# Experiment 5

**Student Name:** Rahul Prasad **UID:** 22BET10167

**Branch:** IT **Section/Group:** 701/A

**Semester:** 6th **Date :** 21/02/2025

**Subject Name:** Advanced Programming Lab - 2 **Subject Code:** 22ITP-351

## 1. Problem 1:

### ➢ Merge Sorted Array:

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.

### ➢ Code:

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

```
            } else {
                nums1[k--] = nums2[j--];
            }
        }
        while (j >= 0) {
            nums1[k--] = nums2[j--];
        }
    }
}
```

➢ **Output**

2. **Problem 2:**

➢ **Search a 2D Matrix II:**

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.

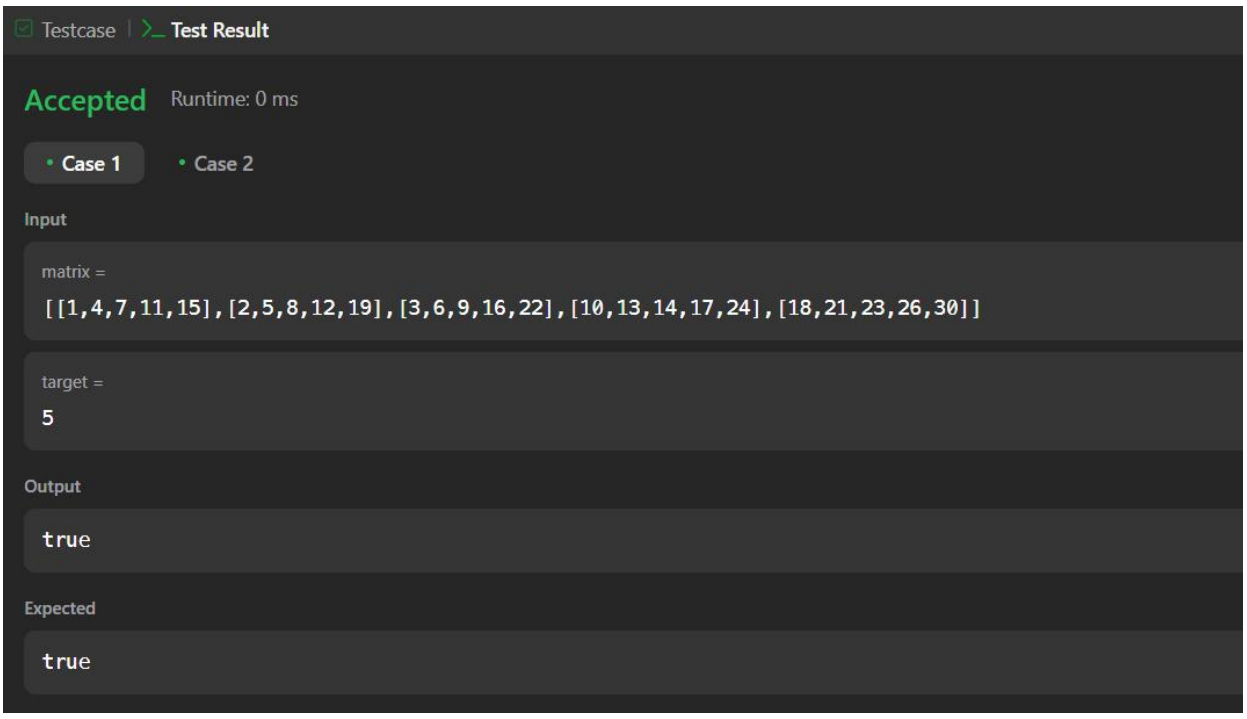
Integers in each column are sorted in ascending from top to bottom.

➢ **Code:**

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int n = matrix.length;
        int m = matrix[0].length;
        int row = n-1;
        int col = 0;
        while(row>=0 && col <m){
            if(matrix[row][col] == target){
                return true;
            }
            else if(matrix[row][col]>target){
                row--;
            }
```

```
            else{
                col++;
            }
        }
        return false;
    }
}
```

➢ **Output:**



3. **Problem - 3:**

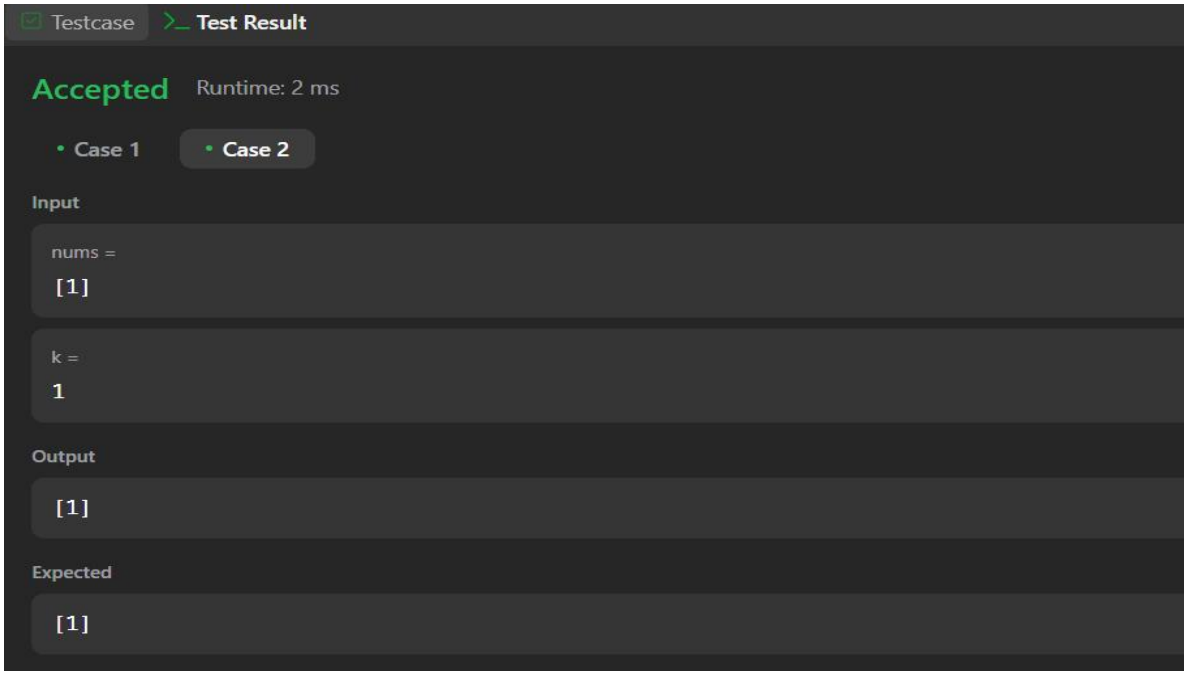
➢ **Top K Frequent Elements:**

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

➢ **Code:**

```
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 key : freqMap.keySet()) {
            minHeap.add(key);
            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;
    }
}
```

➢ **Output:**




## 4. Problem - 4:

➢ **Kth Largest Element in an Array:**

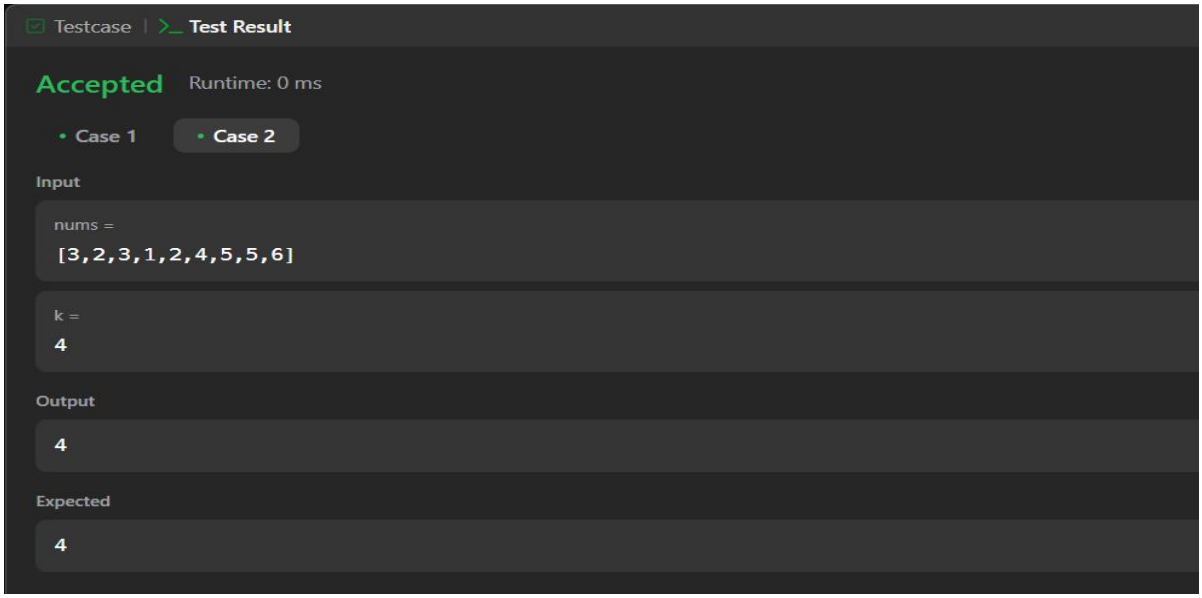
Given an integer array nums and an integer k, return the kth largest element in the array.
Note that it is the kth largest element in the sorted order, not the kth distinct element.
Can you solve it without sorting?

➢ **Code:**

```
import java.util.*;
class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : nums) {
            minHeap.add(num);
            if (minHeap.size() > k) {
                minHeap.poll();
```

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

➢ **Output:**




## 5. Problem - 5:

➢ **Median of Two Sorted Arrays:**

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)).
.

➢ **Code:**

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int n = nums1.length;
        int m = nums2.length;
        int i = n - 1;
        int j = m - 1;
        int z = n + m;
```

```
int k = z - 1;

int num[] = new int[z];

while(i >= 0 && j>=0 ){
    if(nums1[i] > nums2[j]){
        num[k] = nums1[i];
        i--;
    }

    else{
        num[k] = nums2[j];
        j--;
    }
    k--;
}

while (i >= 0) {
    num[k] = nums1[i];
    i--;
    k--;
}




while (j >= 0) {
    num[k] = nums2[j];
    j--;
    k--;
}

double result;
if(z%2 != 0){
    result = num[(z-1)/2];
}

else{
    double a = num[z/2] + num[(z/2)-1];
    result = a/2;
}
```

```
            return result;
        }
    }
```

➢ **Output:**


## 6. Problem - 6:

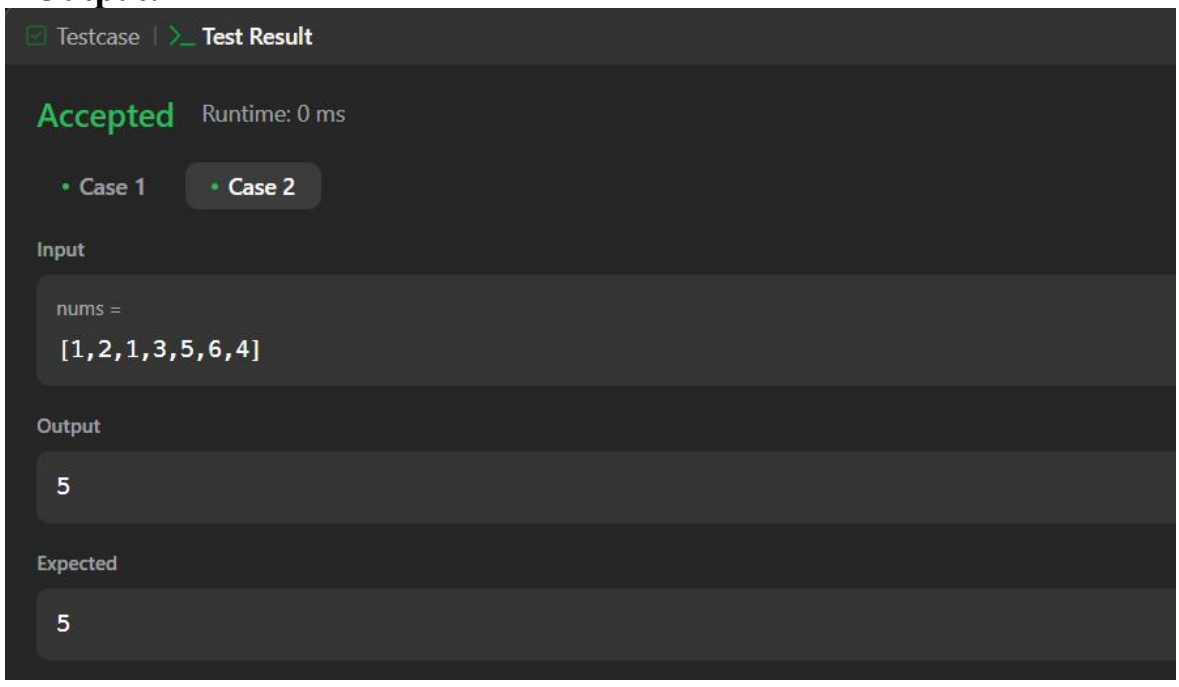
➢ **Find Peak Element:**

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.

➢ **Code:**

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

➢ **Output:**



7. **Problem - 7:**

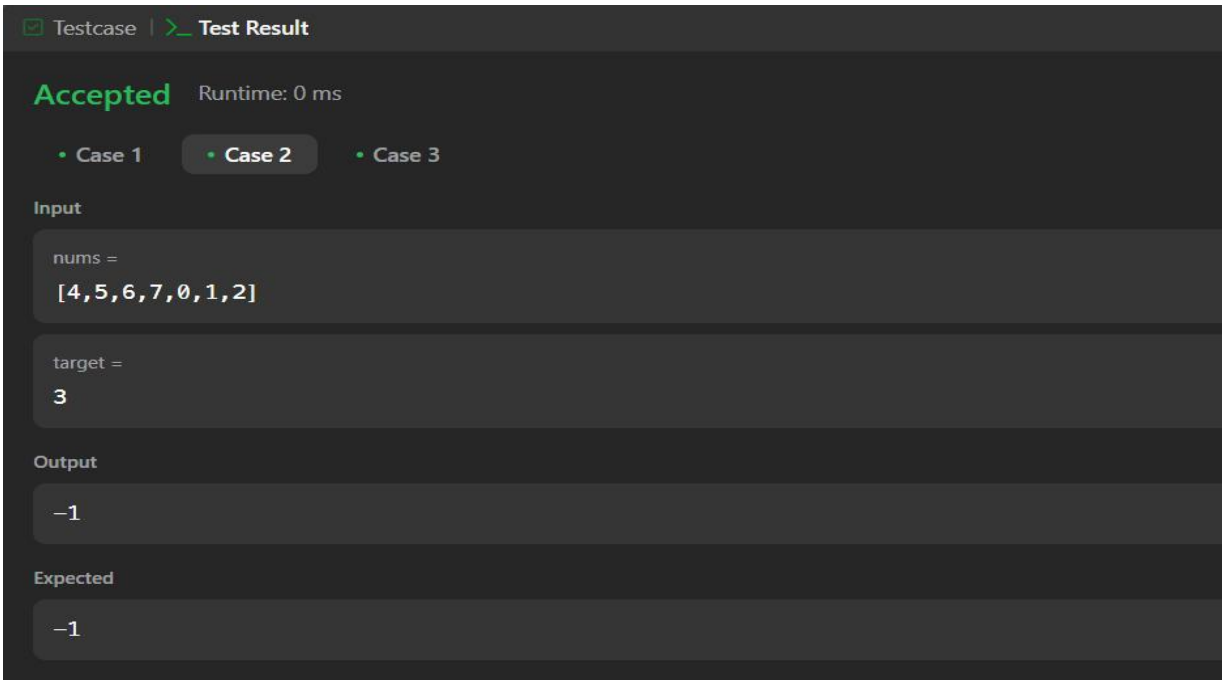
➢ **Search in Rotated Sorted Array:**

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.

➢ **Code:**

```
class Solution {
public int search(int[] nums, int target) {
    int low =0; int high = nums.length-1;
    while(low<=high){
        int mid = low + (high - low) / 2;
        if(nums[mid] == target) return mid;

        if(nums[low]<=nums[mid]){
            if(nums[low] <= target && target<= nums[mid]){
                high = mid - 1;

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

➢ **OutPut:**




❖ **Leraning Outcomes:**

- ➢ Efficient use of binary search for optimized searching.
- ➢ Application of heaps and sorting for selection problems.
- ➢ Techniques for merging and manipulating sorted data.
- ➢ Optimized matrix search and traversal strategies.
- ➢ In-place array modification and sorting variations.