Experiment-5

Student Name: Yasir Malik **UID :**22BCS13301
Branch: BE-CSE **Section/Gro up:**22BCS-IOT-640-B
Semester: 6th **Date of Performance:** 21/02/2025
Subject Name: AP Lab-2 **Subject Code:** 22CSH-359

1. **Aim:** Sorting and Searching

2. **Problem Statements:**

   Problem 1.1: 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.

   Problem 1.2: 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.

   Problem 1.3: Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

   Problem 1.4: 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*.
   Note that it is the k^{th} *smallest element* ***in the sorted order****, not the* k^{th} ***distinct*** *element.*

3. **Implementation/Code:**

Problem 1.1

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = m - 1;
```

```
        int j = n - 1;
        int k = m + n - 1;

        // Merge from the end of nums1
        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--];
        }
    }
}
```

Problem 1.2:

```
import java.util.Random;

class Solution {
    public int findKthLargest(int[] nums, int k) {
```

```
    return quickSelect(nums, 0, nums.length - 1, nums.length - k);
}

private int quickSelect(int[] nums, int left, int right, int kIndex) {
    if (left == right) return nums[left]; // Base case

    Random rand = new Random();
    int pivotIndex = left + rand.nextInt(right - left + 1);
    pivotIndex = partition(nums, left, right, pivotIndex);

    if (pivotIndex == kIndex) {
        return nums[pivotIndex];
    } else if (pivotIndex < kIndex) {
        return quickSelect(nums, pivotIndex + 1, right, kIndex);
    } else {
        return quickSelect(nums, left, pivotIndex - 1, kIndex); // Search left part
    }
}

private int partition(int[] nums, int left, int right, int pivotIndex) {
    int pivot = nums[pivotIndex];
    swap(nums, pivotIndex, right);
    int storeIndex = left;
```

```
        for (int i = left; i < right; i++) {
            if (nums[i] < pivot) {
                swap(nums, storeIndex, i);
                storeIndex++;
            }
        }
        swap(nums, storeIndex, right); // Move pivot to final position
        return storeIndex;
    }

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

Problem 1.3:

```
import java.util.*;

class Solution {
    public int[][] merge(int[][] intervals) {
        if (intervals.length <= 1) return intervals;
```

```
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

        List<int[]> merged = new ArrayList<>();

        int[] currentInterval = intervals[0];
        merged.add(currentInterval);

        for (int[] interval : intervals) {
            int currentEnd = currentInterval[1];
            int nextStart = interval[0];
            int nextEnd = interval[1];

            if (nextStart <= currentEnd) {

                currentInterval[1] = Math.max(currentEnd, nextEnd);
            } else {

                currentInterval = interval;
                merged.add(currentInterval);
            }
        }

        return merged.toArray(new int[merged.size()][]);
    }
```

```
}
```

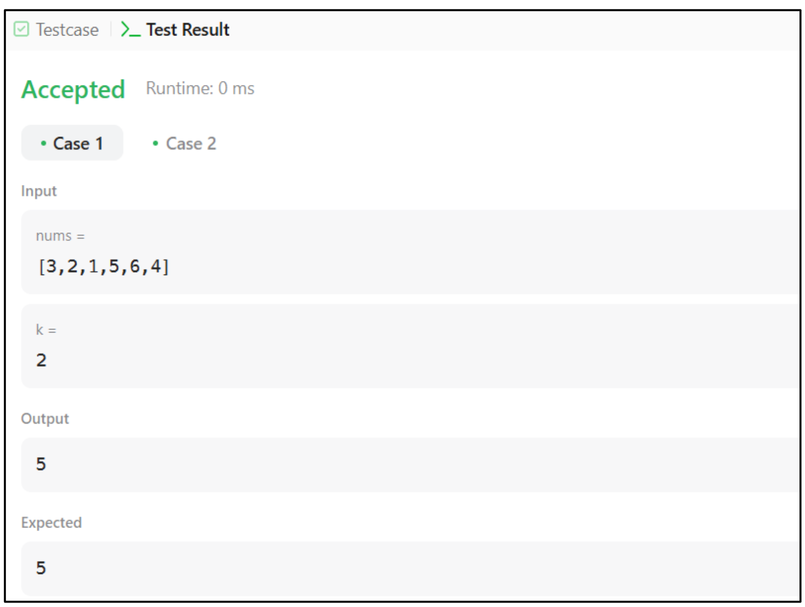
Problem 1.4:

```
import java.util.PriorityQueue;

public class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;

        PriorityQueue<int[]> minHeap = new PriorityQueue<>((a, b) -> a[0] - b[0]);

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

        for (int i = 0; i < k - 1; i++) {
            int[] curr = minHeap.poll();
            int val = curr[0], row = curr[1], col = curr[2];

            if (col + 1 < n) {
                minHeap.offer(new int[]{matrix[row][col + 1], row, col + 1});
            }
        }
```

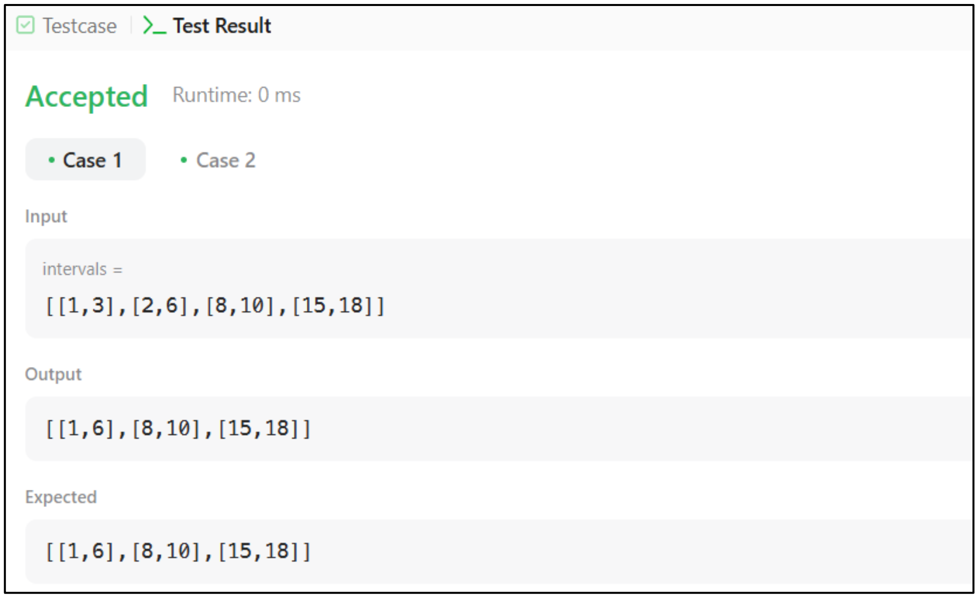
```
        return minHeap.poll()[0];
    }

    public static void main(String[] args) {
        Solution sol = new Solution();

        int[][] matrix1 = {
            {1, 5, 9},
            {10, 11, 13},
            {12, 13, 15}
        };
        System.out.println(sol.kthSmallest(matrix1, 8)); // Output: 13

        int[][] matrix2 = {
            {-5}
        };
        System.out.println(sol.kthSmallest(matrix2, 1)); // Output: -5
    }
}
```

4. Output:

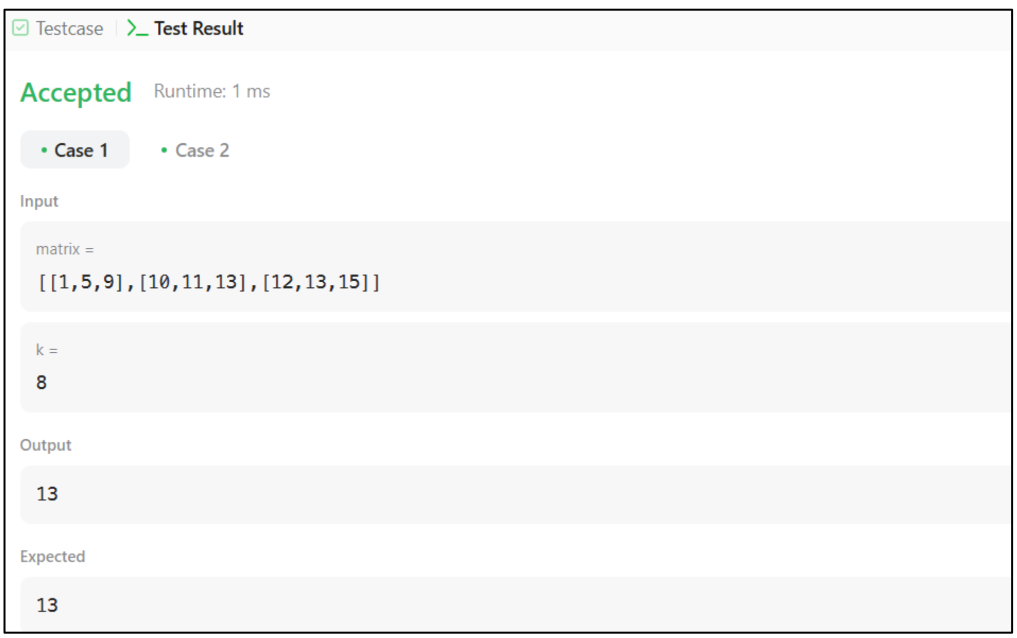
(Fig. 1- Problem 1.1 Output)



(Fig. 2- Problem 1.2 Output)



(Fig. 3- Problem 1.3 Output)



(Fig. 4- Problem 1.4 Output)

5. Learning Outcome:

1. Understand how to merge two sorted arrays into one sorted array in-place, using the two-pointer technique.
2. Learn how to find the Kth largest element in an array using the Quick Select algorithm, which is an optimized version of QuickSort.
3. Learn how to merge overlapping intervals by sorting and comparing the intervals.
4. Understand how to efficiently find the Kth smallest element in a matrix, where each row is sorted, using a min-heap (priority queue).