Experiment-5

Student Name: Vickey Kumar
UID: 22BET10148
Branch: BE-IT
Section: 22BET_IOT-702 'B'
Semester: 6th
Date of Performance:21/02/25
Sub Name: Advanced Programming Lab-2
Subject Code: 22ITP-351

Problem 1

1. Aim:

You are given two integer arrays, nums1 and nums2, both sorted in non-decreasing order, along with two integers, m and n, which represent the number of elements in nums1 and nums2, respectively. Your task is to merge nums2 into nums1, ensuring the final array remains sorted in non-decreasing order

2. Objective:

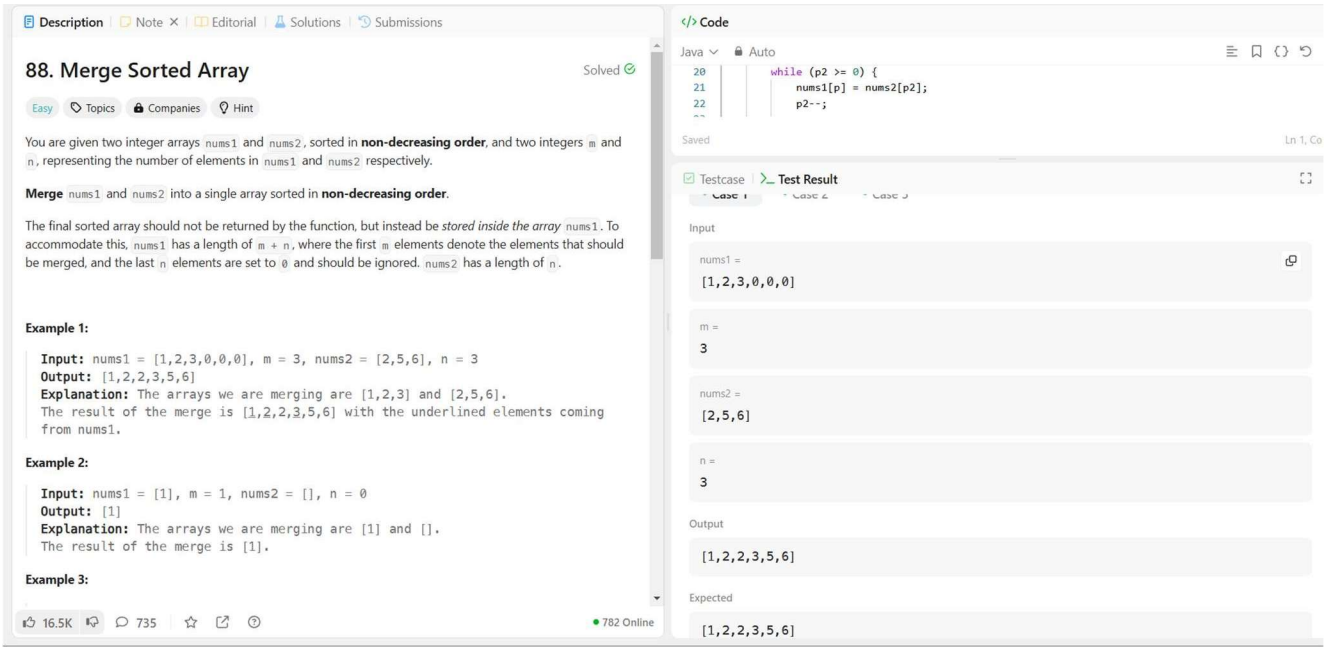
1. Merge the elements of nums2 into nums1 while maintaining a non-decreasing order.
2. Perform the merging in-place within nums1 without using extra space.
3. Ensure that the final merged array is fully sorted and occupies the first m + n positions of nums1.
4. Efficiently handle the merging process using an optimal approach, such as the two-pointer technique.

3. Code:

```
class Solution {
public void merge(int[] nums1, int m, int[] nums2, int n) {
int p1 = m - 1;
int p2 = n - 1;
int p = m + n - 1;
while (p1 >= 0 && p2 >= 0) {
if (nums1[p1] > nums2[p2]) {
nums1[p] = nums1[p1];
```

```
p1--;
} else {
nums1[p] = nums2[p2];
p2--;
}
p--;
}
while (p2 >= 0) {
nums1[p] = nums2[p2];
p2--;
p--;
}
}
}
```

4. Output:



Merge Sorted Array

5. Learning Outcomes:

1. Understand and implement the in-place merging of two sorted arrays without using extra space.
2. Apply the two-pointer technique to efficiently merge sorted arrays in O(m + n) time complexity.
3. Learn how to handle edge cases, such as empty arrays or all elements being greater/smaller in one array.
4. Develop problem-solving skills related to array manipulation and sorting techniques in Java.

Problem 2

1. Aim:

As a product manager, you are leading a team developing a new product. The latest version has failed the quality check, and since each version builds on the previous one, all subsequent versions are also defective.

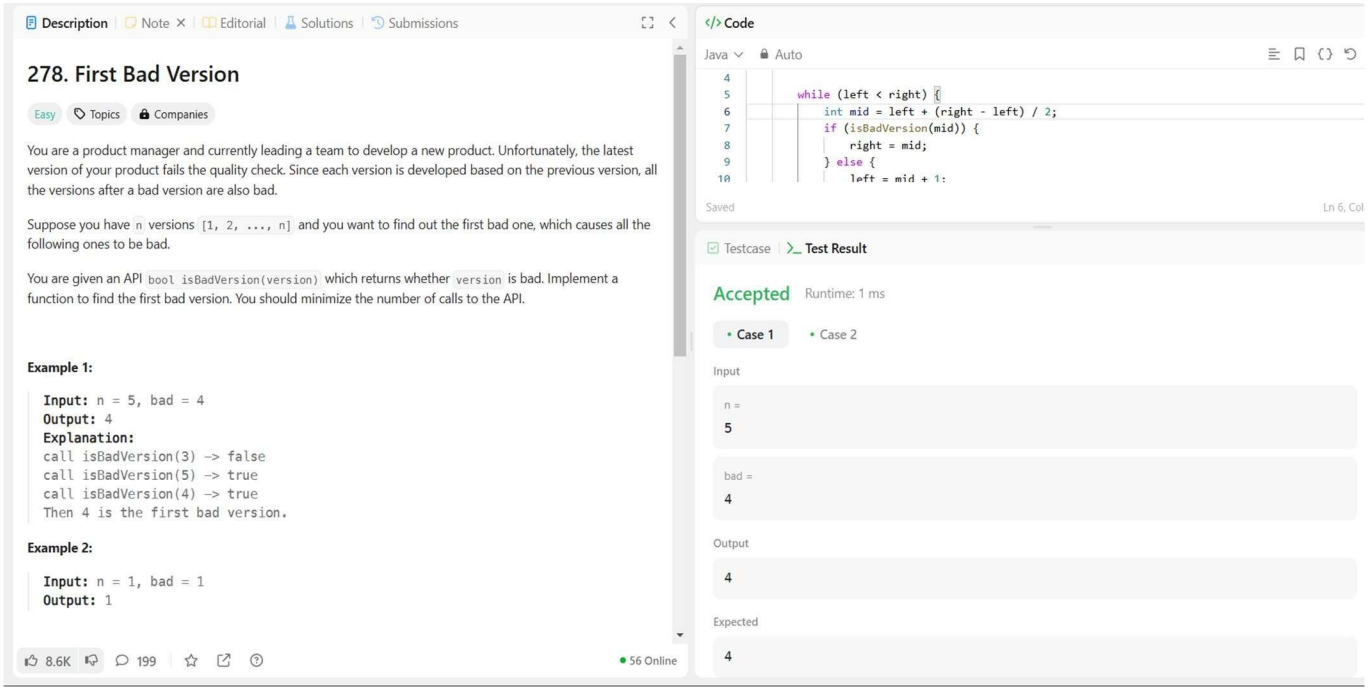
2. Objective:

1. Identify the first defective version using the isBadVersion API.
2. Optimize the search process to minimize API calls.
3. Implement an efficient solution using binary search to achieve O(log n) time complexity.
4. Ensure accurate detection to prevent faulty versions from being released.

3. Code:

```
public class Solution extends VersionControl {

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;

}
```

4. Output:



First Bad Version

5. Learning Outcomes:

1. Understand how to apply binary search to efficiently locate the first occurrence of a condition in a sorted sequence.
2. Learn to optimize algorithms for O(log n) time complexity by reducing the search space iteratively.
3. Gain experience in using API-based decision-making to solve real-world problems.
4. Develop problem-solving skills in handling edge cases, such as when the first version is bad or all versions are good.

Problem 3

1. Aim:

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.

2. Objective:

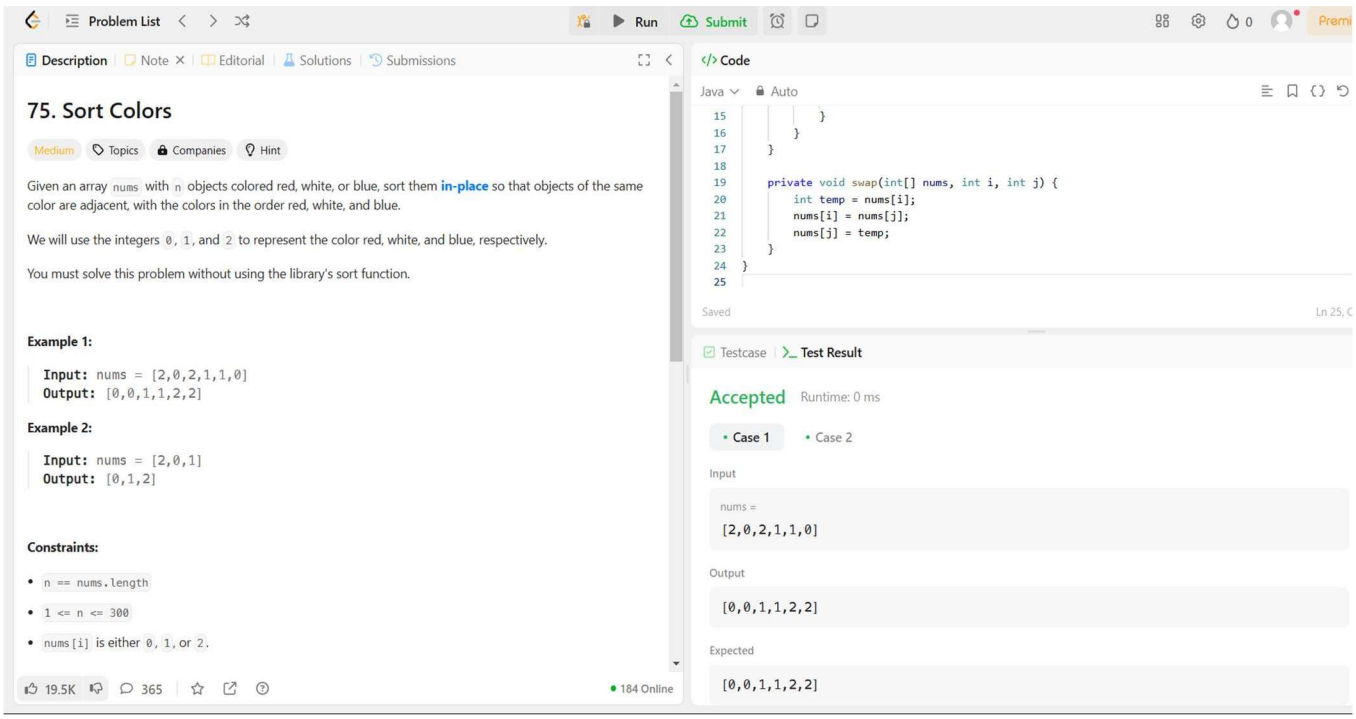
1. Sort an array containing only 0s, 1s, and 2s in non-decreasing order.
2. Implement an in-place sorting algorithm without using extra space.
3. Optimize the sorting process to run in O(n) time complexity.
4. Utilize the Dutch National Flag Algorithm to efficiently partition the array with minimal swaps.

3. Code:

```
class Solution {

public void sortColors(int[] nums) {

int low = 0, mid = 0, high = nums.length - 1;

while (mid <= high) {

if (nums[mid] == 0) {

swap(nums, low, mid);

low++;

mid++;

} else if (nums[mid] == 1) {

mid++;

} else {

swap(nums, mid, high);

high--;
```

```
}
}
}
private void swap(int[] nums, int i, int j) {
int temp = nums[i];
nums[i] = nums[j];
nums[j] = temp;
}
}
```

4.Output:



Sort Colors

5. Learning Outcomes:

1. Understand and apply the Dutch National Flag Algorithm for efficient in-place sorting.
2. Learn how to sort an array containing a limited range of distinct values in O(n) time complexity.
3. Gain experience in using three-pointer techniques to optimize sorting problems.
4. Develop problem-solving skills in in-place array manipulation without extra space.

Problem 4

1.Aim:

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

2.Objective:

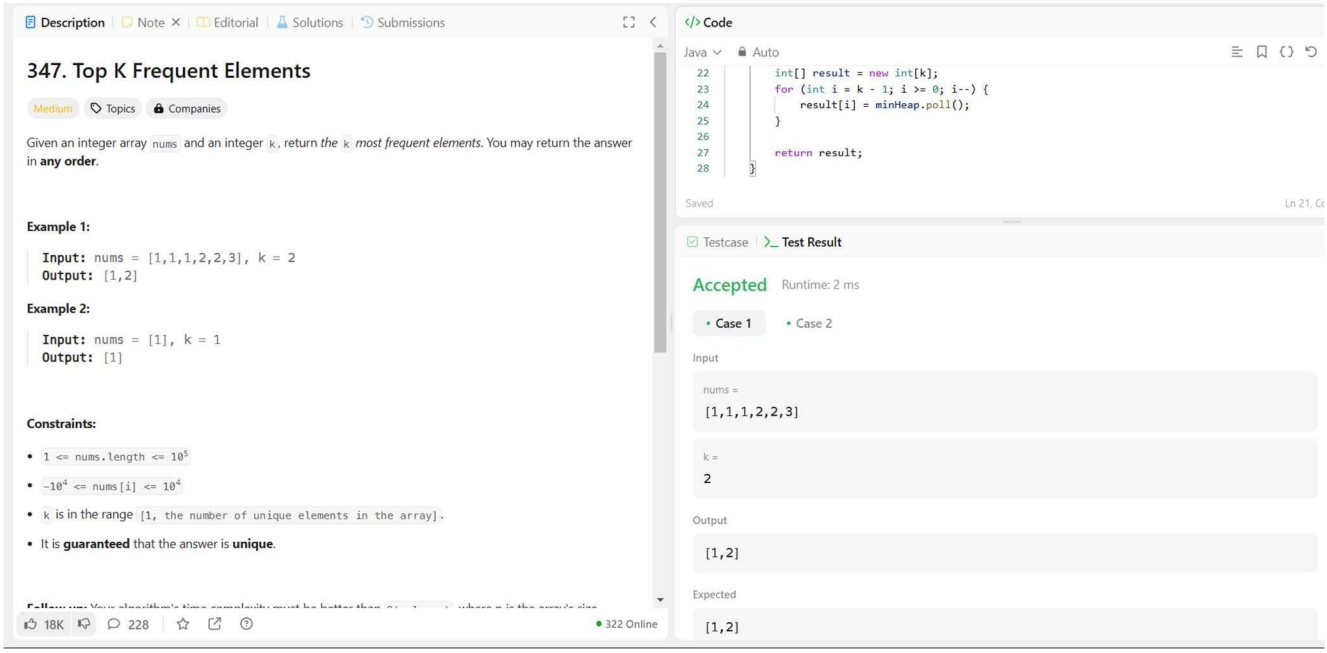
1. Identify the k most frequent elements in a given array.
2. Use an efficient approach to ensure optimal time complexity.
3. Implement a solution that utilizes hashing and heap-based data structures.
4. Optimize space usage while maintaining an in-place or auxiliary structure.

3.Code:

```
import java.util.*;
class Solution {
public int[] topKFrequent(int[] nums, int k) {
Map<Integer, Integer> freqMap = new HashMap<>();
for (int num : nums) {
freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
}
PriorityQueue<Integer> minHeap = new PriorityQueue<>(Comparator.comparingInt(freqMap::get));
for (int num : freqMap.keySet()) {
minHeap.offer(num);
if (minHeap.size() > k) {
minHeap.poll();
}
}
int[] result = new int[k];
for (int i = k - 1; i >= 0; i--) {
result[i] = minHeap.poll();
}
return result;
```

```
}
}
```

4.Output:



Top K Frequent Elements

5.Learning Outcomes:

1. Understand how to use HashMaps to count element frequencies efficiently.
2. Learn how to implement a Min-Heap (Priority Queue) to track the top k frequent elements.
3. Gain experience in optimizing problems using heap-based sorting with O(n log k) complexity.
4. Explore an alternative Bucket Sort approach to achieve O(n) time complexity for frequency-based problems.

Problem 5

1. Aim:

Given an integer array nums and an integer k, return *the* kth *largest element in the array*.
Note that it is the kth largest element in the sorted order, not the kth distinct element.

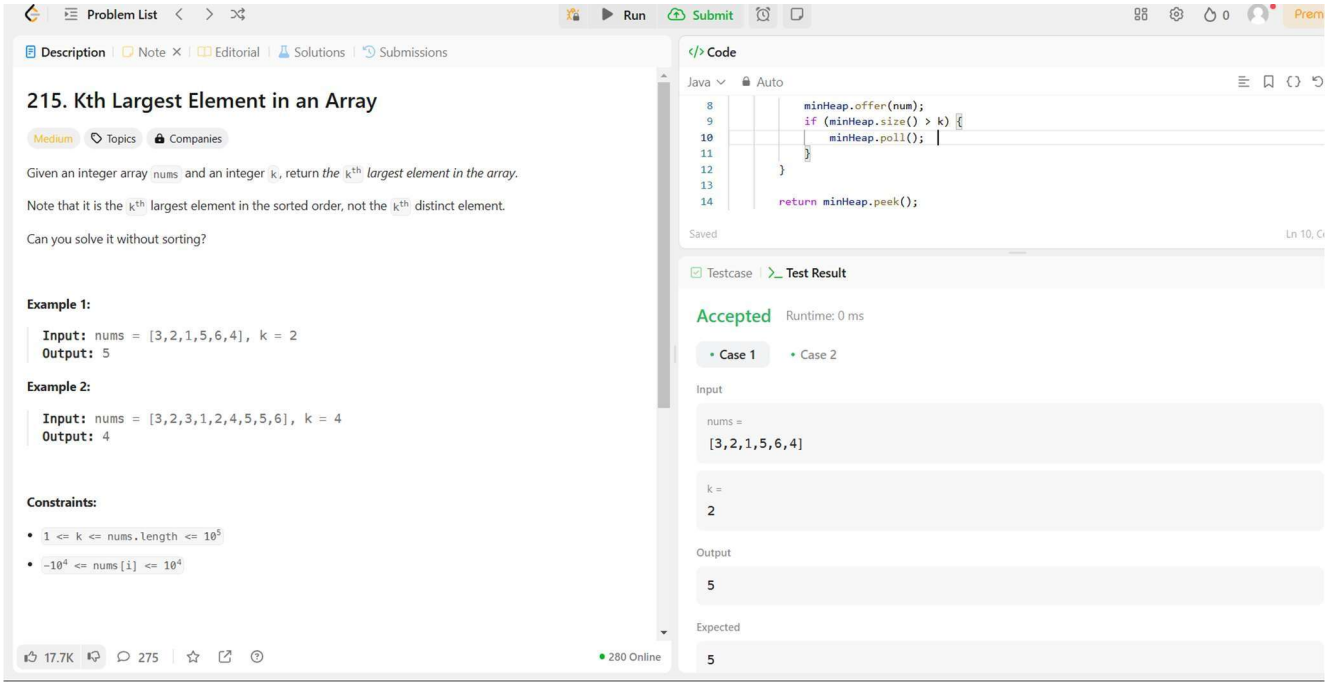
2. Objective:

1. Find the k-th largest element in an unsorted array efficiently.
2. Implement an optimized approach using a Min-Heap (Priority Queue) or QuickSelect.
3. Achieve a time complexity of O(n log k) using a heap or O(n) on average using QuickSelect.
4. Ensure the solution works for large inputs while maintaining minimal space complexity.

3. Code:

```
import java.util.*;
class Solution {
public int findKthLargest(int[] nums, int k) {
PriorityQueue<Integer> minHeap = new PriorityQueue<>();

for (int num : nums) {
minHeap.offer(num);
if (minHeap.size() > k) {
minHeap.poll();
}
}
return minHeap.peek();
}
}
```

4. Output:



Kth Largest Element in Array

5.Learning Outcomes:

1. Understand how to use a Min-Heap (Priority Queue) to efficiently find the k-th largest element in O(n log k) time.
2. Learn the QuickSelect algorithm, an optimized approach based on Hoare's Partition Scheme, which runs in O(n) on average.
3. Gain experience in selecting the right algorithm based on constraints (heap for stability, QuickSelect for efficiency).
4. Develop problem-solving skills in handling array partitioning, sorting, and optimization techniques.