Experiment 5

Student Name: Nisha Kumari **UID: 22BET10118**

Branch: IT **Section/Group: 22BET_IOT-701/A**

Semester: 6th **Date of Performance:21.02.25**

Subject Name: AP Lab - 2 **Subject Code: 22ITP-351**

1. **Aim:** The aim is to enhance problem-solving skills by solving diverse LeetCode problems using techniques like binary search, dynamic programming, and advanced data structures.
   i.) Find Peak Element
   ii.) Merge Intervals
   iii.) Search in Rotated Sorted Array
   iv.) Search 2d matrix 2
   v.) Kth smallest element in a sorted matrix
   vi.) Merge Sorted Array
   vii.) Median of Two Sorted Arrays

2. **Objective:**
   - Apply binary search to efficiently solve search-related problems.
   - Utilize dynamic programming for optimized solutions to overlapping subproblems.
   - Implement divide and conquer strategies for complex problem breakdowns.
   - Master bit manipulation for efficient low-level computations.
   - Use advanced data structures like heaps and trees for optimal performance.
   - Develop skills to handle sorted arrays and matrices effectively.
   - Improve algorithmic thinking through solving real-world coding challenges.

3. **Code:**

Problem 1: Find Peak Element

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

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

```
            // Check if the mid element is greater than its next element
            if (nums[mid] > nums[mid + 1]) {
                // Peak is on the left side (including mid)
                right = mid;
            } else {
                // Peak is on the right side
                left = mid + 1;
            }
        }
        // left == right, pointing to a peak element
        return left;
    }
};
```

Problem 2: Merge Intervals

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end());
        vector<vector<int>> merged;
         merged.push_back(intervals[0]);
         // Step 2: Merge overlapping intervals
        for (int i = 1; i < intervals.size(); ++i) {
        // If the current interval overlaps with the last interval in merged
        if (intervals[i][0] <= merged.back()[1]) {
            // Merge by updating the end of the last interval
            merged.back()[1] = max(merged.back()[1], intervals[i][1]);
        } else {
            // No overlap, add the interval to merged
            merged.push_back(intervals[i]);
        }
    }
    return merged;
}
// Function to print the intervals
void printIntervals(const vector<vector<int>>& intervals) {
    for (const auto& interval : intervals) {
        cout << "[" << interval[0] << ", " << interval[1] << "] ";
    }
    cout << endl;
```

```
}
};
```

Problem 3: Search in Rotated Sorted Array

```
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;
        // Found the target
        if (nums[mid] == target) {
            return mid;
        }
        // Check if the left half is sorted
        if (nums[left] <= nums[mid]) {
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;  // Search in the left half
            } else {
                left = mid + 1;  // Search in the right half
            }
        }
        // Right half is sorted
        else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;  // Search in the right half
            } else {
                right = mid - 1;  // Search in the left half
            }
        }
    }
    return -1;  // Target not found
}
};
```

Problem 4: Search a 2D Matrix II

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size();
```

```
        if (m == 0) return false;
        int n = matrix[0].size();
        // Start from the top-right corner
        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--;  // Move left
            } else {
                row++;  // Move down
            }
        }
        return false;
    }
};
```

Problem 5: Kth Smallest Element in a sorted matrix

```
class Solution {
public:
    int countLessEqual(vector<vector<int>>& matrix, int mid, int n) {
        int elementCount = 0;
        int row = n - 1;  // Start from the bottom-left corner
        int col = 0;

        while (row >= 0 && col < n) {
            if (matrix[row][col] <= mid) {
                elementCount += row + 1;  // Count all elements in the current column up to 'row'
                col++;
            } else {
                row--;  // Move up
            }
        }
        return elementCount;
    }

    // Only ONE definition of kthSmallest should exist
    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int n = matrix.size();
        int left = matrix[0][0];
```

```
        int right = matrix[n - 1][n - 1];

        // Binary search on the value range
        while (left < right) {
            int mid = left + (right - left) / 2;
            int elementCount = countLessEqual(matrix, mid, n);

            if (elementCount < k) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    }
};
```

Problem 6: Merge Array

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int i = m - 1;         // Last element in nums1's initial part
        int j = n - 1;         // Last element in nums2
        int k = m + n - 1;     // Last position in nums1

        // Start merging from the back
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
        // If elements are left in nums2, copy them
        while (j >= 0) {
            nums1[k--] = nums2[j--];
        }
        // No need to copy nums1 elements since they're already in place
    }
};
```

Problem 7: Merge Two Sorted Array

```
class Solution {
public:
   double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
      if (nums1.size() > nums2.size()) {
          return findMedianSortedArrays(nums2, nums1);  // Ensure nums1 is the smaller
array
      }
      int m = nums1.size();
      int n = nums2.size();
      int totalLeft = (m + n + 1) / 2;
      int left = 0;
      int right = m;
      while (left < right) {
         int i = left + (right - left + 1) / 2;
         int j = totalLeft - i;
         if (nums1[i - 1] > nums2[j]) {
            right = i - 1;
         } else {
            left = i;
         }
      }
      int i = left;
      int j = totalLeft - i;
      int nums1LeftMax = i == 0 ? INT_MIN : nums1[i - 1];
      int nums1RightMin = i == m ? INT_MAX : nums1[i];
      int nums2LeftMax = j == 0 ? INT_MIN : nums2[j - 1];
      int nums2RightMin = j == n ? INT_MAX : nums2[j];
      if ((m + n) % 2 == 1) {
         return max(nums1LeftMax, nums2LeftMax);
      } else {
          return (double)(max(nums1LeftMax, nums2LeftMax) + min(nums1RightMin,
nums2RightMin)) / 2;
      }
   }
};
```

4. **Output:**

Fig 1. Find Peak element

Fig 2. Merge Intervals

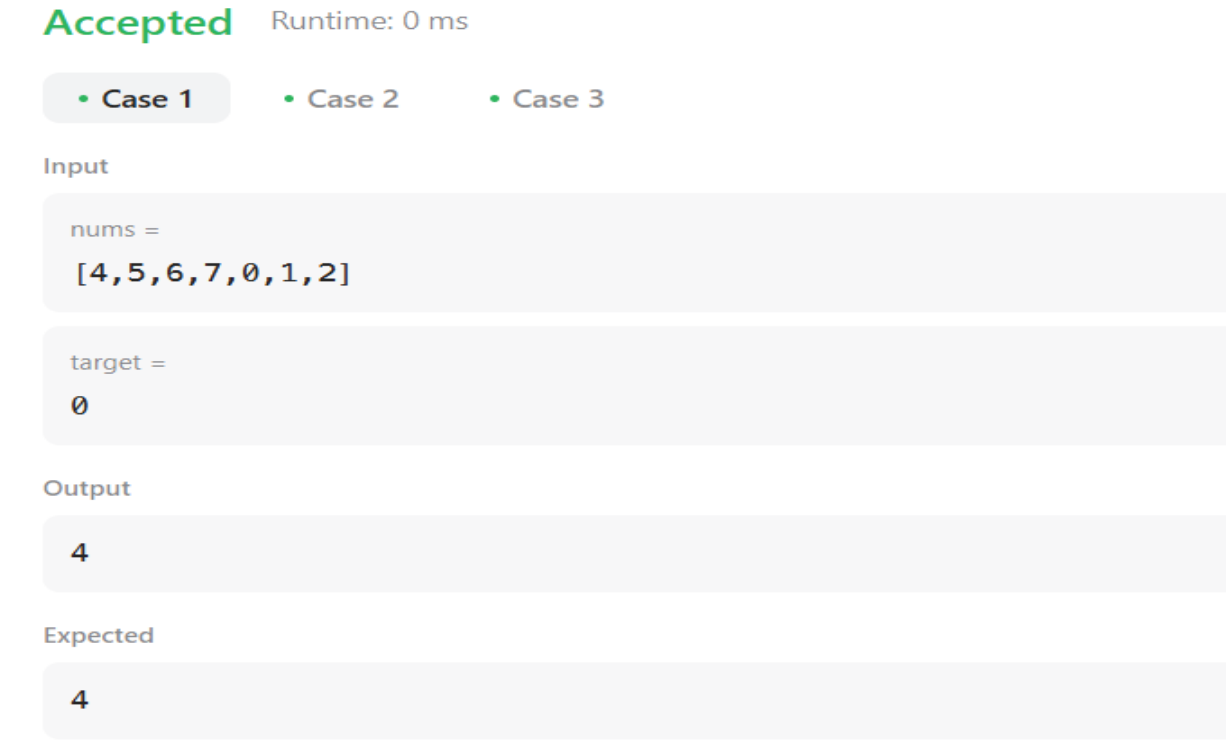

Fig 3. Search in Rotated Sorted Array

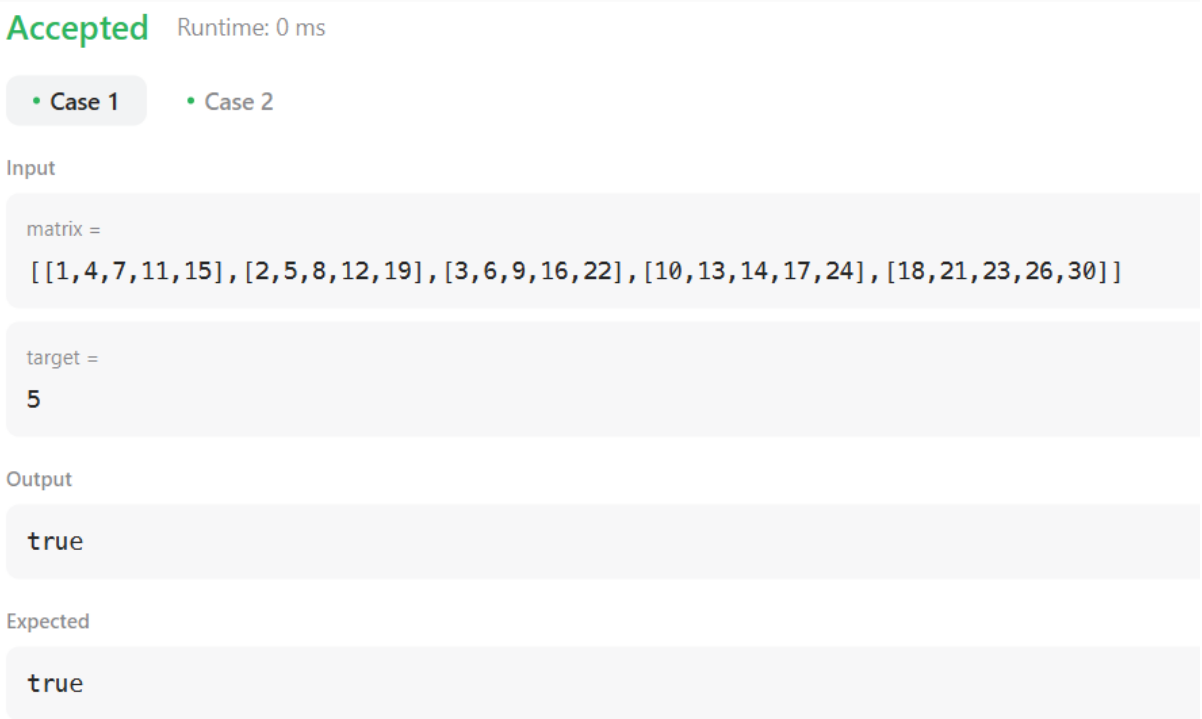

Fig 4. Search a 2D Matrix

Fig 5. Kth Smallest Element in a Sorted Matrix

Fig 6. Merge Sorted Array

Fig 7. Median of Sorted Array

5. Learning Outcomes:

- Understand and apply binary search in various problem scenarios.
- Solve problems using dynamic programming techniques.
- Break down complex problems using divide and conquer strategies.
- Perform efficient calculations using bit manipulation.
- Implement and utilize advanced data structures like heaps and trees.
- Merge and search in sorted arrays and matrices effectively.
- Develop faster and more efficient problem-solving skills.