updateBit(int n, int pos, int value) {
    int mask = ~(1 << pos);
    n = n & mask;
    return (n | (value << pos));
}
```

Supplementary material

Compute XOR from 1 to n (direct method) :

```
int computeXOR(int n)
{
    if (n % 4 == 0)
        return n;
    if (n % 4 == 1)
        return 1;
    if (n % 4 == 2)
        return n + 1;
    else
        return 0;
}
```

Input: 6
Output: 7

Equal Sum and XOR

Problem: Given a positive integer n, find count of positive integers i such that 0 <= i <= n and n+i = n⊕i (⊕ is the XOR operation)

Instead of using looping (Brute force method), we can directly find it by a mathematical trick i.e.

Let x be the number of unset bits in the number n.

Answer = 2^x

XOR of all subsequences of an array

The answer is always 0 if the given array has more than one element. For an array with a single element, the answer is the value of the single element.

Logic: If the array has more than one element, then element occurs.

Number of leading zeros, trailing zeroes and number of 1's of a number

It can be done by using inbuilt function i.e.

Number of leading zeroes: *builtin_clz(x)*

Number of trailing zeroes: *builtin_ctz(x)*

Number of 1-bits: __*builtin_popcount(x)*

Convert binary numbers directly into a decimal integer in C++.

```
#include <iostream>
using namespace std;
 int main()
{
   int number = 0b011;
   cout << number;
   return 0;
}
```

Output: 3

Swap 2 numbers using bit operations:

```
    a ^= b;
    b ^= a;
    a ^= b;
```

Flip the bits of a number:

It can be done by a simple way, just simply subtract the number from the value obtained when all the bits are equal to 1 .

For example:

Number: Given Number

Value : A number with all bits set in a given number.

Flipped number = Value – Number.

Example :

Number = 23,

Binary form: 10111

After flipping digits number will be: 01000

Value: 11111 = 31

We can find the most significant set bit in O(1) time for a fixed size integer. For example below code is for a 32-bit integer.

```
int setBitNumber(int n)
{
   n |= n>>1;

   n |= n>>2;
    n |= n>>4;
   n |= n>>8;
   n |= n>>16;

   n = n + 1;
    return (n >> 1);
}
```

Practice questions:

1. [Reverse bits]
2. [Hamming distance]

Bit Manipulation - II

Prerequisites: knowledge of binary number system

Count set bits

n & (n - 1) sets the first set-bit to zero.

Explanation: n = XXX100

n - 1 = XXX011

n & (n - 1) = XXX000

```
int numberofones(int n) {
    int count = 0;
    while (n) {
        n = n & (n - 1);
        count++;
    }
    return count;
}
```

Power of two

From our past knowledge of the binary number system,

Numbers of the type 2^n have only 1 set bit.

Explanation: n = 000100

n - 1 = 000011

n & (n - 1) = 000000

!(n & (n - 1)) = 000001

If the number only had one set bit, then n & (n - 1) would be zero.

```
bool ispowerof2(int n) {
    return (n && !(n & n - 1));
}
```

Generate Subset

Explanation: if the j^{th} bit is set, then we take the j^{th} element.

There are a total of 2^n subsets.

```
void subsets(int arr[], int n) {
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < n; j++) {
            if ( i & (1 << j)) {
                cout << arr[j] << " ";
            }
        } cout << endl;
    }
}
```

Practice Questions:

1. Counting bits
2. Power of four

Bit Manipulation Challenges

Challenge 1

Write a program to find a unique number in an array where all numbers except one, are present twice.

Hint: A ⊕ B ⊕ B ⊕ A ⊕ C = C. All those numbers which occur twice will get nullified after ⊕ operation.

Sample Test Case:

Input:

1	2	3	4	1	2	3

Output: 4

Code

```
#include<iostream>
using namespace std;
int unique(int arr[], int n) {
    int xorsum = 0;
    for (int i = 0; i < n; i++) {
        xorsum = xorsum ^ arr[i];
    }
    return xorsum;
}
int main() {
    int arr[] = {1, 2, 3, 4, 1, 2, 3};
    cout << unique(arr, 7) << endl;
    return 0;
}
```

Challenge 2

Q2. Write a program to find 2 unique numbers in an array where all numbers except two, are present twice.

Logic

1. Take XOR of all the elements and let that xor value be x. All the repeating elements will get nullified and xor of only two unique elements will last. (as a⊕a = 0).
2. There will be at least one bit set in x. Using that set bit, divide the original set of numbers into two sets
   a. First set which contains all the elements with that bit set.
   b. Second set which contains all the elements with that bit unset.
3. Take xor of both the sets individually, and let those xor values be x1 and x2.
4. Voila, x1 and x2 are our unique numbers. Reason: both the above sets contain one of the unique elements and rest elements of the sets occur twice which will get nullified after ⊕ operation.

Sample Test Case:

Input:

2	4	6	7	4	5	6	2

Output: 5 7

Code

```cpp
#include<iostream>
using namespace std;
int setBit(int n, int pos) {
    return ((n & (1 << pos)) != 0);
}
void unique(int arr[], int n) {
    int xorsum = 0;
    for (int i = 0; i < n; i++) {
        xorsum = xorsum ^ arr[i];
    }
    int tempxor = xorsum;
    int setbit = 0;
    int pos = 0;
    while (setbit != 1) {
        setbit = xorsum & 1;
        pos++;
        xorsum = xorsum >> 1;
    }
    int newxor = 0;
    for (int i = 0; i < n; i++) {
        if (setBit(arr[i], pos - 1)) {
            newxor = newxor ^ arr[i];
        }
    }
    cout << newxor << endl;
    cout << (tempxor ^ newxor) << endl;

}
int main() {
    int arr[] = {1, 2, 3, 1, 2, 3, 5, 7};
    unique(arr, 8);
    return 0;
}
```

Challenge 3

Q3. Write a program to find a unique number in an array where all numbers except one, are present thrice.

Sample Test Case:

Input:

1	3	2	3	4	2	1	1	3	2

Output: 4

Logic

1. We will maintain an array of 64 size which will store the number of times i^{th} bit has occurred in the array.

3	6	4	0	0	0	0	0	0	0	.... 64 bits

2. Take the modulo of each element of this array with 3. Resultant array will represent the binary representation of the unique number.
3. Convert that binary number to decimal number and output it.

Code

```
#include<iostream>
using namespace std;

bool getBit(int n, int pos) {
    return ((n & (1 << pos)) != 0);
}
int setBit( int n, int pos) {
    return (n | (1 << pos));
}
int unique(int arr[], int n) {
    int result = 0;
    for (int i = 0; i < 64; i++) {
        int sum = 0;
        for (int j = 0; j < n; j++) {
            if (getBit(arr[j], i)) {
                sum++;
            }
        }
        if (sum % 3 != 0) {
            result = setBit(result, i);
        }
    }
    return result;
}
int main() {
    int arr[] = {1, 2, 3, 4, 1, 2, 3, 1, 2, 3};
    cout << unique(arr, 10) << endl;
    return 0;
}
```