Experiment 4

Student Name: Karan Goyal **UID:-22BCS15864**

Branch: BE-CSE **Group-Ntpp_IOT_602-A**

Semester: 5TH **Dateof Performance:27-1-25**

Subject Name:-Advance Programming lab-2 **Subject Code:-22CSP-351**

Problem 1

Aim:- 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**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

Objective:- The objective is to merge two sorted arrays, nums1 and nums2, into a single sorted array in-place. The function efficiently places elements from the end of nums1 and nums2 into nums1 from the back, ensuring sorted order without using extra space.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm for the Two Sum Problem:

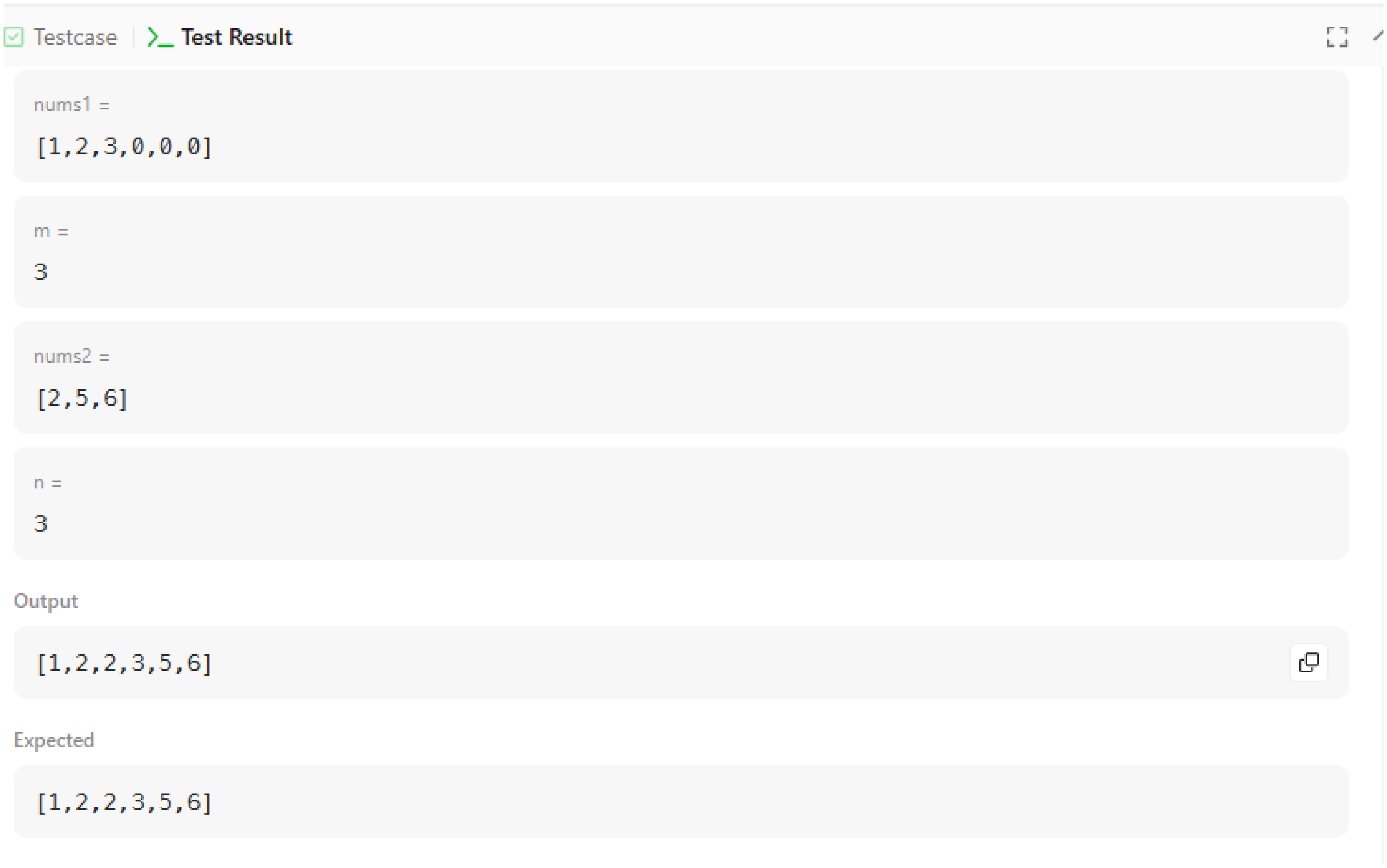
1. **Check Base Condition** – If the root is nullptr, return 0 (empty tree has depth 0).
2. **Recursively Compute Left Depth** – Call maxDepth(root->left) to compute the depth of the left subtree.
3. **Recursively Compute Right Depth** – Call maxDepth(root->right) to compute the depth of the right subtree.
4. **Compare Depths** – Take the maximum of the left and right subtree depths.
5. **Increment Depth** – Add 1 to include the current root node in the depth count.
6. **Return Result** – Return the computed depth value.

Code:

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int midx = m - 1;
        int nidx = n - 1;
        int right = m + n - 1;
        while (nidx >= 0) {
            if (midx >= 0 && nums1[midx] > nums2[nidx]) {
                nums1[right] = nums1[midx];
                midx--;
            } else {
                nums1[right] = nums2[nidx];
                nidx--;
            }
            right--;
        }
    }
};
```

- **Time complexity**: $O(m+n)$
- **Space complexity:** $O(1)$

Output- All the test cases passed



```
nums1 =
[1,2,3,0,0,0]

m =
3

nums2 =
[2,5,6]

n =
3

Output
[1,2,2,3,5,6]

Expected
[1,2,2,3,5,6]
```

Problem-2

Aim:- 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.

Objective- The objective is to efficiently identify the first bad version in a sequence of product versions using the isBadVersion(version) API. By minimizing the number of API calls, the solution should implement an optimized search approach, ensuring quick detection while reducing computational overhead, enabling faster debugging and resolution.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm

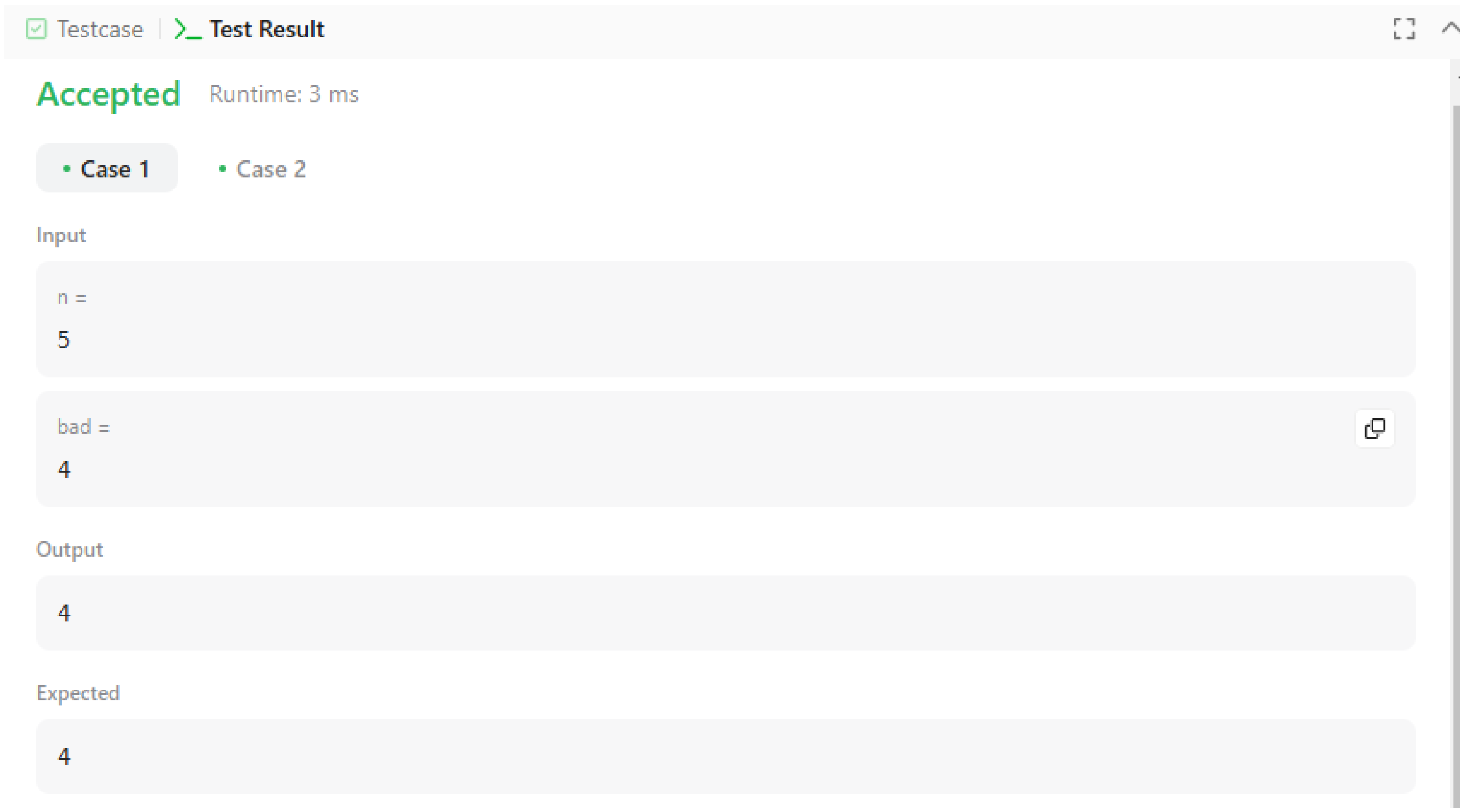
1. Initialize Variables – Set low = 1 and high = n to define the search range.
2. Perform Binary Search – Use a while loop to continue searching while low ≤ high.
3. Calculate Midpoint – Compute mid = low + (high - low) / 2 to avoid overflow.
4. Check Version Status – Call isBadVersion(mid).
5. Adjust Search Range – If mid is bad, set high = mid - 1; otherwise, set low = mid + 1.
6. Return Result – When the loop exits, low will be the first bad version. Return low.

Code-

```
class Solution {
public:
   int firstBadVersion(int n) {
      int low=1;
      int high=n;
      while(low<=high)
      {
         int mid=low+(high-low)/2;
         if(isBadVersion(mid))high=mid-1;
         else low=mid+1;
      }
      return low;
   }
};
```

- **Time Complexity**: O(log n).
- **Space Complexity:** O(1)

Result-All test cases passes



Testcase | Test Result

Accepted Runtime: 3 ms

• Case 1 • Case 2

Input

```
n =
5
```

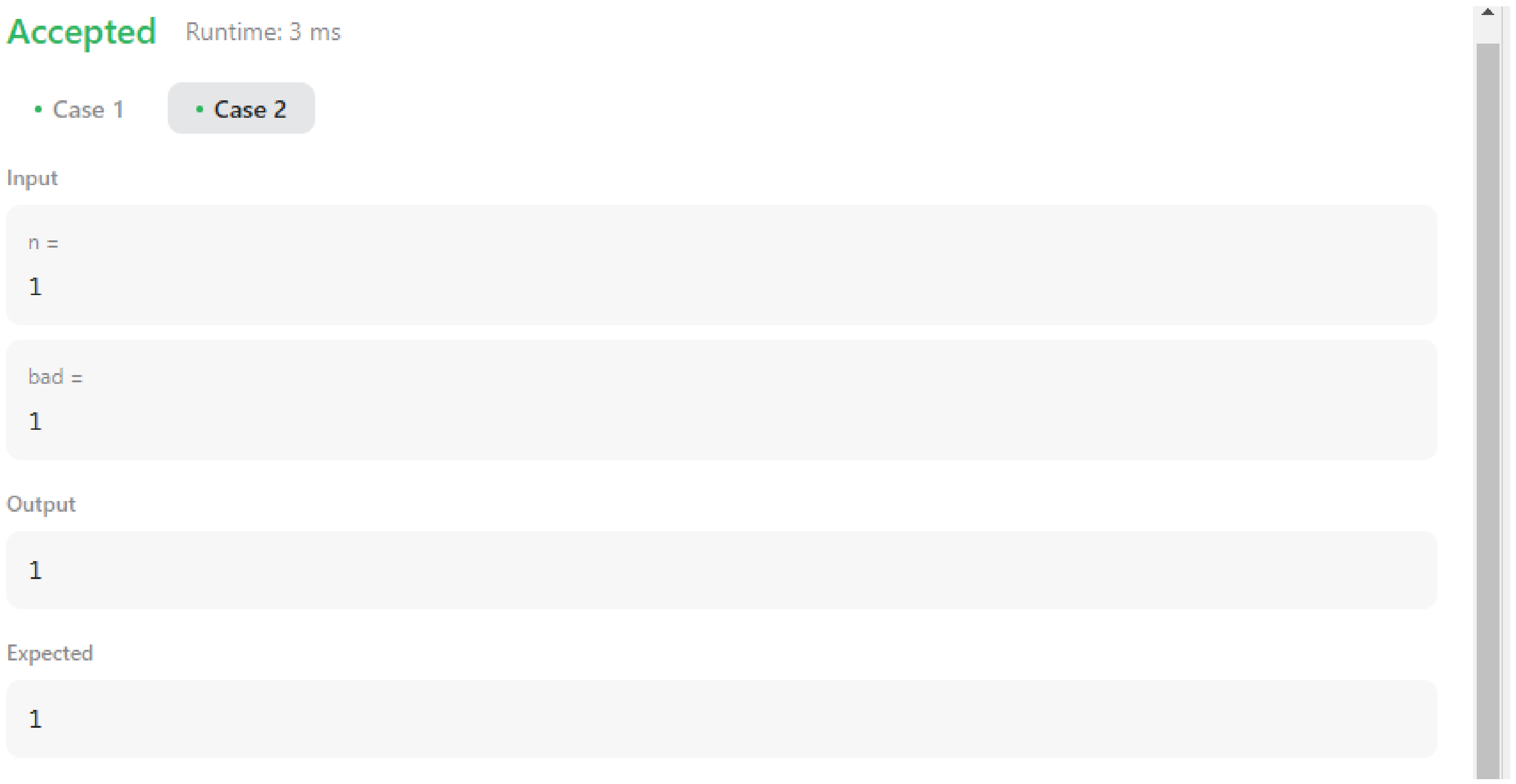
```
bad =
4
```

Output

```
4
```

Expected

```
4
```



Accepted Runtime: 3 ms

• Case 1 • Case 2

Input

```
n =
1
```

```
bad =
1
```

Output

```
1
```

Expected

```
1
```

Problem-3

Aim- 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.

Objective- The objective is to efficiently find a peak element in a given 0-indexed integer array nums, where a peak is strictly greater than its neighbors. The solution must run in O(log⁡n)O(\log n) time using a binary search approach to identify any peak index while minimizing computational overhead.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm

1. Initialize Pointers:
   Set left = 0 and right = nums.size() - 1 to define the search range.
2. Perform Binary Search:
   Run a loop while left < right to narrow down the search space.
3. Calculate Midpoint:
   Compute mid = left + (right - left) / 2 to find the middle index.
4. Compare with Right Neighbor:
   - If nums[mid] > nums[mid + 1], move right = mid (peak lies on the left side).
   - Otherwise, move left = mid + 1 (peak lies on the right side).
5. Repeat Until Converged:
   The loop continues until left == right, ensuring the peak is found.
6. Return the Peak Index:
   The final value of left (or right) is the index of a peak element.

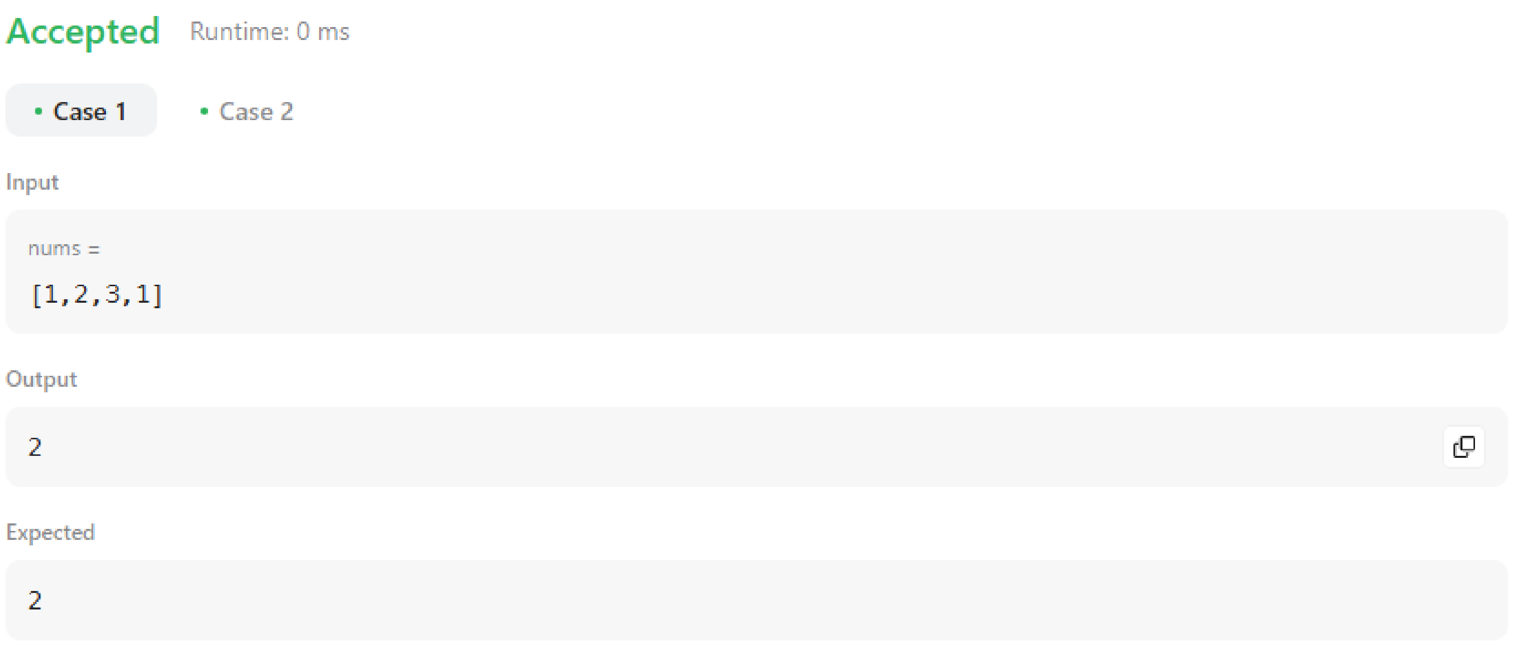
Code-

```
class Solution {

public:

   int findPeakElement(vector<int>& nums) {

      int left = 0;
```

```
        int 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;

    }

};
```

Time Complexity: O(logn)O(log n) (binary search)
Space Complexity: O(1)O(1) (constant extra space)

Result- All test cases passes



Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

nums =

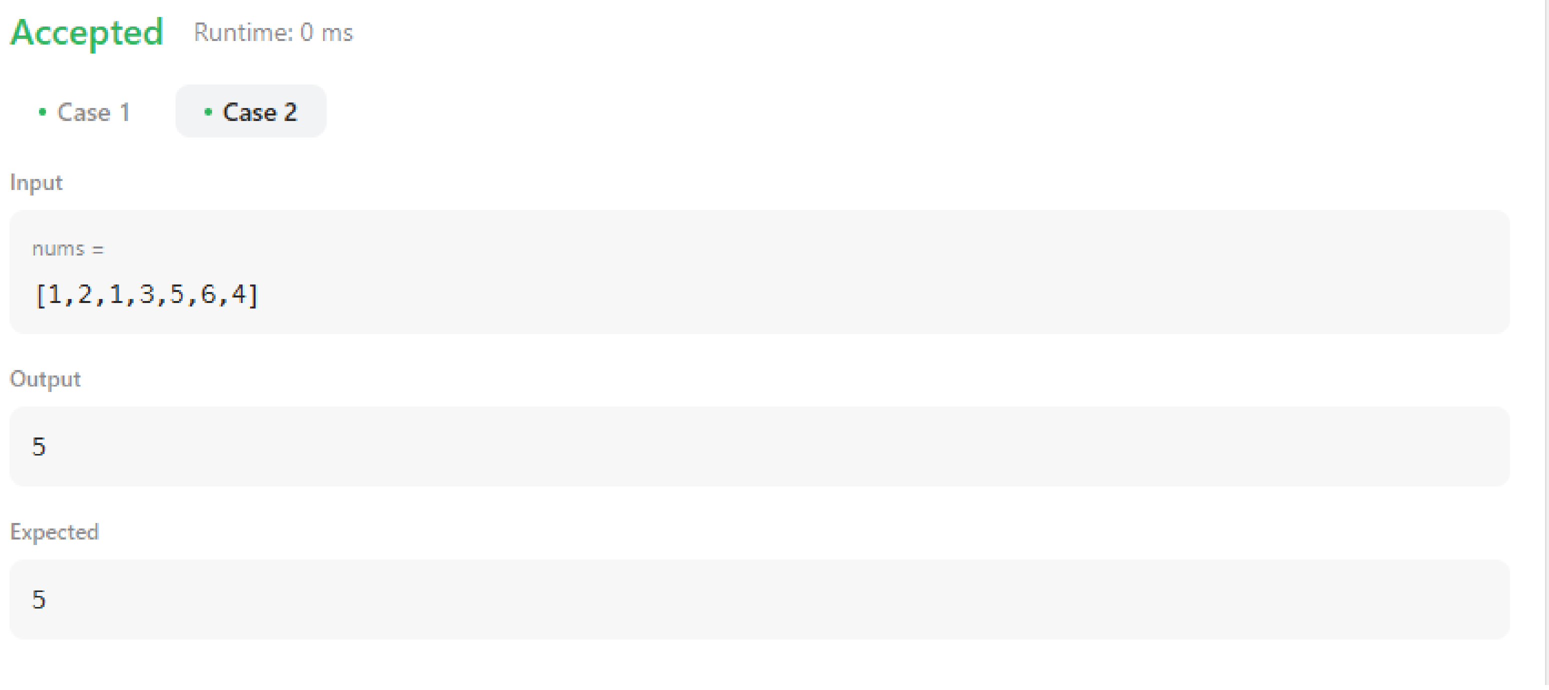
```
[1,2,3,1]
```

Output

```
2
```

Expected

```
2
```



Input

```
nums =
[1,2,1,3,5,6,4]
```

Output

```
5
```

Expected

```
5
```

Learning Outcomes:

Here are the combined learning outcomes in 5 points with subpoints:

1. **Efficient Search Techniques**:
    - Apply binary search principles to solve problems like finding a peak element or the first bad version.
    - Use the binary search approach to reduce the search space and optimize time complexity.
    - Achieve a time complexity of O(log n) in problems where elements are sorted or can be divided.
2. **Merging and Modifying Arrays**:
    - Merge two sorted arrays into one sorted array, modifying one of the arrays in place.
    - Use two pointers to compare elements from both arrays and insert the larger element in the correct position.
    - Optimize space complexity by performing the merge in-place without using extra arrays.
3. **Handling Edge Cases**:
    - Manage edge cases such as empty arrays or when the peak element is at the boundaries of the array.
    - Handle scenarios where the search space reduces to a single element in problems like finding the first bad version.
4. **Optimal Time Complexity**:
    - Minimize the time complexity by using efficient algorithms like binary search and in-place array modification.

      Enhance performance by avoiding unnecessary iterations and using logical condition checks.
5. **Problem-Solving with Binary Search**:
    - Understand how binary search can be used not just for searching in sorted arrays but for optimizing other problems.
    - Apply binary search to find specific elements (e.g., first bad version, peak elements) in an unsorted array.

- Reduce unnecessary computations by narrowing the search space at each step based on element comparisons.