Experiment 4

Student Name: Shivansh Singh **UID: 22BET10105**
Branch: IT **Section/Group: 22BET_IOT-701/A**
Semester: 6th **Date of Performance:21.02.25**
Subject Name: AP Lab - 2 **Subject Code: 22ITP-351**

1. **Aim:** To find the top K most frequent elements in a given array using efficient data structures like hash maps and heaps.
   i.) Find Peak Element
   ii.) Merge Intervals
   iii.) Search in Rotated Sorted Array
   iv.) Search a 2D Matrix II
   v.) Kth smallest element in a sorted matrix
   vi.) Median of Two Sorted Arrays
   vii.) Top K frequent elements

2. **Objective:**

- Understand and implement efficient searching and sorting algorithms.
- Improve proficiency in binary search and divide & conquer techniques.
- Solve problems related to peak finding, searching in rotated arrays, and 2D matrices.
- Enhance knowledge of interval merging and matrix-based problem-solving.
- Optimize code for time and space efficiency in searching and merging operations.
- Strengthen logical reasoning and debugging skills for complex search-based problems.
- Gain hands-on experience with problem-solving on LeetCode.

3. **Code:**

Problem 1: Find Peak Element

```
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]) {
  // Peak is on the left or at mid
    right = mid;
```

```
        } else {
  // Peak is on the right
                left = mid + 1;
            }
           }
           return left; // or return right, both are the same
       }
 };
```

Problem 2: Merge Intervals

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if (intervals.empty()) return {}; // Edge case
        // Step 1: Sort intervals by the start time
        sort(intervals.begin(), intervals.end());
        vector<vector<int>> merged;
        merged.push_back(intervals[0]); // Add the first interval
        for (int i = 1; i < intervals.size(); i++) {
            // Get the last interval in merged
            vector<int>& last = merged.back();
            // Check if there is an overlap
            if (intervals[i][0] <= last[1]) {
                // Merge intervals by updating the end time
                last[1] = max(last[1], intervals[i][1]);
            } else {
                // No overlap, add the current interval
                merged.push_back(intervals[i]);
            }
        }
        return merged;
    }
};
```

Problem 3: Search in Rotated Sorted Array

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            // If target found, return index
            if (nums[mid] == target) return mid;
            // Determine which half is sorted
            if (nums[left] <= nums[mid]) { // Left half is sorted
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1; // Search in left half
                } else {
                    left = mid + 1;  // Search in right half
                }
            }
```

```
                else { // Right half is sorted
                if (nums[mid] < target && target <= nums[right])
                left = mid + 1;  // Search in right half
                } else {
                    right = mid - 1; // Search in left half
                }
            }
        }
        return -1; // Target not found
    }
};
```

Problem 4: Search a 2D Matrix II

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size(), n = matrix[0].size();
        int row = 0, col = n - 1; // Start from the top-right
        while (row < m && col >= 0) {
            if (matrix[row][col] == target) return true;
            else if (matrix[row][col] > target) col--;  // Move left
            else row++;  // Move down
        }
        return false; // Target not found
    }
};
```

Problem 5: Kth smallest element in a sorted matrix

```
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int n = matrix.size();
        priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, greater<>> minHeap;
        // Push first column elements into minHeap
        for (int i = 0; i < n; i++) {
            minHeap.emplace(matrix[i][0], i, 0);
        }
        // Extract k times
        int result = 0;
        for (int i = 0; i < k; i++) {
            auto [val, row, col] = minHeap.top();
            minHeap.pop();
            result = val;
            // Push the next element in the row if it exists
            if (col + 1 < n) {
                minHeap.emplace(matrix[row][col + 1], row, col + 1);
            }
        }
```

```
        return result;
    }
};
```

Problem 6: Median of Two Sorted Arrays

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int m = nums1.size(), n = nums2.size();
        // Ensure nums1 is the smaller array for binary search optimization
        if (m > n) return findMedianSortedArrays(nums2, nums1);
        int low = 0, high = m, halfLen = (m + n + 1) / 2;
        while (low <= high) {
            int i = (low + high) / 2;
            int j = halfLen - i;
            // Handle edge cases where partition goes out of bounds
            int L1 = (i > 0) ? nums1[i - 1] : INT_MIN;
            int R1 = (i < m) ? nums1[i] : INT_MAX;
            int L2 = (j > 0) ? nums2[j - 1] : INT_MIN;
            int R2 = (j < n) ? nums2[j] : INT_MAX;
            if (L1 <= R2 && L2 <= R1) {
                // Found the correct partition
                if ((m + n) % 2 == 0) {
                    return (max(L1, L2) + min(R1, R2)) / 2.0;
                } else {
                    return max(L1, L2);
                }
            } else if (L1 > R2) {
                high = i - 1;  // Move left
            } else {
                low = i + 1;   // Move right
            }
        }
        return 0.0; // Should never reach here
    }
};
```

Problem 7: Top K frequent elements

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> freqMap;
        int n = nums.size();
        // Step 1: Count frequencies
        for (int num : nums) {
            freqMap[num]++;
        }
        // Step 2: Create frequency buckets
        vector<vector<int>> buckets(n + 1);
```

```
        for (auto& pair : freqMap) {
            int num = pair.first, freq = pair.second;

        buckets[freq].push_back(num);
        }
        // Step 3: Collect top k elements
        vector<int> result;
        for (int i = n; i >= 0 && result.size() < k; --i) {
            for (int num : buckets[i]) {
                result.push_back(num);
                if (result.size() == k) return result;
            }
        }
        return result;
    }
};
```

4. Output:

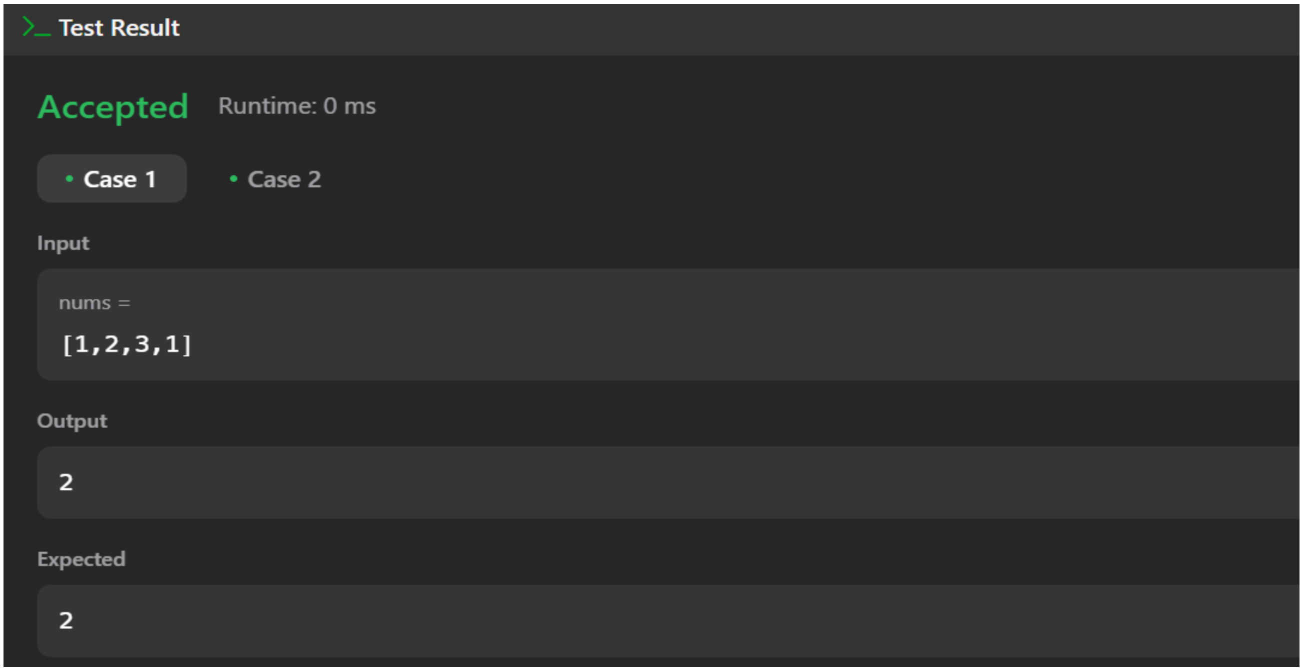

Fig 1. Find Peak Element

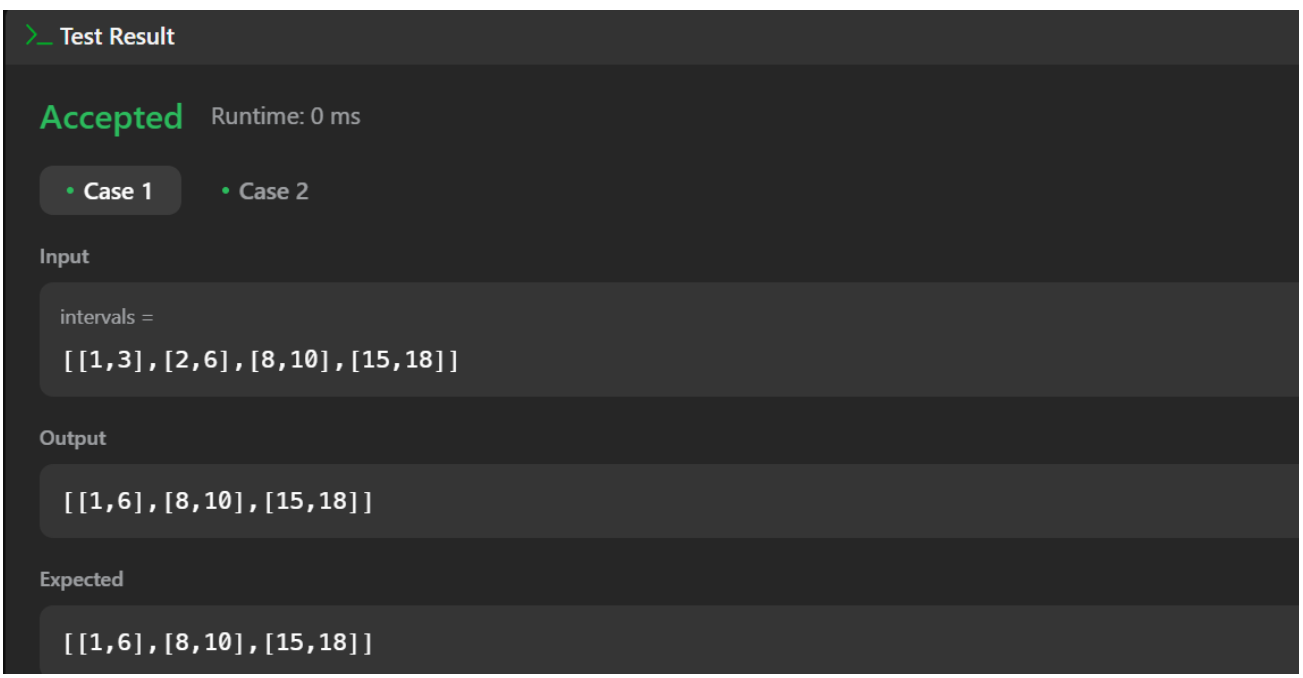

Fig 2. Merge Intervals

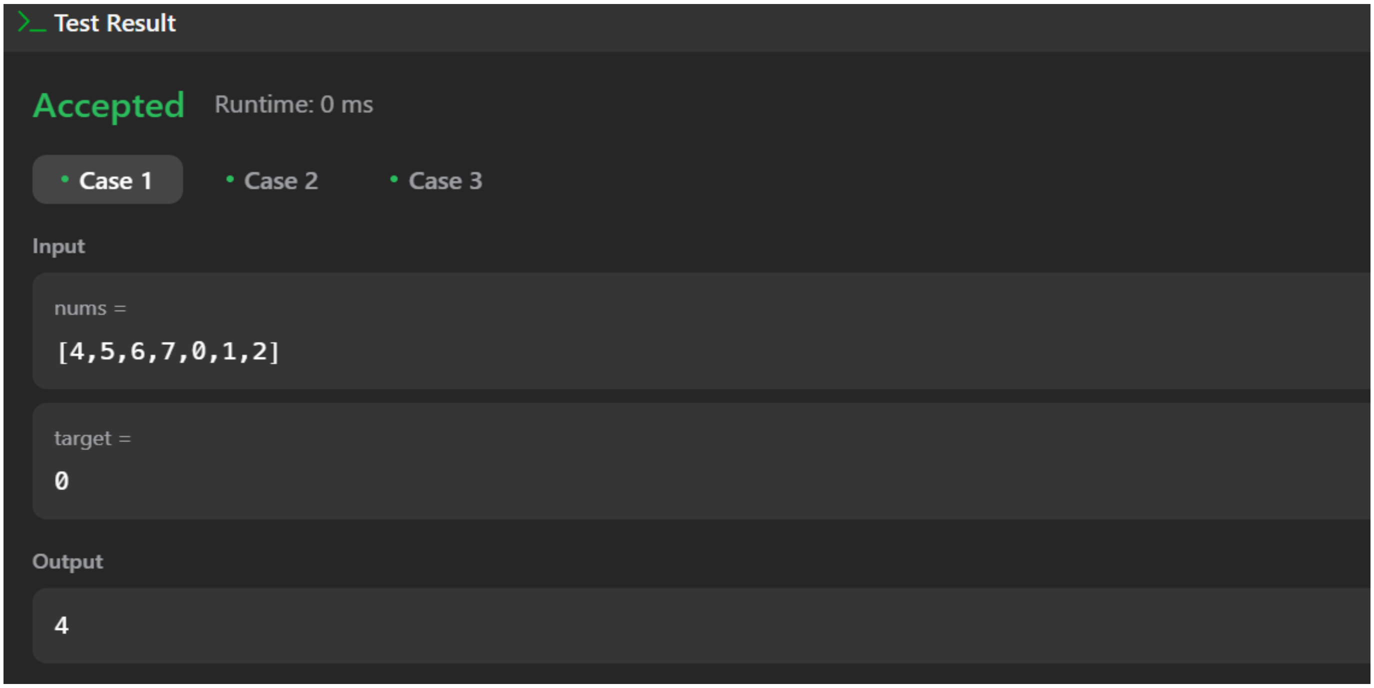

Fig 3. Search in Rotated Sorted Array

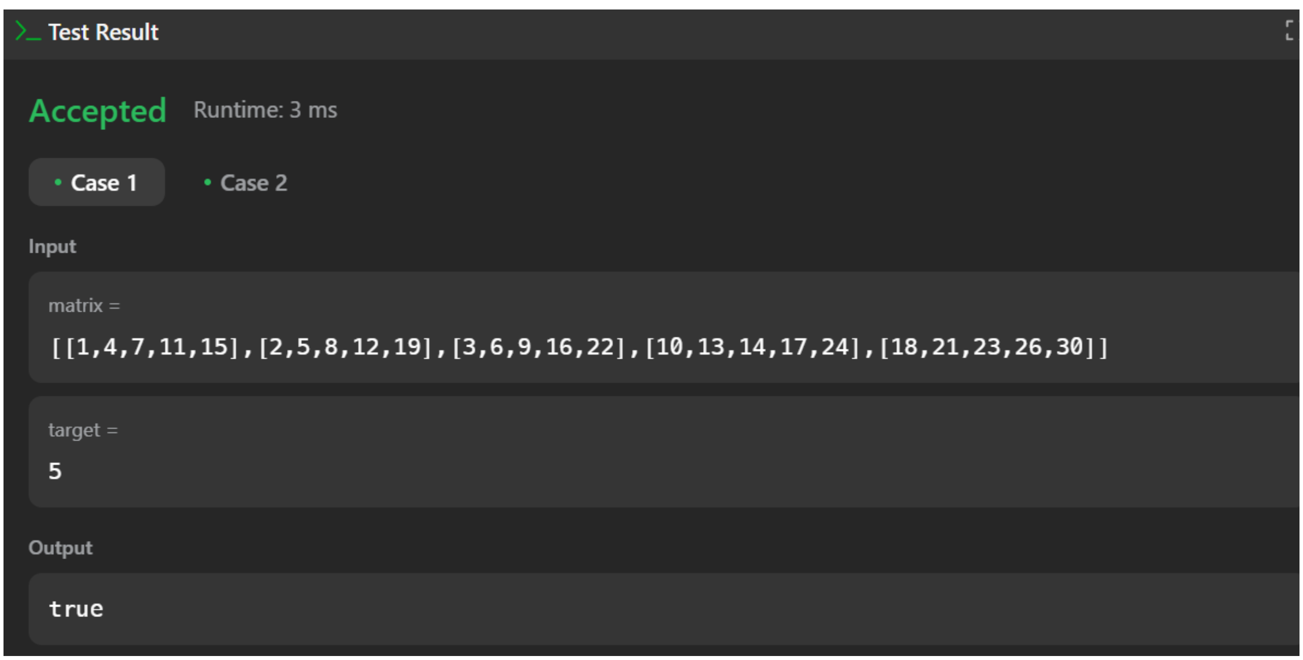

Fig 4. Search a 2D Matrix II

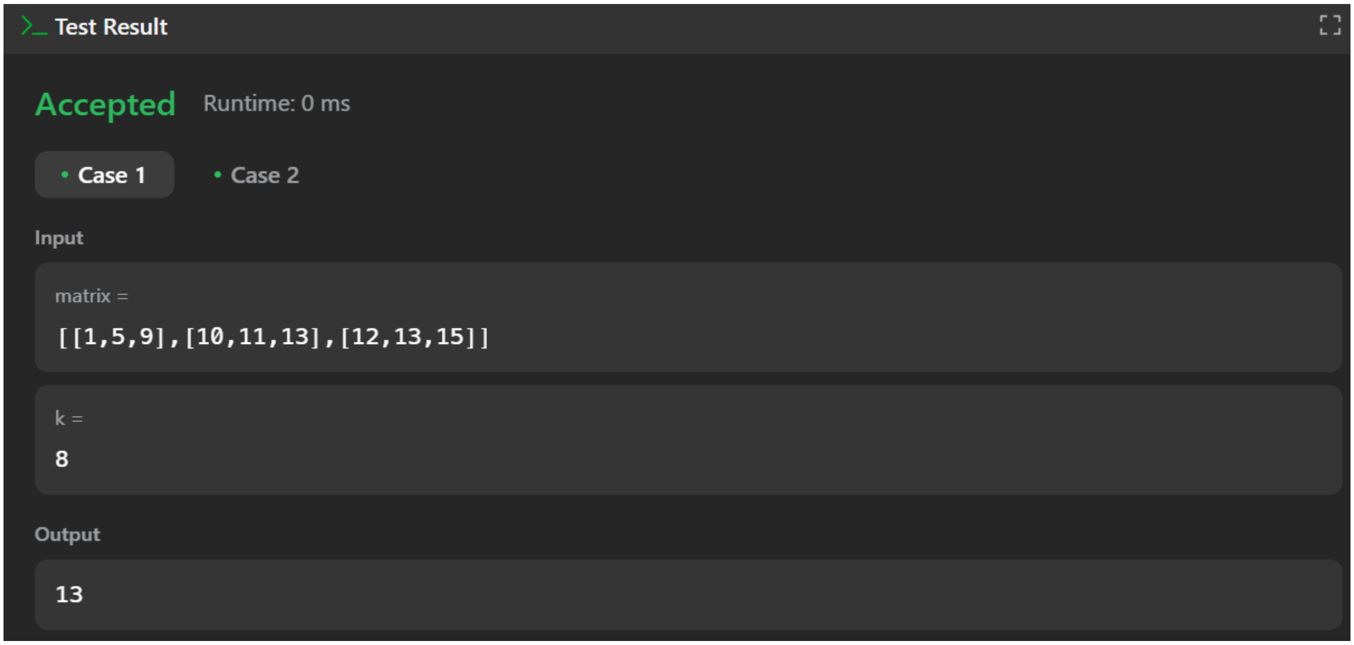

Fig 5. Kth smallest element in a sorted matrix

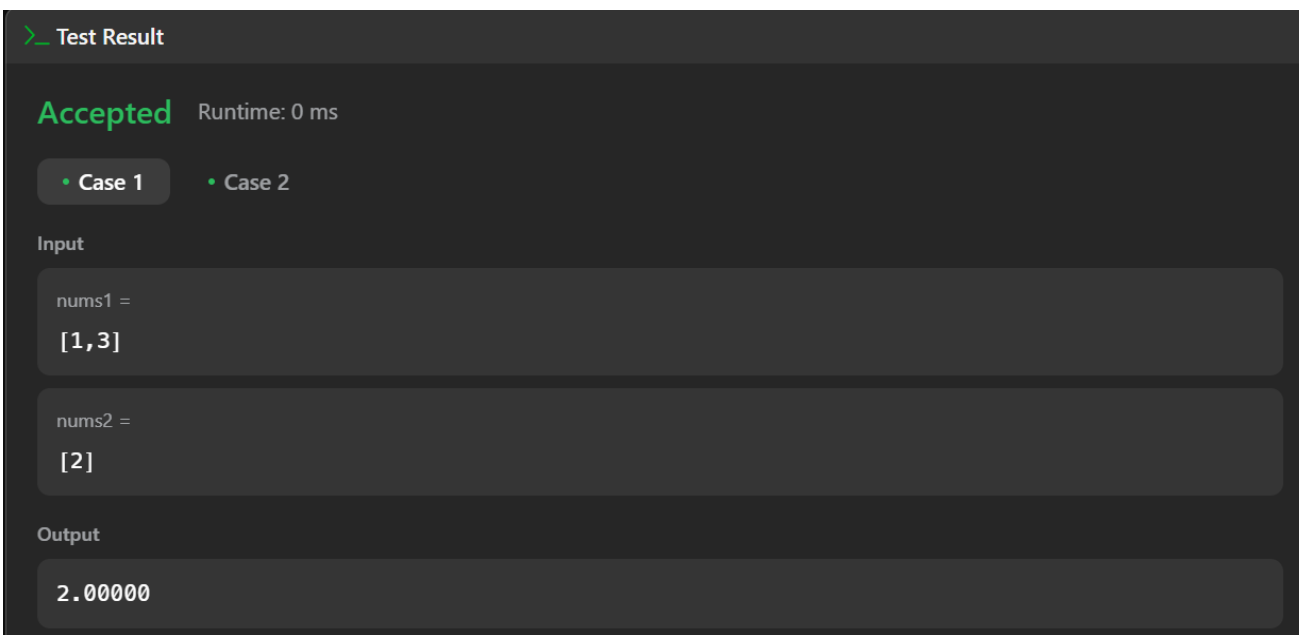

Fig 6. Median of Two Sorted Arrays

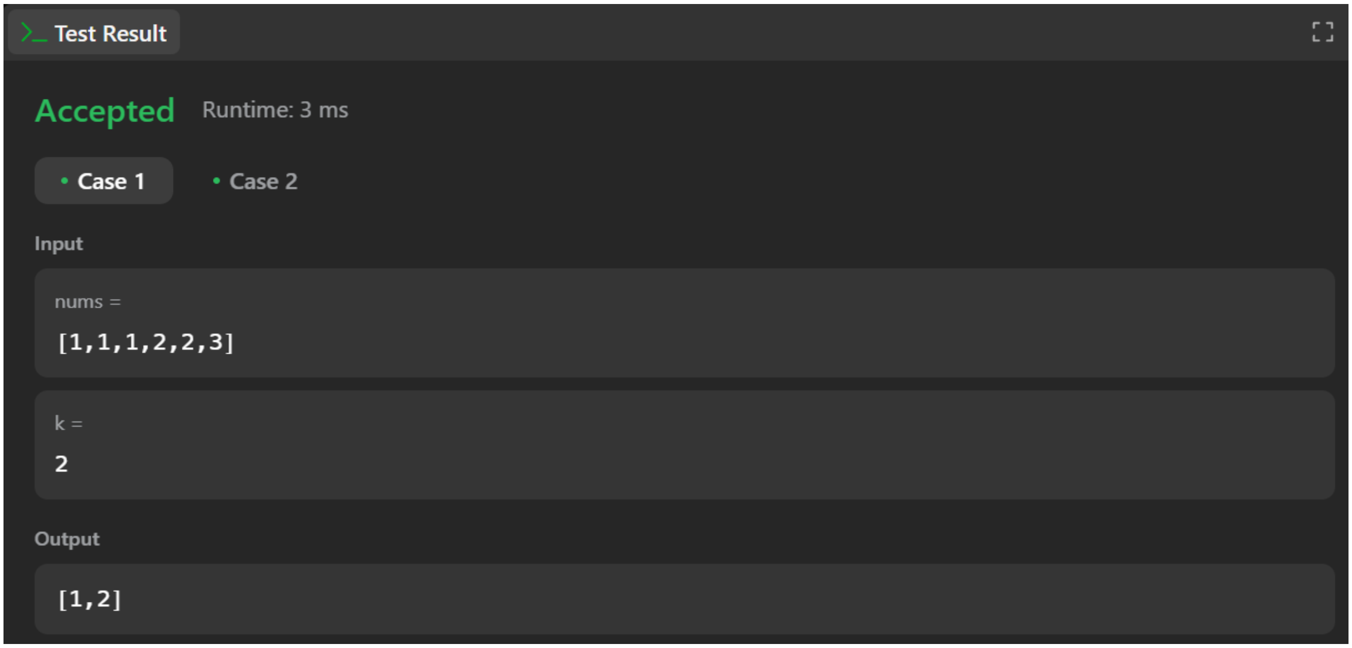

Fig 7. Top K frequent elements

5. Learning Outcomes:

- Understand and apply binary search techniques in problems like searching in a rotated sorted array and a 2D matrix.
- Master divide and conquer approach for solving problems such as finding the median of two sorted arrays.
- Enhance problem-solving skills in sorting and searching for efficiently handling sorted data structures.
- Improve understanding of greedy and sorting-based algorithms in problems like merging overlapping intervals.
- Strengthen knowledge of heap and hash map data structures for solving frequency-based problems like top K frequent elements.
- Optimize time complexity for large datasets by implementing efficient algorithms with logarithmic or linearithmic complexity.
- Develop critical thinking for real-world applications such as data processing, search engines, and recommendation systems.

6. Learning Outcomes:

- Gained expertise in linked lists and advanced data structures, including Fenwick Trees, Segment Trees, and Heaps.
- Enhanced proficiency in searching and sorting algorithms, utilizing binary search and merge sort-based counting techniques.
- Developed a strong understanding of bit manipulation for optimizing operations on binary representations.
- Applied modular arithmetic principles to efficiently manage large computations in problems like **Super Pow**.
- Tackled intricate computational geometry challenges using priority queues and sweep line algorithms.