# **Sahana Lakshmipathy\_AI&DS\_DSA\_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.HashMap;

import java.util.Map;

class Solution {

// Function to check whether two strings are anagrams of each other.

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

// If lengths are not equal, they can't be anagrams

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

return false;

}

// Count frequency of each character in s1

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

for (char c : s1.toCharArray()) {

frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);

}

// Subtract frequency of each character in s2

for (char c : s2.toCharArray()) {

if (!frequencyMap.containsKey(c)) {

return false; // Character in s2 not in s1

}

frequencyMap.put(c, frequencyMap.get(c) - 1);

if (frequencyMap.get(c) < 0) {

return false; // More occurrences in s2 than in s1

}

}

// All counts should be zero if they are anagrams

return true;

}

}

public class Main {

public static void main(String[] args) {

String s1 = "listen";

String s2 = "silent";

if (Solution.areAnagrams(s1, s2)) {

System.out.println(s1 + " and " + s2 + " are anagrams.");

} else {

System.out.println(s1 + " and " + s2 + " are not anagrams.");

}

String s3 = "hello";

String s4 = "world";

if (Solution.areAnagrams(s3, s4)) {

System.out.println(s3 + " and " + s4 + " are anagrams.");

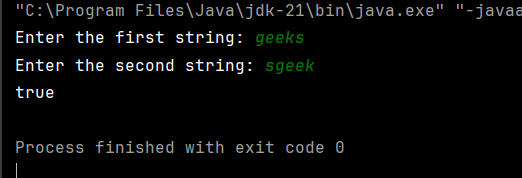
} else {

System.out.println(s3 + " and " + s4 + " are not anagrams.");

}

}

}



**Time Complexity: O(n)**

**2.Row with minimum (maximum) no of ones**

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.

**Expected Time Complexity:** O(n+m)   
**Expected Auxiliary Space:** O(1)

**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:**

class Solution {

// Function to find the row with the minimum number of 1's

int minRow(int mat[][]) {

int minRowIndex = -1; // Index of the row with minimum 1's

int minCount = Integer.MAX\_VALUE; // Initialize with a large number

// Traverse each row in the matrix

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

int countOnes = 0;

// Count the number of 1's in the row

for (int j = 0; j < mat[i].length; j++) {

if (mat[i][j] == 1) {

countOnes++;

}

}

// Update minRowIndex if we find a row with fewer 1's

if (countOnes < minCount) {

minCount = countOnes;

minRowIndex = i;

}

}

return minRowIndex;

}

}

public class Main {

public static void main(String[] args) {

int mat[][] = {

{0, 1, 0},

{1, 1, 0},

{0, 0, 0},

{1, 1, 1}

};

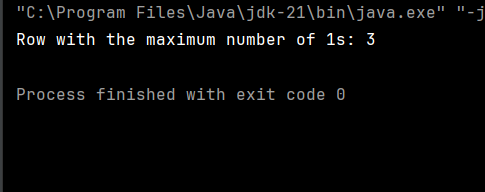
Solution solution = new Solution();

int rowIndex = solution.minRow(mat);

System.out.println("Row with the minimum number of 1's: " + rowIndex);

}

}



**Time Complexity: O(m×n)**

**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.HashSet;

public class Solution {

// Function to find the length of the longest consecutive subsequence

public int findLongestConseqSubseq(int[] arr) {

if (arr == null || arr.length == 0) {

return 0;

}

// Create a HashSet to store elements for O(1) lookup

HashSet<Integer> set = new HashSet<>();

// Add all elements to the HashSet

for (int num : arr) {

set.add(num);

}

int longestStreak = 0;

// Iterate through each number in the array

for (int num : arr) {

// Check if this is the start of a sequence

if (!set.contains(num - 1)) {

int currentNum = num;

int currentStreak = 1;

// Check for the next consecutive numbers

while (set.contains(currentNum + 1)) {

currentNum += 1;

currentStreak += 1;

}

// Update longest streak if the current streak is longer

longestStreak = Math.max(longestStreak, currentStreak);

}

}

return longestStreak;

}

public static void main(String[] args) {

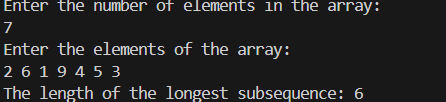
Solution solution = new Solution();

int[] arr = {100, 4, 200, 1, 3, 2};

System.out.println("Longest Consecutive Subsequence Length: " + solution.findLongestConseqSubseq(arr));

}

}



**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 scanner = new Scanner(System.in);

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

        String str = scanner.nextLine();

        String result = longestPalindrome(str);

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

        scanner.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);

            }

        }

        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 scanner = new Scanner(System.in);

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

        int n = scanner.nextInt();

        int[][] mat = new int[n][n];

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

        for (int i = 0; i < n; i++) {

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

                mat[i][j] = scanner.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 + " ");

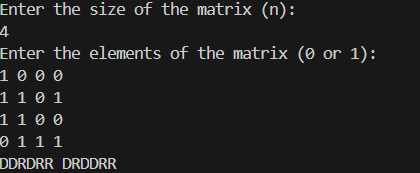
            }

        }

        scanner.close();

    }

}



**Time complexity: O(n2)+O(k⋅log(k))**