Experiment 5

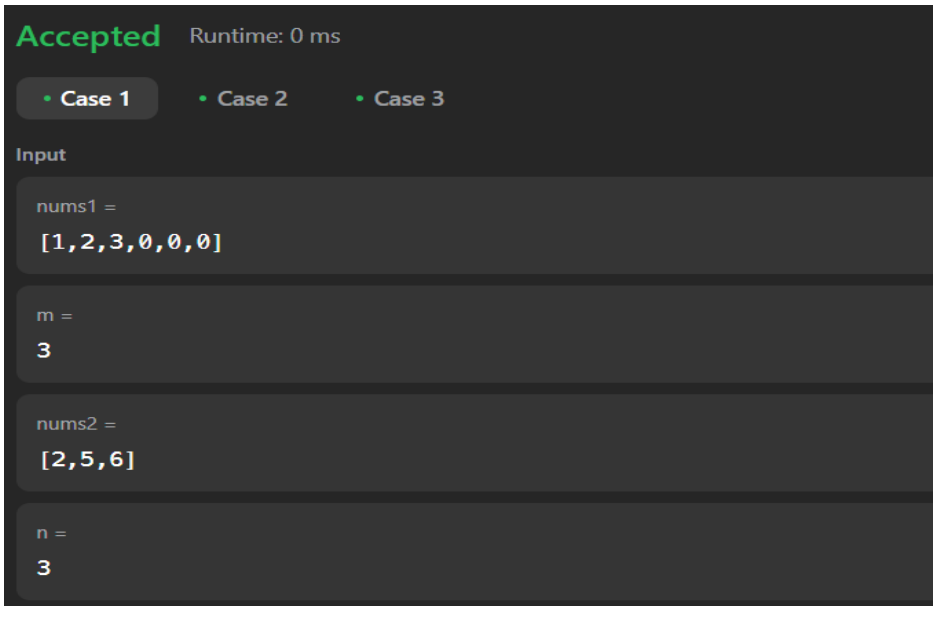
Name: Manik Naharia
UID: 22BET10004
Branch: IT
Section: 22BET_IOT-701/A
Semester: 6th
Date of Performance: 21/02/25
Subject: Advanced Programming - 2
Subject Code: 22ITP-351

Problem 1. You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively. Merge nums1 and nums2 into a single array sorted in non-decreasing order.

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());
    }
};
```

Output:

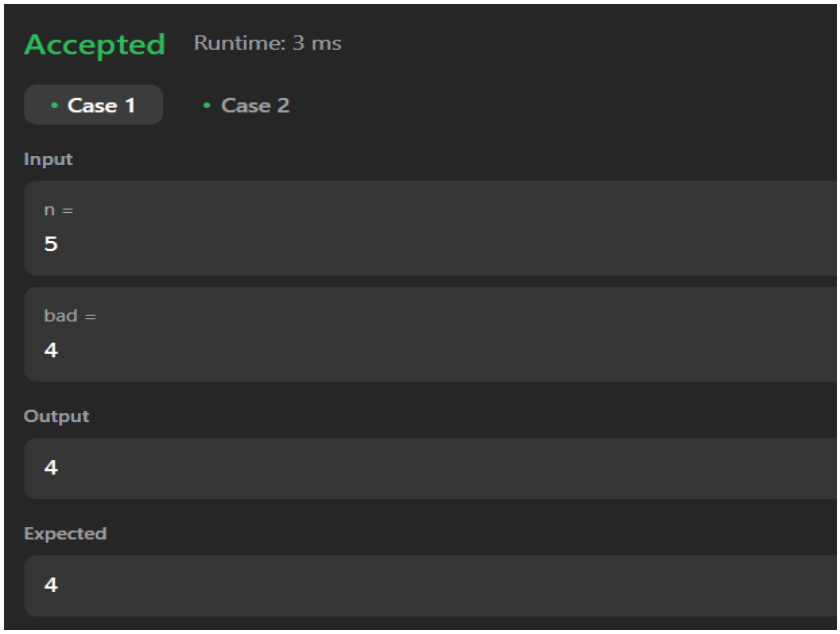


Problem 2. You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Code:

```
class Solution {
public:
    int firstBadVersion(int n) {
        int first = 1;
        int last = n;
        while (first < last) {
            int mid = first + (last - first) / 2;

            if (isBadVersion(mid)) {
                last = mid;
            } else {
                first = mid + 1;
            }
        }
        return first;
    }
};
```

Output:

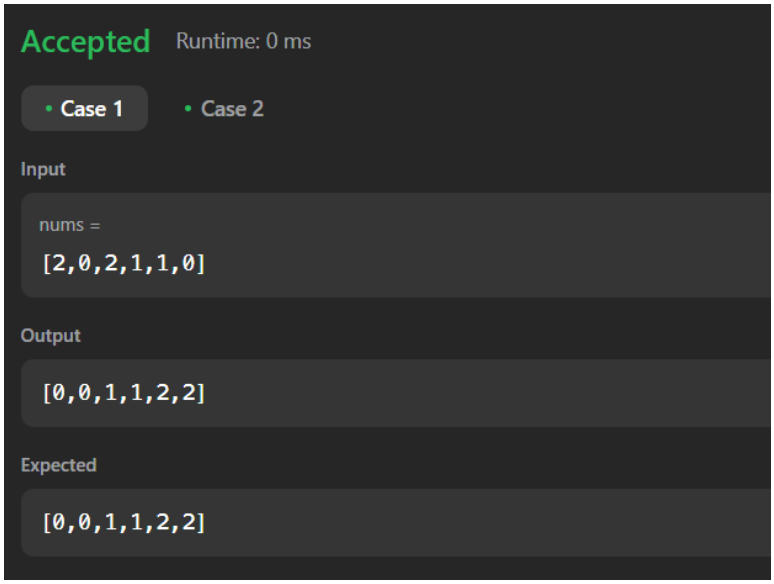


Problem 3. Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

Code:

```
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]);
                low++;
                mid++;
            }
            else if(nums[mid] == 1){
                mid++;
            }
            else{
                swap(nums[mid], nums[high]);
                high--;
            }
        }
    }
};
```

Output:

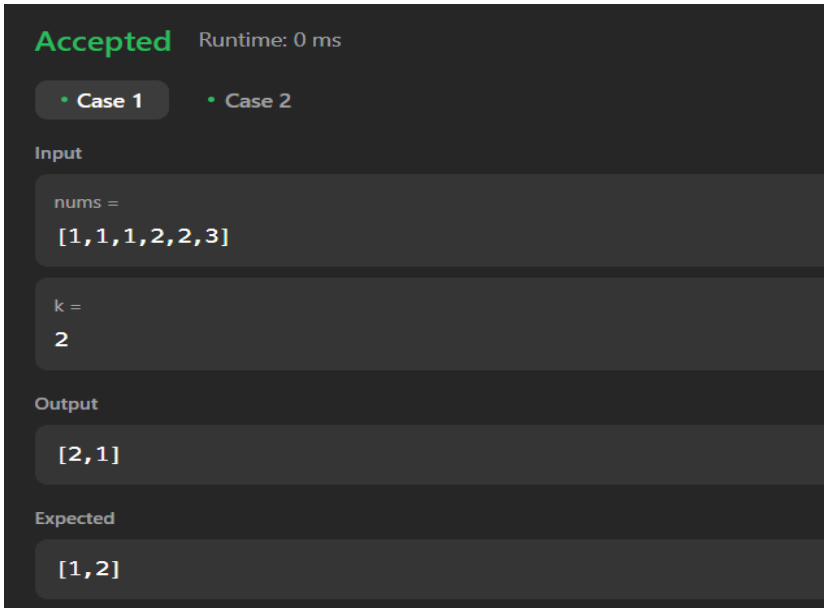


Problem 4. Given an integer array nums and an integer k, return *the* k *most frequent elements*. You may return the answer in **any order**.

Code:

```
struct T {
  int num;
  int freq;
  T(int num, int freq) : num(num), freq(freq) {}
};

class Solution {
 public:
  vector<int> topKFrequent(vector<int>& nums, int k) {
    const int n = nums.size();
    vector<int> ans;
    unordered_map<int, int> count;
    auto compare = [](const T& a, const T& b) { return a.freq > b.freq; };
    priority_queue<T, vector<T>, decltype(compare)> minHeap(compare);

    for (const int num : nums)
      ++count[num];

    for (const auto& [num, freq] : count) {
      minHeap.emplace(num, freq);
      if (minHeap.size() > k)
        minHeap.pop();
    }

    while (!minHeap.empty())
      ans.push_back(minHeap.top().num), minHeap.pop();

    return ans;
  }
};
```

Output:

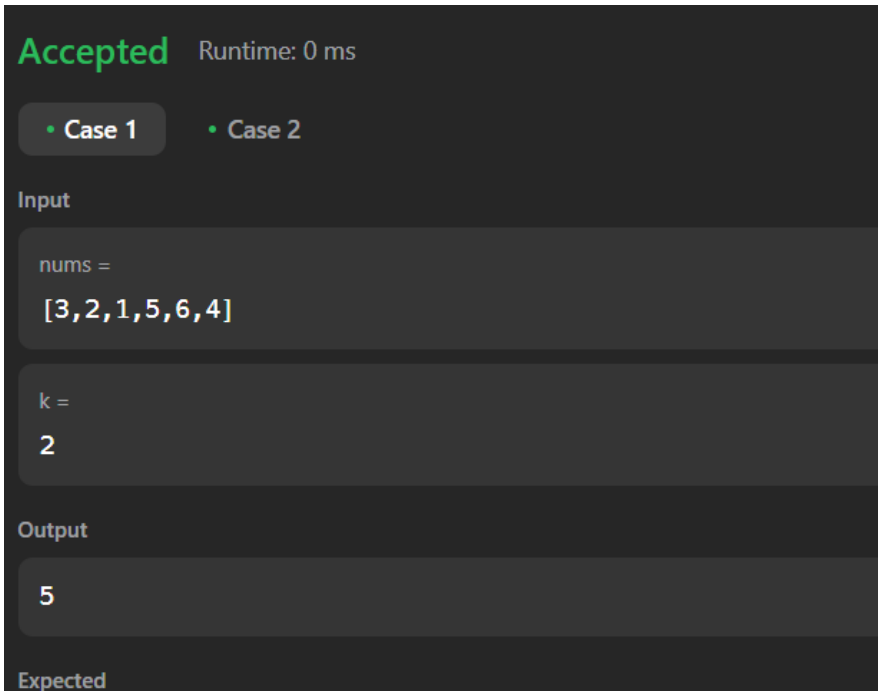


Problem 5. Given an integer array nums and an integer k, return *the* kth *largest element in the array*.

Code:

```
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(); // Remove smallest element
         }
      }

      return minHeap.top();
   }
};
```

Output:



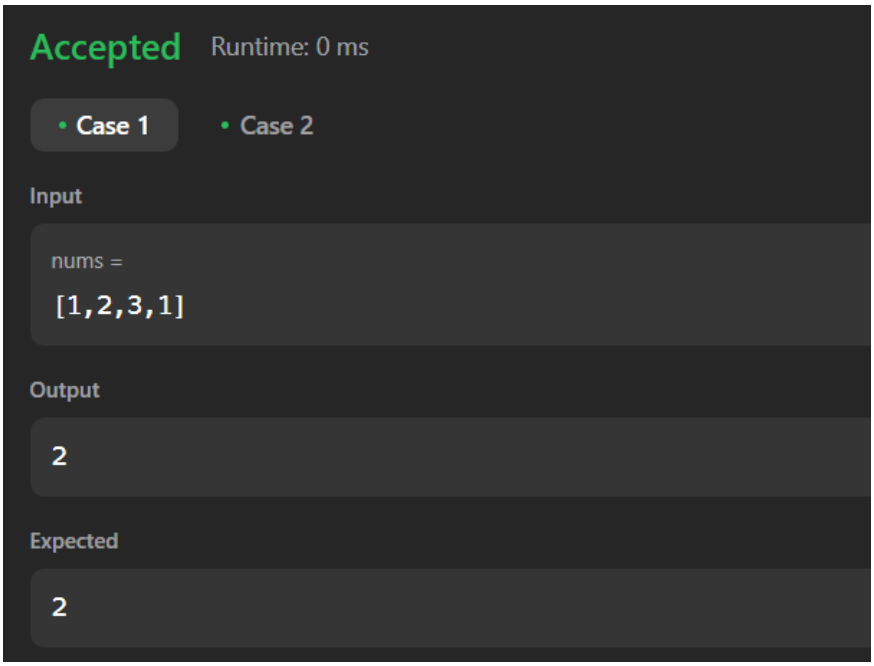
Problem 6. Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

Code:

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

```
        return -1;
    }
};
```

Output:



Problem 7. Given an array of intervals where intervals[i] = [$start_i$, end_i], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

Code:

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end()); // Sort intervals by start time
        int k = 0; // Index for merged intervals

        for (int i = 1; i < intervals.size(); i++) {
            if (intervals[k][1] >= intervals[i][0]) { // Overlap detected
                intervals[k][1] = max(intervals[k][1], intervals[i][1]); // Merge
            } else {
                k++; // Move to the next position
                intervals[k] = intervals[i]; // Replace in-place
            }
        }
```

```
        intervals.resize(k + 1); // Resize to include only merged intervals
        return intervals;
    }
};
```

Output:

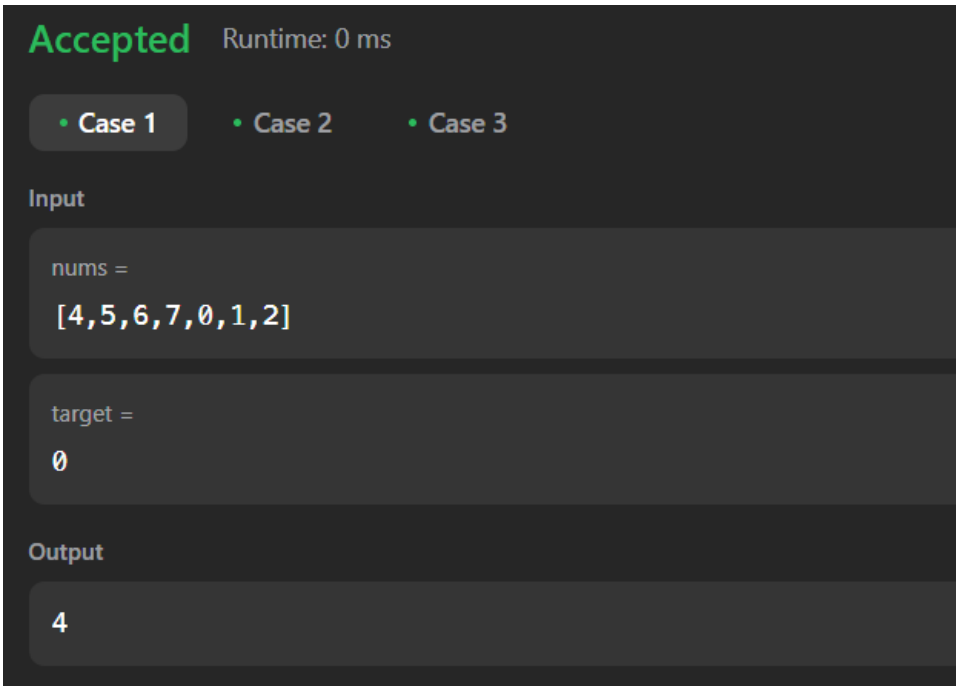
Problem 8. Given the array nums **after** the possible rotation and an integer target, return *the index of* target *if it is in* nums, *or* -1 *if it is not in* nums. You must write an algorithm with O(log n) runtime complexity.

Code:

```
class Solution {
public:
    int find(vector<int>& nums, int target,int low,int high){
        int mid=(high-low)/2 +low;
        if(low>high) return -1; //returning -1 if not found
        if(nums[mid]==target)  return mid; //if found then returning index
        if(nums[mid]>=nums[low]){
            if(target<nums[mid]&&target>=nums[low]) return find(nums,target,low,mid-1);
            else return find(nums,target,mid+1,high);
        }
```

```
        else{
           if(target>nums[mid]  &&  target<=nums[high]) return find(nums,target,mid+1,high);
           else return find(nums,target,low,mid-1);}
         }

    int search(vector<int>& nums, int target) {
        return find(nums,target,0,nums.size()-1); //calling the find
        }
};
```

Output:



Problem 9. Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:

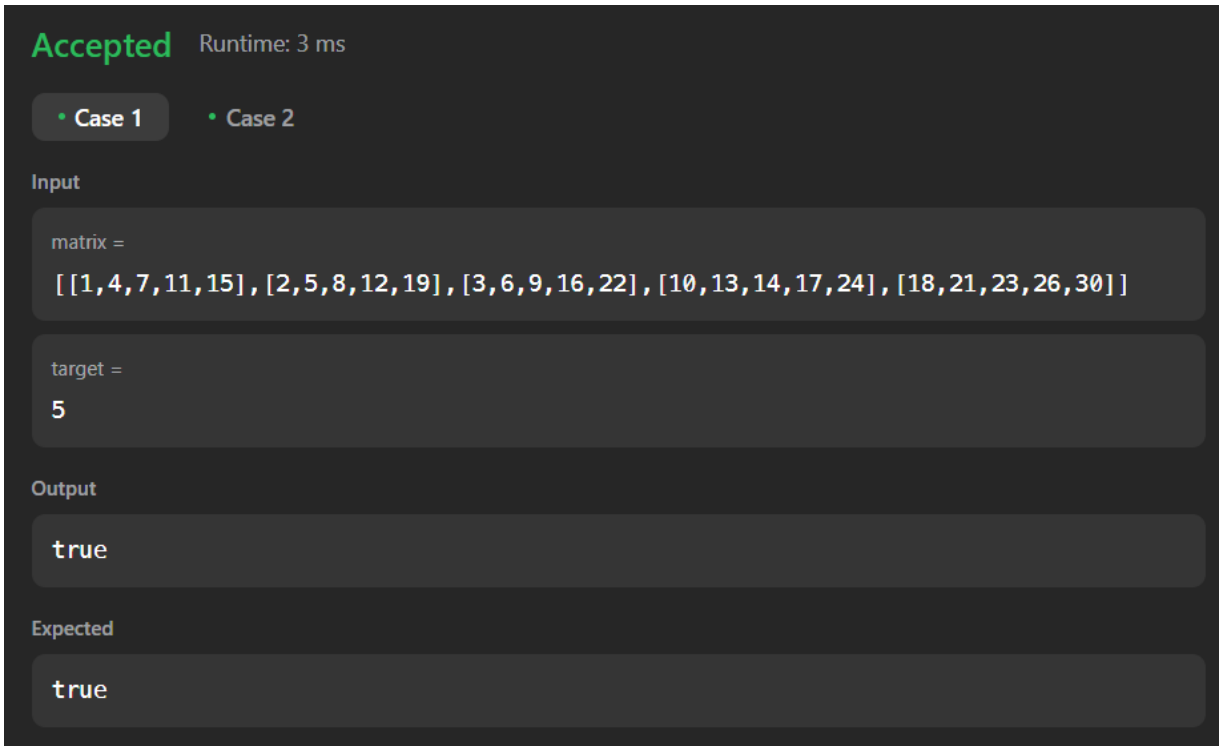
- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Code:

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int n = matrix.size(), m = matrix[0].size();
        int row = 0, col = m - 1;
        while (row < n && col >= 0) {
            if (matrix[row][col] == target) return true;
```

```
            else if (matrix[row][col] < target) row++;
            else col--;
        }
        return false;
    }
};
```

Output:



Problem 10. Given an n x n matrix where each of the rows and columns is sorted in ascending order, return *the* kth *smallest element in the matrix*.

Code:

```
class Solution {
public:
    int kthSmallest(vector<vector<int>>& matrix, int z) {
        int n = matrix.size(), m = matrix[0].size();
        int a[n*m], k=0;
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                a[k] = matrix[i][j];
                k++;
            }
```

```
        }
        sort(a, a+(n*m));
        return a[z-1];
    }
};
```

Output: