Experiment-5

Student Name: Mohit	**UID:** 22BCS14664
Branch: BE-CSE	**Section/Group:** 22BCS-IOT-640-B
Semester: 6th	**Date of Performance:** 04/02/2025
Subject Name: AP Lab-2	**Subject Code:** 22CSH-359

1. **Aim:** Sorting And Searching.

2. **Problem Statements:**
Problem 1.1: 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)).

3. **Problem 1.2:** 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.

4. **Problem 1.3:** 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.

5. **Problem 1.4:** 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.

3. Implementation/Code:

Problem 1.1

```
import java.util.*;


class Solution {

    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int n1 = nums1.length;
        int n2 = nums2.length;
        int n = n1 + n2;
        int[] new_arr = new int[n];

        int i=0, j=0, k=0;

        while (i<=n1 && j<=n2) {
            if (i == n1) {
      }
                while(j<n2) new_arr[k++] = nums2[j++];
                break;
            } else if (j == n2) {
                while (i<n1) new_arr[k++] = nums1[i++];
                break;
            }

            if (nums1[i] < nums2[j]) {
                new_arr[k++] = nums1[i++];
            } else {
                new_arr[k++] = nums2[j++];
            }
        }

        if (n%2==0) return (float)(new_arr[n/2-1] + new_arr[n/2])/2;
        else return new_arr[n/2];
    }
 }
```

Problem 1.2:

```
import java.util.*;

class Solution {


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

        // Sort by ascending starting point
        Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0], i2[0]));

        List<int[]> result = new ArrayList<>();
        int[] newInterval = intervals[0];
        result.add(newInterval);
        for (int[] interval : intervals) {
            if (interval[0] <= newInterval[1]) // Overlapping intervals, move the end if
needed
                newInterval[1] = Math.max(newInterval[1], interval[1]);
            else {                           // Disjoint intervals, add the new interval to the list
                newInterval = interval;
                result.add(newInterval);
            }
        }

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

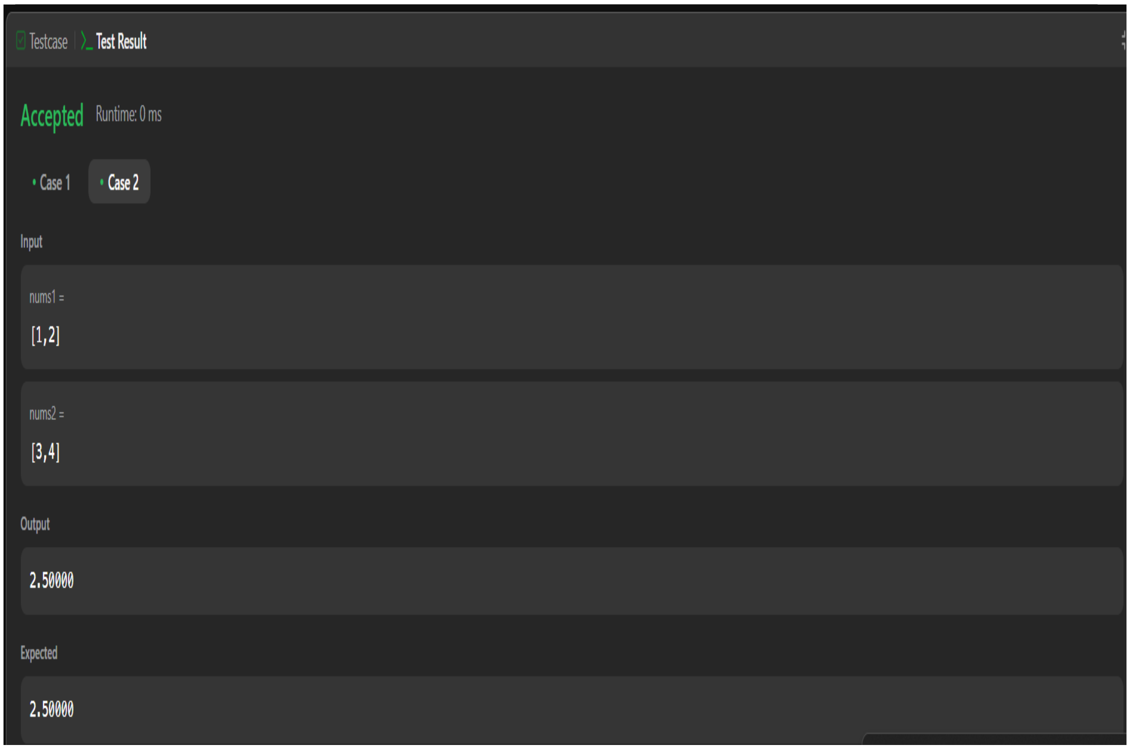
Problem 1.3:

```
import java.util.*;
class Solution {
    public int findPeakElement(int[] nums) {
        int n=nums.length;
        if(nums.length==1)return 0;
        if (nums[0] > nums[1]) return 0; // Peak at the beginning
        if (nums[n - 1] > nums[n - 2]) return n - 1; // Peak at the end
        int low=1;
        int high=nums.length-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 low;
    }
}
```

Problem 1.4:

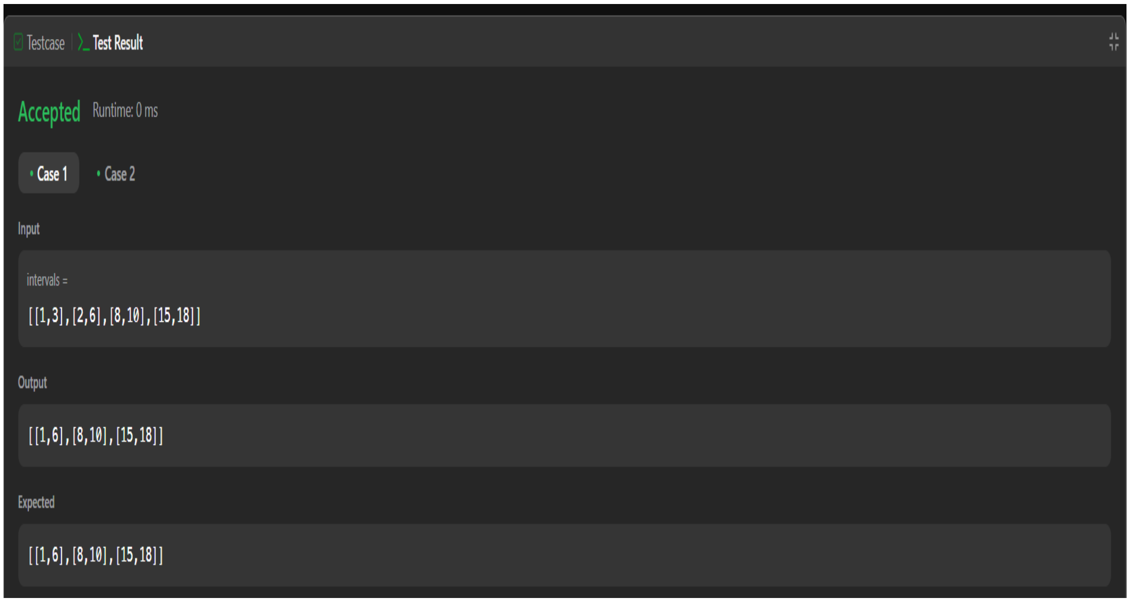
```
import java.util.*;
class Solution {
public void merge(int[] nums1, int m, int[] nums2, int n) {
     for (int i = 0; i < n; i++) {
          nums1[m + i] = nums2[i];
      }
      Arrays.sort(nums1);
   }
}
```

4. Output:



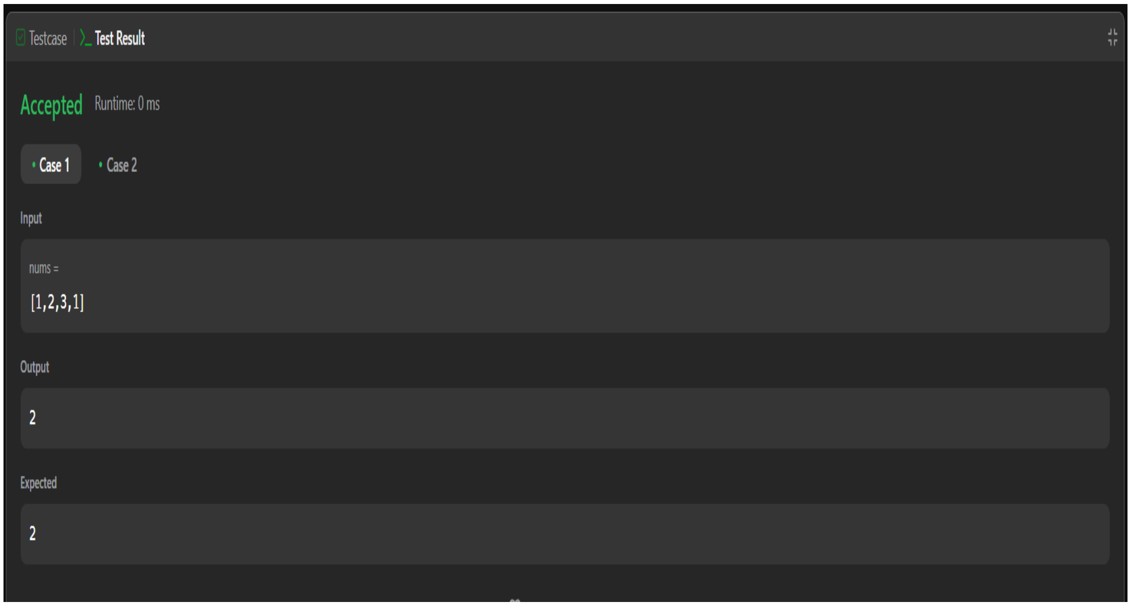
```
Testcase | Test Result

Accepted Runtime: 0 ms

Case 1   Case 2

Input
nums1 =
[1,2]
nums2 =
[3,4]

Output
2.50000

Expected
2.50000
```

(Fig. 1- Problem 1.1 Output)



```
Testcase | Test Result

Accepted Runtime: 0 ms

Case 1   Case 2

Input
intervals =
[[1,3],[2,6],[8,10],[15,18]]

Output
[[1,6],[8,10],[15,18]]

Expected
[[1,6],[8,10],[15,18]]
```

(Fig. 2- Problem 1.2 Output)



Accepted Runtime: 0 ms

Input

nums =

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

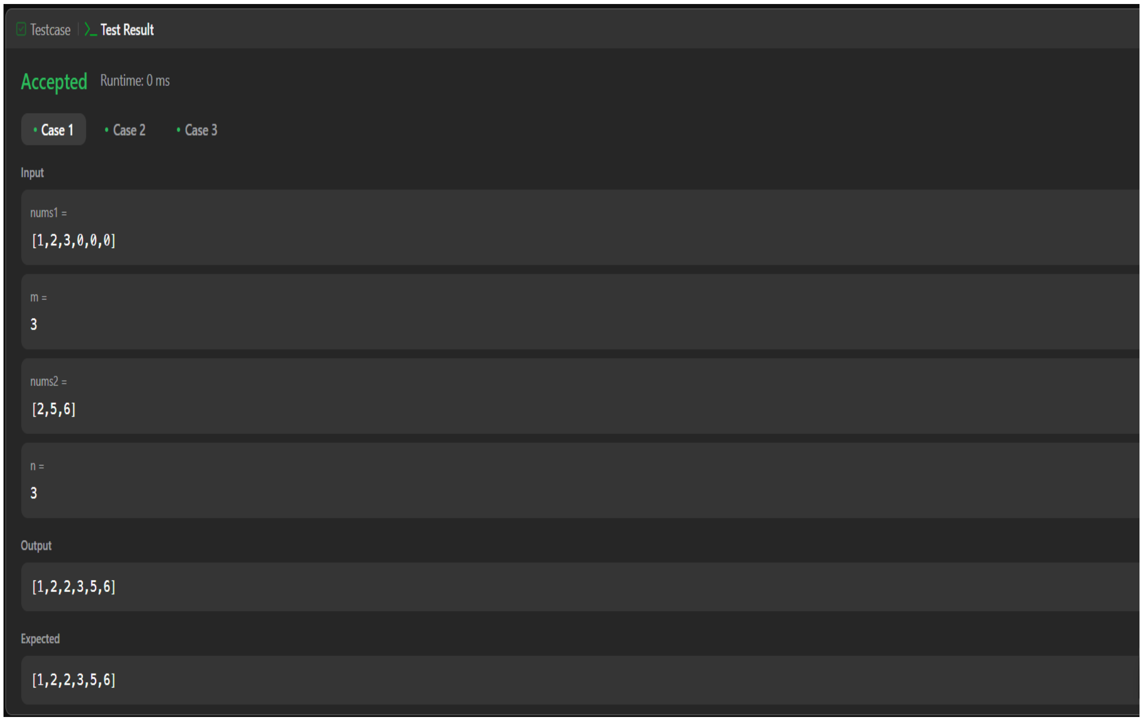
Output

```
2
```

Expected

```
2
```

(Fig. 3- Problem 1.3 Output)



Accepted Runtime: 0 ms

Input

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]
```

(Fig. 4- Problem 1.4 Output)

5. Learning Outcome:

1. Learn how to use priority queues (heaps) to efficiently track dynamic height changes in skyline problems.
2. Learn how merge sort can be extended to solve problems beyond sorting, such as counting important reverse pairs.
3. Learn how to efficiently traverse a sorted 2D matrix using a strategic approach from the top-right corner.
4. Learn how to divide and conquer a string problem by recursively identifying substrings that satisfy given conditions.