1. Bitwise AND : &

```
int main()
{
    int x=3;
    int y=6;
    Cout<<(x&y);
    Return 0;
}
```

output : 2

3 : 000 ---- 0011
6 : 000 ---- 0110 &
---
00 --- 0010

I st	II nd	&
0	0	0
0	1	0
1	0	0
1	1	1

2. Bitwise OR : |

```
int main()
{
    int x=3;
    int y=6;
    Cout<<(x|y);
    Return 0;
}
```

output : 7

3 : 00 ---- 0011
6 : 00 ---- 0110
---
00 --- 0111

I st	II nd	|
0	0	0
0	1	1
1	0	1
1	1	1

3) Bitwise XOR : ^

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

output : 5

```
3: 00.....0011
6: 00 ....0110   ^
-----------------
   00----0101
```

I st	II nd	^
0	0	0
0	1	1
1	0	1
1	1	0

4) Left shift : <<

```
int main()
{
  int x=3;
  cout<<(x<<1)<<endl;
  cout<<(x<<2)<<endl;
  int y=4;
  int z=(x<<y);
  cout<<z;
  return 0;
}
```

```
x   : 0000 --- 000011
x<<1: 0000 -- 000110
x<<2: 0000 --- 001100
x<<y: 0000 -- 110000
```

O/P: 6
12
48

⇒ Remove and ignore leading y bits.

⇒ Move remaining 32 - y bits to leftmost

⇒ Append y 0's at the end.

5) Right Shift : >>

```
int main()
{
  int x=33;
  cout<<(x>>1)<<endl;
  cout<<(x>>2)<<endl;
  int y=4;
  int z=(x>>y);
  cout<<z;
  return 0;
}
```

x : 0000 ... 100001

x>>1 : 0000 -- 010000

x>>2 : 0000 -- 001000

x>>y : 0000 -- 000010

o/p: 16
8
2

→ Remove and Ignore trailing y bits.

→ move remaining 32 y bits to rightmost.

→ Add y 0's at the begining

① x<<y is equivalent to $x*2^y$. if leading y bits are 0 in x.

x	y	x<<y	
3	1	6	$(3*2^1)$
3	2	12	$(3*2^2)$
3	4	48	$(3*2^4)$

② x>>y is equivalent to $\lfloor x/2^y \rfloor$

x	y	x>>y	
33	1	16	$\lfloor 33/2^1 \rfloor$
33	2	8	$\lfloor 33/2^2 \rfloor$
33	4	2	$\lfloor 33/2^4 \rfloor$

6) Bitwise NOT : ~

x = 1 :

00...01

~x

11...10

x = 5

00...101

~x

11...010

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

Output :

4294967294

4294967290

Please Note : $2^{32} - 1 = 4294967296 - 1$

$= 4294967295$

7) Bitwise Not: ~ (Signed Numbers)

$(-2^{31}$ to $2^{31} - 1)$

x = 1

00...01

~x

11....10

$(2^{32} - 1 - 1)$

x = 5

00...101

~x

11...010

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

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

Output

-2

-6

2's Complement Of $x = 2^{32} - x$

⇒ Negative Numbers are Represented in 2's Complement form.

⇒ Range of Numbers : $[-2^{n-1}$ to $2^{n-1}-1]$

Here n is no. of bits.

⇒ Steps to get 2's Complement

a) Invert all bits

b) Add 1

Direct formula = $2^n - x$

Example - n=4

Range : $[-2^3$ to $2^3-1]$

Binary Representation of

x=3

3 : 0011

2's Complement =

(1100+1)

= 1101

Why 2's Complement form?

1) We have only one representation of zero.

2) The arithmetic operations are easier to perform. Actually 2's Complement form is derived from the idea of 0-x.

3) The leading bit is always 1.

Check If K^{th} Bit is Set :-

I/p: $n = 5$, $K = 1$ $\quad n = 00\cdots0101$

o/p: yes

I/p: $n = 8$, $K = 2$ $\quad n = 00\cdots1000$

o/p: No

I/p: $n = 0$, $K = 3$ $\quad n = 00\cdots0000$

o/p: No

K ≤ No. of bits in the binary representation of n

→ How do you check the last bit?

```
if((n&1)!=0)
    print("yes")
else
    print("No")
```

→ How do you check the K^{th} bit?

We mainly need to do bitwise and with number with only K^{th} bit set $00\cdots010\cdots00$

(K^{th} bit)

Naive Solution

```
void isSet(int n, int k)
{
    int x=1;
    for(int i=0; i<(k-1); i++)
        x=x*2;
    if((n&x)!=0)
        print("yes")
    else
        print("No")
}
```

n=5 (0...0101)

k=3

After loop:

x=4 (0...0100)

```
  0...0101
& 0...0100
-----------
  0...0100
```

↓

Representation of 4.

Alternate Naive Sol^n

We Reduce n to $\lfloor n/(2^{k-1}) \rfloor$

Time: $\theta(k)$

```
void isSet(int n, int k)
{
    for(int i=0; i<(k-1); i++)
        n=n/2;
    if((n&1)!=0)
        print("yes")
    else
        print("No")
}
```

n=5 (0...0101)

k=3

After loop

n=1 (0...0001)

Efficient Method I

```
void isKthSet(int n, int k)
{
  int x = (1<<(k-1)); // 2^(k-1)
  if ((n&x)!=0)
     print("yes");
  else
     print("No");
}
```

n=5 (00...0101)
k=3
x=4 (00...0100)

```
  00...0101
& 00...0100
  00...0100
```

Efficient Method II

```
void isKthSet(int n, int k)
{
 int x=(n>>(k-1)); // ⌊n/(2^(k-1))⌋
 if ((x&1)!=0)
     print("yes")
 else
     print("No")
}
```

n=5 (00...0101)
k=3
x=1 (00...0001)

```
   00....0001   x
&  00...0001   1
   00....0001
```

Count Set Bits

I/P : n=5

O/P : 2

n=5

101

Binary Representation.

I/P : n=7

O/P : 3

n=7

111

I/P : n=13

O/P : 3

n=13

1101

Naive Solution

1) Initialize : res=0

2) Traverse from the last bit to MSB and increment res for set bits

3) Return res

MSB:

Most Significant Bit

13 : 00...01101

MSB

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

n = 5 (00...0101)
res = 0

1st Iteration -
res = 1, n = 2 (00...010)

IInd Iteration -
res = 1, n = 1 (00...001)

IIIrd Iteration -
res = 2, n = 0 (00...000)

Time = $\Theta(d)$

d = No. of bits from Last to MSB.

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

Brian Kernigam's Algorithm

Idea:

Traverse through only the set bits.

Time: θ(set bits)

```
int countSetBits(int n)
{
  int res=0;
  while(n>0)
  {
    n= ?        → This expression should make the last set bit as 0.
    res=res+1;
  }
  return res;
}
```

$n = 40\ (101000)$

After Ist Iteration

$n = 32\ (100000)$

After IInd Iteration

$n = 0\ (000000)$

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

$n = 40$: 101000

$(n-1) = 39$: 100111

$n\&(n-1) = 32$: 100000

$n = 32$: 100000

$n-1 = 31$: 011111

$n\&(n-1) = 0$: 000000

if we subtract a number by 1 and do it bitwise & with itself (n & (n-1)), we unset the rightmost set bit.

Lookup table Solution

The idea is to set count bits in O(1) time with some preprocessing involved.

Assumption - we have 32 bit numbers.

⇒ Precompute counts for 8 bit numbers (0 to 255)

```
tbl[0] = 0
for i=1 to 255
    tbl[i] = tbl[i & (i-1)] + 1
```

⇒ Now to count set bits in a given no. n.

n = 13 : 0....0 0....0 0....0 00001101

8 8 8 8

→ How do you count set bits in the 4 segments individually using tbl[].

```
int tbl[256];
```

Lookup table

```
void initialize()
{
    tbl[0] = 0;
    for(int i=1; i<256; i++)
        tbl[i] = tbl[i & (i-1)] + 1
}
```

Pre processing

00..0 (24) 1..1 (8)

```
int countSetBits(int n)
{
    return tbl[n & 255] +
           tbl[(n>>8) & 255] +
           tbl[(n>>16) & 255] +
           tbl[(n>>24)];
}
```

O(1) function

n = 13

0...0 (24) 0...0 (24) 0...0 (24) 00001101 (8)

tbl[13] + tbl[0] + tbl[0] + tbl[0]

Power of Two $n \geq 0$

I/P : n = 4

O/P : True

I/P : n = 6

O/P : False

I/P : n = 1

O/P : True

Naive Solution

1) Explicitly Handle 0

2) Repeatedly divide by 2

n = 6

I^st Iteration : n = 3

II^nd Iteration : return false

n = 8

I^st Iteration : n = 4

II^nd Iteration : n = 2

III^rd Iteration : n = 1

return true

```
bool isPow2(int n)
{
    if (n==0)
        return false;
    while (n!=1)
    {
        if (n%2 != 0)
            return false;
        n = n/2;
    }
    return true;
}
```

Efficient Solution

Binary Representations of Powers of 2 have only one set bit.

If Count Set bits is 1
return true.
Else
return false.

1 : 00 01

2 : 00 10

4 : 00 100

n=4 : 00 .. 0100
&(n-1) =3: 00 .. 0011
---
00 .. 0000

n=6 : 00 .. 0110
&(n-1)=5: 00 .. 0101
---
00 .. 0100

```
bool isPow2(int n)
{
    if(n==0)
        return 0;
    return ((n&(n-1))==0);
}
```

return (n&((n&(n-1))==0))

Find the only Odd Occurring Number

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

O/p: 3

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

O/p: 8

Naive Solution

```
int findOdd(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)
            return arr[i];
    }
}
```

{7, 3, 7, 7, 7}

i=0: count=4

i=1: count=1

return 3

Efficient Solution Idea

Interesting Properties of ^

$x \wedge 0 = x$

$x \wedge y = y \wedge x$

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

$x \wedge x = 0$

$\underbrace{x \wedge x \wedge \cdots x}_{\text{odd times}} = x$

$\underbrace{x \wedge x \wedge \cdots x}_{\text{even time}} = 0$

```
int findOdd(int arr[], int n)
{
    int res = arr[0];
    for(int i=1; i<n; i++)
        res = res ^ arr[i];
    return res;
}
```

arr[] = {4, 4, 7, 4, 8, 7, 7, 7, 8}

4^4^7^4^8^7^7^7^8

= (4^4^4)^(7^7^7^7)^(8^8)

= 4^0^0

= 4^0

= 4

Time : Θ(n)

Aux space : O(1)

1) XOR of any number n with itself gives us 0, i.e., n ^ n = 0
2) XOR of any number n with 0 gives us n, i.e., n ^ 0 = n
3) XOR is cumulative and associative.

Find Two Odd Appearing Numbers

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

O/p: 5 6

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

O/p: 2 3

Naive Solution

Traverse through the array, count Occurances of every number. If count is odd, print the number.

$\Theta(n^2)$ Time

$\Theta(1)$ Aux Space

Naive Solution

```
Void printodd(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]);
    }
}
```

{10, 3, 3, 5}

i=0 : count = 1
print(10)

i=1 : count = 2

i=2 : count = 2

i=3 : count = 1
print(5)

Idea for Efficient solution

1) Find XOR of all the numbers

[5, 6, 10, 6, 10, 6, 3, 3]

$x = 5 \wedge 6 \wedge 10 \wedge 6 \wedge 10 \wedge 6 \wedge 3 \wedge 3$

$= 5 \wedge 6$ (00...0101 ^ 00...0110)

$= 3$ (00...0011)

2) How do we find the numbers 5 and 6 from value 3.

Hint:- A set bit in 3 means, the bit is having different values in 5 and 6.

Idea is to use the fact that xor2 is '1' in indexes where bits of x and y are different. So we separate x and y to different groups, along with rest of the numbers of list, based on whether the number has same set-bit or not.

We choose the rightmost set bit of xor2 as it is easy to get rightmost set bit of a number (bit magic). If we bitwise AND a number with its negative counterpart, we get rightmost set bit. (just an observation based property, do remember). So, (xor2) & (-xor2) will give us right set bit. Find (-number) by 2's complement, that is ((1's complement) +1). It can also be written as (~number)+1.

In third step, we separate x and y in different groups : We now know that for selected set bit index, x and y have different corresponding bits. If we AND all numbers in list with set bit, some will give 0 and others will give 1. We will put all numbers giving zeroes in one group and ones in another. x and y will fall in different groups.

```
Void oddAppearing(int arr[], int n)
{
    int x = arr[0];
    for(int i=1; i<n; i++)
        x = x ^ arr[i];
//right set bit int k = (x & (~(x-1)));
    int res1=0, res2=0;
    for(int i=0; i<n; i++)
    {
        if((arr[i] & k)!=0)
            res1 = res1 ^ arr[i];
        else
            res2 = res2 ^ arr[i];
    }
    print(res1, res2);
}
```

x=3 : 00 0011

x-1 =2 : 00 0010

~(x-1) : 11 1101

x & ~(x-1) : 00 0001

arr = {1, 6, 5, 6, 6, 1}

x = 1 ^ 6 ^ 5 ^ 6 ^ 6 ^ 1 = 3

k = 1

res1 = 1 ^ 5 ^ 1 = 5

res2 = 6 ^ 6 ^ 6 = 6

How does x & ~(x-1) work?

It finds a number which has only 1 bit set and the set bit corresponds to last set bit of x.

Example of (number) & (-number) = right set bit :

Power Set Using Bitwise Operators

$s = "abc", n = 3$

→ We consider binary representations of number from 0 to 7.

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

```
void printPowerset(string s)
{
  int n = s.length();
  int psize = (1<<n);
  for(int i=0; i<psize; i++)
  {
    for(int j=0; j<n; j++)
      // Check if jth bit in the counter is set
      if(i&(1<<j)!=0)
        // If set then print jth element from set
        print(s[j]);
    print("\n");
  }
}
```

Output:
```
""
"a"
"b"
"ab"
```

$s = "ab", n = 2$

psize = 4

i = 0 (00): j = 0
j = 1

i = 1 (01): j = 0 print(a)
j = 1

i = 2 (10): j = 0
j = 1 print(b)

i = 3 (11): j = 0 print(a)
j = 1 print(b)