DSA PRACTICE – DAY 5

Name: Deepak S

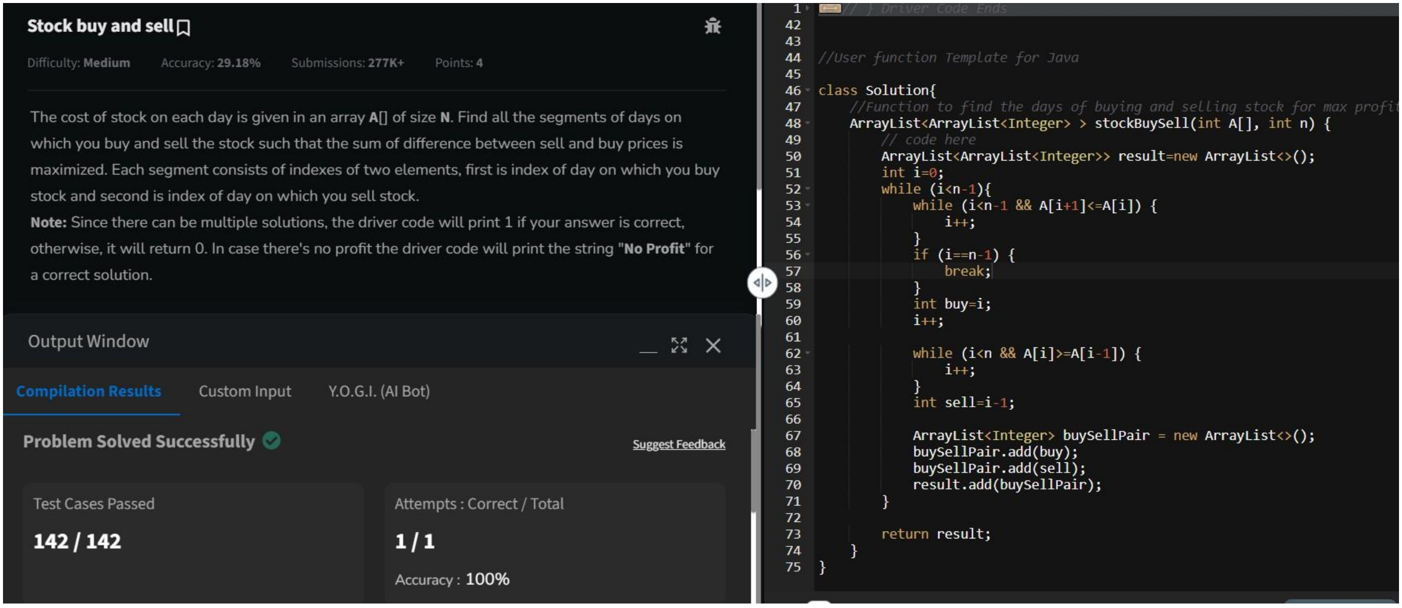
Reg No: 22IT018

1.Stock Buy and Sell

Code Solution:

```
class Solution{
   ArrayList<ArrayList<Integer>> stockBuySell(int A[], int n) {
      ArrayList<ArrayList<Integer>> result=new ArrayList<>();
      int i=0;
      while (i<n-1){
         while (i<n-1 && A[i+1]<=A[i]) {
            i++;}
         if (i==n-1) break;
         int buy=i;
         i++;
         while (i<n && A[i]>=A[i-1]) {
            i++;}
         int sell=i-1;
         ArrayList<Integer> buySellPair = new ArrayList<>();
         buySellPair.add(buy);
         buySellPair.add(sell);
         result.add(buySellPair);
      }
      return result;
   }
}
```

Output:



Stock buy and sell

Difficulty: **Medium** Accuracy: **29.18%** Submissions: **277K+** Points: **4**

The cost of stock on each day is given in an array **A[]** of size **N**. Find all the segments of days on which you buy and sell the stock such that the sum of difference between sell and buy prices is maximized. Each segment consists of indexes of two elements, first is index of day on which you buy stock and second is index of day on which you sell stock.

Note: Since there can be multiple solutions, the driver code will print 1 if your answer is correct, otherwise, it will return 0. In case there's no profit the driver code will print the string **"No Profit"** for a correct solution.

Problem Solved Successfully

Test Cases Passed: **142 / 142**

Attempts : Correct / Total: **1 / 1**

Accuracy : 100%

```
// } Driver Code Ends


//User function Template for Java

class Solution{
    //Function to find the days of buying and selling stock for max profit
    ArrayList<ArrayList<Integer> > stockBuySell(int A[], int n) {
        // code here
        ArrayList<ArrayList<Integer>> result=new ArrayList<>();
        int i=0;
        while (i<n-1){
            while (i<n-1 && A[i+1]<=A[i]) {
                i++;
            }
            if (i==n-1) {
                break;
            }
            int buy=i;
            i++;

            while (i<n && A[i]>=A[i-1]) {
                i++;
            }
            int sell=i-1;

            ArrayList<Integer> buySellPair = new ArrayList<>();
            buySellPair.add(buy);
            buySellPair.add(sell);
            result.add(buySellPair);
        }

        return result;
    }
}
```

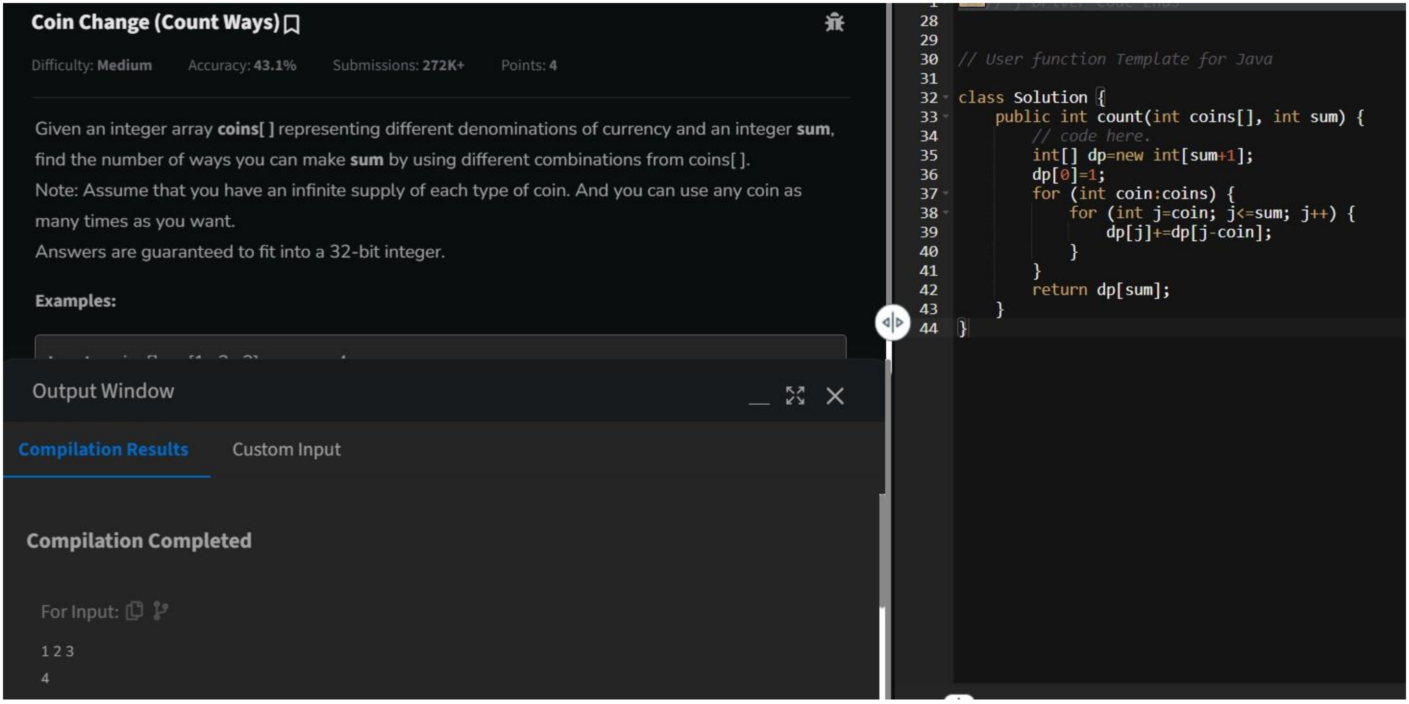
Time complexity: O (n)

2.Minimize heights II

Code Solution:

```
class Solution {
    public int count(int coins[], int sum) {
        // code here.
        int[] dp=new int[sum+1];
        dp[0]=1;
        for (int coin:coins) {
            for (int j=coin; j<=sum; j++) {
                dp[j]+=dp[j-coin];
            }
        }
        return dp[sum];
    }
}
```

Output:



Coin Change (Count Ways)

Difficulty: **Medium** Accuracy: **43.1%** Submissions: **272K+** Points: **4**

Given an integer array **coins[ ]** representing different denominations of currency and an integer **sum**, find the number of ways you can make **sum** by using different combinations from coins[ ].

Note: Assume that you have an infinite supply of each type of coin. And you can use any coin as many times as you want.

Answers are guaranteed to fit into a 32-bit integer.

Examples:

Output Window

Compilation Results Custom Input

Compilation Completed

For Input:

```
1 2 3
4
```

```
// User function Template for Java

class Solution {
    public int count(int coins[], int sum) {
        // code here.
        int[] dp=new int[sum+1];
        dp[0]=1;
        for (int coin:coins) {
            for (int j=coin; j<=sum; j++) {
                dp[j]+=dp[j-coin];
            }
        }
        return dp[sum];
    }
}
```

Time Complexity: O (m*n)
Space Complexity: O (m)

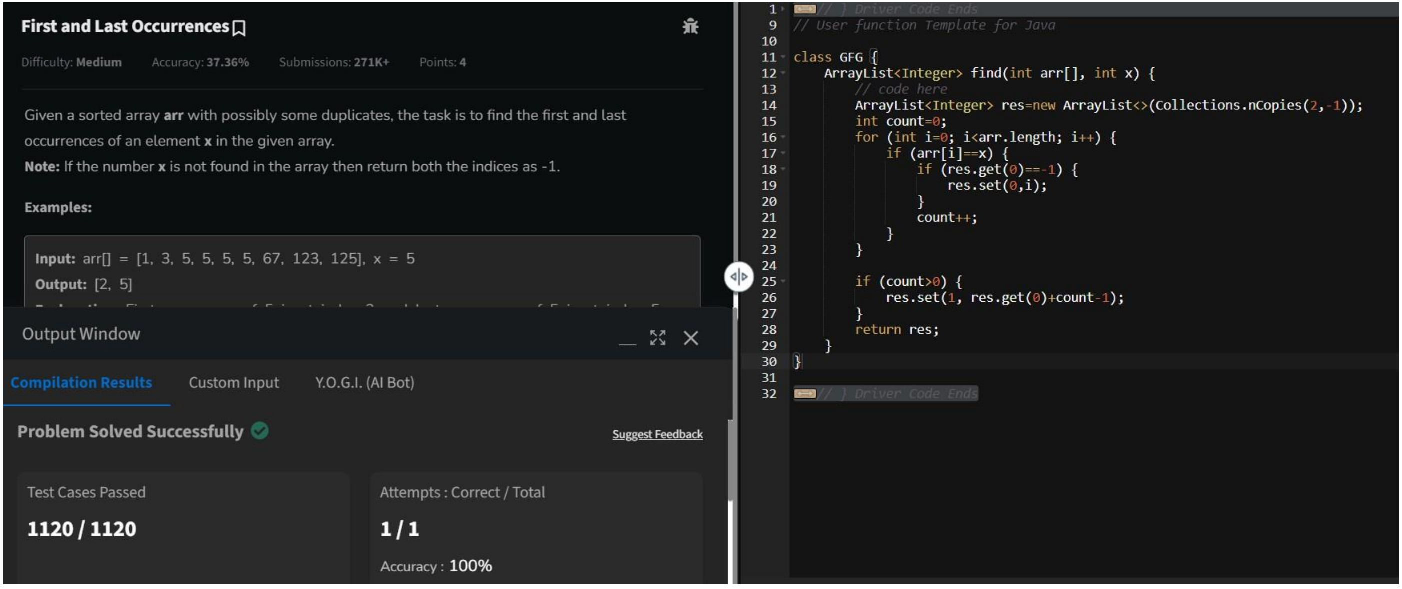
3.First and Last Occurences

Code Solution:

```
class GFG {
    ArrayList<Integer> find(int arr[], int x) {
        // code here
        ArrayList<Integer> res=new ArrayList<>(Collections.nCopies(2,-1));
        int count=0;
        for (int i=0; i<arr.length; i++) {
            if (arr[i]==x) {
                if (res.get(0)==-1) {
                    res.set(0,i);
                }
                count++;
            }
        }

        if (count>0) {
            res.set(1, res.get(0)+count-1);
        }
        return res;
    }
}
```

Output:



First and Last Occurrences

Difficulty: **Medium** Accuracy: **37.36%** Submissions: **271K+** Points: **4**

Given a sorted array **arr** with possibly some duplicates, the task is to find the first and last occurrences of an element **x** in the given array.

Note: If the number **x** is not found in the array then return both the indices as -1.

Examples:

```
Input: arr[] = [1, 3, 5, 5, 5, 5, 67, 123, 125], x = 5
Output: [2, 5]
```

Problem Solved Successfully ✓

Test Cases Passed **1120 / 1120**

Attempts : Correct / Total **1 / 1**

Accuracy : 100%

```
// } Driver Code Ends
// User function Template for Java

class GFG {
    ArrayList<Integer> find(int arr[], int x) {
        // code here
        ArrayList<Integer> res=new ArrayList<>(Collections.nCopies(2,-1));
        int count=0;
        for (int i=0; i<arr.length; i++) {
            if (arr[i]==x) {
                if (res.get(0)==-1) {
                    res.set(0,i);
                }
                count++;
            }
        }

        if (count>0) {
            res.set(1, res.get(0)+count-1);
        }
        return res;
    }
}

// } Driver Code Ends
```

Time complexity: O (n)

Space Complexity: O (1)

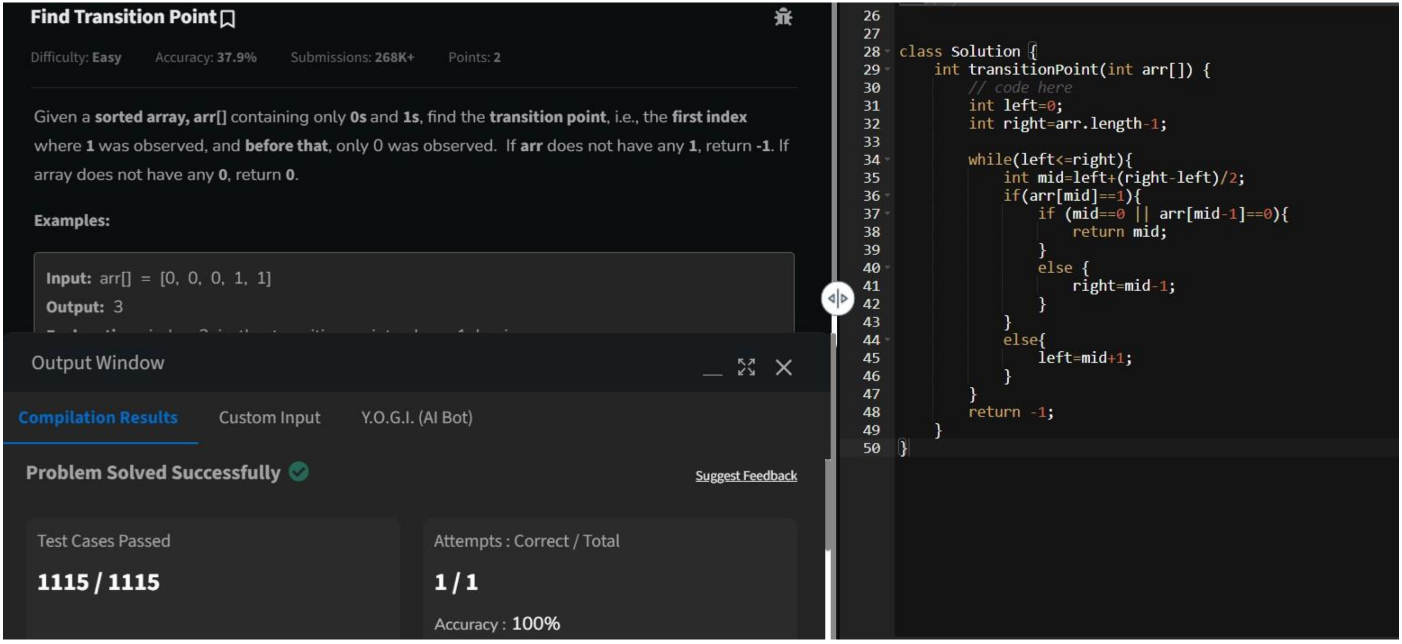
4.Fins Transition Point

Code Solution:

```
class Solution {
    int transitionPoint(int arr[]) {
        // code here
        int left=0;
        int right=arr.length-1;

        while(left<=right){
            int mid=left+(right-left)/2;
            if(arr[mid]==1){
                if (mid==0 || arr[mid-1]==0){
                    return mid;
                }
                else {
                    right=mid-1;
                }
            }
            else{
                left=mid+1;
            }
        }
        return -1;
    }
}
```

Output:



Find Transition Point

Difficulty: **Easy** Accuracy: **37.9%** Submissions: **268K+** Points: **2**

Given a **sorted array, arr[]** containing only **0s** and **1s**, find the **transition point**, i.e., the **first index** where **1** was observed, and **before that**, only 0 was observed. If **arr** does not have any **1**, return **-1**. If array does not have any **0**, return **0**.

Examples:

Input: arr[] = [0, 0, 0, 1, 1]
Output: 3

Output Window

Problem Solved Successfully ✓

Test Cases Passed
1115 / 1115

Attempts : Correct / Total
1 / 1

Accuracy : 100%

```
class Solution {
    int transitionPoint(int arr[]) {
        // code here
        int left=0;
        int right=arr.length-1;

        while(left<=right){
            int mid=left+(right-left)/2;
            if(arr[mid]==1){
                if (mid==0 || arr[mid-1]==0){
                    return mid;
                }
                else {
                    right=mid-1;
                }
            }
            else{
                left=mid+1;
            }
        }
        return -1;
    }
}
```

Time Complexity: O (logn)

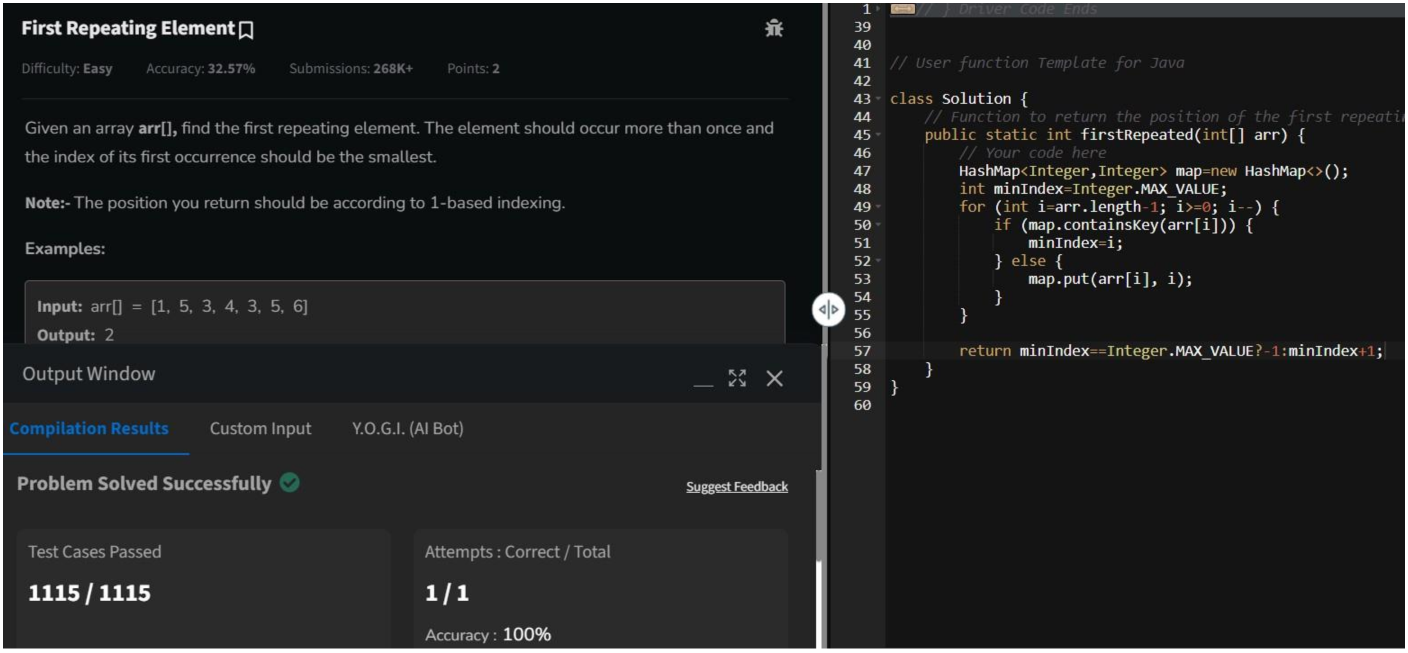
Space Complexity: O (1)

5. First Repeating Element

Code Solution:

```
class Solution {
    // Function to return the position of the first repeating element.
    public static int firstRepeated(int[] arr) {
        // Your code here
        HashMap<Integer,Integer> map=new HashMap<>();
        int minIndex=Integer.MAX_VALUE;
        for (int i=arr.length-1; i>=0; i--) {
            if (map.containsKey(arr[i])) {
                minIndex=i;
            } else {
                map.put(arr[i], i);
            }
        }
        return minIndex==Integer.MAX_VALUE?-1:minIndex+1;
    }
}
```

Output:



First Repeating Element

Difficulty: **Easy** Accuracy: **32.57%** Submissions: **268K+** Points: **2**

Given an array **arr[],** find the first repeating element. The element should occur more than once and the index of its first occurrence should be the smallest.

Note:- The position you return should be according to 1-based indexing.

Examples:

```
Input: arr[] = [1, 5, 3, 4, 3, 5, 6]
Output: 2
```

Output Window

Compilation Results | Custom Input | Y.O.G.I. (AI Bot)

Problem Solved Successfully ✓

Suggest Feedback

Test Cases Passed

1115 / 1115

Attempts : Correct / Total

1 / 1

Accuracy : 100%

```
// } Driver Code Ends


// User function Template for Java

class Solution {
    // Function to return the position of the first repeati
    public static int firstRepeated(int[] arr) {
        // Your code here
        HashMap<Integer,Integer> map=new HashMap<>();
        int minIndex=Integer.MAX_VALUE;
        for (int i=arr.length-1; i>=0; i--) {
            if (map.containsKey(arr[i])) {
                minIndex=i;
            } else {
                map.put(arr[i], i);
            }
        }

        return minIndex==Integer.MAX_VALUE?-1:minIndex+1;
    }
}
```

Time Complexity: O (n)
Space Complexity: O (n)

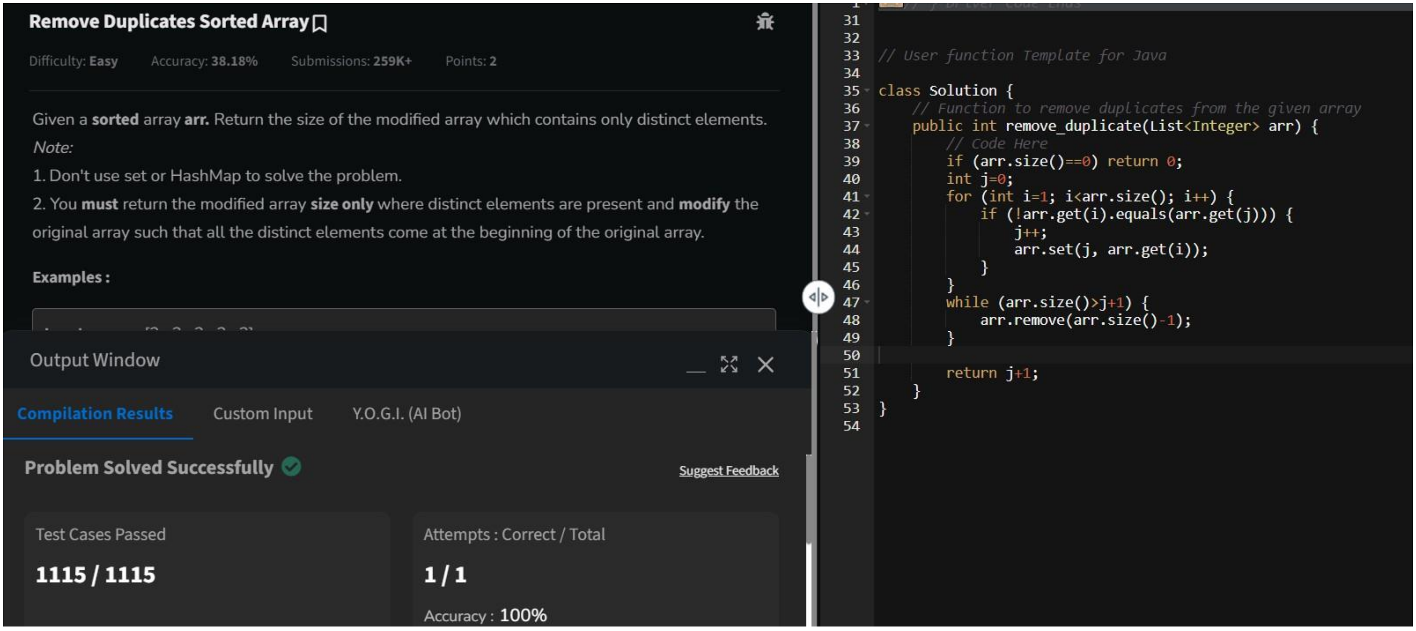
6.Remove Duplicates Sorted Array

Code Solution:

```
class Solution {
    // Function to remove duplicates from the given array
    public int remove_duplicate(List<Integer> arr) {
        // Code Here
        if (arr.size()==0) return 0;
        int j=0;
        for (int i=1; i<arr.size(); i++) {
            if (!arr.get(i).equals(arr.get(j))) {
                j++;
                arr.set(j, arr.get(i));
            }
        }
        while (arr.size()>j+1) {
            arr.remove(arr.size()-1);
        }

        return j+1;
    }
}
```

Output:



Remove Duplicates Sorted Array

Difficulty: **Easy** Accuracy: **38.18%** Submissions: **259K+** Points: **2**

Given a **sorted** array **arr.** Return the size of the modified array which contains only distinct elements.

Note:

1. Don't use set or HashMap to solve the problem.
2. You **must** return the modified array **size only** where distinct elements are present and **modify** the original array such that all the distinct elements come at the beginning of the original array.

Examples :

Output Window

Problem Solved Successfully

Test Cases Passed

1115 / 1115

Attempts : Correct / Total

1 / 1

Accuracy : 100%

```java
// User function Template for Java

class Solution {
    // Function to remove duplicates from the given array
    public int remove_duplicate(List<Integer> arr) {
        // Code Here
        if (arr.size()==0) return 0;
        int j=0;
        for (int i=1; i<arr.size(); i++) {
            if (!arr.get(i).equals(arr.get(j))) {
                j++;
                arr.set(j, arr.get(i));
            }
        }
        while (arr.size()>j+1) {
            arr.remove(arr.size()-1);
        }

        return j+1;
    }
}
```

Time Complexity: O (n)

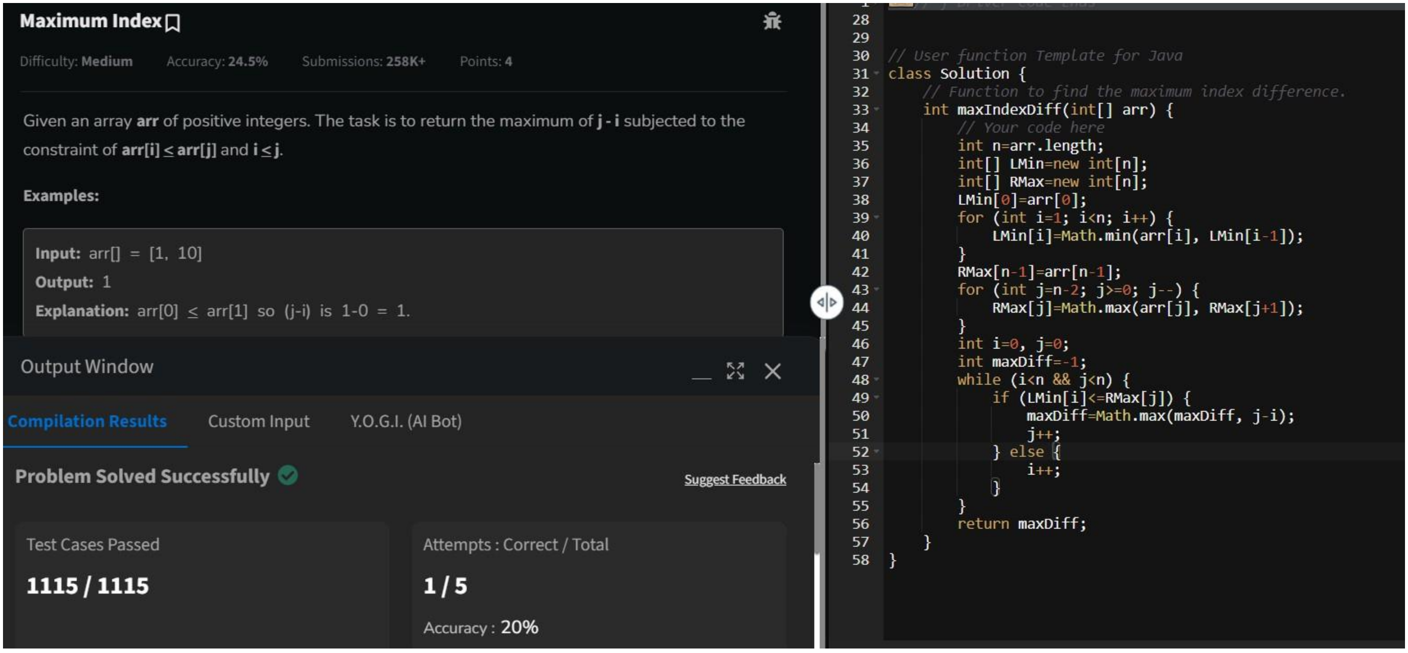
Space Complexity: O (1)

7.Maximum Index

Code Solution

```
class Solution {
    // Function to find the maximum index difference.
    int maxIndexDiff(int[] arr) {
        // Your code here
        int n=arr.length;
        int[] LMin=new int[n];
        int[] RMax=new int[n];
        LMin[0]=arr[0];
        for (int i=1; i<n; i++) {
            LMin[i]=Math.min(arr[i], LMin[i-1]);
        }
        RMax[n-1]=arr[n-1];
        for (int j=n-2; j>=0; j--) {
            RMax[j]=Math.max(arr[j], RMax[j+1]);
        }
        int i=0, j=0;
        int maxDiff=-1;
        while (i<n && j<n) {
            if (LMin[i]<=RMax[j]) {
                maxDiff=Math.max(maxDiff, j-i);
                j++;
            } else {
                i++;
            }
        }
        return maxDiff;
    }
}
```

Output:



Maximum Index

Difficulty: **Medium** Accuracy: **24.5%** Submissions: **258K+** Points: **4**

Given an array **arr** of positive integers. The task is to return the maximum of **j - i** subjected to the constraint of **arr[i] ≤ arr[j]** and **i ≤ j**.

Examples:

```
Input: arr[] = [1, 10]
Output: 1
Explanation: arr[0] ≤ arr[1] so (j-i) is 1-0 = 1.
```

Output Window

Problem Solved Successfully

Test Cases Passed **1115 / 1115**

Attempts : Correct / Total **1 / 5**

Accuracy : **20%**

```


// User function Template for Java
class Solution {
    // Function to find the maximum index difference.
    int maxIndexDiff(int[] arr) {
        // Your code here
        int n=arr.length;
        int[] LMin=new int[n];
        int[] RMax=new int[n];
        LMin[0]=arr[0];
        for (int i=1; i<n; i++) {
            LMin[i]=Math.min(arr[i], LMin[i-1]);
        }
        RMax[n-1]=arr[n-1];
        for (int j=n-2; j>=0; j--) {
            RMax[j]=Math.max(arr[j], RMax[j+1]);
        }
        int i=0, j=0;
        int maxDiff=-1;
        while (i<n && j<n) {
            if (LMin[i]<=RMax[j]) {
                maxDiff=Math.max(maxDiff, j-i);
                j++;
            } else {
                i++;
            }
        }
        return maxDiff;
    }
}
```

Time Complexity: O (n)