DSA Coding Practice-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 $\mathbf{k^{th}}$ **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.

Code:

```
class KthSmall {
    public static int findKthSmallest(int[] arr, int k) {
        return quickSelect(arr, 0, arr.length - 1, k - 1);
    }

    private static int quickSelect(int[] arr, int left, int right, int k) {
        if (left == right) {
            return arr[left];
        }

        int pivotIndex = partition(arr, left, right);
        if (k == pivotIndex) {
            return arr[k];
        } else if (k < pivotIndex) {
```

```
            return quickSelect(arr, left, pivotIndex - 1, k);
        } else {
            return quickSelect(arr, pivotIndex + 1, right, k);
        }
    }
    private static int partition(int[] arr, int left, int right) {
        int pivot = arr[right];
        int i = left;
        for (int j = left; j < right; j++) {
            if (arr[j] < pivot) {
                swap(arr, i, j);
                i++;
            }
        }
        swap(arr, i, right);
        return i;
    }
    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] arr1 = {7, 10, 4, 3, 20, 15};
        int k1 = 3;
        System.out.println("3rd smallest element is " + findKthSmallest(arr1, k1));

        int[] arr2 = {2, 3, 1, 20, 15};
        int k2 = 4;
```

```
        System.out.println("4th smallest element is " + findKthSmallest(arr2, k2));
    }
}
```

Output:

3rd smallest element is 7

4th smallest element is 15

Time Complexity: O(n)

2. Minimize the height

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.

Code:

```
import java.util.Arrays;


class MinHeight {
    public static int getMinDifference(int[] arr, int k) {
        int n = arr.length;
```

```
        if (n == 1) return 0;
        Arrays.sort(arr);
        int diff = arr[n - 1] - arr[0];
        int minHeight = arr[0] + k;
        int maxHeight = arr[n - 1] - k;
        for (int i = 0; i < n - 1; i++) {
            int currentMin = Math.min(minHeight, arr[i + 1] - k);
            int currentMax = Math.max(maxHeight, arr[i] + k);
            diff = Math.min(diff, currentMax - currentMin);
        }
        return diff;
    }
    public static void main(String[] args) {
        int[] arr1 = {1, 5, 8, 10};
        int k1 = 2;
        System.out.println("Minimum difference is " + getMinDifference(arr1, k1));
        int[] arr2 = {3, 9, 12, 16, 20};
        int k2 = 3;
        System.out.println("Minimum difference is " + getMinDifference(arr2, k2));
    }
}
```

Output:

Minimum difference is 5

Minimum difference is 11

Time Complexity: O(nlogn)

3. Parantheses 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.

Code:

```
import java.util.Stack;


class Parantheses {
    public static boolean isBalanced(String s) {
        Stack<Character> stack = new Stack<>();


        for (char ch : s.toCharArray()) {
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            } else if (ch == ')' && !stack.isEmpty() && stack.peek() == '(') {
                stack.pop();
            } else if (ch == '}' && !stack.isEmpty() && stack.peek() == '{') {
                stack.pop();
            } else if (ch == ']' && !stack.isEmpty() && stack.peek() == '[') {
                stack.pop();
            } else {
                return false;
```

```
            }
        }
        return stack.isEmpty();
    }
    public static void main(String[] args) {
        String s1 = "{([])}";
        System.out.println(isBalanced(s1));
        String s2 = "()";
        System.out.println(isBalanced(s2));
        String s3 = "([]";
        System.out.println(isBalanced(s3));
    }
}
```

Output:

true

true

false

Time 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.

Code:

```
class Equilibrium {
    public static int findEquilibriumPoint(int[] arr) {
        int totalSum = 0;
        for (int num : arr) {
            totalSum += num;
        }
        int leftSum = 0;
        for (int i = 0; i < arr.length; i++) {
            totalSum -= arr[i];
            if (leftSum == totalSum) {
                return i + 1;
            }
            leftSum += arr[i];
        }
        return -1;
    }
    public static void main(String[] args) {
        int[] arr1 = {1, 3, 5, 2, 2};
        System.out.println(findEquilibriumPoint(arr1));
    }
}
```

Output:3

Time Complexity:O(n)

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.

Code:

```
class BinarySearch {
    int binarySearch(int arr[], int low, int high, int x) {
        if (high >= low) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == x)
                return mid;
            if (arr[mid] > x)
                return binarySearch(arr, low, mid - 1, x);
            return binarySearch(arr, mid + 1, high, x);
        }
        return -1;
    }

    public static void main(String args[]) {
        BinarySearch ob = new BinarySearch();
        int arr[] = { 2, 3, 4, 10, 40 };
```

```
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr, 0, n - 1, x);
        if (result == -1)
            System.out.println("Element is not present in array");
        else
            System.out.println("Element is present at index " + result);
    }
}
```

Output: Element is present at index 3

Time Complexity: O(logn)

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.

Code:

```
import java.util.*;
```

```
class NextGreaterElement {
    public static int[] findNextGreaterElements(int[] arr) {
        int n = arr.length;
        int[] result = new int[n];
        Stack<Integer> stack = new Stack<>();
        for (int i = n - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= arr[i]) {
                stack.pop();
            }
            result[i] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(arr[i]);
        }
        return result;
    }
    public static void main(String[] args) {
        int[] arr1 = {1, 3, 2, 4};
        System.out.println(Arrays.toString(findNextGreaterElements(arr1))); // Output: [3,
4, 4, -1]
    }
}
```

Output:

[3, 4, 4, -1]

Time Complexity:O(n)

6. 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.

Code:

```
import java.util.*;
class Union {
    public static int findUnionCount(int[] a, int[] b) {
        List<Integer> unionList = new ArrayList<>();
        for (int i = 0; i < a.length; i++) {
            if (!unionList.contains(a[i])) {
                unionList.add(a[i]);
            }
        }
        for (int i = 0; i < b.length; i++) {
            if (!unionList.contains(b[i])) {
                unionList.add(b[i]);
            }
        }
```

```
        return unionList.size();
    }
    public static void main(String[] args) {
        int[] a1 = {1, 2, 3, 4, 5};
        int[] b1 = {1, 2, 3};
        System.out.println(findUnionCount(a1, b1));
    }
}
```

Output:5

Time Complexity: $O(n^2+m^2)$