**P.Bhavya 12.11.2024**  
**AIDS**

**1. Anagram**

**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.**

**Program:**

import java.util.HashMap;

import java.util.Scanner;

class Solution {

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

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

if (s1.length() != s2.length()) return false;

int len = s1.length();

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

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

}

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

if (!map.containsKey(s2.charAt(i)) || map.get(s2.charAt(i)) == 0) return false;

map.put(s2.charAt(i), map.get(s2.charAt(i)) - 1);

}

return true;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter first string: ");

String s1 = scanner.nextLine();

System.out.print("Enter second string: ");

String s2 = scanner.nextLine();

if (areAnagrams(s1, s2)) {

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

} else {

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

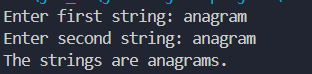
}

scanner.close();

}

}

**Output:**

****

**Time Compllexity:** O(n)

**2. You are given a 2D array consisting of 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.**

**Program:**

import java.util.Scanner;

class RowWithMaxOne {

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 scanner = new Scanner(System.in);

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

int rows = scanner.nextInt();

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

int cols = scanner.nextInt();

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

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

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

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

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

}

}

RowWithMaxOne solution = new RowWithMaxOne();

int result = solution.rowWithMax1s(arr);

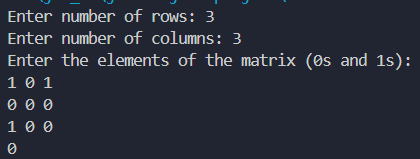
System.out.println(result);

scanner.close();

}

}

**Output:**

****

**Time Complexity:** O(R×C)

**3. Longest consecutive subsequence**

**Given an array of non-negative integers. Find the length of the longest subsequence 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.**

**Program:**

import java.util.HashSet;

import java.util.Scanner;

public class Solution {

public static int findLongestConseqSubseq(int[] arr) {

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

for (int num : arr) {

set.add(num);

}

int longestStreak = 0;

for (int num : arr) {

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

int currentNum = num;

int currentStreak = 1;

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

currentNum++;

currentStreak++;

}

longestStreak = Math.max(longestStreak, currentStreak);

}

}

return longestStreak;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int n = scanner.nextInt();

int[] arr = new int[n];

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

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

arr[i] = scanner.nextInt();

}

int result = findLongestConseqSubseq(arr);

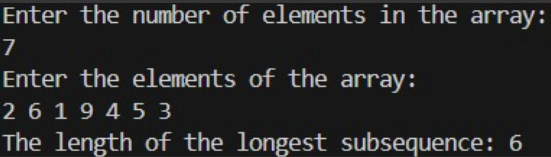
System.out.println("Length of the longest consecutive subsequence is: " + result);

scanner.close();

}

}

**Output:**

****

**Time Complexity:** O(n)

**4. Longest palindrome in a string**

**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".**

**Program:**

import java.util.Scanner;

public 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) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a string: ");

String S = scanner.nextLine();

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

}

}

**Output:**

****

**Time Complexity:** O(n^2)