Date: 13.11.24

DSA PRACTICE – DAY 4

1. Kth 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.

Solution:

class Solution {

public static int kthSmallest(int[] arr, int k) {

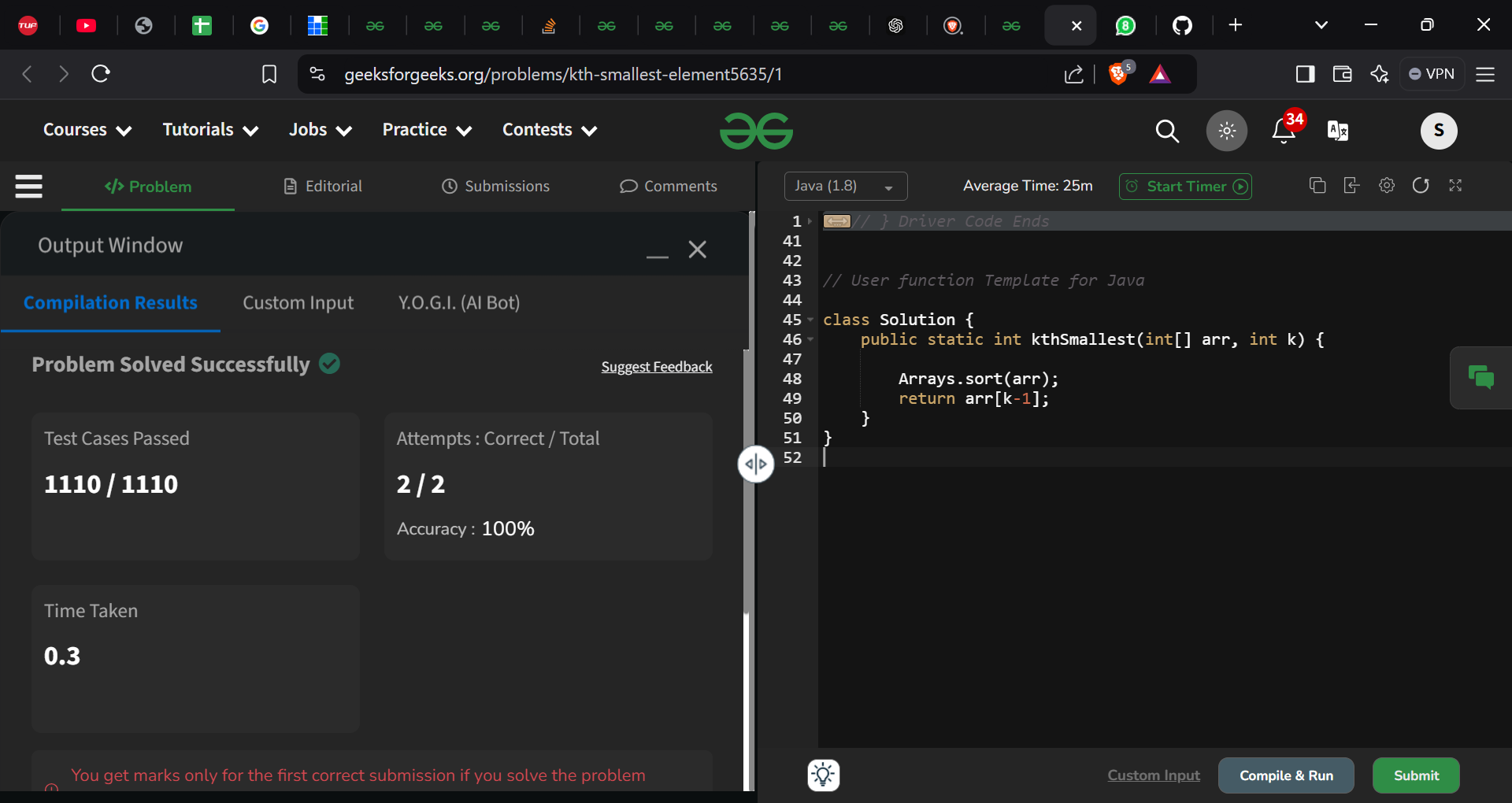
Arrays.sort(arr);

return arr[k-1];

}

}

Output:



Time Complexity: O(n\*log n)

Space Complexity: O(log n)

1. 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.

Solution:

class Solution {

// Function to check if brackets are balanced or not.

static boolean isParenthesisBalanced(String s) {

// code here

Stack<Character> st=new Stack<>();

for(char ch:s.toCharArray()){

if (ch=='(' || ch=='{' || ch=='[') st.push(ch);

else if(!st.empty() && ((ch==')' && st.peek()=='(') || (ch=='}' && st.peek()=='{') || (ch==']' && st.peek()=='['))){

st.pop();

}

else return false;

}

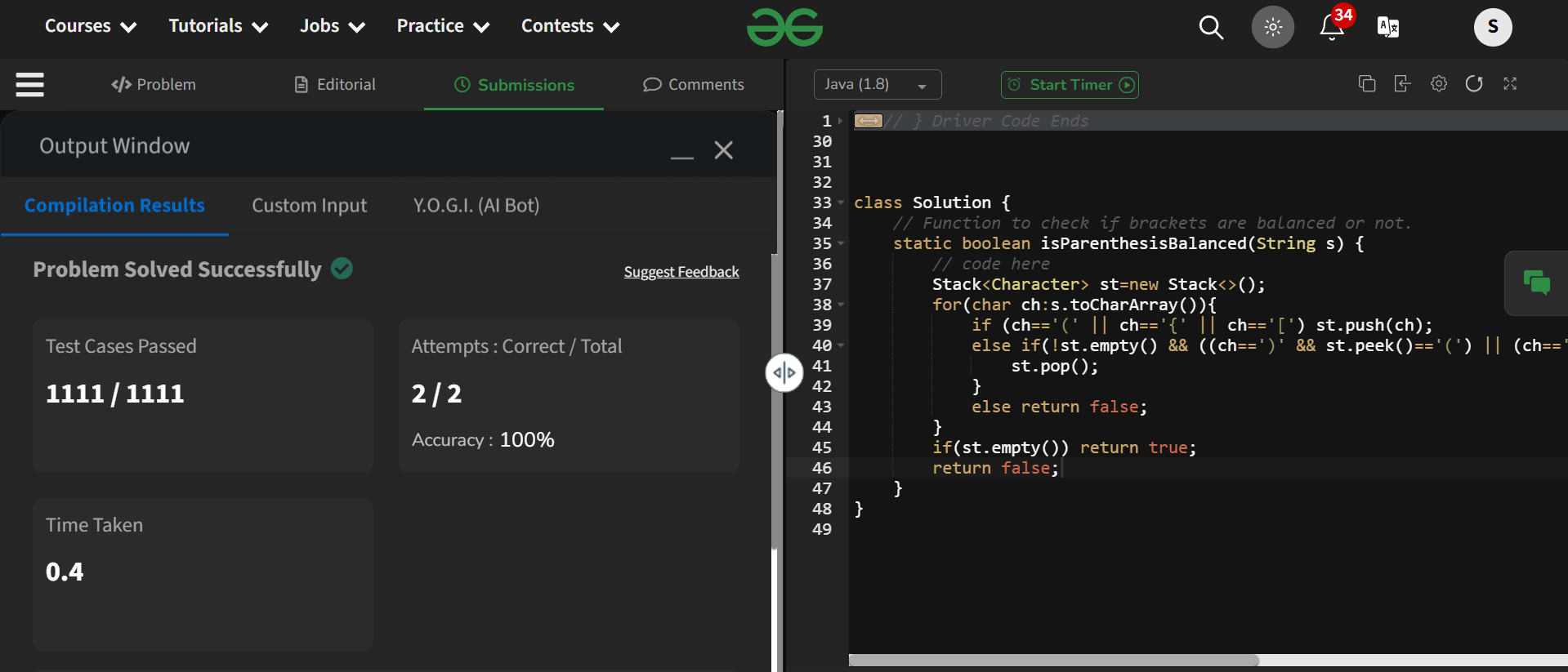
if(st.empty()) return true;

return false;

}

}

Output:



Time Complexity: O(n)

Space Complexity: O( n)

1. 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.

Solution:

class Solution {

// Function to find equilibrium point in the array.

public static int equilibriumPoint(int arr[]) {

// code here

int f=0;

int l=0;

for(int a:arr) l+=a;

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

if (f==l-arr[i]) return i+1;

f+=arr[i];

l-=arr[i];

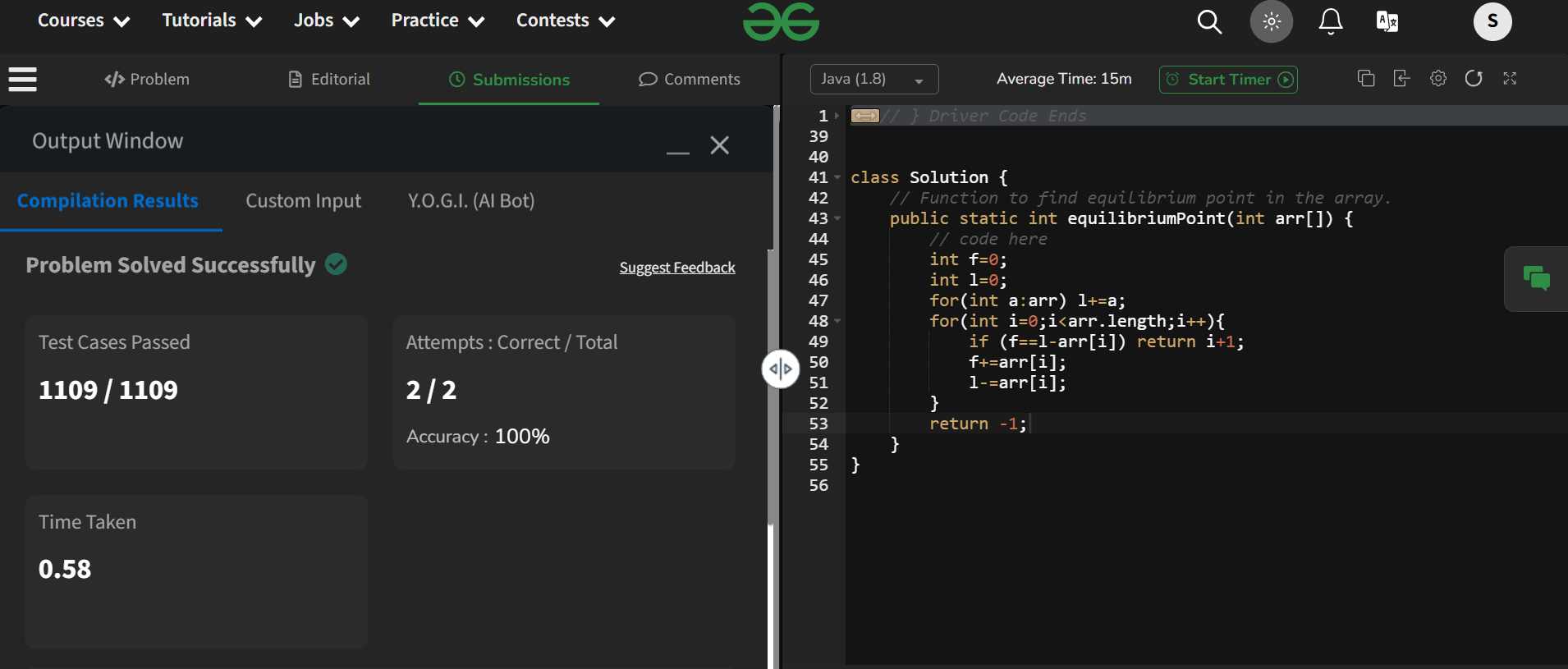
}

return -1;

}

}

Output:



Time Complexity: O(n)

Space Complexity: O(1)

1. 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.

Solution:

class Solution {

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

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

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

for(int i=arr.length-1;i>=0;i--){

while(!st.empty() && arr[i]>=st.peek()){

st.pop();

}

if (st.empty()) nge.add(-1);

else nge.add(st.peek());

st.push(arr[i]);

}

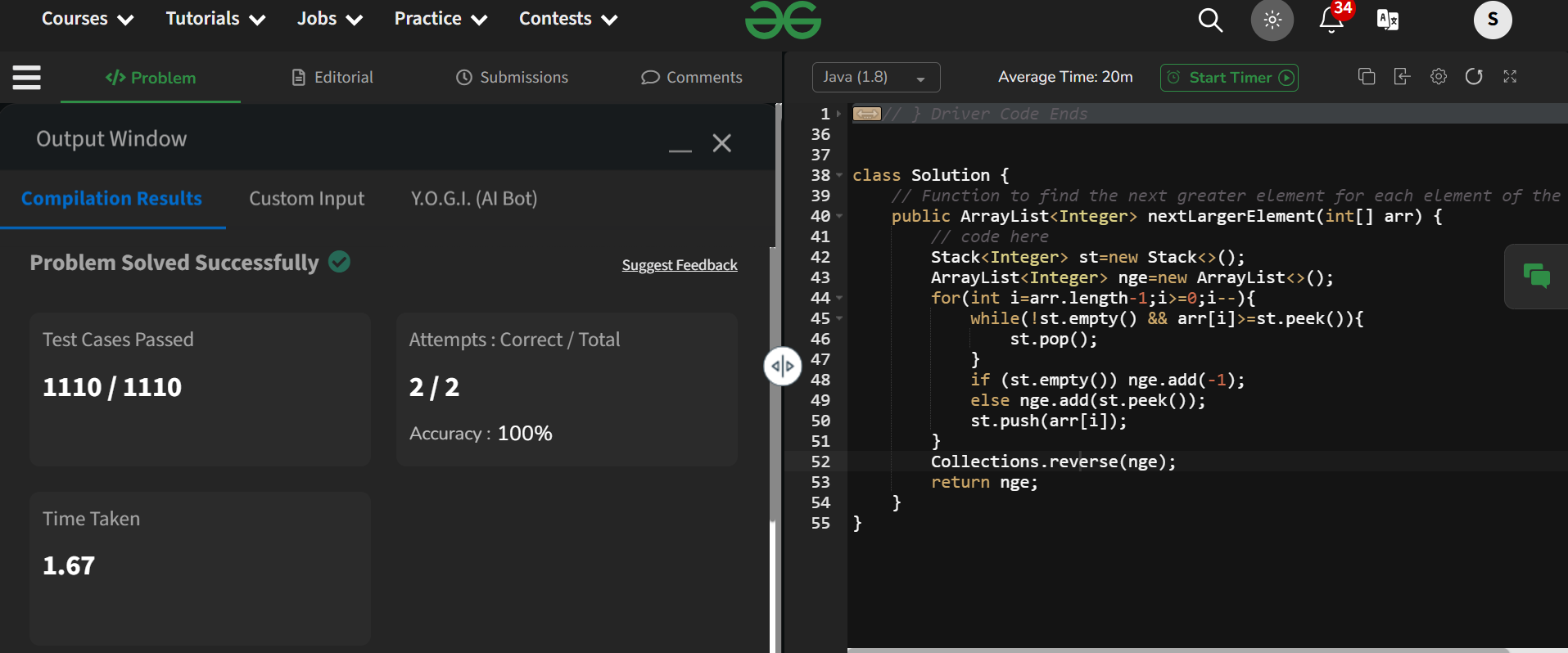
Collections.reverse(nge);

return nge;

}

}

Output:



Time Complexity: O(n)

Space Complexity: O(n)

1. Binary search

Solution:

import java.util.Scanner;

public class BinarySearch {

    public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);

        System.out.print("Enter the size of array: ");

        int n=sc.nextInt();

        System.out.println("Enter the array elements: ");

        int[] arr=new int[n];

        for(int i=0;i<n;i++) arr[i]=sc.nextInt();

        System.out.print("Enter the key to search: ");

        int key=sc.nextInt();

        sc.close();

        int start=0;

        int end=n-1;

        int middle;

        int ans=-1;

        while(start<=end){

            middle=(start+end)/2;

            if (arr[middle]==key){

                ans=middle;

                break;

            }

            else if (arr[start]<=arr[middle]){

                if(arr[start]<=key && arr[middle]>key) end=middle-1;

                else start=middle+1;

            }

            else{

                if (arr[middle+1]<=key && arr[end]>=key) start=middle+1;

                else end=middle-1;

            }

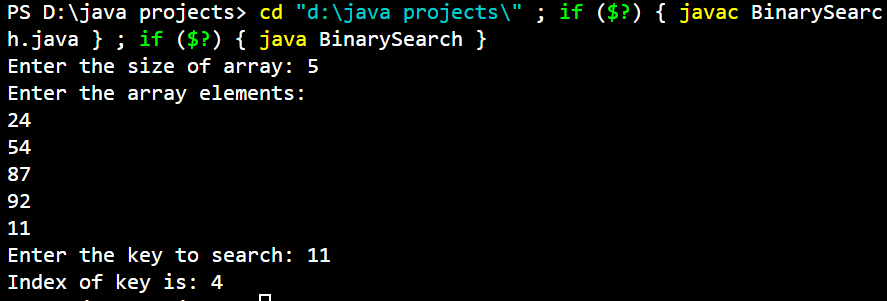
        }

        System.out.println("Index of key is: "+ans);

    }

}

Output:



Time Complexity: O(log n)

Space Complexity: O(1)

1. Union of two sorted arrays with duplicates

Given two **sorted** arrays **a[]** and **b[]**, where each array may contain **duplicate** elements , the task is to return the elements in the **union** of the two arrays in **sorted** order.

Solution:

class Solution {

public static ArrayList<Integer> findUnion(int a[], int b[]) {

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

int i=0,j=0;

while(i<a.length && j<b.length){

if (a[i]<b[j]){

if ((arr.isEmpty()) || (!arr.isEmpty() && arr.get(arr.size()-1)<a[i])){

arr.add(a[i]);

}

i++;

}else{

if ((arr.isEmpty()) || (!arr.isEmpty() && arr.get(arr.size()-1)<b[j])){

arr.add(b[j]);

}

j++;

}

}

while(i<a.length){

if ((arr.isEmpty()) || (!arr.isEmpty() && arr.get(arr.size()-1)<a[i])){

arr.add(a[i]);

}

i++;

}

while(j<b.length){

if ((arr.isEmpty()) || (!arr.isEmpty() && arr.get(arr.size()-1)<b[j])){

arr.add(b[j]);

}

j++;

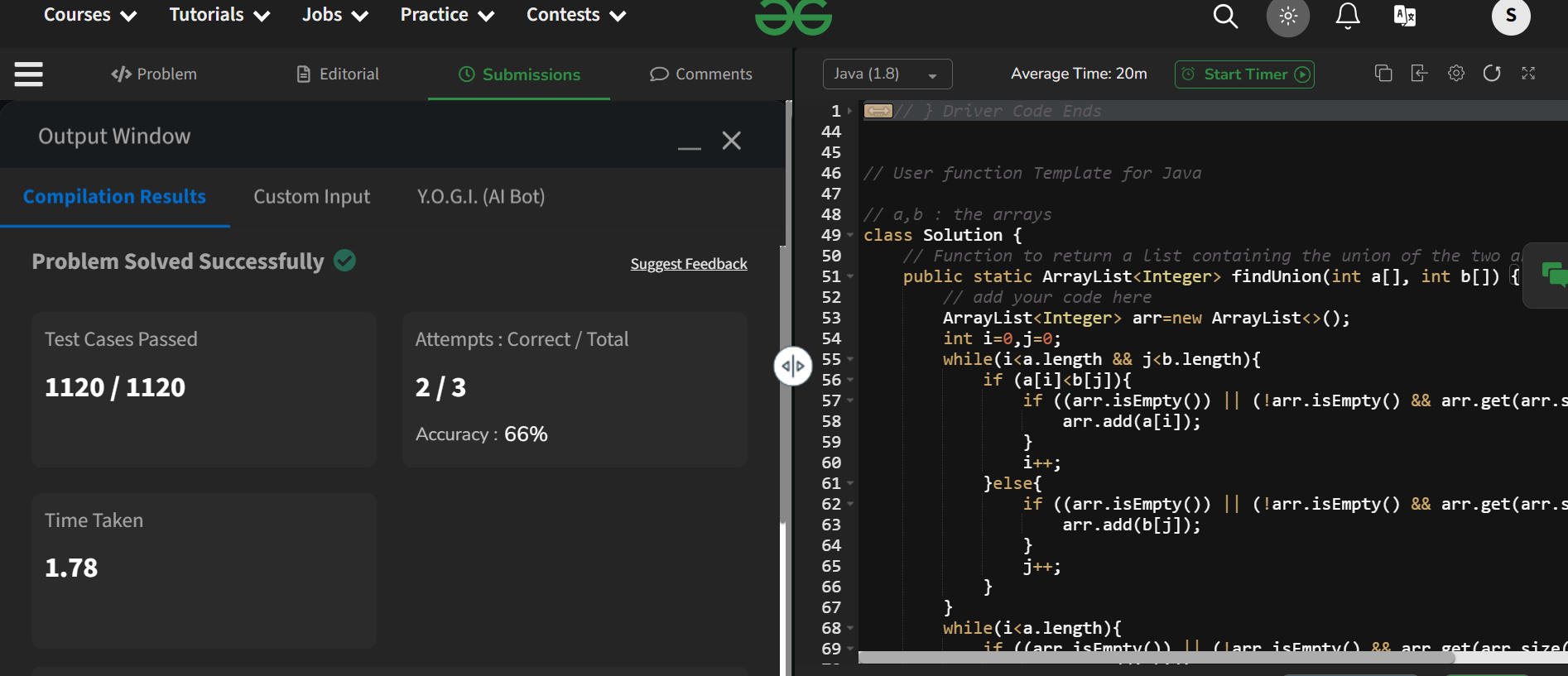
}

return arr;

}

}

Output:



Time Complexity: O(m+n)

Space Complexity: O(m+n)

1. 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.

Solution:

import java.util.Arrays;

public class MinimiseHeights2 {

    public static void main(String[] args) {

        int[] arr1=new int[]{3,9,12,16,20};

        int k1=3;

        System.out.println("The array is: ");

        for(int i=0;i<arr1.length;i++) System.out.print(arr1[i]+" ");

        System.out.println();

        System.out.println("The minimum difference in heights is "+MinHeight(arr1,k1));

        int[] arr2=new int[]{1,1,4,6,7,8,9,10};

        int k2=7;

        System.out.println("The array is: ");

        for(int i=0;i<arr2.length;i++) System.out.print(arr2[i]+" ");

        System.out.println();

        System.out.println("The minimum difference in heights is "+MinHeight(arr2,k2));

    }

    static int MinHeight(int[] arr,int k){

        int n=arr.length;

        if(n==1) return 0;

        Arrays.sort(arr);

        int min=arr[0]+k;

        int max=arr[n-1]-k;

        int diff=arr[n-1]-arr[0];

        int min1,max1;

        for(int i=0;i<n-1;i++){

            min1=Math.min(arr[i+1]-k,min);

            max1=Math.max(arr[0]+k,max);

            if(min1<0) continue;

            diff=Math.min(diff,max1-min1);

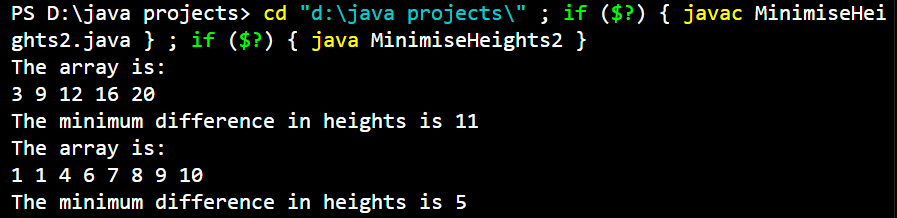
        }

        return diff;

    }

}

Output:



Time Complexity: O(n)

Space Complexity: O(1)