**1.Anagram Strings**

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

Program:

class Solution {

public:

int areAnagram(string s1, string s2) {

int x=s1.size();

int y=s2.size();

if(x!=y) return false;

sort(s1.begin(),s1.end());

sort(s2.begin(),s2.end());

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

{

if(s1[i]!=s2[i]) return false;

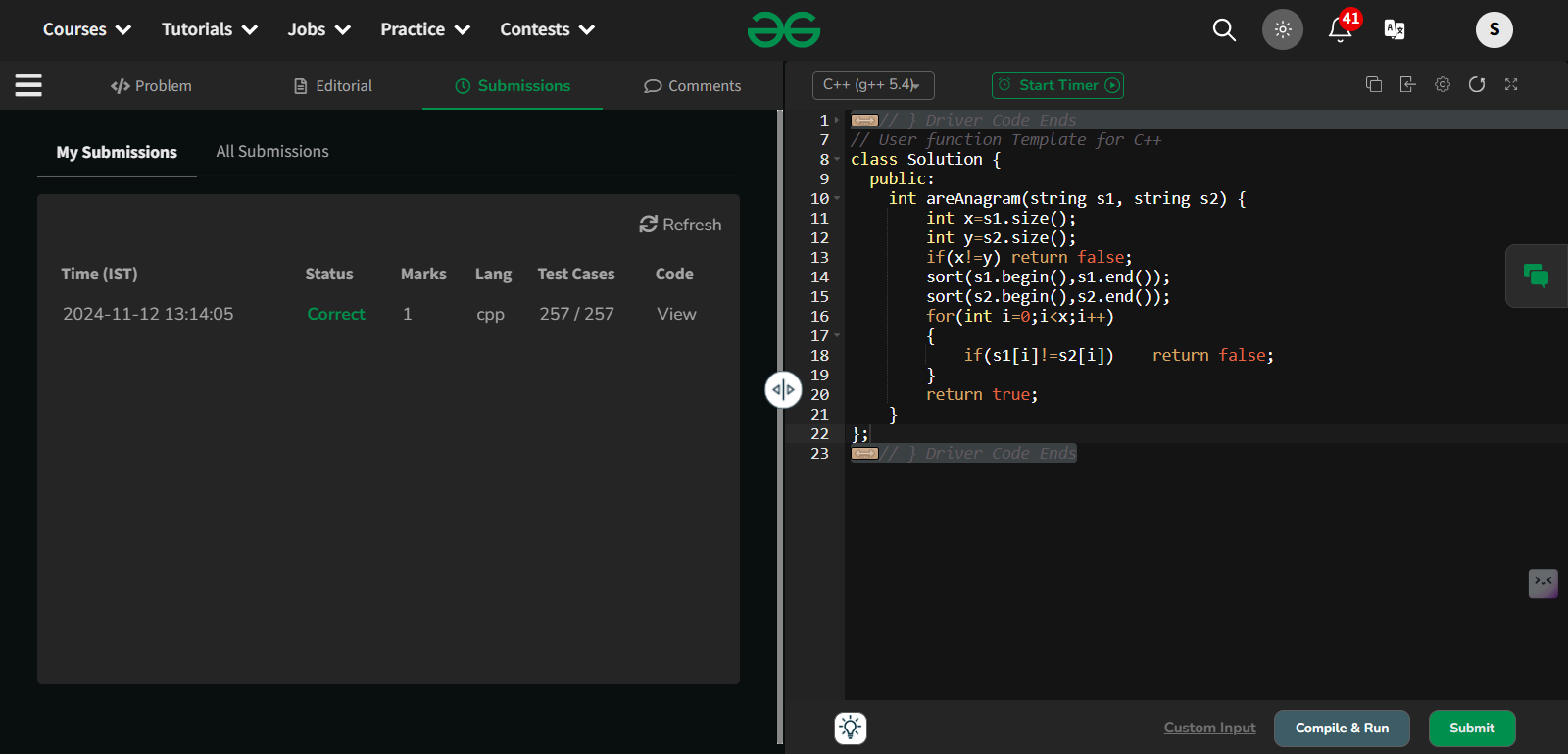
}

return true;

}

};

Output:



**2.maximum number of 1’s row**

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

**Program:**

class Solution

{

public:

int maxOnes (vector <vector <int>> &mat, int N, int M)

{

int maxone=INT\_MIN;

int rowind=-1;

for(int i=0;i<N;i++)

{

int currones=0;

for(int j=0;j<M;j++)

{

if (mat[i][j]==1) currones++;

}

if(maxone<currones)

{

maxone=currones;

rowind=i;

}

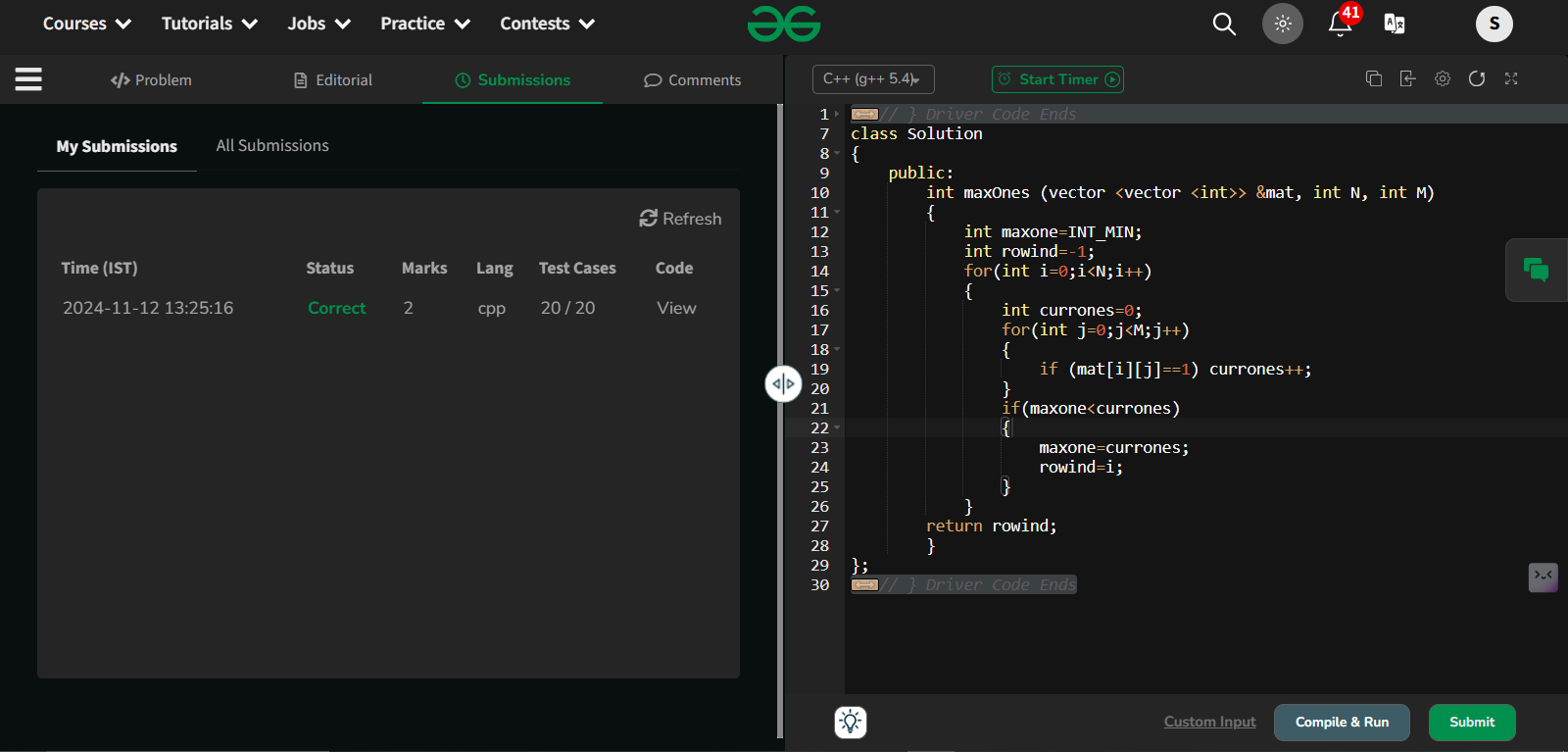
}

return rowind;

}

};

**Output:**

****

3. **Longest consecutive subsequence**

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

**Program:**

class Solution {

public:

// Function to return length of longest subsequence of consecutive integers.

int findLongestConseqSubseq(vector<int>& arr) {

int n=arr.size();

if(n==0) return 0;

if(n==1) return 1;

sort(arr.begin(),arr.end());

int maxcon=1;

int con=1;

for(int i=1;i<n;i++)

{

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

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

else

{

con=1;

}

maxcon=max(con,maxcon);

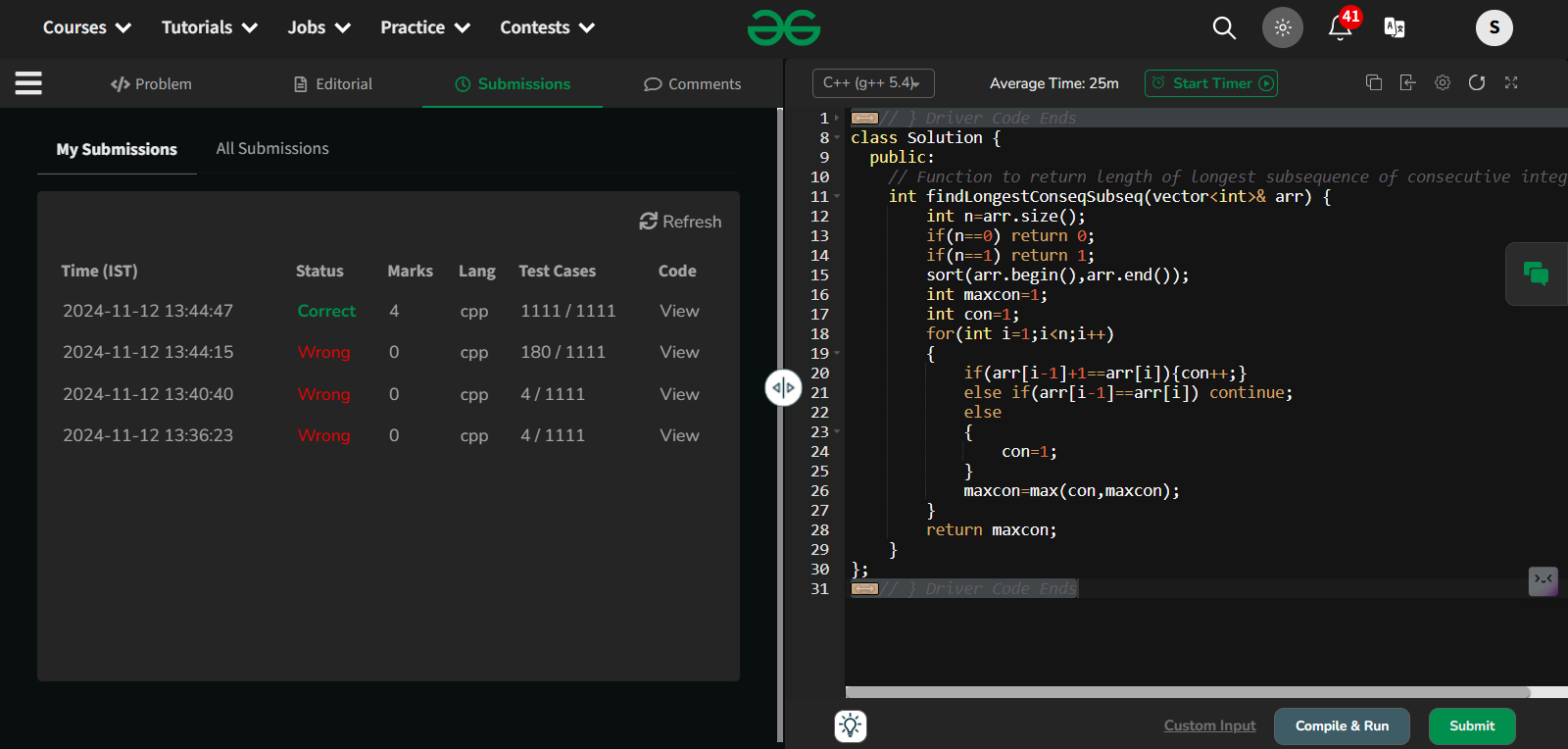
}

return maxcon;

}

};

Output:



4. **Longest Palindrome in a String**

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 ≤ i ≤ j < 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 ≤ s.size() ≤ 103The string s consists of **only lowercase English letters** ('a' to 'z').

Program:

class Solution {

public:

int expandAroundCenter(const string& s, int left, int right) {

while (left >= 0 && right < s.size() && s[left] == s[right]) {

left--;

right++;

}

// Return the length of the palindrome

return right - left - 1;

}

string longestPalindrome(string s) {

if (s.empty()) return "";

int start = 0, maxLength = 1;

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

// Check for odd-length palindromes centered at `i`

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

// Check for even-length palindromes centered between `i` and `i+1`

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

int len = max(len1, len2);

if (len > maxLength) {

maxLength = len;

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

}

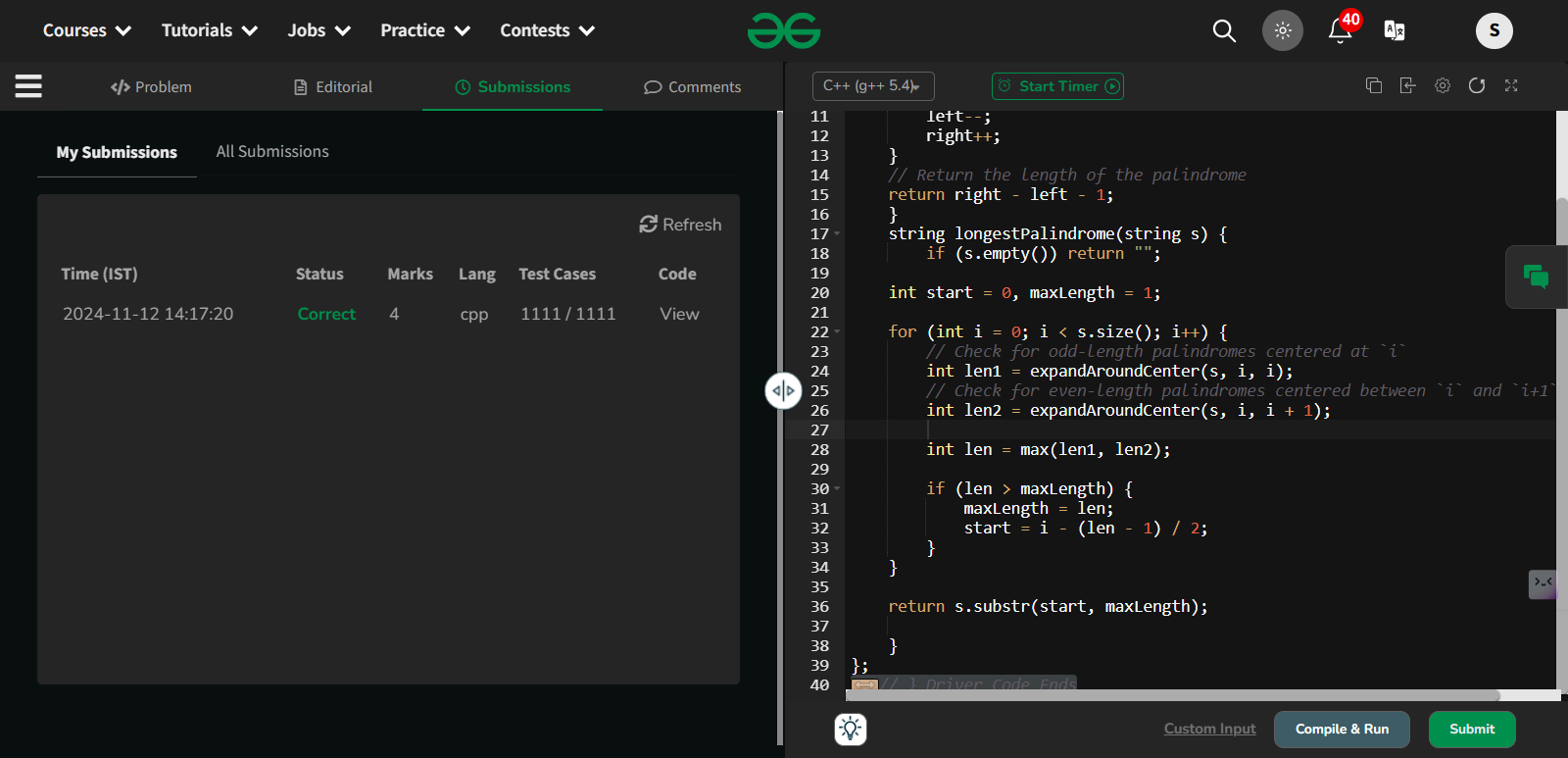
}

return s.substr(start, maxLength);

}

};

Output:



5. **Rat in a Maze Problem - I**

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

Program:

class Solution {

void solve(vector<vector<int>>&mat,vector<string>&ans,string &path,int i,int j,

vector<vector<int>>&vis){

int n=mat.size();

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

ans.push\_back(path);

return;

}

if(i>=n || j>=n){

return;

}

vis[i][j]=1;

if(i-1>=0 && mat[i-1][j]!=0 && !vis[i-1][j]){

path.push\_back('U');

solve(mat,ans,path,i-1,j,vis);

path.pop\_back();

}

if(i+1<n && mat[i+1][j]!=0 && !vis[i+1][j]){

path.push\_back('D');

solve(mat,ans,path,i+1,j,vis);

path.pop\_back();

}

if(j-1>=0 && mat[i][j-1]!=0 && !vis[i][j-1]){

path.push\_back('L');

solve(mat,ans,path,i,j-1,vis);

path.pop\_back();

}

if(j+1<n && mat[i][j+1]!=0 && !vis[i][j+1]){

path.push\_back('R');

solve(mat,ans,path,i,j+1,vis);

path.pop\_back();

}

vis[i][j]=0;

}

public:

vector<string> findPath(vector<vector<int>> &mat) {

int n=mat.size();

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

return {"-1"};

}

vector<string>ans;

string path;

vector<vector<int>>vis(n,vector<int>(n,0));

solve(mat,ans,path,0,0,vis);

return ans;

}

};

Output:

