**Experiment 5**

**Student Name:** Dixit Balotra **UID:** 22BET10238

**Branch:** BE -IT **Section/Group:**22BET\_IOT-702/B

**Semester:** 6th **Date of Performance:**21/2/2025 **Subject Name:** Advanced Programming Lab-2 **Subject Code:** 22ITP-351

1. **Merge Sorted Array**

# Aim :

# To implement an efficient algorithm to merge two sorted arrays into one sorted array without using extra space.

# Objectives :

# Understand the concept of merging two sorted arrays.

# Implement an in-place merge algorithm.

# Optimize time and space complexity.

# Implementation/Code :

# class Solution {

# public void merge(int[] nums1, int m, int[] nums2, int n) {

# // Pointers for nums1, nums2, and the end of the merged array

# int p1 = m - 1; // Last element in nums1's valid range

# int p2 = n - 1; // Last element in nums2

# int p = m + n - 1; // Last position in nums1

# // Merge in reverse order to avoid overwriting elements in nums1

# while (p1 >= 0 && p2 >= 0) {

# if (nums1[p1] > nums2[p2]) {

# nums1[p] = nums1[p1];

# p1--;

# } else {

# nums1[p] = nums2[p2];

# p2--;

# }

# p--;

# }

# // If there are leftover elements in nums2, copy them over

# while (p2 >= 0) {

# nums1[p] = nums2[p2];

# p2--;

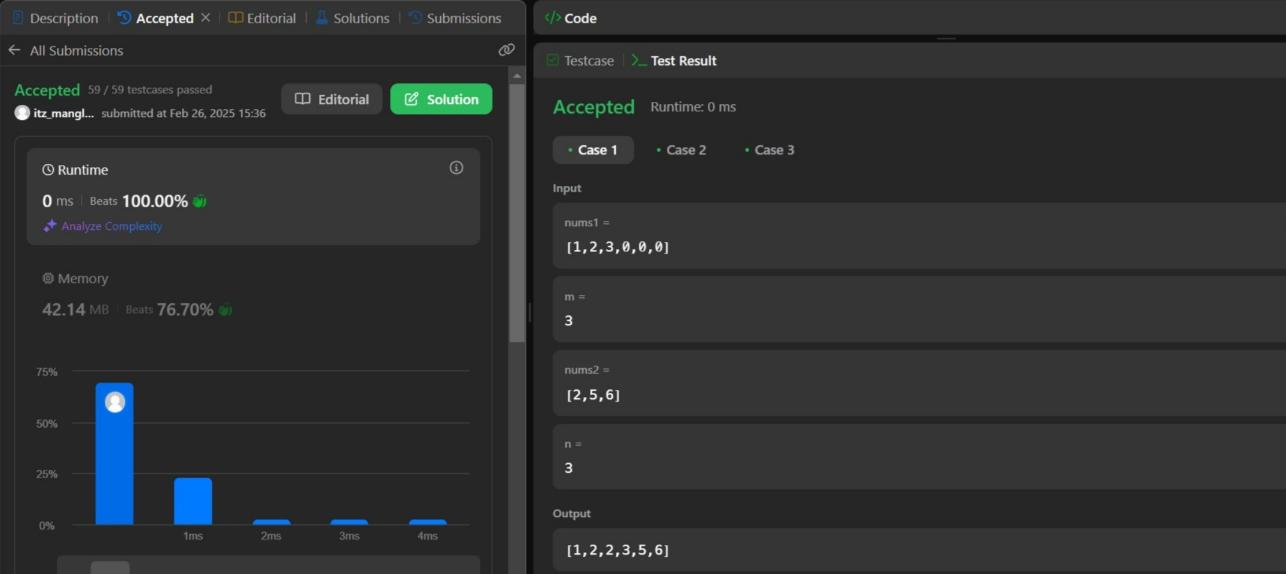
# p--;

# }

# }

# }

# Output :



1. **First Bad Version**

**Aim :**

To identify the first bad version in a series of versions using an optimized search algorithm.

# Objective :

* Understand the concept of binary search.
* Learn how to minimize API calls for optimized performance.
* Implement a solution using a divide-and-conquer approach.

# Implementation/Code :

# /\* The isBadVersion API is defined in the parent class VersionControl.

# boolean isBadVersion(int version); \*/

# public class Solution extends VersionControl {

# public int firstBadVersion(int n) {

# int left = 1; // start from version 1

# int right = n; // end at version n

# 

# while (left < right) { // binary search loop

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

# if (isBadVersion(mid)) {

# right = mid; // move left to search for earlier bad version

# } else {

# left = mid + 1; // move right to look for bad version

# }

# }

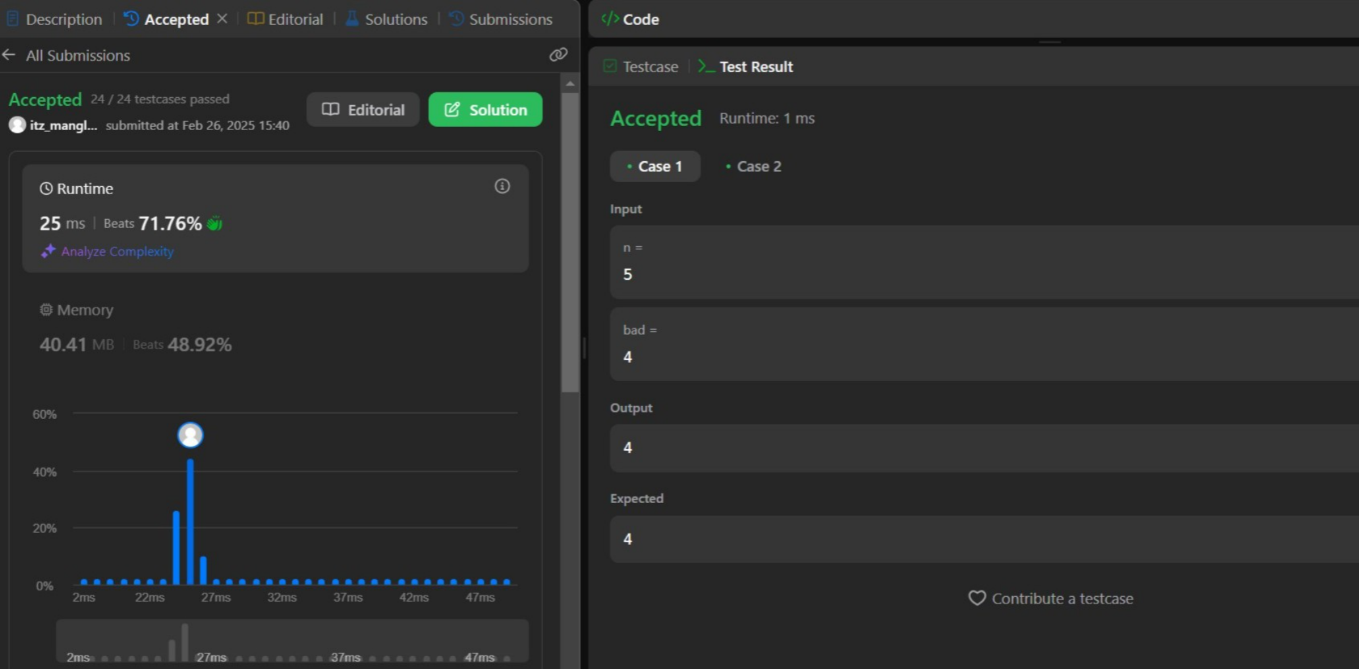
# 

# return left; // left will point to the first bad version

# }

# }

# Output :



1. **Sort Colors**

**Aim :**

To implement a sorting algorithm that rearranges an array of colors represented by integers (0, 1, 2) in a specific order without using built-in sorting methods.

# Objective :

# Learn the Dutch National Flag problem.

# Implement an in-place sorting algorithm.

# Optimize the sorting process to linear time complexity.

# Implementation/Code :

class Solution {

public void sortColors(int[] nums) {

int low = 0; // Pointer for 0s (red)

int mid = 0; // Pointer for 1s (white)

int high = nums.length - 1; // Pointer for 2s (blue)

while (mid <= high) {

if (nums[mid] == 0) { // If current element is 0

swap(nums, low, mid);

low++;

mid++;

} else if (nums[mid] == 1) { // If current element is 1

mid++;

} else { // If current element is 2

swap(nums, mid, high);

high--;

}

}

}

// Helper method to swap elements

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

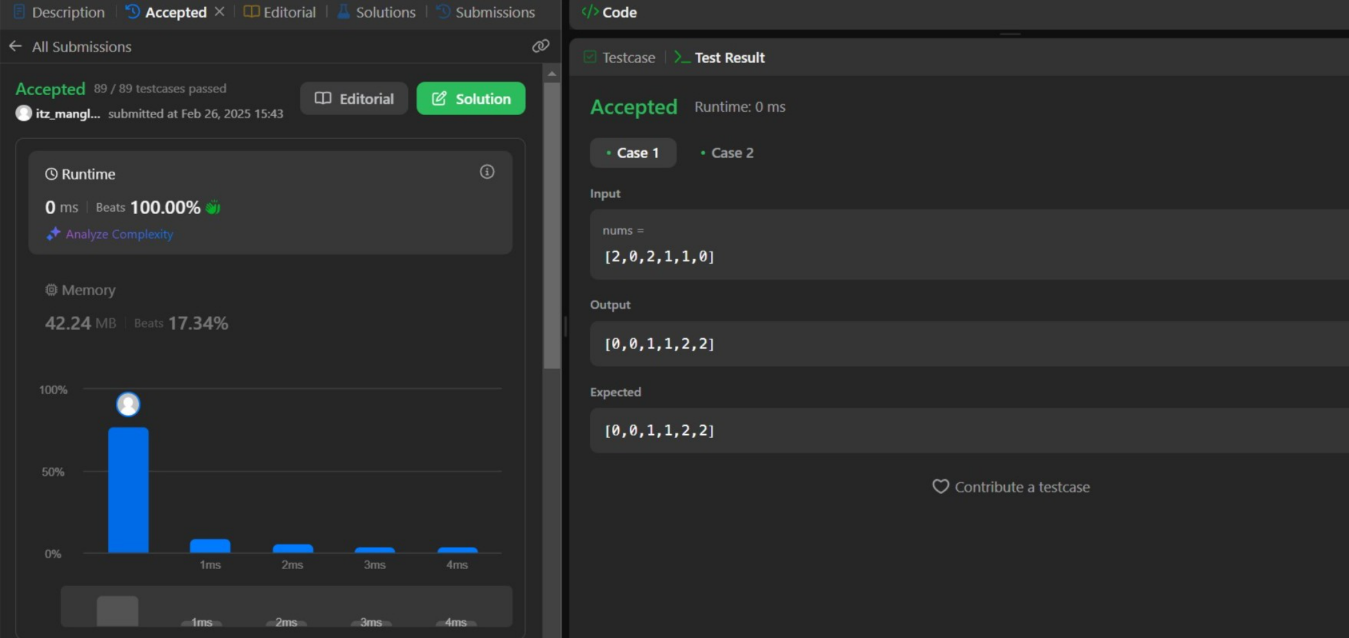
nums[i] = nums[j];

nums[j] = temp;

}

}

# Output :



1. **Top K Elements**

**Aim :**

To find the top K most frequent elements in an array.

**Objective :**

* Learn about frequency counting using hashmaps.
* Understand the use of priority queues and heaps.
* Implement optimized solutions using data structures like HashMap and Heap.

# Implementation/Code :

# import java.util.\*;

# class Solution {

# public int[] topKFrequent(int[] nums, int k) {

# // Step 1: Count frequencies using a HashMap

# Map<Integer, Integer> freqMap = new HashMap<>();

# for (int num : nums) {

# freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);

# }

# 

# // Step 2: Use a Min-Heap (Priority Queue) to store k most frequent elements

# PriorityQueue<Map.Entry<Integer, Integer>> minHeap = new PriorityQueue<>(

# (a, b) -> a.getValue() - b.getValue() // Sort by frequency (ascending)

# );

# for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {

# minHeap.offer(entry); // Add to heap

# if (minHeap.size() > k) {

# minHeap.poll(); // Remove the least frequent element if size exceeds k

# }

# }

# // Step 3: Extract k most frequent elements

# int[] result = new int[k];

# int index = 0;

# while (!minHeap.isEmpty()) {

# result[index++] = minHeap.poll().getKey();

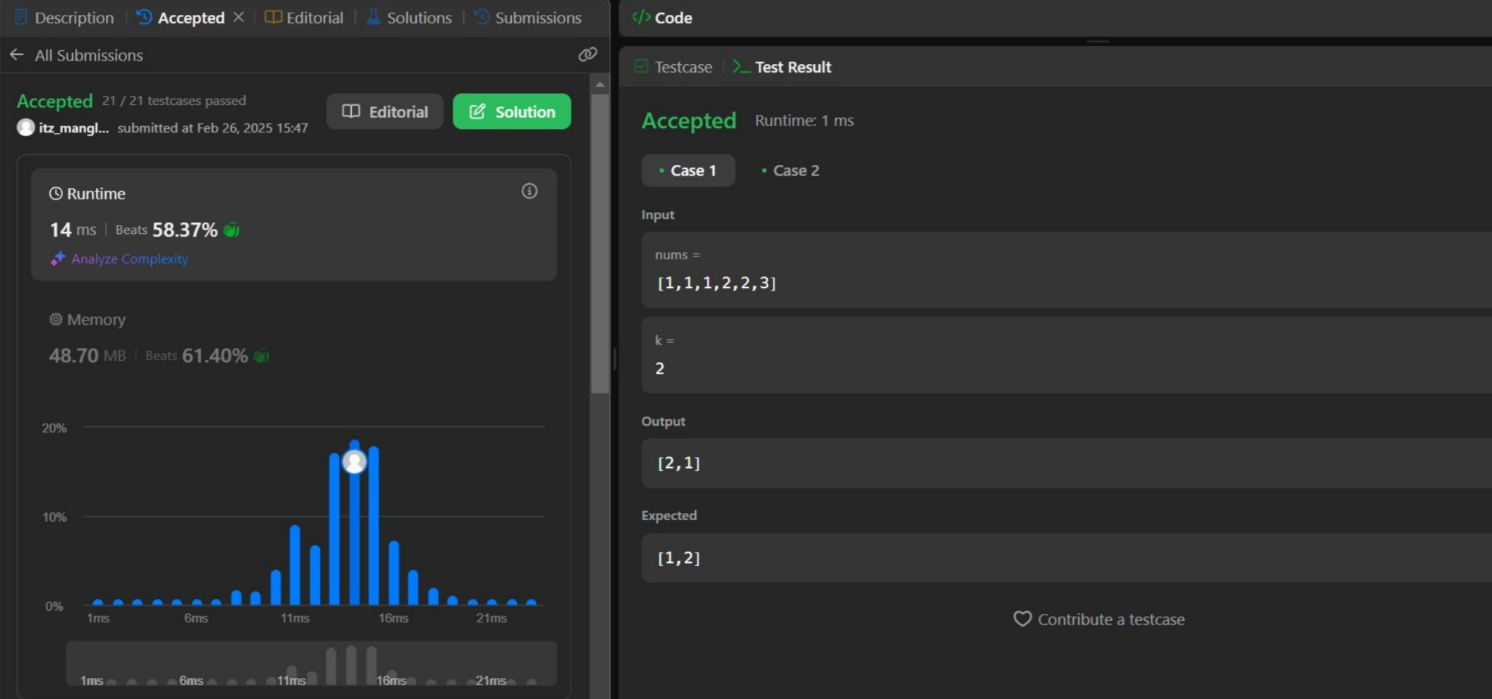
# }

# return result;

# }

# }

# Output :



1. **Kth Largest Element**

**Aim :**

# To find the Kth largest element in an unsorted array without sorting the entire array.

# Objectives :

# Understand the concept of order statistics.

# Implement solutions using heaps and Quickselect algorithms.

# Analyze and compare different approaches for time and space efficiency.

# Implementation/Code :

# import java.util.PriorityQueue;

# class Solution {

# public int findKthLargest(int[] nums, int k) {

# // Min-Heap with size k

# PriorityQueue<Integer> minHeap = new PriorityQueue<>();

# // Add elements to the heap

# for (int num : nums) {

# minHeap.offer(num); // Add current element to heap

# if (minHeap.size() > k) {

# minHeap.poll(); // Remove the smallest element if heap size exceeds k

# }

# }

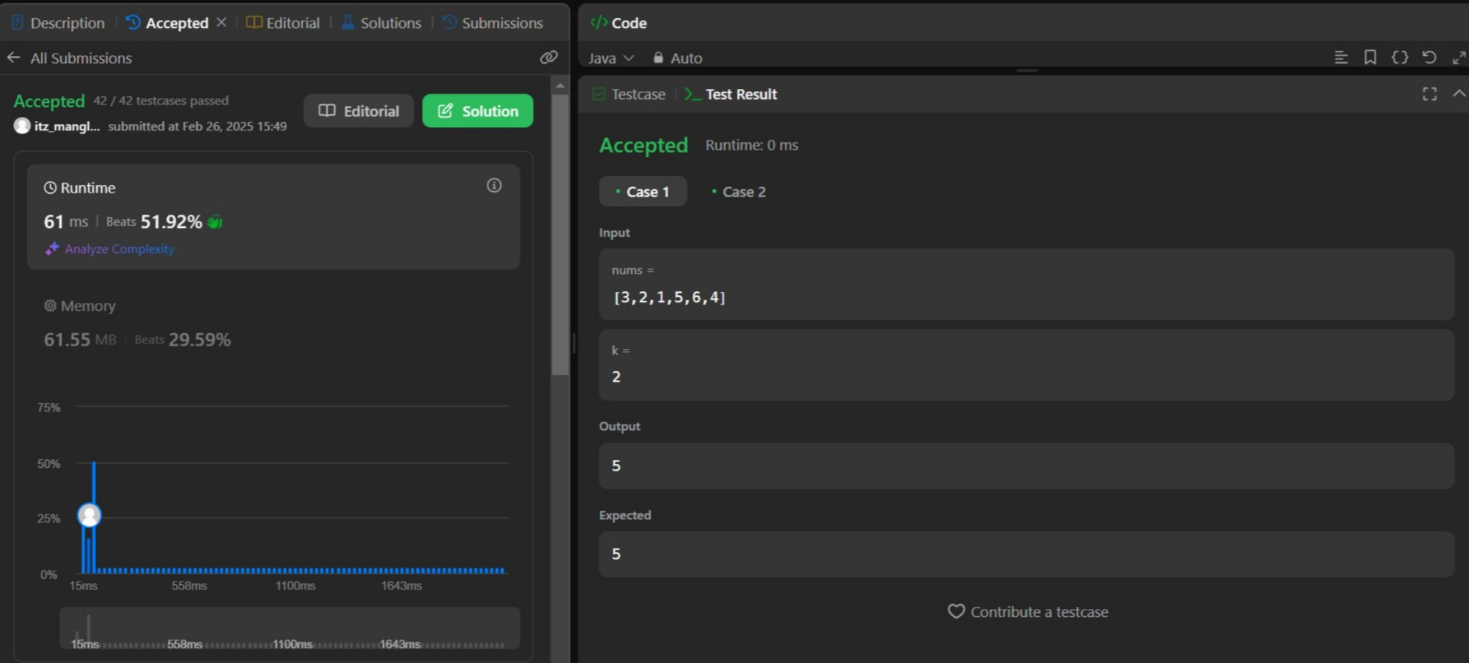
# // The root of the min-heap is the k-th largest element

# return minHeap.peek();

# }

# }

# Output :



**Learning Outcomes :**

* + Master heap-based solutions for selection problems.
  + Implement Quickselect for linear time complexity (on average).
  + Gain a deeper understanding of hashmap and heap operations.
  + Solve frequency-based problems efficiently.
  + Optimize space and time complexities.