SYLLABUS

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

Applications:

1. Introduction
2. Counting Bits
3. Palindrome Permutation
4. Remove All Ones with Row and Column Flips
5. Encode Number.

Introduction

In programming, an n-bit integer is internally stored as a binary number that consists of n bits. For example, the Java type int is a 32-bit type, which means that every int number consists of 32 bits. For example, the bit representation of the int number 43 is 00000000000000000000000000101011.

The bits in the representation are indexed from right to left. To convert a bit representation $b_k \ldots b_2 b_1 b_0$ into a number, the formula

$$b_k 2^k + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0 .\text{can}$$

be used. For example,

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

- The bit representation of a number is either *signed* or *unsigned*.
- Usually a signed representation is used, which means that both negative and – positive numbers can be represented.
- A signed variable of n bits can contain any integer between 2^{n-1} and2^{n-1} -1. For example, the int type in Java is a signed type, so an int variable can contain any integer between -2^{31} and 2^{31} -1.
- The first bit in a signed representation is the sign of the number (0 for nonnegative numbers and 1 for negative numbers), and the remaining n 1 bits contain the magnitude of the number.
- *Two's complement* is used, which means that the opposite number of a number is calculated by first inverting all the bits in the number and then increasing the number by one. For example, the bit representation of the number -43 is 11111111111111111111111111010101.
- There is a connection between the representations: a signed number $-x$ equals an unsigned number $2^n - x$. For example, the following code shows that the signed number $x = -43$ equals the unsigned number $y = 2^{32} - 43$.
- If a number is larger than the upper bound of the bit representation, the number will overflow.
- In a signed representation, the next number after $2^{n-1} - 1$ is -2^{n-1}.
- Initially, the value of x is $2^{31} - 1$. This is the largest value that can be stored in an intvariable, so the next number after $2^{31} - 1$ is -2^{31}.

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 to finding the green portion of an RGB value:

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

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

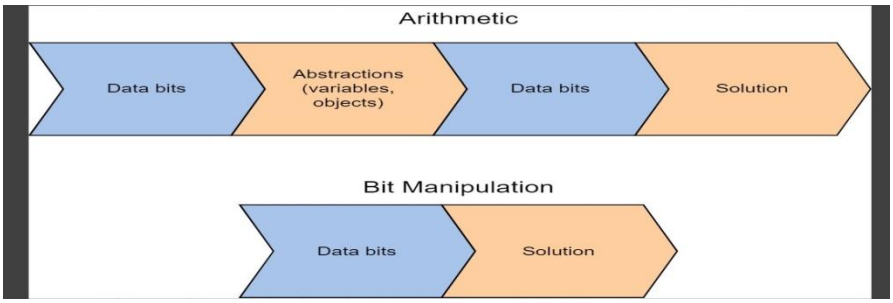
While both do the same thing, the second option is considerably faster, as it works directly within memory rather than through a level of abstraction.

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

1. 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

2. 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

3. 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

- ~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 mid point.

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
```

4. 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 ^ 1 => yields to 0.
- 0 ^ 0 => yields to 0.
- 1 ^ 0 => yields to 1.
- 0 ^ 1 => 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

Bit Shift Operators:

5. 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.

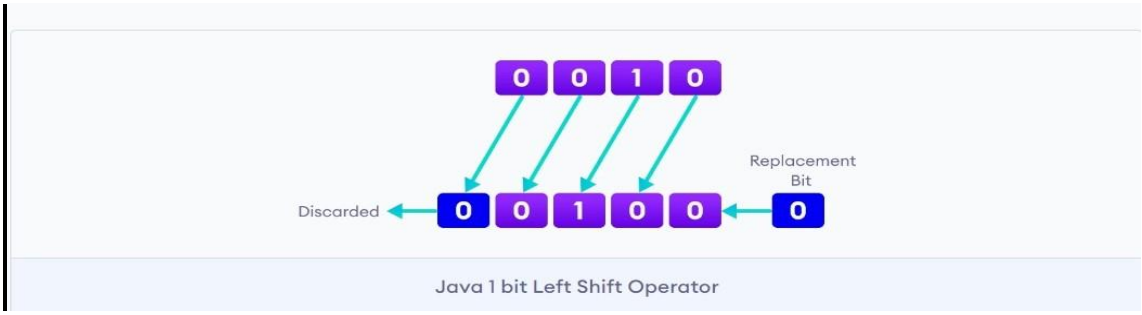
These operators can be applied to integral types such as int, long, short, byte, or char.

There are three types of shift:

- **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 00000110

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

6<<1=00000000 00000000 00000000 00001100

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.**

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

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

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 >>1 = 1101 1010.

Formula: $x >> y = x/2^y$

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

10110101>>>1= 01011010

Formula: $a >> b = a/2^b$

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 manipulaton:

1. 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
= 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

2. Using X-OR: Write a java progam 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 ^= 32.
- This is because the binary representation of lowercase and uppercase letters are 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
```

3. Using Bitwise AND: Write a Java program to count the number of bits set to 1 (set bits) of an 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
```

4. Using Bitwise OR: Number of Flips: Write a 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 ie. 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

5. **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 ^ 0 = a
- If we take XOR of two same bits, it will return 0: a ^ a = 0
- For n numbers, the below math can be applied: a ^ b ^ a = (a ^ a) ^ b = 0 ^ b = b

For example,

1 ^ 5 ^ 1 =

$(1^1)5=$

$0^5=5$

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

6. Using Bitwise Left Shift: Get First Set BitGiven 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 bet 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
```

7. 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)**

1. Counting Bits

Given an integer n, return *an array* ans *of length* n + 1 *such that for each* i (0 <= i <=n), 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 brute-force approach O(nlogn):
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 variableif last bit is 1, there by increasing count to 1.
4. Drop the last bit and repeat the process of counting the 1 bits.

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  count=0;
                int x=i;
                while(x>0)
                {
```

```
                    count += (x & 1);
                    x= x >> 1;
                }
                r[i]= count;
        }
  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]);
}
}
```

Time Complexity:

Your Java implementation correctly counts the number of 1s in the binary representation of numbers from 0 to n, but it uses a **brute-force** approach with O(log n) complexity for each number, making the overall complexity **O(n log n)**.

Optimized Approach:

You can improve the efficiency to **O(n)** using dynamic programming, just like in the C++ version. Instead of checking each bit one by one, we can use the relation:

$$countBits(i)=countBits(i/2)+(i\%2)$$

Procedure:

1. **Use a DP array (dp[])**:
   - Let dp[i] store the count of 1s in the binary representation of i.
   - Initialize dp[0] = 0 since 0 has no 1s.
2. **Use a Recurrence Relation:**
   - Every number i can be expressed as **i = i/2 + (i % 2)**:
   - i / 2 is the right-shifted version of i (removing the last bit).
   - (i % 2) is 1 if i is odd, 0 if even.
   - This leads to the relation: dp[i]=dp[i/2]+(i%2)
   - Since i/2 is already computed, we can fetch its result in **O(1) time**.
3. **Iterate from 1 to n and compute dp[i] using the formula.**
   - This ensures an **O(n) time complexity**.

Write a Java Program to count the number of 1 in bitrepresentation from 0 to a given value with O(n) Time Complexity:

```
import java.util.*;
class CountingBits
 {
   public static int[] countBits(int n)
    {
      int[] dp = new int[n + 1];
      dp[0] = 0;

      for (int i = 1; i <= n; i++)
      {
         dp[i] = dp[i / 2] + (i % 2);
      }
      return dp;
   }

   public static void main(String[] args)
  {
       Scanner s=new Scanner(System.in);
        int n=s.nextInt();
      int[] result = countBits(n);

      for (int num : result)
       {
         System.out.print(num + " ");
      }
   }
}
```

Example-1:
Input:
5
Output:
0 1 1 2 1 2

Example-2:
Input:
15
Output:
0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4

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

Approach:

A string can be rearranged into a palindrome if:

1. **For even-length strings** → All characters must appear an **even number** of times.
2. **For odd-length strings** → Only **one character** can appear an **odd number** of times, while all others must appear an **even number** of times.

Procedure:

Step 1: Initialize a Bitmask

- We use an integer bitmask (bitmask) to track character occurrences using bitwise XOR (^) operations.
- Since there are 26 lowercase letters, we use the first 26 bits of an integer.

Step 2: Process Each Character

- For each character ch in the string:
- Compute its bit position using 1 << (ch - 'a').
- Toggle the corresponding bit using bitmask ^= (1 << (ch - 'a')).
- If a character appears twice, its bit resets to 0 (even count).
- If a character appears once, its bit remains 1 (odd count).

Step 3: Check Palindrome Condition

- After processing the string, the bitmask represents the odd-count characters.
- A valid palindrome permutation must have at most one bit set in bitmask.
  - We check this condition using:
  - return (bitmask == 0 || (bitmask & (bitmask - 1)) == 0);
  - bitmask == 0 → All characters have even counts (valid palindrome).
  - (bitmask & (bitmask - 1)) == 0 → At most one character has an odd count (valid palindrome).

Step 4: Read Input and Call the Function

Write a Java Program to determine if a permutation of a string isa 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 ^= (1 << (ch - 'a'));
      }
     return (bitmask == 0 || (bitmask & (bitmask-1)) == 0 );
   }
   public static void main(String[] args)
   {
      Scanner s=new Scanner(System.in);

      String ps=s.next();
      System.out.println(canPermutePalindrome(ps));
   }
}
```

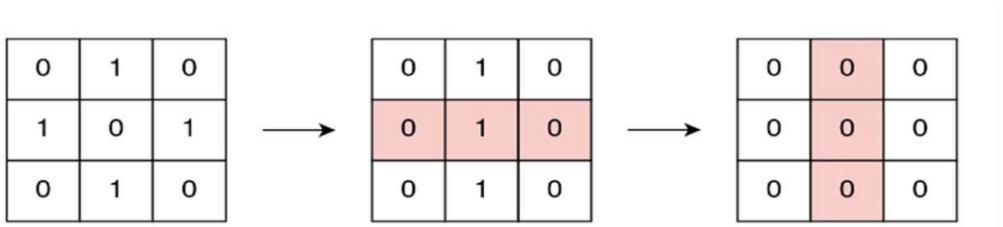
3. Remove All Ones with Row and Column Flips

We are given an m x n binary matrix grid.

In one operation, you can choose **any** row or column and flip each value in that rowor 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 ofoperations or false otherwise.

Example 1:



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

Example 2:

1	1	0
0	0	0
0	0	0

Input: grid = [[1,1,0],[0,0,0],[0,0,0]]
Output: false
Explanation: It is impossible to remove all 1's from grid.

Example 3:
Input: grid = [[0]]
Output: true
Explanation: There are no 1's in grid.

Approach:

Step 1: Process the First Row

- We consider the first row as a reference.
- Every other row must be either identical to it or its complement.

Step 2: Compare Each Row with the First Row

- For every row in the matrix:
  - If it matches the first row or its bitwise complement, it can be transformed to all 0s.
  - Otherwise, it is impossible to remove all 1s.

Step 3: Return the Result

- If all rows can be converted to either the first row or its complement, return true.
- Otherwise, return false.

Conditions checked:

- If `sum == 0`: The row consists entirely of `0`s □ (Valid)
- If `sum == col`: The row consists entirely of `1`s □ (Valid)
- If `sum` is anything **in between**, the row contains a **mix of `0`s and `1`s** □ (Invalid case → return `false`)

Step	**Time Complexity**
Construct Complement Row	O(n)
Compare Each Row	O(m×n)
Overall Complexity	**O(m×n)**

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

```
import java.util.*;
class RemoveAllOneswithFlips
{
   public boolean removeOnes(int[][] grid)
 {
       int row = grid.length; // get dimensions of grid
       int col = grid[0].length;

       for (int c= 0; c < col; c++)
     {
          // flip columns so that first row only has 0's
         if (grid[0][c] == 1)
```

```
            {
              for (int r = 0; r < row; r++)
                {
                 // flips a column
                 grid[r][c] ^= 1;
                }
            }
        }
        for (int r = 1; r < row; r++)
         {
          // checks if each row has all 0's or all 1's
            int sum = 0;
            for (int c = 0; c < col; c++)
             {
                sum += grid[r][c];
            }
            /*if (sum == 0 || sum == col)
            {
                continue;
            }
            ans = false;*/
            if (sum!=0 && sum!=col)
            {
                return false;
            }
         }
        return true;
    }
  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 RemoveAllOneswithFlips().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 <= n < 3, then f(n) is a binary string with length 1.
If 3 <= n < 7, then f(n) is a binary string with length 2.
If 7 <= n < 15, then `f(n) is a binary string with length 3.

Procedure

1. The problem requires us to:
   - Convert `num + 1` to **binary**.
   - Remove the **leading `1`** from the binary representation.
   - Return the remaining binary string.
2. Instead of using `Integer.toBinaryString()`, we can **manually extract bits** using **bit manipulation**.

Approach:

1. Increment num
   Since the encoding rule is based on num + 1, we start by incrementing num:
2. Extract Bits Using Bit Manipulation
   We iterate while num > 1 and extract the **least significant bit (LSB)** using:
   - num & 1 extracts the last bit of the binary number.
   - Then, we **right shift (>>)** to remove the extracted bit:
3. Stop at 1
   We **stop when num becomes 1**, because this is the **leading 1** in num + 1, which we must ignore.
4. Reverse the Result
   - Since we extract bits from right to left, we **reverse the string** at the end:

Example 1: num = 23

1. num + 1 = 24
2. 24 in binary: "11000"
3. Removing the leading 1: "1000"
4. Output: "1000"

Step-by-step Execution

Step	num Value	num & 1 (Extracted Bit)	Appended to String
1	24 (11000)	0	"0"
2	12 (1100)	0	"00"
3	6 (110)	0	"000"
4	3 (11)	1	"0001"
Stop	1 (1) - Stop	-	Reverse → "1000"

Write a java program to display the encoded form of a givenstring.

EncodeNumber.java

```
import java.util.*;
class Solution
{
   public String encode(int num)
   {
      StringBuilder encoded = new StringBuilder();

      // Process each bit of (num + 1) from right to left

      num += 1; // Work with num + 1
```

```
        while (num > 1) // Stop when num becomes 1 (leading '1' is ignored)

         {
           encoded.append(num & 1); // Extract the last bit
            num >>= 1;  // Right shift to process next bit
         }

        return encoded.reverse().toString(); // Reverse to get correct order
    }

   public static void main(String[] args)
    {
          Scanner s=new Scanner(System.in);
          System.out.println("Enter a number");
          int n=s.nextInt();
          System.out.println(encode(n));
    }
}
```