



## Experiment-7

**Student Name:** Jatin Garg

**Branch:** BE-CSE

**Semester:** 6<sup>th</sup>

**Subject Name:** AP LAB-II

**UID:** 22BCS15676

**Section/Group:** 640/B

**Date of Performance:** 10/03/25

**Subject Code:** 22CSP-351

### 1. Aim:

Dynamic Programming (DP) is an optimization technique used to solve complex problems by breaking them into smaller overlapping subproblems, solving each subproblem only once, and storing the results to avoid redundant calculations. It is mainly applied to optimization problems where we need to find the minimum, maximum, shortest, or longest solution.

### 2. Introduction to the Searching and Sorting:

#### Key Steps in DP Approach:

- Break the problem into smaller subproblems (like divide-and-conquer).
- Find the optimal solution for each subproblem.
- Store results of subproblems (memoization) to avoid recomputation.
- Reuse stored results to efficiently compute the final solution.

#### Types of DP Approaches:

1. Top-Down (Memoization) – Uses recursion with caching to store intermediate results.
2. Bottom-Up (Tabulation) – Uses an iterative approach to build solutions from smaller subproblems.

### 3. Implementation/Code:

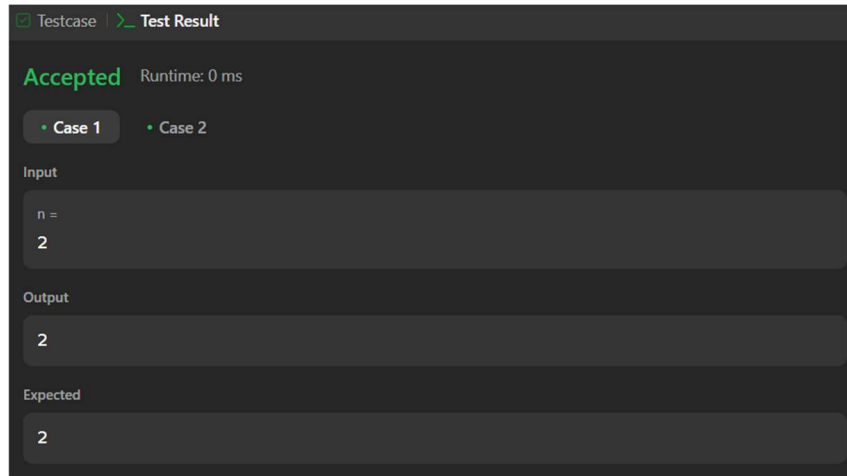
#### 70 Climbing Stairs:

```
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) return n;

        int first = 1, second = 2;
        for (int i = 3; i <= n; i++) {
            int temp = first + second;
            first = second;
            second = temp;
        }

        return second;
    }
}
```

## 70 Output:



### 53 Maximum Subarray:

```
class Solution {
    public int maxSubArray(int[] nums) {
        return findMaxSubArray(nums, 0, nums.length - 1);
    }

    private int findMaxSubArray(int[] nums, int left, int right) {
        if (left == right) return nums[left];

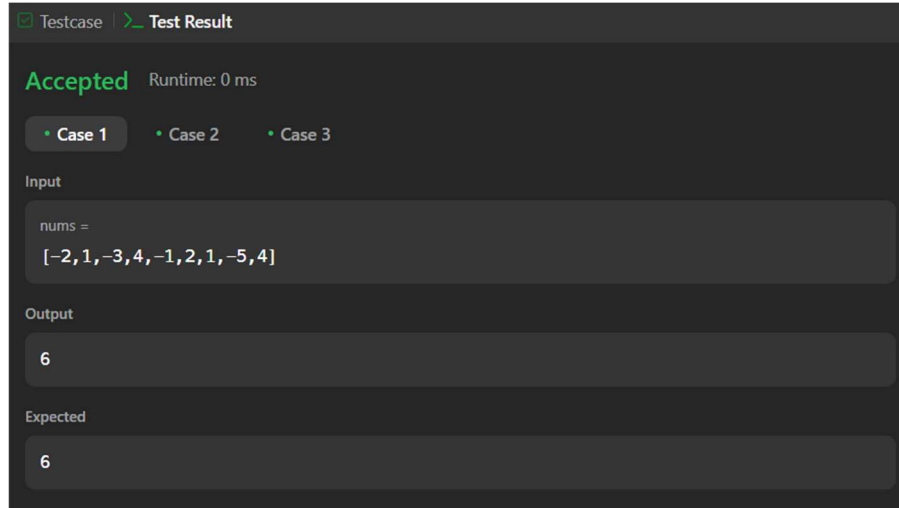
        int mid = left + (right - left) / 2;
        int leftMax = findMaxSubArray(nums, left, mid);
        int rightMax = findMaxSubArray(nums, mid + 1, right);
        int crossMax = findMaxCrossingSum(nums, left, mid, right);
        return Math.max(leftMax, Math.max(rightMax, crossMax));
    }

    private int findMaxCrossingSum(int[] nums, int left, int mid, int right) {
        int leftSum = Integer.MIN_VALUE, rightSum = Integer.MIN_VALUE;
        int sum = 0;
        for (int i = mid; i >= left; i--) {
            sum += nums[i];
            leftSum = Math.max(leftSum, sum);
        }

        sum = 0;
        for (int i = mid + 1; i <= right; i++) {
            sum += nums[i];
            rightSum = Math.max(rightSum, sum);
        }

        return leftSum + rightSum;
    }
}
```

## 53 Output:



Testcase | Test Result

**Accepted** Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

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

Output

6

Expected

6

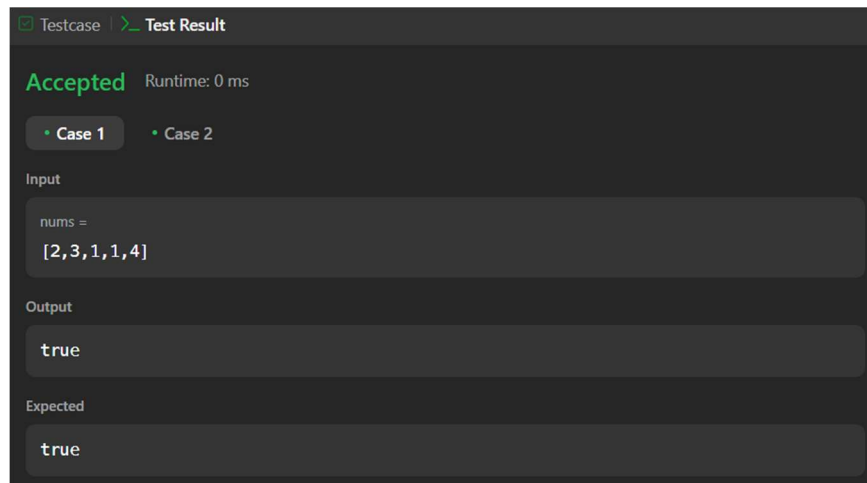
## ▪ 55 Jump Game:

```
class Solution {
    public boolean canJump(int[] nums) {
        int n = nums.length;
        boolean[] dp = new boolean[n];
        dp[0] = true; // Start is always reachable

        for (int i = 0; i < n; i++) {
            if (!dp[i]) continue; // If not reachable, skip
            for (int j = i + 1; j <= Math.min(i + nums[i], n - 1); j++) {
                dp[j] = true;
            }
        }

        return dp[n - 1];
    }
}
```

## 55 Output:



Testcase | Test Result

**Accepted** Runtime: 0 ms

• Case 1 • Case 2

Input

nums =  
[2, 3, 1, 1, 4]

Output

true

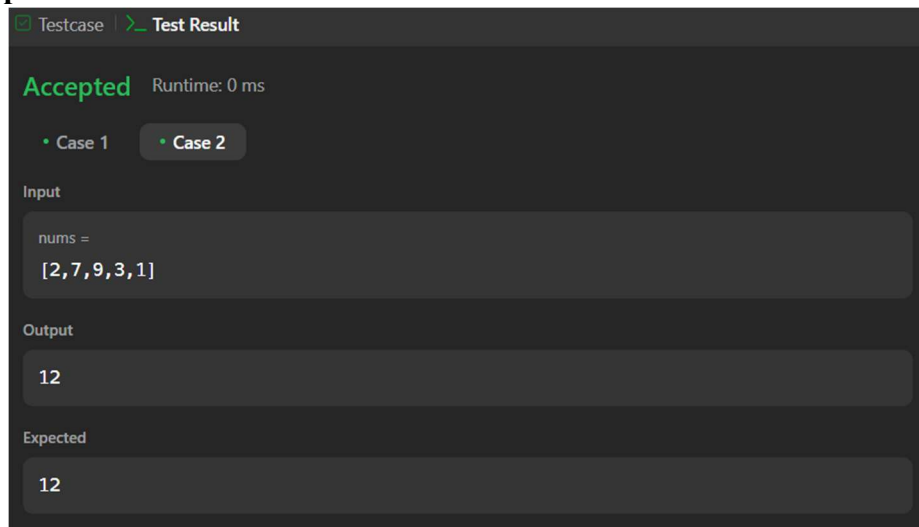
Expected

true

## ▪ 198 House Robber:

```
class Solution {  
    public int rob(int[] nums) {  
        if (nums.length == 0) return 0;  
        if (nums.length == 1) return nums[0];  
  
        int prev2 = 0, prev1 = 0;  
  
        for (int num : nums) {  
            int newRob = Math.max(num + prev2, prev1);  
            prev2 = prev1;  
            prev1 = newRob;  
        }  
  
        return prev1;  
    }  
}
```

## 198 Output:



## 4. Learning Outcome

- Learned to break down problems into overlapping subproblems.
- Understanding binary search to efficiently solve problems involving sorted data structures.
- Learned recursive and iterative DFS/BFS approaches for tree-based problems.