DSA Practice 2 (12-11-2024)

Name: Atchayaa T

Department: B.Tech CSBS

Register No.: 22CB004

1. **Check if two Strings are Anagrams of each other**
   Given two strings **s1** and **s2** consisting of **lowercase** characters, the task is to check whether the two given strings are **anagrams** 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.

   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. s1 has extra character **‘y’** and s2 has extra characters ‘i’ and ‘c’, so they are not anagrams.*
   Input: *s1 = “g”, s2 = “g”*
   Output: *true*
   Explanation: *Characters in both the strings are same, so they are anagrams.*

   Code:

```java
import java.util.*;
public class CheckAnagram {
    public static Boolean solution(String s1, String s2){
        HashMap<Character,Integer> h = new HashMap<>();
        for (int i=0;i<s1.length();i++){
            h.put(s1.charAt(i),h.getOrDefault(s1.charAt(i),0)+1);
        }
        for (int i=0;i<s2.length();i++){
            h.put(s2.charAt(i),h.getOrDefault(s2.charAt(i),0)-1);
        }
        for (var i : h.entrySet()){
            if (i.getValue()!=0) return false;
        }
        return true;
    }
    public static void main(String[] args) {
        String s1 = "geeks";
        String s2 = "keegs";

        System.out.println(solution(s1,s2));
    }
}
```

   Output: true
   Time Complexity: O(n)

2. Find the row with maximum number of 1s

Given a **binary** 2D array, where each row is **sorted**. Find the row with the maximum number of 1s.

Examples:

Input matrix : *0 1 1 1*
0 0 1 1
1 1 1 1
0 0 0 0

Output: *2*

Explanation: *Row = 2 has maximum number of 1s, that is 4.*

Input matrix : *0 0 1 1*
0 1 1 1
0 0 1 1
0 0 0 0

Output: *1*

Explanation: *Row = 1 has maximum number of 1s, that is 3.*

Code:

```java
public class RowWithMaximumNumberOf1{
    public static void main(String[] args) {
        int m[][] = { { 0, 0, 0, 1 },
        { 0, 1, 1, 1 },
        { 1, 1, 1, 1 },
        { 0, 0, 0, 0 } };

        int r = m.length;
        int c = m[0].length;
        int ans = -1;

        int ri=0;
        int ci = m[0].length-1;

        while (ri<r && ci>=0){
            if (m[ri][ci]==1){
                ans = ri;
                ci = ci-1;
            }
            else{
                ri=ri+1;
            }
        }
        ans = ans+1;
        System.out.println("Row with Maximum Number of 1s is "+ans+"th row.");


    }
}
```

Output: Row with Maximum Number of 1s is 3th row.

Time Complexity: O(n)

3. Longest Consecutive Subsequence

Given an array of 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[] = {1, 9, 3, 10, 4, 20, 2}*
Output: *4*
Explanation: *The subsequence 1, 3, 4, 2 is the longest subsequence of consecutive elements*

Input: *arr[] = {36, 41, 56, 35, 44, 33, 34, 92, 43, 32, 42}*
Output: *5*
Explanation: *The subsequence 36, 35, 33, 34, 32 is the longest subsequence of consecutive elements.*

Code:

```java
import java.util.HashSet;

public class LongestConsecutiveSubsequence {
    public static void main(String[] args) {
        int arr[] = { 1, 9, 3, 10, 4, 20, 2 };
        HashSet<Integer> h = new HashSet<>();
        for (int i=0;i<arr.length;i++){
            h.add(arr[i]);
        }
        int ans=-1;
        for (int i=0;i<arr.length;i++){
            if (!h.contains(arr[i]-1)){
                int temp = arr[i];
                while (h.contains(temp)){
                    temp+=1;
                }
            ans = Math.max(ans,temp-arr[i]);
            }
        }
        System.out.println("Longest Consecutive Subsequence is of length: "+ans);
    }
}
```

Output: Longest Consecutive Subsequence is of length: 4
Time Complexity: O(n)

4. Longest Palindromic Substring

Given a string **str**, the task is to find the longest substring which is a palindrome. If there are multiple answers, then return the first appearing substring.

Examples:

Input: *str = “forgeeksskeegfor”*
Output: *“geeksskeeg”*
Explanation: *There are several possible palindromic substrings like “kssk”, “ss”, “eeksskee” etc. But the substring “geeksskeeg” is the longest among all.*

Input: *str = “Geeks”*
Output: *“ee”*

Input: *str = “abc”*
Output: *“a”*

Input: *str = “”*
Output: *“”*

Code:

```java
public class Longest_Palindromic_Substring {
    static String solution(String s) {
        int n = s.length();
        boolean[][] dp = new boolean[n][n];

        int maxLen = 1;
        int start = 0;

        for (int i = 0; i < n; ++i)
            dp[i][i] = true;

        for (int i = 0; i < n - 1; ++i) {
            if (s.charAt(i) == s.charAt(i + 1)) {
                dp[i][i + 1] = true;
                start = i;
                maxLen = 2;
            }
        }

        for (int k = 3; k <= n; ++k) {
            for (int i = 0; i < n - k + 1; ++i) {
                int j = i + k - 1;

                if (dp[i + 1][j - 1] && s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = true;

                    if (k > maxLen) {
                        start = i;
                        maxLen = k;
                    }
                }
            }
        }

        return s.substring(start, start + maxLen);
    }

    public static void main(String[] args) {
        String s = "forgeeksskeegfor";
        System.out.println(solution(s));
    }

}
```

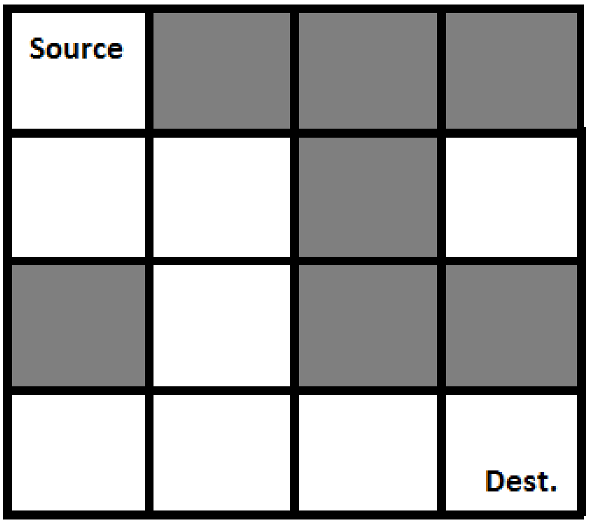
Output: geeksskeegs
Time Complexity: O(n^2)

5. **Rat in a Maze Problem**

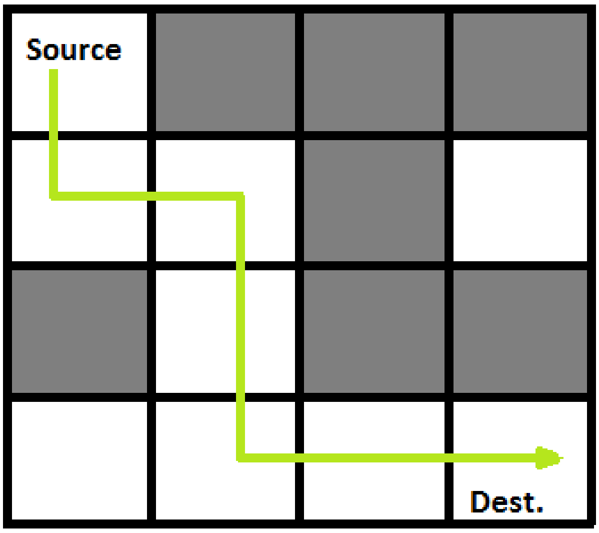
Consider a rat placed at **(0, 0)** in a square matrix of order **N * N**. It has to reach the destination at **(N – 1, N – 1)**. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are **‘U'(up)**, **‘D'(down)**, **‘L’ (left)**, **‘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. Return the list of paths in lexicographically increasing order.

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.

Examples:
Input:

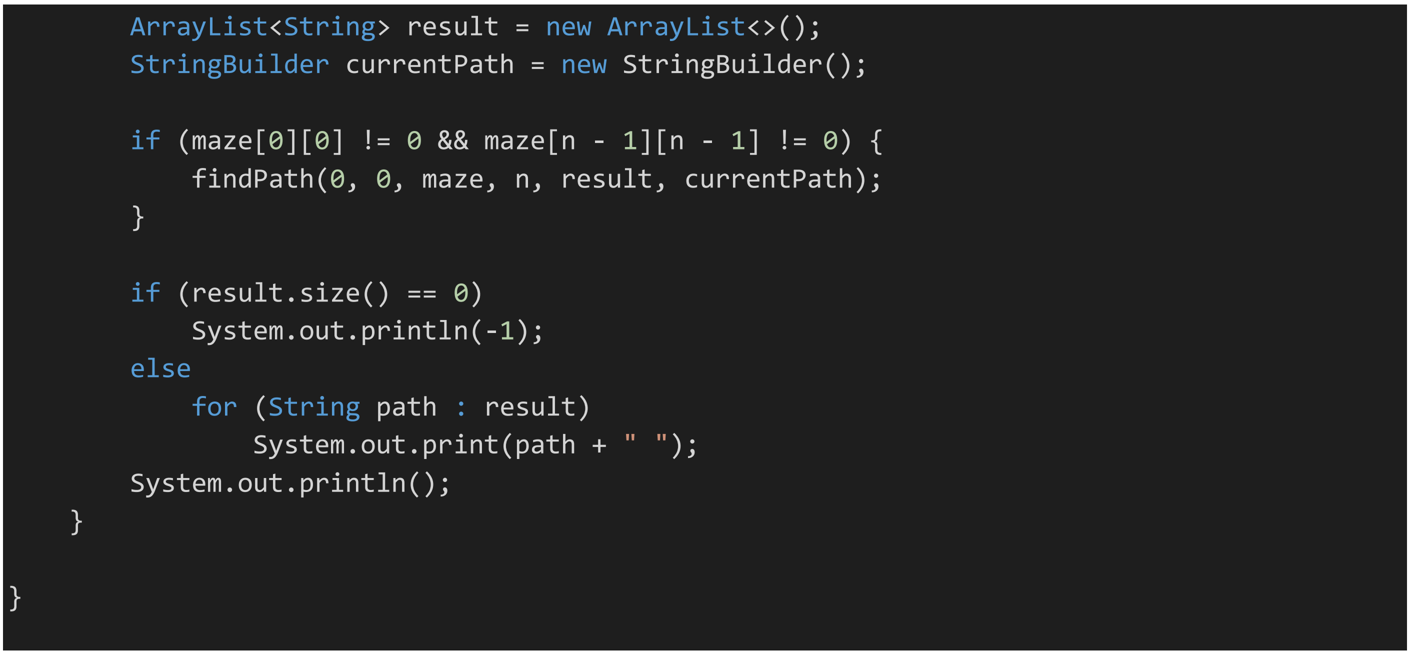


Output: DRDDRR
Explanation:



Code:

```
import java.util.ArrayList;
import java.util.List;
public class RatInAMaze {

    static String direction = "DLRU";
    static int[] dr = { 1, 0, 0, -1 };
    static int[] dc = { 0, -1, 1, 0 };

    static boolean isValid(int row, int col, int n, int[][] maze) {
        return row >= 0 && col >= 0 && row < n && col < n && maze[row][col] == 1;
    }

    static void findPath(int row, int col, int[][] maze, int n, ArrayList<String> ans,
StringBuilder currentPath) {
        if (row == n - 1 && col == n - 1) {
            ans.add(currentPath.toString());
            return;
        }
        maze[row][col] = 0;

        for (int i = 0; i < 4; i++) {
            int nextrow = row + dr[i];
            int nextcol = col + dc[i];

            if (isValid(nextrow, nextcol, n, maze)) {
                currentPath.append(direction.charAt(i));
                findPath(nextrow, nextcol, maze, n, ans, currentPath);
                currentPath.deleteCharAt(currentPath.length() - 1);
            }
        }
        maze[row][col] = 1;
    }

    public static void main(String[] args) {
        int[][] maze = { { 1, 0, 0, 0 }, { 1, 1, 0, 1 }, { 1, 1, 0, 0 }, { 0, 1, 1, 1 } };
        int n = maze.length;
```



```
        ArrayList<String> result = new ArrayList<>();
        StringBuilder currentPath = new StringBuilder();

        if (maze[0][0] != 0 && maze[n - 1][n - 1] != 0) {
            findPath(0, 0, maze, n, result, currentPath);
        }

        if (result.size() == 0)
            System.out.println(-1);
        else
            for (String path : result)
                System.out.print(path + " ");
        System.out.println();
    }

}
```

Output: DDRDRR DRDDRR

Time Complexity: O(3^(m*n))