EXPERIMENT 4

Student Name: Harshit Shandilya	**UID:** 22BET10274
Branch: BE-IT	**Section/Group:** BET-701-A
Semester: 6	**Date of Performance:**21.02.2025
Subject Name: Advanced Programming	**Subject Code:** 22ITP-351

Aim:

Merge Sorted Array

Source code:

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int i = m - 1;  // Last element in the valid part of nums1
        int j = n - 1;  // Last element in nums2
        int k = m + n - 1;  // Last position in nums1 (extended array)

        // Merge nums1 and nums2 from the end to the front
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
                nums1[k] = nums1[i];
                i--;
            } else {
                nums1[k] = nums2[j];
                j--;
            }
            k--;
        }

        // If there are remaining elements in nums2, copy them over to nums1
        while (j >= 0) {
            nums1[k] = nums2[j];
            j--;
```

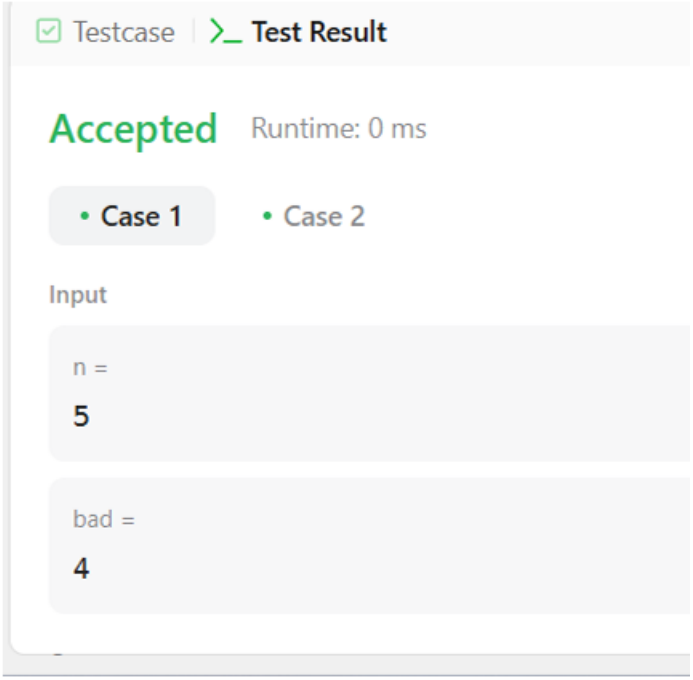
```
            k--;
        }
    }
};
```

Aim: First Bad Version

CODE:

```
class Solution {
public:
    int firstBadVersion(int n) {
        int left = 1, right = n;
        while (left < right) {
            int mid = left + (right - left) / 2; // Calculate mid safely
            if (isBadVersion(mid)) {
                right = mid; // If mid is bad, the first bad version is at mid or earlier
            } else {
                left = mid + 1; // If mid is good, the first bad version is later
            }
        }
```

```
            return left;
        }
    };
```

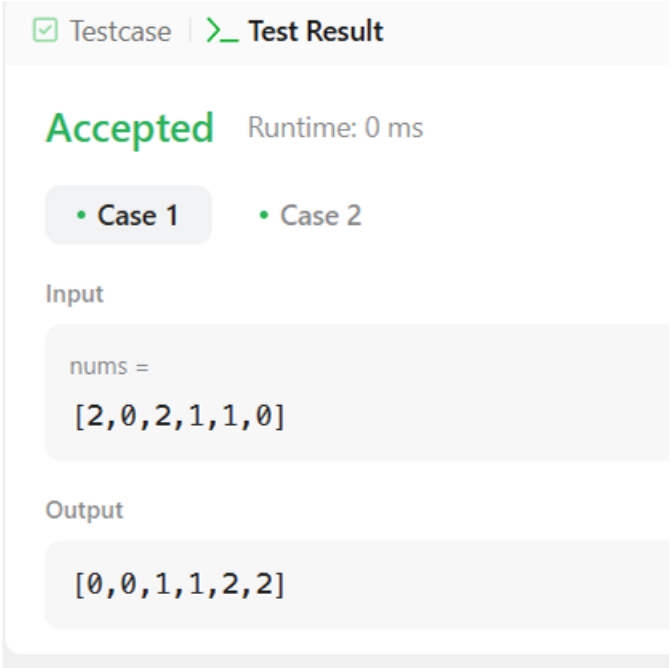


AIM: Sort Colors

CODE:

```
#include <vector>
#include <algorithm> // For swap

using namespace std;

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 0 to the left part
                swap(nums[low], nums[mid]);
```

```
            low++;
            mid++;
        } else if (nums[mid] == 1) {
            // Move mid pointer forward
            mid++;
        } else {
            // Swap 2 to the right part
            swap(nums[mid], nums[high]);
            high--;
        }
    }
  }
};
```

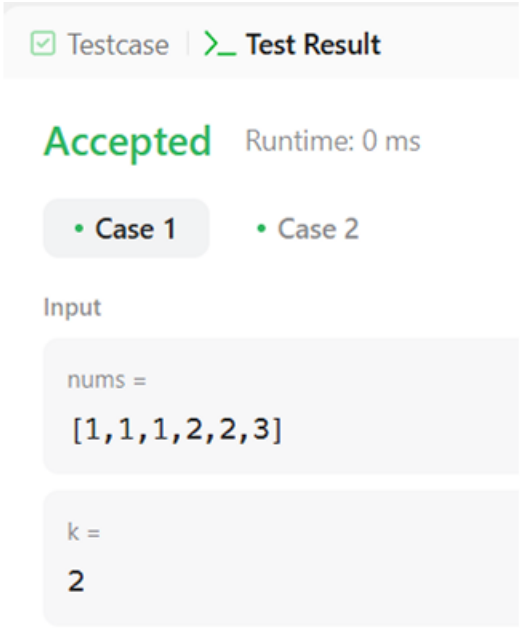


AIM: Top K frequent elements

CODE:

```
#include <vector>
#include <unordered_map>
#include <queue>

using namespace std;

class Solution {
```

```
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> freqMap;
        for (int num : nums) {
            freqMap[num]++;
        }

        // Min-heap based on frequency
        auto cmp = [](pair<int, int>& a, pair<int, int>& b) {
            return a.second < b.second; // Corrected comparison for min-heap
        };
        priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp)>
minHeap(cmp);

        // Add elements to the heap
        for (auto& entry : freqMap) {
            minHeap.push(entry);
            if (minHeap.size() > k) {
                minHeap.pop(); // Remove element with the smallest frequency
            }
        }

        // Extract the k most frequent elements
        vector<int> result;
        while (!minHeap.empty()) {
            result.push_back(minHeap.top().first);
            minHeap.pop();
        }

        return result;
    }
};
```

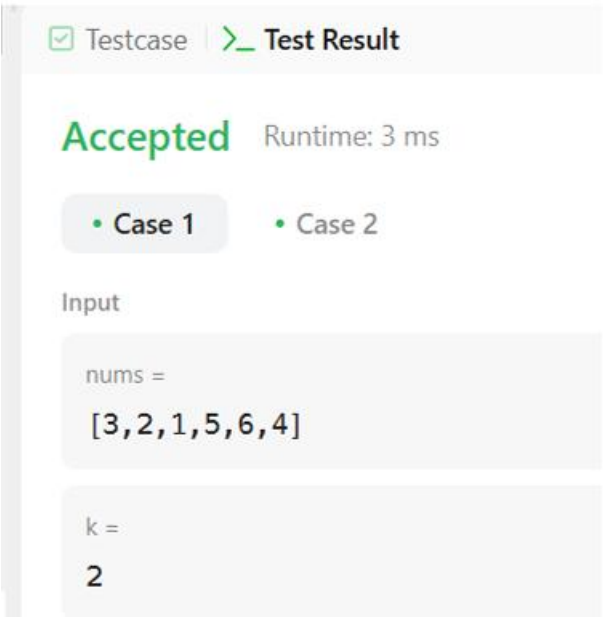


AIM: Kth Largest element in an array

CODE:

```
 #include <vector>
#include <queue>

using namespace std;

class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        priority_queue<int, vector<int>, greater<int>> minHeap;

        // Build a min-heap of size k
        for (int num : nums) {
            minHeap.push(num);
            if (minHeap.size() > k) {
                minHeap.pop(); // Remove the smallest element
            }
        }

        // The root of the min-heap is the kth largest element
        return minHeap.top();
    }
```

};

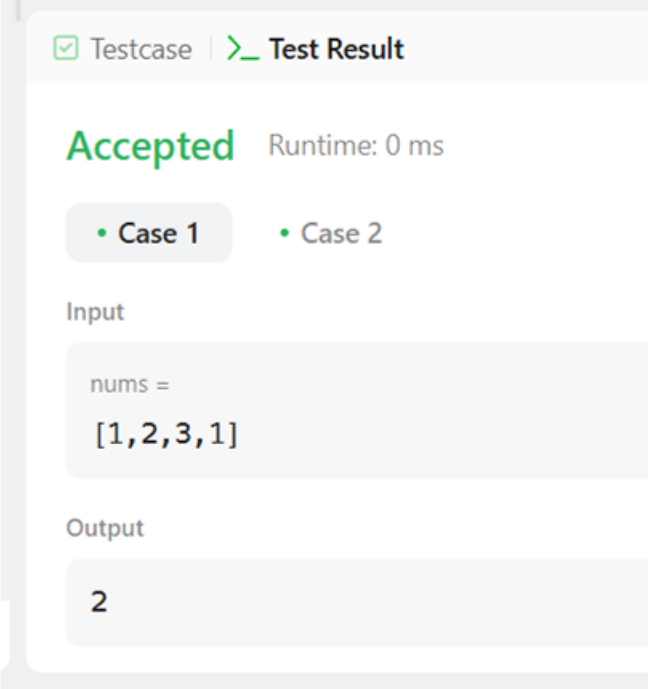


AIM: Find Peak Element

CODE:

```
#include <vector>

using namespace std;

class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;

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

            // Compare mid with its next element to decide the direction
            if (nums[mid] < nums[mid + 1]) {
                // Peak must be on the right half
                left = mid + 1;
            } else {
                // Peak must be on the left half or at mid itself
                right = mid;
            }
```

```
        }

        // At the end of the loop, left == right, which is the peak index
        return left;
    }
};
```

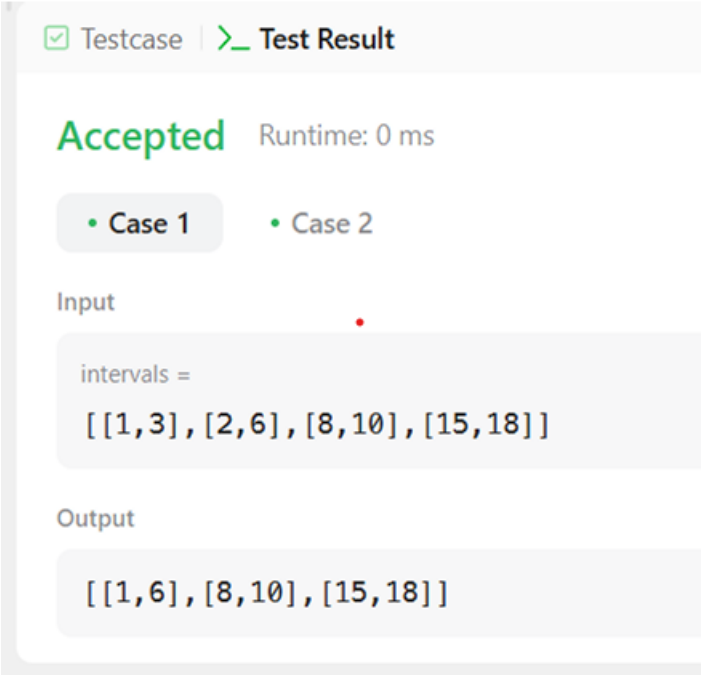


AIM: Merge Intervals

CODE:

```
#include <vector>
#include <algorithm>

using namespace std;

class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end());

        vector<vector<int>> merged;
        for (const auto& interval : intervals) {
            if (merged.empty() || merged.back()[1] < interval[0]) {
```

```
            merged.push_back(interval);
        } else {
            merged.back()[1] = max(merged.back()[1], interval[1]);
        }
    }
    return merged;
  }
};
```

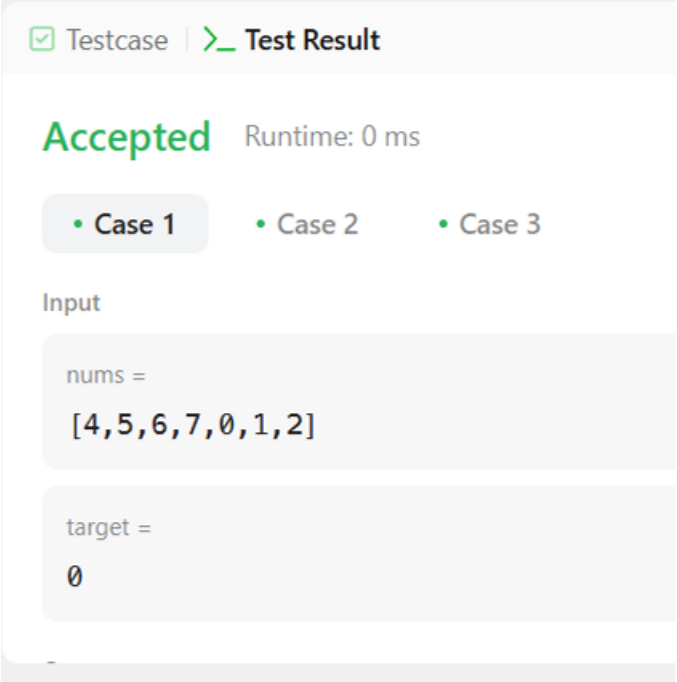


AIM: Search in Rotated Sorted Array

CODE:

```
#include <vector>

using namespace std;

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 (nums[mid] == target) {
                return mid;
            }

            if (nums[left] <= nums[mid]) {
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else {
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        return -1;
    }
};
```



AIM: Search a 2D Matrix II

CODE:

```
#include <vector>

using namespace std;

class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size();
        int n = matrix[0].size();

        int row = 0;
        int col = n - 1;

        while (row < m && col >= 0) {
            if (matrix[row][col] == target) {
                return true;
            } else if (matrix[row][col] > target) {
                col--;
            } else {
```

```
            row++;
          }
        }

        return false;
    }
};
```