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**

**Input:** s1 = "geeks", s2 = "kseeg" **Output:** true**Explanation:** Both the string have same characters with same frequency. So, they are anagrams.

CODE:

class Solution {

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

int x=s1.length();

int y=s2.length();

if(x!=y){

return false;

}

char[] arr1=s1.toCharArray();

char[] arr2=s2.toCharArray();

Arrays.sort(arr1);

Arrays.sort(arr2);

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

if(arr1[i]!=arr2[i]){

return false;

}

}

return true;

}

}

TimeComplexity:O(n)

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.

**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.

class Solution {

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

int maxone = 0;

int maxrow = 0;

boolean flag = true;

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

int count = 0;

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

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

count += 1;

flag = false;

}

if (count > maxone) {

maxone = count;

maxrow = i;

}

}

}

if (flag) {

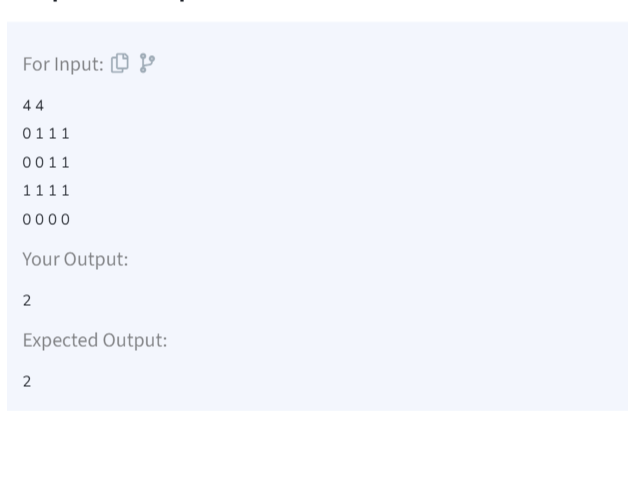
return -1;

}

return maxrow;

}

}



Time Complexity:O(n\*m)

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.**

**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.

CODE:

class Solution {

public int findLongestConseqSubseq(int[] arr) {

if (arr.length == 0) return 0;

Arrays.sort(arr);

int longestStreak = 1;

int currentStreak = 1;

for (int i = 1; i < arr.length; i++) {

if (arr[i] == arr[i - 1]) continue;

if (arr[i] == arr[i - 1] + 1) {

currentStreak++;

} else {

longestStreak = Math.max(longestStreak, currentStreak);

currentStreak = 1;

}

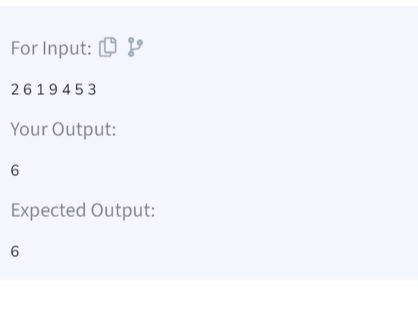
}

longestStreak = Math.max(longestStreak, currentStreak);

return longestStreak;

}

}



Time Complexity:O(n)

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 ).

**Input:**

S = "aaaabbaa"

**Output:**

aabbaa

**Explanation:**

The longest palindrome string present in

the given string is "aabbaa".

CODE:

class Solution{

public static 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 static 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) {

String S = "aaaabbaa";

System.out.println(longestPalindrome(S));

}

}



Time Complexity:O(n^2)

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.Input: mat[][] = [[1, 0], [1, 0]] Output: -1 Explanation: No path exists and destination cell is blocked.

CODE:

import java.util.\*;

public class RatInMaze {

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

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

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

return paths;

}

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

dfs(mat, n, 0, 0, "", paths, visited);

if (paths.isEmpty()) {

paths.add("-1");

}

return paths;

}

private static void dfs(int[][] mat, int n, int row, int col, String path, List<String> paths, boolean[][] visited) {

if (row == n - 1 && col == n - 1) {

paths.add(path);

return;

}

visited[row][col] = true;

if (isSafe(mat, n, row + 1, col, visited)) {

dfs(mat, n, row + 1, col, path + "D", paths, visited);

}

if (isSafe(mat, n, row, col - 1, visited)) {

dfs(mat, n, row, col - 1, path + "L", paths, visited);

}

if (isSafe(mat, n, row, col + 1, visited)) {

dfs(mat, n, row, col + 1, path + "R", paths, visited);

}

if (isSafe(mat, n, row - 1, col, visited)) {

dfs(mat, n, row - 1, col, path + "U", paths, visited);

}

visited[row][col] = false;

}

private static boolean isSafe(int[][] mat, int n, int row, int col, boolean[][] visited) {

return row >= 0 && row < n && col >= 0 && col < n && mat[row][col] == 1 && !visited[row][col];

}

public static void main(String[] args) {

int[][] mat = {

{1, 0},

{1, 0}

};

int n = mat.length;

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

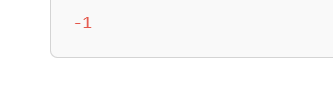
for (String path : result) {

System.out.println(path);

}

}

}



Time Complexity: O(2(n×n))