Bitwise operators

Bitwise AND:

Java Represents Integers in binary representation as 32 bits.

```
x : 0....0011
y : 0....0110
-------------
(x&y) : 0...0010
```

```
public class Test {
  public static void main(String [] args)
  {
     int x=3, y=6;
     System.out.println(x&y);
  }
}
```

o/p: 2

Bitwise OR:

```
x : 0....0011
y : 0...0110
-------------
(x|y) : 0...0111
```

```
public class Test {
  public static void main(String[] args)
  {
     int x=3, y=6;
     Sout(x|y);
  }
}
```

o/p: 7

Bitwise XOR:

produces o/p: 1 when two I/p bits are different

o/p: 0 " " " same.

```
x : 0....0011
y : 0....0110
-------------
(x^y) : 0...0101
```

$x \wedge y = 5$

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

o/p: 2 7 5

Bitwise NOT:

x : 000 ..... 01 (32 bits)

~x : 111 ..... 1 (2) $(2^{32}-1-1)$

If leftmost is "0" then numbr is +ve

" " is "1" " is -ve

```
class Test {
public static void main (String[] Jo
{
int x=1;
System.out.println(~x);
}
}
```

o/p: ? -2

In Java, negative numbers are stored in 2's complement representation:

representation of $(-x) = 2^{32} - x$

$2^3 - 1 = 111$ (3 1's)

$2^{32} - 1 = 111 \ldots 111$ (value in binary is 1) 32 ⇒ 1's

$2^{32} - 1 - 1 = 111 \ldots 110$

$2^{32} - 2$

Range of int: -2^{31} to $2^{31}-1$

(or)

-2,147,483,648 to 2147483647

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

$-2 = 2^{32} - 2$ ⇒ ans is -2

```
class Test {
psvm() {
int x=5;
Sout(~x);
}
}
```

x : 000 0101

~x : 111 ..... 1010 ($2^3 + 2^1$) ⇒ $2^{32} - 1 - 5$

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

$x = -6$

o/p: -6

Left Shift (<<):

x : 000 0011

x<<1 : 000 0110

x<<2 : 000 1100

x<<4 : 000 110000

```
class Test {
psvm() {
int x=3;
Sout(x<<1); ⇒ 6
Sout(x<<2); ⇒ 12
int y=4;
int z=(x<<y);
Sout(z); ⇒ 48
}
}
```

(3 << 30)

1100 00

```
class Test {
  psvm() {
    int x = -1;
    Sout (x<<1);
  }
}
```

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

⇓

111 . . . 11,1 (32 bits (32 ones))

⇓ left shift

111 . . . 110

⇓

Still -ve number

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

ans: -2

x << y

⇓

x * 2^y

Right Shift (>>):

x : 00 - - - 10000 1

(x>>1): 00 010000

x>>2 : 00 001000 ⟹ o/p : 8

```
class Test {
  psvm() {
    int x = 33;
    Sout (x>>1)
  }
}
```

o/p : 16

-ve numbers:

```
class Test {
  psvm() {
    int x = -2;
    Sout (x>>1);
  }
}
```

$2^{32} - 2$ = 1111 . . . 10 (31 1's, 0)

x>>1 : 1111 . . . 11 ⇒ $2^{32} - 1$

⇓

In case of -ve numbers we fill 1 as leading bit

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

x>>2 : 111 . . . 11 ⇒ $2^{32} - 1$ ⇒ o/p : -1

o/p : -1

If we shift -2 (-ve number) with any value ≥ 1 we get o/p as -1.

Unsigned Right Shift: >>>

for -ve numbers >> => Right shift fills leading bits as 1's

>>> => fills leading bit as 0's

x: 111---10

(x>>>1): 0 11---11

31 => 1's

$2^{31}-1$ =

```
class Test {
    psvm() {
        int x = -2
        sout (x >>> 1);
    }
}
```

O/P: 2147483647

(x>>>2): 00 11---11

30 => 1's

$2^{30}-1$

(x>>>2)

$2^{30}-1$ = 1073741823

check if Kth bit is set:

I/P: n=5, K=1

O/P: yes

000---0101 (K)

29 => 0's

1 is set so O/P is

I/P: n=8, K=2 ⟹ 000---01000

O/P: NO

I/P: n=0, K=3 ⟹ 0000---000

O/P: No

K ≤ No of bits in binary representation

method 1 : (left shift):

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

I/p: n = 5 , k = 3

o/p: yes

n = 000...0101 (29 zeros, 3)

1 = 0000...001

k-1 = 2

1<<2 ⇒ 000...100

1<<(k-1) ⇒

```
n = 000..0101
1<<(k-1) = 000..0100
        &
4 ⇒ 000..0100
```

method 2 : (Right shift):

```
void kth Bit (int n, int k)
{ if (((n>>(k-1)) & 1) == 1)
    print ("yes");
  else
    print ("No");
```

I/p: n=13 k=3

o/p: yes

```
n = 0....1101
n>>(k-1) = 0...0011
           0...0001
1 ⇒ &      0...0001
           0....0001
```

Count Set Bits

I/P: n = 5 => Binary Representation : 000...0101 (29 bits, 3 bits)
O/P: 2 — Set bits = 2

I/P: n = 7 => 000...0111
O/P: 3 — Set bits = 3

I/P: n = 13 => 000...1101
O/P: 3 — Set bits = 3

Naive Solution :

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

000...0101 (29 bits)
lastbit is 1 => res = 1 and Right shift

000...0010 (30 Bits)
Last bit is 0 => res is same, Right Shift

000...001 (31 Bits)
Last bit is 1 => res = 2, Right shift

0000...000 (32 Bits)
Stop the program.

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

TC = O(Total bits in Binary repr of n)

n = 5 = 000...0101
1 = 000...0001
000-0001 => 1

Count set bits

Brain Kerningam's Algorithm:

T.C = Θ(set Bit Count)

n = 40

Binary Representation

Initial : 00. . . . 0101000

After Ist Iteration : 00. . . . 0100000

" IInd Iteration : 00. . . . 0000000

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

when we subtract 1 from n. After last set bit 0's becomes 1's set bit become 0.

n = 40 => 101000

n-1 = 39 => 100111

n & n-1 => 100000 => 32

n = 32 => 1000000

n-1 = 31 => 011111

n & n-1 => 000000 => 0

Count set Bits

Lookup Table Method for 32 bit number:

T.C = $\Theta(1)$ => effecient solution

It requires some preprocessing.

divide our 32 bits into 8 bit chunks.

Range of 8 bits => 0 to 255

⇓

0 to $2^8 - 1$

n = 13

0...0 | 0...0 | 0...0 | 0...1101

8bits | 8bits | 8bits | 8bits

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

precomputation

Naive method

0x ff → Hexadecimal representation of 8 set bits 1---1111 all 1s 8bits

```
        1--1111
&   0--0 0--0 0--0  0--1101
-------------------------
    0--0 0--0 0--0  0--1101
>>
    0--0 0--0 0--0  0--0
```

Power Of Two

I/P: n=4
O/P: yes

I/P: n=6
O/P: NO

$2^0, 2^1, 2^2, 2^3, 2^4 \ldots$
$1, 2, 4, 8, 16 \ldots$

method 1 (Naive):

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

6 ⇒ $\frac{6}{2}=3$, $\frac{3}{2}$ = odd ⇒ false

8 ⇒ $\frac{8}{2}=4$, $\frac{4}{2}=2$, $\frac{2}{2}=1$ ⇒ power of 2

In powers of two there is only one set bit. So $T.C = O(\text{set bits}) = O(1)$.

method 2 (effecient):

```
bool is pow2 (int n) {
  if (n==0)
    return false;
  return ((n & (n-1))==0);
}
```

(or)

```
bool is pow2 (int n) {
  return (n!=0) && ((n&(n-1))==0);
```

ex: 1:
n = 4 : 00---100
n-1 = 3 : 00---011
00---000

ex: 2:
n=6 : 00---110
n-1=5 : 00---101
00---100

find the only odd occurring number:

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

O/p: 3

2) I/p: arr[ ] = {8, 7, 7, 8, 3}

O/p: 3

Naive:

```
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]);
}
```

T.C = O(n²)

efficient solution:

T.C = O(n)
Auxiliary space = O(1)

XOR

- x ^ 0 = x
- x ^ y = y ^ x
- x ^ (y ^ z) = (x ^ y) ^ z
- x ^ x = 0

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

Variation Question:

Given an array of n number that has values in range [1, n+1]. Every no. appears exactly once. Hence one no is missing find the missing no.

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

O/P: 2

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

O/P: 4

T.C = O(n)
AS = O(1)

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

find two odd appearing numbers

I/p array size must be atleast 2.

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: T.C = $\Theta(n^2)$

```
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]);
    }
}
```

efficient: T.C: $\Theta(n)$

① X-OR of all the numbers

XOR = 3^4^3 - - - - - 17

= 5^6

= 3

5: 101

^6: 110

011

group 1 = {3, 3, 5, 7, 7} = 5

group 2 = {4, 4, 4, 4, 6} = 6

3 : 00...011

3-1=2 : 00....010

~(3-1) : 11...101

3 & ~(3-1) = (00...011) & (11...101)

= (0....010)

Efficient method ⇒ O(n)

```
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); // Right most SetBit
  for (int i=0; i<n; i++)
  {
    if ((arr[i] & Sn) != 0)
      res1 = res1 ^ arr[i];
    else
      res2 = res2 ^ arr[i];
  }
  print (res1, res2);
}
```

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

o/p: 8 32

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

= 8^32

8 : 0....0001000

32 : 0...0100000

XOR : 0...0101000

: 40

XOR-1 : 0...0100111

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

& 00...01000

Sn = 8

res1 = 8

res2 = 3^4^3^4^8^4^4^32^7^7

= 32

(res1, res2) = (8, 32)

Power set using Bitwise operators

I/P: s = "abc"

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

I/P: s = "ab"

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

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 $2^n - 1$

ordering o/p

s = "ab"

0	00	""
1	01	"a"
2	10	"b"
3	11	"ab"

Value Varies from 0 to $2^n - 1$

```
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++)
    { if (counter & (1<<j) != 0)
        print (str[j]);
    }
    print ("\n");
  }
```

$T.C = \theta(2^n * n)$

ex: counter = 3
j = 0 to j = 2

j = 0, 1<<j = 1 ⇒ a

j = 1, 1<<j = 2 ⇒ b

j = 2, 1<<2 = 4 ⇒ "ab"