DEPARTMENTOF

COMPUTERSCIENCE&ENGINEERING

Discover. Learn. Empower.

Experiment 7

Student Name: Akshant Kumar **UID:22BCS13418**

Branch: CSE **Section/Group: NTPP-IOT627/A**

Semester: 6th **Date of Performance:28-3-25**

Subject Name :AP Lab-2 **Subject Code:** 22CSP-351

Problem-1

Aim: Climbing Stairs

Objective: The objective of this code is to calculate the number of distinct ways to climb a staircase with n steps, where you can take either 1 or 2 steps at a time. It uses a dynamic programming approach to efficiently compute the total ways, leveraging a Fibonacci-like sequence.

Implementation/Code:

```
class Solution {
public:
    int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }
        int dp[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
};
```

Output:

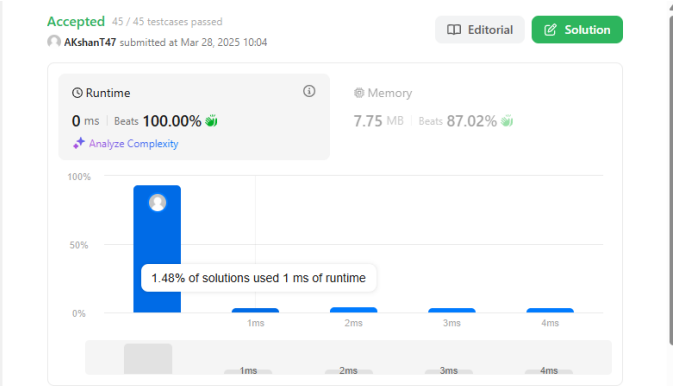

Figure1

Learning Outcome:

- Understanding dynamic programming and its application in solving problems with overlapping sub-problems.

- Recognizing how the Fibonacci sequence can be applied to solve real-world scenarios.
- Exploring techniques to optimize space complexity for efficient solutions.
- Improving problem-solving skills by modeling real-world challenges mathematically and implementing algorithms effectively.

Problem-2

Aim: Jump Game

Objective: The objective of the code is to determine whether it is possible to reach the last index of the array, given that each element represents the maximum number of steps you can jump from that position. It uses a greedy approach to efficiently track the farthest reachable index during traversal and checks if the last index can be reached.

Implementation/Code:

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int maxReach = 0;

        for (int i = 0; i < nums.size(); i++) {
            if (i > maxReach) {
                return false;
            }
            maxReach = max(maxReach, i + nums[i]);
        }
        return true;
    }
};
```

Output:

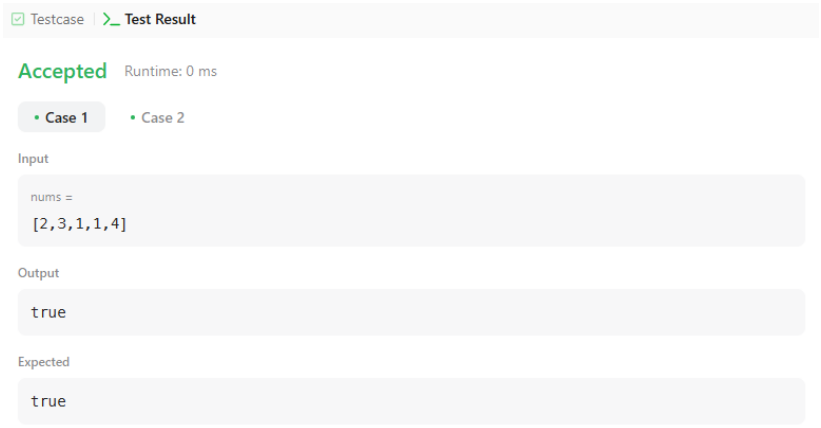

Figure2

Learning Outcomes:

- Gaining proficiency in implementing the **greedy algorithm approach** to solve problems by making the

best possible decision at each step.

- Developing the ability to analyze and track variables efficiently, such as maintaining the **maximum reachable index** during array traversal.
- Understanding how to handle scenarios with constraints and edge cases (e.g., positions with a jump length of 0).
- Improving problem-solving skills by translating the problem's requirements into an optimized algorithm with O(n)O(n) time complexity and O(1)O(1) space complexity.

Problem–3

Aim: Maximum Product Sub-array

Objective: The objective of the code is to find the sub-array within the given integer array nums that has the largest product and return this maximum product. It handles positive, negative, and zero values efficiently using dynamic programming to maintain the maximum and minimum products at each step, ensuring correctness even when negative numbers flip the product values.

Implementation/Code:

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int n = nums.size();
        int maxProd = nums[0];
        int currMax = nums[0];
        int currMin = nums[0];

        for (int i = 1; i < n; i++) {
            if (nums[i] < 0) {
                swap(currMax, currMin);
            }
            currMax = max(nums[i], currMax * nums[i]);
            currMin = min(nums[i], currMin * nums[i]);
            maxProd = max(maxProd, currMax);
        }
        return maxProd;
    }
};
```

Output:

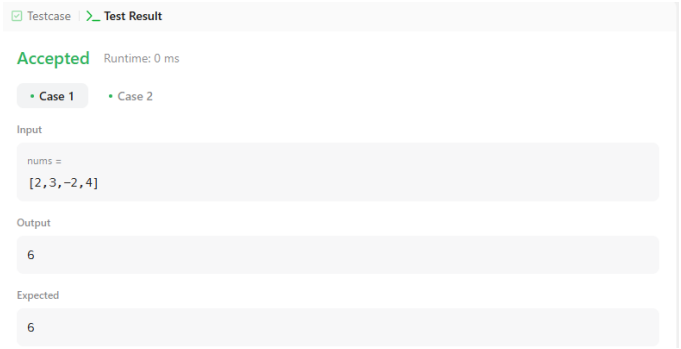

Figure3

Learning Outcomes:

- Understanding how to use **dynamic programming** to track multiple states (maximum and minimum products) for solving complex problems.
- Developing the ability to handle **edge cases**, such as arrays with negative numbers and zeros, to ensure robustness in algorithmic solutions.
- Gaining insights into optimizing both **time complexity** to O(n)O(n) and **space complexity** to O(1)O(1) for efficient implementation.
- Improving problem-solving skills by learning to translate real-world mathematical problems into efficient computational algorithms.