

**SYLLABUS**

**Bit Manipulation:** Introduction, Applications: Counting Bits, Palindrome Permutation, Remove All Ones with Row and Column Flips, Encode Number.



## Bit Operations

**And Operation** The *and* operation  $x \& y$  produces a number that has '1' bit in positions where both  $x$  and  $y$  have '1' bits. For example,  $22 \& 26 = 18$ , because

$$\begin{array}{r} 10110 \text{ (22)} \\ 11010 \text{ (26)} \& \\ \hline = 10010 \text{ (18)}. \end{array}$$

Using the **and** operation, we can check if a number  $x$  is even because  $x \& 1 = 0$  if  $x$  is even, and  $x \& 1 = 1$  if  $x$  is odd. More generally,  $x$  is divisible by  $2^k$  exactly when  $x \& (2^k - 1) = 0$ .

**Or Operation** The *or* operation  $x | y$  produces a number that has '1' bit in positions where at least one of  $x$  and  $y$  have '1' bits. For example,  $22 | 26 = 30$ , because

$$\begin{array}{r} 10110 \text{ (22)} \\ 11010 \text{ (26)} | \\ \hline 11110 \text{ (30)} \end{array}$$

**Xor Operation** The *xor* operation  $x \wedge y$  produces a number that has '1' bit in positions where exactly one of  $x$  and  $y$  have '1' bits. For example,  $22 \wedge 26 = 12$ , because

$$\begin{array}{r} 10110 \text{ (22)} \\ 11010 \text{ (26)} \\ \hline 01100 \text{ (12)} \end{array}$$

**Not Operation:** The *not* operation  $\sim x$  produces a number where all the bits of  $x$  have been inverted. For example,  $\sim 29$  is 30. The result of the not operation at the bit level depends on the length of the bit representation, because the operation inverts all bits. For example, if the numbers are 32-bit integers, the result is as follows:

$$\begin{aligned} x &= 29 \rightarrow 000000000000000000000000011101 \\ \sim x &= -30 \rightarrow 111111111111111111111111100010 \end{aligned}$$

**Bit Shifts:** The *left shift*  $x \ll k$  appends  $k$  zero bits to the number, and the *right shift*  $x \gg k$  removes the  $k$  last bits from the number.

For example,  $14 \ll 2 = 56$ , because 14 and 56 correspond to 1110 and 111000.

Similarly,  $49 \gg 3$  is 6, because 49 and 6 correspond to 110001 and 110.

$x \ll k$  corresponds to multiplying  $x$  by  $2^k$ , and

$x \gg k$  corresponds to dividing  $x$  by  $2^k$  rounded down to an integer.

## What is Bit Manipulation?

Bit manipulation is the process of applying logical operations on a sequence of bits, the smallest form of data in a computer, to achieve a required result. Bit manipulation has constant time complexity and process in parallel, meaning it is very efficient on all systems.

Most programming languages will have you work with abstractions, like *objects* or *variables*, rather than the bits they represent. However, direct bit manipulation is needed to improve performance and reduce error in certain situations.

Bit manipulation requires a strong knowledge of *binary* and *binary conversion*.

### Here's a few examples of tasks that require bit manipulation:

- Low-level device control
- Error detection and correction algorithms
- Data compression
- Encryption algorithms
- Optimization

**For example**, take a look at the difference between an arithmetic and bit manipulation approach for finding the green portion of an RGB value:

```
//arithmetic  
(rgb / 256) % 256
```

```
//bit  
(rgb >> 8) & 0xFF
```

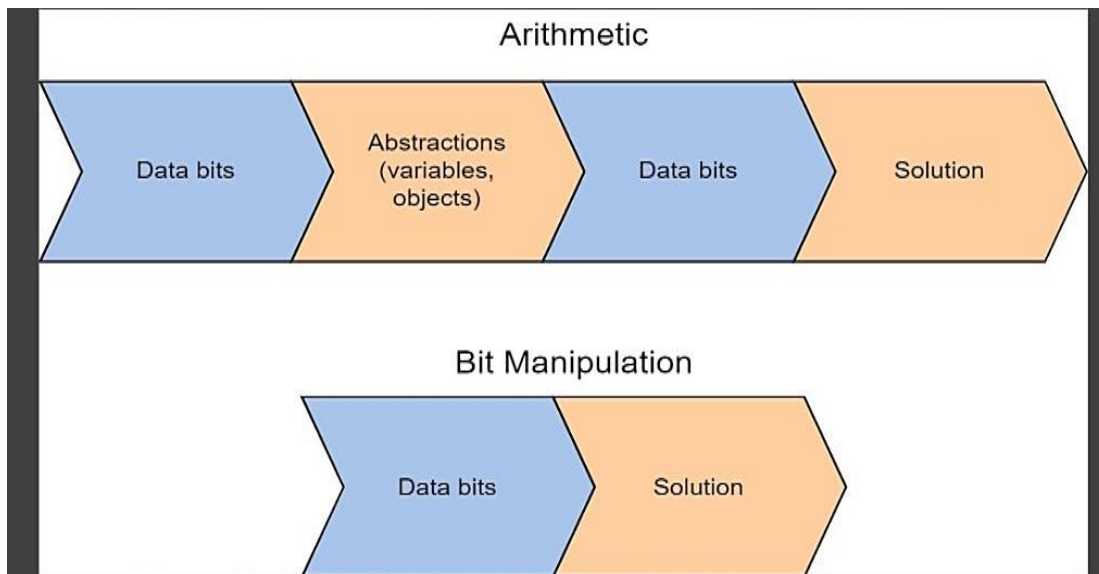
While both do the same thing, the second option is considerably faster, as it works directly within memory rather than through a level of abstraction.

We'll explore what each of these operators do later in this article (>> and &).

## Bitwise Operators

Bitwise operations take one or more bit-patterns or binary numerals and **manipulate them at the bit level**. They're essentially our tool to manipulate bits to achieve our operations.

While arithmetic operations perform operations on human-readable values (1+2), bitwise operators manipulate the low-level data directly.



*Difference in steps between arithmetic and bit operations*

### Advantages

- They are fast and simple actions.
- They are directly supported by the processor.
- They are used to manipulate values for comparisons and calculations.
- Bitwise operations are incredibly simple and faster than arithmetic operations.

### List of Bitwise operators

Operator	Name of Operator	Usage
&	Bitwise AND	Used to mask particular part of byte
	Bitwise OR	
~	One's complement/NOT	Used to turn a bit on/off
^	Bitwise XOR	
<<	Left Shift	Used to shift the bit to the left
>>	Right Shift	Used to shift the bit to the right

**Bitwise AND [ & ]:**

**AND (&)** is a binary operator that compares two operands of equal length. The operands are converted from their readable form to binary representation. For each bit, the operation checks if both bits are 1 across both operands. If yes, that bit is set to 1 in the answer. Otherwise, the corresponding result bit is set to 0.

It essentially multiplies each bit by the corresponding bit in the other operand. As multiplying anything by 0 results in 0, the AND comparison with any 0 bit will result in 0.

If two input bits are 1, the output is 1.

In all other cases its 0, for example:

1 & 0 => yields to 0.

0 & 1 => yields to 0.

0 & 0 => yields to 0.

0101 (decimal 5) AND 0011 (decimal 3)

0\*0=00\*0=0

1\*0=01\*0=0

0\*1=00\*1=0

1\*1=11\*1=1

Therefore:

= 0001 (decimal 1)

The operation may be used to determine whether a particular bit is set (1) or clear (0).

It's also used to clear selected bits of a register in which each bit represents an individual Boolean state.

**Example:**

```
class AndOperation
```

```
{
    public static void main( String args[] )
    {
        int x = 12;
        int y = 10;
        System.out.println("Bitwise AND of (" + x + " , " + y + ") is: " + (x & y)); // yields to 8
    }
}
```

**Output:** Bitwise AND of (12 , 10) is: 8

**Bitwise OR( | ):**

The OR operator (|) is a binary operator that takes two equal-length operands but **compares them** in the opposite way to AND; if either corresponding bit is 1, the answer is 1. Otherwise, the answer will be 0. In other words, Bitwise OR returns '1' if one of the inputs given is 1.

If two input bits are 0, the output is 0.

In all other cases, it is 1. For example:

1 | 0 => yields to 1.

0 | 1 => yields to 1.

1 | 1 => yields to 1.

a = 12

b = 10

-----  
a in Binary : 0000 0000 0000 1100

b in Binary : 0000 0000 0000 1010

-----  
a | b : 0000 0000 0000 1110  
-----

This is often used as an interim logic step for solving other problems.

**Example:**

class OROperation

```
{  
    private static int helper(int x, int y)  
    {  
        return x | y;  
    }  
    public static void main(String[] args)  
    {  
        int x = 12;  
        int y = 10;  
        System.out.println("Bitwise OR of " + x + ", " + y + " is: " + helper(x, y)); // yields to 14  
    }  
}
```

**Output:**

Bitwise OR of 12, 10 is: 14

## NOT (~) Operator

NOT (~), or sometimes called the bitwise complement operator, is a unary operation that takes a single input and **swaps each bit** in its binary representation to the opposite value.

All instances of 0 become 1, and all instances of 1 become 0. In other words, NOT inverts each input bit. This inverted sequence is called the **one's complement** of a bit series.

For example, consider  $x = 1$

The binary number representation of  $x$  is:

$x = 00000000\ 00000000\ 00000000\ 00000001$

Now, Bitwise NOT of  $x$  will be:

$x = 11111111\ 11111111\ 11111111\ 11111110$

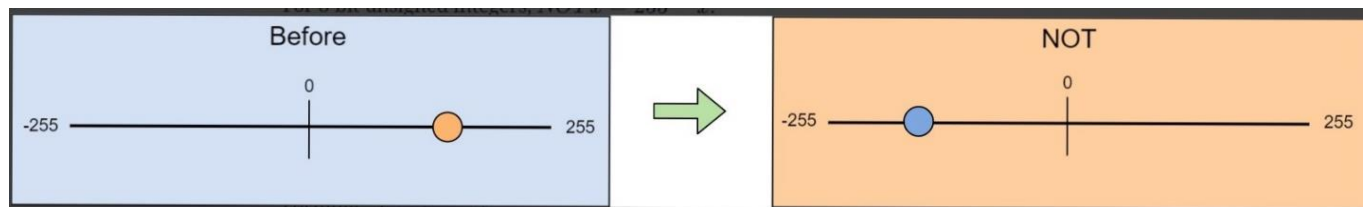
So:  $x$  contains 31 zeros(0's) and one 1

$\sim x$  contains 31 ones(1's) and one 0(zero)

This makes the number negative as any bit collection that starts with 1 is negative.

NOT is useful for flipping unsigned numbers to the mirrored value on the opposite side of their midpoint.

For 8-bit unsigned integers,  $NOTx = 255 - x$ .



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

### Example:

class NOTOperation

```
{
    public static void main( String args[] )
    {
        int a = 1;
        System.out.println("Bitwise NOT of a is : " + ~a);
    }
}
```

### **Output:**

Bitwise NOT of a is : -2



## Bitwise Exclusive OR (XOR) (^) Operator

The bitwise XOR operation (^), short for “Exclusive-Or”, is a binary operator that takes two input arguments and **compares each corresponding bit**. If the bits are opposite, the result has a 1 in that bit position. If they match, a 0 is returned.

$1 \wedge 1 \Rightarrow$  yields to 0.

$0 \wedge 0 \Rightarrow$  yields to 0.

$1 \wedge 0 \Rightarrow$  yields to 1.

$0 \wedge 1 \Rightarrow$  yields to 1.

For example:

a = 12

b = 10

-----  
a in binary : 0000 0000 0000 1100

b in binary : 0000 0000 0000 1010

-----  
a ^ b : 0000 0000 0000 0110  
-----

XOR is used to invert selected individual bits in a register or manipulate bit patterns that represent Boolean states.

XOR is also sometimes used to set the value of a registry to zero as XOR with two of the same input will always result in 0.

Example:

```
class XOROperation
```

```
{  
    public static void main( String args[] )  
    {  
        int x = 12;  
        int y = 10;  
        System.out.println("Bitwise XOR of (x , y) is : " + (x ^ y)); // yields to 6  
    }  
}
```

**Output:**

Bitwise XOR of (x , y) is : 6

## Bitwise Operators comparison

Below is a table showing a comparison of results of all the bitwise operators mentioned above based on different values of the compared bits (A and B).

A	B	A & B	A   B	A ^ B
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0
0	0	0	0	0

**Bitwise Shift Operators****Left and right shift operator**

A bit shift is a Bitwise operation where the order of a series of bits is moved to efficiently perform a mathematical operation. A bit shift moves each digit in a number's binary representation left or right by a number of spaces specified by the second operand.

These operators can be applied to integral types such as int, long, short, byte, or char.

**There are three types of shifts:**

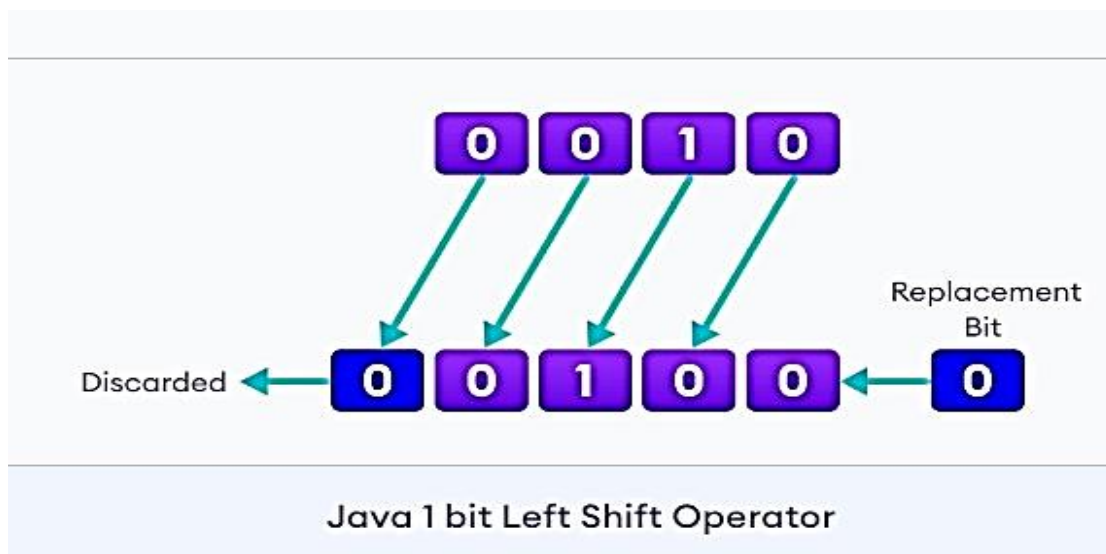
**Left shift:** << is the left shift operator and meets both logical and arithmetic shifts' needs.

**Arithmetic/signed right shift:** >> is the arithmetic (or signed) right shift operator.

**Logical/unsigned right shift:** >>> is the logical (or unsigned) right shift operator.

In Java, all integer data types are signed and << and >> are solely arithmetic shifts.

Here's an example of a left shift:



**6**=00000000 00000000 00000000 00001110

Shifting this bit pattern to the left one position (**6** << 1) results in the number 12:

**6**<<1=00000000 00000000 00000000 00011100

As you can see, the digits have shifted to the left by one position, and the last digit on the right is filled with a zero. Note that shifting left is equivalent to multiplication by powers of 2.

$6 \ll 1 \rightarrow 6 * 2^1 \rightarrow 6 * 2$

$6 \ll 3 \rightarrow 6 * 2^3 \rightarrow 6 * 8$

Well-optimized compilers will use this rule to replace multiplication with shifts whenever possible, as shifts are faster.

**Example:**

```

class LeftShift
{
    private static int helper(int number, int i)
    {
        return number << i; // multiplies `number` with 2^i times.
    }
    public static void main(String[] args)
    {
        int number = 100;
        System.out.println(number + " shifted 1 position left, yields to " + helper(number, 1));
        System.out.println(number + " shifted 2 positions left, yields to " + helper(number, 2));
        System.out.println(number + " shifted 3 positions left, yields to " + helper(number, 3));
        System.out.println(number + " shifted 4 positions left, yields to " + helper(number, 4));
    }
}

```

**Output:**

```

100 shifted 1 position left, yields to 200
100 shifted 2 positions left, yields to 400
100 shifted 3 positions left, yields to 800
100 shifted 4 positions left, yields to 1600

```

With right shift, you can either do arithmetic (>>) or logical (>>) shift.

The difference is that arithmetic shifts maintain the same most significant bit (MSB) or **sign bit**, the leftmost bit which determines if a number is positive or negative.

$10110101 \gg 1 = 1101\ 1010$ .

Formula:  $x \gg y = x/2^y$

On the other hand, a logical shift simply moves everything to the right and replaces the MSB with a 0.

$10110101 \ggg 1 = 01011010$

Formula:  $a \ggg b = a/2^b$

**Example:**

```

class RightShift
{
    public static void main(String[] args)
    {
        int number1 = 8;
        int number2 = -8;
        // 2-bit signed right shift
        System.out.println(number1 >> 2); // prints 2
        System.out.println(number2 >> 2); // prints -2
    }
}

```

**Output:**

```

2
-2

```

## Examples on Bit manipulation:

### Using Bitwise AND : *Check if a number is even*

This one tests your knowledge of how AND works and how even/odd numbers differ in binary. You can simply use:

```
(x & 1) == 0
0110 (6)
0001 (1) &
0000 TRUE
```

This solution relies on two things:

2 equates to 0001

The rightmost number for all odd numbers greater than 2 is 1

Any time the final bit evaluates to 1, you know that it matched and is, therefore, an odd number. If it instead evaluates to 0, you know that no numbers matched and therefore it's even.

public class Test

```
{
    public static void main(String[] args)
    {
        int x=2 ;
        if((x & 1) == 0)
            System.out.println("true");
        else
            System.out.println("false");
    }
}
```

### Output:

True

**Using X-OR: Java program for Convert characters to uppercase or lowercase**

This trick tests your knowledge of uppercase and lowercase characters in binary. You can convert any character, ch, to the opposite case using  $ch \wedge 32$ .

This is because the binary representation of lowercase and uppercase letters is nearly identical, with only 1 bit of difference.

Using the XOR operation lets us toggle that single bit and swap it to the opposite value, therefore making a lowercase character uppercase or vice versa.

**Program:**

```
public class Test
{
    static int x=32;
    //Toggle Case = swaps CAPS to lower case and lower case to CAPS
    static String toggleCase(char[] a)
    {
        for (int i=0; i<a.length; i++)
        {
            //Bitwise XOR with 32
            a[i]^=32;
        }
        return new String(a);
    }
    public static void main(String[] args)
    {
        String str = "CheRrY";
        System.out.print("Toggle case: ");
        str = toggleCase(str.toCharArray());
        System.out.println(str);

        System.out.print("Original string: ");
        str = toggleCase(str.toCharArray());
        System.out.println(str);
    }
}
```

**Output:**

Toggle case: cHErRy

Original string: CheRrY

**Using Bitwise AND: *Java program to count the number of bits set to 1 (set bits) of a given integer***

In this approach, we count only the set bits. So,

If a number has 2 set bits, then the while loop runs two times.

If a number has 4 set bits, then the while loop runs four times.

- ✓ Our while loop iterates until  $n = 0$ , dividing by 2 each time via the AND operator.
- ✓ On pass 1, 125 becomes 62, and count increases by 1.
- ✓ On the second pass, 62 becomes 31, and the count increases to 2.
- ✓ This continues until  $n$  becomes 0 and the count is then returned.

**Program:**

```
class CountSetBit
{
    private static int helper(int n)
    {
        int count = 0;
        while (n > 0)
        {
            n &= (n - 1);
            count++;
        }
        return count;
    }

    public static void main(String[] args)
    {
        int number = 125;
        System.out.println("SetBit Count is : " + helper(number));
    }
}
```

**Output:**

SetBit Count is : 6

## Using Bitwise OR

**Number of Flips:** *Program that takes 3 integers and uses the lowest number of flips to make the sum of the first two numbers equal to the third. The program will return the number of flips required.*

**A flip is changing one single bit to the opposite value i.e., 1 --> 0 or 0 --> 1.**

**Input:** a = 2, b = 6, c = 5

**Output:** 3

### Explanation:

First, we initialize ans to 0. Then we loop through from a range of 0 - 31.

We initialize bitA, bitB, and bitC to equal our right shift formula ANDed with 1: (a) & 1

Then, we check if bitA | bitB equals bitC. If yes, we move on to check if bitC = 0.

From there, if bitA = 1 and bitB = 1 then we increase ans by 2. Otherwise, we increase ans by 1.

Finally, we return ans, which has increased by one on every operation.

### Program:

class MinFlips

```
{
    private static int helper(int a, int b, int c)
    {
        int ans = 0;
        for (int i = 0; i < 32; i++)
        {
            int bitC = ( (c >> i) & 1 );
            int bitA = ( (a >> i) & 1 );
            int bitB = ( (b >> i) & 1 );

            if ( (bitA | bitB) != bitC )
            {
                ans += ( bitC == 0 ) ? ( bitA == 1 && bitB == 1 ) ? 2 : 1 : 1;
            }
        }
        return ans;
    }
    public static void main(String[] args)
    {
        int a = 2;
        int b = 6;
        int c = 5;
        System.out.println("Min Flips required to make two numbers equal to third is : " +
                           helper(a, b, c));
    }
}
```

**Output:** Min Flips required to make two numbers equal to third is : 3

**Using Bitwise XOR: *Single Number-*** Find the element in an array that is not repeated.

Input: nums = { 4, 1, 2, 9, 1, 4, 2 }

Output: 9

This solution relies on the following logic:

If we take XOR of zero and some bit, it will return that bit:  $a \wedge 0 = a$

If we take XOR of two same bits, it will return 0:  $a \wedge a = 0$

For n numbers, the below math can be applied:  $a \wedge b \wedge a = (a \wedge a) \wedge b = 0 \wedge b = b$

For example,

$$1 \wedge 5 \wedge 1 = (1 \wedge 1) \wedge 5 = 0 \wedge 5 = 5$$

Therefore, we can XOR all bits together to find the unique number.

**Program:**

```
class SingleNumber
{
    private static int singleNumber(int[] nums)
    {
        int xor = 0;
        for (int num : nums)
        {
            xor ^= num;
        }
        return xor;
    }
    public static void main(String[] args)
    {
        int[] nums = {4, 1, 2, 9, 1, 4, 2};
        System.out.println("Element appearing one time is " + singleNumber(nums));
    }
}
```

**Output:**

Element appearing one time is 9



**Using Bitwise Left Shift:** *Get First Set Bit* Given an integer, find the position of the first set-bit (1) from the right.

Input: n = 18

18 in binary = 0b10010

Output: 2

**Procedure:**

The logic of this solution relies on a combination of left shifting and the AND operation.

Essentially, we first check if the rightmost significant bit is the set bit using bit & 1. If not, we keep shifting left and checking until we find the bit that makes our AND operation yield 1.

The number of shifts is tracked by our pointer, k. Once we do find the set bit, we return k as our answer.

**Program**

```
class FirstSetBitPosition
{
    private static int helper(int n)
    {
        if (n == 0)
        {
            return 0;
        }
        int k = 1;
        while (true)
        {
            if ((n & (1 << (k - 1))) == 0)
            {
                k++;
            }
            else
            {
                return k;
            }
        }
    }
    public static void main(String[] args)
    {
        System.out.println("First setbit position for number: 18 is -> " + helper(18));
        System.out.println("First setbit position for number: 5 is -> " + helper(5));
        System.out.println("First setbit position for number: 32 is -> " + helper(32));
    }
}
```

**Output:**

First setbit position for number: 18 is -> 2

First setbit position for number: 5 is -> 1

First setbit position for number: 32 is -> 6

**3. Using XOR: Swapping two numbers:**

```
import java.util.Scanner;
public class Test8
{
    public static void main(String args[])
    {
        int a, b;
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first number: ");
        a = scanner.nextInt();
        System.out.print("Enter the second number: ");
        b = scanner.nextInt();
        System.out.println("Before swapping:");
        System.out.println("a = " +a +", b = " +b);
        a = a ^ b;
        b = a ^ b;
        a = a ^ b;
        System.out.println("After swapping:");
        System.out.print("a = " +a +", b = " +b);
    }
}
```

input=

Enter the first number: 7

Enter the second number: 9

output=

Before swapping:

7 9

After swapping:

9 7

**Complexity Analysis**

- **Time Complexity: O(1)**
- **Space Complexity: O(1)**

## Applications

- ✓ Counting Bits
- ✓ Palindrome Permutation
- ✓ Remove All Ones with Row and Column Flips
- ✓ Encode Number.

# 1. Counting Bits

Given an integer  $n$ , return *an array ans of length  $n + 1$  such that for each  $i$  ( $0 \leq i \leq n$ ),  $\text{ans}[i]$  is the number of 1's in the binary representation of  $i$ .*

**Input:**  $n = 2$

Output: [0,1,1] Explanation:

0 --> 0

1 --> 1

2 --> 10

**Input:**  $n = 5$

**Output:** [0,1,1,2,1,2]

**Explanation:**

0 --> 0

1 --> 1

2 --> 10

3 --> 11

4 --> 100

5 --> 101

**Procedure:**

1. Since for decimal number 0 , no. of '1' bits in its binary representation = 0
2. For each value from 1 loop to iterate each bit of the number and count no. of '1' bits till the number becomes 0 i.e.it has no '1' bits remaining in its binary representation.
3. 0 will be added to count if last bit is 0 , and 1 will be added to count variable if last bit is 1, thereby increasing count to 1.
4. Drop the last bit and repeat the process of counting the 1 bit.

**Write a Java Program to count the number of 1 in bitrepresentation from 0 to a given value**

**CountingBits.java**

```
import java.util.*;
class CountingBits
{
    public static int[] countBits(int n)
    {
        int r[]= new int[n+1];
        r[0]=0;
        for(int i=1; i<=n; i++)
        {
            int ct=0; int x=i;
            while(x>0)
            {
                ct+= (x & 1);
                x= x >> 1;
            }
            r[i]= ct;
        }
        return r;
    }
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        int n=s.nextInt();
        int r[]=new int[n+1];
        r=countBits(n);
        for(int i=0;i<=n;i++)
            System.out.println(" "+r[i]);
    }
}
```

## 2. Palindrome Permutation

Given a string, determine if a permutation of the string could form a palindrome.

**Example 1:**

Input: "code"

Output: false

**Example 2:**

Input: "aab"

Output: true

**Example 3:**

Input: "carerac"

Output: true

If a string with an even length is a palindrome, every character in the string must always occur even number of times. If the string with an odd length is a palindrome, every character except one of the characters must occur even number of times. Thus, in case of a palindrome, the number of characters with odd number of occurrences can't exceed 1 (1 in case of odd length and 0 in case of even length).

**Procedure:**

5. Retrieve character from the string
6. Subtract ascii value of a from the character and perform the output value number of left shifts on 1 and perform exclusive or operation with a bitmask value whose initial value is 0.
7. Repeat step 1 till the end of the string.
8. If the same character is seen next time bitmask value will be zero.
9. If there are even number of characters in the string and if it can be a palindrome the bitmask value will be zero
10. If there are odd number of characters and if it can be a palindrome the bitmask value with a bitwise & operation on bitmask-1 will be zero
11. If step 5 or step 6 is satisfied return true otherwise return false.

**Write a Java Program to determine if a permutation of a string is a palindrome or not**

**PermutePalindrome.java**

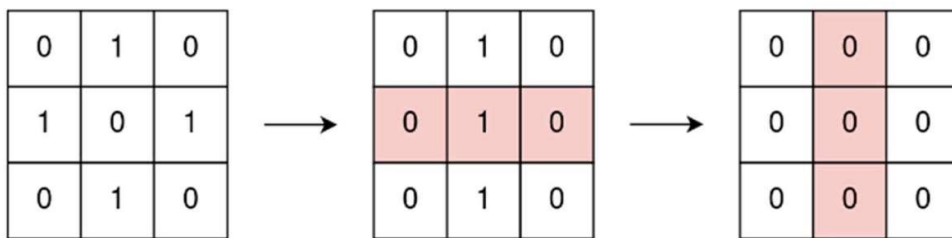
```
import java.util.*;
class PermutePalindrome
{
    static boolean canPermutePalindrome(String s)
    {
        int bitmask = 0;
        for(int i=0;i<s.length();i++)
        {
            char ch=s.charAt(i);
            bitmask = bitmask ^ (1 << (ch - 'a'));
        }
        return (bitmask == 0 || (bitmask & (bitmask-1)) == 0 );
    }
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("Enter the string");
        String ps=s.next();
        System.out.println(canPermutePalindrome(ps));
    }
}
```

### 3. Remove All Ones with Row and Column Flips

We are given an  $m \times n$  binary matrix grid.

In one operation, you can choose **any** row or column and flip each value in that row or column (i.e., changing all 0's to 1's, and all 1's to 0's).

Return true if it is possible to remove all 1's from the grid using any number of operations or false otherwise.



Input: grid = [ [0,1,0], [1,0,1], [0,1,0] ]

Output: true

Explanation: One possible way to remove all 1's from grid is to:

- Flip the middle row
- Flip the middle column

#### Procedure:

1. Compare first row with every row of the matrix.
2. If every row values are equal to first row or reversed row then it is possible to remove all 1's in the rows by row and column flips



**Java Program to determine whether it is possible to remove all 1's from a binary matrix with row and column flips.**

```
import java.util.*;
class RowColFlips
{
    public boolean removeOnes(int[][] grid)
    {
        // if the matrix only has one row then it is always true
        if (grid.length <= 1) return true;

        // start with the second row compare all values to the first row
        for (int i=1; i<grid.length; i++)
        {
            if (!equalsOrComplement(grid[0], grid[i]))
                return false;
        }
        return true;
    }

    private boolean equalsOrComplement(int[] row1, int[] otherrow)
    {
        boolean equals = true, complement = true;
        /*Every row should be equal to first row or complement to first row
        or else there will always be an extra value*/

        for (int i=0; i<row1.length; i++)
        {
            equals = equals && (row1[i]==otherrow[i]);
            complement = complement && (row1[i]!=otherrow[i]);
        }
        return equals || complement;
    }

    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        int m=sc.nextInt();
        int n=sc.nextInt();
        int grid[][]=new int[m][n];
        for(int i=0;i<m;i++)
            for(int j=0;j<n;j++)
                grid[i][j]=sc.nextInt();

        System.out.println(new RowColFlips().removeOnes(grid));
    }
}
```

## 4. Encode Number

Given a non-negative integer `num`, Return its encoding string.

The encoding is done by converting the integer to a string using a secret function that you should deduce from the following table:

N	f(n)
0	""
1	"0"
2	"1"
3	"00"
4	"01"
5	"10"
6	"11"
7	"000"

**Example 1:**

**Input:** `num = 23`

**Output:** "1000"

**Example 2:**

**Input:** `num = 107`

**Output:** "101100"

If `n` is 0, then `f(n)` is "". If  $1 \leq n < 3$ , then `f(n)` is a binary string with length 1. If  $3 \leq n < 7$ , then `f(n)` is a binary string with length 2. If  $7 \leq n < 15$ , then `f(n)` is a binary string with length 3.

**Procedure**

1. For the given number `num`, obtain the number of bits `bits` for number `num + 1`, where the number of bits means the number of bits remaining after removing leading zeros.
2. Calculate difference as the difference between `num + 1` and  $2^{\text{bits}}$ , and return the binary representation of difference.

**Write a java program to display the encoded form of a given string.**

**EncodeNumber.java**

```
import java.util.*;

class EncodeNumber
{
    public String encode(int num)
    {
        StringBuilder sb = new StringBuilder();
        double cur = Math.log(num + 1) / Math.log(2);
        int bits = (int)(Math.floor(cur));
        int offset = num - (int)(Math.pow(2, bits)) + 1;
        while (offset > 0)
        {
            sb.append(offset % 2);
            offset /= 2;
        }
        int l = sb.length();
        for (int i = l; i < bits; i++)
        {
            sb.append("0");
        }
        return sb.reverse().toString();
    }

    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        int s=sc.nextInt();
        System.out.println(new EncodeNumber().encode(s));
    }
}
```