Experiment 7

Student Name: Nisha Kumari **UID: 22BET10118**
Branch: IT **Section/Group: 22BET_IOT-701/A**
Semester: 6th **Date of Performance:19.03.25**
Subject Name: AP Lab - 2 **Subject Code: 22ITP-351**

1. **Aim:** To design effective Dynamic Programming (DP) approaches for solving optimization problems, such as counting paths, maximizing sums and products, assessing reachability, and identifying optimal subsequences, while evaluating and optimizing time and space complexity.
   i.) Climbing Stairs
   ii.) Maximum Subarray
   iii.) House Robber
   iv.) Jump Game
   v.) Unique Paths
   vi.) Coin Change
   vii.) Longest Increasing Subsequence
   viii.) Maximum Product Subarray
   ix.) Decode Ways

2. **Objective:**

- Utilize Dynamic Programming (DP) techniques to efficiently solve optimization and sequence-based problems.
- Comprehend state transitions and recurrence relations in challenges such as Climbing Stairs and Coin Change.
- Enhance solutions for subarray and subsequence problems, including Maximum Subarray, Maximum Product Subarray, and Longest Increasing Subsequence.
- Evaluate feasibility in problems like Jump Game and Decode Ways using both greedy and DP strategies.
- Compute unique paths and devise optimal strategies in cases like Unique Paths and House Robber.
- Assess and refine time and space complexity, contrasting $O(n^2)$ DP approaches with $O(n \log n)$ methods.**Code:**

<u>Problem 1: Climbing Stairs</u>

```
class Solution {
public:
    int climbStairs(int n) {
        if (n == 1) return 1;
```

```
        if (n == 2) return 2;
        int prev1 = 2, prev2 = 1, current;
        for (int i = 3; i <= n; i++) {
            current = prev1 + prev2;
            prev2 = prev1;
            prev1 = current;
        }
        return current;
    }
};
```

Problem 2: Maximum Subarray

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxSum = INT_MIN, currentSum = 0;
        for (int num : nums) {
            currentSum += num;
            maxSum = max(maxSum, currentSum);
            if (currentSum < 0) currentSum = 0;
        }
        return maxSum;
    }
};
```

Problem 3: House Robber

```
class Solution {
public:
    int rob(vector<int>& nums) {
         if (nums.empty()) return 0;
        if (nums.size() == 1) return nums[0];
        int prev1 = 0, prev2 = 0;
        for (int num : nums) {
            int temp = max(prev1, prev2 + num);
            prev2 = prev1;
            prev1 = temp;
        }
        return prev1;
    }
};
```

Problem 4: Jump Game

```
class Solution {
public:
    bool canJump(vector<int>& nums)
         int maxReach = 0;
        int n = nums.size();
        for (int i = 0; i < n; i++) {
```

```
            if (i > maxReach) return false;  // Can't move forward
            maxReach = max(maxReach, i + nums[i]);  // Update max reachable index
            if (maxReach >= n - 1) return true;  // Early exit if last index is reachable
        }
        return true;
    }
};
```

Problem 5: Unique Paths

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> dp(n, 1); // Initialize first row with 1
        for (int i = 1; i < m; i++) {  // Start from row 1
            for (int j = 1; j < n; j++) {
                dp[j] += dp[j - 1];  // Update current cell
            }
        }
        return dp[n - 1];
    }
};
```

6: Coin Change

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
         vector<int> dp(amount + 1, amount + 1);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (i >= coin) {
                    dp[i] = min(dp[i], 1 + dp[i - coin]);
                }
        }
        return (dp[amount] == amount + 1) ? -1 : dp[amount];
    }
};
```

Problem 7: Longest Increasing Subsequence

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
         int n = nums.size();
        vector<int> dp(n, 1);  // Initialize all values to 1
        int maxLength = 1;
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
```

```
                if (nums[j] < nums[i]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
            maxLength = max(maxLength, dp[i]);
        }
        return maxLength;
    }
};
```

Problem 8: Maximum Product Subarray

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
         int n = nums.size();
        int maxProd = nums[0], minProd = nums[0], result = nums[0];
        for (int i = 1; i < n; i++) {
            if (nums[i] < 0) swap(maxProd, minProd);  // Swap when encountering negative
            maxProd = max(nums[i], nums[i] * maxProd);
            minProd = min(nums[i], nums[i] * minProd);
            result = max(result, maxProd);  // Update global max
        }
        return result;
    }
};
```

Problem 9: Decode Ways

```
class Solution {
public:
    int numDecodings(string s) {
        int n = s.size();
        if (s[0] == '0') return 0; // Cannot start with zero

        vector<int> dp(n + 1, 0);
        dp[0] = 1; // Empty string has one way
        dp[1] = 1; // First character (if not '0') has one way

        for (int i = 2; i <= n; i++) {
            if (s[i - 1] != '0') {
                dp[i] += dp[i - 1]; // Single digit decoding
            }
            int twoDigit = stoi(s.substr(i - 2, 2)); // Get last two digits
            if (twoDigit >= 10 && twoDigit <= 26) {
                dp[i] += dp[i - 2]; // Two-digit decoding
            }
        }
        return dp[n];
    }
};
```

3. Output:

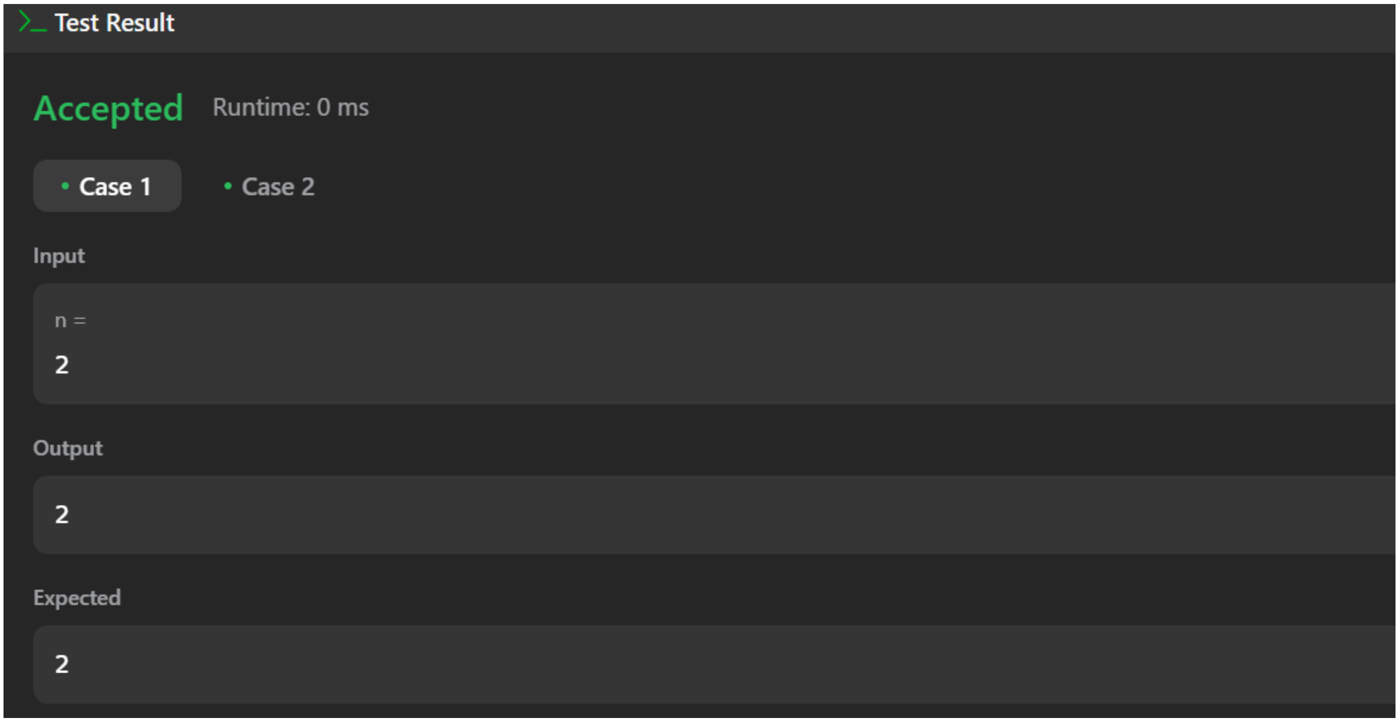

Fig 1. Climbing Stairs

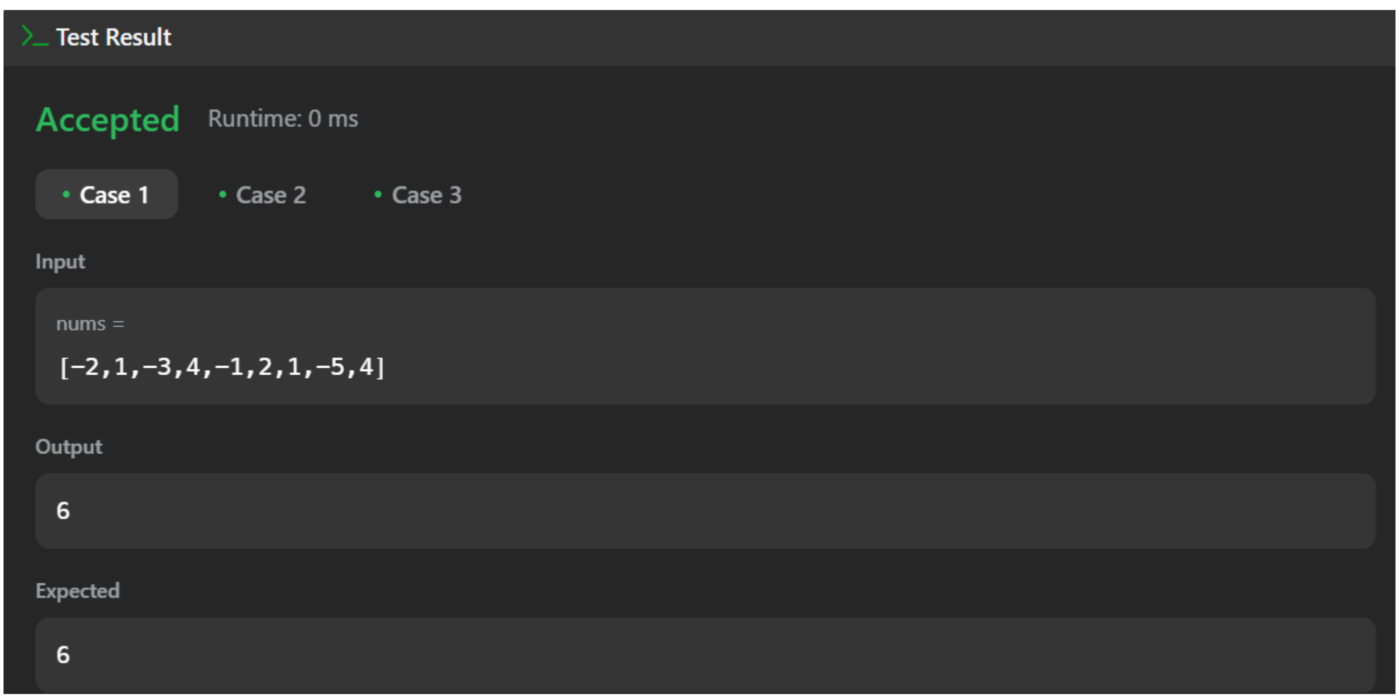

Fig 2. Maximum Subarray

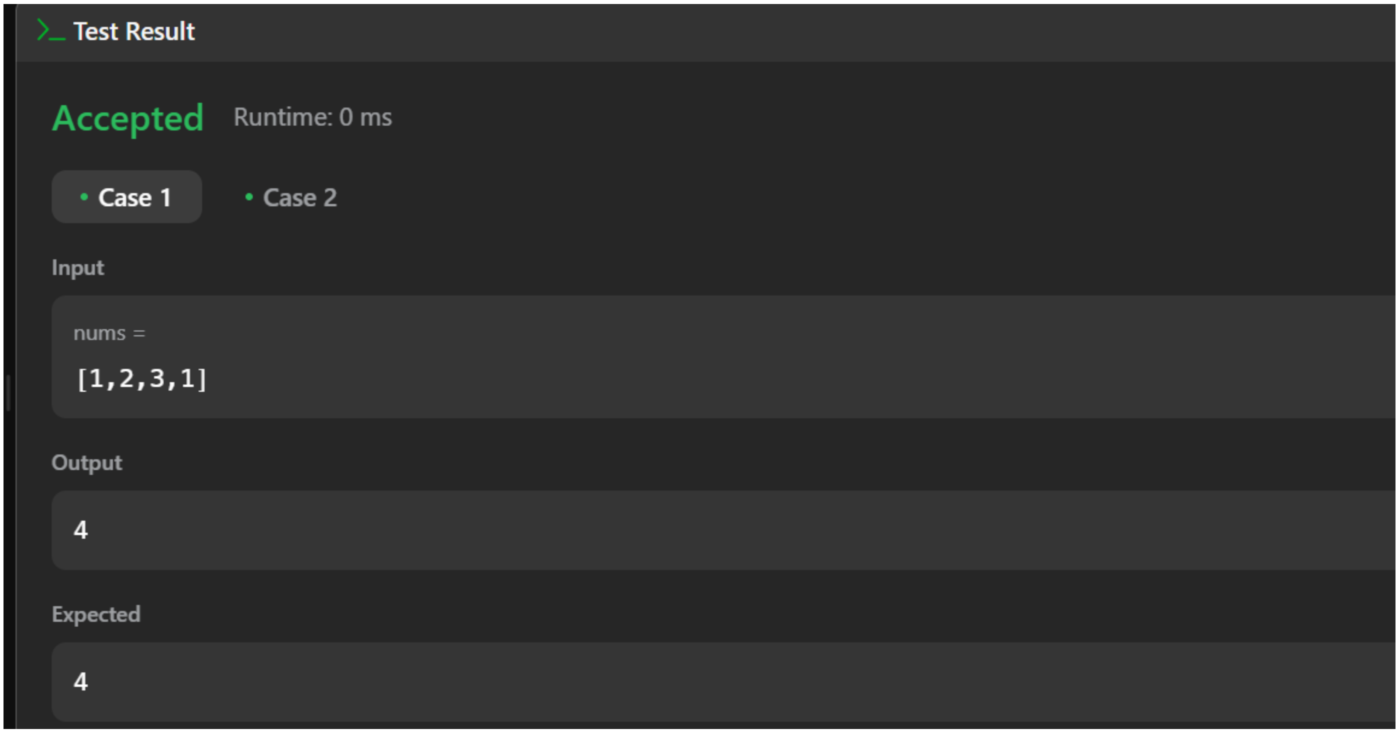

Fig 3. House Robber

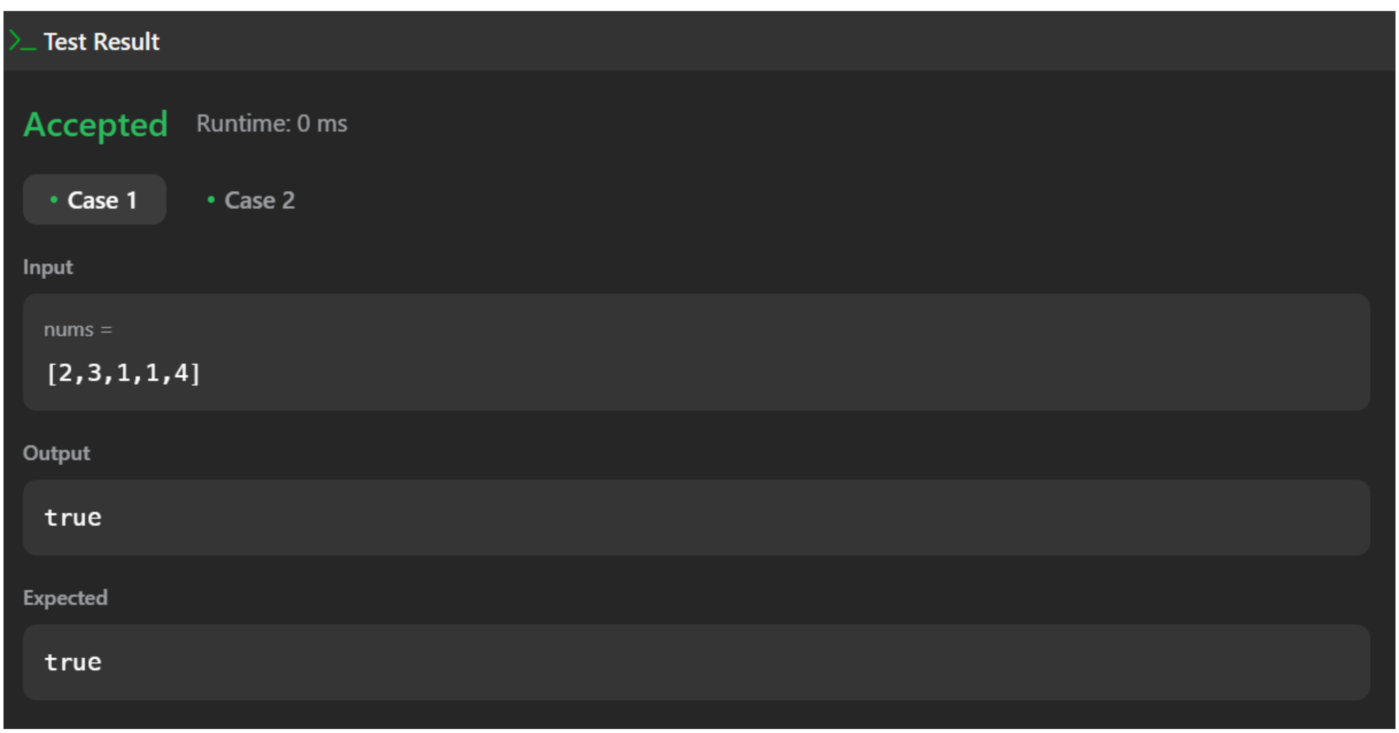

Fig 4. Jump Game

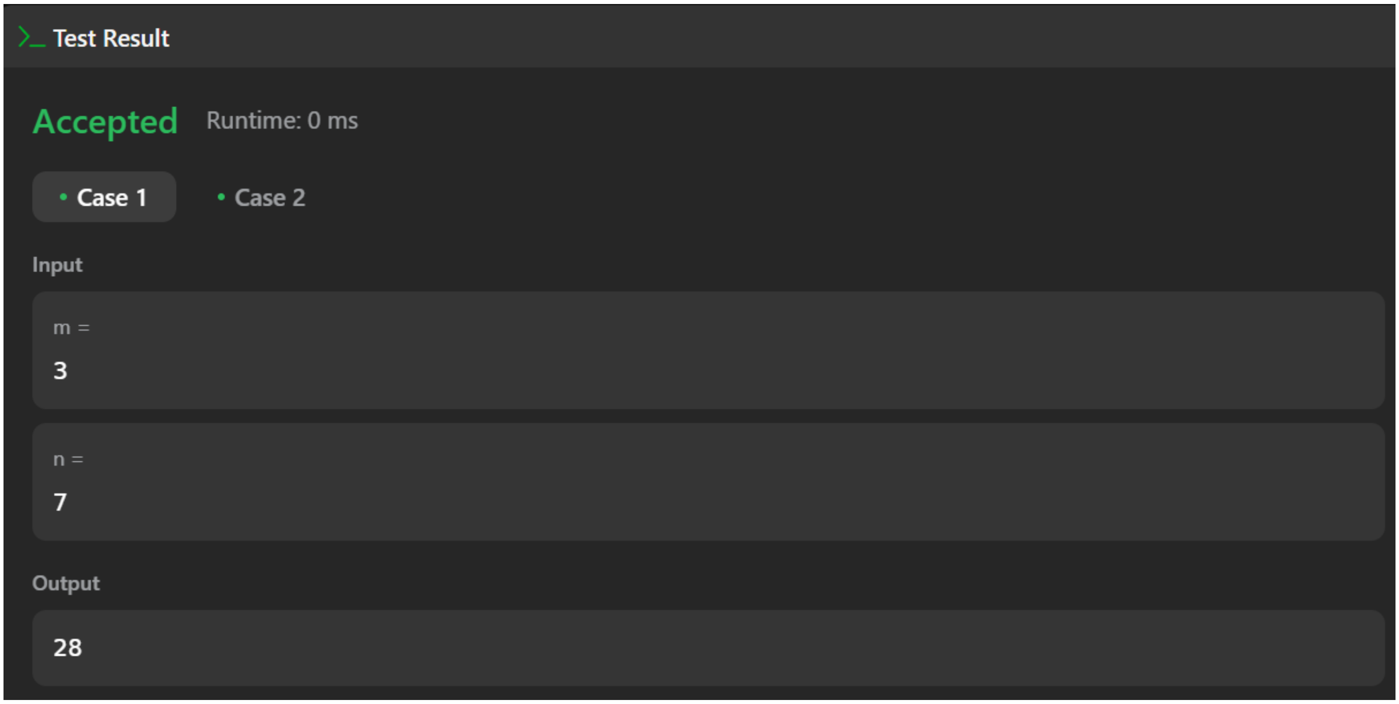

Fig 5. Unique Paths

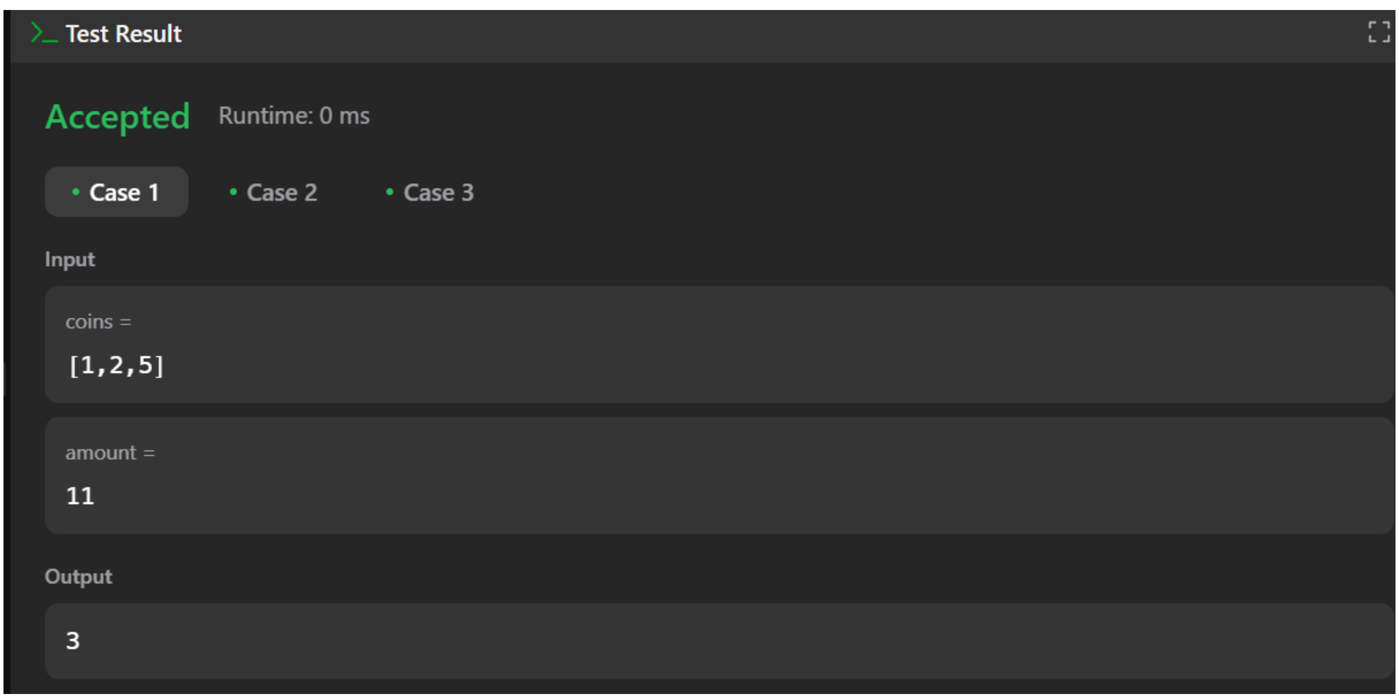

Fig 6. Coin Change

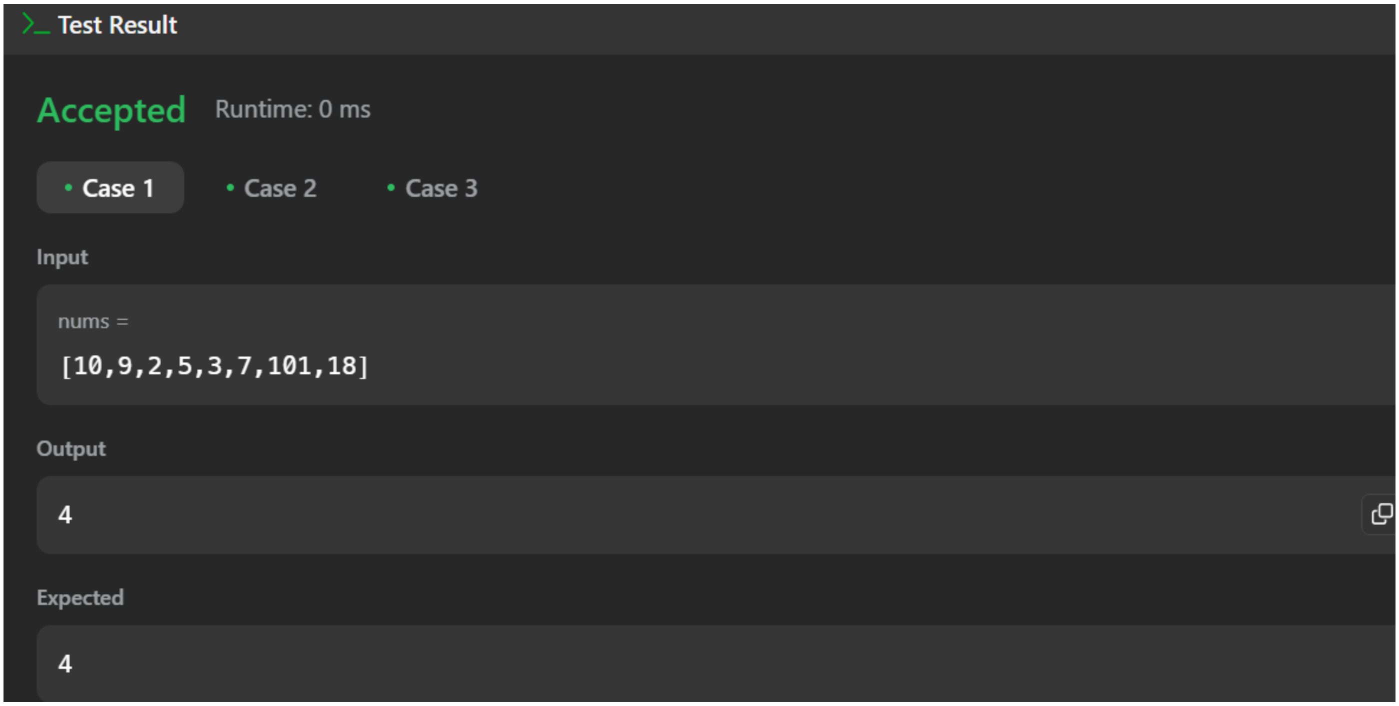

Fig 7. Longest Increasing Subsequence

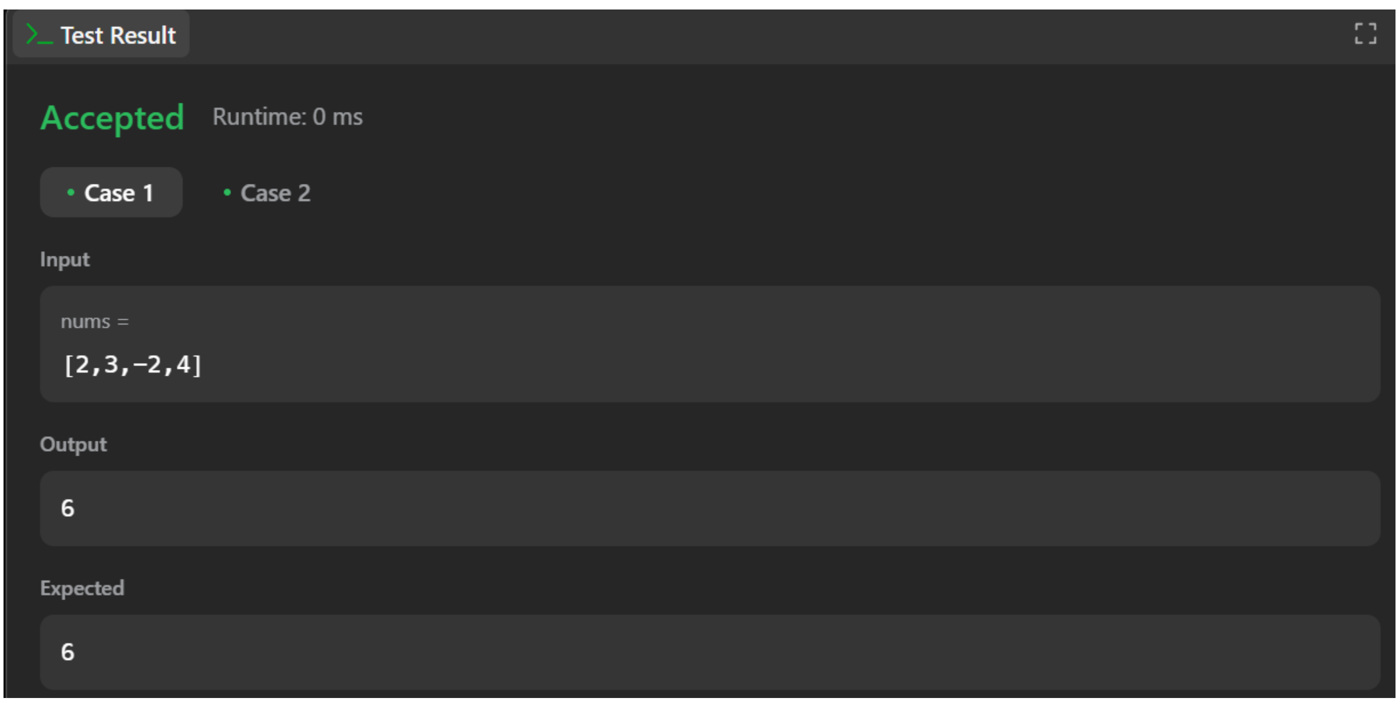

Fig 8. Maximum Product Subarray

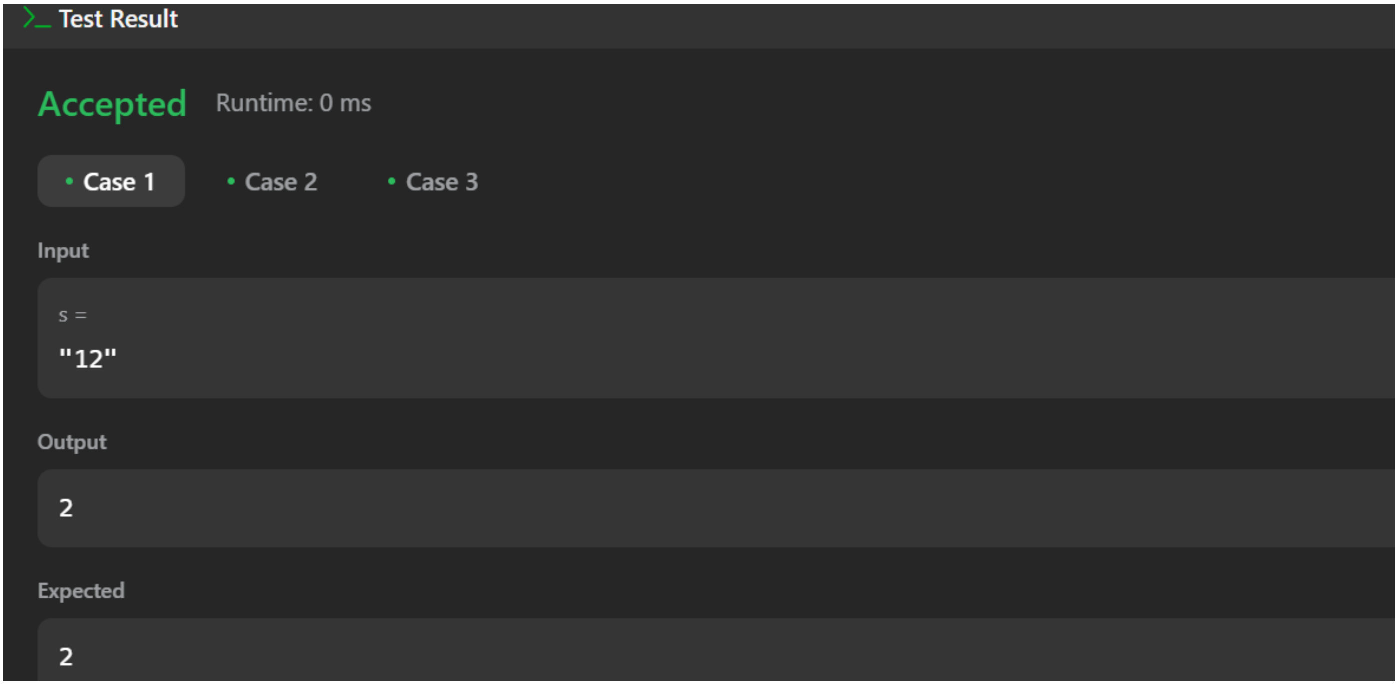

Fig 9. Decode Ways

4. Learning Outcomes:

- Develop a strong understanding of Dynamic Programming (DP) to efficiently tackle optimization problems.
- Master the management of subproblems and overlapping substructures using a bottom-up DP approach.
- Implement state transitions and recurrence relations in classic problems such as Coin Change, Longest Increasing Subsequence (LIS), and Word Break.
- Compare and optimize DP solutions by analyzing $O(n^2)$ approaches and exploring $O(n \log n)$ alternatives.
- Utilize HashSets and Binary Search to enhance efficiency in word segmentation and sequence-based problems.