Experiment – 2

Student Name:Gourav Kumar **UID: 22BCS10854**
Branch: BE-CSE **Section/Group: NTPP -603/A**
Semester: 6th **Date of Performance: 11/01/24**
Subject Name: AP Lab - 2 **Subject Code: 22CSP-351**

Problem-1

1. Aim : Two Sum

2. Objective :

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

3. Implementation & Output :

Code

Java Auto

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int n = nums.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i, j};
                }
            }
        }
        return new int[]{}; // No solution found
    }
}
```

Saved Ln 13, Col 2

Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

nums =
[2,7,11,15]

target =
9

Output

[0,1]

Expected

[0,1]

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]

4. Learning Outcomes :

- **Nested Loops:** Demonstrates how to iterate through all pairs of elements in an array using nested loops.
- **Brute Force Approach:** Solves the problem using a brute force method with $O(n^2)$ time complexity, highlighting its inefficiency for large inputs.
- **Array Indexing:** Teaches how to access and compare elements in an array by their indices.
- **Edge Case Handling:** Handles cases where no solution is found by returning an empty array

Problem-2

1. Aim : Jump Game

2. Objective :

You are given a **0-indexed** array of integers `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`. In other words, if you are at `nums[i]`, you can jump to any `nums[i + j]` where:

- `0 <= j <= nums[i]` and
- `i + j < n`

Return *the minimum number of jumps to reach* `nums[n - 1]`. The test cases are generated such that you can reach `nums[n - 1]`.

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from
index 0 to 1, then 3 steps to the last index.
```

3. Implementation & Output :

Code

Java Auto

```java
class Solution {
    public int jump(int[] nums) {
        int near = 0, far = 0, jumps = 0;

        while (far < nums.length - 1) {
            int farthest = 0;
            for (int i = near; i <= far; i++) {
                farthest = Math.max(farthest, i + nums[i]);
            }
            near = far + 1;
            far = farthest;
            jumps++;
        }

        return jumps;
    }
}
```

Saved Ln 15, Col 30

Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

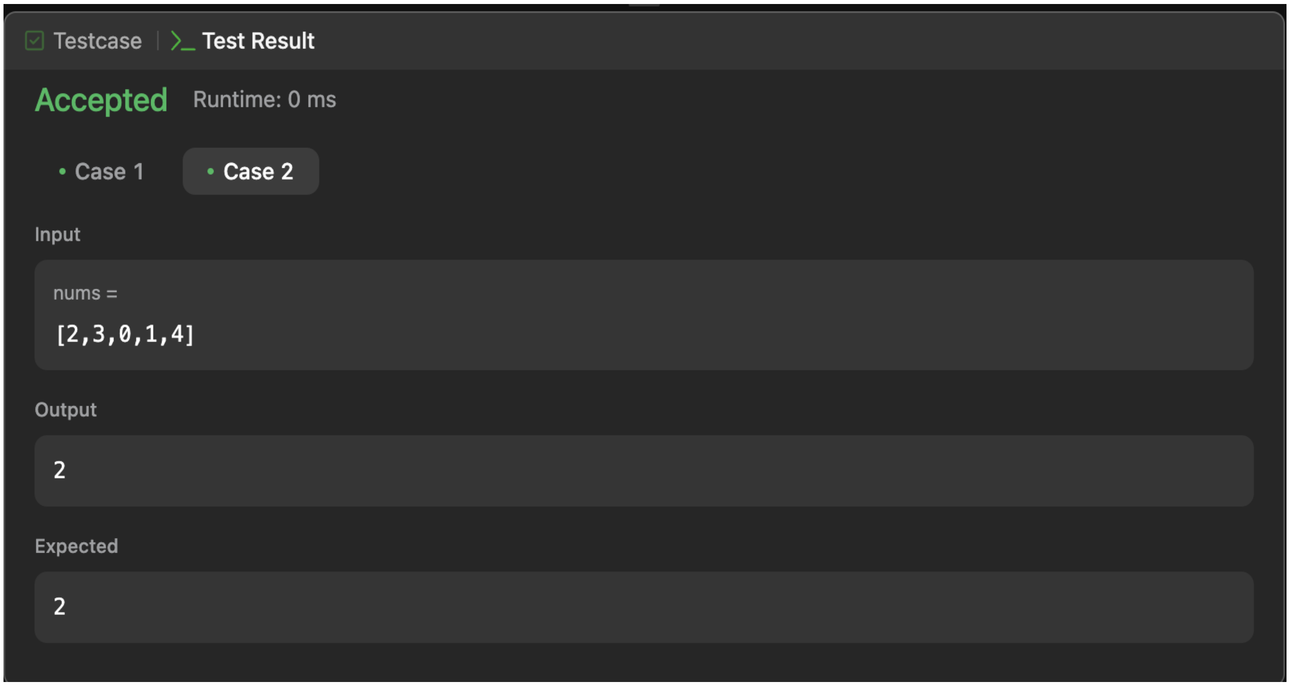
nums =
[2,3,1,1,4]

Output

2

Expected

2



4. Learning Outcomes :

- **Greedy Strategy:** Learn how to apply a greedy approach to solve optimization problems by always choosing the farthest reachable index to minimize jumps.
- **Two Pointer Technique:** Understand how to use two pointers (near and far) to track the current jump range and expand it iteratively.
- **Efficient Jump Calculation:** Learn to calculate the minimum number of jumps required to reach the end of an array while traversing the input only once.
- **Time Complexity Awareness:** Recognize that the algorithm operates in O(n) time complexity, optimizing performance over brute force methods.