Experiment 1

Student Name: Rhythm Tyagi
UID: 22BCS17203
Branch: CSE
Section: NTPP_602-A
Semester: 6th
DOP: 20/01/25
Subject: AP-LAB-2
Subject Code:22CSP-351

Aim:

Problem 1.2.1: Two Sum

Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to target. Each input has exactly one solution, and you cannot use the same element twice.

Problem 1.2.2: Jump Game II

You are given a 0-indexed array nums of length n. You are initially positioned at nums[0]. Each element nums[i] represents the maximum length of a forward jump from index i. Return the minimum number of jumps to reach nums[n - 1].

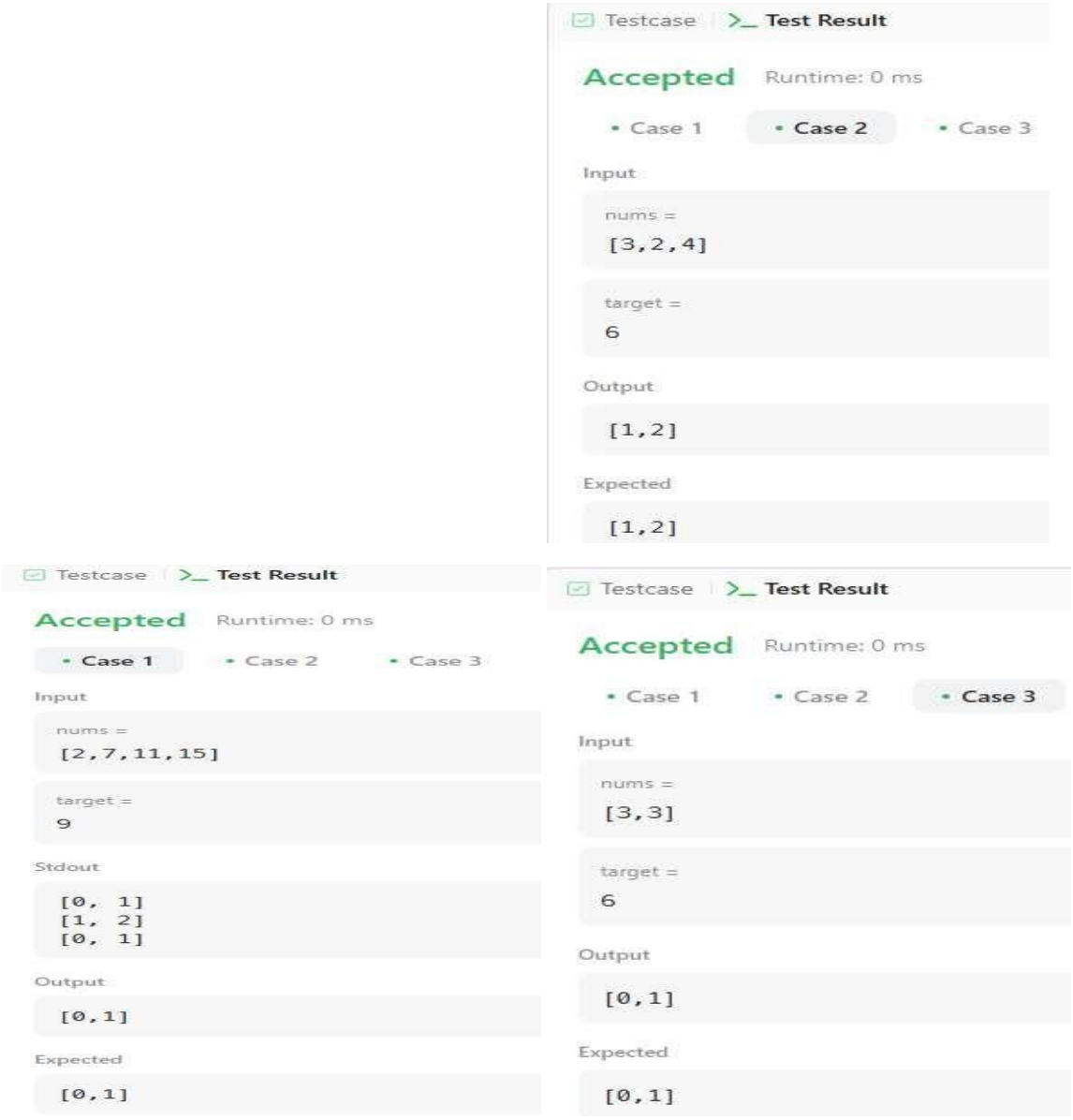
Problem 1.2.3: Remove Duplicates from Sorted Array

Given an integer array nums sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* nums

Code: 1.2.1

```
def two_sum(nums, target):
    num_map = {}  # Dictionary to store value-to-index mapping
    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_map:
            return [num_map[complement], i]
        num_map[num] = i
    return []
print(two_sum([2,7,11,15], 9))  # Output: [0,1]
print(two_sum([3,2,4], 6))
```

Output:


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[3,2,4]
target =
6
Output
[1,2]
Expected
[1,2]
Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[2,7,11,15]
target =
9
Stdout
[0, 1]
[1, 2]
[0, 1]
Output
[0,1]
Expected
[0,1]
Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[3,3]
target =
6
Output
[0,1]
Expected
[0,1]

CODE: 1.2.2

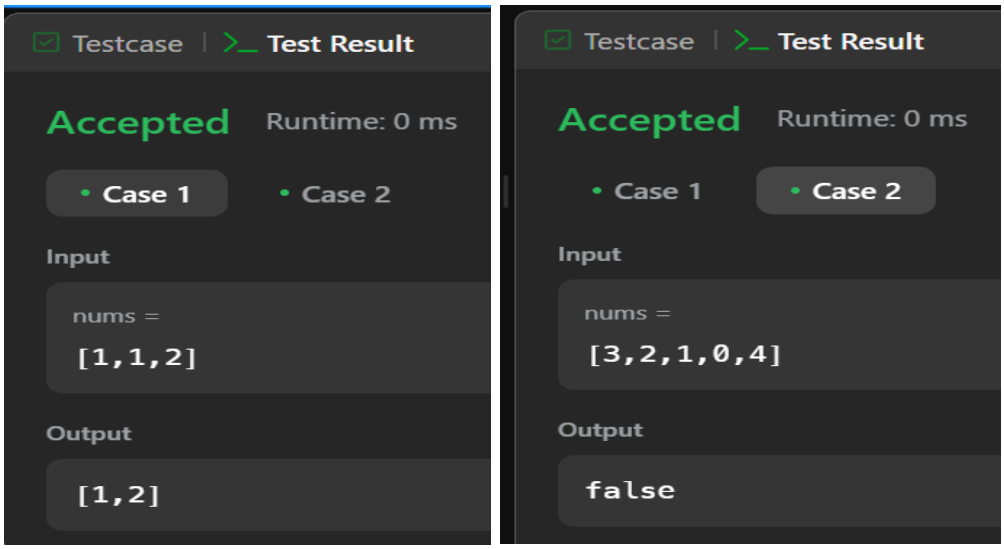
```
public class Solution {
    public boolean canJump(int[] nums) {
        int maxReach = 0;  // Track the farthest index we can reach
                for (int i = 0; i < nums.length; i++) {
            if (i > maxReach) {
                return false;  // If current index is unreachable, return false
            }
            maxReach = Math.max(maxReach, i + nums[i]);  // Update the farthest reachable index
                    // Early exit if we can already reach the last index
            if (maxReach >= nums.length - 1) {
                return true;
            }       }
        return maxReach >= nums.length - 1;
    }
    public static void main(String[] args) {
        Solution sol = new Solution();
                int[] nums1 = {2, 3, 1, 1, 4};
        System.out.println(sol.canJump(nums1));  // Output: true
        int[] nums2 = {3, 2, 1, 0, 4};
        System.out.println(sol.canJump(nums2));  // Output: false
    }
}
```

OUTPUT:

CODE: 1.2.3

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;

        int slow = 0; // Pointer for unique elements
        for (int fast = 1; fast < nums.length; fast++) {
            if (nums[fast] != nums[slow]) {
                slow++; // Move the slow pointer
                nums[slow] = nums[fast]; // Copy unique element
            }
        }
        return slow + 1; // Unique count (1-based index)
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        int[] nums1 = {1, 1, 2};
        int k1 = solution.removeDuplicates(nums1);
        System.out.println("Output: " + k1);
        System.out.print("Modified array: ");
        for (int i = 0; i < k1; i++) {
            System.out.print(nums1[i] + " ");
        }
        System.out.println();

        int[] nums2 = {0, 0, 1, 1, 1, 2, 2, 3, 3, 4};
        int k2 = solution.removeDuplicates(nums2);
        System.out.println("Output: " + k2);
        System.out.print("Modified array: ");
        for (int i = 0; i < k2; i++) {
            System.out.print(nums2[i] + " ");
        }
        System.out.println();
    }
}
```

OUTPUT:



Learning Outcomes:

- Understand how to efficiently solve the Two Sum Problem using a hash map to optimize time complexity.
- Learn to implement a greedy algorithm for Jump Game II to minimize jumps while maintaining linear time complexity.
- Acquire skills in processing and simplifying file paths using stacks in the Simplify Path problem.
- Develop problem-solving techniques for handling string manipulation and edge cases in Unix-style file systems.
- Enhance algorithmic thinking by applying data structures like hash maps and stacks to solve real-world problems effectively.