Exclusive-OR Operator (^).

Exclusive-OR returns a bit value of 1 if both bits are of opposite (different), nature , otherwise Exclusive-OR returns 0

XOR is really a surprising operator. You can't imagine the things that make it possible for us to do.

Remember the properties of XOR operator:

• If you take xor of a number with 0 (zero), it would return the same number again.

Means, n ^ 0 = n

Example:

6 ^ 0 = 6 (because 110 ^ 000 = 110)

27 ^ 0 = 27 (because 11011^ 00000 = 11011)

• If you take xor of a number with itself, it would return 0 (zero).

Means, n^ n = 0

Example:

6 ^ 6 = 0 (because 110 ^ 110 = 000)

27 ^ 27 = 0 (because 11011^11011 = 00000)

1. We can swap the values of two variables without using any third (temp) variable.

Example: **a = 5, b = 10**

temp = a | a = a ^ b (a = 5 ^ 10 = 0101 ^ 11010 = 1111 = 15)

a = b | b= a^ b (b = 15 ^ 10 = 1111 ^ 1010 = 0101 = 5)

b = temp | a = a^b (a = 15 ^ 5 = 1111 ^ 0101 = 1010 = 10)

a = 10, b = 5

2. Toggling (Changing/Flipping) the k-th Bit (from right) of a binary number :

Question : For a given number "n", toggle the k-th bit (from right)

Solution : Let, n = 27 (Binary Representation of n = 11011) K = 3

We'll use this expression for toggling: n ^ (1 << (k-1)).

Means, 11011 ^ (00001 << 2)

11011 ^ (00100)

11011 ^ 00100

11111 (Decimal of 11111 = 31)

3. Find the Missing number from the list of numbers :

Question : You are given a list of n-1 integers , And these integers are in the range of 1 to n. There are no duplicates in the list. One of the integers is missing in the list. Now, we need to find that missing number.

• **Method 1: By finding the sum of first "n" natural numbers.**

Step 1: First , find the sum of all numbers from 1 to n (means find the sum of first "n" natural numbers)

Step 2: Subtract each element (all elements) of the given list from the sum. And we'll get the missing number.

*** There might be an integer overflow while adding large numbers.**

• **Method 2 : Using XOR operator.**

Step 1 : We'll take the xor of all numbers from 1 to n (Means , we'll take xor of the first "n" natural numbers).

Step 2: We'll take the xor of all elements of the given array (list).

Step 3: xor of Step 1 and Step 2 will give the required missing number.

Example: Given list arr = [4, 2, 1, 6, 8, 5, 3, 9] n = 9 (given)

Step 1: Step1_result = 1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8 ^ 9 = 1

Step 2: Step2_result = 4 ^ 2 ^ 1 ^ 6 ^ 8 ^ 5 ^ 3 ^ 9 = 6

Step 3: Final Result = Step1_result ^ Step2_result = 1 ^ 6 = 7

But , How Final_Result calculated the missing number?

Final_Result = (1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8 ^ 9) ^ (4 ^ 2 ^ 1 ^ 6 ^ 8 ^ 5 ^ 3 ^ 9).

Remember these properties : n ^ n= 0 and n ^ 0 = n

So, here,

Final_Result = (1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8 ^ 9) ^ (4 ^ 2 ^ 1 ^ 6 ^ 8 ^ 5 ^ 3 ^ 9).

= 1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8 ^ 9 ^ 4 ^ 2 ^ 1 ^ 6 ^ 8 ^ 5 ^ 3 ^ 9.

= (1 ^ 1) ^ (2 ^ 2) ^ (3 ^ 3) ^ (4 ^ 4) ^ (5 ^ 5) ^ (6 ^ 6) ^ (7) ^ (8 ^ 8) ^ (9 ^ 9)

= 0 ^ 0 ^ 0 ^ 0 ^ 0 ^ 0 ^ 7 ^ 0 ^ 0 (property applied)

= 0 ^ 7 (because we know 0 ^ 0 = 0).

= 000 ^ 111

= 111

= 7 (Required Result)

4. Find the duplicate element in the given array of all positive integers.

Input_arr = [23, 21, 24, 27, 22, 27, 26, 25] *Output = 27*

Method 1: By finding “min”, “max” and “sum”. —— o(n)

Step 1: First , find the sum of all numbers from “min” to “max”.

Step 2 : Subtract each element (all elements) of the given array from the sum.

And At last, we would be having a negative of the duplicate element in sum. Then , Just return the absolute value of that negative value in sum.

Method 2: Using XOR Operator

Step 1: Find “min” and “max" values in the given array. —— O(n)

Step 2: Find XOR of all integers from range “min” to “max” (inclusive).

Step 3: Find XOR of all elements of the given array.

Step 4: XOR of Step 2 and Step 3 will give the required duplicate number.

5. Construct an array from XOR of all elements of the array except elements at the same index.

Given an array A[] having "n" positive elements. The task is to create another array B[ ] such as B[i] is XOR of all elements of the array A[ ] except A[i].

Example : A[] = { 2 , 1 , 5 , 9 }   O/P = B[] = { 13 , 14 , 10 , 63 }

13: 1 ^ 5 ^ 9
14: 2 ^ 5 ^ 9
10: 2 ^ 1 ^ 9
63: 2 ^ 1 ^ 5

Efficient Solution : Using XOR Operator

Step 1: Find XOR of all elements of the given array.

Step 2: Now, for each element of A[], Calculate A[i] = Step1_result ^ A[i]

Time Complexity : O(n) [because for loop will run for each element of the array]

```
int Step1_result = A[0];
int n =  sizeof(A) / sizeof(A[0]);       // Calculating size of the array A
for(int i=1;i<n;i++)                     // Step 1
        Step1_result = Step1_result ^ A[i];

for(int i=0;i<n;i++)                     // Step 2
        A[i] = Step1_result ^ A[i];

return A;
```

Remember

- n ^ n = 0
- n ^ 0 = n

- Example : A[] = { 2 , 1 , 5 , 9 }   n = 4   O/P = B[] = { 13 , 14 , 10 , 63 }

Step1 : Step1_result = 2 ^ 1 ^ 5 ^ 9 = 15

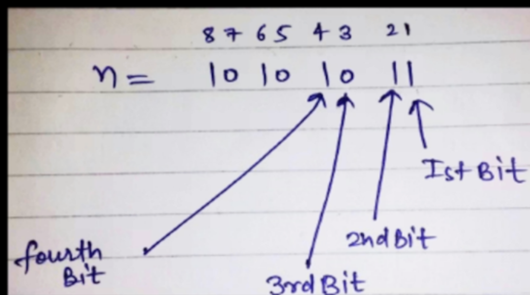
Step2 : for i = 0 , A[0] = 15 ^ A[0] = 15 ^ 2 = 13

(2 ^ 1 ^ 5 ^ 9) ^ 2 = (2 ^ 2) ^ 1 ^ 5 ^ 9 = 0 ^ 1 ^ 5 ^ 9 = 1 ^ 5 ^ 9 = 13

Similarly , for each element of A[ ]

Question : Given a Binary number n = 10101011 , Write a program to check whether k-th bit (from right) is set (Means , 1 at k-th position) or not ?

Solution : n = 10101011 , Let k = 4 (Given in Question)

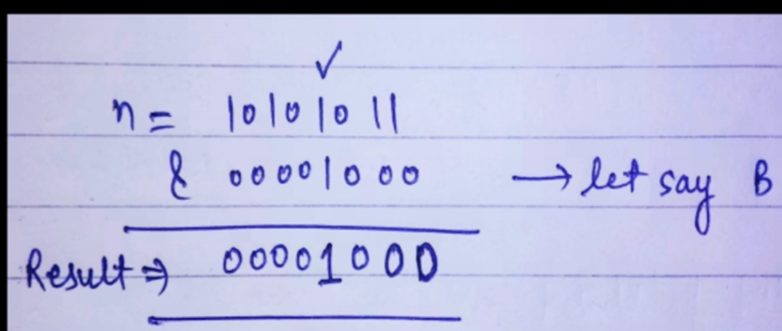


So , n = 1010**1**011 (the 1 which is bigger is 4^{th}-bit from right)

- The idea is very simple and very efficient :

We need to check only whether 4^{th} bit (from right) is set or not. We do not need to think about all rest bits (except 4^{th} one).

So , the approach is to make all the rest bits of "n" as 0 (except the 4^{th} bit).

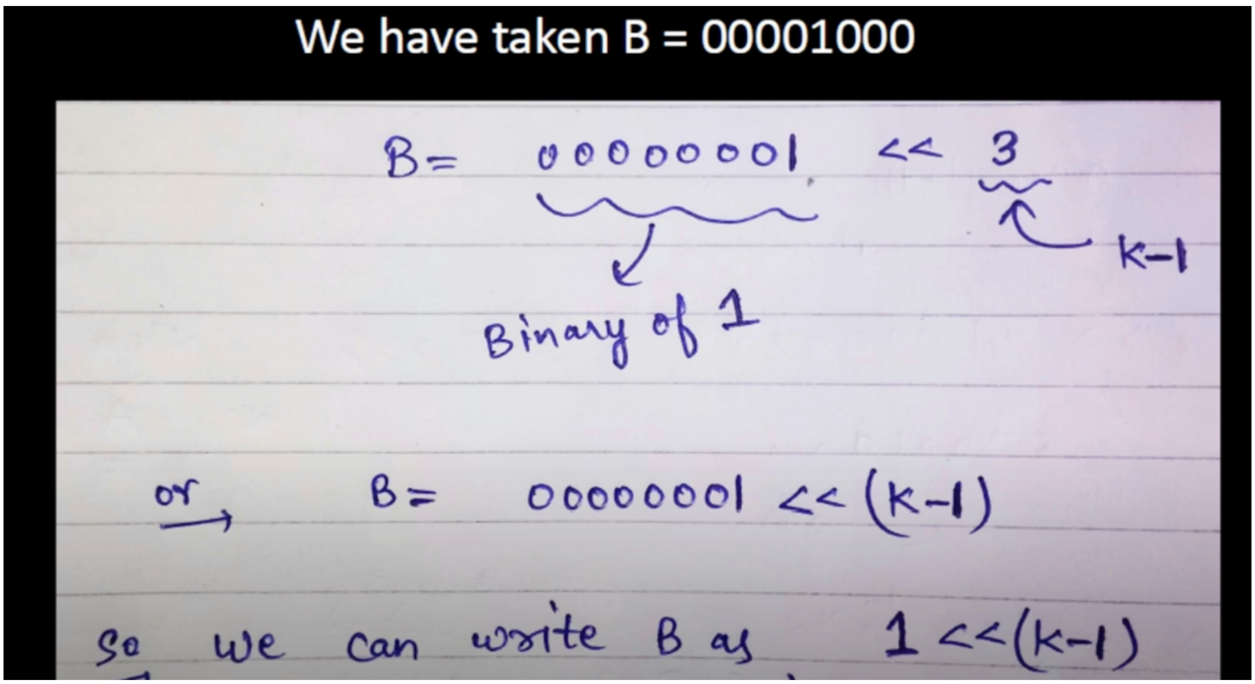


So, If we notice the result :

result = 00001000

And as we know that, after performing "& operation” on two numbers, the result will be 0 (zero) only if all the bits of the result are zero. Otherwise, the result will be non-zero.

Means , Here , result will return “non-zero value” because 4th bit (from right is not zero. So, It means, 4th bit is set 4th bit is 1) in given n = 10101011




```
result = n & (1<<(k-1));
if(result != 0 ){
  System.out.println("k-th bit is SET");
}else{
  System.out.println("k-th bit is NOT SET");
}
```

Bitwise Left Shift Operator (<<)

- **Bitwise Left Shift Operator takes two operands , like : X << k**

  Here , X is the first operand , k is the second operand

- **Then , bitwise left shift operator shifts bits of first operand to the left by kth positions and fills empty / vacated bit positions with 0**

Bitwise Left Shift Operator moves / shifts all the bits of a number to the left by the specified position and fills empty / vacated bit positions with 0

Example : X << k (X is a Decimal number, k is the specified position)

X= 23 (Given Decimal Number), k= 2 (specified position)

Binary Equivalent of 23 is : 10111

And if number X is stored as a 32-bit "int" , then

Binary Equivalent would be : **00**000000 00000000 00000000 00010111

Now, shift 10111 to the left by 2 (specified position)

So, 10111 << 2 : 00000000 00000000 00000000 01011100

Now , Decimal Equivalent of 10111 << 2 would be : 92

Remember

• Integers are stored in the memory as a series of bits. So, for example , if integer number 6 is stored as a 32-bit “int”, then

Binary Equivalent of 6 would be : 00000000 00000000 00000000 00000110

• Bitwise Left Shift Operator should not be used for negative numbers (Means

if Any of the operand of Bitwise left shift operand is negative), otherwise it might result undefined value.

• This Bitwise Left Shift Operator can be applied to int, long , and possibly short, byte , or char

• If Any number is shifted more than the size of integers, then the behavior /result might also be undefined.

Example: If the integers are stored using 32-bits ,

then X <<33, X << 34, X << 35, X << 36 , ... and so on might result in undefined value.

Inference / Conclusion

1. **5 << 3 = 40 | 40 = 5 * 8 = 5 * (2^3) (^ mean power)**
2. **2 << 3 = 16 | 16 = 2 * 8 = 2 * (2^3) (^ mean power)**
3. **23 << 2 = 92 | 92 = 23 * 4 = 23 * (2^2) (^ mean power)**
4. **6 << 1 = 12 | 12 = 6 * 2 = 6 * (2^1) (^ mean power)**
5. **6 << 3 = 48 | 48 = 6 * 8 = 6 * (2^3) (^ mean power)**

- **In X << k, Bitwise Left Shift Operator multiplies the number X with (2^k)**