Experiment 2.1

Student Name: Aditya Patel

UID: 22BCS11543

Branch: BE-CSE

Section/Group: 22BCS-IOT-640(B)

Semester: 6th

Date of Performance: 21/02/25

Subject Name: ADVANCED PROGRAMMING LAB - 2

Subject Code: 22CSP-351

PROGRAM-1

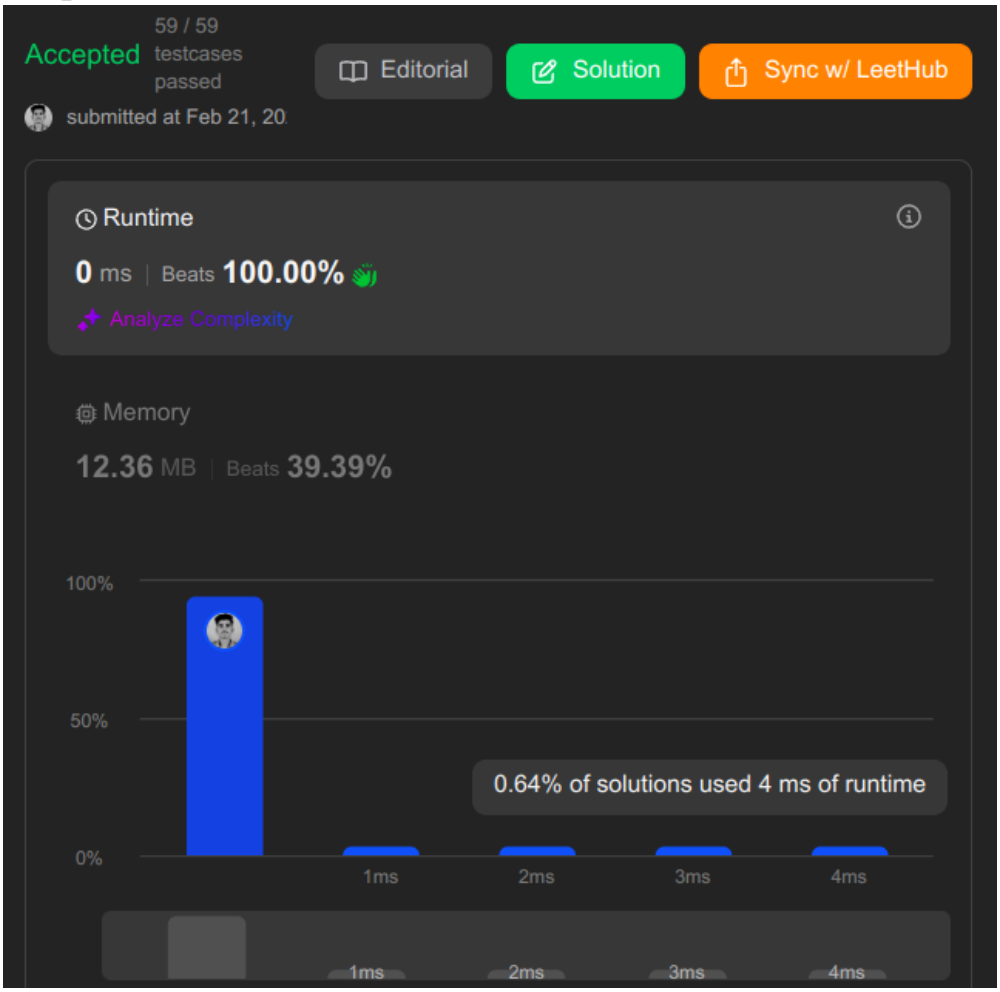
1) Aim: Merge Sorted Array.

2) Objective: The objective of the merge function is to combine two sorted arrays into a single sorted array while maintaining the sorted order. This is achieved using a two-pointer technique to efficiently merge elements from both arrays.

3) Implementation/Code:

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        for (int j = 0, i = m; j<n; j++){
            nums1[i] = nums2[j];
            i++;
        }
        sort(nums1.begin(),nums1.end());
    }
};
```

4) Output:



5) Learning Outcomes:

- **Array Manipulation:** Understand how to combine two sorted arrays into one by appending elements and using sorting methods.
- **Time Complexity Awareness:** Recognize the time complexity implications of sorting merged arrays compared to more efficient merging techniques.
- **Edge Case Handling:** Identify and manage scenarios where one or both arrays may be empty.
- **Efficiency Evaluation:** Compare simple implementations with optimized algorithms for merging sorted arrays.
- **Array Manipulation:** Understand how to combine two sorted arrays into one by appending elements and using sorting methods.

PROGRAM-2

1) Aim: Top K Frequent Elements.

2) Objective: The objective of the topKFrequent function is to identify and return the top K most frequent elements from a given vector of integers. The function utilizes a hash map to count the frequency of each element and a priority queue (max heap) to efficiently retrieve the K elements with the highest frequencies.

3) Implementation/Code:

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> ump;
        for(int i: nums){
            ump[i]++;
        }
        priority_queue<pair<int, int>>pq;
        for(auto i: ump){
            pq.push({i.second,i.first});
        }
        vector<int> res;
        while(k--){
            auto [elem, count] = pq.top();
            res.push_back(count);
            pq.pop();
        }
        return res;
    }
};
```

4) Output:

5) Learning Outcomes:

- **Understand Frequency Counting:** Learners will grasp how to count the occurrences of elements in a collection using hash maps (unordered maps).
- **Implement Priority Queues:** Participants will learn how to use priority queues (heaps) to efficiently manage and retrieve elements based on their frequency.
- **Develop Problem-Solving Skills:** Learners will enhance their ability to solve algorithmic problems involving frequency analysis and data structure manipulation.
- **Analyze Algorithm Efficiency:** Participants will evaluate the time and space complexity of their solutions, gaining insights into the performance characteristics of their implementations.

PROGRAM-3

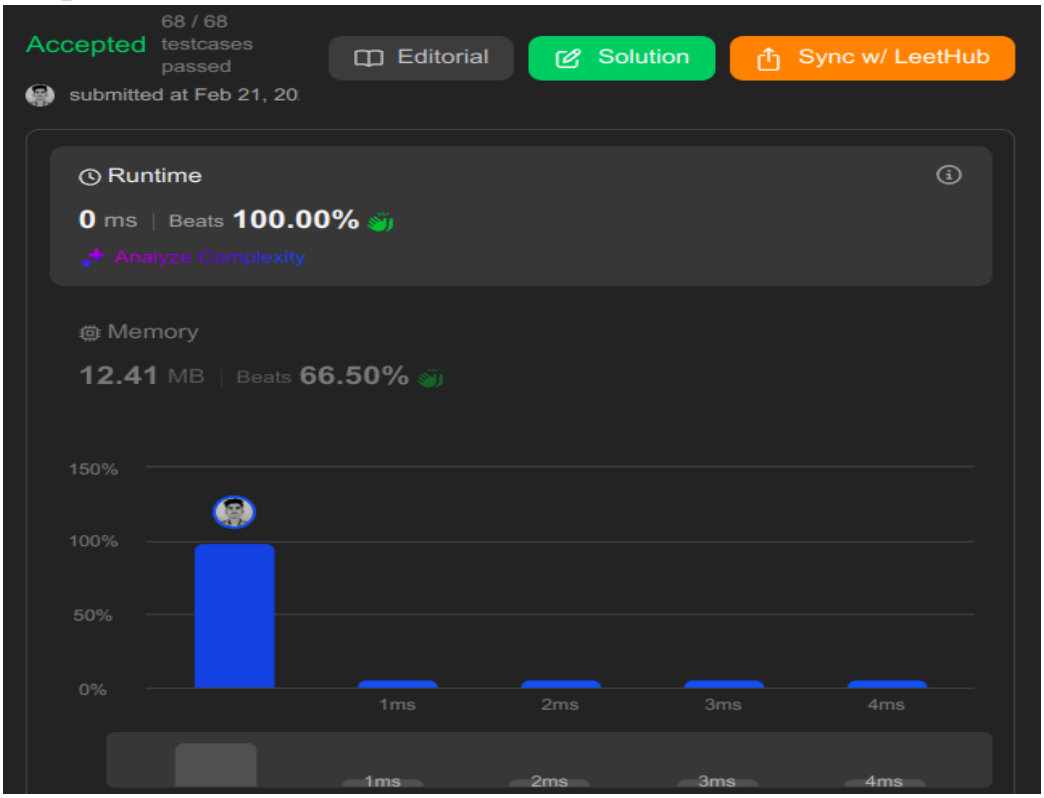
1) Aim: Find Peak Element.

2) Objective: The objective of the findPeakElement function is to identify a peak element in an array, where a peak is defined as an element that is greater than or equal to its neighbors.

3) Implementation/Code:

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int n = nums.size();
        int r = n-1;
        int l = 0;
        while(r>l){
            int mid = (r+l)/2;
            if(nums[mid] > nums[mid+1]) r = mid;
            else l = mid +1;
        }
        return l;
    }
};
```

4) Output:



5) Learning Outcomes:

- **Identify Peak Elements:** Learners will be able to identify and define peak elements within an array, recognizing their properties and significance.
- **Apply Binary Search Techniques:** Participants will develop the ability to implement binary search strategies to efficiently find peak elements, enhancing their problem-solving skills.
- **Evaluate Edge Cases:** Learners will understand how to handle edge cases, such as arrays with one element or arrays where all elements are the same, ensuring robustness in their solutions.
- **Optimize Algorithm Performance:** Participants will analyze and compare the performance of their algorithms, focusing on achieving optimal time complexity in finding peak elements.

PROGRAM-4

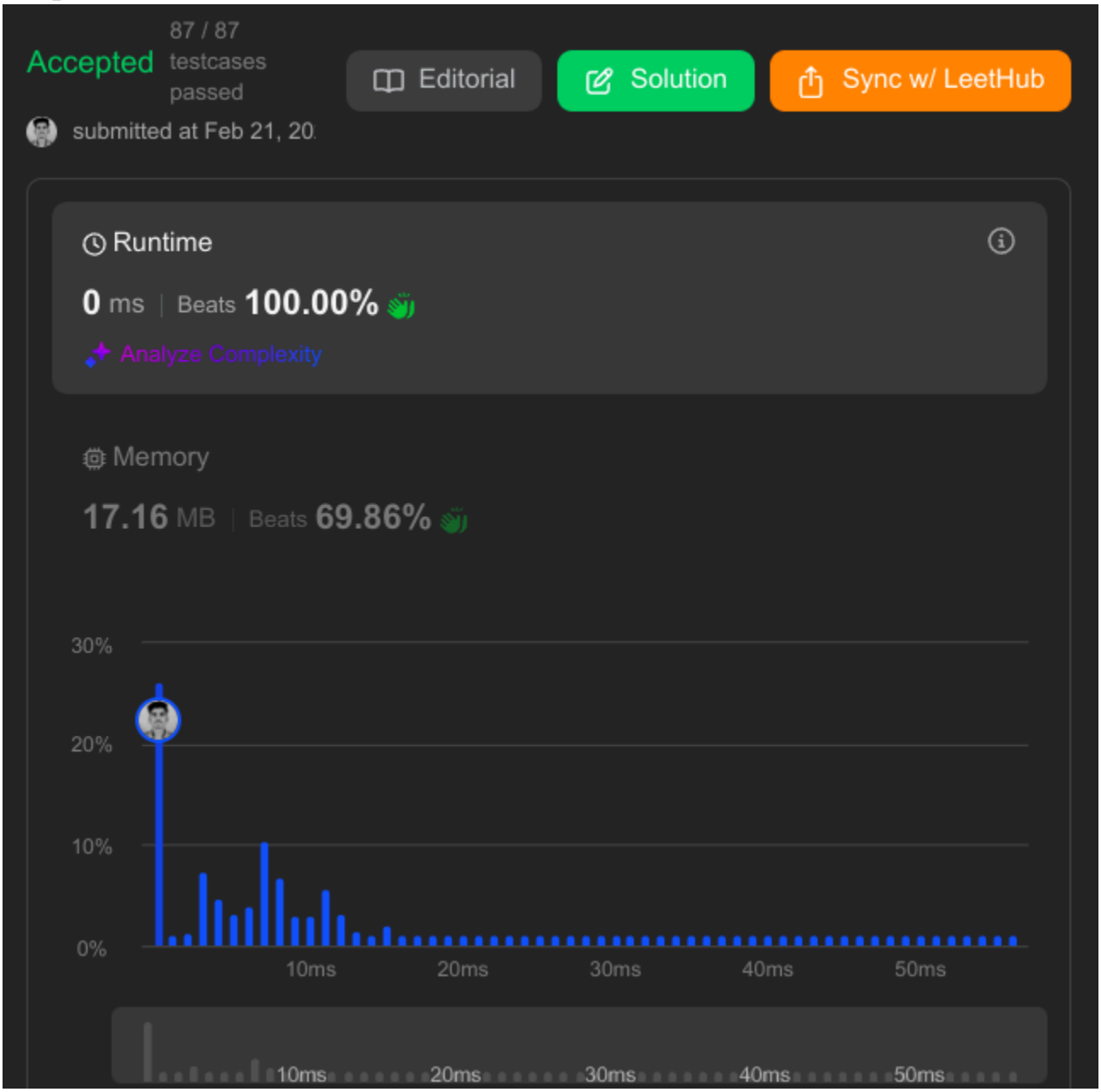
1) Aim: Kth Smallest Element in a Sorted Matrix.

2) Objective: The objective of the kthSmallest function is to find the k-th smallest element in a sorted 2D matrix, where each row and each column is sorted in ascending order. This is achieved using a binary search approach combined with a counting method.

3) Implementation/Code:

```
class Solution {
public:
    int m, n;
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        m = matrix.size(), n = matrix[0].size();
        int left = matrix[0][0], right = matrix[m-1][n-1], ans = -1;
        while (left <= right) {
            int mid = (left + right) >> 1;
            if (countLessOrEqual(matrix, mid) >= k) {
                ans = mid;
                right = mid - 1;
            } else left = mid + 1;
        }
        return ans;
    }
    int countLessOrEqual(vector<vector<int>>& matrix, int x) {
        int cnt = 0, c = n - 1;
        for (int r = 0; r < m; ++r) {
            while (c >= 0 && matrix[r][c] > x) --c;
            cnt += (c + 1);
        }
        return cnt;
    }
};
```

4) Output:



5) Learning Outcomes:

- **Implement Binary Search:** Learners will understand and implement a binary search algorithm to efficiently find the k-th smallest element in a sorted matrix.
- **Count Elements Less Than or Equal:** Participants will learn how to count the number of elements in the matrix that are less than or equal to a given value using a two-pointer technique.
- **Analyze Algorithm Efficiency:** Learners will analyze the time and space complexity of the binary search approach, enhancing their understanding of efficient searching techniques in sorted data structures.

PROGRAM-5

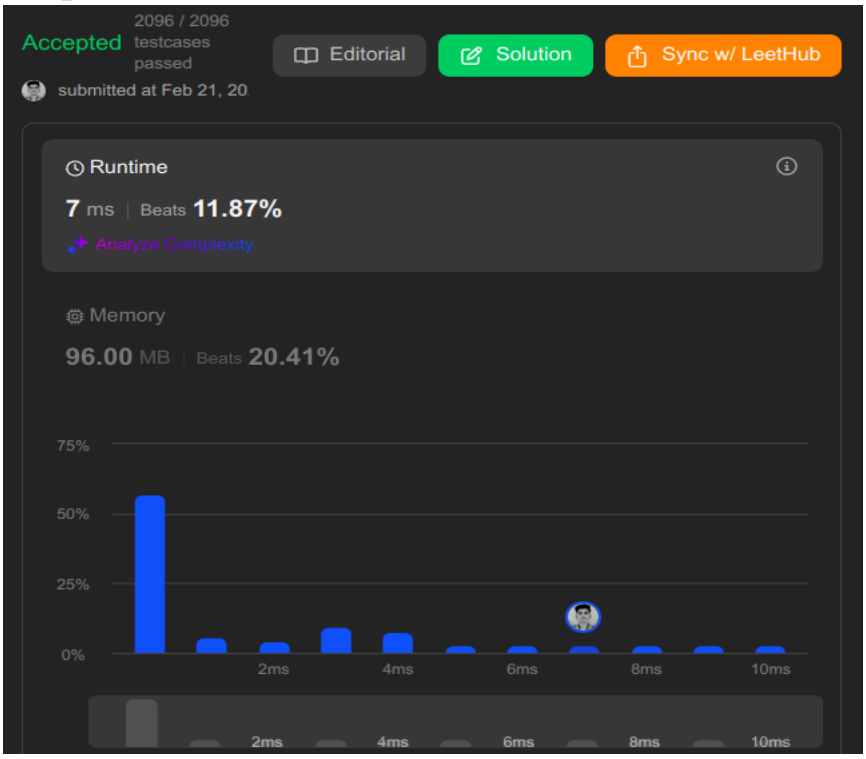
1) Aim: Median of Two Sorted Arrays.

2) Objective: The objective of the findMedianSortedArrays function is to find the median of two sorted arrays. The function combines the two arrays, sorts the combined array, and then calculates the median based on the size of the combined array.

3) Implementation/Code:

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        vector<int>v;
        for(auto num:nums1)
            v.push_back(num);
        for(auto num:nums2)
            v.push_back(num);
        sort(v.begin(),v.end());
        int n=v.size();
        return n%2?v[n/2]:(v[n/2-1]+v[n/2])/2.0;
    }
};
```

4) Output:



5) Learning Outcomes:

- **Combine and Sort Arrays:** Learners will understand how to merge two arrays and sort the resulting array to prepare for median calculation.
- **Calculate Median:** Participants will learn how to compute the median based on the size of the combined array, handling both even and odd cases.
- **Analyze Algorithm Efficiency:** Learners will analyze the time and space complexity of the approach, gaining insights into the efficiency of sorting and median finding in combined datasets.