DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Experiment 6

Student Name: Rhythm Tyagi	UID: 22BCS17203
Branch: CSE	Section: NTPP_602-A
Semester: 6th	DOP: 20/02/25
Subject: AP-LAB-2	Subject Code:22CSP-351

Aim:

Problem 6.1: **Maximum Subarray**

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

Problem 6.2: **Climbing Stairs**

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

Problem 6.3: **Jump Game**

You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

Algorithms:

Algo 6.1:

1. **Initialize:**
2. maxSum = nums[0] → Stores the maximum subarray sum found so far.
3. currentSum = nums[0] → Tracks the max sum ending at the current index.
4. **Iterate through nums from index 1 to n-1:**
5. Update currentSum = max(nums[i], currentSum + nums[i])
   (Choose to start a new subarray or extend the existing one).
6. Update maxSum = max(maxSum, currentSum).
7. **Return maxSum** as the maximum subarray sum

Algo 6.2:

1. **Base Cases:**
2. If n == 1, return 1 (only one way to climb).
3. Initialize dp array with dp[1] = 1 and dp[2] = 2.
4. **Iterate from i = 3 to n:**
5. Use recurrence relation: dp[i] = dp[i - 1] + dp[i - 2]
   (ways to reach i by taking 1 or 2 steps).
6. **Return dp[n]** as the total number of ways to reach the nth step.

Algo 6.3:

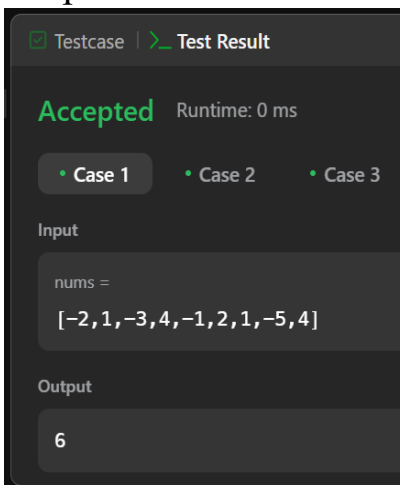
1. **Initialize maxReach = 0** → Tracks the farthest index we can reach.
2. **Iterate through nums array** using index i:
3. If i > maxReach, return false (current index is unreachable).

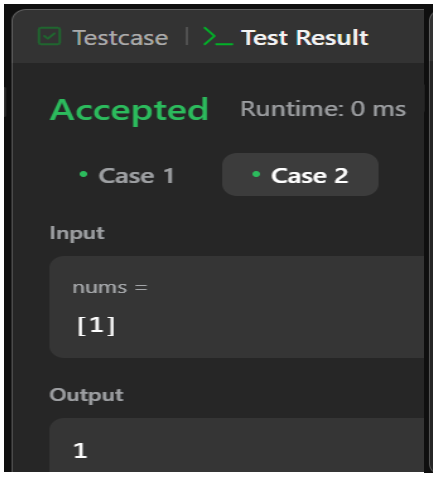
4. Update maxReach = max(maxReach, i + nums[i]) (maximum reach from current position).
5. If maxReach ≥ last index, return true (we can reach the end early).
6. **Return true if maxReach covers the last index, else false.**

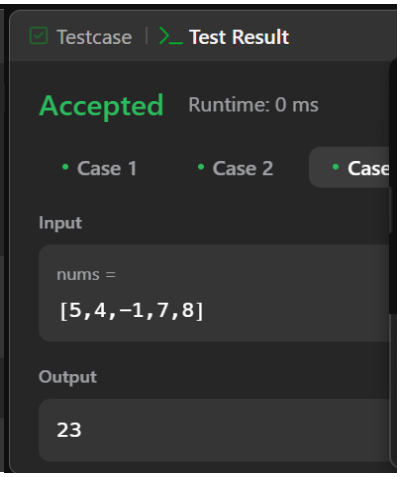
Code: 6.1

```
class Solution {
  public int maxSubArray(int[] nums) {
    int maxSum = nums[0];  // Stores the maximum subarray sum found so far
    int currentSum = nums[0];  // Tracks the max sum ending at the current index

    for (int i = 1; i < nums.length; i++) {
      currentSum = Math.max(nums[i], currentSum + nums[i]);  // Either extend the subarray or start new
      maxSum = Math.max(maxSum, currentSum);  // Update maxSum if currentSum is larger
    }

    return maxSum;
  }
}
```

Output:



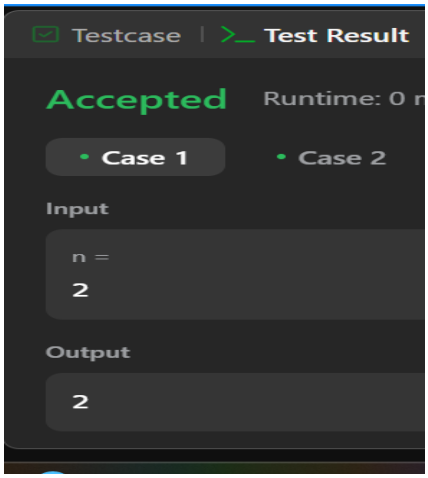


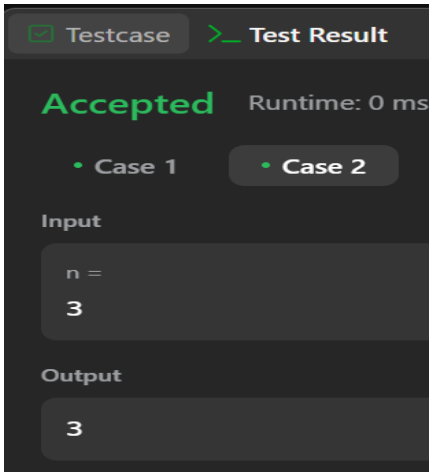


CODE: 6.2

```
class Solution {
    public int climbStairs(int n) {
        if (n == 1) return 1;
        int[] dp = new int[n + 1];  // Array to store computed results
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];  // Recurrence relation
        }
        return dp[n];  // Return the total ways to reach the nth step
    }
}
```

OUTPUT:

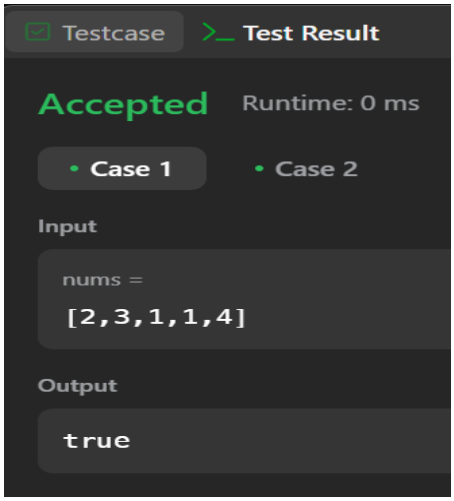


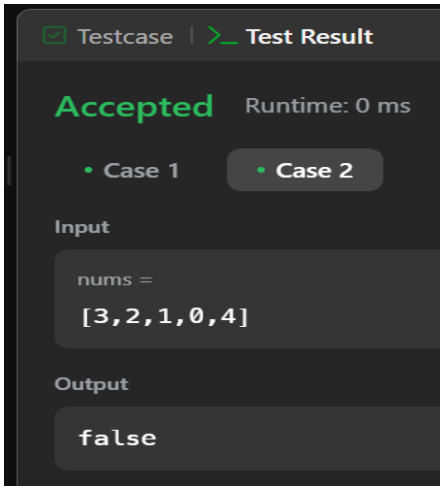


CODE: 6.3

```
public class Solution {
    public boolean canJump(int[] nums) {
        int maxReach = 0;  // Track the farthest index we can reach
                for (int i = 0; i < nums.length; i++) {
            if (i > maxReach) {
                return false;  // If current index is unreachable, return false
            }
            maxReach = Math.max(maxReach, i + nums[i]);  // Update the farthest reachable index
                    // Early exit if we can already reach the last index
            if (maxReach >= nums.length - 1) {
                return true;
            }
        }
        return maxReach >= nums.length - 1;
    }
    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] nums1 = {2, 3, 1, 1, 4};
        System.out.println(sol.canJump(nums1));  // Output: true
        int[] nums2 = {3, 2, 1, 0, 4};
        System.out.println(sol.canJump(nums2));  // Output: false
    }
}
```

OUTPUT:





Learning Outcomes:

Greedy & DP mastery – Used optimal strategies for subarray sum and jump game.
Efficient updates – Maximized reach or sum with simple comparisons.
Early termination – Stopped iteration when the goal was reached.
Space efficiency – Used O(1)O(1)O(1) extra space for both problems.
Iterative approach – Avoided recursion for better performance.