CODING PRACTICE-4(13.11.24)

1.Kth Smallest:

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^{th} 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

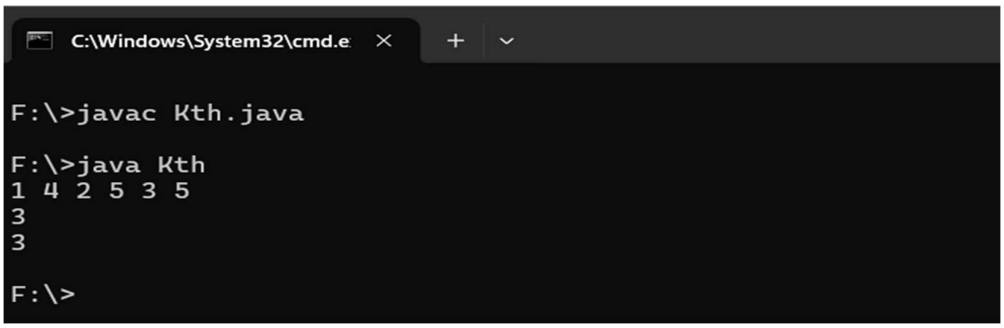
Explanation: 4th smallest element in the given array is 15.

Solution:

```
import java.io.*;
import java.util.*;
class Kth{
   public static void main(String args[]) throws IOException {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      PrintWriter out = new PrintWriter(System.out);
      int t = Integer.parseInt(in.readLine().trim());
      while (t-- > 0) {
         String line = in.readLine();
         String[] tokens = line.split(" ");
         ArrayList<Integer> array = new ArrayList<>();
         for (String token : tokens) {
            array.add(Integer.parseInt(token));
         }
        int[] arr = new int[array.size()];
         int idx = 0;
```

```
            for (int i : array) arr[idx++] = i;


            int key = Integer.parseInt(in.readLine().trim());
            Solution ob = new Solution();
            out.println(ob.kthSmallest(arr, key));
            out.println("~");
        }
        out.flush();
    }
}
class Solution {
    public static int kthSmallest(int[] arr, int k) {
        // Your code here
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);
        for (int num : arr) {
            maxHeap.add(num);
            if (maxHeap.size() > k) {
                maxHeap.poll();
            }
        }
        return maxHeap.peek();
    }
}
```

Output:



Time complexity:O(n)

Space Complexity:O(1)

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.
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.

Solution:

```
import java.io.*;

import java.util.*;

public class Height{

   public static void main(String[] args) throws Exception {

      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

      int tc = Integer.parseInt(br.readLine().trim());

      while (tc-- > 0) {

         String[] inputLine;
```

```
            inputLine = br.readLine().trim().split(" ");
            int k = Integer.parseInt(inputLine[0]);
            inputLine = br.readLine().trim().split(" ");
            if (inputLine == null || inputLine.length == 0) {
                System.out.println("Invalid input");
                continue;
            }
            int[] arr = new int[inputLine.length];
            for (int i = 0; i < inputLine.length; i++) {
                arr[i] = Integer.parseInt(inputLine[i]);
            }
            int ans = new Solution().getMinDiff(arr, k);
            System.out.println(ans);
        }
    }
}
class Solution {
    int getMinDiff(int[] arr, int k) {
        int n =arr.length;
        if( n == 1){
            return 0;
        }
        Arrays.sort(arr);
        int ans = arr[n-1] - arr[0];
        int min = arr[0] + k;
        int max = arr[n-1] - k;
        for(int i=0; i<n-1; i++){
            int min_ele = Math.min(min, arr[i+1]-k);
            int max_ele = Math.max(max, arr[i]+k);
```

```
            if(min_ele < 0)
            continue;
          ans = Math.min(ans, max_ele - min_ele);
      }
      return ans;
   }
}
```

Output:

```
F:\>javac Height.java

F:\>java Height
1
2
1 5 8 10
5

F:\>
```

Time complexity:O(n log n)

Space Complexity:O(1)

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.

Solution:

```
import java.io.*;

import java.lang.*;

import java.util.*;

class Main {

   public static void main(String args[]) {

      Scanner sc = new Scanner(System.in);

      int t = sc.nextInt();

      while (t-- > 0) {

         String st = sc.next();

         if (new Solution().isParenthesisBalanced(st) == true)

            System.out.println("true");

         else

            System.out.println("false");

         System.out.println("~");

      }

   }

}

class Solution {

   static boolean isParenthesisBalanced(String s) {

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

```
                for(int i=0;i<s.length();i++){
                        char ch = s.charAt(i);
                        if(ch == '(' || ch == '{' || ch == '['){
                                st.push(ch);
                        }
                        else if(ch == ')' || ch == '}' || ch == ']'){
                                if(st.isEmpty()){
                                        return false;
                                }
                                char top = st.pop();
            if ((ch == '}' && top != '{') ||
               (ch == ')' && top != '(') ||
               (ch == ']' && top != '[')) {
               return false;
            }
                        }
                }
                return st.isEmpty();
    }
}
```

Output:

Time complexity:O(n)

Space Complexity:O(n)

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 before that index is the same as the sum of elements after it.

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.

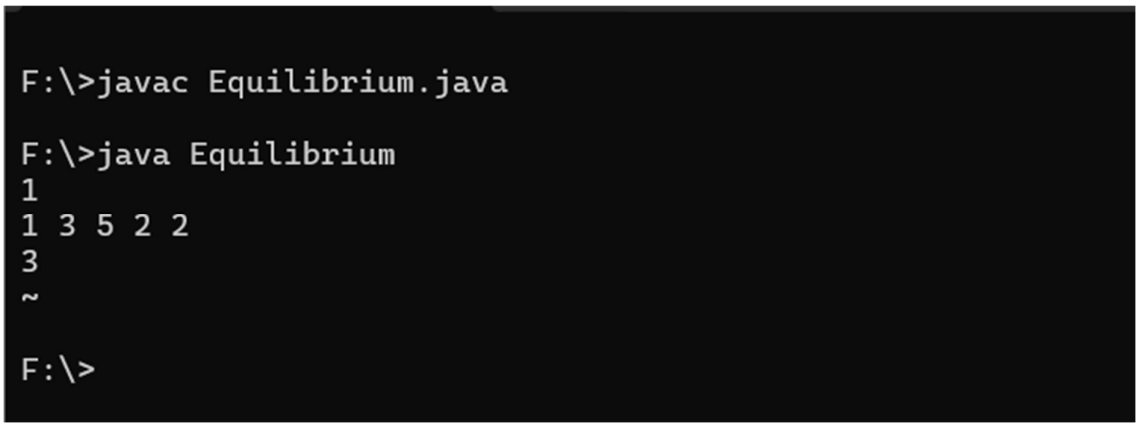
Solution:

```
import java.io.*;

import java.util.*;

class Equilibrium {

   public static void main(String[] args) throws IOException {

      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

      int t = Integer.parseInt(br.readLine().trim());

      while (t-- > 0) {

         String line = br.readLine();

         String[] tokens = line.split(" ");

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

         for (String token : tokens) {

            array.add(Integer.parseInt(

               token));

         }
```

```
            int[] arr = new int[array.size()];
            int idx = 0;
            for (int i : array) arr[idx++] = i;
            Solution obj = new Solution();
            System.out.println(obj.equilibriumPoint(arr));
            System.out.println("~");
        }
    }
}
class Solution {
    public static int equilibriumPoint(int arr[]) {
        int n=arr.length;
        if(n==1){
            return 1;
        }
        long [] prefix = new long[n];
        long []suffix = new long[n];
        prefix[0]=arr[0];
        for(int i=1;i<n;i++){
          prefix[i]=prefix[i-1]+arr[i];
        }
        suffix[n-1]= arr[n-1];
        for(int i=n-2;i>=0;i--){
            suffix[i]=suffix[i+1]+arr[i];
        }
        for(int i=0;i<n;i++){
            if(prefix[i]==suffix[i]){
                return i+1;
```

```
        }
      }
      return -1;
   }
}
```

Output:



TimeComplexity:O(n)

SpaceComplexity:O(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.

Solution:

```
import java.io.*;
import java.util.*;
public class Binary{
   public static void main(String[] args) throws IOException {
      Scanner sc = new Scanner(System.in);
      int t = sc.nextInt();
      sc.nextLine();
      while (t-- > 0) {
         int k = sc.nextInt();
         sc.nextLine();
         String input = sc.nextLine();
         String[] strNumbers = input.split(" ");
         int[] arr = new int[strNumbers.length];
         for (int i = 0; i < strNumbers.length; i++) {
            arr[i] = Integer.parseInt(strNumbers[i]);
         }
         Solution ob = new Solution();
         int res = ob.binarysearch(arr, k);
         System.out.println(res);
System.out.println("~");
}
      sc.close();
   }
}
class Solution {
   public int binarysearch(int[] arr, int k) {
      int l=0;
      int r = arr.length-1;
      while(l<=r){
```

```
        int mid=(l+r)/2;
        if(arr[mid]==k){
            return mid;
        }
        else if(arr[mid]<k){
            l=mid+1;
        }
        else if(arr[mid]>k){
            r=mid-1;
        }
      }
      return -1;
   }
}
```

Output:

Time Complexity:O(log n)

SpaceComplexity:O(1)

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.

Solution:

```
import java.util.Stack;


public class Nge {


   public static void printNextGreaterElements(int[] arr) {
      int n = arr.length;
      int[] nge = new int[n];
      Stack<Integer> stack = new Stack<>();
      for (int i = n - 1; i >= 0; i--) {
         int current = arr[i];
         while (!stack.isEmpty() && stack.peek() <= current) {
```

```
            stack.pop();
        }
        if (stack.isEmpty()) {
            nge[i] = -1;
        } else {
            nge[i] = stack.peek();
        }
        stack.push(current);
    }
    for (int i = 0; i < n; i++) {
        System.out.println(arr[i] + " --> " + nge[i]);
    }
  }

  public static void main(String[] args) {
    int[] arr = {4, 5, 2, 25};
    printNextGreaterElements(arr);
  }
}
```

Output:

Time Complexity:O(n)

SpaceComplexity:O(n)

7. Union of two arrays with duplicate elements:

Given two arrays **a[]** and **b[]**, the task is to find the number of elements in the union between these two arrays.

The Union of the two arrays can be defined as the set containing distinct elements from both arrays. If there are repetitions, then only one element occurrence should be there in the union.

Note: Elements are not necessarily distinct.

Examples

Input: a[] = [1, 2, 3, 4, 5], b[] = [1, 2, 3]

Output: 5

Explanation: 1, 2, 3, 4 and 5 are the elements which comes in the union setof both arrays. So count is 5.

Input: a[] = [85, 25, 1, 32, 54, 6], b[] = [85, 2]
Output: 7

Explanation: 85, 25, 1, 32, 54, 6, and 2 are the elements which comes in the union set of both arrays. So count is 7.

Input: a[] = [1, 2, 1, 1, 2], b[] = [2, 2, 1, 2, 1]
Output: 2

Explanation: We need to consider only distinct. So count is 2.

Solution:

```
import java.io.*;

import java.lang.*;

import java.util.*;

class Union {

   public static void main(String args[]) throws IOException {

      BufferedReader read = new BufferedReader(new InputStreamReader(System.in));

      int t = Integer.parseInt(read.readLine().trim());

      while (t-- > 0) {

         String line1 = read.readLine().trim();

         String[] numsStr1 = line1.split(" ");
```

```
            int[] a = new int[numsStr1.length];
            for (int i = 0; i < numsStr1.length; i++) {
                a[i] = Integer.parseInt(numsStr1[i]);
            }
            String line2 = read.readLine().trim();
            String[] numsStr2 = line2.split(" ");
            int[] b = new int[numsStr2.length];
            for (int i = 0; i < numsStr2.length; i++) {
                b[i] = Integer.parseInt(numsStr2[i]);
            }
            Solution ob = new Solution();
            System.out.println(ob.findUnion(a, b));
            System.out.println("~");
        }
    }
}
class Solution {
    public static int findUnion(int a[], int b[]) {
        Set<Integer> set = new HashSet<Integer>();
        int max = Math.max(a.length,b.length);
        for(int i=0;i<max;i++){
            if(i<a.length)
                set.add(a[i]);
            if(i<b.length)
                set.add(b[i]);
        }
    return set.size();
}
}
```

Output:

Time Complexity:O(m+n)

Space Complexity: O(m+n)