Experiment-5

Student Name: Garv Kumar **UID:** 22BET10103

Branch: BE-IT **Section/Group:** 22BET_IOT-702/A

Semester: 6th **Date of Performance:** 21/02/2025

Subject Name: Advanced Programming Lab-2 **Subject Code:** 22ITP-351

Problem-1

1. Aim:

The goal is to merge two sorted integer arrays, nums1 and nums2, where nums1 has enough space to accommodate all elements from nums2. The integers m and n specify the valid elements in nums1 and nums2, respectively. The result should be a single sorted array in non-decreasing order.

2. Objective:

- Merge Sorted Arrays: Combine the elements of nums1 and nums2 into a single sorted array while maintaining non-decreasing order.
- In-Place Modification: Store the merged result directly in nums1, utilizing its extra space without using additional arrays.

3. Implementation:

```
class Solution {
public:
   void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
      int i = m - 1, j = n - 1, k = m + n - 1;
      while (i >= 0 && j >= 0) {
         if (nums1[i] > nums2[j]) {
            nums1[k--] = nums1[i--];
         } else {
            nums1[k--] = nums2[j--];
         }
      }
```

```
        while (j >= 0) {
            nums1[k--] = nums2[j--];
        }
    }
};
```

4. Output:

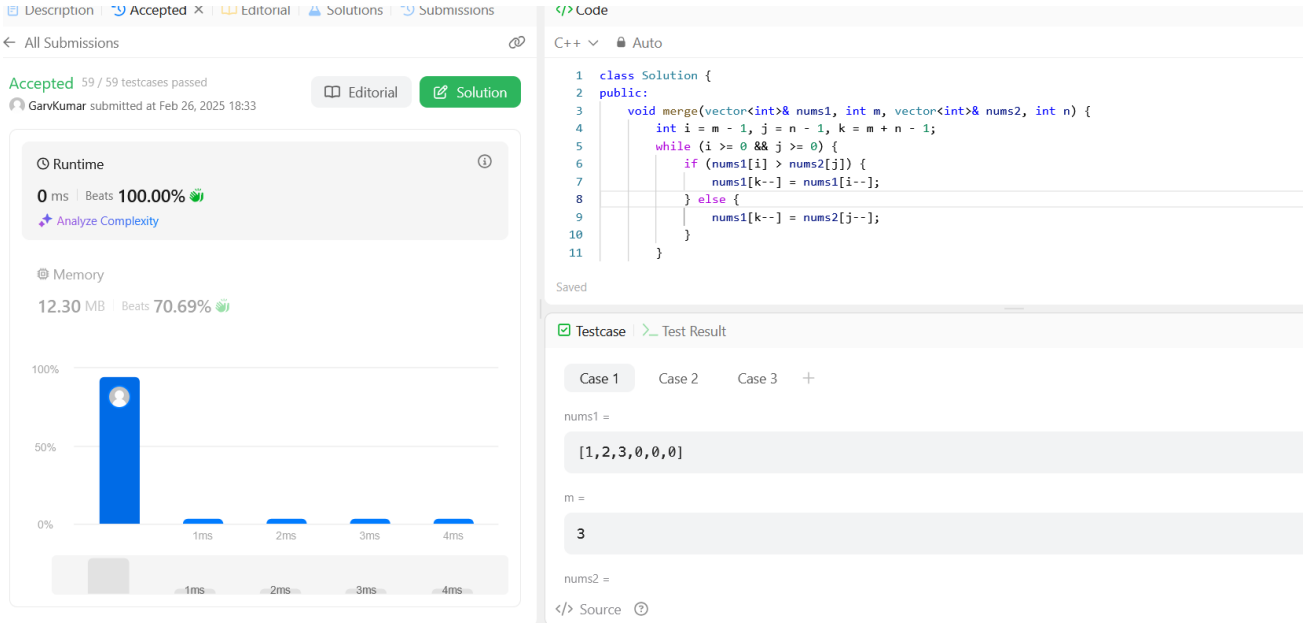

Fig: Merge Sorted Array.

Problem-2

1. Aim:

As a product manager, you discover that the latest product version has failed quality testing. Since each version builds on the previous one, all subsequent versions are also flawed. The goal is to find the first defective version to prevent further issues.

2. Objective:

1 Identify the First Defective Version: Determine the earliest version that fails the quality check to prevent further issues in product development.

2 Optimize Debugging and Production: Efficiently locate the faulty version to minimize production delays and ensure a high-quality product release.

3. Implementation:

```
class Solution {
public:
    int firstBadVersion(int n) {
        int left = 1, right = n;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (isBadVersion(mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
};
```

4. Output:

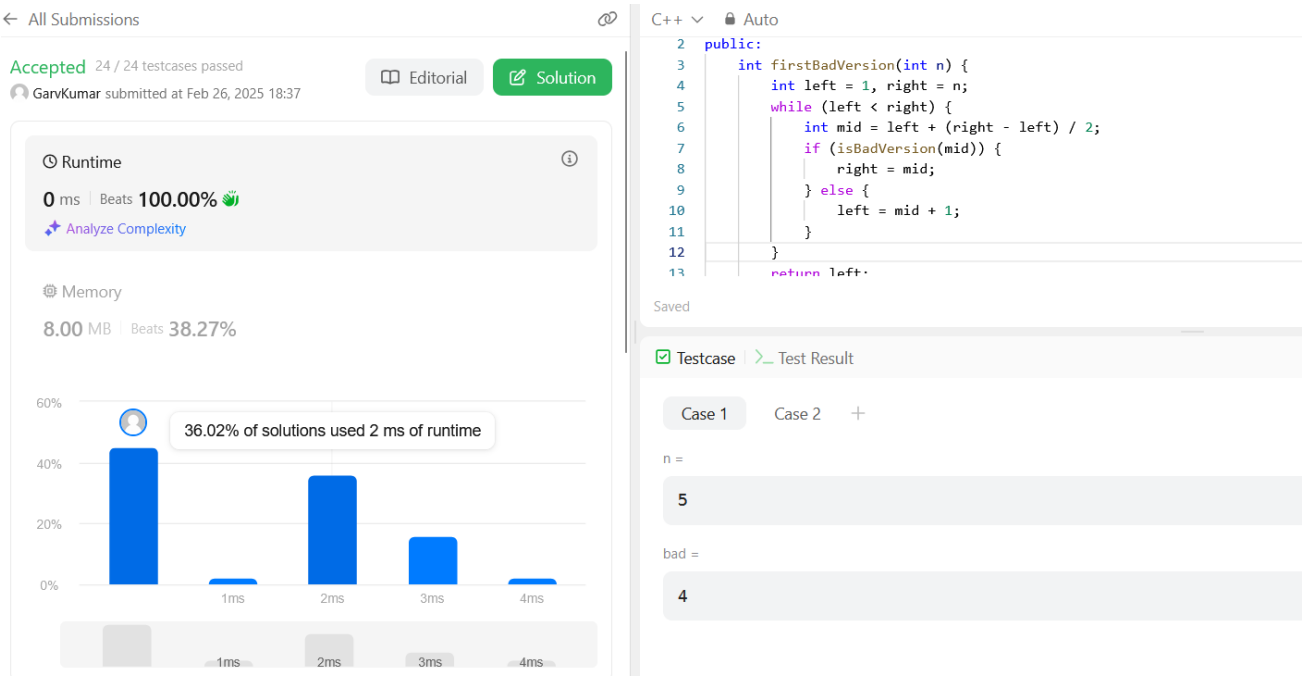

Fig: First Bad Version.

Problem-3

1. Aim:

Given an array nums containing n objects colored red, white, or blue, rearrange them in-place so that identical colors are grouped together in the order: red, white, and blue.

2. Objective:

1 Sort Colors Efficiently: Reorder the elements in-place to ensure all red objects appear first, followed by white, and then blue.

2 Preserve In-Place Sorting: Achieve the sorting without using extra space, maintaining the original array structure.

3. Implementation:

```
class Solution {

public:

   void sortColors(vector<int>& nums) {

      int low = 0, mid = 0, high = nums.size() - 1;

      while (mid <= high) {

         if (nums[mid] == 0) {

            swap(nums[low++], nums[mid++]);

         } else if (nums[mid] == 1) {

            mid++;

         } else {

            swap(nums[mid], nums[high--]);

         }

      }

   }

    };
```

4. Output:

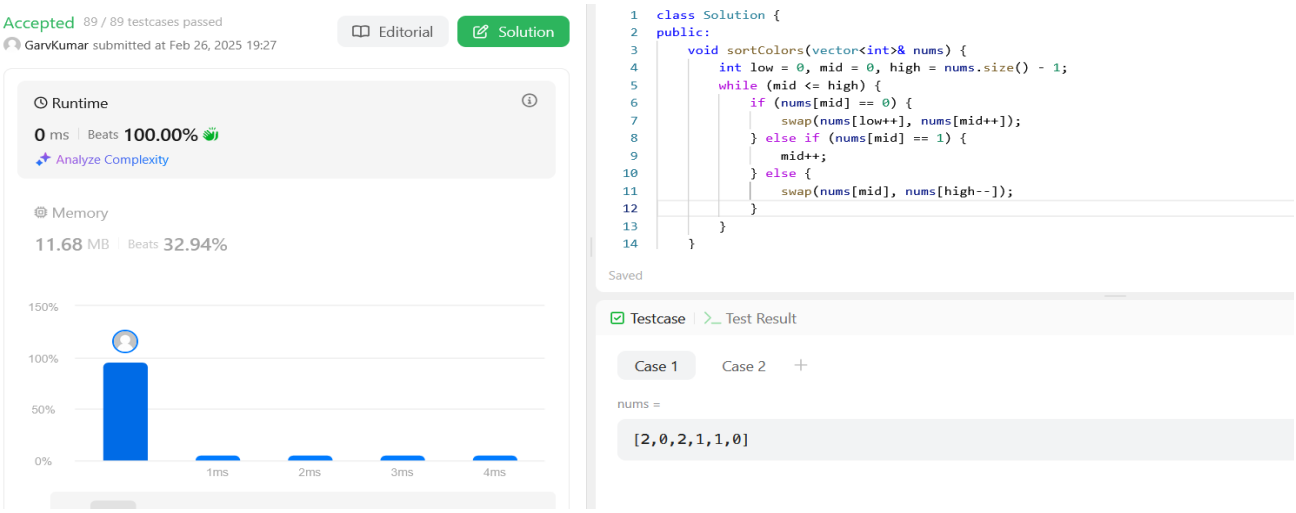

Fig: Sort Colors.

Problem-4

1. Aim:

Given an integer array nums and a number k, identify the k most frequently occurring elements. The result can be returned in any order.

2. Objective:

1 Identify Top k Frequent Elements: Determine the k elements that appear most often in the array.

2 Efficient Retrieval: Implement an optimized approach to extract the k most frequent elements while minimizing time complexity.

3. Implementation:

```
class Solution {
public:
   vector<int> topKFrequent(vector<int>& nums, int k) {
      unordered_map<int, int> freq;
      for (int num : nums) {
         freq[num]++;
      }
      priority_queue<pair<int, int>> pq;
```

```
    for (auto& [num, count] : freq) {
        pq.push({count, num});
    }
    vector<int> result;
    while (k-- > 0) {
        result.push_back(pq.top().second);
        pq.pop();
    }
    return result;
}
};
```

4. Output:

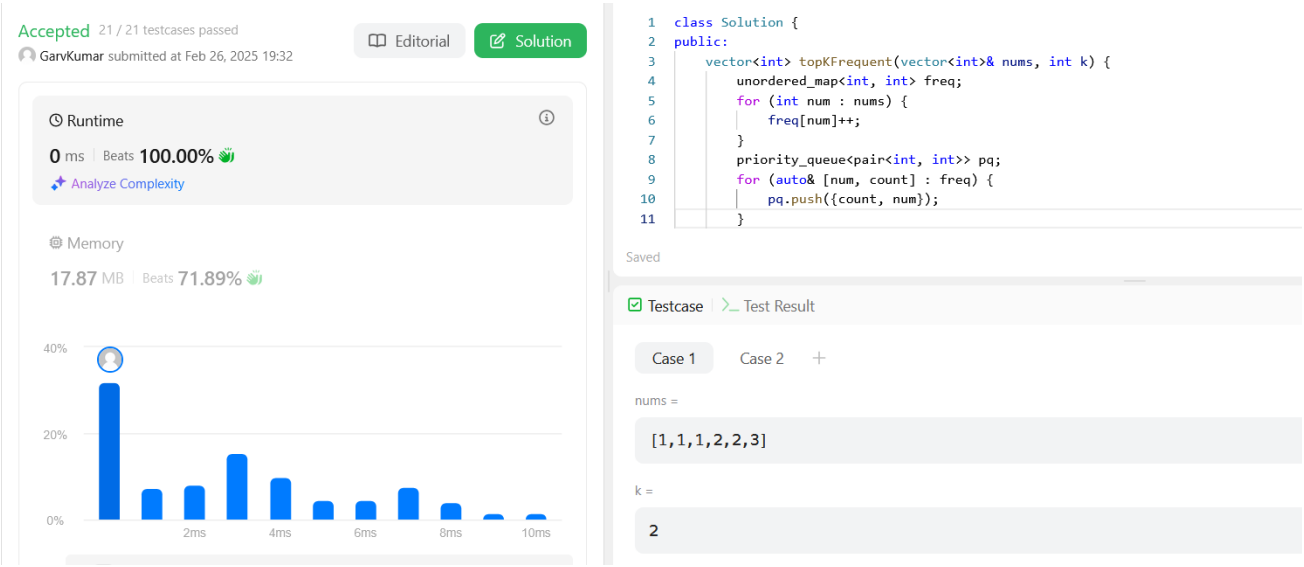

Fig: Top K Frequent Elements.

Problem-5

1. Aim:

Given an integer array nums and a number k, find the kth largest element in the array based on sorted order, not distinct values. The solution should avoid direct sorting if possible.

2. Objective:

1 Find the kth Largest Element: Identify the kth largest number in the array while maintaining efficiency.

2 Optimize Without Sorting: Implement a solution that avoids full sorting to improve performance, especially for large datasets.

3. Implementation:

```
class Solution {
public:
   int findKthLargest(vector<int>& nums, int k) {
      priority_queue<int, vector<int>, greater<int>> minHeap;
      for (int num : nums) {
         minHeap.push(num);
         if (minHeap.size() > k) {
            minHeap.pop();
         }
      }
      return minHeap.top();
   }
    };
```

4. Output:

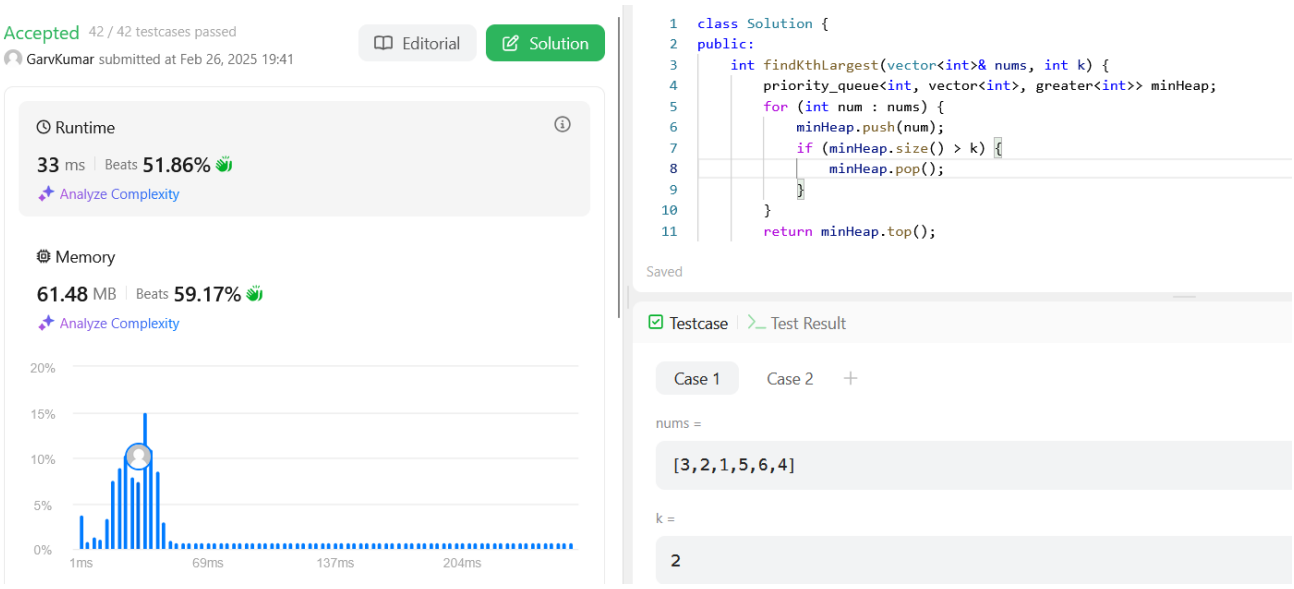

Fig: Kth Largest Element in an Array.

Problem-6

1. Aim:

Given a 0-indexed integer array nums, find and return the index of a peak element, which is strictly greater than its neighbors. If multiple peaks exist, return any one. The solution must run in O(log n) time.

2. Objective:

1 Identify a Peak Element: Locate an element that is strictly greater than its adjacent values.

2 Optimize Search Efficiency: Use an O(log n) approach, such as binary search, to find a peak efficiently.

3. Implementation:

```
class Solution {

public:

   int findPeakElement(vector<int>& nums) {

      int left = 0, right = nums.size() - 1;

      while (left < right) {

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

         if (nums[mid] > nums[mid + 1]) {

            right = mid;

         } else {

            left = mid + 1;

         }

      }

      return left;

   }
```

};

4. Output:

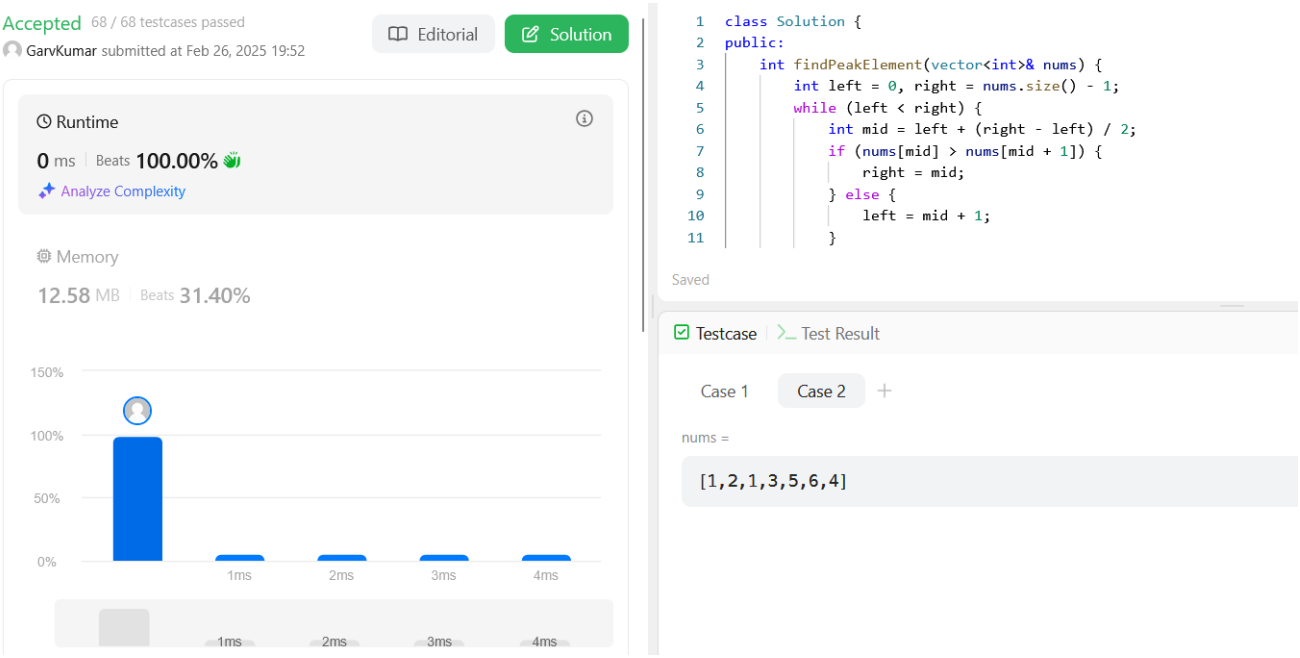

Fig: Find Peak Element.

5. Learning Outcomes:

1. **Learning Outcomes for Merging Two Sorted Arrays:**
   1 In-Place Merging: Learn how to merge two sorted arrays directly within the first array without using extra space.
   2 Maintaining Sorted Order: Understand techniques to efficiently combine elements while preserving non-decreasing order.
2. **Learning Outcomes for Identifying the First Defective Product Version:**
   1 Binary Search for Efficiency: Apply binary search to quickly locate the first bad version while minimizing unnecessary checks.
   2 Minimizing Debugging Efforts: Understand how early identification of defects can optimize product quality and reduce delays.
3. **Learning Outcomes for Sorting Colors In-Place:**
   1 Optimized Sorting Methods: Learn how to sort elements with a limited range of values efficiently using in-place techniques.
   2 Applying the Two-Pointer Approach: Understand how the two-pointer or Dutch National Flag algorithm can be used for optimal sorting.
4. **Learning Outcomes for Finding the k Most Frequent Elements:**

1 Efficient Frequency Counting: Learn how to track occurrences of elements using hash maps or frequency arrays.

2 Heap and Bucket Sort Usage: Understand how to apply heaps or bucket sort to extract the k most frequent elements efficiently.

5. **Learning Outcomes for Finding the kth Largest Element:**

1 Understanding Selection Algorithms: Learn how to determine the kth largest element without fully sorting the array.

2 Applying Heap and Quickselect: Gain proficiency in using heap data structures or the Quickselect algorithm for optimized performance.

6. **Learning Outcomes for Finding a Peak Element:**

1 Binary Search for Peak Finding: Learn how binary search can efficiently locate a peak element in O (log n) time.

2 Handling Boundary Conditions: Understand how to treat out-of-bound indices as negative infinity to simplify edge cases.