CODING PRACTICE PROBLEMS

(Sowmya A) **DATE: 13/11/2024**

1. K-th 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.

Follow up: Don't solve it using the inbuilt sort function.

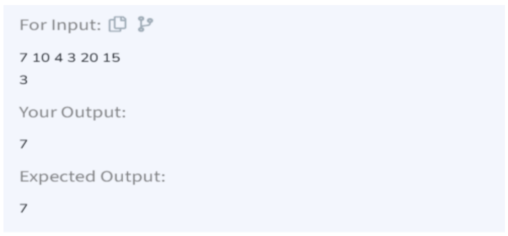
Input: arr[] = [7, 10, 4, 3, 20, 15], k = 3

Output: 7

Explanation: 3rd smallest element in the given array is 7

CODE:

```
class Solution {
    public static int kthSmallest(int[] arr, int k) {
        int a=arr[0];
        int temp;
        for(int i=0;i<arr.length-1;i++){
            for(int j=i+1;j<arr.length;j++){
                if(arr[i]>=arr[j]){
                    temp=arr[i];
                    arr[i]=arr[j];
                    arr[j]=temp;
                }
            }
        }
        return arr[k-1];
    }
}
```


For Input:

```
7 10 4 3 20 15
3
```

Your Output:

```
7
```

Expected Output:

```
7
```

Time Complexity:O(n^2)

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**.

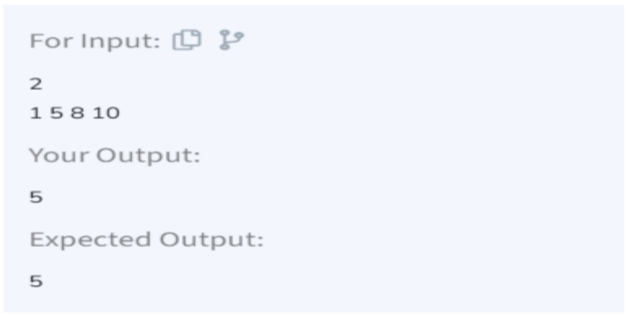
Input: k = 2, arr[] = {1, 5, 8, 10}

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:

```
class Solution {
   int getMinDiff(int[] arr, int k) {
      int n = arr.length;
      if (n == 1) {
         return 0;
      }
      Arrays.sort(arr);
      int initialDifference = arr[n - 1] - arr[0];
      int minHeight, maxHeight;
      int minDifference = initialDifference;
      for (int i = 1; i < n; i++) {
            if (arr[i] >= k) {
                minHeight = Math.min(arr[0] + k, arr[i] - k);
                maxHeight = Math.max(arr[n - 1] - k, arr[i - 1] + k);
                minDifference = Math.min(minDifference, maxHeight - minHeight);
            }
        }
        return minDifference;
    }
}
```



```
For Input:
2
1 5 8 10
Your Output:
5
Expected Output:
5
```

Time Complexity: O(NlogN)

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.

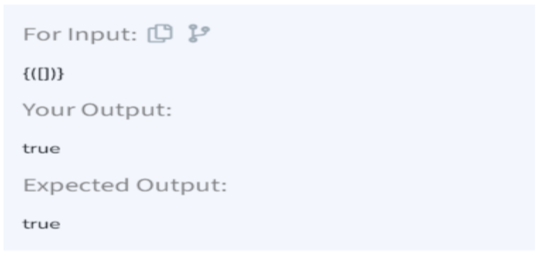
Input: s = "{([])}"

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:

```
class Solution {
    static boolean isParenthesisBalanced(String s) {
        Stack<Character> stack = new Stack<>();
        for (char ch : s.toCharArray()) {
            if (ch == '{' || ch == '[' || ch == '(') {
                stack.push(ch);
            }
            else if (ch == '}' || ch == ']' || ch == ')') {
                if (stack.isEmpty()) {
                    return false;
                }
                char top = stack.pop();
                if ((ch == '}' && top != '{') ||
                    (ch == ']' && top != '[') ||
                    (ch == ')' && top != '(')) {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }
}
```



```
For Input:
{([])}
Your Output:
true
Expected Output:
true
```

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.

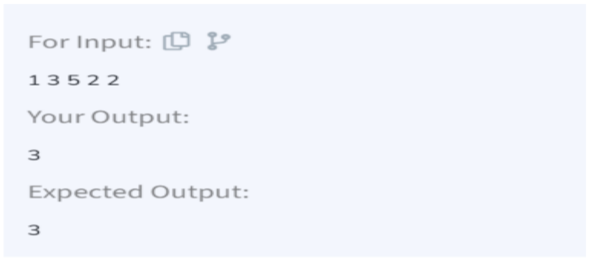
Note: Return equilibrium point in 1-based indexing. Return -1 if no such point exists.

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).

CODE:

```
  public static int equilibriumPoint(int arr[]) {
    int totalSum = 0;
    for (int num : arr) {
      totalSum += num;
    }
    int leftSum = 0;
    for (int i = 0; i < arr.length; i++) {
      int rightSum = totalSum - leftSum - arr[i];
      if (leftSum == rightSum) {
        return i + 1;
      }
      leftSum += arr[i];
    }
    return -1;
  }
}
```



```
For Input:
1 3 5 2 2
Your Output:
3
Expected 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.

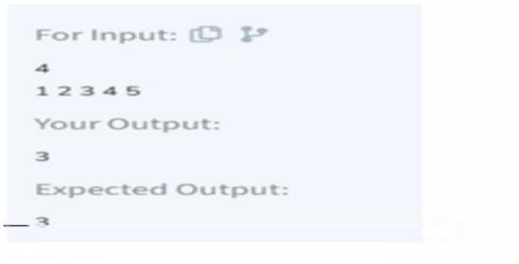
Input: arr[] = [1, 2, 3, 4, 5], k = 4

Output: 3

Explanation: 4 appears at index 3.

CODE:

```
import java.io.*;

class BinarySearch {

  int binarySearch(int arr[], int x)

  {

    int low = 0, high = arr.length - 1;

    while (low <= high) {

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

      if (arr[mid] == x)

        return mid;

      if (arr[mid] < x)

        low = mid + 1;

      else

        high = mid - 1;

    }

   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, x);

    if (result == -1)

      System.out.println(

        "Element is not present in array");

    else

      System.out.println("Element is present at "
                        + "index " + result);
    }
}
```



```
For Input:
4
1 2 3 4 5
Your Output:
3
Expected Output:
3
```

Time Complexity:O(nlogn)

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.

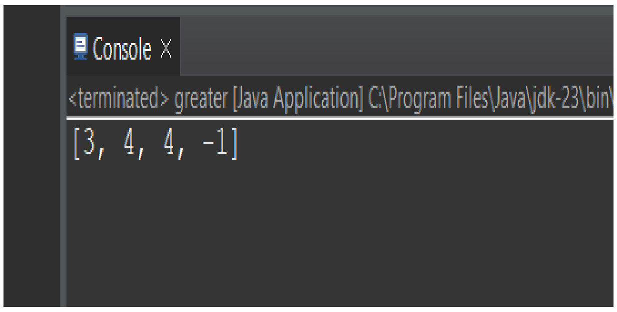
Input: arr[] = [1, 3, 2, 4]

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.

CODE:

```
import java.util.Arrays;
import java.util.Stack;
public class Solution {
    public static int[] nextGreaterElement(int[] arr) {
        int n = arr.length;
        int[] result = new int[n];
        Arrays.fill(result, -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[i] = stack.peek();
            }
            stack.push(arr[i]);
        }
        return result;
    }
    public static void main(String[] args) {
        int[] arr = {1, 3, 2, 4};
        int[] result = nextGreaterElement(arr);
        System.out.println(Arrays.toString(result));
    }
}
```



```
<terminated> greater [Java Application] C:\Program Files\Java\jdk-23\bin\
[3, 4, 4, -1]
```

Time Complexity: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.

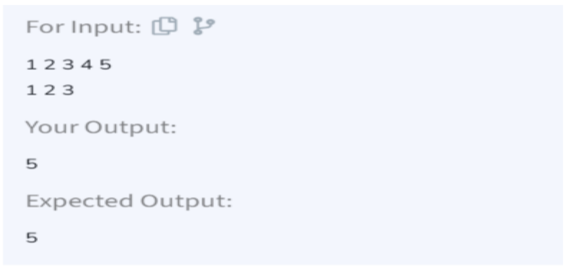
Input: a[] = [1, 2, 3, 4, 5], b[] = [1, 2, 3]

Output: 5

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

CODE:

```
class Solution {
    public static int findUnion(int a[], int b[]) {
         HashSet<Integer> unionSet = new HashSet<>();
        for (int num : a) {
            unionSet.add(num);
        }
        for (int num : b) {
            unionSet.add(num);
        }
        return unionSet.size();
    }
}
```



```
For Input:
1 2 3 4 5
1 2 3
Your Output:
5
Expected Output:
5
```

Time Complexity:O(N+M)