1. **K-th smallest element**

Given an array arr[] and an integer k where k is smaller than the size of the array, the task is to find the kth smallest element in the given array.

Follow up: Don't solve it using the inbuilt sort function.

Examples :

Input: arr[] = [7, 10, 4, 3, 20, 15], k = 3

Output: 7

Explanation: 3rd smallest element in the given array is 7.

Input: arr[] = [2, 3, 1, 20, 15], k = 4

Output: 15

Explanation: 4th smallest element in the given array is 15.

Expected Time Complexity: O(n+(max\_element) )

Expected Auxiliary Space: O(max\_element)

Constraints:

1 <= arr.size <= 106

1<= arr[i] <= 106

1 <= k <= n  
  
**Program:**  
class Solution {

public:

// arr : given array

// k : find kth smallest element and return using this function

void swap(int &a,int&b)

{

int temp=a;

a=b;

b=temp;

}

int partition(vector<int> &arr,int low,int high)

{

int pivot=arr[high];

int i=low-1;

for(int j=low;j<high;j++)

{

if(arr[j]<pivot)

{

i++;

swap(arr[i],arr[j]);

}

}

swap(arr[i+1],arr[high]);

return i+1;

}

void quicksort(vector<int> &arr,int low,int high)

{

if (low < high) {

int pivotIndex = partition(arr, low, high);

quicksort(arr, low, pivotIndex - 1);

quicksort(arr, pivotIndex + 1, high);

}

}

int kthSmallest(vector<int> &arr, int k) {

int low=0;

int high=arr.size()-1;

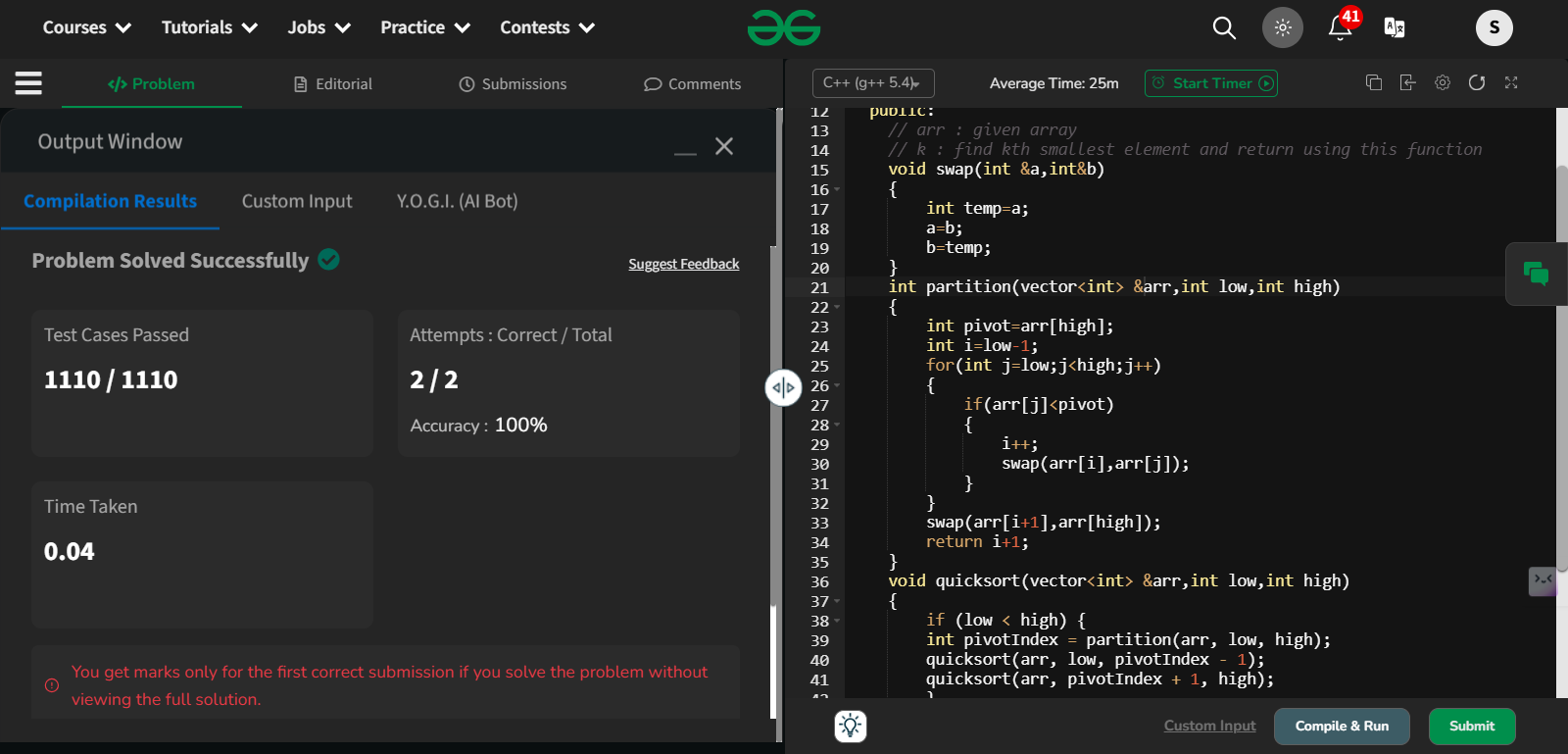
quicksort(arr,low,high);

return arr[k-1];

}

};

**Output:**



Time Complexity: O(N) – average case  
  
**2.Minimize the heights-II**

**Given an array arr[] denoting heights of N towers and a positive integer K.**

**For each tower, you must perform exactly one of the following operations exactly once.**

* **Increase the height of the tower by K**
* **Decrease the height of the tower by K**

**Find out the minimum possible difference between the height of the shortest and tallest towers after you have modified each tower.**

**You can find a slight modification of the problem**[**here**](https://practice.geeksforgeeks.org/problems/minimize-the-heights-i/1/)**.  
Note: It is compulsory to increase or decrease the height by K for each tower. After the operation, the resultant array should not contain any negative integers.**

**Examples :**

**Input: k = 2, arr[] = {1, 5, 8, 10}**

**Output: 5**

**Explanation: The array can be modified as {1+k, 5-k, 8-k, 10-k} = {3, 3, 6, 8}.The difference between the largest and the smallest is 8-3 = 5.**

**Input: k = 3, arr[] = {3, 9, 12, 16, 20}**

**Output: 11**

**Explanation: The array can be modified as {3+k, 9+k, 12-k, 16-k, 20-k} -> {6, 12, 9, 13, 17}.The difference between the largest and the smallest is 17-6 = 11.**

**Expected Time Complexity: O(n\*logn)  
Expected Auxiliary Space: O(n)**

**Constraints  
1 ≤ k ≤ 107  
1 ≤ n ≤ 105  
1 ≤ arr[i] ≤ 107**

**Program:**

class Solution {

public:

int getMinDiff(vector<int> &arr, int k) {

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

int n=arr.size();

int range=arr[n-1]-arr[0];

int mini=arr[0]+k;

int maxi=arr[n-1]-k;

for(int i=0;i<n-1;i++)

{

int minh=min(mini,arr[i+1]-k);

int maxh=max(maxi,arr[i]+k);

if(minh<0) continue;

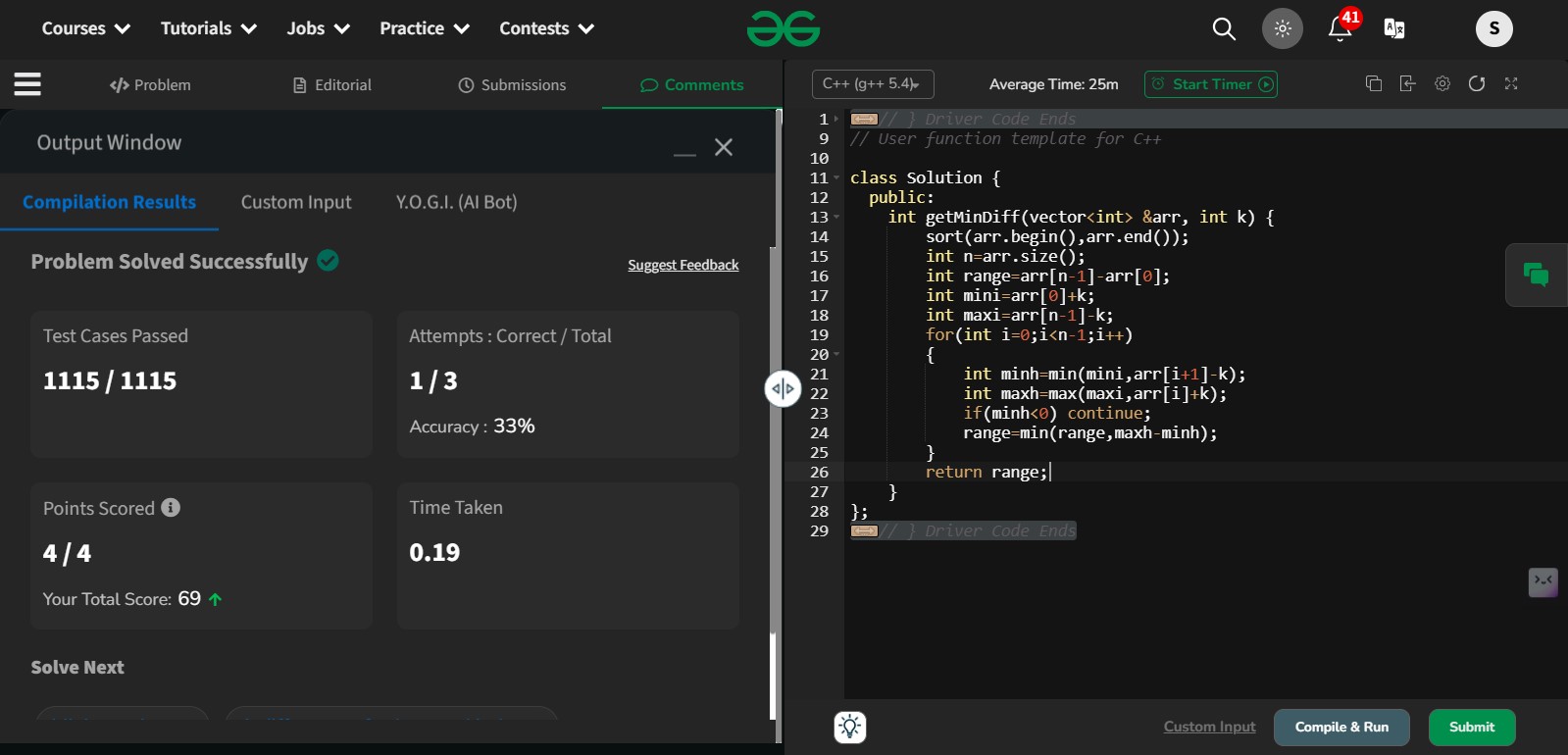
range=min(range,maxh-minh);

}

return range;

}

};  
  
**Output:**



3.Parenthesis Checker

You are given a string **s** representing an expression containing various types of brackets: {}, (), and []. Your task is to determine whether the brackets in the expression are balanced. A balanced expression is one where every opening bracket has a corresponding closing bracket in the correct order.

**Examples :**

**Input**: s = "{([])}"

**Output**: true

**Explanation**:   
- In this expression, every opening bracket has a corresponding closing bracket.  
- The first bracket { is closed by }, the second opening bracket ( is closed by ), and the third opening bracket [ is closed by ].  
- As all brackets are properly paired and closed in the correct order, the expression is considered balanced.

**Input**: s = "()"

**Output**: true

**Explanation**:   
- This expression contains only one type of bracket, the parentheses ( and ).  
- The opening bracket ( is matched with its corresponding closing bracket ).  
- Since they form a complete pair, the expression is balanced.

**Input**: s = "([]"

**Output**: false

**Explanation**:   
- This expression contains only one type of bracket, the parentheses ( and ).  
- The opening bracket ( is matched with its corresponding closing bracket ).  
- Since they form a complete pair, the expression is balanced.

**Constraints:**  
1 ≤ s.size() ≤ 106s[i] ∈ {'{', '}', '(', ')', '[', ']'}

Program:

class Solution {

public:

bool isParenthesisBalanced(string& s) {

stack<char> t;

int start=0;

int end=s.size();

bool a=true;

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

{

//cout<<i<<" ";

if(s[i]=='[' || s[i]=='(' || s[i]=='{')

{

t.push(s[i]);

}

else if(t.empty()) return false;

else if(s[i]==')' && !t.empty() && t.top()=='(')

{

t.pop();

}

else if(s[i]==']' && !t.empty() && t.top()=='[')

{

t.pop();

}

else if(s[i]=='}' && !t.empty() && t.top()=='{' )

{

t.pop();

}

else{

break;

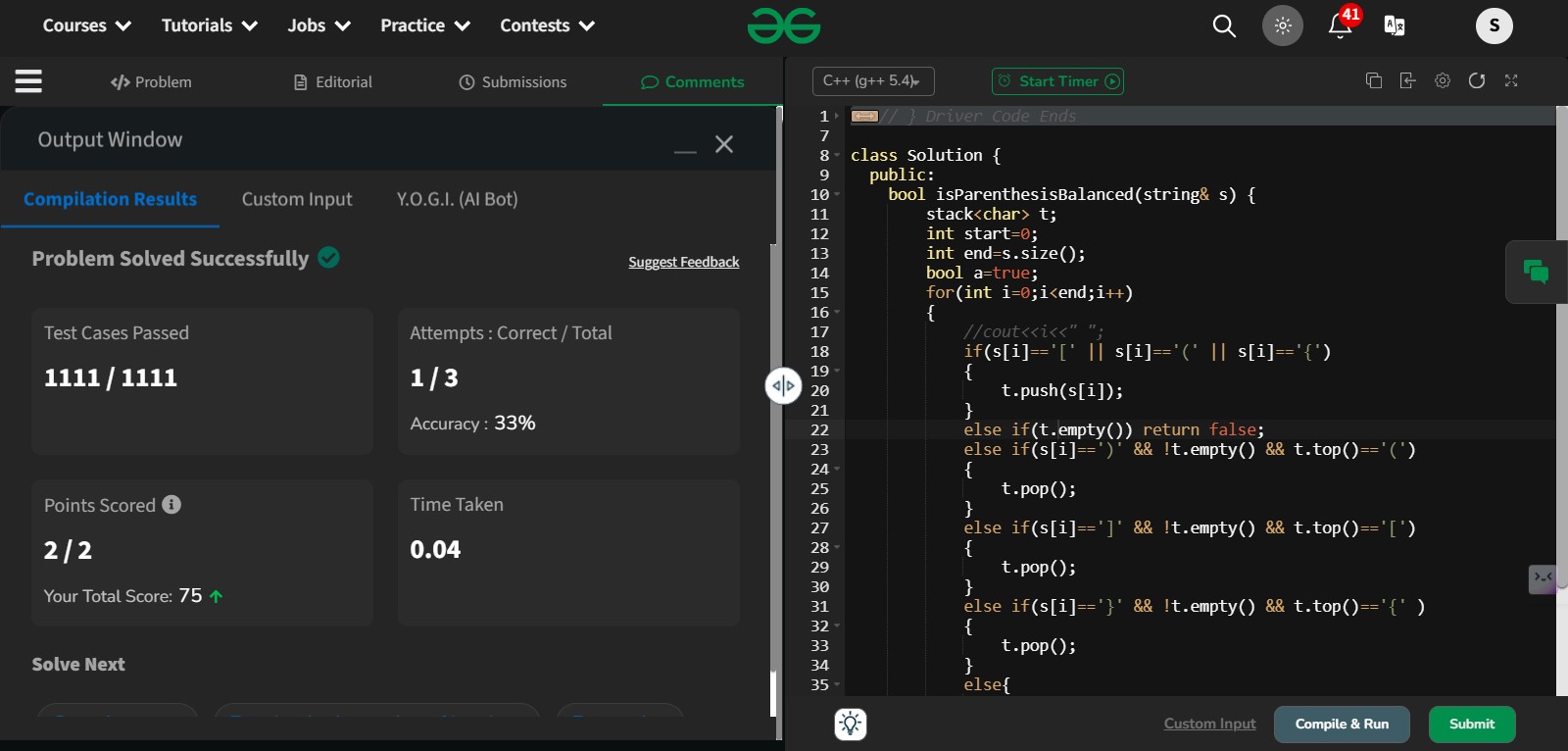
}

}

return t.empty();

}

};  
  
Output:



4. **Equilibrium Point**

Given an array**arr**of non-negative numbers. The task is to find the first **equilibrium point** in an array. The equilibrium point in an array is an index (or position) such that the sum of all elements beforethat index is the same as the sumof elements afterit.

**Note:** Return equilibrium point in 1-based indexing. Return -1 if no such point exists.

**Examples:**

**Input:** arr[] = [1, 3, 5, 2, 2]  
**Output:** 3

**Explanation:** The equilibrium point is at position 3 as the sum of elements before it (1+3) = sum of elements after it (2+2).

**Input:** arr[] = [1]  
**Output:** 1

**Explanation:** Since there's only one element hence it's only the equilibrium point.

**Input:** arr[] = [1, 2, 3]  
**Output:** -1

**Explanation:** There is no equilibrium point in the given array.

**Expected Time Complexity:**O(n)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= arr.size <= 106  
0 <= arr[i] <= 109

Program:

class Solution {

public static int equilibriumPoint(int arr[]) {

int total=0;

int left=0;

for(int i : arr){

total+=i;

}

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

if(left==total-left-arr[i]){

return i+1;

}

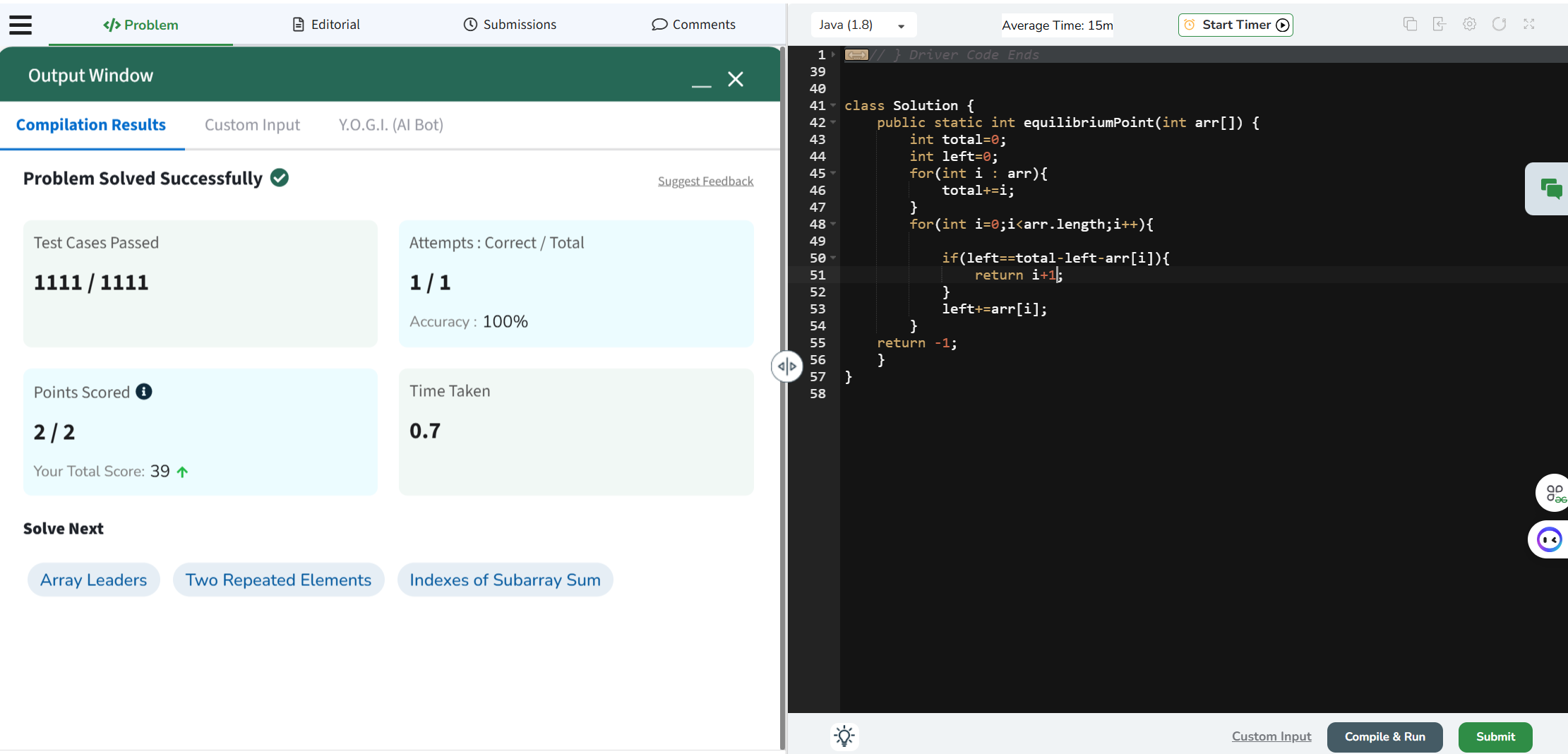
left+=arr[i];

}

return -1;

}

}



5.**Binary Search**

Given a sorted array **arr** and an integer **k**, find the position(0-based indexing) at which k is present in the array using binary search.

Note: If multiple occurrences are there, please return the smallest index.

**Examples:**

**Input:** arr[] = [1, 2, 3, 4, 5], k = 4

**Output:** 3

**Explanation:** 4 appears at index 3.

**Input:** arr[] = [11, 22, 33, 44, 55], k = 445

**Output:** -1

**Explanation:** 445 is not present.

*Note: Try to solve this problem in constant space i.e O(1)*

**Constraints:**1 <= arr.size() <= 1051 <= arr[i] <= 1061 <= k <= 106

Program:

class Solution {

public int binarysearch(int[] arr, int k) {

int low=0;

int high=arr.length-1;

while(low <= high){

int mid=(low+high)/2;

if(arr[mid]==k){

return mid;

}else if(k<arr[mid]){

high=mid-1;

}else{

low=mid+1;

}

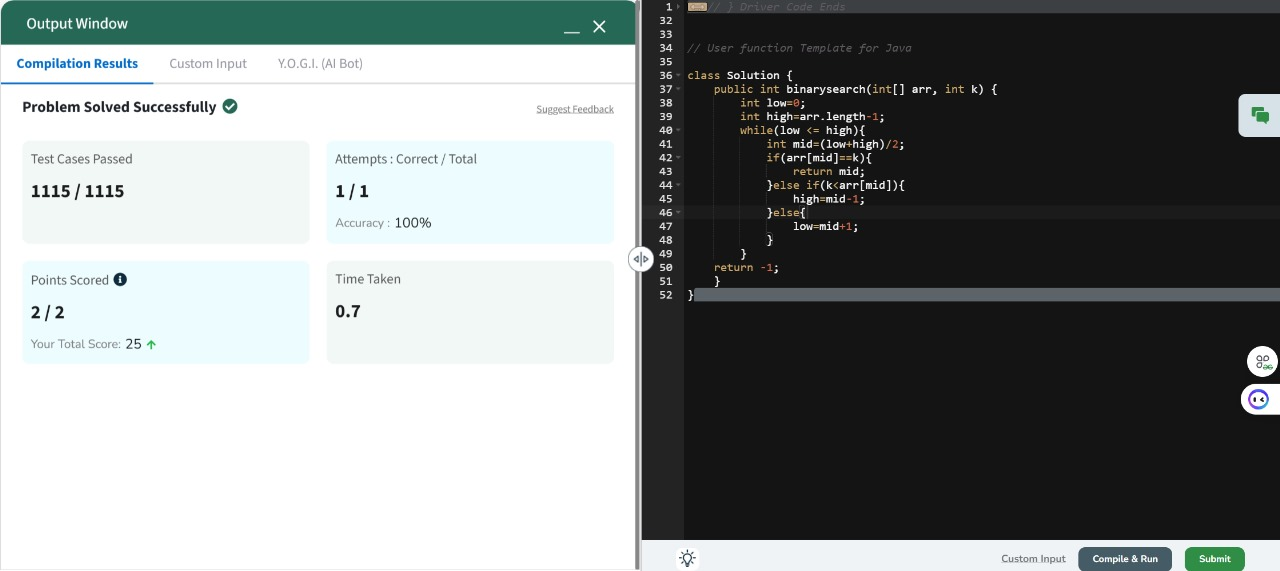
}

return -1;

}

};

Output:



6. **Next Greater Element**

Given an array **arr[ ]** of integers, the task is to find the next greater element for each element of the array in order of their appearance in the array. Next greater element of an element in the array is the nearest element on the right which is greater than the current element.  
If there does not exist next greater of current element, then next greater element for current element is -1. For example, next greater of the last element is always -1.

**Examples**

**Input**: arr[] = [1, 3, 2, 4]

**Output**: [3, 4, 4, -1]

**Explanation**: The next larger element to 1 is 3, 3 is 4, 2 is 4 and for 4, since it doesn't exist, it is -1.

**Input**: arr[] = [6, 8, 0, 1, 3]

**Output**: [8, -1, 1, 3, -1]

**Explanation**: The next larger element to 6 is 8, for 8 there is no larger elements hence it is -1, for 0 it is 1 , for 1 it is 3 and then for 3 there is no larger element on right and hence -1.

**Input**: arr[] = [10, 20, 30, 50]

**Output**: [20, 30, 50, -1]

**Explanation**: For a sorted array, the next element is next greater element also exxept for the last element.

**Input**: arr[] = [50, 40, 30, 10]

**Output**: [-1, -1, -1, -1]

**Explanation**: There is no greater element for any of the elements in the array, so all are -1.

**Constraints:**  
1 ≤ arr.size() ≤ 106  
0 ≤ arr[i] ≤ 109

Program:

class Solution {

public ArrayList<Integer> nextLargerElement(int[] arr) {

int n = arr.length;

int[] result = new int[n];

Arrays.fill(result, -1);

Stack<Integer> stack = new Stack<>();

for (int i = n - 1; i >= 0; i--) {

while (!stack.isEmpty() && arr[stack.peek()] <= arr[i]) {

stack.pop();

}

if (!stack.isEmpty()) {

result[i] = arr[stack.peek()];

}

stack.push(i);

}

ArrayList<Integer> resultList = new ArrayList<>();

for (int val : result) {

resultList.add(val);

}

return resultList;

}

}

Output:

