BIT MANIPULATION

Lesson 01

Operations with bits are used in Data compression (data is compressed by converting it from one representation to another, to reduce the space), Exclusive-or Encryption (an algorithm to encrypt the data for safety issues). In order to encode, decode or compress files we have to extract the data at bit level. Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

We all know that 1 byte comprises 8 bits and an integer or character can be represented using bits in computers, which we call its binary form(contains only 1 or 0) or in its base 2 form.

Example:

1) $14 = \{1110\}_2$
$= 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$
$= 14.$

BITWISE OPERATORS

AND (&):- Binary AND Operator copies a bit to the result if it exists in both operands. E.g. (1000 & 1000) = 1000.

OR (|):- Binary OR Operator copies a bit if it exists in either operand.

E.g (0011 | 1101) = 1111.

XOR(^):- Binary XOR Operator copies the bit if it is set in one operand but not both. E.g (0011 ^ 1101) = 1110.

NOT (~) :- Binary One's Complement Operator is unary and has the effect of 'flipping' bits. E.g (~ 0011) = 1100.

Left Shift (<<):- Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. E.g. (0001 << 1) = 0010.

Right Shift (>>);- Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. E.g. (1100 << 1) = 0110.

Problem Solving

1) Write a programme to print binary representation of a given number.

Approach :-

```
step 1: Check n > 0
step 2: Right shift the number by 1 bit and recursive
function call
step 3: Print the bits of number
```

Code :-

```
#include <bits/stdc++.h>
using namespace std;

void bin(unsigned n)
{
    if (n > 1)
```

```
        bin(n >> 1);

    printf("%d", n & 1);
}

// Driver code
int main(void)
{
    bin(131);
    printf("\n");
    bin(3);
    return 0;
}
```

2) How to check if a given number is a power of 2?

Property:-

Properties for numbers that are powers of 2 is that they have one and only one bit set in their binary representation. If the number is neither zero nor a power of two. It will have 1 in more than one place. So if X is a power of 2 then x & (x - 1) will be 0.

It might not seem obvious with these examples, but a binary representation of (x-1) can be obtained by simply flipping all the bits to the right of rightmost 1 in x and also including the rightmost 1.

Now think about x & (x-1). x & (x-1) will have all the bits equal to the x except for the rightmost 1 in x.

Code :-

```
bool PowerOfTwo(int x)
  {
```

```
    If !( X & (X - 1) return true;

    return false;
}
```

3) Count the number of ones in the binary representation of the given number.

Property:-

As explained in the previous algorithm, the relationship between the bits of x and x-1. So as in x-1, the rightmost 1 and bits right to it is flipped, then by performing x&(x-1), and storing it in x, will reduce x to a number containing a number of ones(in its binary form) less than the previous state of x, thus increasing the value of count in each iteration.

Code :-

```
int count_one (int n)
    {
       while( n )
       {
       n = n&(n-1);
         count++;
       }
       return count;
}
```

4) **Find the Element that appears once.**

Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. The expected time complexity is O(n) and O(1) extra space.

Approach:-

Following is another O(n) time complexity and O(1) extra space method suggested by *aj*. We can sum the bits in the same positions for all the numbers and take modulo with 3. The bits for which sum is not multiple of 3, are the bits of number with a single occurrence.

Let us consider the example array {5, 5, 5, 8}. The 101, 101, 101, 1000

Sum of first bits%3 = (1 + 1 + 1 + 0)%3 = 0;

Sum of second bits%3 = (0 + 0 + 0 + 0)%3 = 0;

Sum of third bits%3 = (1 + 1 + 1 + 0)%3 = 0;

Sum of fourth bits%3 = (1)%3 = 1;

Hence number which appears once is 1000

Code:-

```
#include <bits/stdc++.h>
using namespace std;
#define INT_SIZE 32

int getSingle(int arr[], int n)
```

```
{
    // Initialize result
    int result = 0;

    int x, sum;

    // Iterate through every bit
    for (int i = 0; i < INT_SIZE; i++) {

        // Find sum of set bits at ith position in
all
        // array elements
        sum = 0;
        x = (1 << i);
        for (int j = 0; j < n; j++) {
            if (arr[j] & x)
                sum++;
        }

        // The bits with sum not multiple of 3, are
the
        // bits of element with single occurrence.
        if ((sum % 3) != 0)
            result |= x;
    }

    return result;
}

int main()
{
    int arr[] = { 12, 1, 12, 3, 12, 1, 1, 2, 3, 2,
2, 3, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "The element with single occurrence is "
<< getSingle(arr, n);
    return 0;
```

```
}
```

5) Find the number occurring an odd number of times.

Given an array of positive integers. All numbers occur an even number of times except one number which occurs an odd number of times. Find the number in O(n) time & constant space.

Approach:-

A **Simple Solution** is to run two nested loops. The outer loop picks all elements one by one and the inner loop counts the number of occurrences of the element picked by the outer loop. The time complexity of this solution is $O(n^2)$.

The **Best Solution** is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring elements. Please note that the XOR of two elements is 0 if both elements are the same and the XOR of a number x with 0 is x.

Code :-

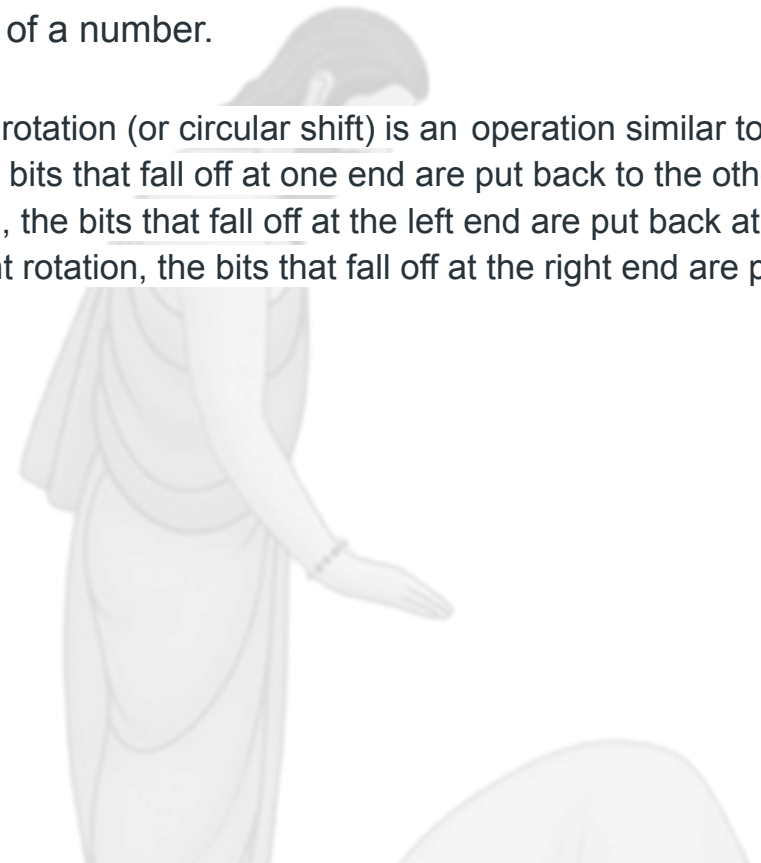
```
#include <bits/stdc++.h>
using namespace std;

int getOddOccurrence(int ar[], inar_size)
{
    int res = 0;
    for (int i = 0; i < ar_size; i++)
```

```
        res = res ^ ar[i];

    return res;
}
/* Driver function to test above function
*/
int main()
{
    int ar[] = {2, 3, 5, 4,5,2,4,3,5,2};

    int n = sizeof(ar)/sizeof(ar[0]);

    // Function calling
    cout << getOddOccurrence(ar, n);

    return 0;
}
```

6) Rotate bits of a number.

Bit Rotation: A rotation (or circular shift) is an operation similar to a shift except that the bits that fall off at one end are put back to the other end. In the left rotation, the bits that fall off at the left end are put back at the right end. In the right rotation, the bits that fall off at the right end are put back at the left end.

Example:



Let n is stored using 8 bits. Left rotation of n = 11100101 by 3 makes n = 00101111 (Left shifted by 3 and the first 3 bits are put back in last). If n is stored using 16 bits or 32 bits then the left rotation of n (000…11100101) becomes 00..00**11100101**000.

Right rotation of n = 11100101 by 3 makes n = 10111100 (Right shifted by 3 and last 3 bits are put back in first) if n is stored using 8 bits. If n is stored using 16 bits or 32 bits then the right rotation of n (000…11100101) by 3 becomes **101**000..00**11100**.

Code :-

```
#include<iostream>

using namespace std;
#define INT_BITS 32
class gfg
{
public:
int leftRotate(int n, unsigned int
d)
{

    return (n << d)|(n >> (INT_BITS
- d));
}

/*Function to right rotate n by d
bits*/
int rightRotate(int n, unsigned int
d)
{

    return (n >> d)|(n << (INT_BITS
- d));
}
};

/* Driver code*/
```

```
int main()
{
    gfg g;
    int n = 16;
    int d = 2;
    cout << "Left Rotation of " << n
<<
            " by " << d << " is ";
    cout << g.leftRotate(n, d);
    cout << "\nRight Rotation of "
<< n <<
            " by " << d << " is ";
    cout << g.rightRotate(n, d);
    getchar();
}
```

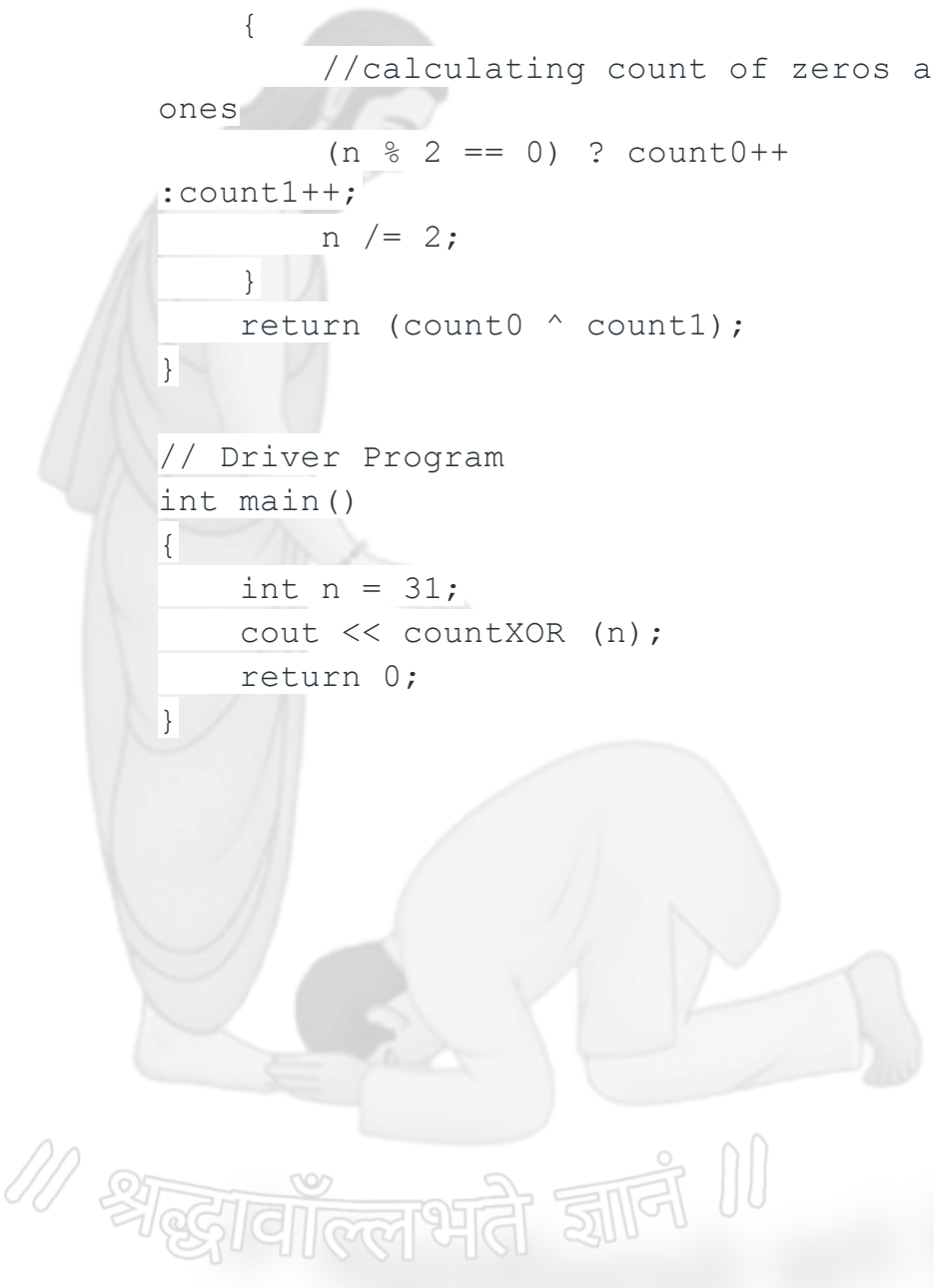
7) XOR counts of 0s and 1s in the binary representation of a number.

Given a number, the task is to find XOR of the count of 0s and count of 1s in the binary representation of a given number.

Approach:- The idea is simple, we traverse through all bits of a number, count 0s and 1s and finally return XOR of two counts.

Code:-

```
#include<iostream>
using namespace std;

int countXOR(int n)
{
    int count0 = 0, count1 = 0;
    while (n)
    {
        //calculating count of zeros and
ones
        (n % 2 == 0) ? count0++
:count1++;
        n /= 2;
    }
    return (count0 ^ count1);
}

// Driver Program
int main()
{
    int n = 31;
    cout << countXOR (n);
    return 0;
}
```



8) Multiply a given number with 3.5.

Given an integer x, write a function that multiplies x with 3.5 and returns the integer result. You are not allowed to use %, /, *.

Approach:-

We can get x*3.5 by adding 2*x, x, and x/2. To calculate 2*x, left shift x by 1, and to calculate x/2, right shift x by 2.

Code:-

```
#include <bits/stdc++.h>


int multiplyWith3Point5(int x)
{
    return (x<<1) + x + (x>>1);
}


/* Driver program to test above
functions*/
int main()
{
    int x = 4;
    printf("%d", multiplyWith3Point5(x));
    getchar();
    return 0;
```

```
}
```

9) Check if a given number is even or odd using a bitwise operator.

Given a number **N**, the task is to check whether the number is even or odd using a bitwise operator.

Approach:-

The idea is to check whether the last bit of the number is set or not. If the last bit is set then the number is odd, otherwise even.

As we know bitwise AND Operation of the Number by 1 will be 1, If it is odd because the last bit will be already set. Otherwise, it will give 0 as output.

Code:-

```
#include <iostream>
using namespace std;

// Returns true if n is even, else odd
bool isEven(int n)
{
    // n&1 is 1, then odd, else even
    return (!(n & 1));
```

```
}

// Driver code
int main()
{
    int n = 101;
    isEven(n)
    ? cout << "Even"
    : cout << "Odd";
    return 0;
}
```

10) Count minimum bits to flip such that XOR of A and B equal to C.

Given a sequence of three binary sequences A, B, and C of N bits. Count the minimum bits required to flip in A and B such that XOR of A and B is equal to C.For **Example :**

```
Input: N = 3
       A = 110
       B = 101
       C = 001
Output: 1
We only need to flip the bit of 2nd position
of either A or B, such that A ^ B = C i.e.,
100 ^ 101 = 001
```

Approach:-

A **Naive approach** is to generate all possible combinations of bits in A and B and then XORing them to Check whether it is equal to C or not. **The time complexity** of this approach grows exponentially so it would not be better for a large value of N.

An **Efficient** approach is to use the concept of XOR.

If we generalize, we will find that at any position of A and B, we just only need to flip i^{th} (0 to N-1) position of either A or B otherwise we will not able to achieve the minimum no of Bits.

So at any position of i (0 to N-1) you will encounter two type of situation i.e., either A[i] == B[i] or A[i] != B[i]. Let's discuss it one by one.

- If A[i] == B[i] then XOR of these bits will be 0, two cases arise in C[]: C[i]==0 or C[i]==1.

  If C[i] == 0, then no need to flip the bit but if C[i] == 1 then we have to flip the bit either in A[i] or B[i] so that 1^0 == 1 or 0^1 == 1.

- If A[i] != B[i] then XOR of these Bits gives a 1, In C two cases again arise i.e., either C[i] == 0 or C[i] == 1.

  Therefore if C[i] == 1, then we need not to flip the bit but if C[i] == 0, then we need to flip the bit either in A[i] or B[i] so that 0^0==0 or 1^1==0

Code:-

```
#include<bits/stdc++.h>
```

```
using namespace std;



int totalFlips(char *A, char *B, char *C, int N)

{

    int count = 0;

    for (int i=0; i < N; ++i)

    {

        // If both A[i] and B[i] are equal

        if (A[i] == B[i] && C[i] == '1')
```

```
            ++count;



        // If Both A and B are unequal

        else if (A[i] != B[i] && C[i] == '0')

            ++count;

    }

    return count;

}
```

```
int main()

{

    //N represent total count of Bits

    int N = 5;

    char a[] = "10100";

    char b[] = "00010";

    char c[] = "10011";



    cout << totalFlips(a, b, c, N)return 0;}
```