Experiment 6

Student Name: Arsh Saini | **UID: 22BCS12919**
Branch: CSE | **Section/Group:** 901/B
Semester: 6th | **Date of Performance: 12/03/25**
Subject Name: Advanced Programming Lab-2 | **Subject Code:** 22CSP-351

Problem -1

1. **Aim:** Climbing Stairs
2. **Objective:**

- **Compute the Number of Ways to Climb Stairs**: - Determine how many distinct ways one can reach the top by taking either 1 or 2 steps at a time.
- **Optimize Space Complexity**:- Use only constant **O(1)** space instead of an **O(n)** array.
- **Improve Efficiency**: - Implement an **O(n) time complexity** solution using an iterative approach rather than recursion.
- **Leverage Fibonacci-like Transition**: - Utilize the relation **f(n) = f(n-1) + f(n-2)** to compute results efficiently..
- **Handle Base Cases**:- Return **1 for n = 1** and **2 for n = 2** directly to avoid unnecessary computations.

3. **Implementation/Code:**

```
class Solution {
public:
    int climbStairs(int n) {
        if (n == 1) return 1;
        if (n == 2) return 2;
        int a = 1, b = 2, c;
       for (int i = 3; i <= n; i++) {
            c = a + b;  // Fibonacci-like transition
            a = b;
            b = c;
```

```
        }

        return b;

    }

};
```

4. Output:

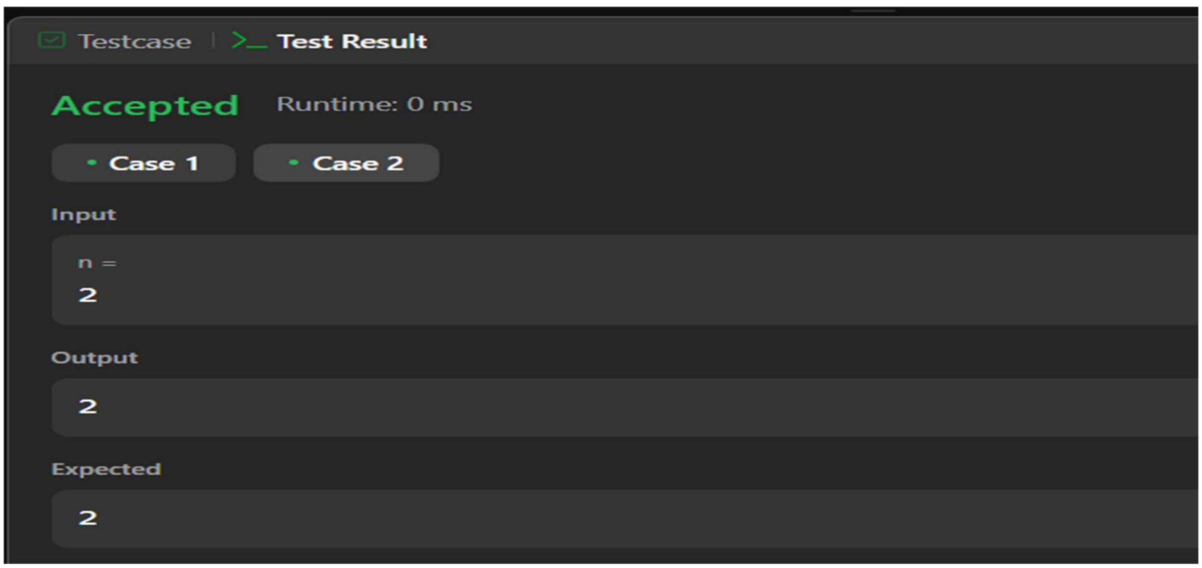

Figure 1

5. Learning Outcome:

- **Understanding Dynamic Programming (DP)** – Learn how to optimize recursive problems using an iterative DP approach with space efficiency.
- **Applying Fibonacci Sequence Logic** – Recognize how problems with overlapping subproblems can be solved using a Fibonacci-like recurrence relation.
- **Optimizing Space Complexity** – Learn how to reduce **O(n) space** (DP array) to **O(1) space** using only two variables.
- **Efficient Iterative Approach** – Understand how avoiding recursion improves performance and prevents stack overflow for large **n**.

Problem-2

1. **Aim:** Maximum subarray
2. **Objectives:**

- **Find the Maximum Subarray Sum**: - Identify the contiguous subarray with the highest sum in the given integer array.
- **Optimize Time Complexity:**- Implement **Kadane's Algorithm** to solve the problem in **O(n) time**, making it efficient for large inputs.
- **Use Dynamic Programming Approach:**- Maintain a running sum (curSum) and update the global maximum (maxSum) dynamically.
- **Handle Negative and Positive Values Efficiently:-** Decide whether to extend the current subarray or start a new one based on the given values.

3. **Implementation/Code:**

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxSum = nums[0], curSum = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            curSum = max(nums[i], curSum + nums[i]);  // Extend or start new subarray
            maxSum = max(maxSum, curSum);  // Update global max
        }

        return maxSum;
    }
};
```

4. **Output:**

5. Learning Outcomes:

- **Understanding Kadane's Algorithm**:- Learn how to efficiently find the maximum subarray sum using a greedy and dynamic programming approach.
- **Optimizing Time Complexity**: - Recognize how an **O(n) linear scan** can be used instead of brute-force **$O(n^2)$ or $O(n^3)$** approaches.
- **Handling Complex Sorting Problems**: - You will gain confidence in solving sorting problems efficiently. This improves your problem-solving ability in technical interviews.
- **Writing Optimized Code**: - The approach ensures sorting is done in one pass. This makes the code faster and reduces unnecessary computations.
- **Better Preparation for Interviews**: - This problem is commonly asked in coding interviews. Practicing it will strengthen your ability to solve sorting-based challenges.

Problem – 3

1. **Aim:** House Robber
2. **Objectives:**

- **Maximize the Loot Without Triggering Security**: - Compute the maximum amount of money that can be robbed without robbing adjacent houses.
- **Optimize Space Complexity**: - Use only **O(1) space** by maintaining two variables (prev1 and prev2) instead of an **O(n) DP array**.
- **Implement Dynamic Programming Efficiently**: - Use a **bottom-up approach** to iteratively compute the best solution without recursion.
- **Ensure Optimal Decision at Each Step**: - At each house, decide whether to rob it (add its value to prev2) or skip it (carry forward prev1).

3. **Implementation/Code:**

```
class Solution {
public:
   int rob(vector<int>& nums) {
      if (nums.empty()) return 0;
      if (nums.size() == 1) return nums[0];

      int prev2 = 0, prev1 = 0;  // prev2: max sum till i-2, prev1: max sum till i-1
      for (int money : nums) {
         int curr = max(prev1, prev2 + money); // Either rob or skip
         prev2 = prev1;  // Move prev1 to prev2
         prev1 = curr;   // Update prev1 to current max
      }
     return prev1;
   }
```

```
};
```

4. **Output:**

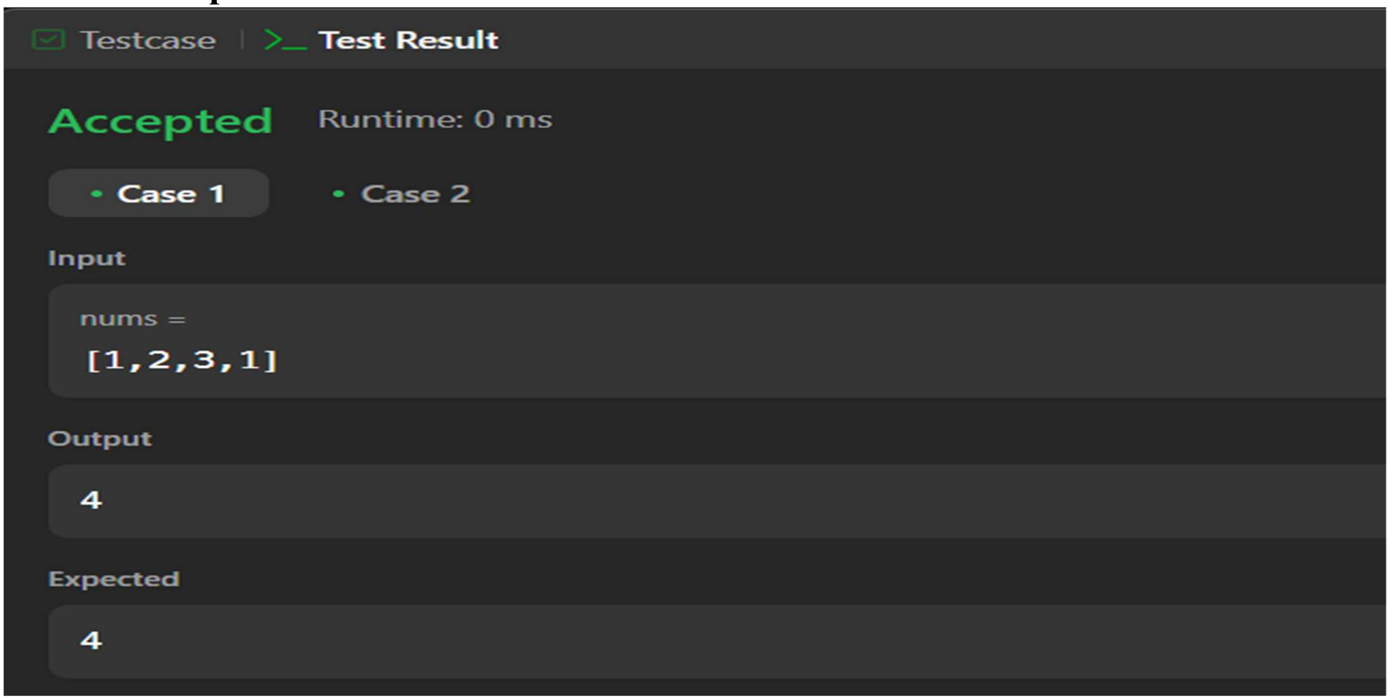

Figure 3

5. Learning Outcomes

- **Efficient Peak Finding**: - You will learn how to locate a peak element without scanning the entire array, using a smarter approach with binary search.
- **Mastering Binary Search Variations**: - You will understand how binary search can be adapted for different problems beyond simple number searching.
- **Developing a Logical Approach**: - You will improve your ability to break down problems logically, making it easier to apply efficient solutions in coding interviews and real-world tasks.
- **Understanding Search Space Reduction**: - You will gain insights into how reducing the search space step by step can lead to significant performance improvements.
- **Building Optimized and Scalable Solutions**:- You will develop the skills to write code that is both time-efficient and scalable, a crucial requirement for competitive programming and software development.