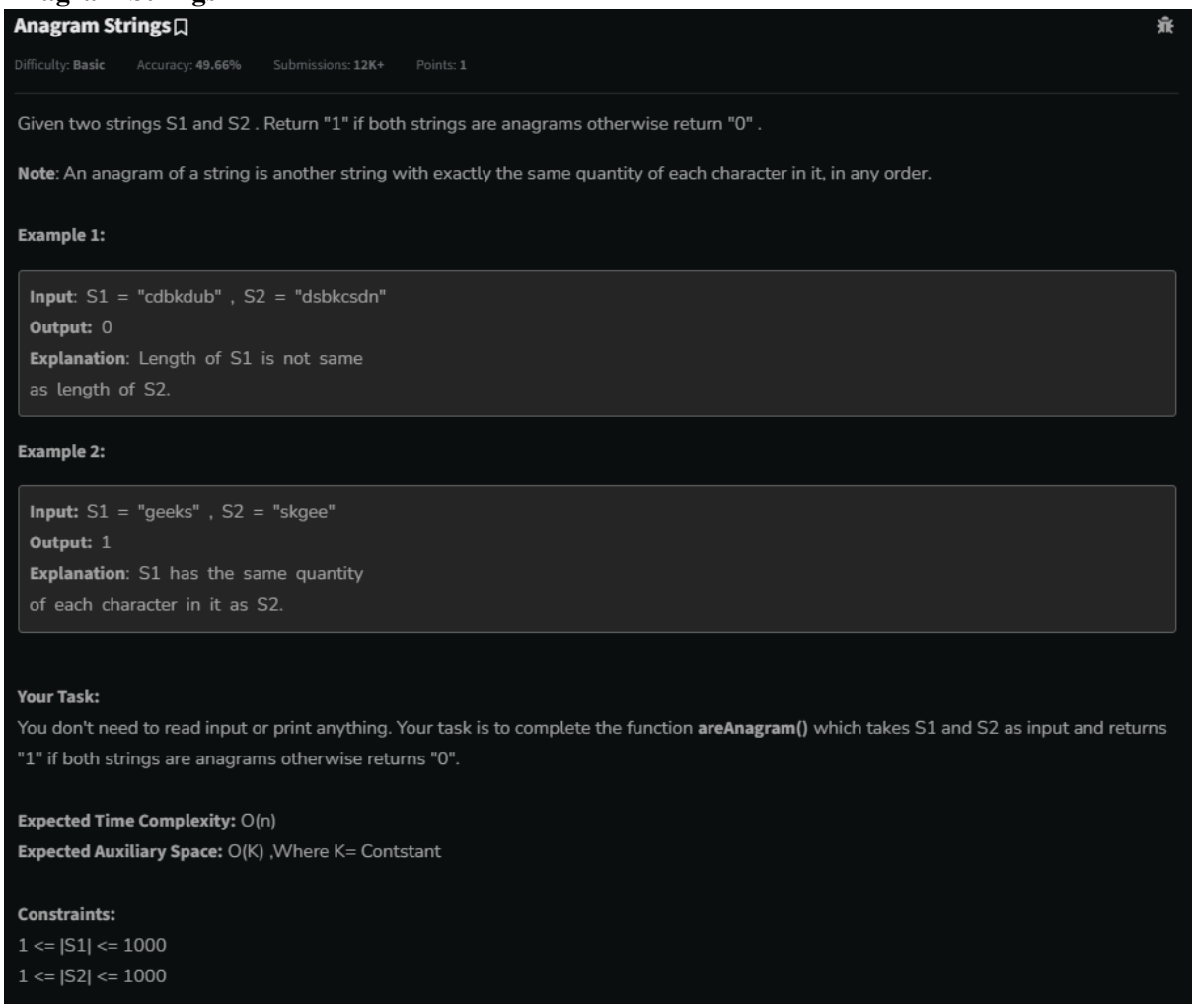
Coding Practice Set-3

1. Anagram Strings



Anagram Strings

Difficulty: **Basic** Accuracy: **49.66%** Submissions: **12K+** Points: **1**

Given two strings S1 and S2 . Return "1" if both strings are anagrams otherwise return "0" .

Note: An anagram of a string is another string with exactly the same quantity of each character in it, in any order.

Example 1:

Input: S1 = "cdbkdub" , S2 = "dsbkcsdn"
Output: 0
Explanation: Length of S1 is not same as length of S2.

Example 2:

Input: S1 = "geeks" , S2 = "skgee"
Output: 1
Explanation: S1 has the same quantity of each character in it as S2.

Your Task:
You don't need to read input or print anything. Your task is to complete the function **areAnagram()** which takes S1 and S2 as input and returns "1" if both strings are anagrams otherwise returns "0".

Expected Time Complexity: O(n)
Expected Auxiliary Space: O(K) ,Where K= Contstant

Constraints:
1 <= |S1| <= 1000
1 <= |S2| <= 1000

Code:

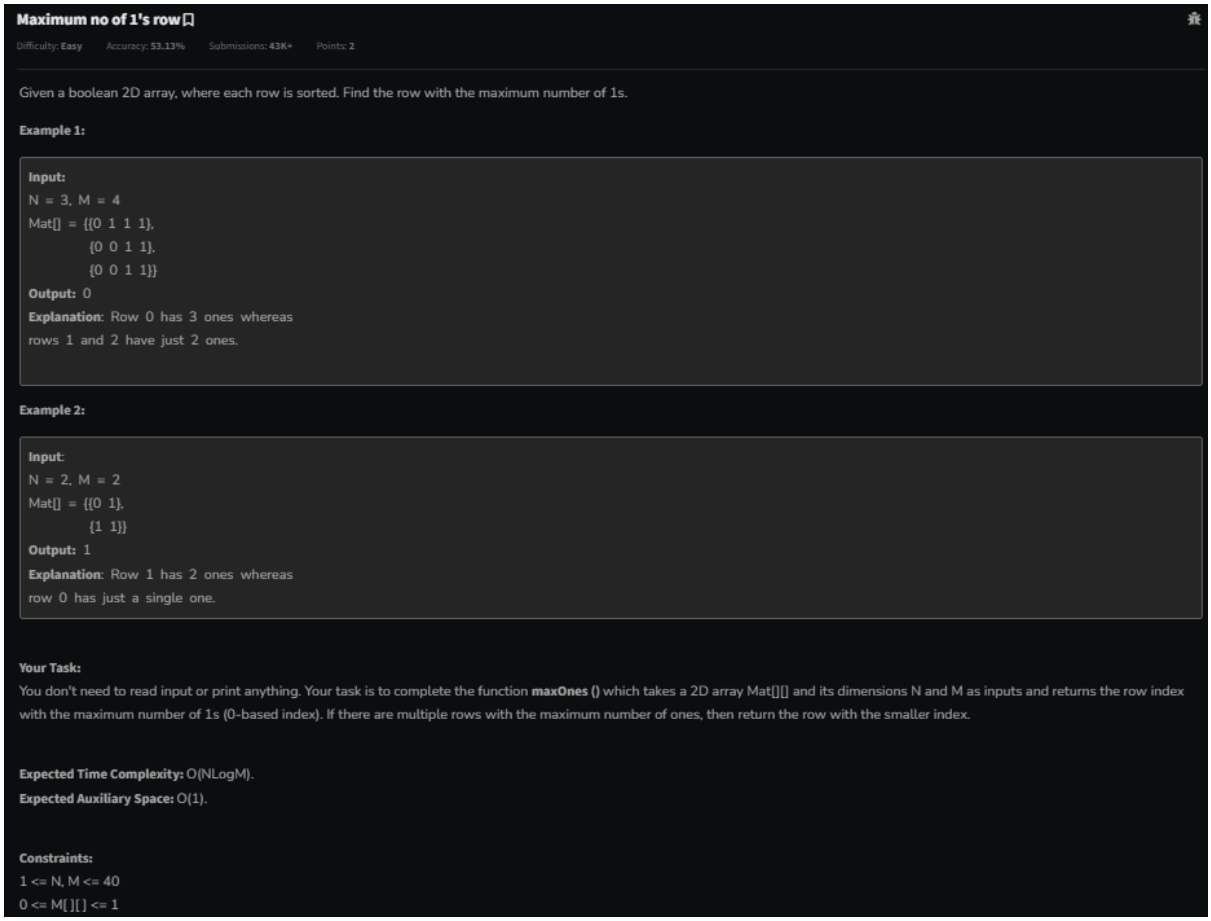
```
import java.util.Arrays;

public class Solution {
   public static String areAnagram(String S1, String S2) {
      if (S1.length() != S2.length()) {
         return "0";
      }

      char[] arr1 = S1.toCharArray();
      char[] arr2 = S2.toCharArray();

      Arrays.sort(arr1);
      Arrays.sort(arr2);

      if (Arrays.equals(arr1, arr2)) {
         return "1";
      } else {
         return "0";
```

```
        }
    }

    public static void main(String[] args) {
        System.out.println(areAnagram("geeks", "skgee"));  // Output: "1"
    }
}
```

Output:

Time Complexity: O(n log n)

2. Row with max 1's



Maximum no of 1's row

Difficulty: Easy Accuracy: 53.13% Submissions: 43K+ Points: 2

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

Example 1:

Input:
N = 3, M = 4
Mat[] = {{0 1 1 1},
{0 0 1 1},
{0 0 1 1}}
Output: 0
Explanation: Row 0 has 3 ones whereas rows 1 and 2 have just 2 ones.

Example 2:

Input:
N = 2, M = 2
Mat[] = {{0 1},
{1 1}}
Output: 1
Explanation: Row 1 has 2 ones whereas row 0 has just a single one.

Your Task:
You don't need to read input or print anything. Your task is to complete the function **maxOnes ()** which takes a 2D array Mat[][] and its dimensions N and M as inputs and returns the row index with the maximum number of 1s (0-based index). If there are multiple rows with the maximum number of ones, then return the row with the smaller index.

Expected Time Complexity: O(NLogM).
Expected Auxiliary Space: O(1).

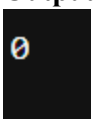
Constraints:
1 <= N, M <= 40
0 <= M[ ][ ] <= 1

Code:

```
public class Solution {
    public static int maxOnes(int[][] Mat, int N, int M) {
        int maxRow = -1;
        int maxCount = -1;

        for (int i = 0; i < N; i++) {
            int count = 0;
            for (int j = 0; j < M; j++) {
                if (Mat[i][j] == 1) {
                    count++;
                }
            }

            if (count > maxCount) {
                maxCount = count;
                maxRow = i;
            }
        }

        return maxRow;
    }
```

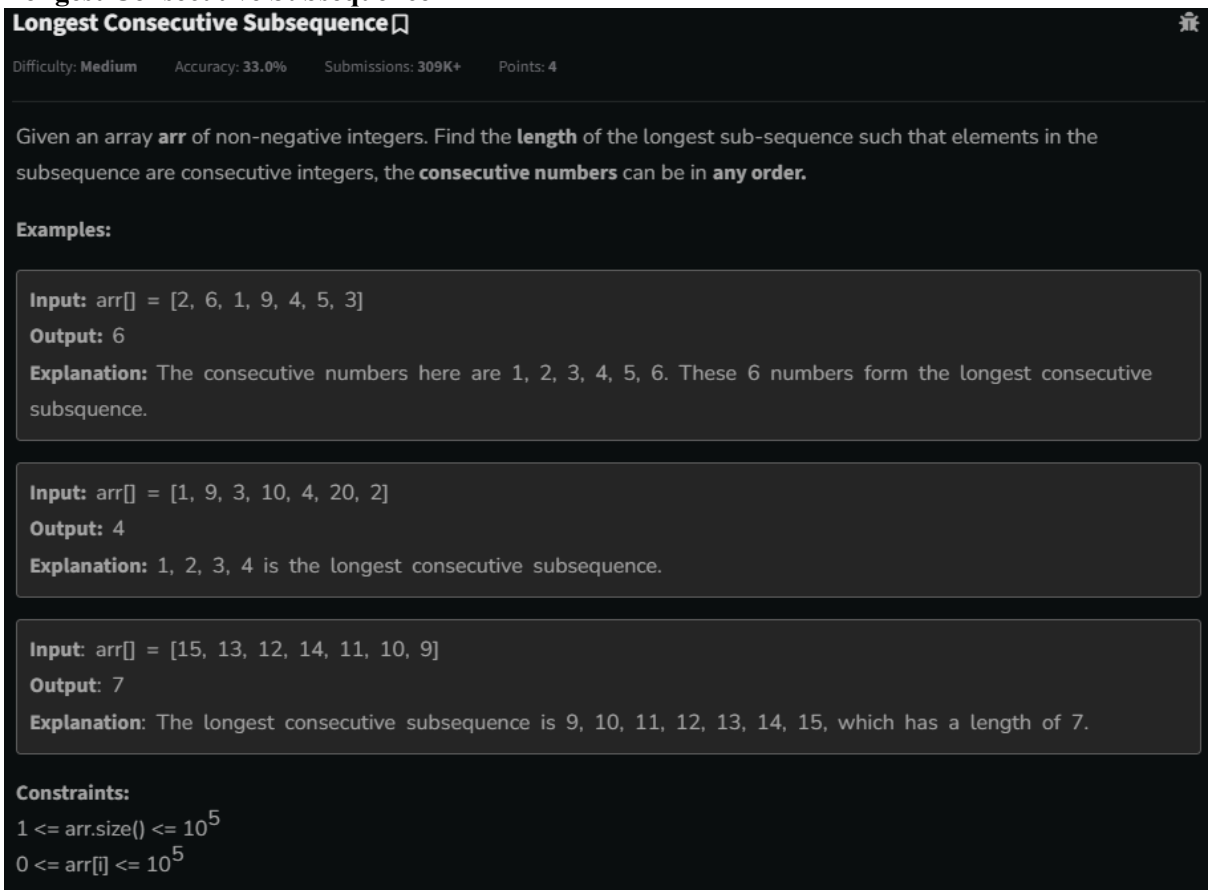
```
    public static void main(String[] args) {
        int[][] Mat = {
            {0, 1, 1, 1},
            {0, 0, 1, 1},
            {0, 0, 1, 1}
        };
        System.out.println(maxOnes(Mat, 3, 4));  // Output: 0
    }
}
```

Output:



Time Complexity: O(N*M)

3. **Longest Consecutive Subsequence**



Longest Consecutive Subsequence

Difficulty: **Medium** Accuracy: **33.0%** Submissions: **309K+** Points: **4**

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.

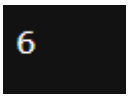
Constraints:
$1 <= arr.size() <= 10^5$
$0 <= arr[i] <= 10^5$

Code:

```
import java.util.*;

public class Solution {
    public static int longestConsecutiveSubsequence(int[] arr) {
        Set<Integer> set = new HashSet<>();
        for (int num : arr) {
            set.add(num);
        }

        int maxLength = 0;

        for (int num : set) {
            if (!set.contains(num - 1)) {
                int currentNum = num;
                int currentStreak = 1;

                while (set.contains(currentNum + 1)) {
                    currentNum++;
                    currentStreak++;
                }

                maxLength = Math.max(maxLength, currentStreak);
            }
```

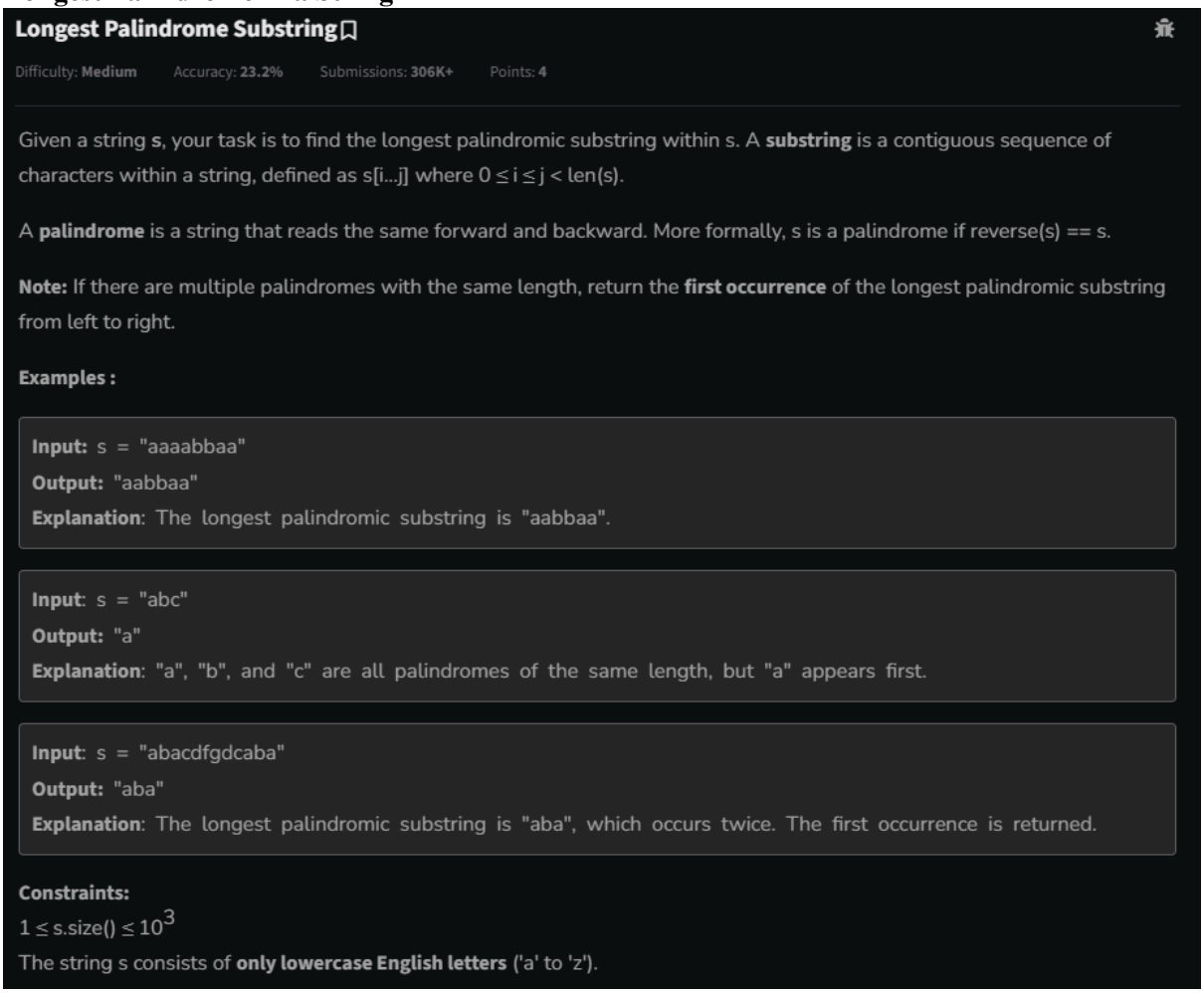
```
        }

        return maxLength;
    }

    public static void main(String[] args) {
        int[] arr = {2, 6, 1, 9, 4, 5, 3};
        System.out.println(longestConsecutiveSubsequence(arr));  // Output: 6
    }
}
```

Output:



Time Complexity: O(N)

4. **Longest Palindrome in a String**



Longest Palindrome Substring

Difficulty: **Medium** Accuracy: **23.2%** Submissions: **306K+** Points: **4**

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 $0 \leq i \leq j < \text{len}(s)$.

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.

Input: s = "abacdfgdcaba"
Output: "aba"
Explanation: The longest palindromic substring is "aba", which occurs twice. The first occurrence is returned.

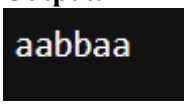
Constraints:
$1 \leq s.size() \leq 10^3$
The string s consists of **only lowercase English letters** ('a' to 'z').

Code:

```
public class Solution {
    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) {
```

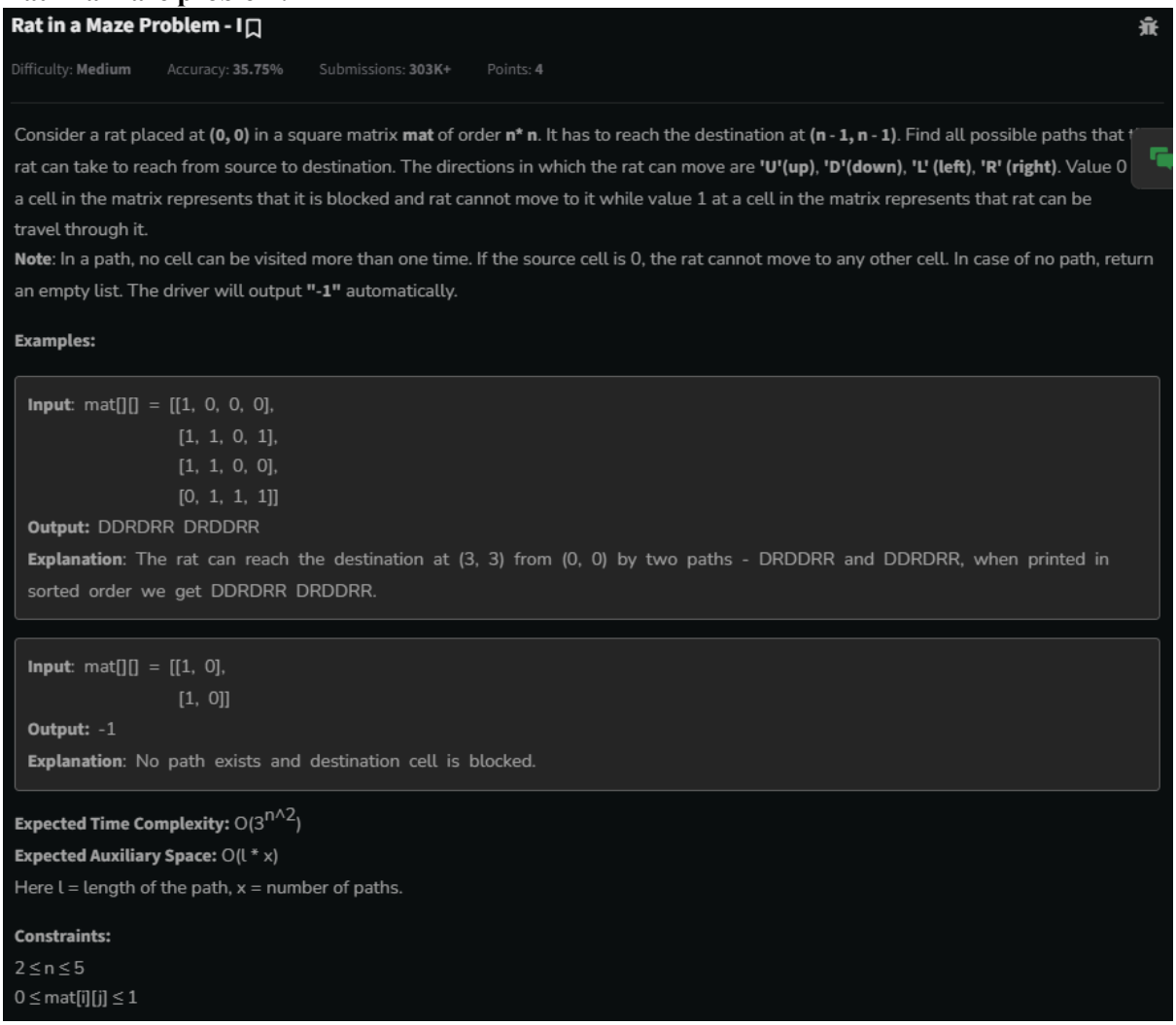
```
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        String s = "aaaabbaa";
        System.out.println(sol.longestPalindrome(s));  // Output: "aabbaa"
    }
}
```

Output:



Time Complexity: $O(N^2)$

5. **Rat in a maze problem:**



Rat in a Maze Problem - I

Difficulty: **Medium** Accuracy: **35.75%** Submissions: **303K+** Points: **4**

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 t 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 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: $O(3^{n^2})$
Expected Auxiliary Space: O(l * x)
Here l = length of the path, x = number of paths.

Constraints:
$2 \leq n \leq 5$
$0 \leq mat[i][j] \leq 1$

Code:

```
import java.util.ArrayList;
import java.util.List;

public class Solution {
    public List<String> findPaths(int[][] mat, int n) {
        List<String> paths = new ArrayList<>();
        boolean[][] visited = new boolean[n][n];
        if (mat[0][0] == 1) {
            findPathsUtil(mat, n, 0, 0, visited, "", paths);
        }
        return paths.isEmpty() ? List.of("-1") : paths;
    }

    private void findPathsUtil(int[][] mat, int n, int i, int j, boolean[][] visited, String path,
List<String> paths) {
        if (i < 0 || j < 0 || i >= n || j >= n || mat[i][j] == 0 || visited[i][j]) {
            return;
        }
```

```
        if (i == n - 1 && j == n - 1) {
            paths.add(path);
            return;
        }

        visited[i][j] = true;

        findPathsUtil(mat, n, i + 1, j, visited, path + "D", paths); // Down
        findPathsUtil(mat, n, i - 1, j, visited, path + "U", paths); // Up
        findPathsUtil(mat, n, i, j + 1, visited, path + "R", paths); // Right
        findPathsUtil(mat, n, i, j - 1, visited, path + "L", paths); // Left

        visited[i][j] = false;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int[][] mat = {{1, 0, 0, 0}, {1, 1, 0, 1}, {1, 1, 0, 0}, {0, 1, 1, 1}};
        List<String> paths = sol.findPaths(mat, 4);
        for (String path : paths) {
            System.out.println(path);
        }
    }
}
```

Output:

```
DDRDRR
DRDDRR
```

Time Complexity: $O(N^2)$