Experiment 6

Student Name: Karan Goyal **UID:-22BCS15864**

Branch: BE-CSE **Group-Ntpp_IOT_602-A**

Semester: 5TH **Date of Performance:-20-2-25**

Subject Name:-Advance Programming lab-2 **Subject Code:-22CSP-351**

Problem 1

Aim:- You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Objective:-The objective is to determine the number of distinct ways to climb a staircase with **n** steps, where each step can be either 1 or 2 at a time. This problem follows the Fibonacci sequence and can be solved using recursion, dynamic programming, or mathematical formulas for efficiency.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm to Count Distinct Ways to Climb Stairs

1. **Initialize Base Cases**
   - If n == 1, return 1 (only one way: single step).
   - If n == 2, return 2 (two ways: (1,1) or (2)).
2. **Use Dynamic Programming**
   - Create an array dp of size n+1.
   - Set dp[1] = 1 and dp[2] = 2.
3. **Iterate from 3 to n**
   - Compute dp[i] = dp[i-1] + dp[i-2] (sum of previous two steps).
4. **Return the Result**
   - Return dp[n], which gives the total distinct ways to reach the top.

Code:

```
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 3) return n;

        int prev1 = 3;
        int prev2 = 2;
        int cur = 0;

        for (int i = 3; i < n; i++) {
            cur = prev1 + prev2;
```

```
            prev2 = prev1;
            prev1 = cur;
        }

        return cur;
    }
};
```

- **Time complexity**: $O(n)$
- **Space complexity:** $O(1)$

Output- All the test cases passed

Learning Outcomes:

1. Understand the Fibonacci-like recurrence relation in the staircase problem.
2. Implement an optimized dynamic programming approach with **O(1)** space complexity.
3. Learn how to use two variables to reduce memory usage instead of an array.
4. Analyze and derive the **O(n)** time complexity of iterative solutions.
5. Gain proficiency in solving combinatorial problems using dynamic programming techniques.

Problem-2

Aim:- Given an integer array nums, find the subarray with the largest sum, and return *its sum*.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Objective- The objective is to find the contiguous subarray within an integer array that has the largest sum and return this sum. This problem is commonly solved using Kadane's Algorithm, which efficiently computes the maximum subarray sum in linear time by iterating through the array while maintaining optimal subarray sums.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm (Kadane's Algorithm) in 6 Steps

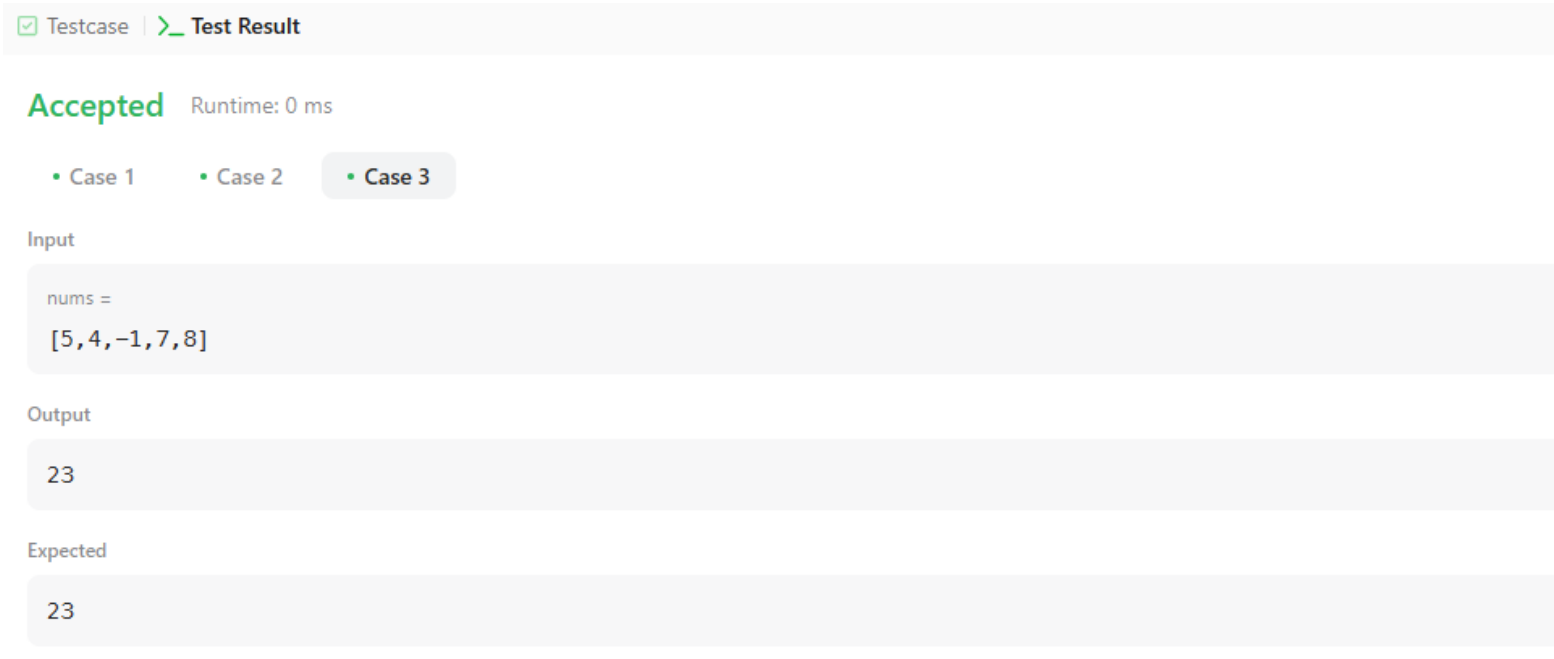
1. **Initialize Variables:** Set res = nums[0] (maximum sum) and total = 0 (current subarray sum).
2. **Loop Through Array:** Iterate over each element n in nums.
3. **Check Negative Sum:** If total < 0, reset total = 0.
4. **Update Subarray Sum:** Add n to total.
5. **Update Maximum Sum:** Set res = max(res, total).
6. **Return Result:** Output res as the largest subarray sum.

Code-

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

        for (int n : nums) {
            if (total < 0) {
                total = 0;
            }

            total += n;
            res = max(res, total);
        }

        return res;
    }
};
```

Time Complexity: O(n).
Space Complexity: O(1)

Result-All test cases passes



Learning Outcomes

1. **Understand Kadane's Algorithm** – Learn how to efficiently find the maximum sum of a contiguous subarray using a greedy approach that dynamically updates the sum while iterating through the array.
2. **Apply Dynamic Programming Concepts** – Recognize how Kadane's Algorithm follows the principle of dynamic programming by maintaining an optimal substructure and solving the problem in a single pass.
3. **Analyze Time and Space Complexity** – Understand that the algorithm runs in **O(n) time complexity** as it processes each element once, and **O(1) space complexity** since it only uses a few extra variables.
4. **Handle Negative Sums Effectively** – Learn the importance of resetting the running sum when it becomes negative, as negative values reduce the sum of future subarrays, making it beneficial to start fresh.
5. **Improve Problem-Solving with Iterative Techniques** – Gain experience in iterating through an array efficiently while maintaining and updating an optimal subarray sum dynamically, a technique useful in various optimization problems.

Problem-3

Aim- You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight **without alerting the police***.

Objective- The objective is to determine the maximum amount of money that can be robbed from a series of houses, given that robbing two adjacent houses triggers an alarm. Using dynamic programming, the optimal solution ensures the highest possible earnings while avoiding consecutive thefts, maximizing profit without alerting the police.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm: House Robber Problem

1. **Initialize Base Cases**: If there is only one house, return its value as the maximum amount.
2. **Create DP Array**: Initialize a dynamic programming (DP) array dp of size n to store maximum money robbed up to each house.
3. **Set Initial Values**: Assign dp[0] = nums[0] and dp[1] = max(nums[0], nums[1]), as the first two houses define initial choices.
4. **Iterate Through Houses**: From house index 2 to n-1, calculate dp[i] as the maximum of:
    - Not robbing the current house (dp[i-1])
    - Robbing the current house (nums[i] + dp[i-2])
5. **Store Maximum Profit**: Update dp[i] with the maximum of the two choices for each house.
6. **Return Result**: The final answer is stored in dp[n-1], which gives the maximum amount that can be robbed without triggering the alarm.

Code-

```
class Solution {

public:

   int rob(vector<int>& nums) {

      int n = nums.size();

      if (n == 1) {

         return nums[0]; }
```

```
        vector<int> dp(n, 0);
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        for (int i = 2; i < n; i++) {
            dp[i] = max(dp[i - 1], nums[i] + dp[i - 2]);
        }
        return dp[n - 1];
    }
};
```

Time Complexity: O(n)
Space Complexity: O(n)

Result- All test cases passes

Learning Outcomes:

1. Understand how **dynamic programming** efficiently solves problems by breaking them into overlapping subproblems and using stored results to avoid redundant calculations.

2. Learn to handle constraints like **avoiding consecutive selections** by defining optimal choices at each step.

3. Gain practical experience in **bottom-up DP table filling**, where results are computed iteratively rather than using recursion.

4. Analyze **time and space complexity**, understanding trade-offs and optimizing space by reducing extra storage when possible.

5. Develop problem-solving skills by implementing **iterative DP solutions**, making decisions based on previous computations to maximize the result.