Experiment: 7

Student Name: Deepak
UID: 22BCS12192
Branch: BE-CSE
Section: NTPP_IOT_603(B)
Semester: 6th
Date of Performance: March 21, 2025
Subject: Advanced Programming Lab-2
Subject Code: 22CSP-351

1. Aim:

Problem 1.1: Climbing Stairs

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?

Problem 1.2: Maximum Subarray

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

Start by initializing maxSum and currentSum with the first element of the array.

- Update currentSum to be either the current element itself or the current element plus the previous currentSum. This decision effectively decides whether to continue the existing subarray or start a new one.
- Update maxSum if currentSum exceeds it.
- **Return Result**: After iterating through the array, maxSum will contain the maximum subarray sum.

Problem 1.3: Coin Change

Problem Statement: You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

- **Initialization**: Create a `dp` array of size `amount + 1`, initialized to `INT_MAX` (to represent infinity). The first element, `dp[0]`, is set to `0` since no coins are needed to make the amount `0`.
- **Iterate Through Coins**: For each coin, update the `dp` array for all amounts from the coin value up to the target amount.
- **Update DP Array**: For each amount `i`, check if using the coin leads to a smaller number of coins than previously recorded in `dp[i]`. If so, update `dp[i]`.
- **Final Check**: After filling the `dp` array, check `dp[amount]`. If it remains `INT_MAX`, return `-1` (indicating it's not possible to form that amount). Otherwise, return `dp[amount]`.

2. Objective:

- Develop proficiency in applying Dynamic Programming to solve various algorithmic problems efficiently.

3. Implementation / Code:

3.1:

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        return max(leftDepth, rightDepth) + 1;
    }
};
```

3.2:

```
class Solution {
```

```
public:
    int maxSubArray(std::vector<int>& nums) {
        return divideAndConquer(nums, 0, nums.size() - 1);
    }

private:
    int divideAndConquer(std::vector<int>& nums, int left, int right) {
        if (left == right) return nums[left];

        int mid = left + (right - left) / 2;

        int leftMax = divideAndConquer(nums, left, mid);
        int rightMax = divideAndConquer(nums, mid + 1, right);

        int crossMax = findCrossMax(nums, left, mid, right);

        return std::max({leftMax, rightMax, crossMax});
    }

    int findCrossMax(std::vector<int>& nums, int left, int mid, int right) {
        int leftSum = INT_MIN, rightSum = INT_MIN;
        int sum = 0;

        for (int i = mid; i >= left; --i) {
            sum += nums[i];
            leftSum = std::max(leftSum, sum);
        }

        sum = 0;

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

        return leftSum + rightSum;
    }
};
```

3.3:

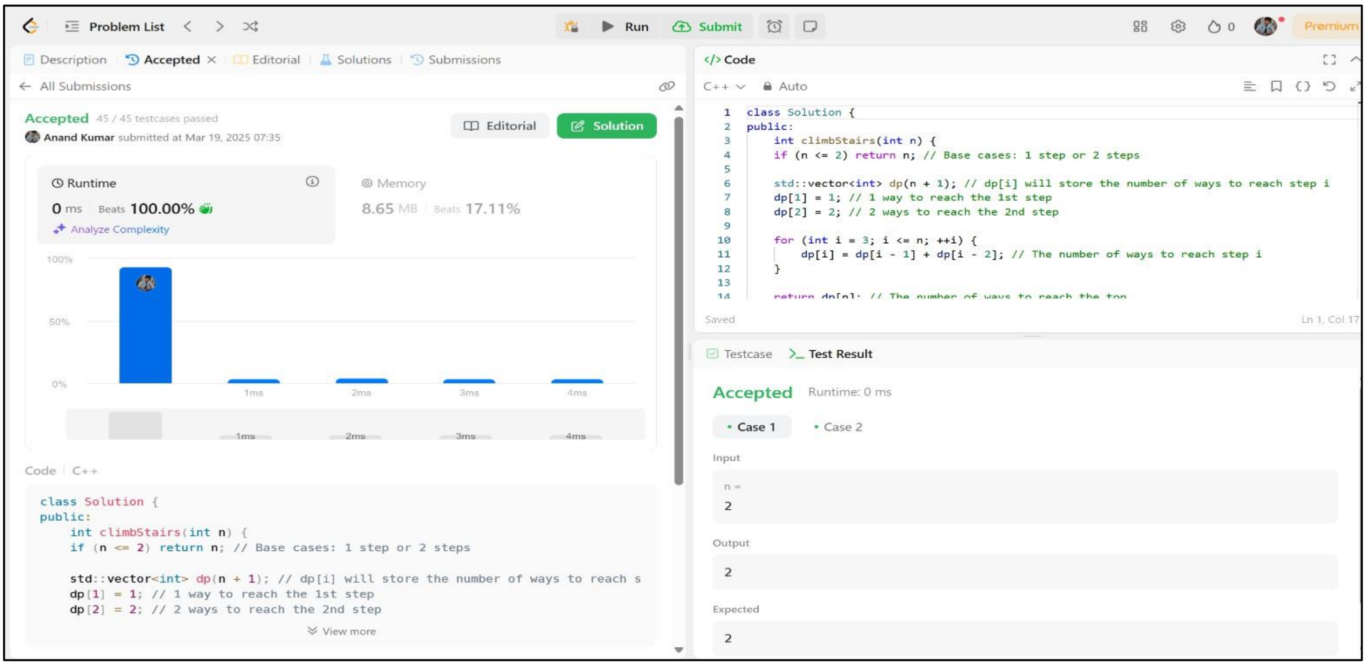
```
class Solution {
public:
    int coinChange(std::vector<int>& coins, int amount) {
        std::vector<int> dp(amount + 1, INT_MAX);
        dp[0] = 0;

        for (int coin : coins) {
            for (int i = coin; i <= amount; ++i) {
                if (dp[i - coin] != INT_MAX) {
                    dp[i] = std::min(dp[i], dp[i - coin] + 1);
                }
            }
```

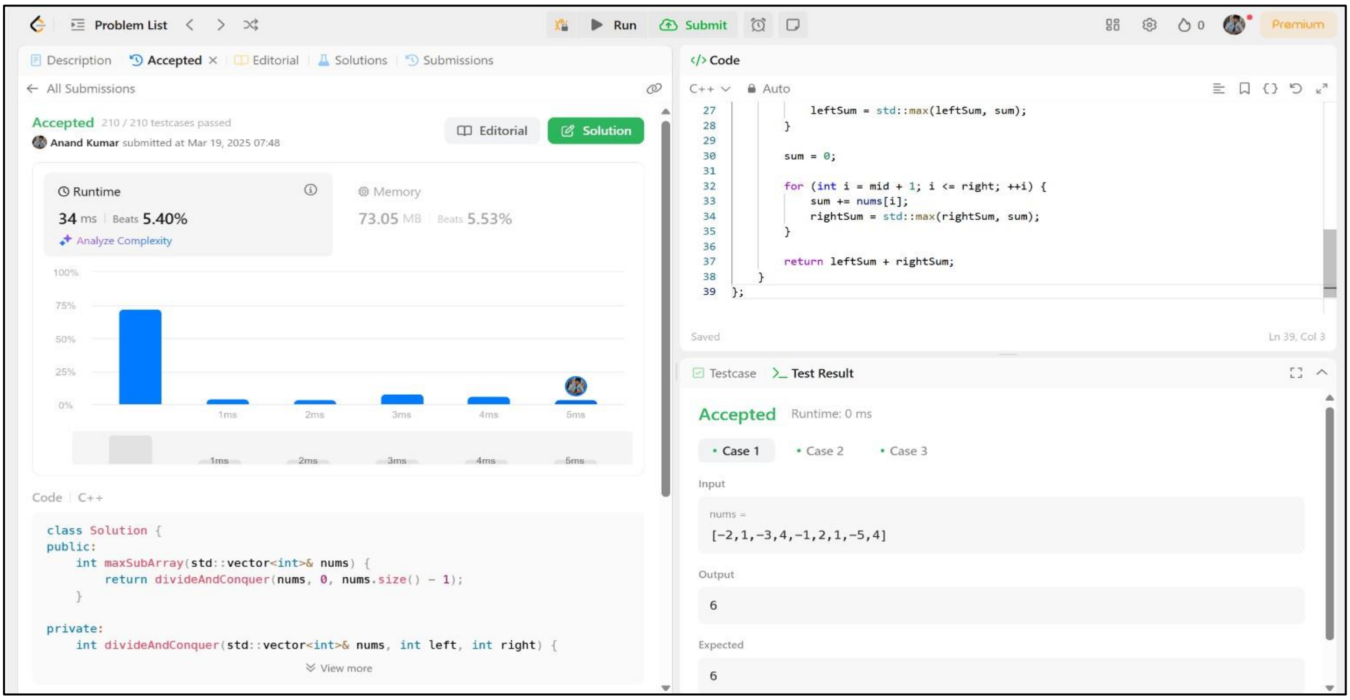
```
        }

        return dp[amount] == INT_MAX ? -1 : dp[amount];
    }
};
```

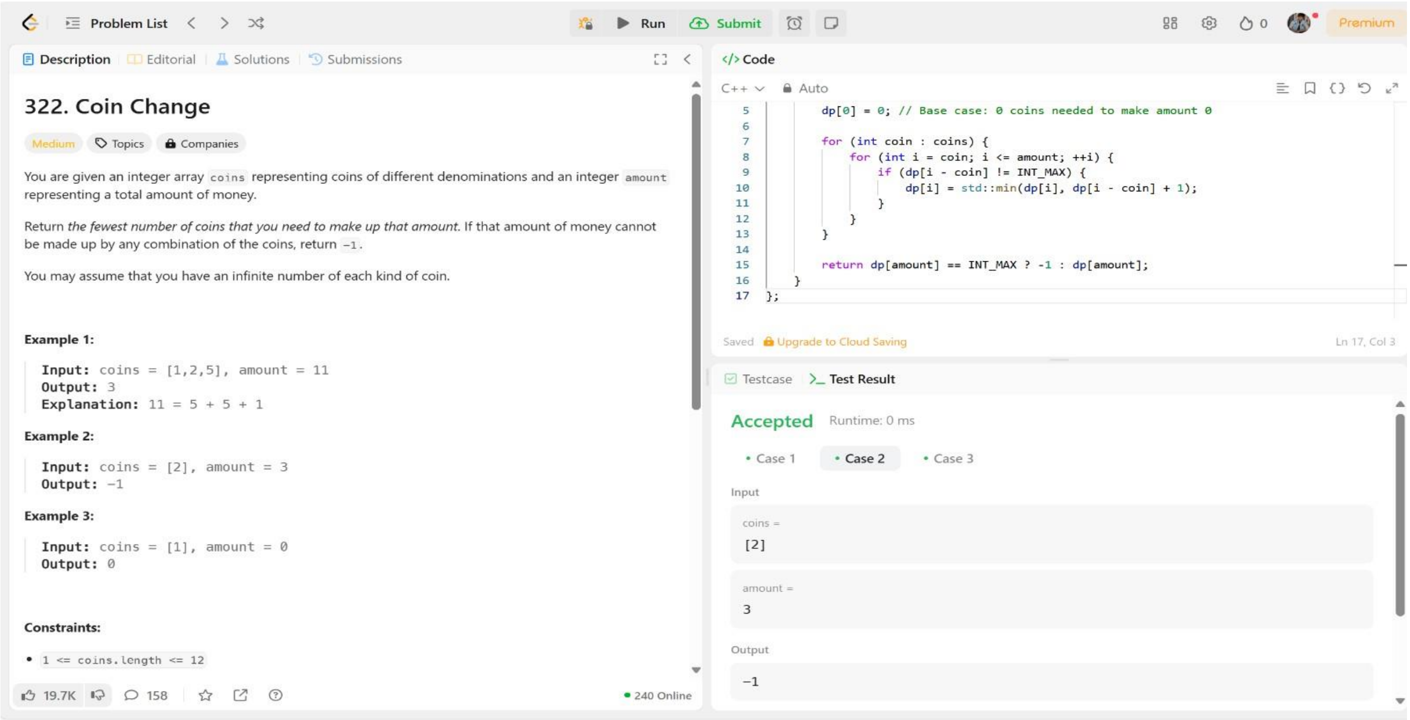
4. Output



Snapshot 1: Climbing Stairs



Snapshot 2: [Maximum Subarray]



Snapshot3: [Coin Change]

5. Learning Outcome:

- Understand how to apply dynamic programming principles to optimize solutions for problems involving overlapping subproblems and optimal substructure.
- Gain experience with Kadane's Algorithm, which efficiently solves the maximum subarray problem in linear time.
- Further explore dynamic programming techniques, particularly in the context of optimization problems.
- Understand how to maintain and update state variables while iterating through an array.
- Learn to formulate recurrence relations that express the relationship between the solution of a problem and smaller subproblems.