3. Bit Wise

AND ($\&$), OR ($|$), XOR ($\wedge$)

Left Shift ($<<$), Right Shift ($>>$)

AND :

1st	2nd	res
0	0	0
0	1	0
1	0	0
1	1	1

OR :

1st	2nd	res
0	0	0
0	1	1
1	0	1
1	1	1

XOR :

1st	2nd	res
0	0	0
0	1	1
1	0	1
1	1	0

3 & 5, 5: 101
3: 011
1: 001

3 | 5, 5: 101
3: 011
7: 111

3 ^ 5, 5: 101
3: 011
6: 110

Bitwise Not ($\sim$)

x: 1011
~x: 0100

1st	res
1	0
0	1

for n bits, unsigned
- Min : 0
- Max : 2^n-1

ex: n=4,
Min = 0, Max = 2^4-1 = 15 (1111)

for n bits, signed
- min : -2^{31}
- Max : $2^{31}-1$

ex. 2's Comp of x in n bits rep = $2^n - x$

-ve x
ex, n=3, x=-2, => 2^3-2 => 8-2 => 6 (110)
2 in 3 bits is

-ve signed : stored in 2's complement representation

(0)001 → +ve | (1)001 → -ve

Left shift : * Bits moves left side

ex:- x : (0)0000110
discarded
00001100
zero is added at end

we shift only 1 bit so only 1 zero is added

* Multiplies the given num by 2^n where n is no of shifts.

ex:- 5 : 0101 & let n=1
x(0)101
1010 zero added
10 ($5 \times 2^1 = 10$)

Right shift : * Bits moves Right side

ex:- x = 0010100(0) → discard
zero added (0)0010100 → 001010

* Devides the given num by 2^n where n is no of shifts.

ex 10 : 1010, n = 1.
101(0) descard
0101
5 i.e. ($\frac{10}{2^1} = 5$).

22: ~~0010110~~
discard (0 0) 1 0 1 1 0
1 0 1 1 0 (0 0) added
1011000 → 88,

22×2^2 → 22 × 4 → 88
no of shifts.

22: 10110, n = 2, leftshift & Right shift

after 2 shifts descard
1 1 0 (0 0)
11000
2 zeros are added as 2 shifts
22×2^2 → 88

//Signed Input (-2^{31} to $2^{31}-1$)

```
int main()
{
    int x=1
    cout<<(~x)<<endl; //-2
    x=5
    cout<<(~x)<<endl; //-6
    return 0;
}
```

//unsigned Input (0 to $2^{n}-1$)

Bitwise Not

x: 000...... 01 (32 bit rep of 1)

~x: (1)1 1 10

signed -ve number

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

$2^{32}-1 \Rightarrow 11....1$ (32 1's)

$(2^{32}-x) \Rightarrow x=(-2)$

x: 000.....0101

~x: (1)11......1010

signed

$2^{32}-1-5 \Rightarrow 2^{32}-6$

$2^{3}-1 \Rightarrow 8-1=7$ 111

1) check if kth bit is set

i) I/p n=5, K=1, o/p = yes

000... 0101

2nd bit (ie K=2), 1st bit (ie K=1)

ii) n=8, K=2, o/p = No

000..01000 Not set

K=2, K=1

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

(<<)

(32 bits) n = 000..... 0101

1<<(k-1) = 000.......0100

(&)

00100

000..0001 << 2

0.....00100

```
void kthBit(int n, int k)
{
    if((n>>(k-1)) & 1) ==1
        print("yes");
    else print("no")
}
```

(>>)

n: 000.....1101   13>>2

n>>(k-1): 000....0011

kth bit moves to last bit position & check (1) &1 of 2

2) Count Set Bits:

i) I/p n=5, o/p = 2

B.R = 0101 → 2 bits are 1.

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

→ O(d), d: Total bits in 'n'

101   res

1 → True +1

10

n=10

10

10 → false

1 → T +1

2

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

→ BRIAN KERNINGAN's Algorithm

→ O(no of set bits)

n=5,

5- 101, 4- 100, 100, res=+1

4-100, 3-011, 000, res+1

0>0 F, res=2

n=7,

7-111, 6-110, 110, +1

6-110, 5-101, 100, +1

4-100, 3-011, 000, +1

0>0 F, res=3

n=11

11 → 1011, 10 → 1010, 1010, +1

10 → 1010, 9 → 1001, 1000, +1

8 → 1000, 7 → 0111, 0000, +1

0>0 F, res=3

Lookup Table method for 32 bit number for countBits :-

$n = 13$

8 bits → 0 to $2^8 - 1$ ⇒ 255

$\underbrace{000\ldots0}_{\text{8 bits, 4th}}$ $\underbrace{00\ldots0}_{\text{8 bits, 3rd}}$ $\underbrace{0\ldots0}_{\text{8 bits, 2nd}}$ $\underbrace{0\ldots1101}_{\text{8 bits, 1st block}}$ } total 32 bit number → 4 blocks of 8 bits

```
int table[256];
void initialize()
{
    table[0] = 0;
    for(int i=1; i<256; i++){
        table[i] = (i&1) + table[i/2];
    }
}
```

for i=5, 101 → 1 → 50 → +2 ; table[5/2] + table[i] ⇒ +1

```
int count(int n)
{
    int res = table[n & 0xff];   // take the 1st block of 8 bits
    n = n>>8;   // right shift to 8 bits
    res = res + table[n & 0xff];
    n = n>>8
    res = res + table[n & 0xff];
    n = n>>8
    res = res + table[n & 0xff];
    return res;
}
```

0xff — Hexa Decimal number of 8

4 blocks

} If 64 bits then 8 blocks are prefer

3) Power of Two :-

i) I/p n=4, o/p = Yes
ii) I/p n=6, o/p = No

Ily, $\frac{4}{2}^2 \rightarrow \frac{2}{2}^1 \rightarrow 1$ 1 yes

$\frac{6}{2}^3 \Rightarrow 3$ 3 odd → No

Naive:

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

TC: $O(\log n)$

Optimal

$2^0 \Rightarrow 1 \Rightarrow 000\ldots\ldots0001$
$2^1 \Rightarrow 2 \Rightarrow 000\ldots\ldots0010$
$2^2 \Rightarrow 4 \Rightarrow 000\ldots\ldots0100$
$2^3 \Rightarrow 8 \Rightarrow 000\ldots\ldots1000$
⋮

} The no of set bits for the value (2^n) is always equal to 1. Use Brian Kernigan's Algo to count set bits.

Method 2:-

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

→ $O(1)$

4 → 100
3 → 011
000 ⇒ 0 ✓ T

4) Find the only odd occurring number:-

ex: I/p: arr[] = [4, 3, 4, 4, 4, 5, 5] → (e→f: 4 → 4, 3 → 1, 5 → 2)

o/p = 3

Here, odd occuring, even occuring : use XOR Bitwise

In XOR, $X \wedge 0 = X$

$X \wedge X = 0$

$X \wedge Y = Y \wedge X$ (∴ computative)

$X \wedge (Y \wedge Z) = (X \wedge Y) \wedge Z$ (∴ Associative)

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

→ TC: O(n)
SC: O(1) Aux Space

Variation Q/A:- Given an array of n numbers that has values in range [1...n+1]. Every no. appears exactly once. Hence one numbers is missing. Find the missing number?

Ip = [1, 4, 3]

o/p = ?

(1 to n+1) numbers :→ $\underbrace{(1 \wedge 2 \wedge 3 \wedge 4)} \wedge \underbrace{(1 \wedge 4 \wedge 3)}_{\text{our array elems}}$

n=4.

→ only 2 comes out as other number occurs 2 times

---

5) Find the two odd appearing number:-

Ip:- arr[] = [3, 4, 3, 4, 5, 4, 4, 6, 7, 7] → (3 → 2, 4 → 4, 6 → 1, 5 → 1, 7 → 2) → o/p {5, 6}

```
void oddAppearing(int a[], int n)
{
    int XOR=0, res1=0, res2=0;
    for(int i=0; i<n; i++){XOR= XOR ^ a[i];}
    int rn = (XOR) & (~(XOR-1)) // Right Most Bit set of XOR
    for(int i=0; i<n; i++)
    {
        if((a[i] & rn) != 0)
            res1 = res1 ^ a[i];
        else
            res2 = res2 ^ a[i];
    }
    return {res1, res2};
```

→ O(n): TC
O(1): SC

5 → 101
6 → 110

XOR ⇒ 5∧6 ⇒ ...011 ⇒ 3

(XOR-1) ⇒ ... 010 ⇒ 2

~(XOR-1) ⇒ ... 101

(XOR) & (~(XOR-1)) ⇒ 1

3 → 011 (1 set) 4 → 100 (0 not set) 110, 101

res2 = (3 ∧ 3 ∧ 5 ∧ 7 ∧ 7) ⇒ 5

res1 = (4 ∧ 4 ∧ 4 ∧ 4 ∧ 6) → 6

{5, 6}

In 10

3 → 011, 10 → 1 set; 6 → 110, 10 → 1 set; 4 → 100, 10 → 0 not set

6) Power Set using Bitwise Operators :-

I/p : s = "abc"

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

for 'n' character we have, 2^n subsets

$n = 3, \; 2^3 \Rightarrow 8$

$n=3$, $2^3 - 1 \Rightarrow 8 - 1 \Rightarrow 7$ series 0 to 7 Decimal.

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"

value varies from 0 to $2^n - 1$

```
void printPowerset(string str)
{
    int n = str.length();   // n=3
    int powersize = pow(2,n); // 8
    for(int c=0; c< powersize; c++)
    {
        for(int j=0; j<n; j++) // abc
        {
            if (c & (1<<j) != 0){
                print(str[j])
            }
        }print("\n");
    }
}
```

011, 10, 3

$i=0 \; 2^0 \Rightarrow 1$, $1<<1$

0001, 0010

$1 \,\&\, (1<<0)$

$1 \times 2^0 \Rightarrow 1 \times 1$

$1 \,\&\, 1 \Rightarrow 1 \rightarrow a$

$2 \,\&\, (1<<0)$

$1 \times 2^0 \rightarrow 1 \times 1$

$1<<1$

$1 \times 2^1 \Rightarrow 2$

10, 10, 0 → b