IInd iteration : res = 4

x = 16 * 16 = 256

n = 1

IIIrd iteration : res = 4 * 256 = 1024

x = 256 * 256 = 65536

n = 0

Time complexity : O(log n)

Aux space : O(1)

03 — Bit Magic

Bitwise Operator in C++ L-01

The bitwise algorithms are used to perform operations at bit-level or to manipulate bits in different ways. The bitwise operations are found to be much faster and are some times used to improve the efficency of a program.

For example: To check if a number is even or odd. This can be easily done by using Bitwise AND (&) operator. If the last bit of the operator is set than it is ODD otherwise it is EVEN. Therefore, if num & 1 not equals to zero than num is ODD

Bitwise Operators

The operators that works at Bit level called bitwise operator. In general there are six types of bitwise operator as discussed below :-

- & (bitwise AND) : Take two number as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. Suppose A = 5 and B = 3, therefore A & B = 1.

A = 5 B = 3

5 = 0 1 0 1

3 = 0 0 1 1

0 0 0 1 → 1

So here we can see A & B = 1

. | (bitwise OR) : Takes two number as operand and does OR on every bit of two numbers. The result OR is 1 if any of two bits is 1. Suppose A = 5 and B = 3 therefore A | B = 7.

A = 5 → 0101

B = 3 → 0011

OR operation 0111 → 7

. ^ (bitwise XOR) : Takes two number as operand and does XOR on every bit of two numbers. The result of XOR is 1 if two bits are different.

Suppose A = 5 and B = 3 therefor A ^ B = 6

A = 5 → 0101

B = 3 → 0011

XOR operation → 0110 = 6

Code →

```
int main() {
    int x = 3, y = 6
    cout << (x & y) << endl;
    cout << (x | y) << endl << (x ^ y);
    return 0;
}
```

- << (left shift): Takes two number, left shift the bits of the first operand, the second operand decide the number of place to shift.

Example:-

Code:

```
int main
{
    int x = 3;
    cout << (x<<1) << endl;
    cout << (x<<2) << endl;

    int y = 4
    int z = (x<<y)
    cout << z;
    return 0;
}
```

Explanation: Here we do the left shift of 3 the value of x which is three will shift for one place.

x = 3 → 0011

x<<1 → 0110 → 6

x<<1 = 6

Now we will shift it for two place

$x << 2$

$x = 3 \longrightarrow 0011$

$x << 2 \longrightarrow 1100 \longrightarrow 12$

$x << 2 = 12$

Now we will shift it for four places

$x << 3$

$x = 3 \longrightarrow 000011$

$x << 4 \longrightarrow 110000 \longrightarrow 48$

$x << 4 \Rightarrow 48$

In left shift we just ignore the number in starting and put zero in the end.

*Note If we assume that the leading y bits are 0, then result of $(x << y)$ is equivalent ~~x * y~~

$x * 2^y$.

- >> (right shift) : Takes two number, right shift the bits of the first operand, the second operand decides the number of places to shift.

Example:

Code :-

```
int main()
{
    int x = 33;
    cout << (x >>1) << endl;
    cout << (x >>2) << endl;

    int y = 4;
    int z = (x >>y);
    cout << z;
    return 0;
}
```

Explanation :

We are doing right shift of x having value 33. Firstly we will shift it for one place

$x = 33 \longrightarrow 100001$

$x >> 1 \longrightarrow$ ~~100010~~ $010000 \rightarrow 16$

$\boxed{x >> 1 = 16}$

Now we will shift it for two place.

$x = 33 \longrightarrow 100001$

$x >> 2 \longrightarrow 00100 \rightarrow 8$

$\boxed{x >> 2 = 8}$

In right shift we will shift the digit to right and add the zero in front and ignore the number at last.

In right shift $(x >> y)$ is equivalent to $\left[\frac{x}{2^y}\right]$

- ~ (Bitwise NOT): Takes one number and invert all bits of it. Suppose A = 5, therefore ~A = 2.

Example:-

Code:

```
int main ( )
{
    unsigned int x = 1;
    Cout << (~x) << endl;
    x = 5;
    Cout << (~x) << endl;
    return 0;
}
```

Explanation:

Unsigned int will range from 32 0s to 32 1s.
So in 32 bits the largest number we can represent $2^{32} - 1 = 4294967296 - 1 \Rightarrow 4294967295$

So here we can see, the value of x is 1 and when we will use not operator then

$x = 1 \longrightarrow$ 00000 01

$\sim x \longrightarrow$ 11111 ..... 10 $= 2^{32} - 1 - 1$

$-x =$ 4294967294

 # Note Taking the ~~number~~ compiler as 32 bit.

Now for $x = 5$

$x = 5 \longrightarrow$ 00000 101

$\sim x \longrightarrow$ 11111 010

$\sim x =$ 4294967290

Now let us ~~see~~ talk about signed input

Code :-

```
int main()
{
    int x = 1;
    cout << (~x) << endl;

    x = 5;
    cout << (~x) << endl;
    return 0;
}
```

Explanation: In signed representation leading bit show the sign of the ~~bit~~ number. If the leading bit is 0, then it is a positive number otherwise it is a negative number.

For the positive number ~~if the~~ simply we will decimal to binary or binary to decimal conversion. Signed bit is represented in 2's complement.

Range of number in signed bit is $(-2^{31}$ to $2^{31}-1)$ in 32 bit compiler.

Signed number when they are negative are stored using 2's complement whereas if they are positive they are stored as normal.

Now for $x = 1$

x : 0 0 ... 0 1

$\sim x$: 1 1 1 ..... 1 0 $(2^{32}-1-1)$

Now for signed number if our computer is using 2's ~~representation~~ complement representation it will be

$$\boxed{2^n - x}$$

So for $x = 1$

$$2^{32} - 1 = 1$$

$$2^{32} - 2$$

So the output will be -2.

For $x = 5$

x: 0 0 . . . . . 0 1 0 1

x: 1 1 . . . . . 1 0 1 0

$$2^{32} - 1 - 5$$

$$= 2^{32} - 6$$

So the output will be -6.

#Note C++ compiler is free to choose any representation. It is not necessary that it will choose only 2's Complement.

It is generally not recommended to apply left shift and right shift operator on at negative number. The behaviar is undefined.

Check if K^{th} bit is set L-06

Write a program which take an input 'n' a & and 'k' and check if K^{th} bit in n is set or not.

I/p: n = 5, k = 1 5 → 0101 (set)

O/p: Yes

I/p : n = 8, k = 2 8 → 1000 (not set)

O/p : No

I/p : n = 0

O/p : No

k ≤ No. of bits in binary representation.

Method 1 → Left shift operator

```
void IthBit (int n, int i)
{
  if (n & (1 << (i-1)) != 0)
    print ("Yes");
  else
    print ("No").
}
```

Explanation: Let us take value of n = 5 and i = 3.

Now 5 → 000.....0101

Now, we are taking 1 → 000.... 0001

Now we will left shift 1 for i-1 times

1 << i-1 → 000..... 0100

```
n =          000.... 0101
1<<(i-1) =   000.... 0100
             ------------
             000... 0100
```

So now all digits will not matter. Only third digit will matter. If the value of third digit is

is 1 then value of AND operation is greater or equal to one and if the value of third digit 0 then the value of AND operation is always 0. as all the other in 1 binary representation is zero.

Method 2 – Right shift

Code :-

```
void KthBit (int n, int K)
{
  if (((n>>(k-1)) & 1) == 1)
     print ("Yes")
  else
     print ("No")
}
```

Explanation : Let us take the value of n = 13 and k = 3

N = 13 ⟶ 00 1101

n>>(k-1) ⟶ 00 0011

1 ⟶ 00 0001

00

Explanation: Here we have right shift the N for k-1 times so that the bit we have to check come at last position. Then we will do his AND operation with one. If the value is equal to one & that mean the bit is one and if the value is equal to zero that means the bit is equal to zero.

Count Set Bits : L-07

Write a program which take an input 'n' and count set bits in n.

I/p: n = 5
O/p: 2

n = 5 → 0.... 0101
Set bits = 2

I/p: n = 7
O/p: 3

n = 7 → 0.... 0111
Set bits = 3

I/p: n = 13
O/p: 3

n = 13 → 0.... 1101
Set bits = 3

Naive Solution :

Code :-

```
int countSet (int n)
{
    int res = 0;
    while (n > 0)
    {
        if (n % 2 != 0)   ⇒   if ((n & 1) == 1)
            res++;                    res++
        n = n / 2                 n = n >> 1;
    }
    return res;
}
```

Explanation : Here the idea is to check whether the last bit is set or not. If the last bit is set we will increment the value of result and then we bring the next bit to the last position.

Brian Kernigam's Algorithm

This algorithm takes time equal to set bit count.

Θ (set bit count).

Let us take $n = 40$.

Now, Initially : 40 → 00.... 0101000

After I'st iteration : 00..... 0100000

After II nd iteration : 00.... 0000000

Code:-

```
int countBits (int n)
{
    int res = 0;
    while (n>0)
    {
        n = (n & (n-1));
        res++;
    }
    return res;
```

#Note When we subtract 1 from any number all the number after the last set bit becomes 1.

E.g $n = 40 \rightarrow 101000$

$n-1 = 39 \rightarrow 100111$

& Here we can see after subtracting 40 from 1 we get all the number after the last set bit.

Explanation: So as we know if subtract one from a number then the number after last set bit will become 1. and after that if we will do AND operation between n and n-1 it will remove the one set bit from the number. When all the set bits are removed we can count the number all return the result.

Lookup Table method

code :

```
int table [256];
void initialize ()
{
    table [0] = 0;
    for (int i = 1; i<256 ; i++)
        table [i] = (i&i) + table [i/2];
}
int count (int n)
{
    int res = table [n & 0xff];
    n = n >>8
    res = res + table [n & 0xff];
    n = n >>8
    res = res + table [n & 0xff];
    n = n >>8
    res = res + table [n & 0xff];
    return res;
}
```

Explanation

Power of two

L-08

Write a program which take an input n and return true if it is power of two and false if n is not power of two.

I/p: n = 4
O/p: Yes

I/p: n = 6
O/p: No

Code:-

```
bool isPowerofTwo(long long n){
    if(n==0){
     return false;
    }

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

```
    else {
        return false;
    }
}
```

Explanation: It is a fact that power of two always have only 1 set bit. So now we will check if the n have one set bit or not. If the number have one set bit it is power of two.

Above code can be written in this way also

Code:-

```
bool isPowofTwo(int n)
{
    if(n==0)
        return false;
{
    return (n!=0) && ((n&(n-1))==0);
}
```

Find the only odd occuring number

L-09

Ques Write a function which will find the element in the array which occur odd times.

I/p : arr[] = {4, 3, 4, 4, 4, 5, 5}

O/p : 3

I/p : arr[] = {8, 7, 7, 8, 8}

O/p : 8

Code :

Naive

```
for (int i = 0; i < n; i++)
{
    int count = 0;
    for (int j = 0; j < n; j++)
        if (arr[j] == arr[i])
            count++;
    if (count % 2 != 0)
        print (arr[i])
}
```

Explanation: Here we are checking for all the elements. We will check number of times the elements are arrived and then increment the value of count. If the value of count is even then we will print that element.

#Note:

$$x \wedge 0 = x$$

$$x \wedge y = y \wedge x$$

$$x \wedge (y \wedge z) = (y \wedge z) \wedge x$$

$$x \wedge x = 0$$

Efficent solution

code:-

```
int findodd (int arr[], int n)
{
    int res = 0;
    for (int i=0; i<n; i++) {
        res = res ^ arr[i];
    }
    return res;
}
```

Explanation: All the elements which occur odd time will get cancelled and only the element which occur odd time will remain.

Variation Question: Given an array of n number that has values in range [1 .... n+1]. Every number appear exactly once. Hence one number is missing. Find the missing number.

I/p: arr[] = {1, 4, 3}

O/p: 2

I/p: arr[] = {1, 5, 3, 2}

O/p: 4

Code :-

```
int getMissingNo (int a[], int n)
{
    int x1 = a[0];
    int x2 = 1;
for (int i = 1 ; i < n , i++)
    x1 = x1 ^ a[i];
for ( int i = 2; i <= n+1 ; i++)
    x2 = x2 ^ i;
return (x1 ^ x2)
}
```

Explanation :- Here firstly we have made two variables x_1 and x_2. Now we have done XOR from 1 to n. After that we will do ~~for~~ XOR from a[0] element to a[n]. Then we will do XOR of $x_1 \wedge x_2$ and all the number which are appeared in both x_1 and x_2 get cancelled and ~~the~~ we will get the missing number.

Another method :

Code :

```
int MissingNumber(vector<int>& array, int n)
{
    int res = (((n+1) * n)/2);
    for(int i = 0; i < n; i++)
        res -= arr[i];
    return res;
}
```

Explanation : Here we will find the sum of the n natural numbers and then subtract it from all the elements in the array. Remaining number is the missing number.

Find two odd appearing number

L-10

We are given an array of integers. Every element in array appears even time except two elements that appear odd number of time. Write a function which print the two odd occuring element.

I/p: arr[] = {3, 4, 3, 4, 5, 4, 4, 6, 7, 7}

O/p: 5, 6

I/p: arr[] = {20, 15, 20, 16}

O/p: 15, 16

Naive solution:

```
void oddAppearing (int arr[], int n)
{
  for (int i=0; i<n; i++)
  { int count = 0;
  for (int j=0 ; j<n ; j++
  { if (arr[i] = arr[j])
      count++;
```

```
        if ( count % 2 != 0 )
            print ( arr[i] );
    }
}
```

Explanation: We are using here two loops. So our idea here is to compare all the element of array with each other. and all those elements which are odd occuring will be print by us.

Time complexity = $\Theta(n^2)$

Efficent method:

```
void oddAppearing ( int arr[] , int n )
{
    int XOR = 0 , res1 = 0 , res2 = 0 ;
    for ( int i = 0 ; i < n ; i++ )
    { XOR = XOR ^ arr[i] ; }
    int sn = XOR & ~(XOR-1) ; // Rightmost set bit
    for ( int i = 0 ; i < n ; i++ )
    { if ( (arr[i] & sn) != 0 )
        res1 = res1 ^ arr[i] ;
```

```
    else
        res2 = res2 ^ arr[i];
}
print (res1, res2);
```

Explanation: let's explain this code with the help of an array.

I/p: arr = { 3, 4, 3, 4, 8, 4, 4, 32, 7 }

O/p : 8, 32

Firstly, we have declared three variables XOR, res1 and res2 and initialize there value as zero.

Now we will do XOR operation for all the elements in array and store it in 'XOR' variable.

Now we will find the rightmost set bit using this line

```
int sn = XOR & ~(XOR-1);
```

This work like this

XOR = { 3 ^ 4 ^ 3 ^ ..... 7 ^ 7 }

XOR = 8 ^ 32

∴ all others number cut themselves as they are even.

8 : 0.... 0001000

32 : 0.... 0100000

XOR : 0.... 0101000 → 40

(XOR-1) : 0.... 0100111

~(XOR-1) : 1.... 1011000

XOR & ~(XOR-1)

1.... 1011000

0.... 0101000

0.....0001000 → This is the rightmost set bit.

Now we will make two groups one having all the elements of array whose same rightmost bit are set. and another group whose same rightmost bit are not set.

Value of sn = 8

res1 = 8

res2 = 3 ^ 4 ^ 3 ^ 4 ^ 4 ^ 4 ^ 32 ^ 7 ^ 7

So here only 18 is stored in ~~xor~~ res1 and ~~and~~ we will print the

for making the groups we will do AND operation with sn. ~~If~~ firstly we will do ~~xor~~ AND operation of res1 and Sn and we will get all the elements whose rightmost set bit are same. After that we will do AND operation of res2 and Sn and ^ we get all the elements which have different rightmost set bit. ~~so~~ So ~~we~~ when we have done ~~AND~~ AND operation we will do XOR of res1 with all the elements of array and same with XOR of sn with res 2. So ~~duplicate~~ all the number which occured at even times will get ~~cancel~~ cancelled.

Power Set using Bitwise Operator

Ques Write a function which ~~take~~ ~~an~~ ~~input~~ print the power set of the string.

I/p : s = "abc"

O/p : "", "a", "b", "ab", "abc", "ac", "bc", "c"

I/p : s = "ab"

O/p : "", "a", "b", "ab"

Code :-

```
void printPowerset(string str)
{
    int n = str.length();
    int powSize = pow(2, n);
    for (int counter = 0; counter < powsize; counter++)
    {
        for(int j=0; j<n; j++)
        ~~{ print(str[j]); }~~
        {
            if ( counter & (1<<j) != 0 )
                { print (str[j]); } print("\n");
        }
```

Explanation :

Counter (Decimal)	Counter (Binary)	Subset
0	000	" "
1	001	"a"
2	010	"b"
3	011	"ab"
4	100	"c"
5	101	"ac"
6	110	"bc"
7	111	"abc"

I/p : s = "abc"

O/p : " ", "a", "b", "c", "ab", "ac", "bc"
"abc".

Power set = $2^n - 1$

So what we are doing here is that we will run a loop for counter = 0 from to the $2^n - 1$.. in that we make another loop and for every value of counter we transverse ~~the loop from~~ through the string.

For every character we check the correspon bit is set or not.

Suppose count = 3, then we will run the loop from j=0 to j = 2. Now for j = 0, we will check whether the last bit of counter is set or not:

We will check it using the expression

Counter & (1<<j)

As value of j=0, so 1<<j = 1 and AND operation counter & 1<<j will be non zero so we will print 'a'.

Now for j = 1

1<<j = 2

Counter & (1<<j)

3 & 2

Here also the value of AND operation is not equal to zero so we will print 'b'

Now for j = 2

1<<j = 4

Counter & (1<<j)