Experiment-5

Name: Manya Dhingra
UID: 22BET10205
Branch: BE-IT
Section/Group: 22BET_IOT-702/B
Semester: 6^{th}
Date of Performance: 21/02/25
Subject Name: AP LAB-II
Subject Code: 22ITP-351

Problem-1

1.Aim:

Given an array of intervals where intervals[i] = [$start_i$, end_i], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

2.Objective:

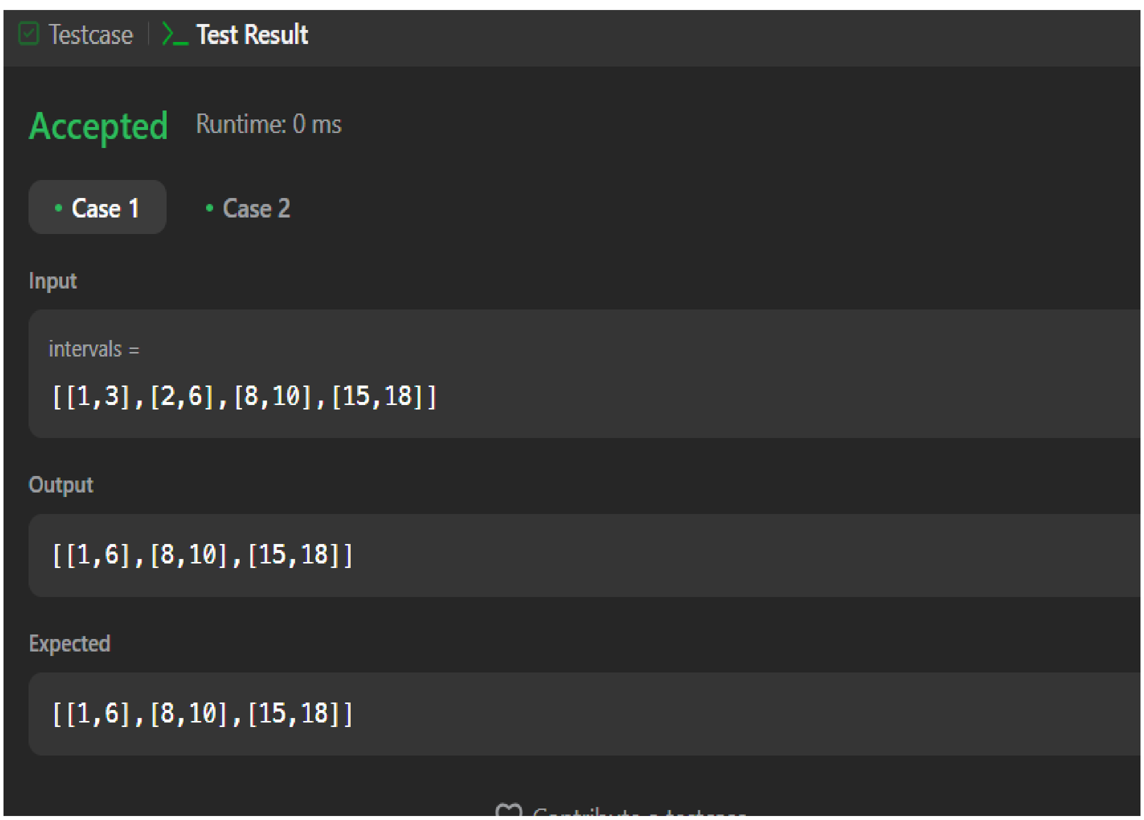
- Sort intervals by start time.
- Iterate through intervals and merge overlapping ones.
- Store merged intervals in a new list.
- Time Complexity: O(nlogn)) due to sorting.

3.Code:

```
class Solution {

public:

    vector<vector<int>> merge(vector<vector<int>>& intervals) {

        if (intervals.empty()) return {};

        sort(intervals.begin(), intervals.end());

        vector<vector<int>> merged;

        merged.push_back(intervals[0]);

        for (int i = 1; i < intervals.size(); ++i) {

            if (intervals[i][0] <= merged.back()[1]) {

                // Merge overlapping intervals

                merged.back()[1] = max(merged.back()[1], intervals[i][1]);
```

```
            } else {
                merged.push_back(intervals[i]);
            }
        }

        return merged;
    }
};
```

4.Output:



Problem-2

1.Aim:

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.

2.Objective:

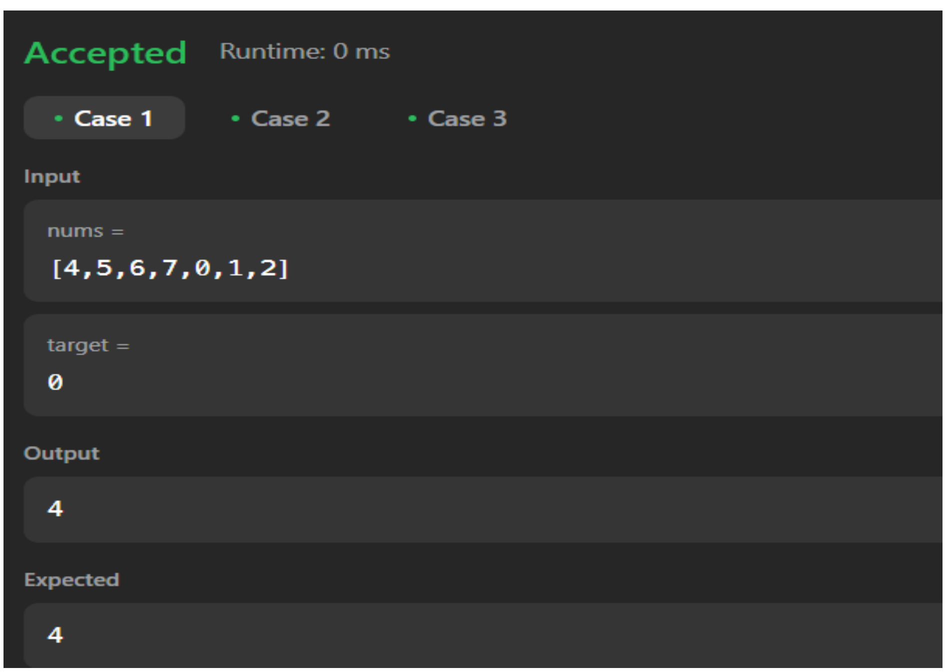
- Use binary search to identify which half of the array is sorted.
- If the target lies within the sorted half, search there.
- Otherwise, search in the other half.
- Time Complexity: O(log(n))due to binary search.

3.Code:

```
#include <vector>

using namespace std;

class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) return mid;

            if (nums[left] <= nums[mid]) {
                if (nums[left] <= target && target < nums[mid]) {
```

```
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1; }}}

    return -1; }
};
```

4.Output:



Problem-3

1.Aim: Given an integer array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3].

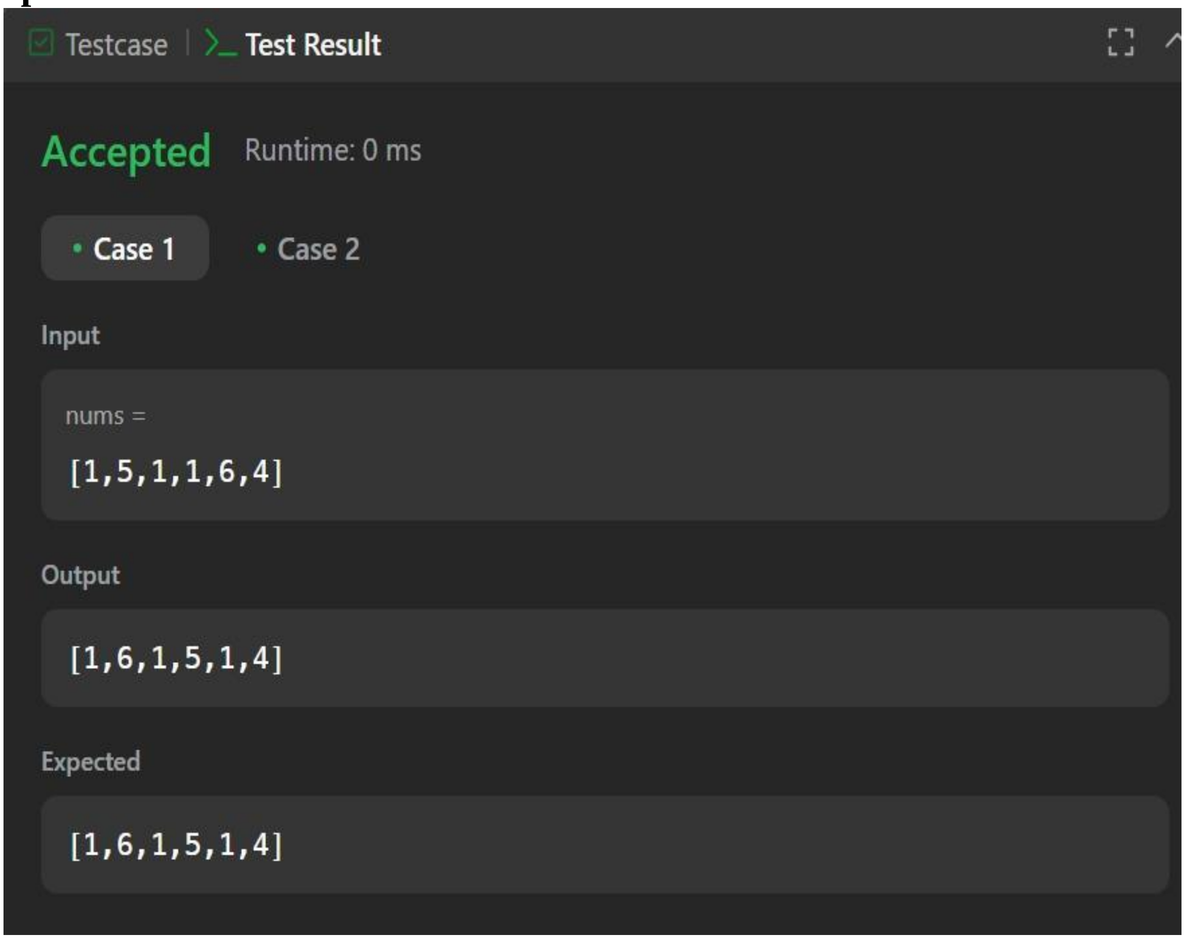
2.Objective:

- Develop an efficient algorithm that rearranges the elements of the array to ensure alternating peaks and valleys, thereby maintaining the given wiggle pattern.
- Ensure the implementation works within the given constraints and handles edge cases effectively.
- Optimize the algorithm to achieve the best possible time complexity for practical use.

3.Code:

```
class Solution {

public:

    void wiggleSort(vector<int>& nums) {

        vector<int> sorted_nums = nums;

        sort(sorted_nums.begin(), sorted_nums.end());

        int n = nums.size();

        int mid = (n - 1) / 2;

        int end = n - 1;

        for (int i = 0; i < n; i++) {

            nums[i] = (i % 2 == 0) ? sorted_nums[mid--] : sorted_nums[end--];

        }

    }

};
```

4.Output:


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Input
nums =
[1,5,1,1,6,4]
Output
[1,6,1,5,1,4]
Expected
[1,6,1,5,1,4]

Problem-4

1.Aim:

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.

2.Objective:

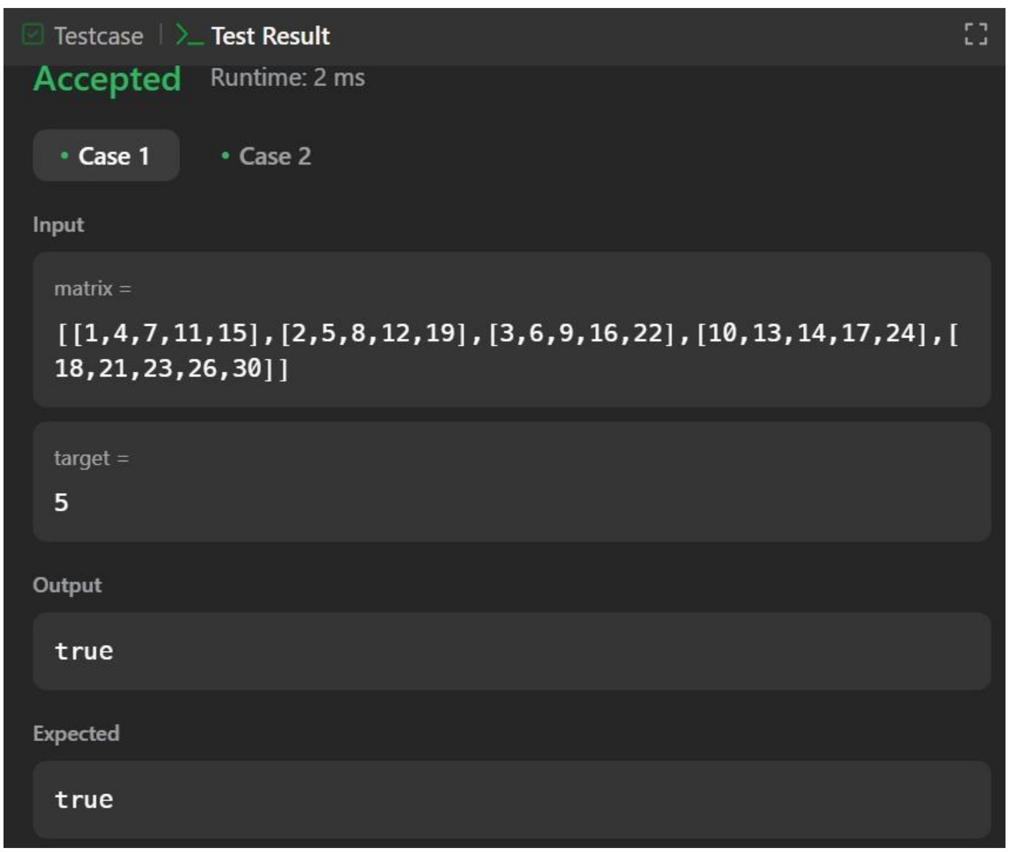
- Develop an efficient search algorithm that finds a target value in an m x n matrix while utilizing its sorted properties.
- Implement a method that ensures an optimal time complexity better than a brute-force approach.
- Utilize a binary search or step-wise elimination technique to efficiently locate the target.

3.Code:

```
class Solution {

 public:

     bool searchMatrix(vector<vector<int>>& matrix, int target) {

         if (matrix.empty() || matrix[0].empty())

             return false;



         int m = matrix.size(), n = matrix[0].size();

         int row = 0, col = n - 1;



         while (row < m && col >= 0) {

             if (matrix[row][col] == target)

                 return true;

             else if (matrix[row][col] > target)

                 col--;

             else
```

```
            row++;
        }

        return false;
    }
};
```

4.Output:



Problem-5

1.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.

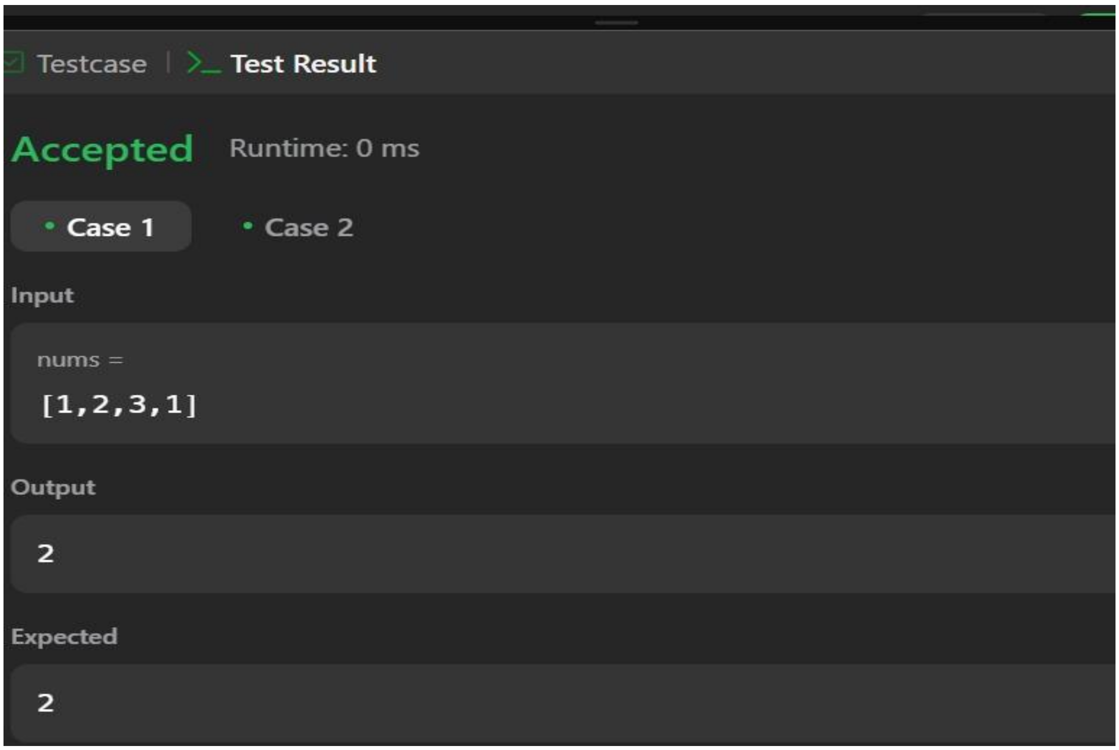
2.Objective:

- Develop an efficient algorithm to identify a peak element in an array.
- Ensure the algorithm runs in O(log n) time complexity using binary search.
- Handle edge cases where the peak might be at the beginning or end of the array.
- Allow flexibility in returning any peak element when multiple peaks exist.

3.Code:

```
class Solution {

public:

    int findPeakElement(vector<int>& nums) {

        int left = 0, 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;

    }

};
```

4.Output:


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Input
nums =
[1,2,3,1]
Output
2
Expected
2

Problem-6

1.Aim:

To find the median of two sorted arrays without merging them, using binary search, in O(log(min(m, n))) time complexity.

2.Objective:

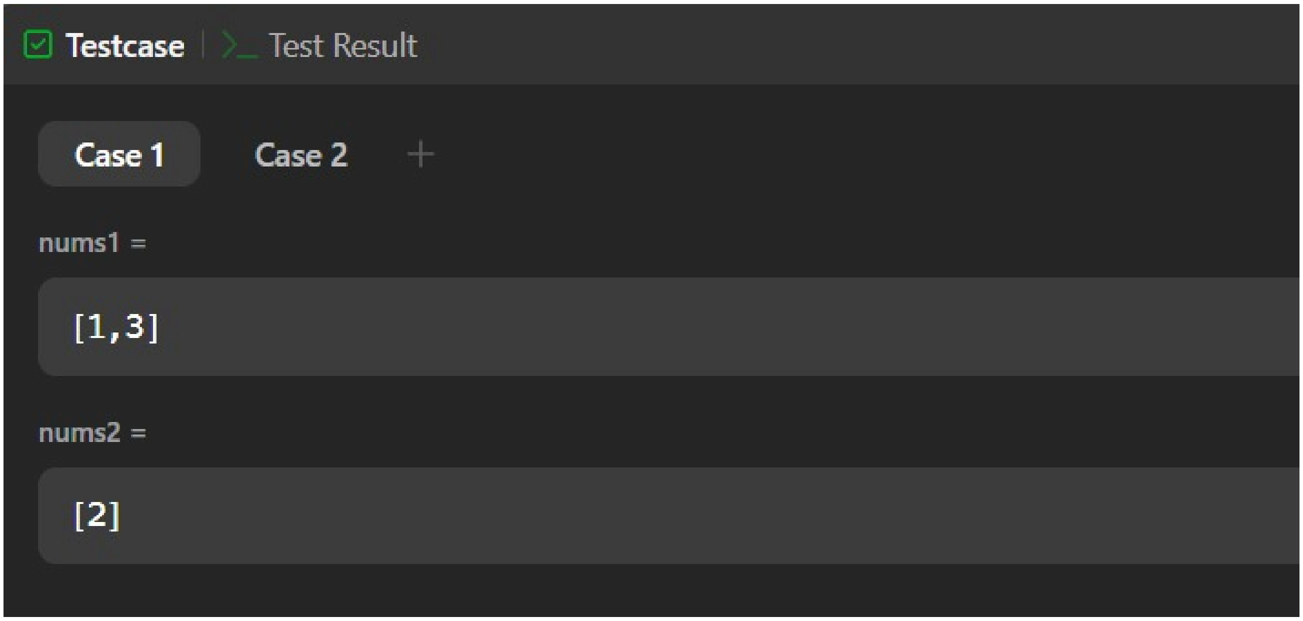
- Reduce the problem size by searching only in the smaller array.
- The algorithm swaps arrays to always perform binary search on the smaller array.

3.Code:

```
class Solution {

public:

    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

      if (nums1.size() > nums2.size()) {

          return findMedianSortedArrays(nums2, nums1);

       }



       int x = nums1.size();

       int y = nums2.size();

       int low = 0, high = x;

       while (low <= high) {

          int partitionX = (low + high) / 2;

          int partitionY = (x + y + 1) / 2 - partitionX;



          int maxLeftX = (partitionX == 0) ? INT_MIN : nums1[partitionX - 1];

          int minRightX = (partitionX == x) ? INT_MAX : nums1[partitionX];
```

```
            int maxLeftY = (partitionY == 0) ? INT_MIN : nums2[partitionY - 1];
            int minRightY = (partitionY == y) ? INT_MAX : nums2[partitionY];

            if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
                if ((x + y) % 2 == 0) {
                    return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2.0;
                } else {
                    return max(maxLeftX, maxLeftY);
                }
            } else if (maxLeftX > minRightY) {
                high = partitionX - 1;
            } else {
                low = partitionX + 1;
            }
        }
        return 0.0;
    }
};
```

4.Output:



Problem-7

1.Aim:

To efficiently find the k-th smallest element in an n × n sorted matrix, where each row and column is sorted in ascending order, using Binary Search

2.Objective:

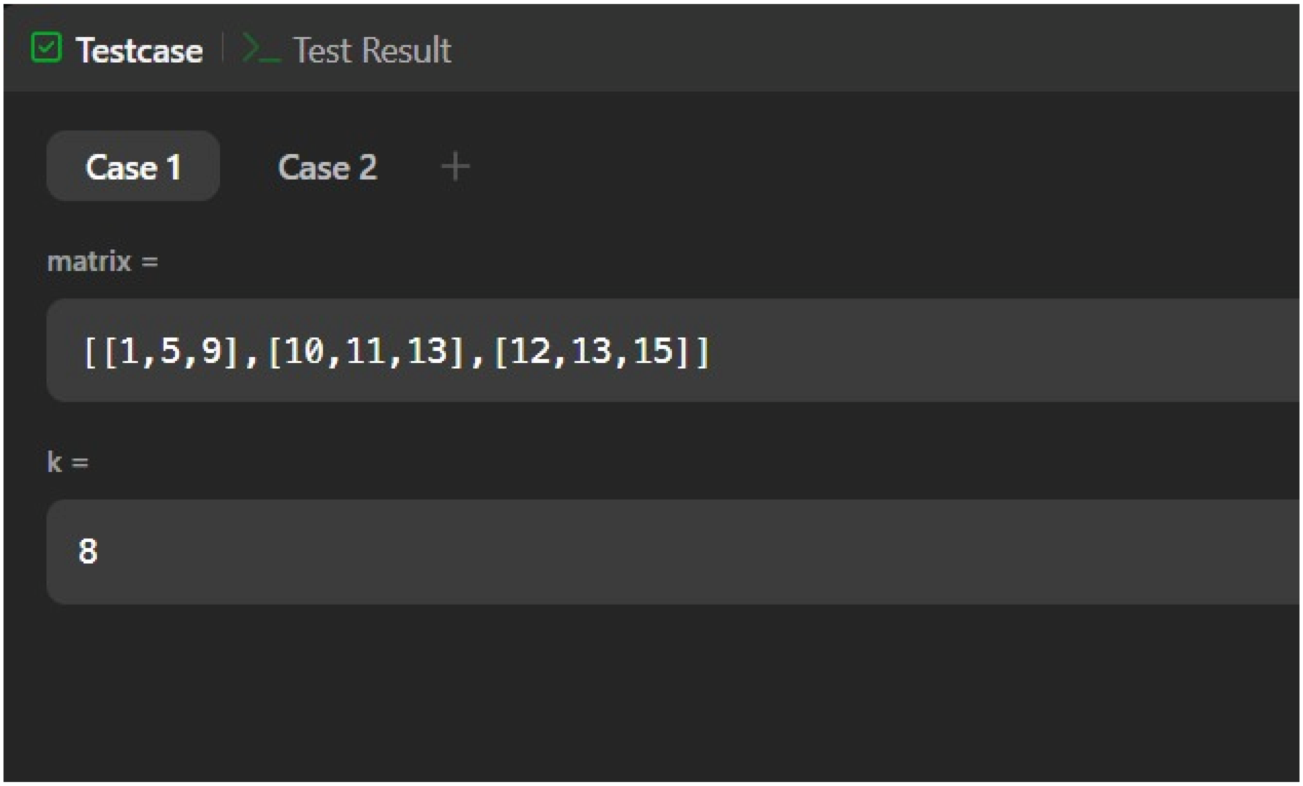
- Implement an approach better than $O(n^2)$ (which is the brute-force approach).
- Implement Binary Search or Min-Heap (Priority Queue) to achieve an efficient solution.

3.Code:

```
class Solution {

public:

    int kthSmallest(vector<vector<int>>& matrix, int k) {

        int n = matrix.size();

        int left = matrix[0][0], right = matrix[n - 1][n - 1];



        while (left < right) {

            int mid = left + (right - left) / 2;

            int count = countLessEqual(matrix, mid, n);



            if (count < k) {

                left = mid + 1;

            } else {

                right = mid;

            }

        }

        return left;

    }
```

```
    private:
      int countLessEqual(vector<vector<int>>& matrix, int mid, int n) {
        int count = 0, row = n - 1, col = 0;

        while (row >= 0 && col < n) {
          if (matrix[row][col] <= mid) {
            count += (row + 1); // All elements in this column above are also ≤ mid
            col++; // Move right
          } else {
            row--; // Move up
          }
        }
        return count;


      }
};
```

4.Output:


Testcase
Test Result
Case 1
Case 2
matrix =
[[1,5,9],[10,11,13],[12,13,15]]
k =
8