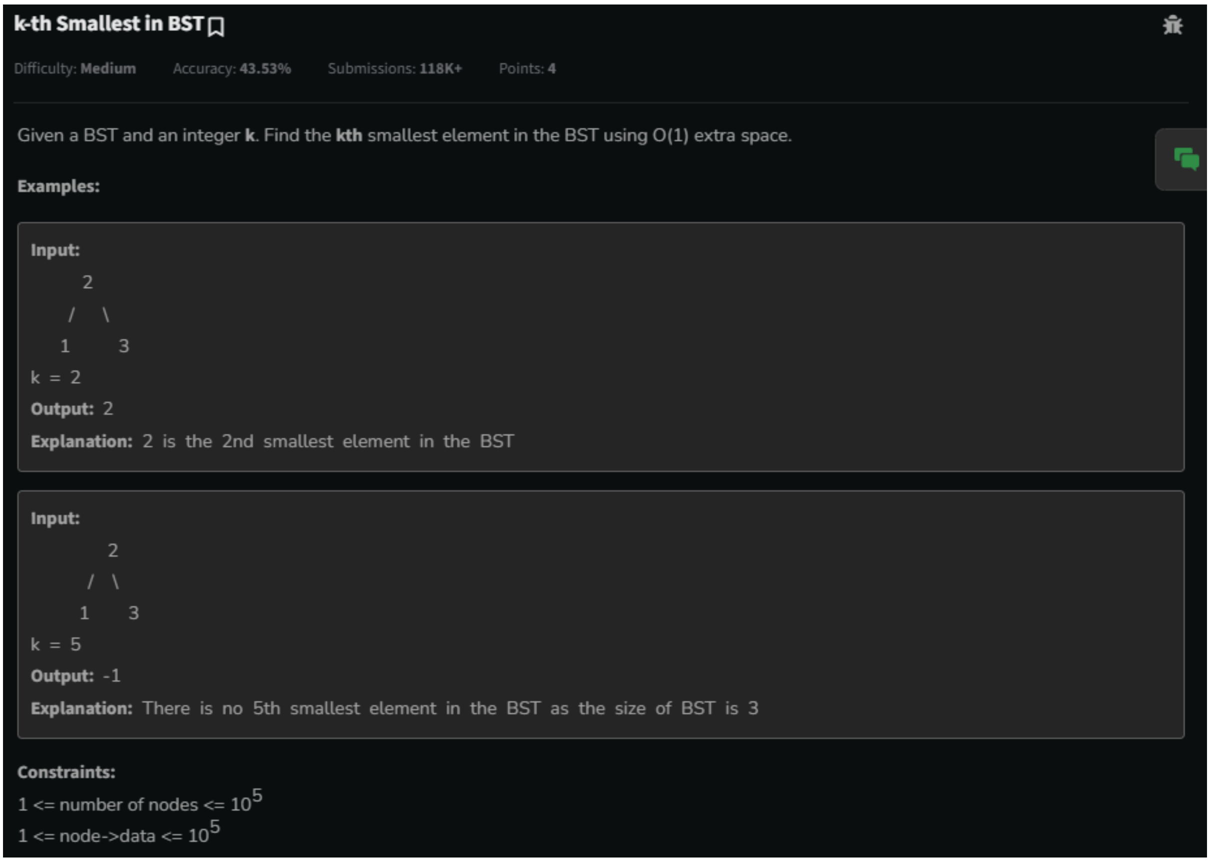
Coding Practice Set – 4

1. **Kth smallest element:**



k-th Smallest in BST

Difficulty: **Medium** Accuracy: **43.53%** Submissions: **118K+** Points: **4**

Given a BST and an integer **k**. Find the **kth** smallest element in the BST using O(1) extra space.

Examples:

```
Input:
     2
   /   \
  1     3
k = 2
Output: 2
Explanation: 2 is the 2nd smallest element in the BST
```

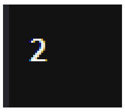
```
Input:
       2
     /  \
    1    3
k = 5
Output: -1
Explanation: There is no 5th smallest element in the BST as the size of BST is 3
```

Constraints:
1 <= number of nodes <= 10^5
1 <= node->data <= 10^5

Code:

```
public class Main {
    public static void main(String[] args) {
        TreeNode root = new TreeNode(2);
        root.left = new TreeNode(1);
        root.right = new TreeNode(3);
        Solution solution = new Solution();
        int k = 2;
        int result = solution.kthSmallest(root, k);

        System.out.println(result);
    }
}

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

class Solution {
    private int count = 0;
    private int result = -1;

    public int kthSmallest(TreeNode root, int k) {
        inorderTraversal(root, k);
        return result;
    }

    private void inorderTraversal(TreeNode root, int k) {
        if (root == null) return;

        inorderTraversal(root.left, k);

        count++;
        if (count == k) {
            result = root.val;
            return;
        }

        inorderTraversal(root.right, k);
    }
}
```

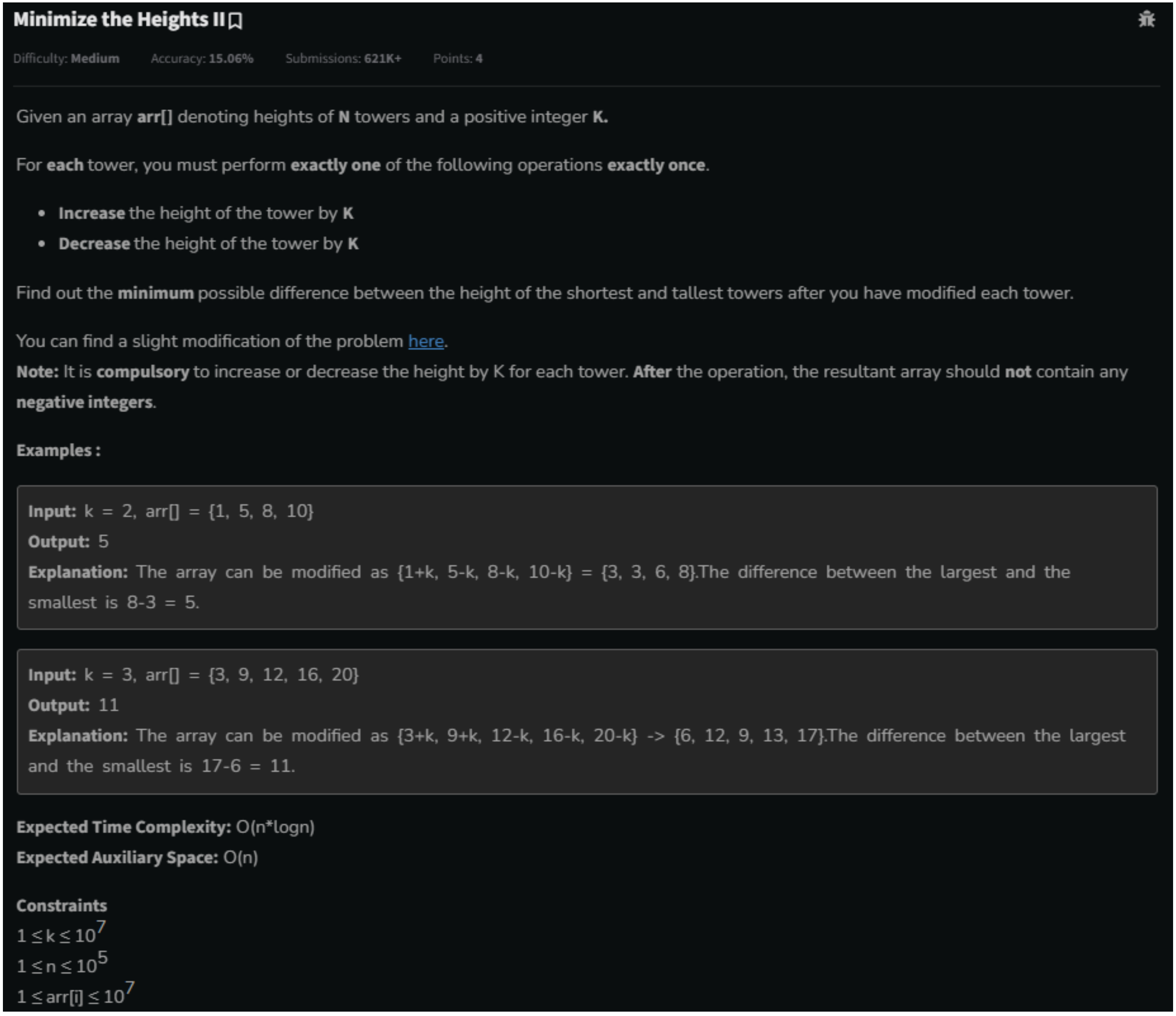
Output:



```
2
```

Time Complexity: O(N)

2. Minimize the heights II



Minimize the Heights II

Difficulty: **Medium** Accuracy: **15.06%** Submissions: **621K+** Points: **4**

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.

Expected Time Complexity: O(n*logn)
Expected Auxiliary Space: O(n)

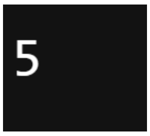
Constraints
$1 \le k \le 10^7$
$1 \le n \le 10^5$
$1 \le arr[i] \le 10^7$

Code:

```
import java.util.Arrays;

class Solution {
   int getMinDiff(int[] arr, int n, int k) {
      Arrays.sort(arr);
      int minDiff = arr[n - 1] - arr[0];
      int smallest = arr[0] + k;
      int largest = arr[n - 1] - k;

      for (int i = 0; i < n - 1; i++) {
         int minHeight = Math.min(smallest, arr[i + 1] - k);
         int maxHeight = Math.max(largest, arr[i] + k);
         if (minHeight >= 0) {
            minDiff = Math.min(minDiff, maxHeight - minHeight);
         }
      }
      return minDiff;
   }
```

```
    public static void main(String[] args) {
        Solution obj = new Solution();
        int[] arr = {1, 5, 8, 10};
        int k = 2;
        int n = arr.length;
        System.out.println(obj.getMinDiff(arr, n, k));
    }
}
```

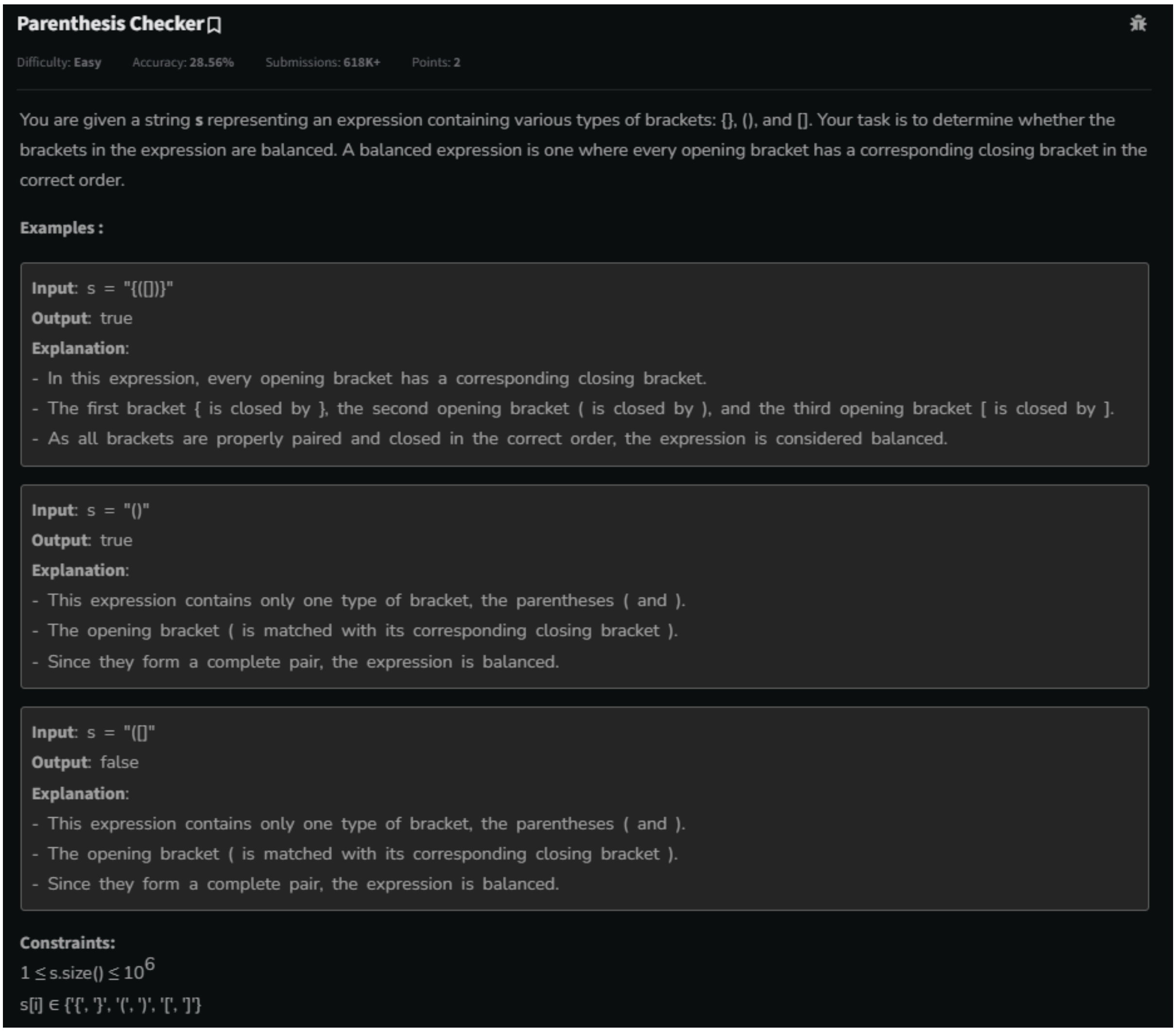
Output:



```
5
```

Time Complexity: O(n log n)

3. Parenthesis Checker



Parenthesis Checker

Difficulty: **Easy** Accuracy: **28.56%** Submissions: **618K+** Points: **2**

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.

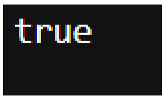
Constraints:
$1 \le s.size() \le 10^6$
s[i] ∈ {'{', '}', '(', ')', '[', ']'}

Code:

```
import java.util.Stack;

public class ParenthesisChecker {
   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 == ')' || ch == '}' || ch == ']') {
            if (stack.isEmpty()) {
               return false;
            }
            char top = stack.pop();
            if ((ch == ')' && top != '(') ||
               (ch == '}' && top != '{') ||
               (ch == ']' && top != '[')) {
               return false;
            }
         }
```

```
        }
        return stack.isEmpty();
    }

    public static void main(String[] args) {
        String s = "{([])}";
        System.out.println(isBalanced(s));
    }
}
```

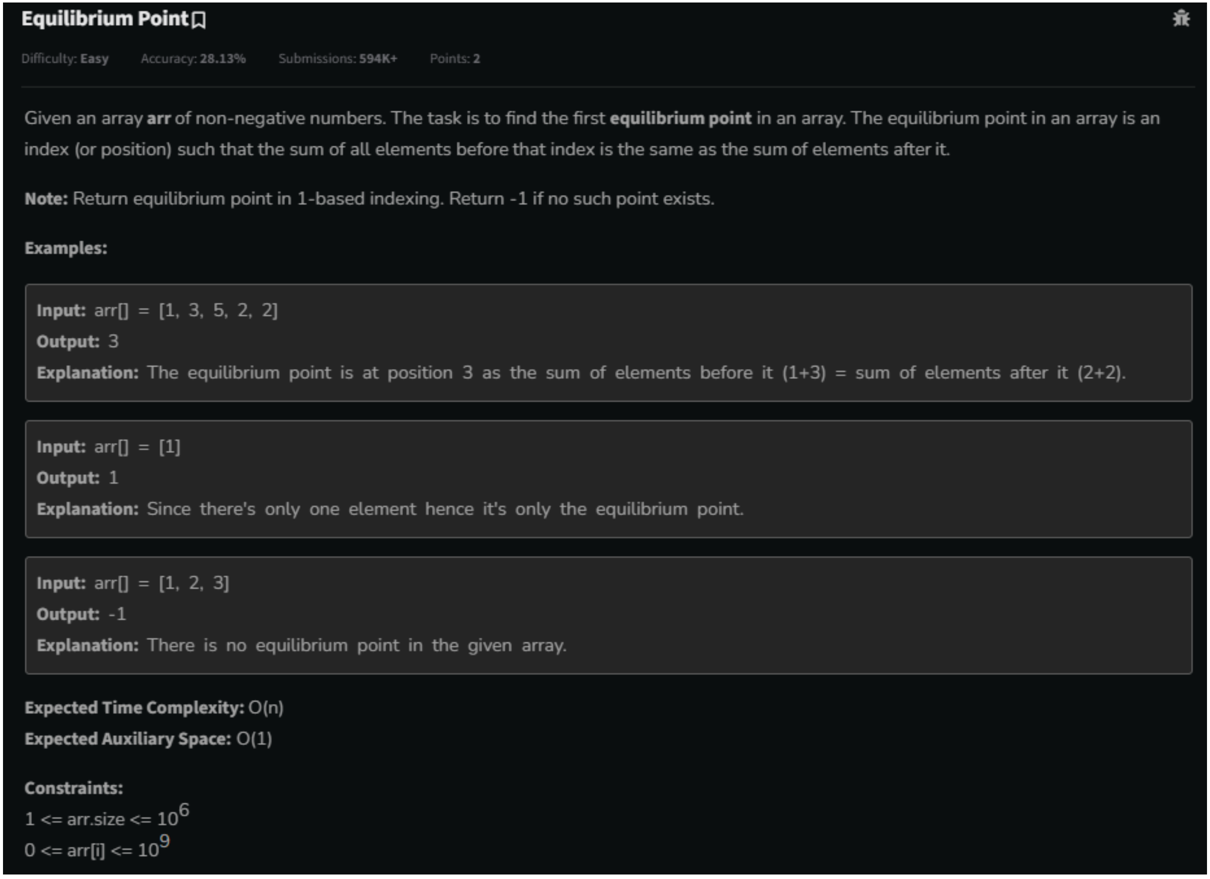
Output:



```
true
```

Time Complexity: O(N)

4. Equilibrium point



Equilibrium Point

Difficulty: **Easy** Accuracy: **28.13%** Submissions: **594K+** Points: **2**

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.

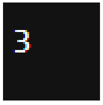
Expected Time Complexity: O(n)
Expected Auxiliary Space: O(1)

Constraints:
1 <= arr.size <= 10^6
0 <= arr[i] <= 10^9

Code:

```
public class EquilibriumPoint {
    public static int findEquilibriumPoint(long[] arr, int n) {
        long totalSum = 0, leftSum = 0;
        for (long num : arr) {
            totalSum += num;
        }
        for (int i = 0; i < n; i++) {
            totalSum -= arr[i];
            if (leftSum == totalSum) {
                return i + 1;
            }
            leftSum += arr[i];
        }
        return -1;
    }

    public static void main(String[] args) {
        long[] arr = {1, 3, 5, 2, 2};
        int n = arr.length;
        System.out.println(findEquilibriumPoint(arr, n));
    }
}
```

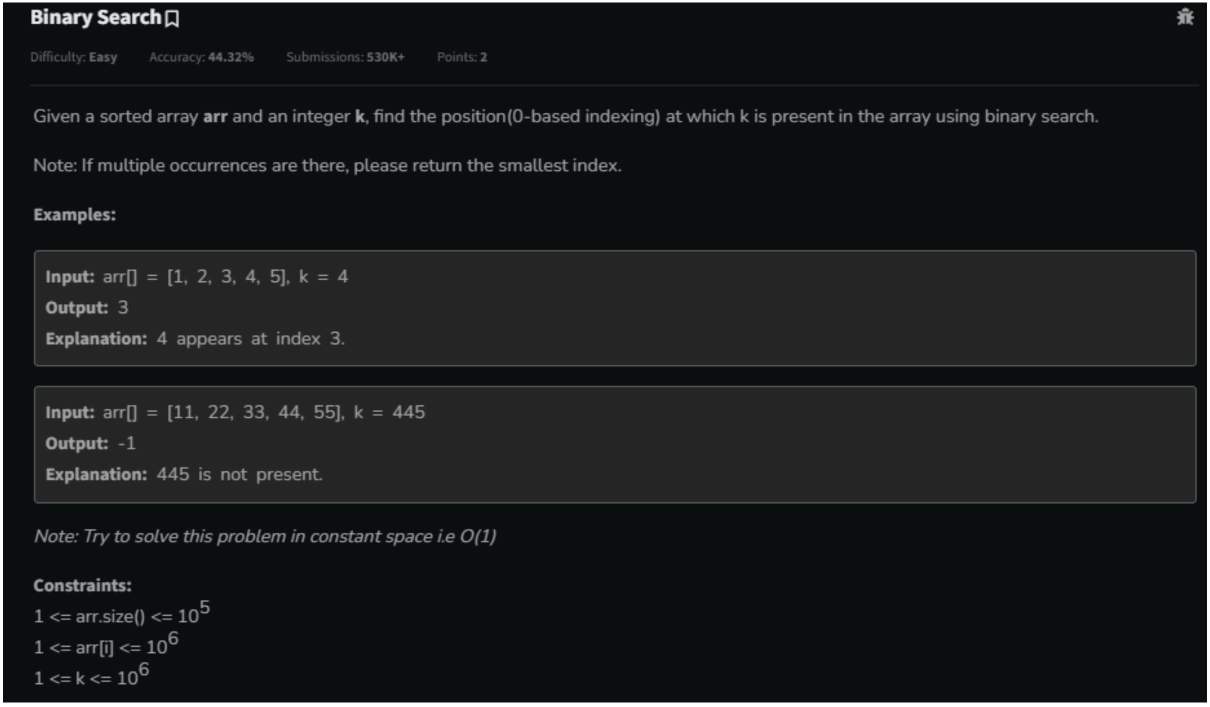
Output:



```
3
```

Time complexity: O(N)

5. Binary Search



Binary Search

Difficulty: **Easy** Accuracy: **44.32%** Submissions: **530K+** Points: **2**

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.

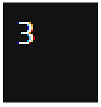
Note: Try to solve this problem in constant space i.e O(1)

Constraints:
1 <= arr.size() <= 10^5
1 <= arr[i] <= 10^6
1 <= k <= 10^6

Code:

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int k) {
        int low = 0, high = arr.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == k) {
                while (mid > 0 && arr[mid - 1] == k) {
                    mid--;
                }
                return mid;
            } else if (arr[mid] < k) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int k = 4;
        System.out.println(binarySearch(arr, k));
    }
}
```

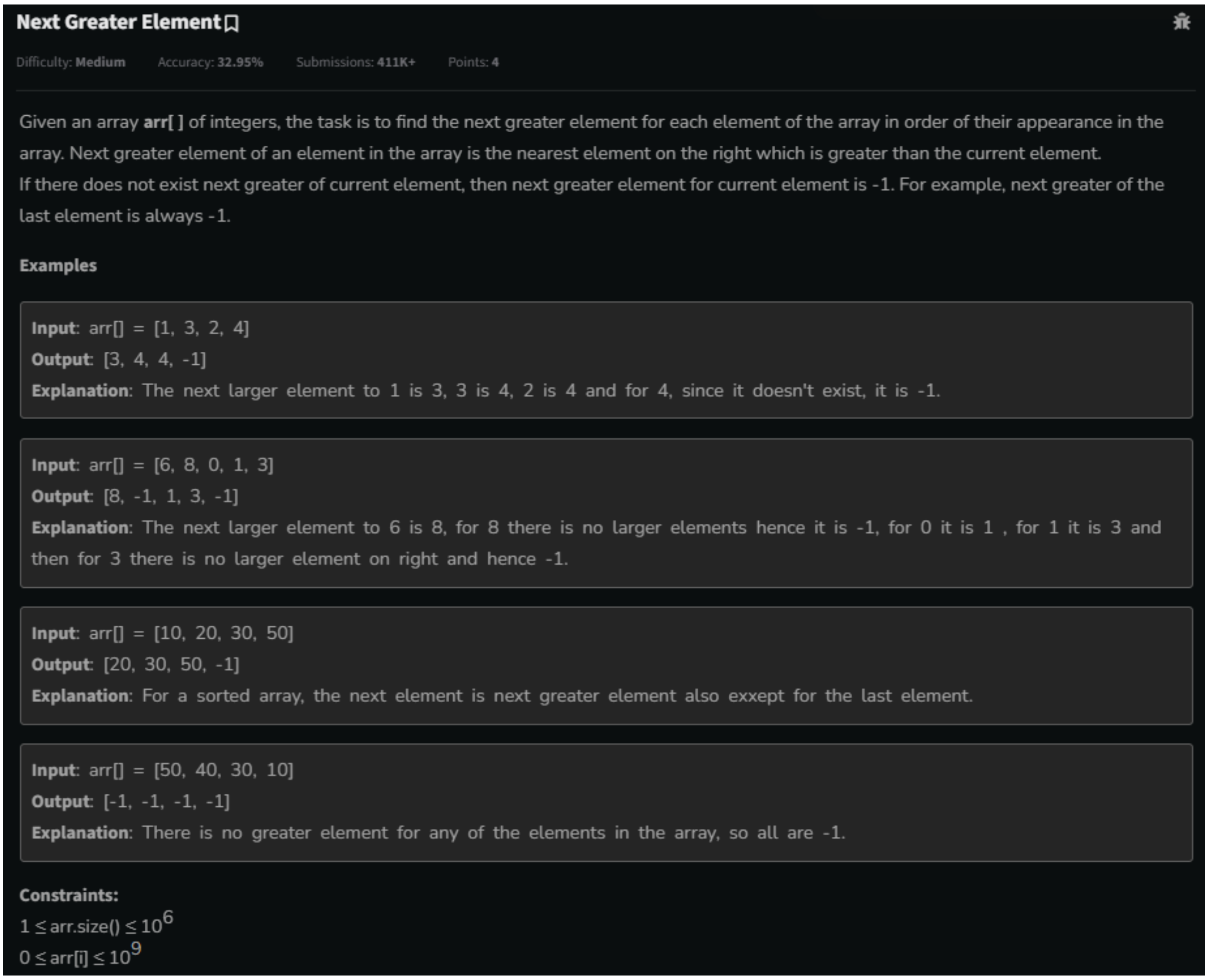
Output:



```
3
```

Time complexity: O(log n)

6. Next Greater element



Next Greater Element

Difficulty: **Medium** Accuracy: **32.95%** Submissions: **411K+** Points: **4**

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.

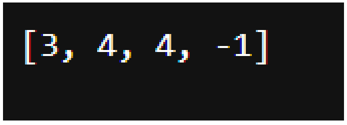
Input: arr[] = [50, 40, 30, 10]
Output: [-1, -1, -1, -1]
Explanation: There is no greater element for any of the elements in the array, so all are -1.

Constraints:
$1 \le arr.size() \le 10^6$
$0 \le arr[i] \le 10^9$

Code:

```
import java.util.*;

public class NextGreaterElement {
    public static int[] findNextGreaterElement(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[] arr = {1, 3, 2, 4};
        int[] result = findNextGreaterElement(arr);
        System.out.println(Arrays.toString(result));
    }
}
```

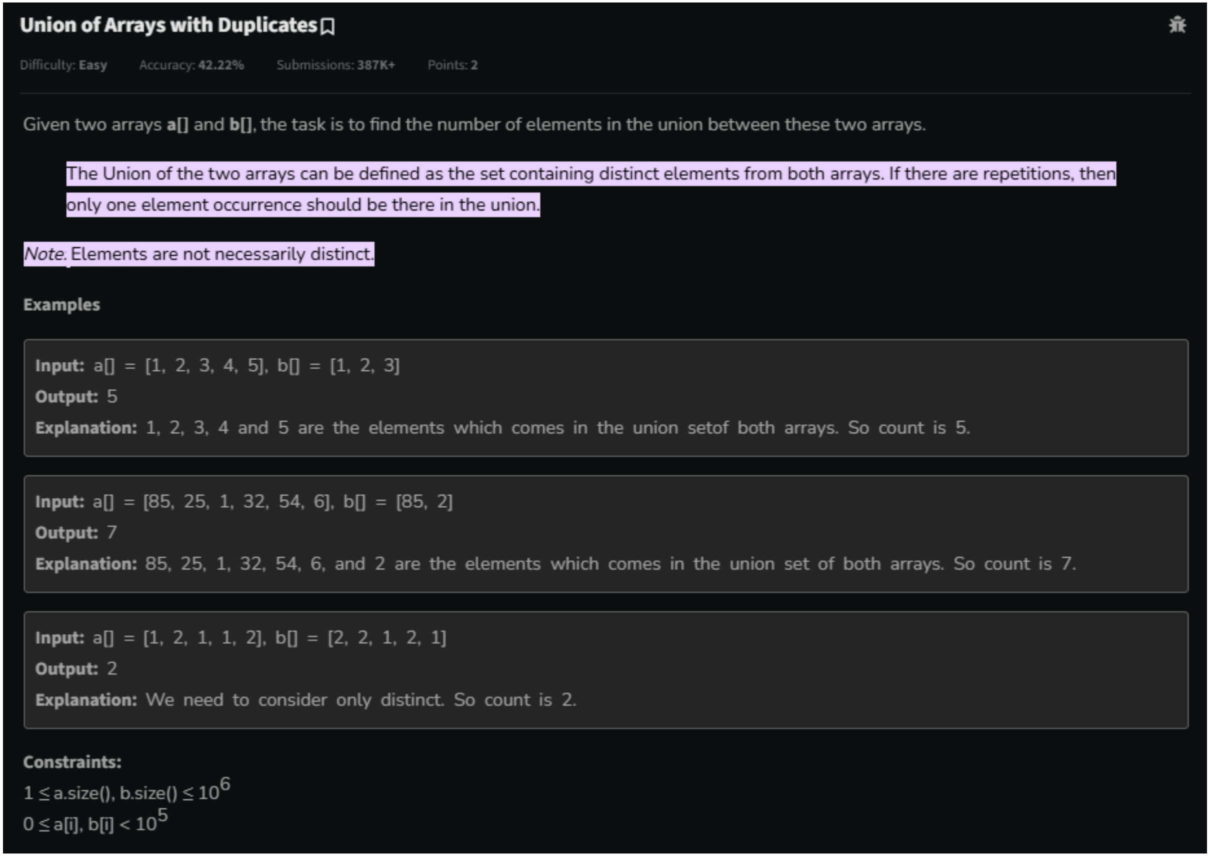
Output:



```
[3, 4, 4, -1]
```

Time complexity: O(N)

7. Union of two arrays with Duplicate elements:



Union of Arrays with Duplicates

Difficulty: Easy   Accuracy: 42.22%   Submissions: 387K+   Points: 2

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.

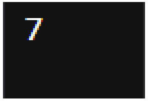
Input: a[] = [1, 2, 1, 1, 2], b[] = [2, 2, 1, 2, 1]
Output: 2
Explanation: We need to consider only distinct. So count is 2.

Constraints:
$1 \le$ a.size(), b.size() $\le 10^6$
$0 \le$ a[i], b[i] $< 10^5$

Code:

```
import java.util.*;

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

    public static void main(String[] args) {
        int[] a = {85, 25, 1, 32, 54, 6};
        int[] b = {85, 2};
        System.out.println(findUnionCount(a, b));
    }
}
```

Output:



```
7
```

Time complexity: O(N+M)