Experiment-5

Student Name: Jatin Garg **UID:** 22BCS15676
Branch: BE-CSE **Section/Group:** 640/B
Semester: 6th **Date of Performance:** 21/02/25
Subject Name: AP LAB-II **Subject Code:** 22CSP-351

1. Aim:

Sorting and searching are fundamental concepts in computer science and play a crucial role in optimizing various algorithms. They are widely used in data processing, databases, searching algorithms, and real-world applications like ranking systems, recommendation engines, and route planning.

2. Introduction to the Searching and Sorting:

Sorting

Sorting refers to arranging data in a specific order, typically in ascending or descending order. It helps in efficient searching, data analysis, and enhances the performance of other algorithms.

Types of Sorting Algorithms

- **Comparison-based Sorting**

These algorithms compare elements to determine their order.

1. Bubble Sort ($O(n^2)$) – Simple but inefficient for large data.
2. Selection Sort ($O(n^2)$) – Selects the smallest element and places it in the correct position.
3. Insertion Sort ($O(n^2)$) – Inserts elements into their correct position one by one.
4. Merge Sort ($O(n \log n)$) – Uses the divide and conquer approach.
5. Quick Sort ($O(n \log n)$) – Selects a pivot and partitions elements around it.
6. Heap Sort ($O(n \log n)$) – Uses a binary heap for sorting.

- **Non-comparison-based Sorting**

These algorithms do not compare elements directly.

1. Counting Sort ($O(n + k)$) – Works well for small integer ranges.
2. Radix Sort ($O(nk)$) – Sorts numbers digit by digit.
3. Bucket Sort ($O(n + k)$) – Divides elements into buckets and sorts them.

Searching

Searching refers to finding an element in a data structure like an array, linked list, or tree.

Types of Searching Algorithms

1. Linear Search ($O(n)$):

- Scans elements one by one.
- Used when data is unsorted or small.
- Example: Searching for a name in an unsorted list.

2. Binary Search (O(log n)):

- Requires sorted data.
- Uses a divide and conquer approach.
- Example: Finding a number in a sorted list.

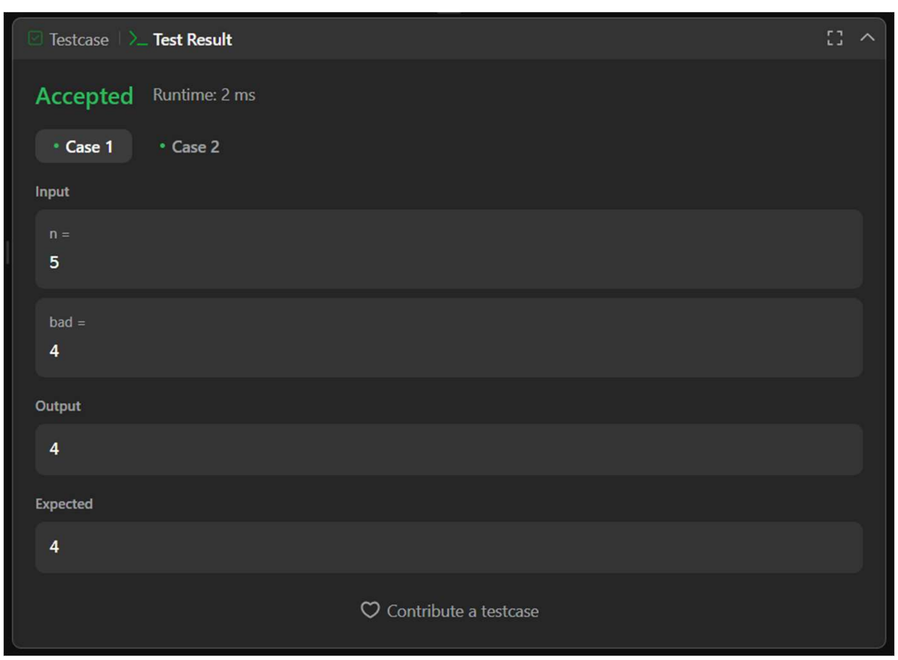
3. Implementation/Code:

- **278 First Bad Version:**

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int left = 1, right = n;
        while (left < right) {
            int mid = left + (right - left) / 2;

            if (isBadVersion(mid)) {
                right = mid; // Narrow search to the left half
            } else {
                left = mid + 1; // Narrow search to the right half
            }
        }
        return left; // or return right (both are same)
    }
}
```

278 Output:

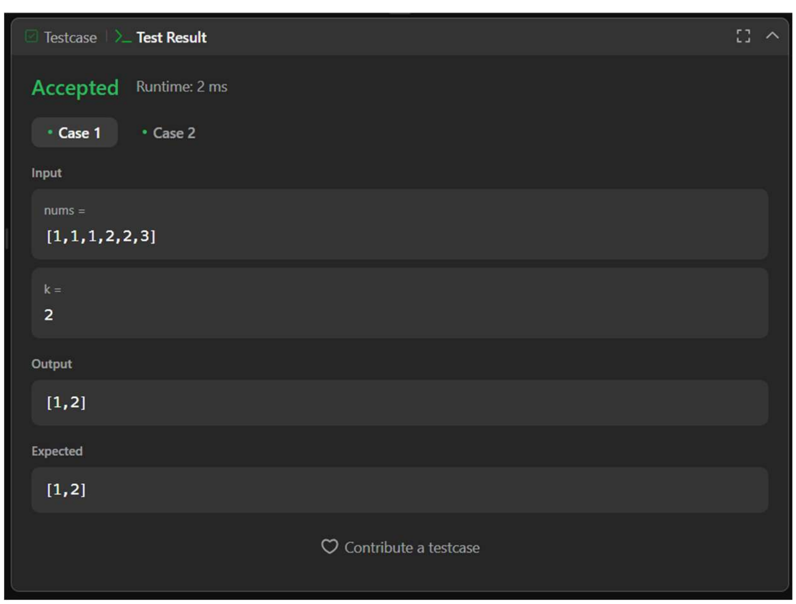


- **347 Top K Frequent Elements:**

```
import java.util.*;
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> freqMap = new HashMap<>();
        for (int num : nums) {
            freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
        }
        List<Integer>[] buckets = new ArrayList[nums.length + 1];
        for (int key : freqMap.keySet()) {
            int freq = freqMap.get(key);
            if (buckets[freq] == null) buckets[freq] = new ArrayList<>();
            buckets[freq].add(key);
        }
        List<Integer> result = new ArrayList<>();
        for (int i = buckets.length - 1; i >= 0 && result.size() < k; i--) {
            if (buckets[i] != null) {
                result.addAll(buckets[i]);


            }
        }
        return result.stream().mapToInt(i -> i).toArray();
    }
}
```

347 Output:



- **162 Find Peak Elements:**

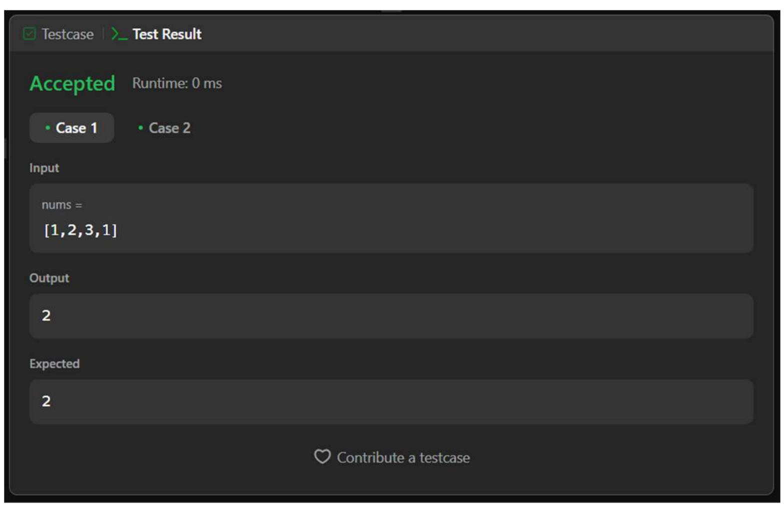
```
class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0, right = nums.length - 1;

        while (left < right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] > nums[mid + 1]) {
                right = mid; // Move left
            } else {
                left = mid + 1; // Move right
            }
        }

        return left; // Peak found
    }
}
```

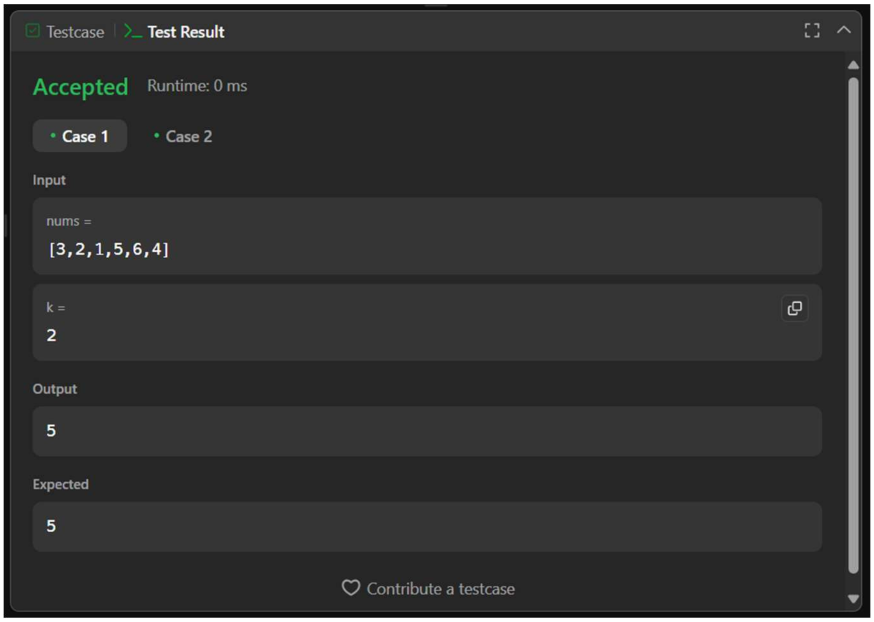
162 Output:



- **215 Kth Largest Element in an Array:**

```
import java.util.Arrays;
class Solution {
    public int findKthLargest(int[] nums, int k) {
        Arrays.sort(nums); // Sort in ascending order
        return nums[nums.length - k]; // Get k-th largest element
    }
}
```

215 Output:



4. Learning Outcome

- ➢ Understand how to use storing and searching.
- ➢ Understanding binary search beyond simple sorted arrays.
- ➢ Learn about how greedy algorithms help in reducing unnecessary computations.
- ➢ Using a max heap to dynamically track the highest building at any given x-coordinate.