Gopika shree M

22AD034

AI&DS

1) Given two strings **s1**and **s2**consisting of lowercase characters. The task is to 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.

**Input:** s1 = "g", s2 = "g"

**Output:** true

**Explanation:** Character in both the strings are same, so they are anagrams.

**Constraints:**  
1 ≤ s1.size(), s2.size() ≤ 105

import java.util.Map;

import java.util.HashMap;

public class Anagram {

public void isAnagram(String s1, String s2) {

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

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

set.put(i, set.getOrDefault(i, 0)+1);

}

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

if (!set.containsKey(i)) {

System.***out***.println(false);

return;

}

else if (set.containsKey(i)) {

set.put(i, set.getOrDefault(i, 0)-1);

}

if (set.get(i) == 0) {

set.remove(i);

}

}

System.***out***.println(set.isEmpty());

}

public static void main(String[] args) {

Anagram obj = new Anagram();

obj.isAnagram("listen", "silent");

obj.isAnagram("hello", "bello");

}

}

2) 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

public class Main {

public void rowWithMax1(int[][] arr) {

int n = arr.length;

int m = arr.length;

int ind = -1;

int r = 0;

int c = m-1;

while (r < n && c >=0) {

if (arr[r][c] == 1) {

ind = r;

c -= 1;

}

else {

r ++;

}

}

System.***out***.println(ind);

}

public static void main(String[] args) {

Main obj = new Main();

obj.rowWithMax1(new int[][] {

{1, 0, 0, 1, 1, 0, 1},

{0, 1, 1, 0, 0, 1, 1},

{1, 1, 1, 1, 0, 0, 0}

});

}

}

3) 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

import java.util.Set;

import java.util.HashSet;

public class Main {

public void rowWithMax1(int[] arr) {

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

int max = 0;

for (int i:arr) {

set.add(i);

}

for (int num:arr) {

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

int curr = num;

int len = 1;

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

curr += 1;

len ++;

}

max = Math.*max*(max,len);

}

}

System.***out***.print(max);

}

public static void main(String[] args) {

Main obj = new Main();

obj.rowWithMax1(new int[]

{3,4,5,6,1,3,4,5,8,9,10,11,12,13,14,15,16,17}

);

}

}

4) Given a string S, find the longest palindromic substring in S.**Substring of string S:** S[ i . . . . j ] where 0 ≤ i ≤ j < len(S)**. Palindrome string:** A string which reads the same backwards. More formally, S is palindrome if reverse(S) = S.**Incase of conflict**, return the substring which occurs first ( with the least starting index ).

**Example 1:**

**Input:**

S = "aaaabbaa"

**Output:**

aabbaa

**Explanation:**

The longest palindrome string present in

the given string is "aabbaa".

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **longestPalindrome()** which takes string Sas input parameters and returns longest pallindrome present in string.

**Expected Time Complexity:** O(|S|2)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ |S| ≤ 104

public class Main {

public String longestPalindrome(String s) {

if (s == null || s.length() < 1) return "";

int start = 0, end = 0;

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

int len1 = expandAroundCenter(s, i, i);

int len2 = expandAroundCenter(s, i, i + 1);

int len = Math.*max*(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private int expandAroundCenter(String s, int left, int right) {

while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {

left--;

right++;

}

return right - left - 1;

}

public static void main(String[] args) {

Main obj = new Main();

System.***out***.println(obj.longestPalindrome("aabbaacc"));

}

}

5) 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.

**Constraints:**  
2 ≤ n ≤ 5  
0 ≤ mat[i][j] ≤ 1

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class Main {

public ArrayList<String> findPath(int[][] m) {

int n = m.length;

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

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

return result;

}

backtrack(m, 0, 0, n, "", result);

Collections.*sort*(result);

return result;

}

private void backtrack(int[][] m, int i, int j, int n, String path, List<String> result) {

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

result.add(path);

return;

}

m[i][j] = 0;

if (isSafe(m, i + 1, j, n)) {

backtrack(m, i + 1, j, n, path + "D", result);

}

if (isSafe(m, i, j + 1, n)) {

backtrack(m, i, j + 1, n, path + "R", result);

}

if (isSafe(m, i - 1, j, n)) {

backtrack(m, i - 1, j, n, path + "U", result);

}

if (isSafe(m, i, j - 1, n)) {

backtrack(m, i, j - 1, n, path + "L", result);

}

m[i][j] = 1;

}

private boolean isSafe(int[][] m, int i, int j, int n) {

return i >= 0 && i < n && j >= 0 && j < n && m[i][j] == 1;

}

public static void main(String[] args) {

Main obj = new Main();int[][] maze = {

{1, 0, 0, 0},

{1, 1, 0, 1},

{1, 1, 0, 0},

{0, 1, 1, 1}

};

ArrayList<String> paths = obj.findPath(maze);

if (paths.isEmpty()) {

System.***out***.println("-1");

} else {

for (String path : paths) {

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

}

}

}

}