**Experiment 5**

**Name: Sanju Arora Branch: BE in IT**

**Uid: 22BET10059 Semester: 6TH**

**Section/Group: 22BET\_IO1\_702/A Date Of Performance: 20/2/2025**

**Subject Name: Advanced Programming Lab-2 Subject Code: 22ITT-367**

* **Aim:**

1. To merge two sorted arrays in-place without using extra space.
2. To find the first bad version in a sequence using binary search to minimize API calls.
3. To sort an array of 0s, 1s, and 2s using the Dutch National Flag algorithm in O(n) time.
4. To find the k most frequent elements in an array using a heap or bucket sort.
5. To find the k-th largest element in an unsorted array using a min heap or QuickSelect.
6. To locate a peak element in an array using binary search for O(log n) efficiency.
7. To merge overlapping intervals in a given list after sorting them by start time.
8. To find a target element in a rotated sorted array using binary search.
9. To search for a target in a matrix sorted row-wise and column-wise using O(m + n) time.
10. To find the k-th smallest element in a sorted matrix using a min heap or binary search.
11. To compute the median of two sorted arrays in O(log (m + n)) time using binary search on partitions.

* **Code:**

1. **Merge Sorted Array**

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;

while (i >= 0 && j >= 0) {

if (nums1[i] > nums2[j]) {

nums1[k] = nums1[i];

i--;

} else {

nums1[k] = nums2[j];

j--;

}

k--; }

while (j >= 0) {

nums1[k] = nums2[j];

j--;

k--;

}

}

}

1. **First Bad Version**

public class Solution extends VersionControl {

public int firstBadVersion(int n) {

int left = 1, right = n;

while (left < right) {

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

if (isBadVersion(mid)) {

right = mid; // Move left to find the first bad version

} else {

left = mid + 1; // Move right

}

}

return left; // Left will point to the first bad version

}

}

1. **Sort Colors**

class Solution {

public void sortColors(int[] nums) {

int low = 0, mid = 0, high = nums.length - 1;

while (mid <= high) {

if (nums[mid] == 0) {

swap(nums, mid, low);

mid++;

low++;

} else if (nums[mid] == 1) {

mid++;

} else { // nums[mid] == 2

swap(nums, mid, high);

high--;

}

}

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

1. **Top K Frequent Elements**

import java.util.\*;

class Solution {

public int[] topKFrequent(int[] nums, int k) {

// Step 1: Count frequencies

Map<Integer, Integer> freqMap = new HashMap<>();

for (int num : nums) {

freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);

}

// Step 2: Min Heap (PriorityQueue) to keep top k elements

PriorityQueue<Integer> minHeap = new PriorityQueue<>((a, b) -> freqMap.get(a) - freqMap.get(b));

for (int num : freqMap.keySet()) {

minHeap.add(num);

if (minHeap.size() > k) {

minHeap.poll(); // Remove the least frequent element

}

}

// Step 3: Extract elements from the heap

int[] result = new int[k];

for (int i = k - 1; i >= 0; i--) {

result[i] = minHeap.poll();

}

return result;

}

}

1. **Kth Largest Element in an Array**

import java.util.\*;

class Solution {

public int findKthLargest(int[] nums, int k) {

// Min Heap to store the top k largest elements

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

for (int num : nums) {

minHeap.add(num);

if (minHeap.size() > k) {

minHeap.poll(); // Remove the smallest element

}

}

return minHeap.peek(); // The root of the heap is the k-th largest element

}

}

1. **Find Peak Element**

class Solution {

public int findPeakElement(int[] nums) {

int left = 0, right = nums.length - 1;

while (left < right) {

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

if (nums[mid] > nums[mid + 1]) {

right = mid; // Peak is in the left half

} else {

left = mid + 1; // Peak is in the right half

}

}

return left; // left and right converge to a peak index

}

}

1. **Merge Intervals**

import java.util.\*;

class Solution {

public int[][] merge(int[][] intervals) {

if (intervals.length == 0) return new int[0][];

// Step 1: Sort intervals by start time

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

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

// Step 2: Merge overlapping intervals

int[] currentInterval = intervals[0];

merged.add(currentInterval);

for (int[] interval : intervals) {

int currentEnd = currentInterval[1];

int nextStart = interval[0];

int nextEnd = interval[1];

if (currentEnd >= nextStart) { // Overlapping case

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

} else { // No overlap, add new interval

currentInterval = interval;

merged.add(currentInterval);

}

}

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

}

}

1. **Search in Rotated Sorted Array**

class Solution {

public int search(int[] nums, int target) {

int left = 0, right = nums.length - 1;

while (left <= right) {

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

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

// Check which half is sorted

if (nums[left] <= nums[mid]) { // Left half is sorted

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

right = mid - 1; // Search left

} else {

left = mid + 1; // Search right

}

} else { // Right half is sorted

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

left = mid + 1; // Search right

} else {

right = mid - 1; // Search left

}

}

}

return -1; // Target not found

}

}

1. **Search a 2D Matrix II**

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

int rows = matrix.length, cols = matrix[0].length;

int row = 0, col = cols - 1;

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

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

return true; // Target found

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

col--; // Move left

} else {

row++; // Move down

}

}

return false; // Target not found

}

}

1. **Kth Smallest Element in a Sorted Matrix**

import java.util.\*;

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int n = matrix.length;

PriorityQueue<int[]> minHeap = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));

// Add first element of each row to the heap

for (int i = 0; i < Math.min(n, k); i++) {

minHeap.add(new int[]{matrix[i][0], i, 0}); // {value, row, col}

}

int count = 0, result = 0;

while (!minHeap.isEmpty()) {

int[] element = minHeap.poll();

result = element[0];

count++;

if (count == k) return result; // k-th smallest found

int row = element[1], col = element[2];

if (col + 1 < n) {

minHeap.add(new int[]{matrix[row][col + 1], row, col + 1});

}

}

return result;

}

}

1. **Median of Two Sorted Arrays**

class Solution {

public double findMedianSortedArrays(int[] nums1, int[] nums2) {

// Ensure nums1 is the smaller array

if (nums1.length > nums2.length) {

return findMedianSortedArrays(nums2, nums1);

}

int m = nums1.length, n = nums2.length;

int leftPartitionSize = (m + n + 1) / 2;

int low = 0, high = m;

while (low <= high) {

int partition1 = low + (high - low) / 2;

int partition2 = leftPartitionSize - partition1;

int left1 = (partition1 == 0) ? Integer.MIN\_VALUE : nums1[partition1 - 1];

int right1 = (partition1 == m) ? Integer.MAX\_VALUE : nums1[partition1];

int left2 = (partition2 == 0) ? Integer.MIN\_VALUE : nums2[partition2 - 1];

int right2 = (partition2 == n) ? Integer.MAX\_VALUE : nums2[partition2];

if (left1 <= right2 && left2 <= right1) {

// Found correct partition

if ((m + n) % 2 == 0) {

return (Math.max(left1, left2) + Math.min(right1, right2)) / 2.0;

} else {

return Math.max(left1, left2);

}

} else if (left1 > right2) {

high = partition1 - 1; // Move left

} else {

low = partition1 + 1; // Move right

}

}

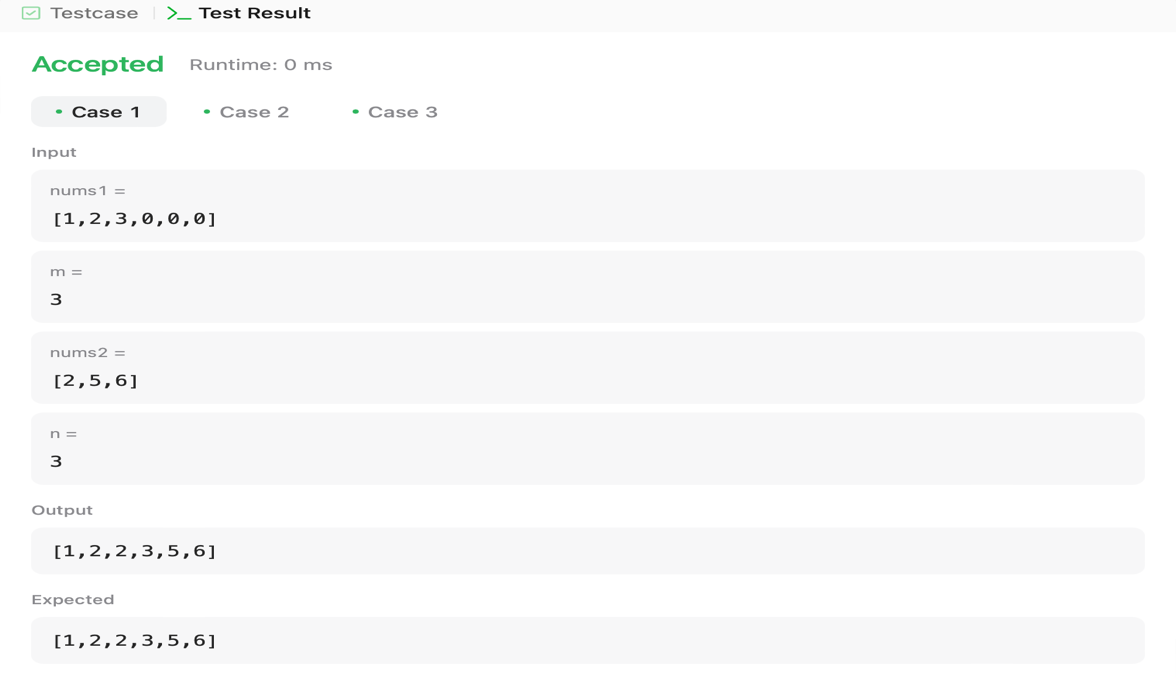
throw new IllegalArgumentException("Invalid input");

}

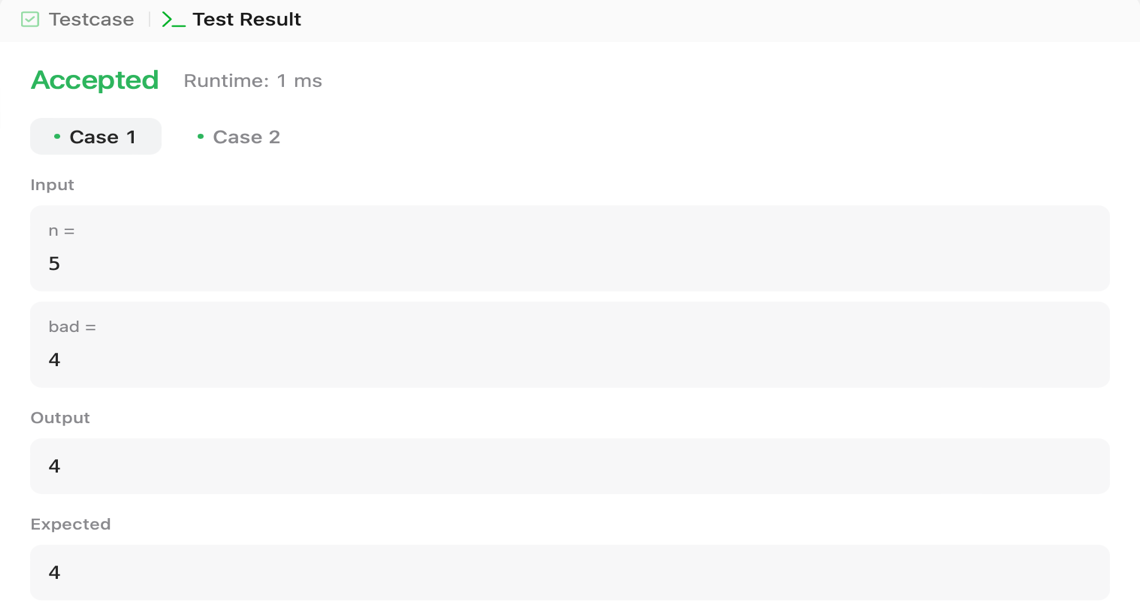
}

* **Output:**

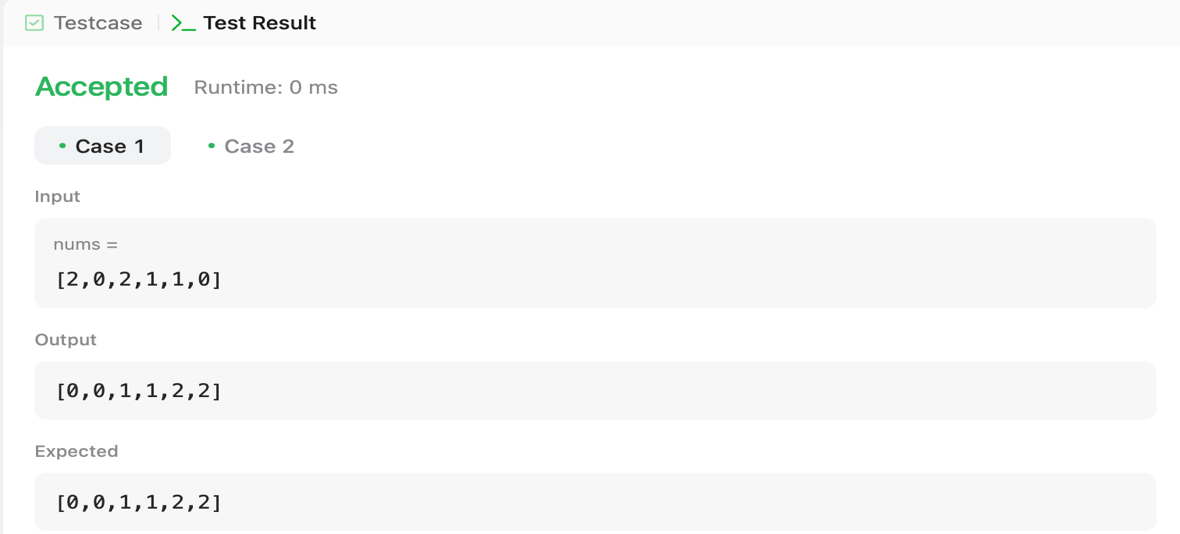
1. **Merge Sorted Array:**



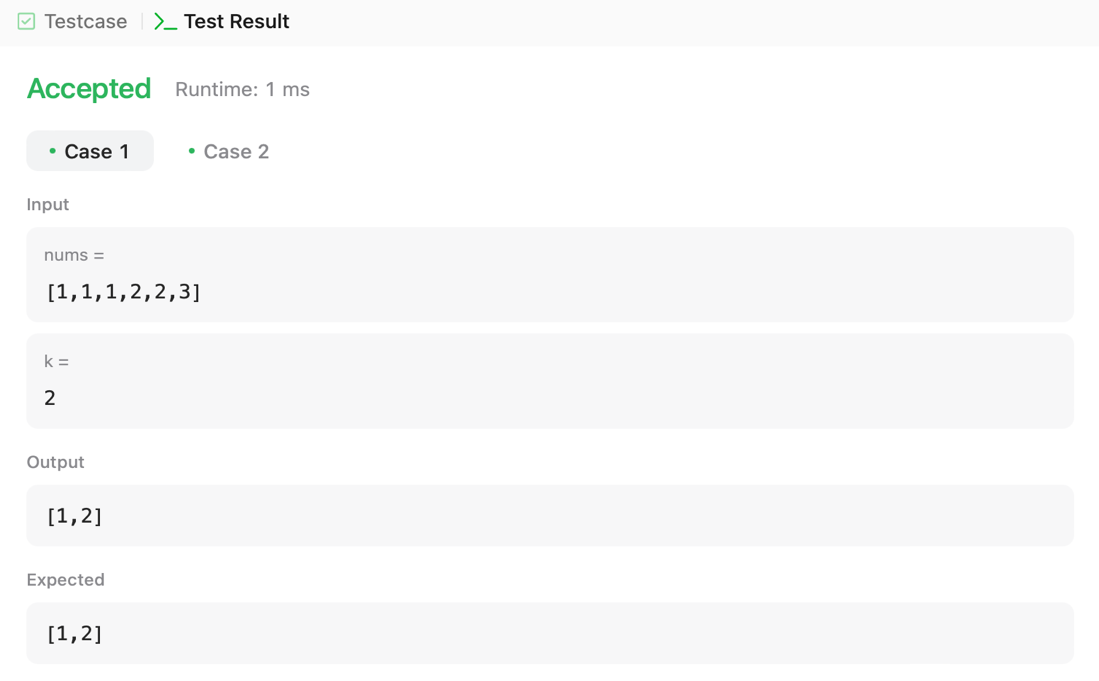
1. **First Bad Version:**



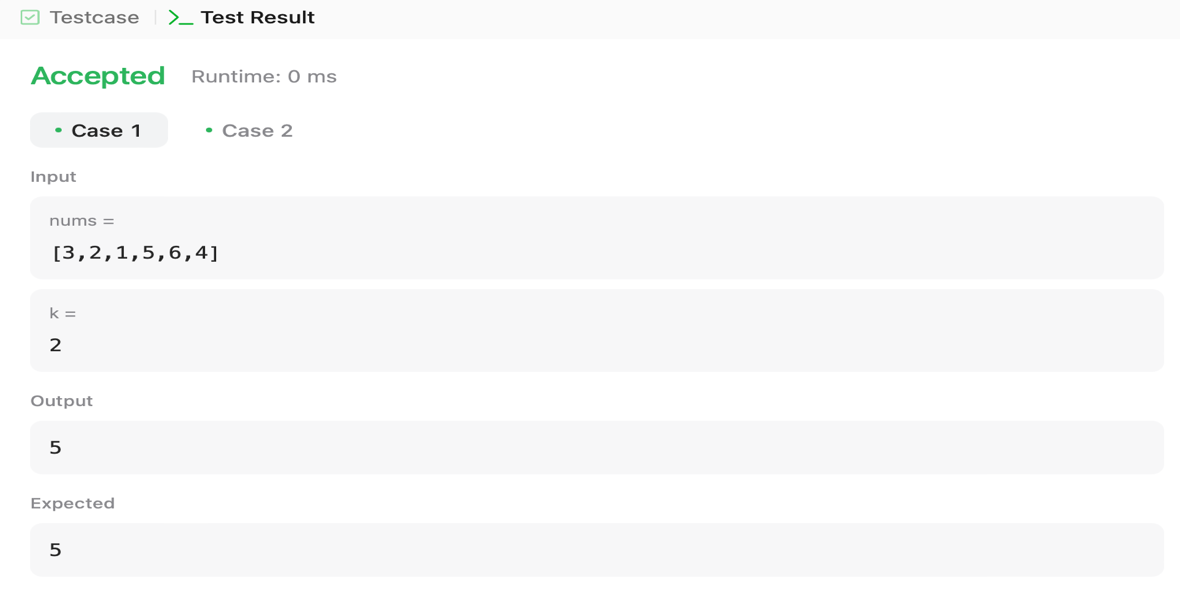
1. **Sort Colors:**



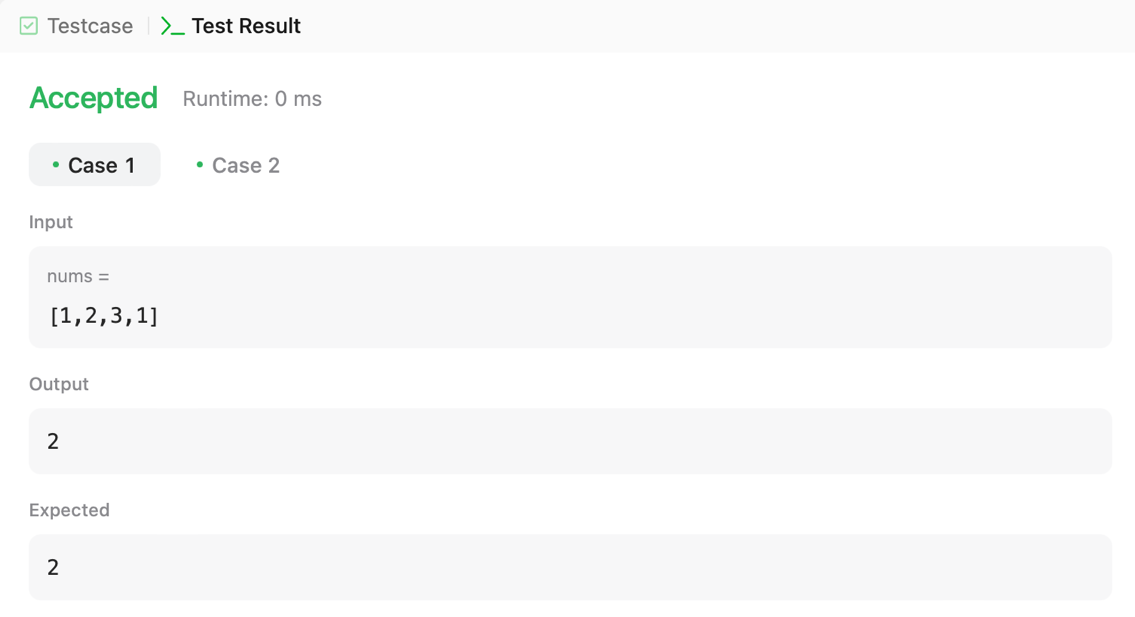
1. **Top K Frequent Elements**



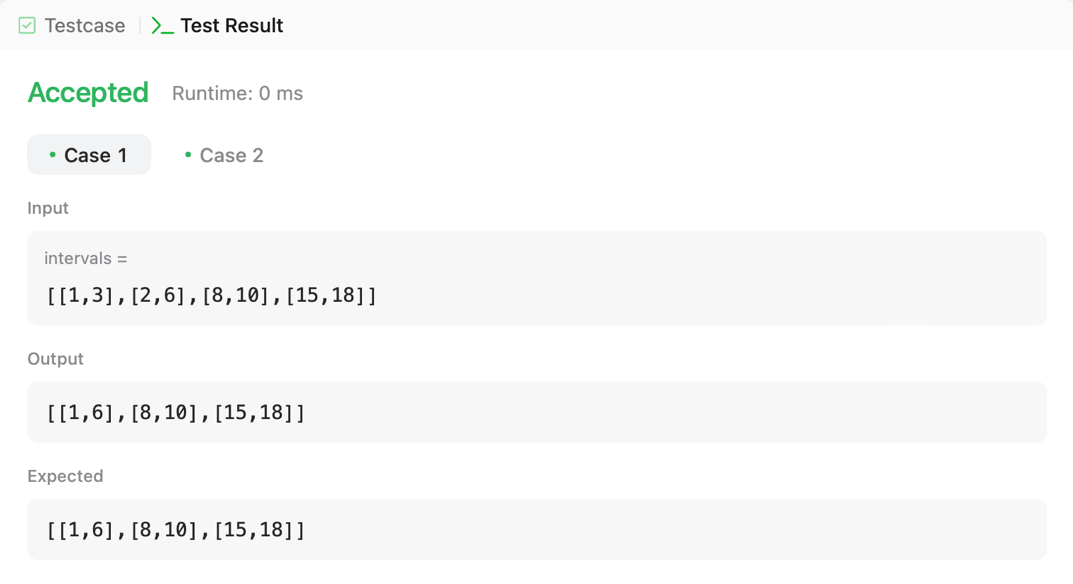
1. **Kth Largest Element in an Array**:



1. **Find Peak Element:**



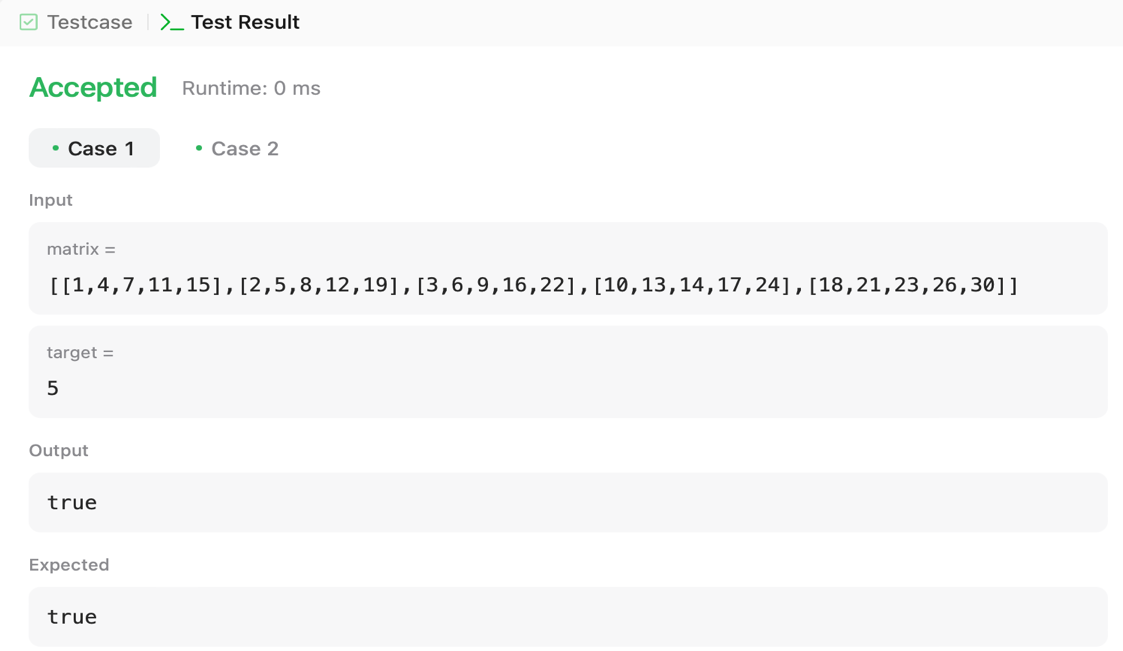
1. **Merge Intervals:**



1. **Search in Rotated Sorted Array:**



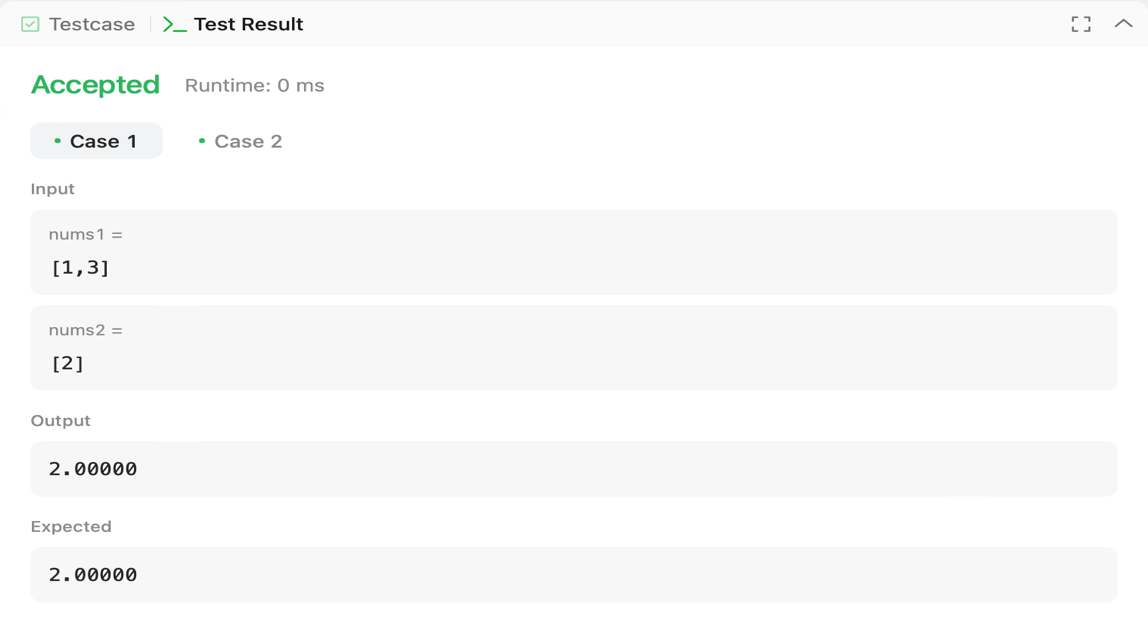
1. **Search a 2D Matrix II:**



1. **Kth Smallest Element in a Sorted Matrix:**



1. **Median of Two Sorted Arrays**

****