Experiment 5

Student Name: KHUSHBOO **UID: 22BET10197**
Branch: BE-IT **Section/Group: 22BET_IOT_701/A**
Semester: 6th **Date of Performance: 21-2-25**
Subject Name: AP-II **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.

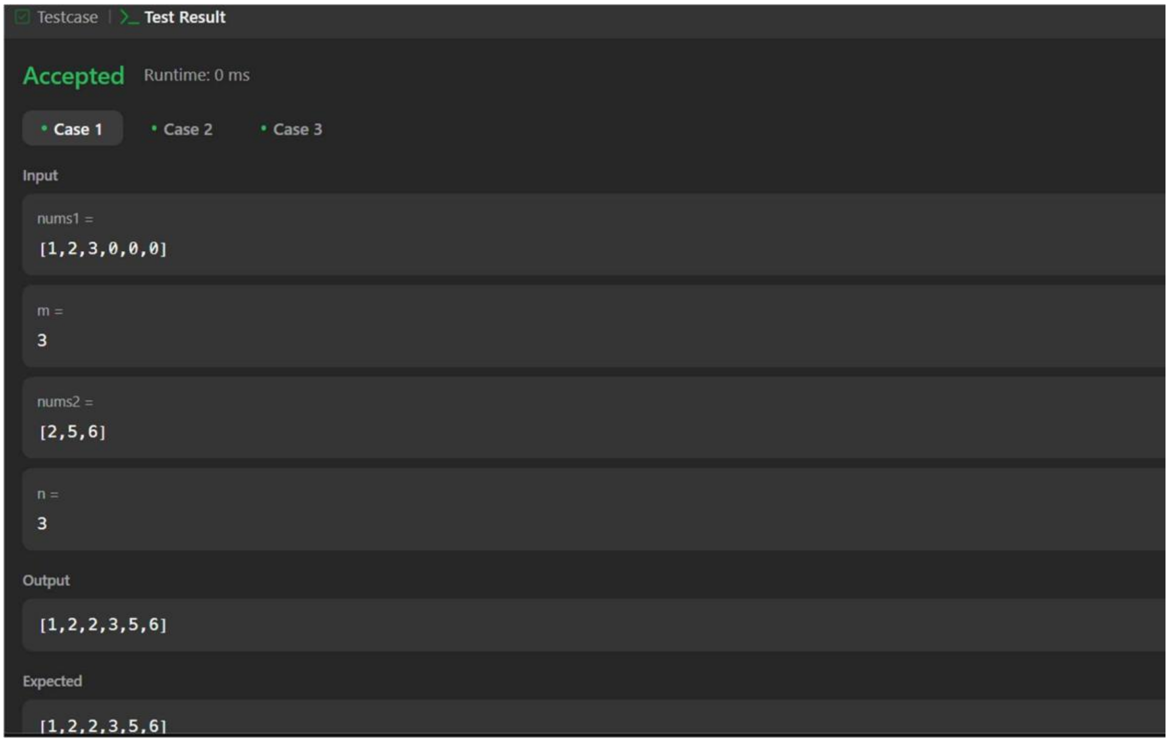
Algorithm:

1. Initialize three pointers:
   - i = m - 1 (last valid element of nums1).
   - j = n - 1 (last element of nums2).
   - k = m + n - 1 (last position of nums1).
2. **Loop while i >= 0 and j >= 0:**
   - Compare nums1[i] and nums2[j].
   - Place the larger element at nums1[k].
   - Move the respective pointer and decrement k.
3. **If elements remain in nums2:**
   - Copy them to the beginning of nums1.

Code:

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

        for(int i=0;i<m+n;i++){
            nums1[i]=v[i];
        }

    }
};
```

Output:



Problem 2. You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.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.

Algorithm:

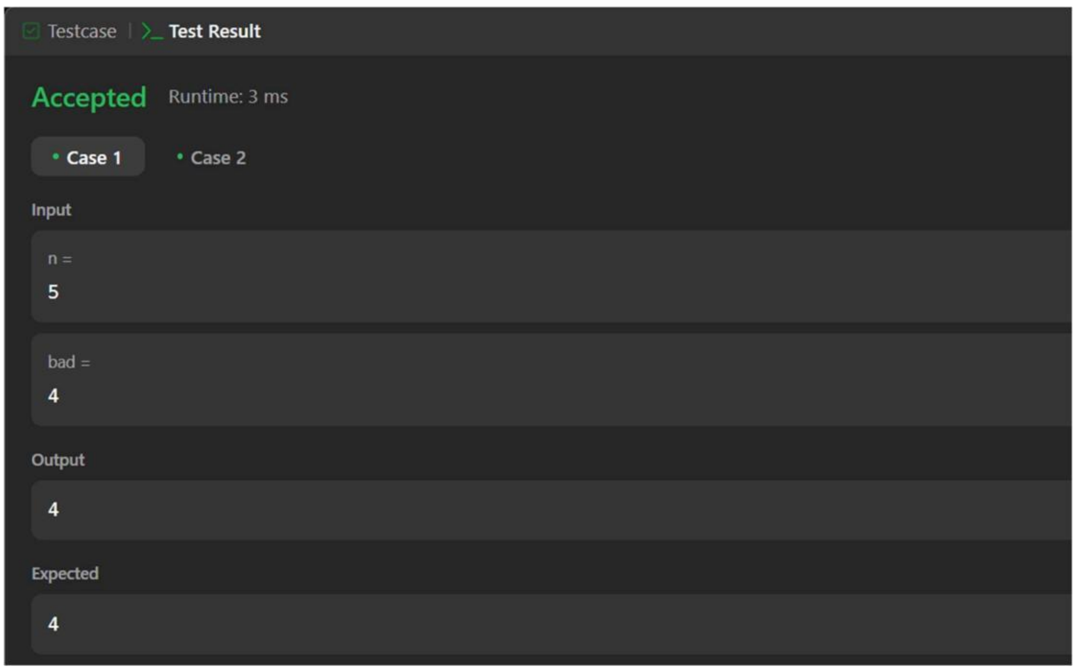
1. Use binary search to efficiently locate the first bad version.
2. Initialize left = 1 and right = n.
3. Find the middle version mid = left + (right - left) / 2.
4. If isBadVersion(mid) == true, search left half (bad version could be earlier).
5. Otherwise, search right half (bad version is ahead).
6. Continue until left points to the 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;
```

```
            if (isBadVersion(mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    }
};
```

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.We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.You must solve this problem without using the library's sort function.

Algorithm:

1. Use three pointers:
   low (starting index for 0s).
   mid (current element being checked).
   high (starting index for 2s).
2. Traverse the array using mid pointer:
   If nums[mid] == 0:
   Swap nums[mid] and nums[low], then move both forward.
   If nums[mid] == 1:

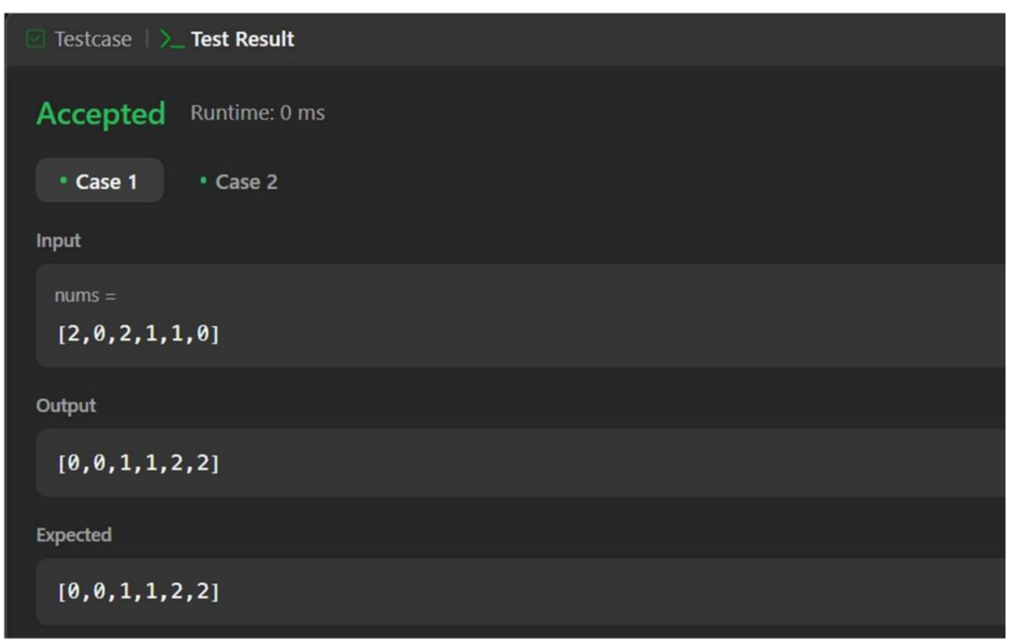
Just move mid forward.
If nums[mid] == 2:
Swap nums[mid] and nums[high], then move high backward.

3. Continue until mid crosses high.

Code:

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        vector <int> v;
        for(int i=0;i<nums.size();i++){
            if(nums[i]==0){
              v.push_back(0);
            }
        }
        for(int i=0;i<nums.size();i++){
            if(nums[i]==1){
              v.push_back(1);
            }
        }
        for(int i=0;i<nums.size();i++){
            if(nums[i]==2){
              v.push_back(2);
            }
        }
        for(int i=0;i<nums.size();i++){
            nums[i]=v[i];
        }
    }
};
```

Output:



Problem 4. You are given the head of a linked list. Delete the middle node, and return *the* head *of the modified linked list*.

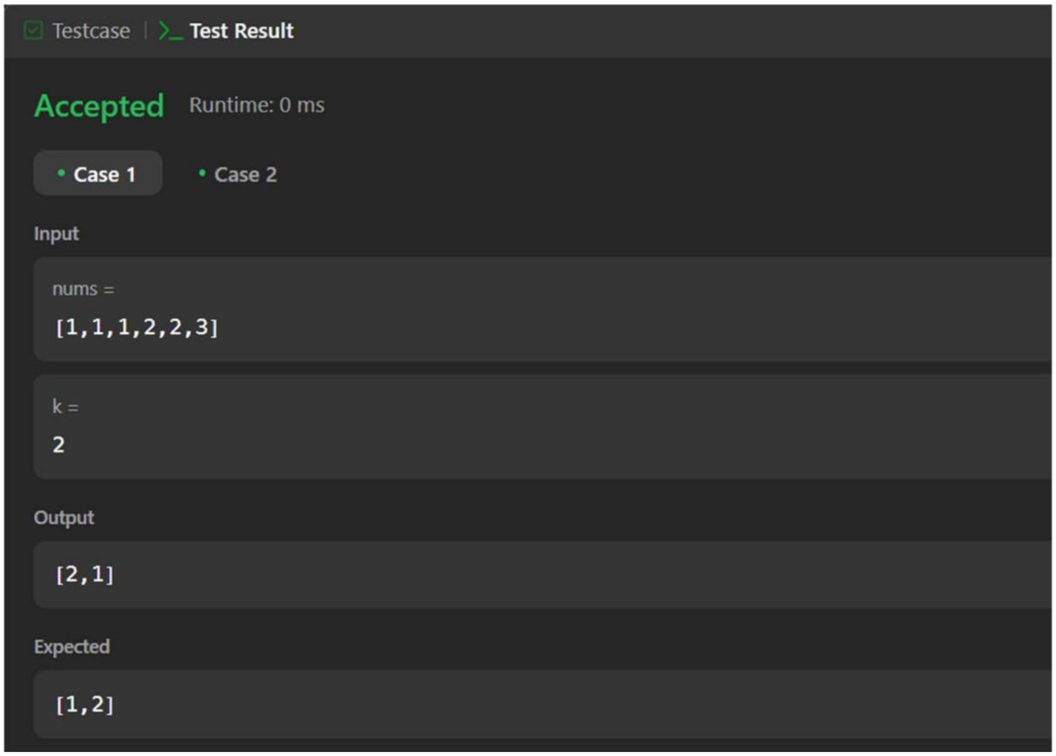
Algorithm:

1. **Use a HashMap** (unordered_map<int, int>) to count the frequency of each number.
2. **Use a Min-Heap** (priority_queue with pair<int, int> storing {frequency, number}) to store the top k elements.
   - If the heap size exceeds k, remove the least frequent element.
3. **Extract the top k elements** from the heap and store them in a result vector.

Code:

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


        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> minHeap;

        for (auto& entry : freqMap) {
            minHeap.push({entry.second, entry.first});
            if (minHeap.size() > k) {
                minHeap.pop();
            }
        }

        vector<int> result;
        while (!minHeap.empty()) {
            result.push_back(minHeap.top().second);
            minHeap.pop();
        }

        return result;

    }
};
```

Output:



Problem 5. A peak element is an element that is strictly greater than its neighbors. 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**.You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.You must write an algorithm that runs in O(log n) time.

Algorithm:

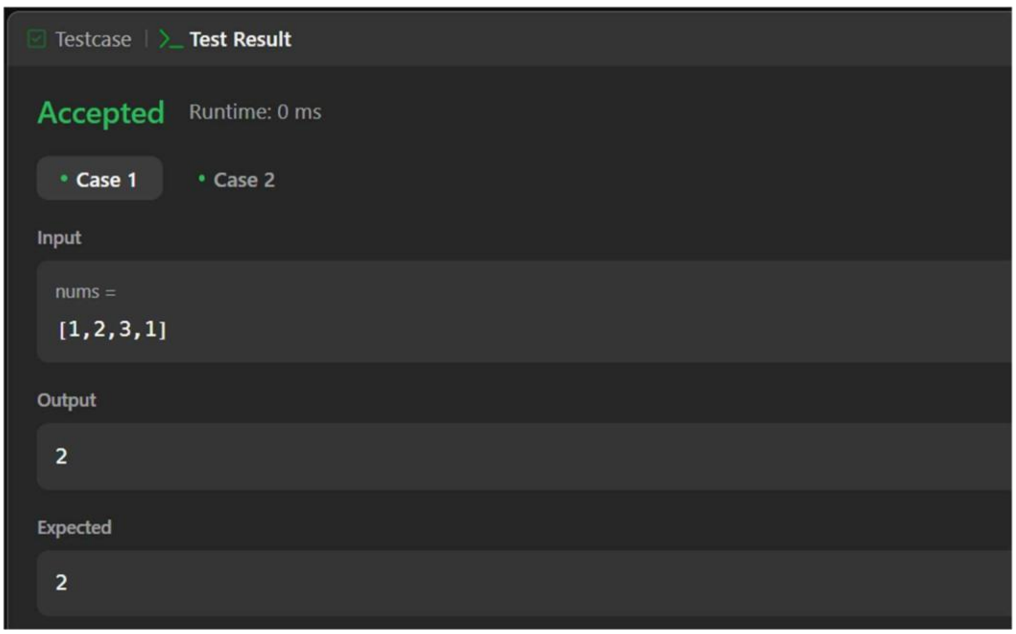
1. **Use binary search** to find a peak element efficiently.
2. **Check middle element (mid)**:
   - If nums[mid] > nums[mid + 1], a peak exists on the **left**, so move right = mid.
   - Otherwise, move left = mid + 1 to explore the **right** side.
3. **Loop until left == right**, where left will be pointing to the peak index.

Code:

```
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]) {
```

```
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    }
};
```

Output:



Problem 6. 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*.

Algorithm:

1. **Sort the intervals** based on the start time (arr[i][0]).
2. **Iterate through the sorted intervals** and maintain a vector v for merged intervals.
3. **Check for overlap:**
    - If the **current interval overlaps** with the last merged interval in v, merge them.
    - Otherwise, add the current interval as a new entry.
4. **Return the merged list v**.

Code:

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& arr) {
```

```
        vector<vector<int>> v;
        sort(arr.begin(), arr.end());
        for (int i = 0; i < arr.size(); i++) {
            if (!v.empty() && v.back()[1] >= arr[i][0]) {

                v.back()[1] = max(v.back()[1], arr[i][1]);
            } else {

                v.push_back(arr[i]);
            }
        }
        return v;
        }
    };
```

Output:

Problem 7. There is an integer array nums sorted in ascending order (with **distinct** values). Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].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.

Algorithm:

1. **Find the middle element (mid)**
    - If nums[mid] == target, return mid.
2. **Determine which half is sorted:**
    - If nums[left] ≤ nums[mid], the **left half is sorted**.

- Else, the **right half is sorted**.

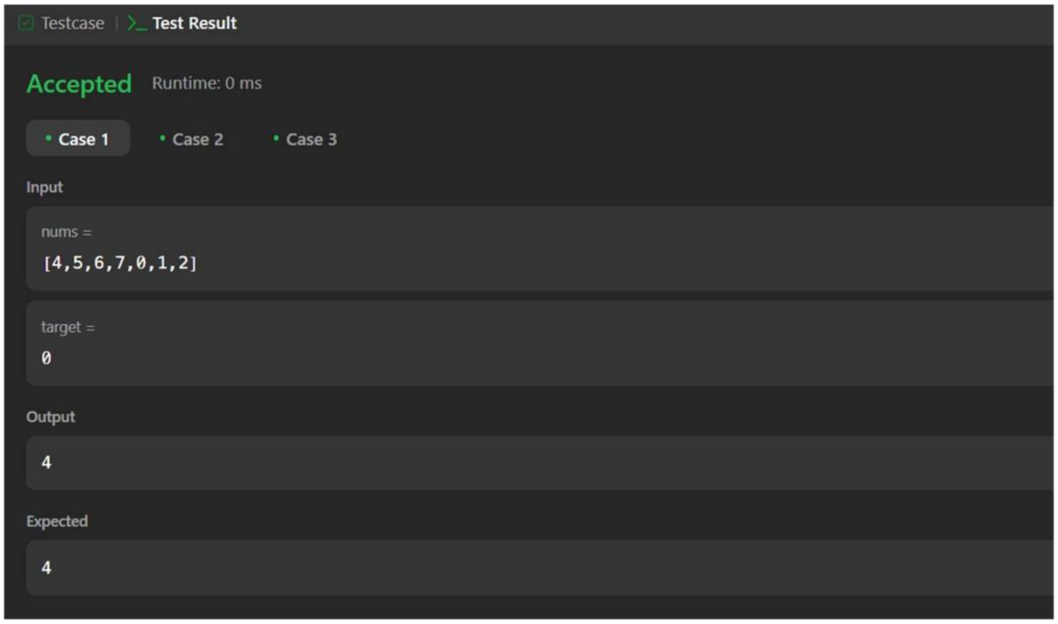
3. **Check if the target lies within the sorted half:**
   - If yes, move to that half.
   - Else, move to the other half.

5. **Repeat until left > right**.

Code:

```
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;
    }
};
```

Output:



Problem 8. 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.

Algorithm:

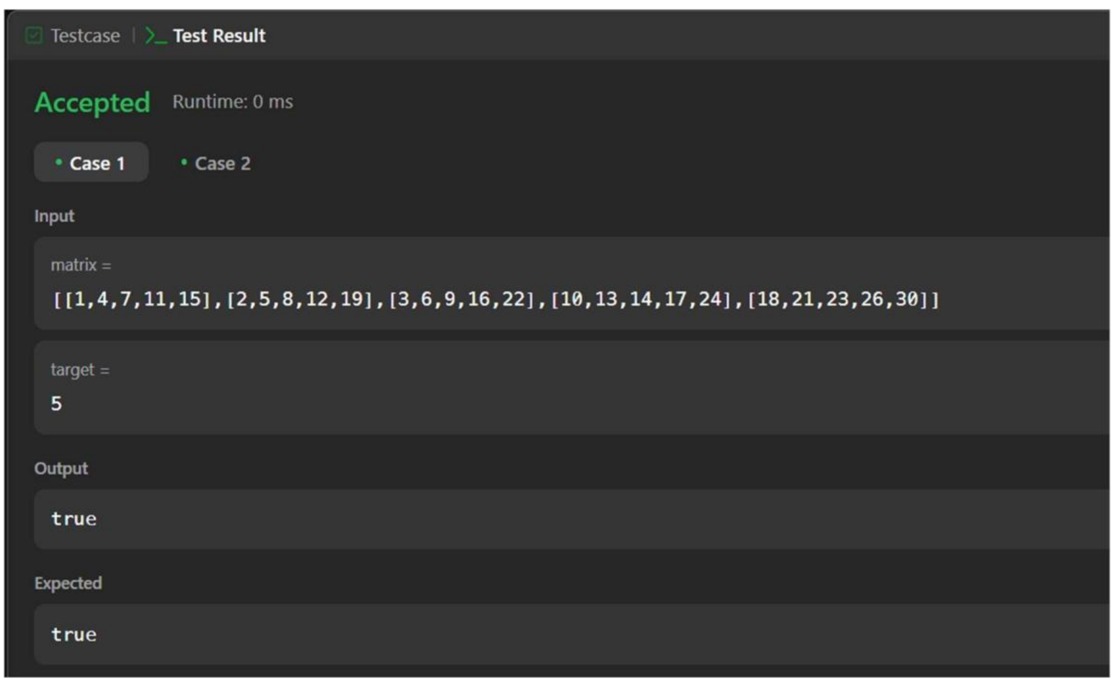
1. **Initialize boundaries**: left = 0, right = (rows × cols) - 1.
2. **Perform Binary Search**:
   - Compute mid = (left + right) / 2.
   - Convert mid into matrix[row][col]:
     - row = mid / cols
     - col = mid % cols
   - If matrix[row][col] == target, return true.
   - If matrix[row][col] < target, search the **right half** (left = mid + 1).
   - Otherwise, search the **left half** (right = mid - 1).
3. **If not found, return false**.

Code:

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
      if (matrix.empty() || matrix[0].empty()) return false;

        int rows = matrix.size();
        int cols = matrix[0].size();
        int left = 0, right = rows * cols - 1;
```

```
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int row = mid / cols;
            int col = mid % cols;

            if (matrix[row][col] == target)
                return true;
            else if (matrix[row][col] < target)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return false;
    }
};
```

Output:



Problem 9. Given the head of a linked list, rotate the list to the right by k places.

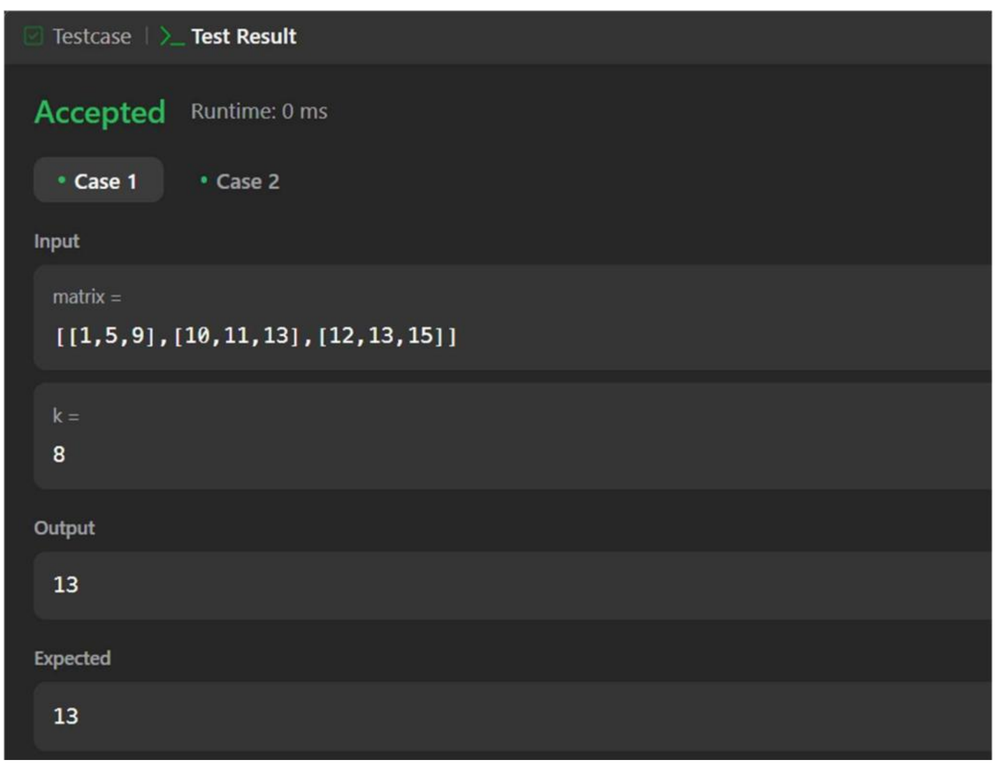
Algorithm:

1. **Use a min-heap** (priority_queue) to store elements in increasing order.
2. **Insert the first column** elements (matrix[i][0]) into the heap along with their row and column index.
3. **Extract the smallest element k times**, pushing the next element from the same row.
4. **Return the kth extracted element**.

Code:

```
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;

        for (int i = 0; i < n; i++)
            minHeap.push({matrix[i][0], i, 0});

        int count = 0, result;
        while (!minHeap.empty()) {
            auto [val, row, col] = minHeap.top();
            minHeap.pop();
            result = val;
            count++;

            if (count == k) return result;

            if (col + 1 < n)
                minHeap.push({matrix[row][col + 1], row, col + 1});
        }
        return -1;
    }
};
```

Output:



Problem 10. Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Algorithm:

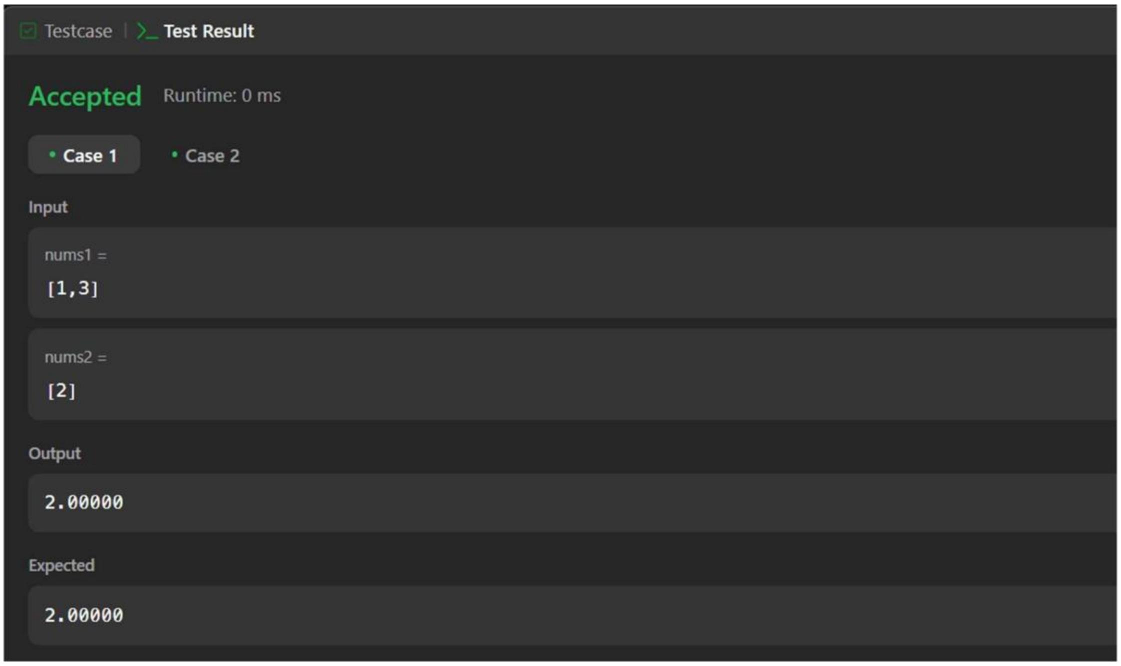
1. **Ensure nums1 is the smaller array** (nums1.size() <= nums2.size()) for optimized performance.
2. **Use binary search** on nums1:
   - Define search space: left = 0, right = nums1.size().
   - Find the **partition** index i for nums1, and calculate j = (m + n) / 2 - i for nums2.
   - Ensure **left partition ≤ right partition**.
3. **Check validity**:
   - If nums1[i-1] ≤ nums2[j] **and** nums2[j-1] ≤ nums1[i], it's a valid partition.
   - Otherwise, adjust the binary search range.
4. **Compute median**:
   - If (m + n) is **odd**, return min(right half).
   - If (m + n) is **even**, return average of max(left half) & min(right half)

Code:

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        if (nums1.size() > nums2.size())
            return findMedianSortedArrays(nums2, nums1);

        int m = nums1.size(), n = nums2.size();
        int left = 0, right = m, medianPos = (m + n + 1) / 2;

        while (left <= right) {
            int i = left + (right - left) / 2;
            int j = medianPos - i;

            int nums1LeftMax = (i == 0) ? INT_MIN : nums1[i - 1];
            int nums1RightMin = (i == m) ? INT_MAX : nums1[i];
            int nums2LeftMax = (j == 0) ? INT_MIN : nums2[j - 1];
            int nums2RightMin = (j == n) ? INT_MAX : nums2[j];

            if (nums1LeftMax <= nums2RightMin && nums2LeftMax <= nums1RightMin) {
                if ((m + n) % 2 == 0)
                    return (max(nums1LeftMax, nums2LeftMax) + min(nums1RightMin, nums2RightMin)) / 2.0;
                else
                    return max(nums1LeftMax, nums2LeftMax);
            }
            else if (nums1LeftMax > nums2RightMin)
                right = i - 1;
            else
                left = i + 1;
        }

        return 0.0;
```

```
    }
};
```

Output:


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Input
nums1 =
[1,3]
nums2 =
[2]
Output
2.00000
Expected
2.00000