Experiment-5

Student Name: Dhruv Vasudev **UID:22BCS17110**
Branch:BE CSE **Section/Group:634/A**
Semester:6th **Date of Performance:20/2/25**
Subject Name:AP-2 **Subject Code:22CSP-351**

1. **Aim:**
   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.
2. **Code:**

```
class Solution {
public:
    int search(std::vector<int>& nums, int target) {
        int low = 0, high = nums.size() - 1;
        while (low <= high) {
            int mid = (low + high) / 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;
                }
```

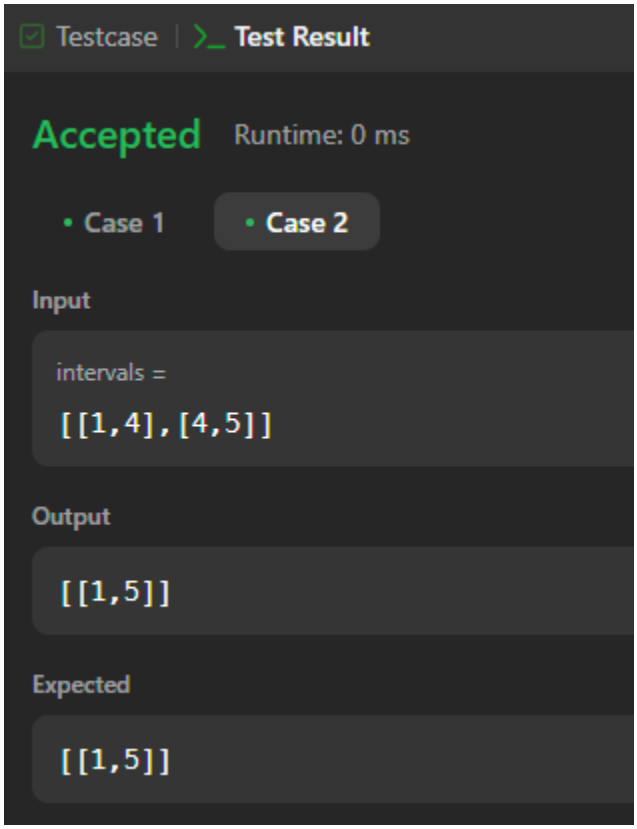
```
            } else {
                if (nums[mid] < target && target <= nums[high]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }}
        return -1;
    }
};
```

3. **Output:**

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. **Code:**

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
      if(intervals.size()==1)
        return intervals;
      vector<pair<int,int>> p;
      for(int i=0;i<intervals.size();i++)
      {
         p.push_back({intervals[i][0],intervals[i][1]});}
      sort(p.begin(),p.end());
      vector<vector<int>> ans;
      int f=p[0].first,s=p[0].second;
      for(int i=0;i<p.size()-1;i++)
      {
         vector<int> a(2);
         if(s>=p[i+1].first)
         {s=max(s,p[i+1].second);}
         else
         {
            a[0]=f;
            a[1]=s;
            f=p[i+1].first;
            s=p[i+1].second;
            ans.push_back(a);}}
      int n=intervals.size();
      ans.push_back({f,s});
      return ans;}};
```

3. **Output:**

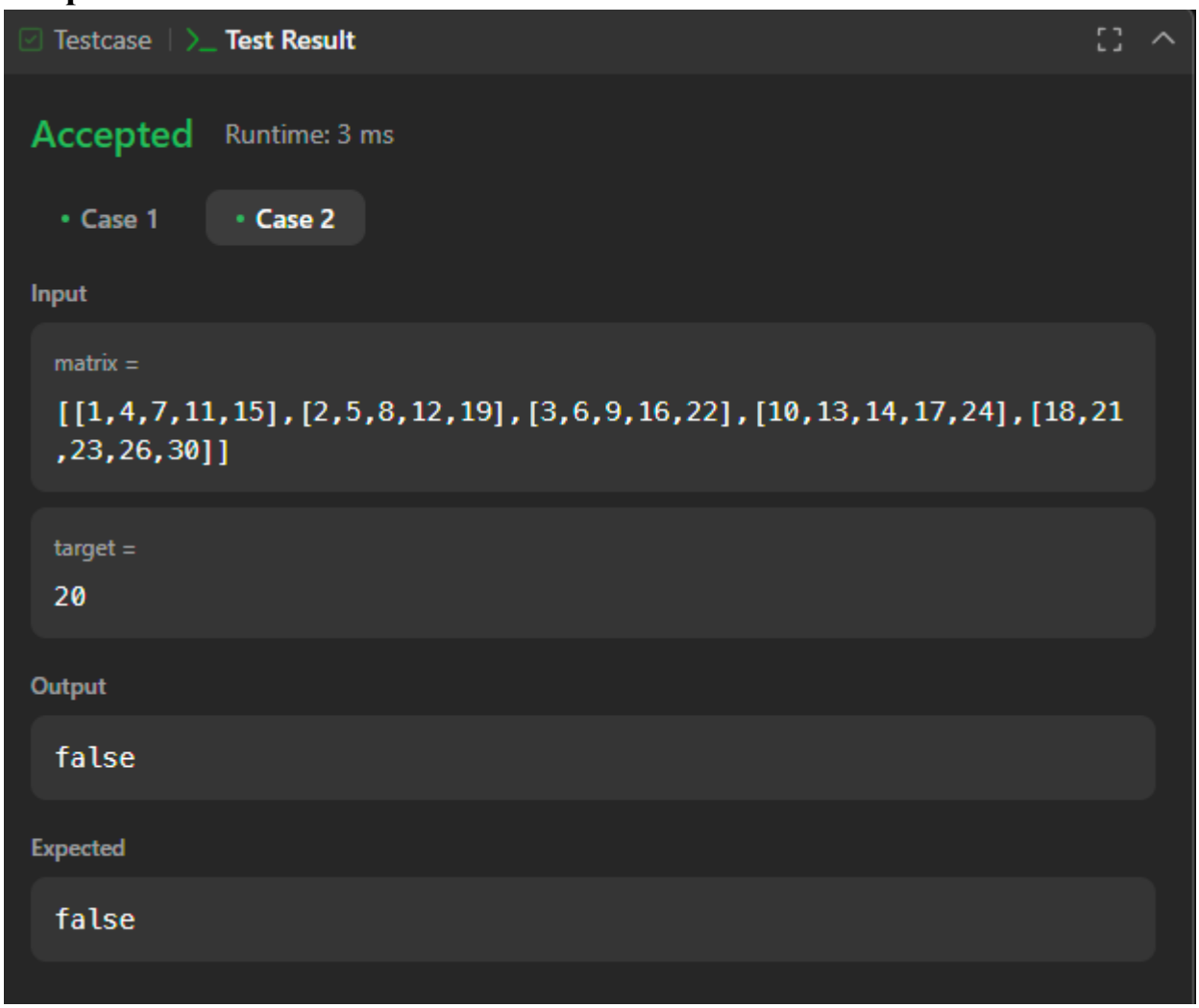


1. **Aim:**
   Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:
   Integers in each row are sorted in ascending from left to right.
   Integers in each column are sorted in ascending from top to bottom.
2. **Code:**

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size(), n = m ? matrix[0].size() : 0, r = 0, c = n - 1;
        while (r < m && c >= 0) {
            if (matrix[r][c] == target) {
```

```
            return true;
        }
        matrix[r][c] > target ? c-- : r++;
    }
    return false;
  }
};
```

3. **Output:**



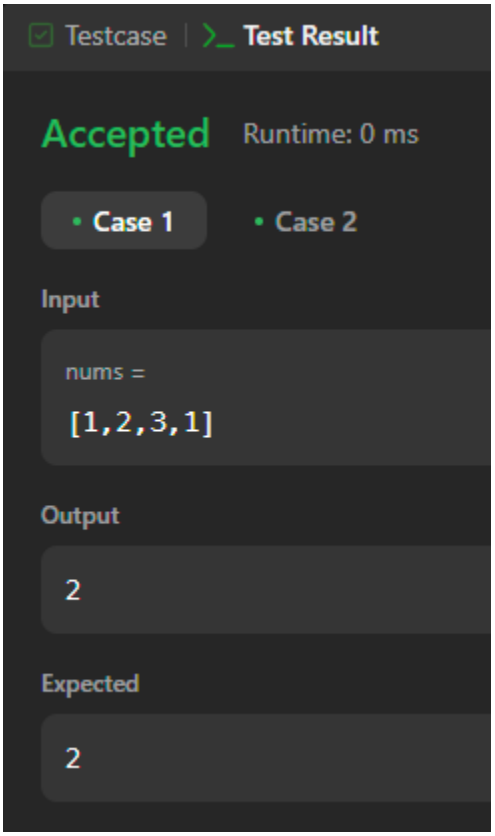
1. **Aim:**

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. 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.

2. **Code:**

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

3. **Output:**



1. **Aim:**

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?

2. **Code:**

```
class Solution {
public:
    int findKthLargest(std::vector<int>& nums, int k) {
```

```
        std::priority_queue<int, std::vector<int>, std::greater<int>>
min_heap(nums.begin(), nums.begin() + k);
        for (int i = k; i < nums.size(); i++) {
            if (nums[i] > min_heap.top()) {
                min_heap.pop();
                min_heap.push(nums[i]);
            }
        }
        return min_heap.top();
    }
};
```

3. **Output:**