**DSA PRACTICE**

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

class Solution {

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

int maximum=arr[0];

for(int i:arr){

maximum=Math.max(i,maximum);

}

int[] count=new int[maximum+1];

for (int i:arr){

count[i]+=1;

}

int freq=0;

for (int i=0; i<=maximum; i++){

if (count[i]!=0){

freq+=count[i];

if (freq>=k){

return i;}

}

}

return -1;

}



**Time complexity: O (m+n)**

**Space Complexity: O (max\_element)**

**2. 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.**

**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.**

class Solution {

int getMinDiff(int[] arr, int k) {

// code here

Arrays.sort(arr);

int n=arr.length;

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

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

int small=Math.min(arr[0]+k, arr[i+1]-k);

int large=Math.max(arr[i]+k, arr[n-1]-k);

if (small<0) continue;

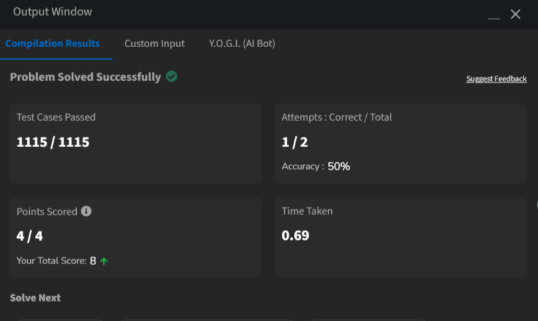
res=Math.min(res, large-small);

}

return res;

}

}



**Time Complexity: O (nlogn)**

**Space Complexity: O (n)**

**3. Given a string s, composed of different combinations of '(' , ')', '{', '}', '[', ']', verify the validity of the arrangement.**

**An input string is valid if:**

**1. Open brackets must be closed by the same type of brackets.**

**2. Open brackets must be closed in the correct order.**

class Solution{

boolean valid(String s){

// code here

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

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

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

st.push(i);

}

else{

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

st.pop();

}else{

return false;

}

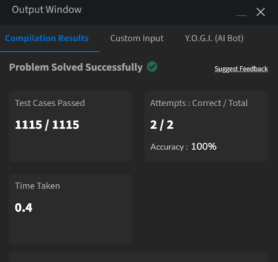
}

}

return true;

}

}



**Time complexity: O (n)**

**Space Complexity: O (n)**

**4. Equilibrium Point**

class Solution {

// Function to find equilibrium point in the array.

public static int equilibriumPoint(int arr[]) {

// code here

int left=0;

int right=arr.length-1;

long leftSum=0;

long rightSum=0;

while (left<right) {

if (leftSum<rightSum) {

leftSum+=arr[left];

left++;

} else {

rightSum+=arr[right];

right--;

}

}

if (leftSum==rightSum) {

return left+1;

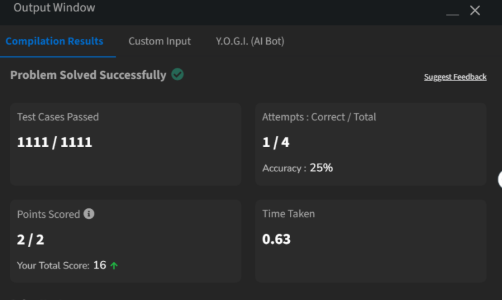
} else {

return -1;

}

}

}



**Time Complexity: O (n)**

**Space Complexity: O (1)**

**5. Binary Search**

class Solution {

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

// Code Here

int low=0;

int high=arr.length-1;

while(low<=high){

int mid=low+(high-low)/2;

if (arr[mid]==k){

return mid;

}

else if (arr[mid]>k){

high=mid-1;

}

else{

low=mid+1;

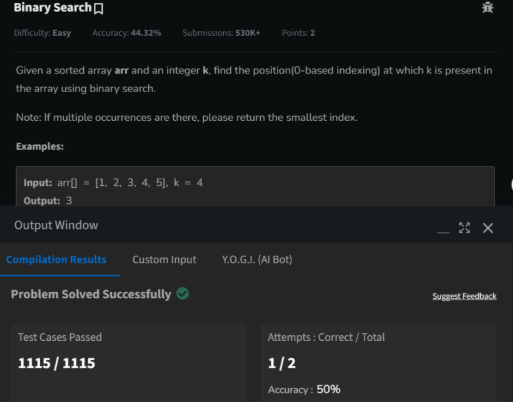
}

}

return -1;

}

}



**Time Complexity: O(logn)**

**Space Complexity: O (1)**

**6. Next Greater Element**

class Solution {

// Function to find the next greater element for each element of the array.

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

// code here

int n=arr.length;

ArrayList<Integer> result = new ArrayList<>(n);

for (int i=0; i<n; i++) {

result.add(-1);

}

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

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

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

stack.pop();

}

if (!stack.isEmpty()) {

result.set(i, stack.peek());

}

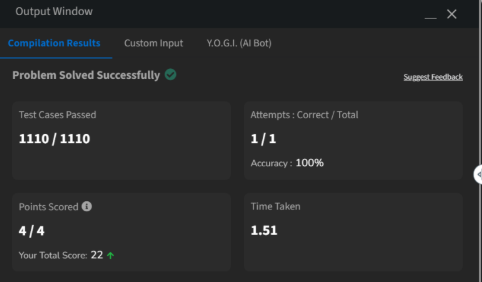
stack.push(arr[i]);

}

return result;

}

}

****

**Time Complexity: O (n)**

**Space Complexity: O (n)**

**7. Union of 2 Arrays with Duplicate elements**

class Solution {

public static int findUnion(int a[], int b[]) {

// code here

HashSet<Integer> set=new HashSet<>();

for(int i:a){

set.add(i);

}

for(int i:b){

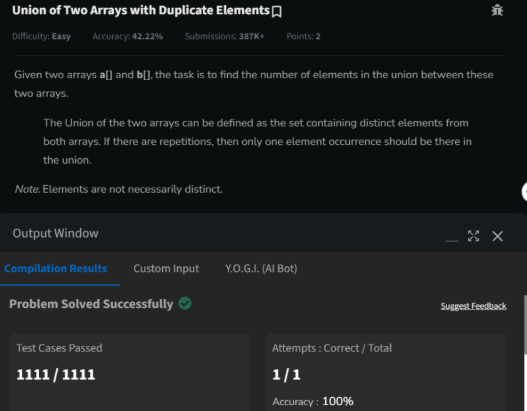
set.add(i);

}

return set.size();

}

}



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

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