Experiment-7

Student Name: Garv Kumar **UID:** 22BET10103
Branch: BE-IT **Section/Group:** 22BET_IOT-702/A
Semester: 6th **Date of Performance:** 20/03/2025
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22ITP-351

Problem-1

1. **Aim:**

   To develop an efficient algorithm to determine the number of distinct ways to climb a staircase with n steps, where one can take either 1 or 2 steps at a time.

2. **Objective:**
   - Implement a dynamic programming or mathematical approach to efficiently compute the number of ways to climb n steps.
   - Optimize time and space complexity to handle large values of n effectively.

3. **Implementation:**

```
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 2) return n;
        int a = 1, b = 2;
        for (int i = 3; i <= n; i++) {
            int temp = a + b;
            a = b;
            b = temp;
        }
        return b;
    }
```

```
};
```

4. Output:

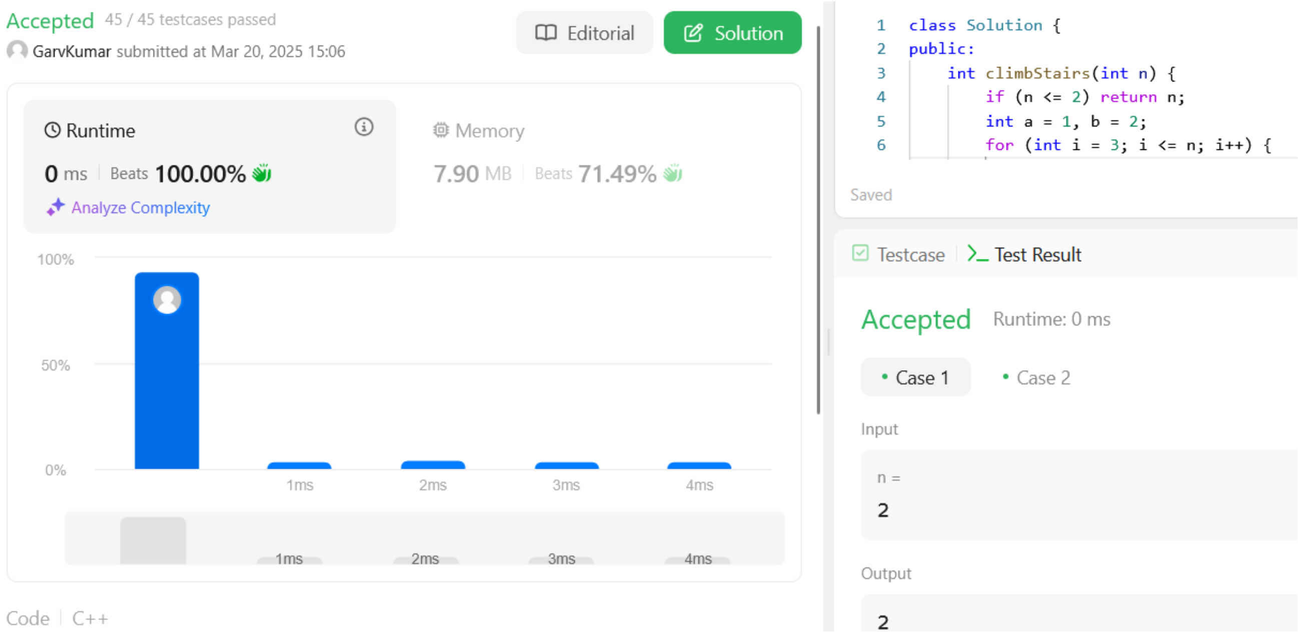

Fig: Climbing Stairs.

Problem-2

1. Aim:

To develop an algorithm that determines the maximum profit that can be achieved by buying and selling a stock on different days, given an array of stock prices.

2. Objective:

1 Implement an efficient approach to find the maximum profit by choosing the best buy and sell days.

2 Optimize time complexity to solve the problem in linear time O(n)O(n) using a single pass solution.

3. Implementation:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int minPrice = INT_MAX, maxProfit = 0;
        for (int price : prices) {
```

```
            minPrice = min(minPrice, price);
            maxProfit = max(maxProfit, price - minPrice);
        }
        return maxProfit;
    }
};
```

4. Output:

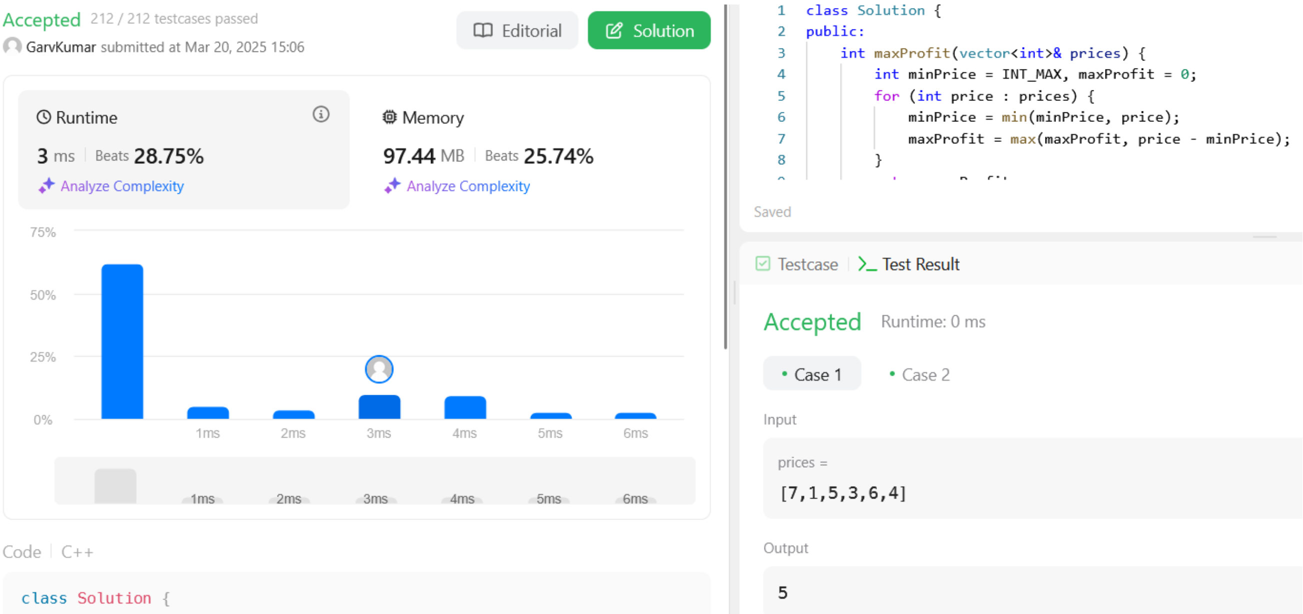

Fig: Best Time to Buy and Sell Stock.

Problem-3

1. Aim:

To design an algorithm that determines the maximum amount of money a robber can steal from a row of houses without robbing two adjacent houses.

2. Objective:

1 Implement a dynamic programming approach to efficiently compute the maximum amount that can be stolen.

2 Optimize time and space complexity to ensure the solution runs efficiently for large inputs.

3. Implementation:

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 1) return nums[0];
        int prev2 = 0, prev1 = 0;
        for (int num : nums) {
            int curr = max(prev1, prev2 + num);
            prev2 = prev1;
            prev1 = curr;
        }
        return prev1;
    }
};
```

4. Output:

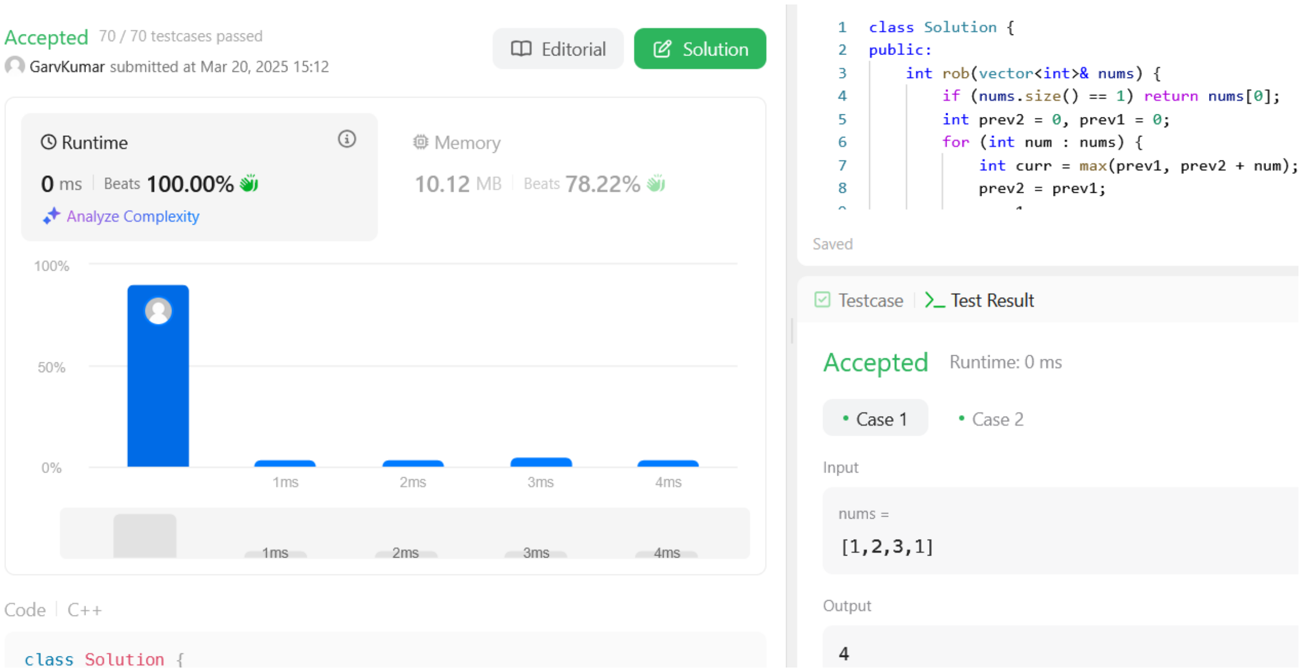

Fig: House Robber.

Problem-4

1. Aim:

To develop an algorithm that finds the minimum number of coins required to make up a given amount using a set of denominations.

2. Objective:

1. Implement a dynamic programming approach to determine the minimum number of coins needed for a given amount.
2. Optimize time and space complexity to handle large inputs efficiently.

3. Implementation:

```
class Solution {

public:

    int coinChange(vector<int>& coins, int amount) {

        vector<int> dp(amount + 1, amount + 1);

        dp[0] = 0;

        for (int i = 1; i <= amount; i++) {

            for (int coin : coins) {

                if (i >= coin) {

                    dp[i] = min(dp[i], dp[i - coin] + 1);

                }

            }

        }

        return dp[amount] == amount + 1 ? -1 : dp[amount];

    }

 };
```

4. Output:

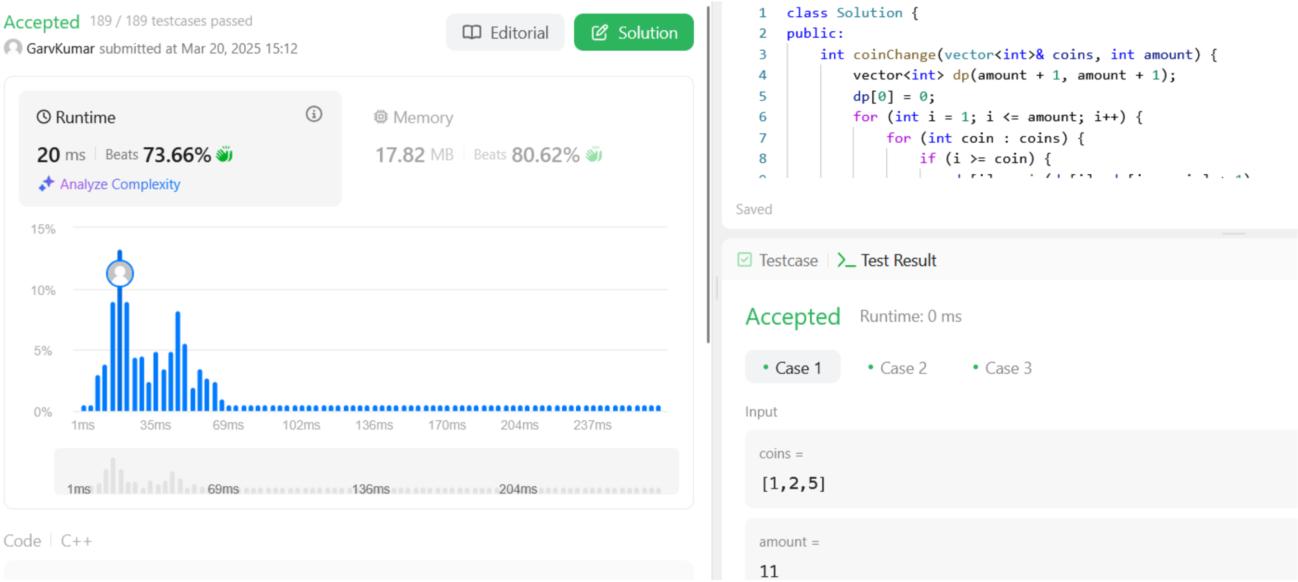

Fig: Coin Change.

Problem-5

1. Aim:

To develop an algorithm that determines the number of ways a given encoded string of digits can be decoded into letters following a specific mapping.

2. Objective:

1 Implement a dynamic programming approach to efficiently count the possible decoding ways.

2 Optimize time and space complexity to handle large input strings effectively.

3. Implementation:

```
class Solution {
public:
    int numDecodings(string s) {
        if (s[0] == '0') return 0;
        int n = s.size();
        int prev2 = 1, prev1 = 1;
        for (int i = 1; i < n; i++) {
```

```
            int curr = 0;
            if (s[i] != '0') curr = prev1;
            int twoDigit = stoi(s.substr(i - 1, 2));
            if (s[i - 1] != '0' && twoDigit >= 10 && twoDigit <= 26)
                curr += prev2;
            prev2 = prev1;
            prev1 = curr;
        }
        return prev1;
    }
};
```

4. Output:

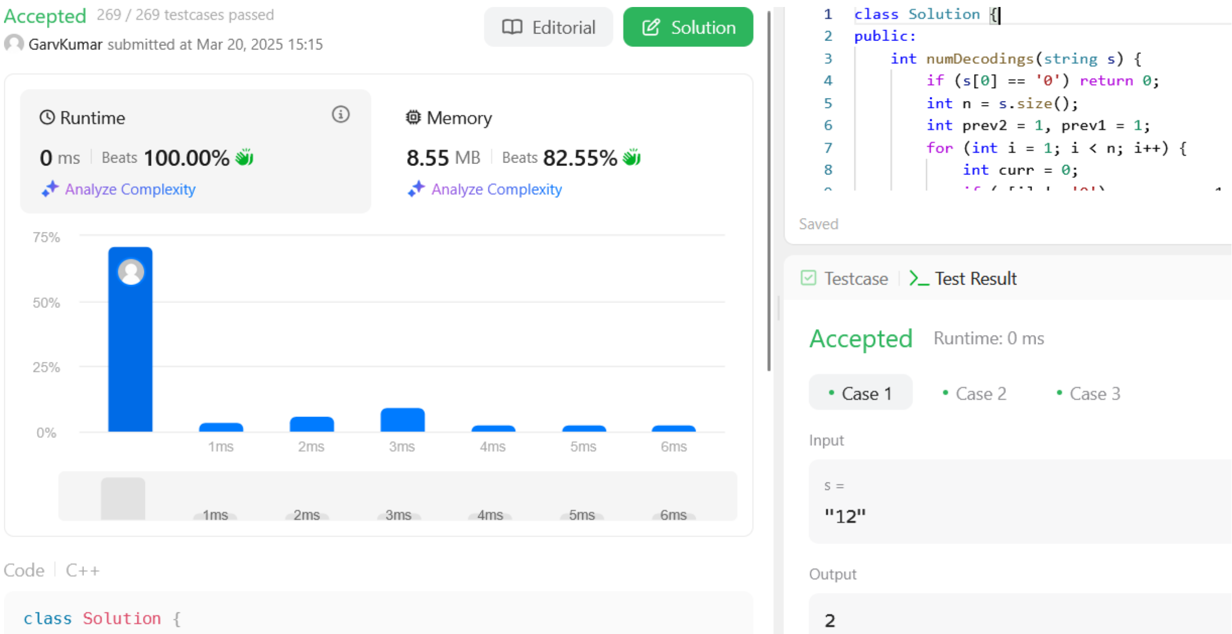

Fig: Decode Ways.

Problem-6

1. Aim:

To design an algorithm that determines the maximum profit from stock trading, considering a cooldown period after selling a stock.

2. Objective:

1 Implement a dynamic programming approach to maximize profit while ensuring no transactions occur during the cooldown period.

2 Optimize time and space complexity to efficiently handle large input arrays.

3. Implementation:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        if (n < 2) return 0;
        int hold = -prices[0], sell = 0, cooldown = 0;
        for (int i = 1; i < n; i++) {
            int prev_sell = sell;
            sell = hold + prices[i];
            hold = max(hold, cooldown - prices[i]);
            cooldown = max(cooldown, prev_sell);
        }
        return max(sell, cooldown);
    }
 };
```

4. Output:

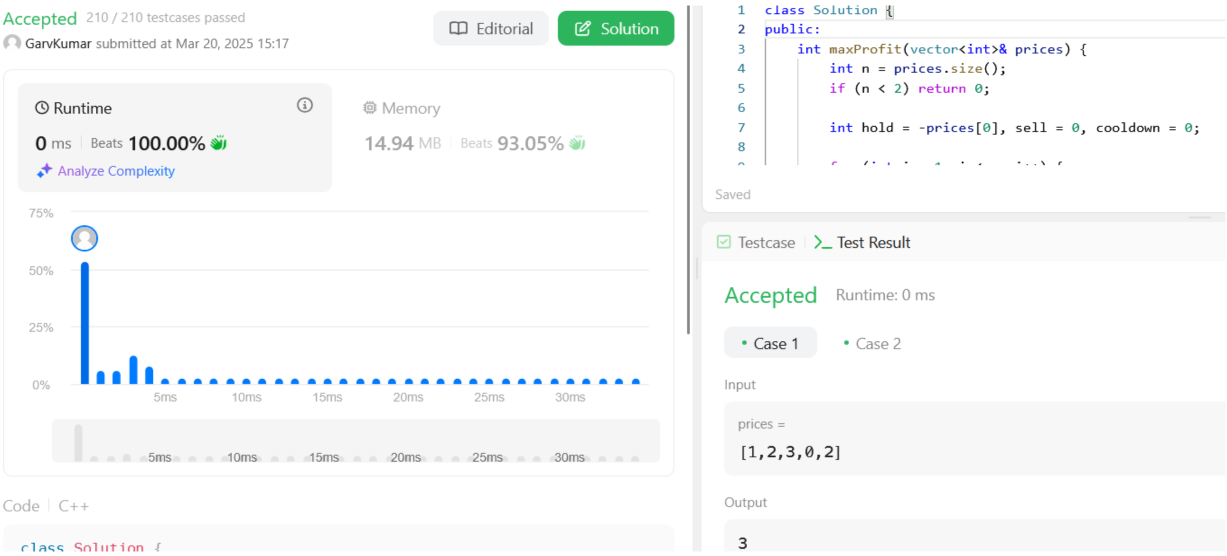

Fig: Best Time to Buy and Sell Stock with Cooldown.

Problem-7

1. Aim:

To develop an algorithm that finds the minimum number of perfect square numbers that sum up to a given integer.

2. Objective:

1 Implement a dynamic programming or mathematical approach to determine the least number of perfect squares needed for a given number.

2 Optimize time and space complexity to handle large inputs efficiently.

3. Implementation:

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;

        for (int i = 1; i * i <= n; i++) {
            int square = i * i;
            for (int j = square; j <= n; j++) {
                dp[j] = min(dp[j], dp[j - square] + 1);
            }
        }
        return dp[n];
    }
 };
```

4. Output:

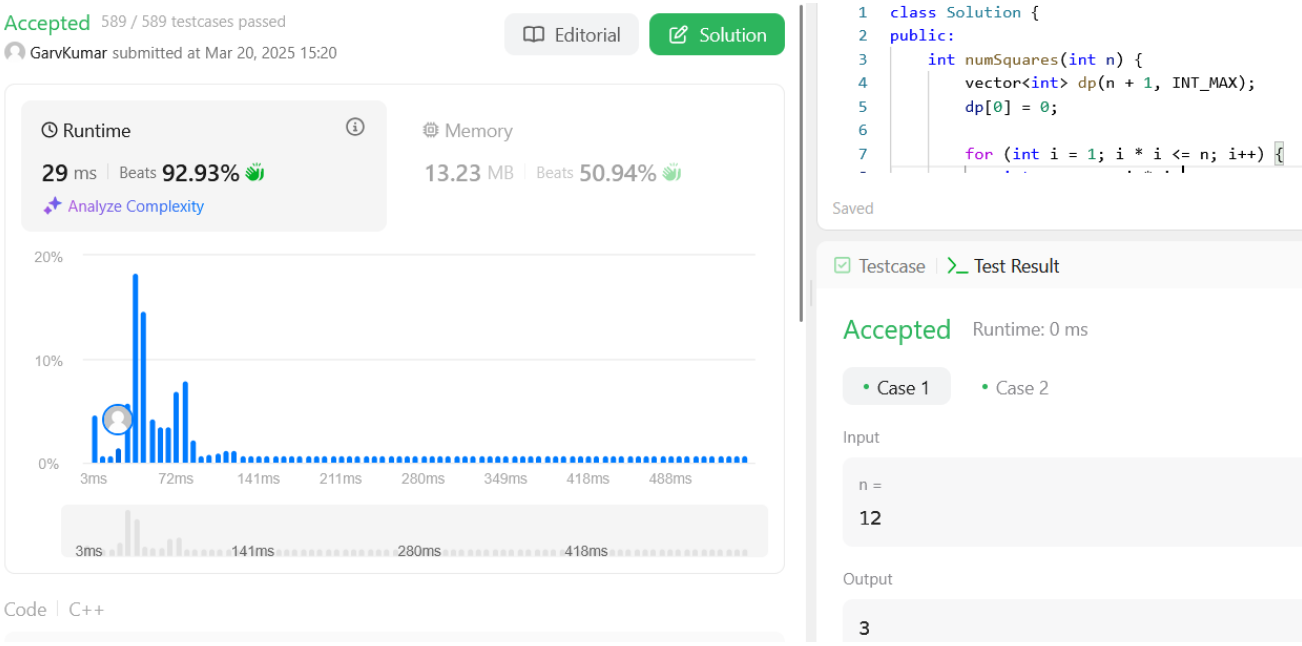

Fig: Perfect Squares.

5. Learning Outcomes:

1. **Climbing Stairs:**
   - Understand and implement recursion and dynamic programming to solve combinatorial problems.
   - Analyze time and space complexity to optimize step-based problems efficiently.
2. **Best Time to Buy and Sell Stock:**
   - Develop an optimal strategy using a single-pass approach to maximize stock trading profits.
   - Learn to track minimum values dynamically to find the maximum difference efficiently.
3. **House Robber:**
   - Apply dynamic programming to solve problems involving non-adjacent selections for maximum gain.
   - Optimize space complexity by reducing redundant computations in decision-making problems.

4. **Coin Change:**
    - Understand the application of dynamic programming in solving the minimum coin change problem.
    - Learn to break down a problem into subproblems and use memoization for efficient computation.
5. **Decode Ways:**
    - Implement dynamic programming to decode digit sequences based on given constraints.
    - Improve problem-solving skills by handling edge cases like leading zeros and invalid encodings.
6. **Best Time to Buy and Sell Stock with Cooldown:**
    - Understand state-based dynamic programming to incorporate cooldown constraints in stock trading.
    - Develop an optimized approach to track multiple states and transitions for maximum profit.
7. **Perfect Squares:**
    - Apply dynamic programming and number theory concepts to minimize the sum of perfect squares.
    - Improve problem-solving skills by identifying optimal substructures and using precomputed results.