BIT Manipulation

Kapil Yadav

INDEX

Bit Manipulation

&, |, ~, !, >>, <<, XOR

1. The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
5. The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
6. The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it.

int a = 5, b = 9

```
  00000101
& 00001001
  --------
  00000001  = 1
```

```
  00000101
| 00001001
  --------
  00001101  = 13
```

XOR

```
  00000101
^ 00001001
  --------
  00001100  = 12
```

Reference - GFG

https://www.linkedin.com/in/kapilyadav22

d) ~ operator :-

```
int a = 5;
```

// By default it is signed integer, stored in 2's compliment manner having range from $-(2^{n-1})$ to $+(2^{n-1}-1)$.

By default int size is 32 bits so, it will have range of -2^{32-1} to $2^{32-1}-1$

So a = 0 00000000 00000000 00000000 00000101

Sign bit → 8 zeros, 8 zeros, 8 zeros

~a = 1 11111111 11111111 11111111 11111010

NOTE :- MSB in signed integers represents the sign of the number.
0 = Positive Number, 1 negative number.

→ since after negative, MSB bit is 1, so the number is negative, Now to find the exact value of it, just do 2's complement

2's complement of ~a = 00000000 00000000 00000000 00000110
= 6

→ Since, it is negative, ~a = −6.

b) Take unsigned int a = 5

→ Unsigned int ranges from 0 to 2^n-1.

So a = 00000000 00000000 00000000 00000101

8 zeros, 8 zeros, 8 zeros

~a = 11111111 11111111 11111111 11111010

since, we are using unsigned int, so ~a will be the decimal representation of it.

~a = 4294967290,

e) left shift :-

a = 8 = 0 0 0 0 1 0 0 0

a << 3 = 0 1 0 0 0 0 0 0 = 64,

∴ left shift does nothing but multiply the number by 2, K times.

$a = a \times 2^3 = 8 \times 8 = 64.$

f) Right shift :-

a = 8 = 0 0 0 0 1 0 0 0

a >> 3 = 0 0 0 0 0 0 0 1 = 1

∴ Right shift divides the number by 2, K times, where K is the number of right shift.

$a = \frac{a}{2^k} = \frac{8}{2^3} = \frac{8}{8} = 1.$

```c
// C Program to demonstrate use of bitwise operators
#include <stdio.h>
int main()
{
    // a = 5(00000101), b = 9(00001001)
    int a = 5, b = 9;

    // The result is 00000001
    printf("a = %d, b = %d\n", a, b);
    printf("a&b = %d\n", a & b);

    // The result is 00001101
    printf("a|b = %d\n", a | b);

    // The result is 00001100
    printf("a^b = %d\n", a ^ b);

    // The result is 11111010
    printf("~a = %d\n", a = ~a);

    // The result is 00010010
    printf("b<<1 = %d\n", b << 1);

    // The result is 00000100
    printf("b>>1 = %d\n", b >> 1);

    return 0;
}

```

∴ The values will be output in 2's complement manner, since we are using %d to represent them.

Q 1.)Check if the ith bit is set or not

$N = 13 = (1101)_2$, $i = 3^{rd}$ bit

↳ 1st bit

We can take mask 0100 and perform & operation, we will get ith bit.

mask = 1<<(i-1).

```
  1101
& 0100
------
  0100  = 4
```

4 != 0, so ith bit is set,
if ans = 0, ith bit is not set.

```
1 void isKthBitSet(int n, int k)
2 {
3     int mask = 1 << (k - 1);
4     if (n & mask)
5         printf("SET");
6     else
7         printf("NOT SET");
8 }
```

Q 2.) Set the ith bit of a number.

$N = 13 = (1101)_2$, $i = 2$

Take mask = 0010 and perform Bitwise |.

mask = 1<<(i-1)

```
  1101
| 0010
------
  1111  = 15
```

```
1 int setKthBit(int n, int k)
2 {
3     int mask = 1<<k-1;
4     return (mask | n);
5 }
```

Q 3.) clear the ith bit of a number.

N = 13
= 1101 , i = 4th

mask = 1 << i-1
= 1000
~mask = 0111

= 1101
& 0111
0101 = 5

```
1 int clearBit(int n, int k)
2 {    int mask =  (1<<k-1);
3       mask = ~mask;
4       return (n & mask);
5 }
6
7 int main()
8 {
9     int n = 5, k = 1;
10    printf("%d\n", clearBit(n, k));
11    return 0;
12 }
```

Q 4.) Remove the last set bit of a number. (Rightmost bit)

N = 214
= 1 1 0 1 0 1 1 0

Output = 212

N = 1 1 0 1 0 1 1 0
& N-1 = 1 1 0 1 0 1 0 1
Ans = 1 1 0 1 0 1 0 0

① 13 = 1101
12 = & 1100
1100

② 8 = 1000
7 = 0111
0000

```
1 int fun(unsigned int n)
2 {
3     return n & (n - 1);
4 }
```

Find the position of rightmost set bit:

(n & ~(n-1) → will return the binary number containing the rightmost set bit as 1.

→ we can do log of above binary number and we will get position of rightmost set bit.

Q 5.) Find whether a number is even or odd

There are four ways to find whether a number is even or odd:

1. Using Mod Operator(%)
2. Using Division Operator(/)
3. Using Bitwise AND operator(&)
4. Using Left shift and right shift operator(<<,>>)

1. Using Mod Operator:

It is the most used method,
ex - 7 % 2 = 1] if remainder is 1 means odd.
6 % 2 = 0] if remainder is 0 means even.

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

void find_even_odd_using_mod(int num){
    if(num%2==1){
        cout<<"Number is odd.";
    }
    else {
    cout<<"Number is even.";
    }
}

int main() {
   int num;
   cin>>num;
   find_even_odd_using_mod(num);
}

```

2. Using Division Operator:

→ first divide and multiplied the number by 2, if the answer is same as number, then it is even else odd.

ex – num = 9

$\left(\frac{9}{2}\right) \times 2 = 8$

$8 \neq 9$ → odd

num = 16

$\left(\frac{16}{2}\right) \times 2 = 16$

$16 = 16$ → even

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

void find_even_odd_using_divisionandmul(int num){
    int calcnum = (num/2) * 2;
    if(calcnum==num){
        cout<<"Number is even.";
    }
    else {
    cout<<"Number is odd.";
    }
}

int main() {
   int num;
   cin>>num;
  find_even_odd_using_divisionandmul(num);
}

```

3. Using Bitwise AND Operator:

ex - num = 10

```
  1010
& 0001
------
  0000  -> even
```

num = 9

```
  1001
& 0001
------
  0001  -> odd
```

Bit 1	Bit 2	&
0	0	0
0	1	0
1	0	0
1	1	1

even : if n&1 == 0
odd : if n&1 == 1

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

void find_even_odd_using_bitwiseand(int num){

    if(num & 1){
        cout<<"Number is odd.";
    }
    else {
    cout<<"Number is even.";
    }
}

int main() {
   int num;
   cin>>num;
   find_even_odd_using_bitwiseand(num);
}
```

4.) Using Left shift and Right Shift operator (<<,>>)

→ It is same as method 2, here we will do operations on bits than number.
→ for division, use Right shift,
→ for multiplication, use left shift.

→ So combined operation will be :- (num >>1) <<1
→ Num will be even if ((num >>1)<<1) == num, else odd.

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

void find_even_odd_using_leftandright(int num){
    int calcnum = (num>>1)<<1;
     if(calcnum==num){
         cout<<"Number is even.";
     }
     else {
     cout<<"Number is odd.";
     }
}

int main() {
    int num;
    cin>>num;
    find_even_odd_using_leftandright(num);
}
```

Q 6.) Check if the number is a power of 2?

N = 8

1000

& N-1 = 0111

ans = 0000

→ if ans = 0, it is power of 2.

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

bool powerof2(int n)
{   int mask = n-1;
    return !(n & mask);
}

int main()
{
    int n;
    cin>>n;
    if(powerof2(n)){
        cout<<n<<" is power of 2";
    }
    else {
       cout<<n<<" is not a power of 2";
    }
    return 0;
}

```

Q 7.) Check if a number is a power of 4:

Approach 1:

Divide the Number by 4 untill we get 1, if the remainder after modulus is not 0, that means it is not a power of 4,
also if the remainder is 0, that doesn't mean, it is a power of 4, example - 8 is not a power of 4.
So if the remainder is 0,divide it by 4,untill n does not equals to 1 or modulus of n!=0.

$$TC = O(\log_4 N)$$

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

bool isPowerOfFour(int n){
    if(n == 0)
        return 0;
    while(n != 1)
    {
    if(n % 4 != 0)
        return 0;
        n = n / 4;
    }
    return 1;
}

int main()
{
int number;
cin>>number;
if(isPowerOfFour(number))
    cout<<number<<" is a power of 4";
else
    cout<<number<<" is not a power of 4";
}

```

Approach 2 :

If a given number is a power of 2 and its only set bit is present at even position like 0,2,4,6,8, then it will also be power of 4.
We can check power of two by doing bitwise and of N and (N-1).
To check the position of set bits, we will create a mask, now understanding the pattern of bits is really important to understand mask.

→ first, check if it is a power of 2 or not.

→ If we do bitwise AND of n with mask, and negate it, if result after negation is 1, then it is power of 4, otherwise not.

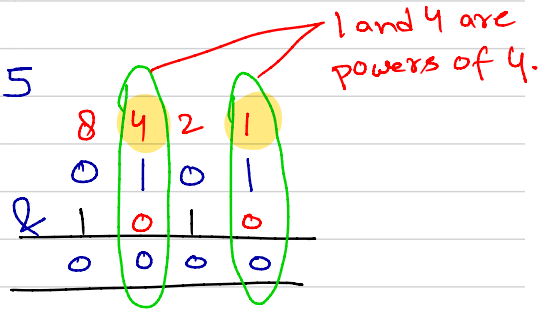
→ Now to get the result as 1, bitwise AND should be 0, means we need to create a mask such that Bitwise AND of n and mask will be 0.

→ Lets take an example : N = 16.

N = 16 ⇒

16	8	4	2	1
1	0	0	0	0
& 0	1	0	1	0
0	0	0	0	0

N = 5



→ If you clearly observe the pattern, if in mask, those who are power of 4, if we set those bits as 0, rest as 1, we can create our mask.

→ Same pattern will by followed to check power of 8 and 16.

→ Int generally is of size 4bytes = 32 bits, so if you create a mask of 32 bits, it will be

1010 1010 1010 1010 1010 1010 1010 1010
A A A A A A A A

= 0XAAAAAAAA
↳ Hexadecimal

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

bool checkPowerOf4(int n){
    if(n==0) return false;
    bool powerof2 = !(n & (n - 1));

    //for mask, bits will be 0, in the place which are power of 4
    //bits will be 1, in the place which are not power of 4
    //A = 1010, 1 and 4 are powers of 4, so mark yhem as 0.
    //every even bit will be a power of 4, so mark them as 0.
    bool mask = !(n & 0xAAAAAAAA);
    return  powerof2 && mask;
}

int main(){
     int n;
    cin>>n;
    if (checkPowerOf4(n)) {
        cout << n << " is a power of 4";
    }
    else {
        cout << n << " is not a power of 4";
    }
    return 0;
}
```

Approach 3 :

N is a power of 4, if it is a power of 2 and if modulus of n by 3 will give remainder 1.

```cpp

#include <iostream>
using namespace std;

bool checkPowerOf4(unsigned n){
    return !(n & (n - 1))&& (n % 3 == 1);
}

int main()
{
    unsigned n;
    cin>>n;

    if (checkPowerOf4(n)) {
        cout << n << " is a power of 4";
    }
    else {
        cout << n << " is not a power of 4";
    }

    return 0;
}
```

Q 8.) Check if a number is power of 8:

Approach 1:

Divide the Number by 8 untill we get 1, if the remainder after modulus is not 0, that means it is not a power of 8,
also if the remainder is 0, that doesn't mean, it is a power of 8, example - 16 is not a power of 8.
So if the remainder is 0,divide it by 8,untill n does not equals to 1 or modulus of n!=0.

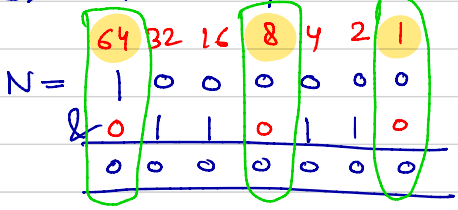
```cpp
#include<iostream>
using namespace std;

bool isPowerOfEight(int n){
    if(n == 0)
        return 0;
    while(n != 1)
    {
    if(n % 8!= 0)
        return 0;
        n = n / 8;
    }
    return 1;
}

int main()
{
  int number;
  cin>>number;
  if(isPowerOfEight(number))
      cout<<number<<" is a power of 8";
  else
      cout<<number<<" is not a power of 8";
}

```

Approach 2:

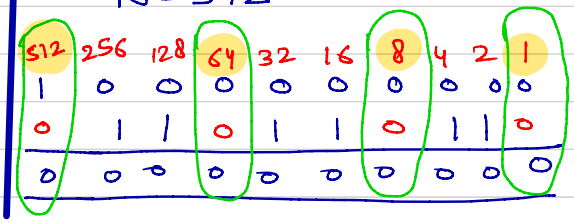
→ Same as Approach 2 of previous question, just the mask will change.

NOTE :- To create mask, those bits who are power of 8, set them as 0, rest as 1.

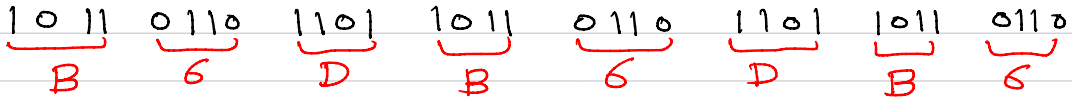
ex- N = 64



N = 512



→ If we clearly observe, we will find that every 3rd bit in the mask, starting from 0th position, is the power of 8, so mask will




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

bool checkPowerOf8(int n){
    if(n==0) return false;
    bool powerof2 = !(n & (n - 1));

    //for mask, bits will be 0, in the place which are power of 8
    //bits will be 1, in the place which are not power of 8
    //0110110, 1,8,64 are powers of 8.
    /*starting from 0th position every third bit will be a power of 8,
    so mark them as 0.
    (0xB6DB6DB6)16 = (10110110110110110110110110110)2
    */

    bool mask = !(n & 0xB6DB6DB6);
    return  powerof2 && mask;
}

int main(){
     int n;
    cin>>n;
    if (checkPowerOf8(n)) {
        cout << n << " is a power of 8";
    }
    else {
        cout << n << " is not a power of 8";
    }
    return 0;
}
```

Approach 3:

N is a power of 8, if it is a power of 2 and if modulus of n by 7 will give remainder 1.

```
1
2 #include <iostream>
3 using namespace std;
4
5 bool checkPowerOf8(unsigned n){
6     return !(n & (n - 1))&& (n % 7 == 1);
7 }
8
9 int main()
10 {
11     unsigned n;
12     cin>>n;
13
14     if (checkPowerOf8(n)) {
15         cout << n << " is a power of 8";
16     }
17     else {
18         cout << n << " is not a power of 8";
19     }
20
21     return 0;
22 }
```

Q 9.) Check if a number is a power of 16:

Approach 1:

Divide the Number by 16 untill we get 1, if the remainder after modulus is not 0, that means it is not a power of 16,
also if the remainder is 0, that doesn't mean, it is a power of 16, example - 32 is not a power of 16.
So if the remainder is 0,divide it by 16,untill n does not equals to 1 or modulus of n!=0.

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

bool isPowerOfSixteen(int n){
    if(n == 0)
        return 0;
    while(n != 1)
    {
    if(n % 16!= 0)
        return 0;
        n = n / 16;
    }
    return 1;
}

int main()
  {
  int number;
  cin>>number;
  if(isPowerOfSixteen(number))
      cout<<number<<" is a power of 16";
  else
      cout<<number<<" is not a power of 16";
  }

```

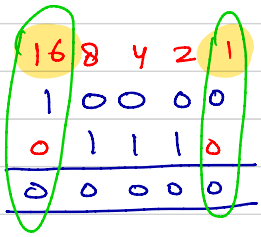
Approach 2:

Same as power of 4, power of 8, we only need to create a different mask.

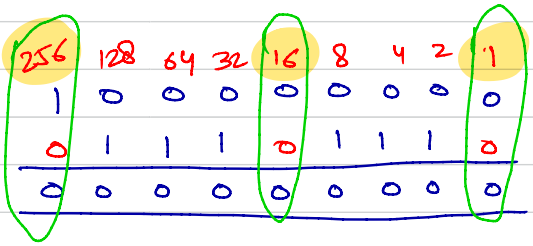
NOTE :- To create mask, those bits who are power of 16, set them as 0, rest as 1.

Ex- N = 16

N = 16



N = 256



→ If we observe, every 4th bit starting from 0th bit is power of 16, set them as 0, rest as 1.

mask = 1110 1110 1110 1110 1110 1110 1110 1110

E E E E E E E E

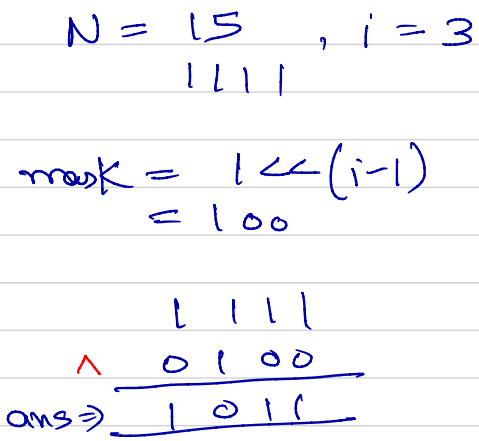
```cpp
#include <iostream>
using namespace std;

bool checkPowerOf16(int n){
    if(n==0) return false;
    bool powerof2 = !(n & (n - 1));

    //for mask, bits will be 0, in the place which are power of 16
    //bits will be 1, in the place which are not power of 16
    //0111101110, 1,16,256 are powers of 16.
    /*starting from 0th position every fourth bit will be a power of 16,
    so mark them as 0.
    (0xEEEEEEEE)16 = (11101110111011101110111011101110)2
    */

    bool mask = !(n & 0xEEEEEEEE);
    return  powerof2 && mask;
}

int main(){
     int n;
    cin>>n;
    if (checkPowerOf16(n)) {
        cout << n << " is a power of 16";
    }
    else {
        cout << n << " is not a power of 16";
    }
    return 0;
}
```

Approach 3:

N is a power of 16, if it is a power of 2 and if modulus of n by 15 will give remainder 1.

```cpp

#include <iostream>
using namespace std;

bool checkPowerOf16(unsigned n){
    return !(n & (n - 1))&& (n % 15 == 1);
}

int main()
{
    unsigned n;
    cin>>n;

    if (checkPowerOf16(n)) {
        cout << n << " is a power of 16";
    }
    else {
        cout << n << " is not a power of 16";
    }

    return 0;
}
```

Q 10.) Toggle ith Bit of a number:




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

int toggleIthBit(int n, int I)
{   int mask = 1 << (I-1);
    return (n^ mask);
}

int main()
{
    int n,I;
    cin>>n>>I;
    cout << toggleIthBit(n , I);
    return 0;
}
```

Q 11.) Count the number of set bits in a number

input = 1110
output = 3.

cnt = 0;

N = 1110
& 1
0000

N = 0111
& 1
0001 cnt = 1

N = 0011
& 1
0001 , cnt = 2

N = 0001
& 1
0001 , cnt = 3

N = 0. stop.

```
1 unsigned int
  countSetBits(unsigned int n)
2 {
3     unsigned int count = 0;
4     while (n!=0) {
5         if(n& 1==1)
6             count++;
7         n >>= 1;
8     }
9     return count;
10 }
```

TC = O(logn)

→ Second way, Better in Some cases :-

Brian Kernighan's Algorithm.

→ clear the rightmost set bit everytime, until N becomes 0.

N = 13
1101

N = N & N-1 = 1100 (1)
= 1000 (2)
= 0000 (3)

→ So we needed only 3 steps, But if N = 7, 15, it is same as previous.

```
1 unsigned int countSetBits(int
  n)
2 {
3     unsigned int count = 0;
4     while (n!=0) {
5         n &= (n - 1);
6         count++;
7     }
8     return count;
9 }
```

TC = O(No of set bits in number).

Q 12. Find the two non-repeating elements in an array of repeating elements/ Single Number III

arr = [1, 1, 2, 5, 3, 2, 3, 4, 7, 4]

Brute force :-

```
for(int i=0; i<n; i++)
{  int cnt = 0;
   for(int j=0; j<n; j++)
   {  if(a[j]==a[i]
         cnt++;
   }
   if(cnt==1)  print(a[i]);
}
```

TC = O(N²).

(2) We can use map :-

```
unordered_map<int,int> mp;
for(int i=0; i<n; i++)
{
    mp[a[i]]++;
}
for(auto it : mp)
{ if( it.second ==1)
      print(it.first);
}
```

TC = O(N).
SC = O(N).

(3.) XOR :-

XOR = 5 ^ 7.

```
  1 0 1
  1 1 1
  -----
  0 1 0
```

second bit is 1, means it is differ in both the numbers, so if i take a rightmost set bit and create two sets

2nd bit is 0 (y)	2nd bit is 1 (x)
~~1~~	~~2~~
~~5~~	~~2~~
~~1~~	7
~~4~~	~~3~~
~~4~~	~~3~~

x = 7, result = 2.
so, result ^ x will give y.

y = (x ^ result) = 7 ^ 2
= 5,
so, x = 7, y = 5.

```cpp
vector<int> singleNumber(vector<int>& nums) {
        long long result=0;
        for(int i=0;i<nums.size();i++) {
            result=result^nums[i];
        }

        long long mask = (result & ~(result - 1));
       //simple way to find rightmost set bit.


        int x=0;
        int y=0;

        for(int i=0;i<nums.size();i++)
        {
            if((nums[i]&mask)>0)
            {
                x^=nums[i];
            }
        }
        y=result^x;

    return vector<int>{x, y};
    }
```

= 2

x = 2 ^ 2 ^ 7 ^ 3 ^ 3
x = 7.

⇒ 2 ^ 7 = 5

TC = O(N) + O(N).

Given an integer array nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in any order.

You must write an algorithm that runs in linear runtime complexity and uses only constant extra space.

Alphabetical Letters Operations

1. Convert Uppercase to LowerCase:

→ We can convert uppercase to lowercase by doing Bitwise OR and space.
(ch | ' ')

```cpp
void uppertolower(char ch){
    cout <<char(ch | ' ');
}

int main() {
    char ch;
    cin>>ch;
    uppertolower(ch);
}
```

2. Convert Lowercase to Uppercase:

→ We can convert a lowercase to uppercase by taking bitwise AND with underscore.
(ch & '_')

```cpp
void lowertoupper(char ch){
    cout <<char(ch & '_');
}

int main() {
    char ch;
    cin>>ch;
    lowertoupper(ch);
}
```

3. Invert Alphabet's Case

→ We can invert the alphabet case from uppercase to lower or vice versa using Bitwise XOR(^) with space.

`(ch ^ ' ')`

NOTE:- Don't forget to typecast it in char.

→ The ASCII code of space(' ') is 0 0 1 0 0 0 0 0. 32
→ The ASCII code of underscore is 0 1 0 1 1 1 1 1

Explaination of above:-

→ Bitwise OR of an uppercase character with ' ', will set the third significant bit and we will get lowercase character.

→ Bitwise AND of a lowercase character with '_', will unset/clear the third bit and we will get uppercase character.

→ Bitwise XOR of an uppercase or lowercase characters with ' ' (space), will toggle the third significant bit.
Uppercase becomes lowercase and vice versa.

```cpp
void invertcase(char ch){
    cout << char(ch ^ ' ');
}

int main() {
    char ch;
    cin>>ch;
    invertcase(ch);
}
```

4. Find Letter Position in alphabet:

→ We can find letter position by taking bitwise AND with ASCI code 31 (00011111 in binary).

('A' & 31) ⇒ 1
('c' & 31) ⇒ 3 } case will not matter here

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

void findalphabeticalposition(char ch){
    cout <<(ch & 31);
}

int main() {
    char ch;
    cin>>ch;
    findalphabeticalposition(ch);
}

```

The left shift and right shift operators should not be used for negative numbers

The bitwise XOR operator is the most useful operator from a technical interview perspective.

Let's solve some problems related to XOR:

Q .1) Given a set of numbers where all elements occur an even number of times except one number, find the odd occurring number.

arr[] = {4,2,2,1,5,6,7,5,6,7,4}.

$= 4 \wedge 2$

$= 4 \wedge 2 \wedge 2 = 4$

$= 4 \wedge 1$

$= 4 \wedge 1 \wedge 5$

$= 4 \wedge 1 \wedge 5 \wedge 6$

$= 4 \wedge 1 \wedge 5 \wedge 6 \wedge 7$

$= 4 \wedge 1 \wedge 5 \wedge 6 \wedge 7 \wedge 5$

$= 4 \wedge 1 \wedge 6 \wedge 7 \wedge 6$

$= 4 \wedge 1 \wedge 7 \wedge 7$

$= 4 \wedge 1 \wedge 4$

$= 1.$

```
1 int getOddOccurrence(int arr[], int arsize)
2 {
3     int res = 0;
4     for (int i = 0; i < arsize; i++)
5         res = res ^ arr[i];
6
7     return res;
8 }
```

Q 2) Swap two numbers using Bit manipulation:

$a = 5, \quad b = 7$

1) $a = a \wedge b$
2) $b = a \wedge b$
3) $a = a \wedge b$

$a = 5 \wedge 7$

$b = 5 \wedge 7 \wedge 7 = 5$

$a = 5 \wedge 7 \wedge 5$

$= 7$

Q 3.) Calculate XOR from 1 to n

$n = 5$

output $= 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 = 1$

∴ We can run for loop n times. TC = O(N). ✗

n	XOR of n
1	= 1
2	= 1 ∧ 2 = 3
3	= 3 ∧ 3 = 0
4	= 0 ∧ 4 = 4
5	= 4 ∧ 5 = 1
6	= 1 ∧ 6 = 7
7	= 7 ∧ 7 = 0
8	= 0 ∧ 8 = 8

→ if (n % 4 == 0) return n;
→ if (n % 4 == 1) return 1;
→ if (n % 4 == 2) return n+1;
→ if (n % 4 == 3) return 0;

```cpp
int computeXOR(int n)
{
  // If n is a multiple of 4
  if (n % 4 == 0)
    return n;

  // If n%4 gives remainder 1
  if (n % 4 == 1)
    return 1;

  // If n%4 gives remainder 2
  if (n % 4 == 2)
    return n + 1;

  // If n%4 gives remainder 3
  return 0;
}
```

Q 4.) Find XOR of numbers frokm the range [L,R]

Input $L = 4, R = 8$
Output $\Rightarrow 4 \wedge 5 \wedge 6 \wedge 7 \wedge 8 = 8$

$\therefore$ Run loop from L to R. TC = O(N) ✗

→ findXOR(L-1) → use above logic = O(1).
→ findXOR(R) → use above logic = O(1).

$\Rightarrow$ findXOR(L-1) ^ findXOR(R)

$\therefore (\cancel{1 \wedge 2 \wedge 3}) \wedge (\cancel{1 \wedge 2 \wedge 3 \wedge 4} \wedge 5 \wedge 6 \wedge 7 \wedge 8)$
$= 4 \wedge 5 \wedge 6 \wedge 7 \wedge 8 = 8.$

```cpp
int findXOR(int n)
{
    int mod = n % 4;
    if (mod == 0)
        return n;

    if (mod == 1)
        return 1;
    if (mod == 2)
        return n + 1;
    if (mod == 3)
        return 0;
}

int findXOR(int l, int r) {
    return (findXOR(l - 1) ^ findXOR(r));
}
```

Q 5.) Check whether the number is even or not

→ one way to find is :-

```
if(number % 2 == 0)
    return 0; //even
else
    return 1; //odd
```

→ using & operator

```
if(number & 1 == 0)
    return 0; //even
else
    return 1; //odd
```

∴ Using & takes less time.

Q 6.) Find the XOR of the XOR of all subsets of an array: $= 0$.

arr = [1,3,2]

{ } → 0
{1} → 1
{3} → 3
{2} → 2
{1,3} → 2
{1,2} → 3
{2,3} → 1
{1,2,3} → 0

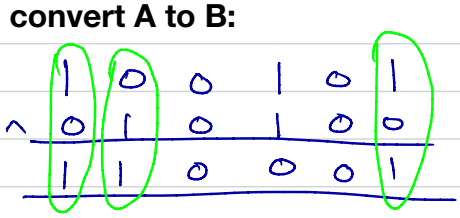
0

→ XOR will be non zero, only if there is only one element in the array. return arr[0] in that case.

```
int findXOR(int Set[], int n)
{
  if (n == 1)
    return Set[0];
  else
    return 0;
}

int main()
{
  int Set[] = { 1, 2, 3 };
  int n = sizeof(Set) / sizeof(Set[0]);
  printf("XOR of XORs of all subsets is %d\n",
   findXOR(Set, n));
  return 0;
}
```

Q 7.) Count Number of bits to be flipped to convert A to B:

A = 37, B = 20



→ First Do XOR, result will contain no. of sets bits to be flipped.

→ Count No. of set bits in XOR of A & B using Brian Kernighan's algorithm n & (n-1).

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

int countflipbits(int A,int B){
    int n = A^B;
    int count = 0;
    while(n){
        n = n & (n-1);
        count++;
    }
    return count;
}

int main()
{
    int A,B;
    cin>>A>>B;
    cout<<"No of bits to be flip to Convert A to B is : "<<countflipbits(A,B);
    return 0;
}

```

Q 8.) Find missing number in an array:

Given an array nums containing n distinct numbers in the range [0,n], return the only number in the range that is missing from the array.

→ We can use XOR to find missing number.

→ first do XOR of all the elements in the array.

→ Array elements are in range of 0 to N, so XOR of 0-N with XOR of all array elements will give us the missing element.

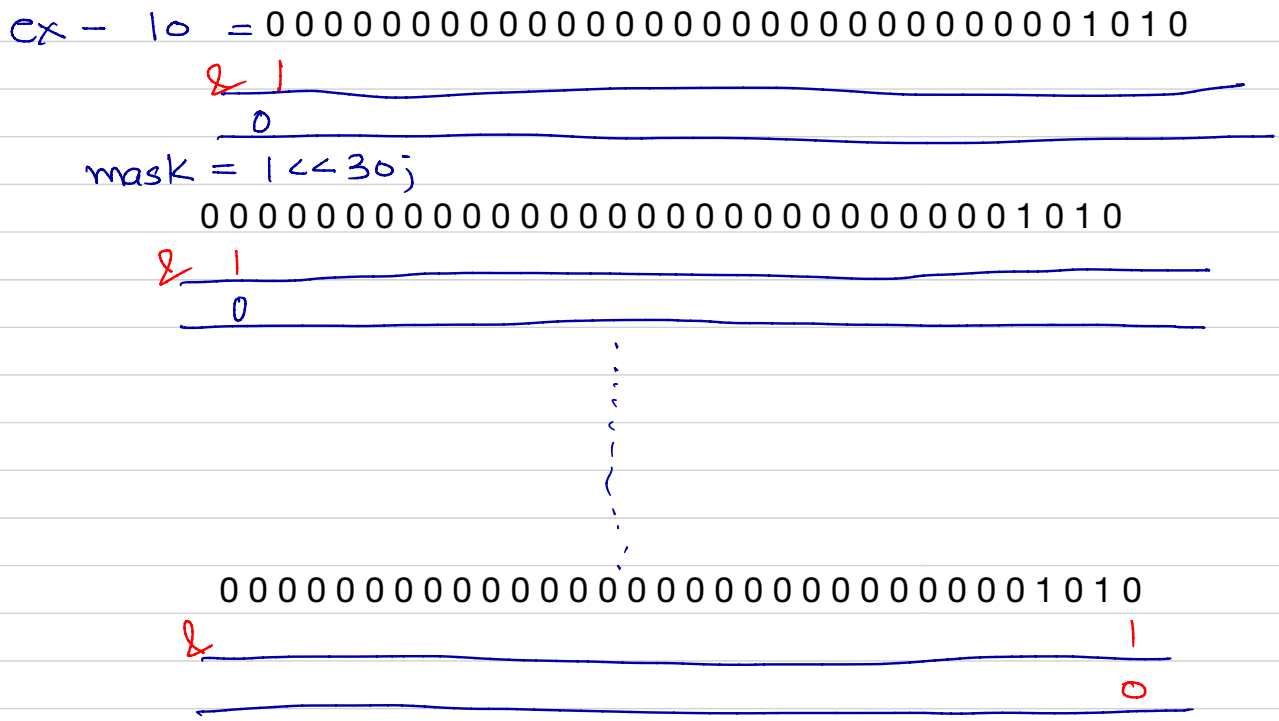
nums = [3, 0, 1]

= (3 ^ 0 ^ 1) ^ (0 ^ 1 ^ 2 ^ 3)

= 2.

```cpp
int missingNumber(vector<int>& nums) {
        int Xor=0;
        int n = nums.size();

    for (int i = 0; i < n; i++) {
        Xor = Xor ^ nums[i];
    }
  //numbers are starting from 0
    for (int i = 0; i <= n; i++) {
        Xor = Xor ^ i;
    }
     return Xor;
    }
```

Q 9) Print the binary representation of decimal number:




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

void decitobinary(int num){
for(int i=31;i>=0;i--)
{
    int mask = 1<<i;
    if(num & mask){
        cout<<"1";
    } else cout<<"0";
}
}

int main()
{
    int num;
    cin>>num;
    cout<<"Binary representation of decimal number is ";
    decitobinary(num);
    return 0;
}

```

Q 10.) Reverse the bits of a number:

n = 00000000000000000000000010101010

Take ans as 0

ans = 00000000000000000000000000000000

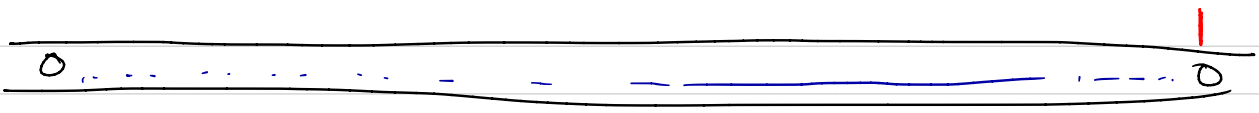
→ We will try to create our ans from LSB to MSB, everytime we will compute our ans and left shift it by 1.

→ Now to compute our ans, we will first check whether the bit is set or not, to check it do (n & 1).

→ Now OR operation of ans with n & 1 will give us ans.

→ Right shift n by 1 everytime till we reach MSB (32 for unsigned int).

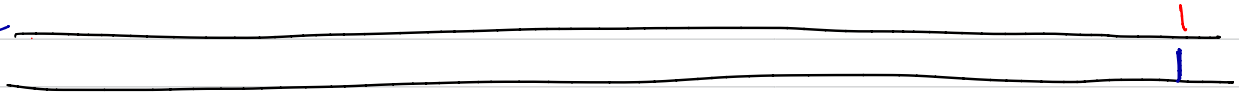
(1) n = 00000000000000000000000010101010



ans = 0 | 0
= 0

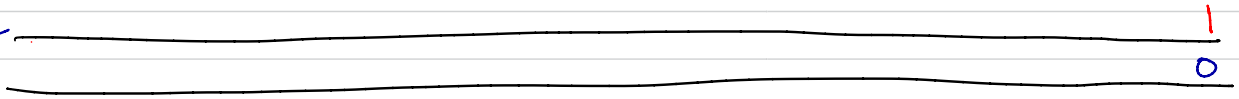
n = n >> 1.

(2) n = 00000000000000000000000001010101
&



ans << = 1.
ans = ans | (n & 1)
= 0 | 1
ans = 1

(3) n = 00000000000000000000000001010100
&



ans << = 1
ans = ans | (n & 1) = 00000 ---------- | 0 = 2.
= 2 | 0
= 2

Everytime answer will be left shift by 1 and or operation will place the correct bit at the position.

```cpp
uint32_t reverseBits(uint32_t n) {
        int ans = 0;
         for(int i=0;i<32;i++)
         {
             ans <<= 1;
             ans = ans|(n&1);
             n >>= 1;
         }
         return ans;
    }
```

Q 11.) Swap the ith and Jth bit.

Given a number num, i and j ranges from 1 to 32.
→ we need to swap i^{th} and j^{th} bit of a number.

ex → num = 43, i = 2, j = 5.
0 0 1 0 1 0 1 1 , output = 57 (0 0 1 1 1 0 0 1)
j i

→ first move the bit value at i^{th} & j^{th} bit
Let A is the bit value at i^{th} bit :-

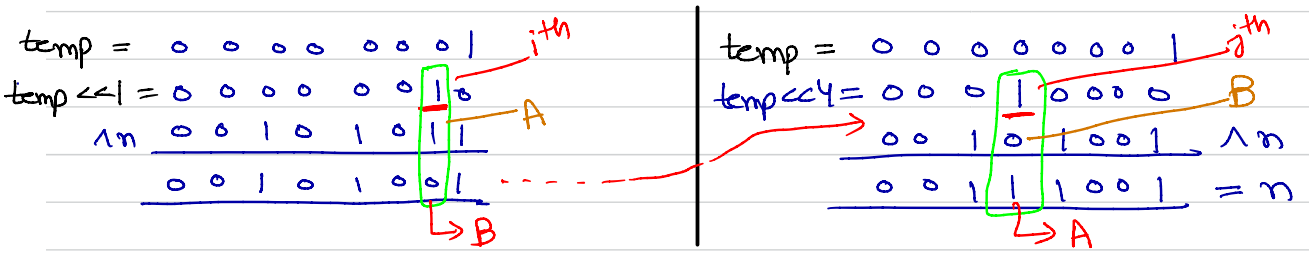
```
n       = 0 0 1 0 1 0 1 1
n>>1    = 0 0 0 1 0 1 0 1
(n>>1)&1 =                1
A       = 0 0 0 0 0 0 0 1
```

move j^{th} bit to rightmost

```
n       = 0 0 1 0 1 0 1 1
n>>4    = 0 0 0 0 0 0 1 0
(n>>4)&1                1
B       = 0 0 0 0 0 0 0 0
```

temp = A ∧ B = 0 0 0 0 0 0 0 1

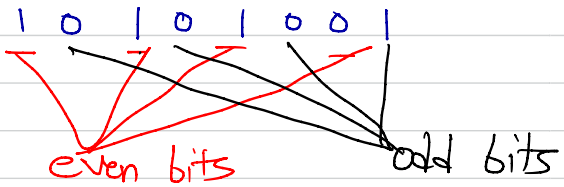
Now, A ∧ B = temp, temp ∧ B = A, temp ∧ A = B
→ Move temp first at i^{th} position & then at j^{th} position.




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

int swapBits(unsigned int n, unsigned int i, unsigned int j)
{   unsigned int A = ((n>>(i-1)) & 1);
    unsigned int B = ((n>>(j-1)) & 1);

    unsigned int temp = A ^ B;
    n = n ^ (temp<< (i-1));
    n = n ^ (temp<<(j-1));
    return n;
}

int main()
{   int n,i,j;
    cin>>n>>i>>j;
    int res = swapBits(n,i,j);
    printf("Result = %d ", res);
    return 0;
}
```

Q 12.) Swap all even and odd bits

Ex -




```
      1 0 1 0 1 0 0 1
even mask  1 0 1 0 1 0 1 0
even bits= 1 0 1 0 1 0 0 0
```

even bits >>= 1;

even bits = 0 1 0 1 0 1 0 0

result = 0 1 0 1 0 1 1 0

```
  1 0 1 0 1 0 0 1
  0 1 0 1 0 1 0 1  - odd mask
  0 0 0 0 0 0 0 1  - odd bits
```

odd bits <<= 1

odd bits = 0 0 0 0 0 0 1 0

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

unsigned int swapevenoddbits(unsigned int N)
{
    unsigned int even_bits = N & 0xAAAAAAAA;
    unsigned int odd_bits = N & 0x55555555;

    // Right shift even bits
    even_bits >>= 1;

    // Left shift odd bits
    odd_bits <<= 1;

    // Combine even and odd bits
    unsigned int result = even_bits | odd_bits;
    return result;
}

int main()
{
    unsigned int N;
    cin>>N;

    cout<<swapevenoddbits(N);
    return 0;
}

```