Naresh M\_AI&DS\_DSA\_Practice-Day-3

**1.anagram**

Given two strings s1 and s2 consisting of lowercase characters. The task isto check whether two given strings are an anagram of each other or not. An anagram of a string is another string that contains the same characters, only the order of characters can be different. For example, act and tac are an anagram of each other. Strings s1 and s2 can only contain lowercase alphabets.

Note: You can assume both the strings s1 & s2 are non-empty.

Examples :

Input: s1 = "geeks", s2 = "kseeg"

Output: true

Explanation: Both the string have same characters with same frequency. So, they are anagrams.

Input: s1 = "allergy", s2 = "allergic"

Output: false

Explanation: Characters in both the strings are not same, so they are not anagrams.

**Program:**

import java.util.\*;

public class Solution {

    public static boolean areAnagrams(String s1, String s2) {

        if (s1.length() != s2.length()) {

            return false;

        }

        Map<Character, Integer> map = new HashMap<>();

        for (int i = 0; i < s1.length(); i++) {

            map.put(s1.charAt(i), map.getOrDefault(s1.charAt(i), 0) + 1);

        }

        for (int i = 0; i < s2.length(); i++) {

            char ch = s2.charAt(i);

            if (!map.containsKey(ch) || map.get(ch) == 0) {

                return false;

            }

            map.put(ch, map.get(ch) - 1);

        }

        return true;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the first string:");

        String s1 = sc.nextLine();

        System.out.println("Enter the second string:");

        String s2 = sc.nextLine();

        if (areAnagrams(s1, s2)) {

            System.out.println("The strings are anagrams.");

        } else {

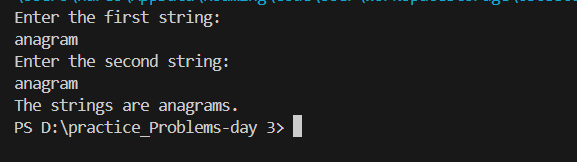
            System.out.println("The strings are not anagrams.");

        }

        sc.close();

    }

}

**Time Complexity: O(n)**

**2.Row with max 1s**

You are given a 2D array consistingof only **1's**and**0's**, where each row is sorted in non-decreasing order. You need to find and return the index of the first row that has the most number of 1s. If no such row exists, return **-1**.  
**Note:**0-based indexing is followed.

**Examples:**

**Input:** arr[][] = [[0, 1, 1, 1],  
 [0, 0, 1, 1],  
 [1, 1, 1, 1],  
 [0, 0, 0, 0]]

**Output:** 2

**Explanation:** Row 2 contains **4** 1's.

**Input:** arr[][] = [[0, 0],   
 [1, 1]]

**Output:** 1

**Explanation:** Row 1 contains **2** 1's.

**Note :**Here n,m refers to the number of rows and columns respectively.

**Constraints:**  
1 ≤ number of rows, number of columns ≤ 1030 ≤ arr[i][j] ≤ 1

**Program:**

import java.util.\*;

public class rowWithMax1s  {

    public int rowWithMax1s(int arr[][]) {

        int maxRowIndex = -1;

        int maxCount = 0;

        for (int i = 0; i < arr.length; i++) {

            int count = countOnes(arr[i]);

            if (count > maxCount) {

                maxCount = count;

                maxRowIndex = i;

            }

        }

        return maxRowIndex;

    }

    public static int countOnes(int[] row) {

        int cnt = 0;

        for (int i = 0; i < row.length; i++) {

            if (row[i] == 1) {

                cnt++;

            }

        }

        return cnt;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of rows:");

        int rows = sc.nextInt();

        System.out.println("Enter the number of columns:");

        int cols = sc.nextInt();

        int[][] arr = new int[rows][cols];

        System.out.println("Enter the elements of the matrix (0s and 1s only):");

        for (int i = 0; i < rows; i++) {

            for (int j = 0; j < cols; j++) {

                arr[i][j] = sc.nextInt();

            }

        }

        rowWithMax1s sol = new rowWithMax1s();

        int result = sol.rowWithMax1s(arr);

        if (result != -1) {

            System.out.println("The row with the maximum number of 1s is: " + result);

        } else {

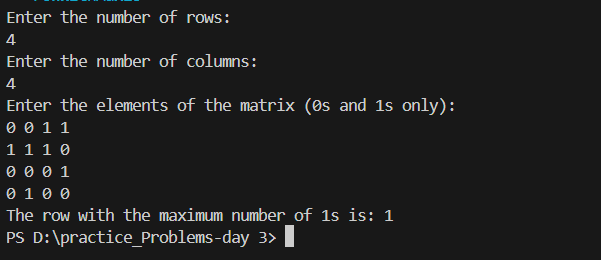
            System.out.println("No row contains 1s.");

        }

        sc.close();

    }

}



**Time Complexity: O(mxn)**

**3.Longest consecutive subsequence**

Given an array **arr** of non-negative integers. Find the **length** of the longest sub-sequence such that elements in the subsequence are consecutive integers, the**consecutive numbers** can be in **any order.**

**Examples:**

**Input:** arr[] = [2, 6, 1, 9, 4, 5, 3]

**Output:** 6

**Explanation:** The consecutive numbers here are 1, 2, 3, 4, 5, 6. These 6 numbers form the longest consecutive subsquence.

**Input:** arr[] = [1, 9, 3, 10, 4, 20, 2]

**Output:** 4

**Explanation:** 1, 2, 3, 4 is the longest consecutive subsequence.

**Input**: arr[] = [15, 13, 12, 14, 11, 10, 9]

**Output**: 7

**Explanation**: The longest consecutive subsequence is 9, 10, 11, 12, 13, 14, 15, which has a length of 7.

**Constraints:**  
1 <= arr.size() <= 105  
0 <= arr[i] <= 105

**Program:**

import java.util.\*;

public class longestConseqSubsequence {

    // Function to return length of longest subsequence of consecutive integers.

    public int findLongestConseqSubseq(int[] arr) {

        int l = 0;

        Set<Integer> numset = new HashSet<>();

        for (int num : arr) {

            numset.add(num);

        }

        for (int num : numset) {

            if (!numset.contains(num - 1)) { // Start of a new sequence

                int len = 1;

                while (numset.contains(num + len)) {

                    len++;

                }

                l = Math.max(l, len);

            }

        }

        return l;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of elements in the array:");

        int n = sc.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements of the array:");

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        longestConseqSubsequence sol = new longestConseqSubsequence();

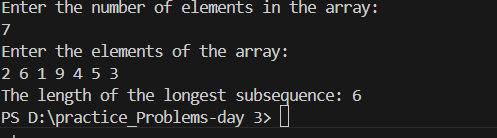
        int result = sol.findLongestConseqSubseq(arr);

        System.out.println("The length of the longest subsequence: " + result);

        sc.close();

    }

}



**Time Complexity: O(n)**

**4.Longest palindrome in a string**

Given a string s, your task is to find the longest palindromic substring within s. A substring is a contiguous sequence of characters within a string, defined as s[i...j] where 0 ≤ i ≤ j < len(s).

A palindrome is a string that reads the same forward and backward. More formally, s is a palindrome if reverse(s) == s.

Note: If there are multiple palindromes with the same length, return the first occurrence of the longest palindromic substring from left to right.

Examples :

Input: s = "aaaabbaa"

Output: "aabbaa"

Explanation: The longest palindromic substring is "aabbaa".

Input: s = "abc"

Output: "a"

Explanation: "a", "b", and "c" are all palindromes of the same length, but "a" appears first.

**Program:**

import java.util.Scanner;

public class  longestPalindromeString{

    // Static method to find the longest palindromic substring

    static String longestPalindrome(String s) {

        String res = "";

        int len = 0;

        for (int i = 0; i < s.length(); i++) {

            int l = i, r = i;

            while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {

                if (r - l + 1 > len) {

                    res = s.substring(l, r + 1);

                    len = r - l + 1;

                }

                l--;

                r++;

            }

            l = i;

            r = i + 1;

            while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {

                if (r - l + 1 > len) {

                    res = s.substring(l, r + 1);

                    len = r - l + 1;

                }

                l--;

                r++;

            }

        }

        return res;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the string:");

        String s = sc.nextLine();

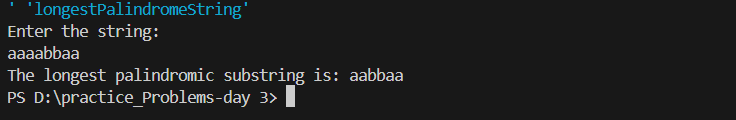
        String result = longestPalindrome(s);

        System.out.println("The longest palindromic substring is: " + result);

        sc.close();

    }

}



**Time Complexity: O(n2)**

**5. Rat in the maze:**

Consider a rat placed at **(0, 0)** in a square matrix **mat**of order **n\* n**. It has to reach the destination at **(n - 1, n - 1)**. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are **'U'(up)**, **'D'(down)**, **'L' (left)**, **'R' (right)**. Value 0 at a cell in the matrix represents that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it.  
**Note**: In a path, no cell can be visited more than one time. If the source cell is 0, the rat cannot move to any other cell. In case of no path, return an empty list. The driver will output **"-1"** automatically.

**Examples:**

**Input**: mat[][] = [[1, 0, 0, 0],

[1, 1, 0, 1],

[1, 1, 0, 0],

[0, 1, 1, 1]]

**Output:** DDRDRR DRDDRR

**Explanation**: The rat can reach the destination at (3, 3) from (0, 0) by two paths - DRDDRR and DDRDRR, when printed in sorted order we get DDRDRR DRDDRR.

**Input**: mat[][] = [[1, 0],

[1, 0]]

**Output:** -1

**Explanation**: No path exists and destination cell is blocked.

**Expected Time Complexity:** O(3n^2)  
**Expected Auxiliary Space:** O(l \* x)  
Here l = length of the path, x = number of paths.

**Program:**

import java.util.\*;

public class ratInTheMaze {

    public static List<String> findPath(int[][] mat, int n) {

        List<String> result = new ArrayList<>();

        if (mat[0][0] == 0 || mat[n - 1][n - 1] == 0) {

            return result;

        }

        boolean[][] visited = new boolean[n][n];

        StringBuilder path = new StringBuilder();

        dfs(mat, 0, 0, n, visited, path, result);

        return result;

    }

    private static void dfs(int[][] mat, int i, int j, int n, boolean[][] visited, StringBuilder path, List<String> result) {

        if (i == n - 1 && j == n - 1) {

            result.add(path.toString());

            return;

        }

        visited[i][j] = true;

        String[] directions = {"D", "L", "R", "U"};

        int[] di = {1, 0, 0, -1};

        int[] dj = {0, -1, 1, 0};

        for (int d = 0; d < 4; d++) {

            int ni = i + di[d];

            int nj = j + dj[d];

            if (isSafe(mat, ni, nj, n, visited)) {

                path.append(directions[d]);

                dfs(mat, ni, nj, n, visited, path, result);

                path.deleteCharAt(path.length() - 1); // Backtrack

            }

        }

        visited[i][j] = false;

    }

    private static boolean isSafe(int[][] mat, int i, int j, int n, boolean[][] visited) {

        return i >= 0 && i < n && j >= 0 && j < n && mat[i][j] == 1 && !visited[i][j];

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the size of the matrix (n):");

        int n = sc.nextInt();

        int[][] mat = new int[n][n];

        System.out.println("Enter the elements of the matrix (0 or 1):");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                mat[i][j] = sc.nextInt();

            }

        }

        List<String> paths = findPath(mat, n);

        if (paths.isEmpty()) {

            System.out.println("-1");

        } else {

            Collections.sort(paths);

            for (String path : paths) {

                System.out.print(path + " ");

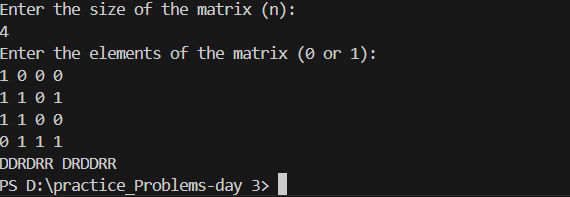
            }

        }

        sc.close();

    }

}



**Time Complexity: O(m×n)**