Experiment 7

Student Name: Payal Singh **UID:22BET10347**
Branch: BE-IT **Section/Group: IOT-702(A)**
Semester: 6 **Date of Performance:20/03/25**
Subject Name: AP LAB-II **Subject Code: 22ITP-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?

Code:

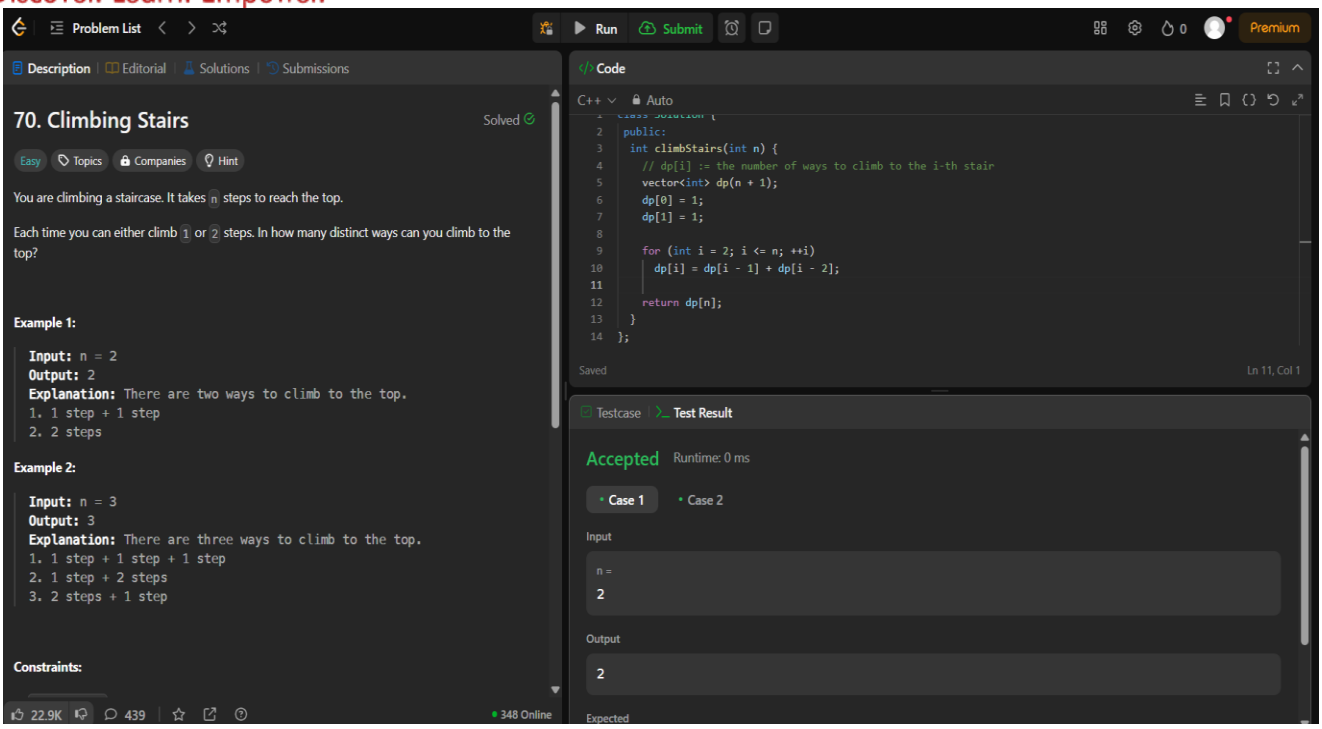
```
class Solution {
 public:
  int climbStairs(int n) {
    // dp[i] := the number of ways to climb to the i-th stair
    vector<int> dp(n + 1);
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i <= n; ++i)
      dp[i] = dp[i - 1] + dp[i - 2];

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

Output:

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.



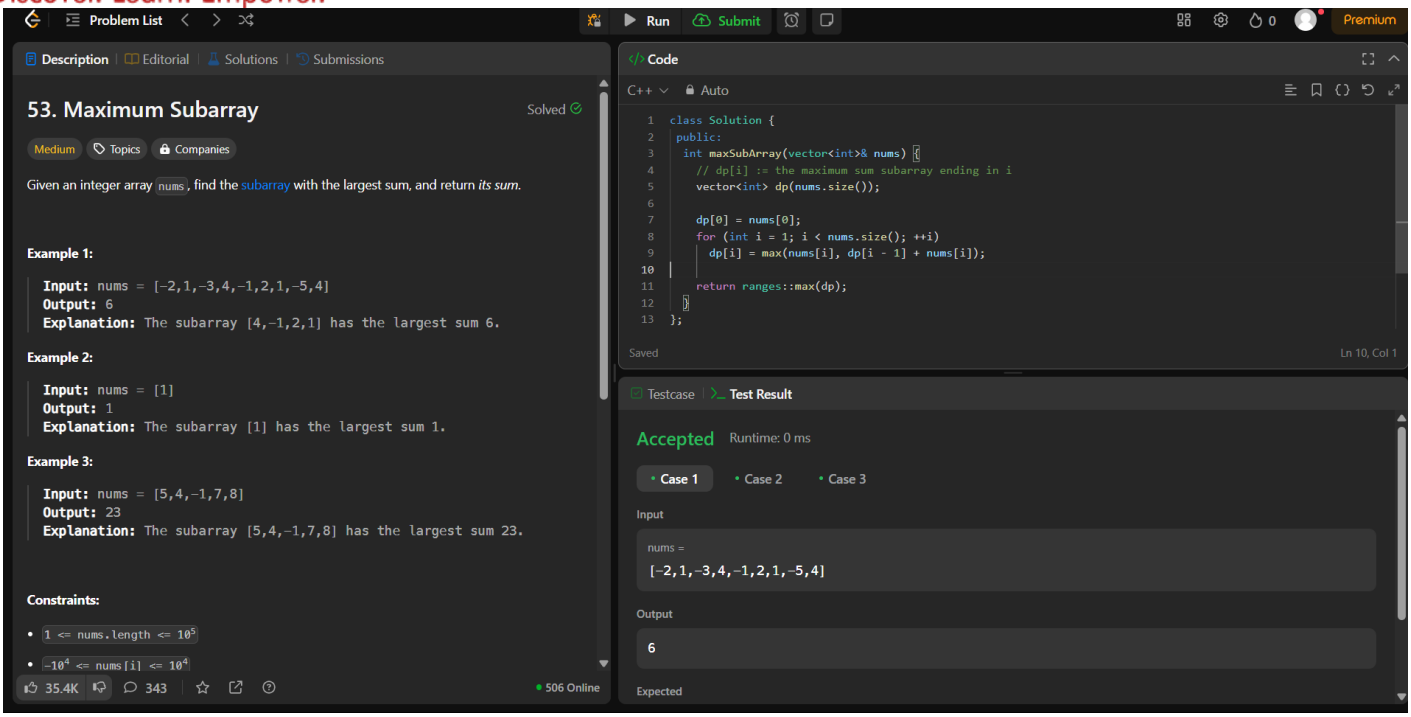
PROBLEM 2:

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

Code:

```
class Solution {
 public:
  int maxSubArray(vector<int>& nums) {
    // dp[i] := the maximum sum subarray ending in i
    vector<int> dp(nums.size());

    dp[0] = nums[0];
    for (int i = 1; i < nums.size(); ++i)
      dp[i] = max(nums[i], dp[i - 1] + nums[i]);

    return ranges::max(dp);
  }
};
```

Output:



PROBLEM 3:

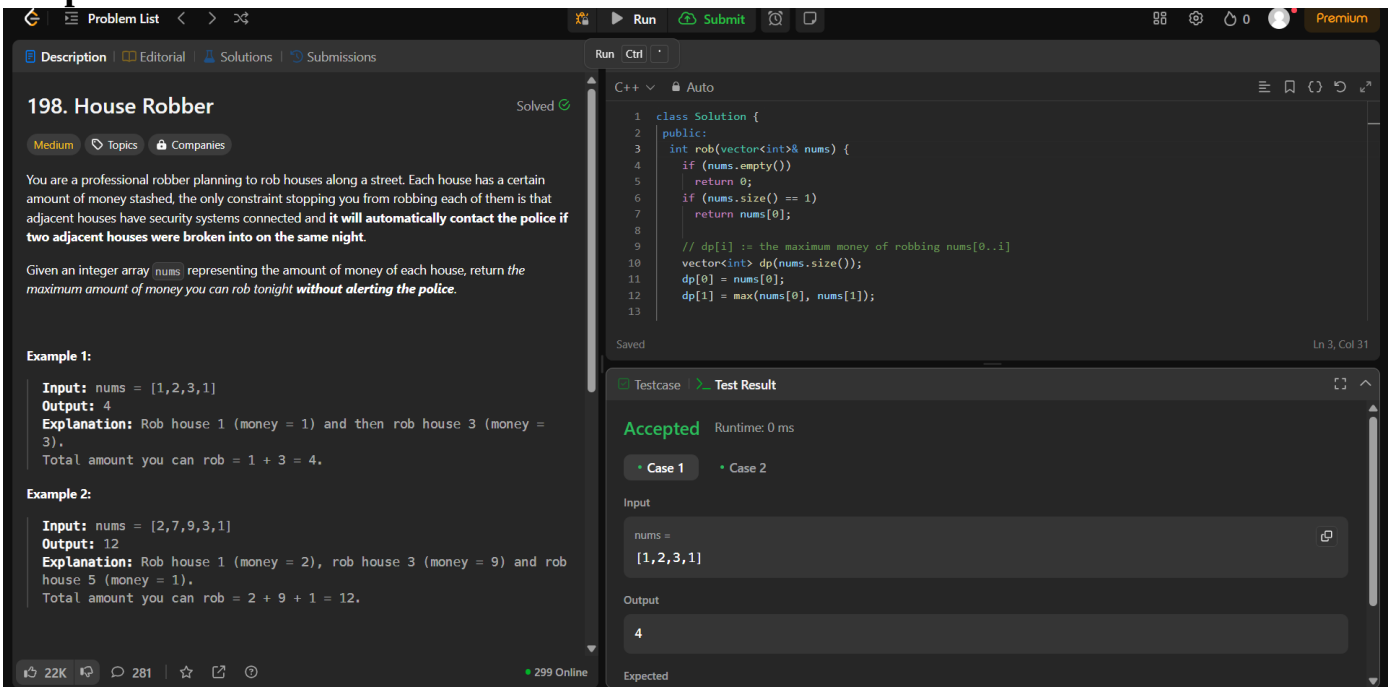
Aim: You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent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

Code:

```
class Solution {

 public:

  int rob(vector<int>& nums) {

    if (nums.empty())

      return 0;

    if (nums.size() == 1)

      return nums[0];



    // dp[i] := the maximum money of robbing nums[0..i]
```

```
    vector<int> dp(nums.size());

    dp[0] = nums[0];

    dp[1] = max(nums[0], nums[1]);


    for (int i = 2; i < nums.size(); ++i)

      dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]);


    return dp.back();

  }

};
```

Output:



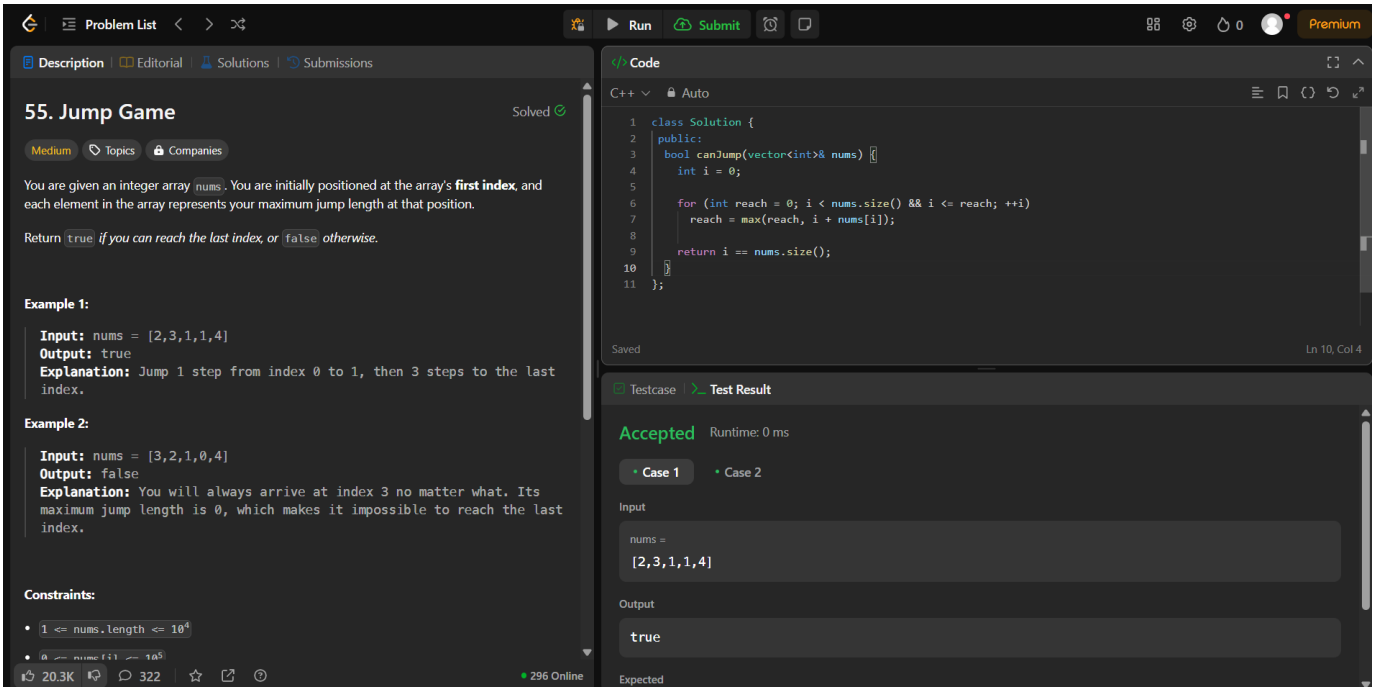
PROBLEM 4:

Aim: 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.
Return true *if you can reach the last index, or* false *otherwise*.

Code:

```
class Solution {
 public:
  bool canJump(vector<int>& nums) {
    int i = 0;

    for (int reach = 0; i < nums.size() && i <= reach; ++i)
      reach = max(reach, i + nums[i]);

    return i == nums.size();
  }
};
```

OUTPUT:



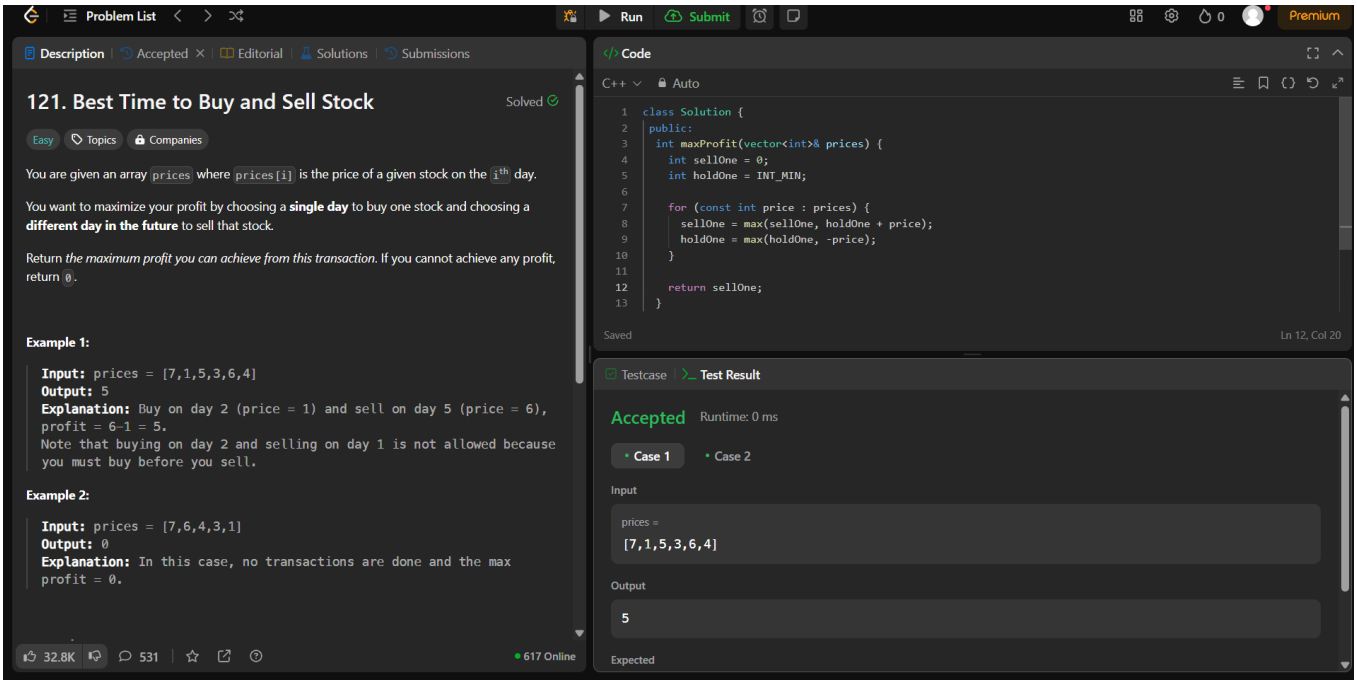
PROBLEM 5:

Aim: You are given an array prices where prices[i] is the price of a given stock on the i^{th} day.
You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.
Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Code:

```
class Solution {
 public:
  int maxProfit(vector<int>& prices) {
    int sellOne = 0;
    int holdOne = INT_MIN;

    for (const int price : prices) {
      sellOne = max(sellOne, holdOne + price);
      holdOne = max(holdOne, -price);
    }

    return sellOne;
  }
};
```

Output:



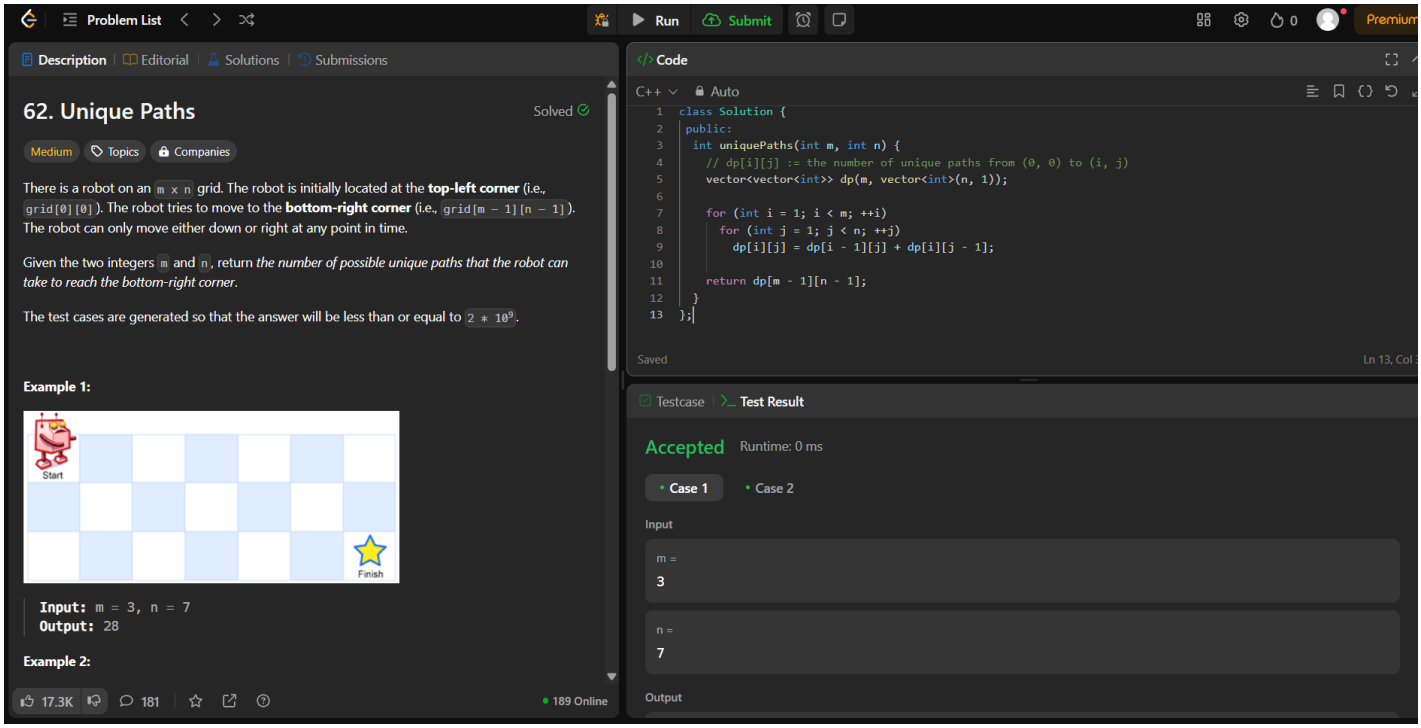
PROBLEM 6:

Aim: There is a robot on an m x n grid. The robot is initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.
Given the two integers m and n, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.
The test cases are generated so that the answer will be less than or equal to $2 * 10^9$.

Code:

```
class Solution {
 public:
  int uniquePaths(int m, int n) {
    // dp[i][j] := the number of unique paths from (0, 0) to (i, j)
    vector<vector<int>> dp(m, vector<int>(n, 1));

    for (int i = 1; i < m; ++i)
      for (int j = 1; j < n; ++j)
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];

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

Output:



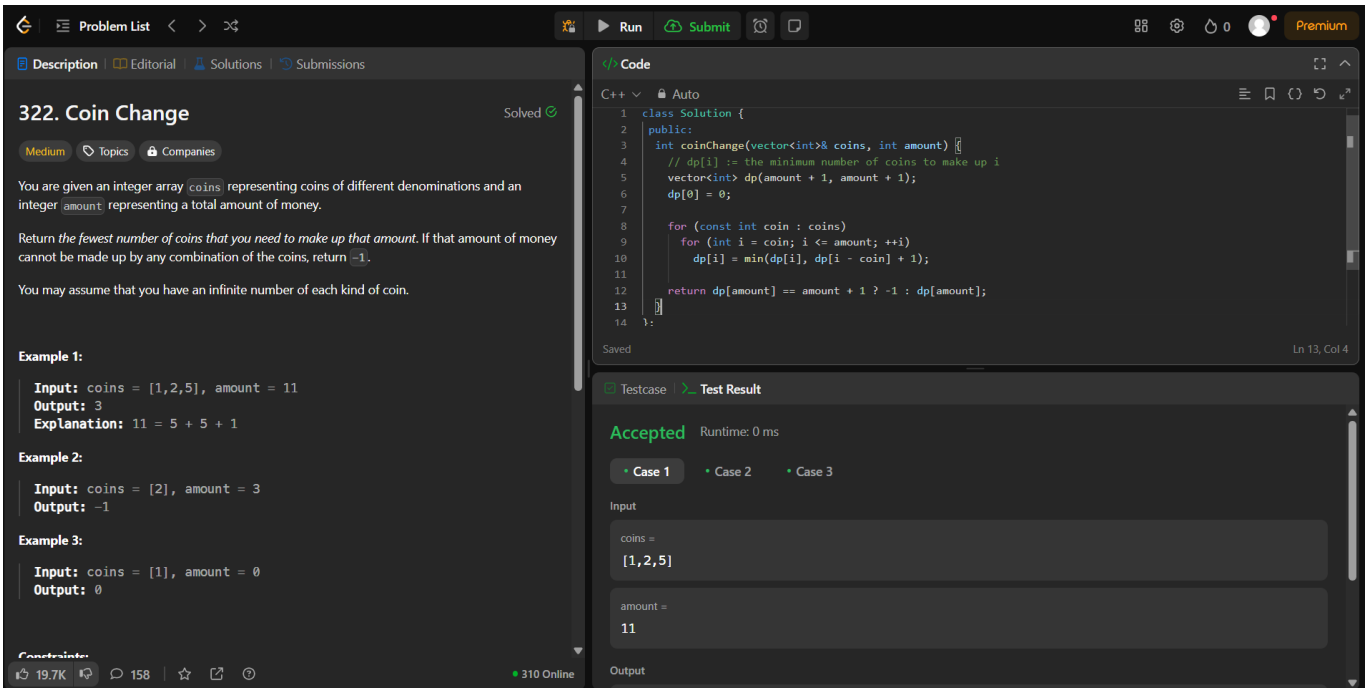
PROBLEM 7:

Aim: 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.
You may assume that you have an infinite number of each kind of coin.

Code:

```
class Solution {
 public:
  int coinChange(vector<int>& coins, int amount) {
    // dp[i] := the minimum number of coins to make up i
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;

    for (const int coin : coins)
      for (int i = coin; i <= amount; ++i)
        dp[i] = min(dp[i], dp[i - coin] + 1);

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

Output:



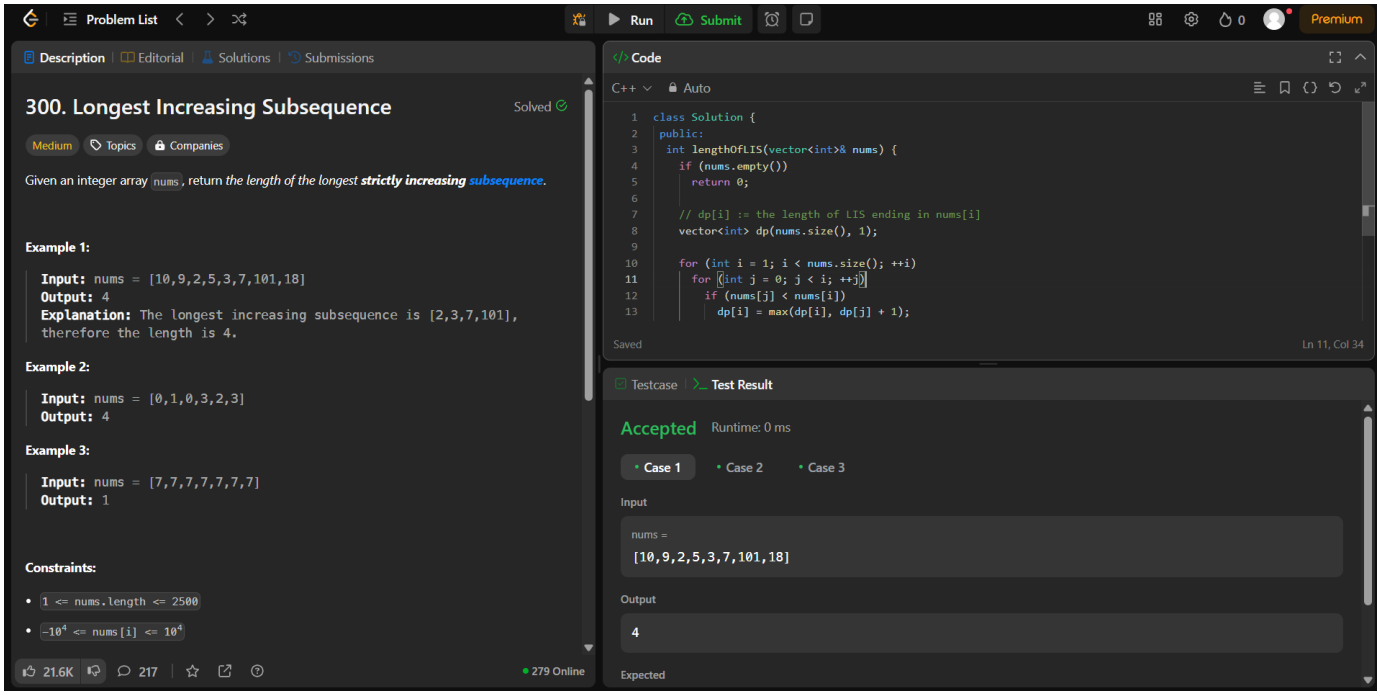
PROBLEM 8:

Aim: Given an integer array nums, return *the length of the longest* ***strictly increasing subsequence***.

Code:

```
class Solution {
 public:
  int lengthOfLIS(vector<int>& nums) {
    if (nums.empty())
      return 0;

    // dp[i] := the length of LIS ending in nums[i]
    vector<int> dp(nums.size(), 1);

    for (int i = 1; i < nums.size(); ++i)
      for (int j = 0; j < i; ++j)
        if (nums[j] < nums[i])
          dp[i] = max(dp[i], dp[j] + 1);

    return ranges::max(dp);
  }
};
```

Output:



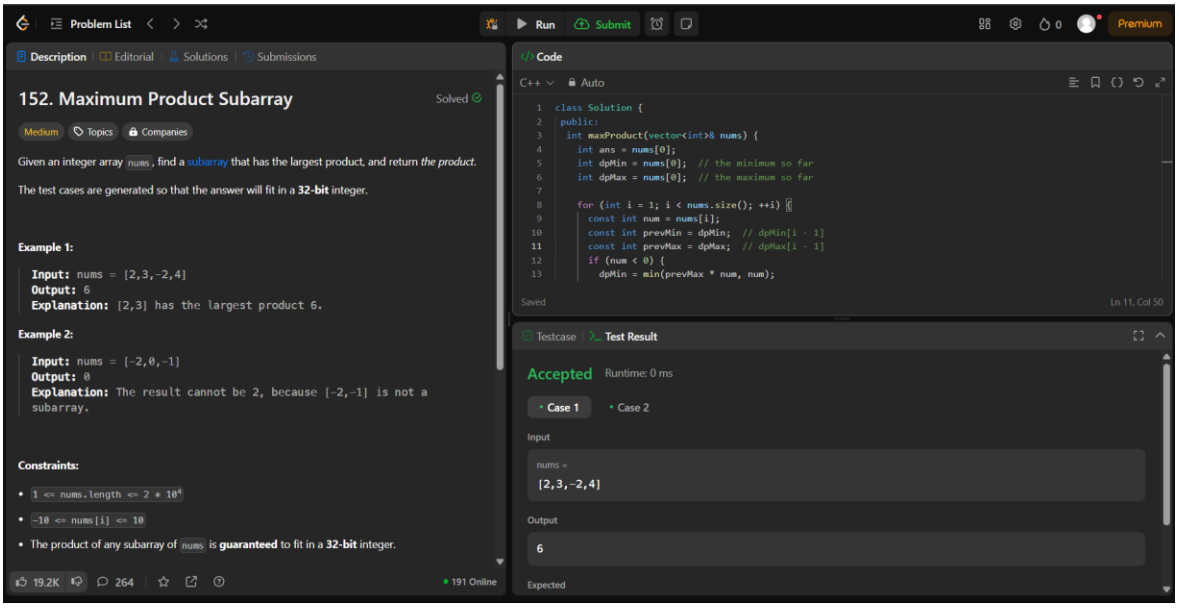
PROBLEM 9:

Aim: Given an integer array nums, find a subarray that has the largest product, and return *the product*. The test cases are generated so that the answer will fit in a **32-bit** integer.

Code:

```
class Solution {
 public:
  int maxProduct(vector<int>& nums) {
    int ans = nums[0];
    int dpMin = nums[0];  // the minimum so far
    int dpMax = nums[0];  // the maximum so far

    for (int i = 1; i < nums.size(); ++i) {
      const int num = nums[i];
      const int prevMin = dpMin;  // dpMin[i - 1]
      const int prevMax = dpMax;  // dpMax[i - 1]
      if (num < 0) {
        dpMin = min(prevMax * num, num);
        dpMax = max(prevMin * num, num);
      } else {
        dpMin = min(prevMin * num, num);
        dpMax = max(prevMax * num, num);
      }
      ans = max(ans, dpMax);
    }

    return ans;
  }
};
```

Output:



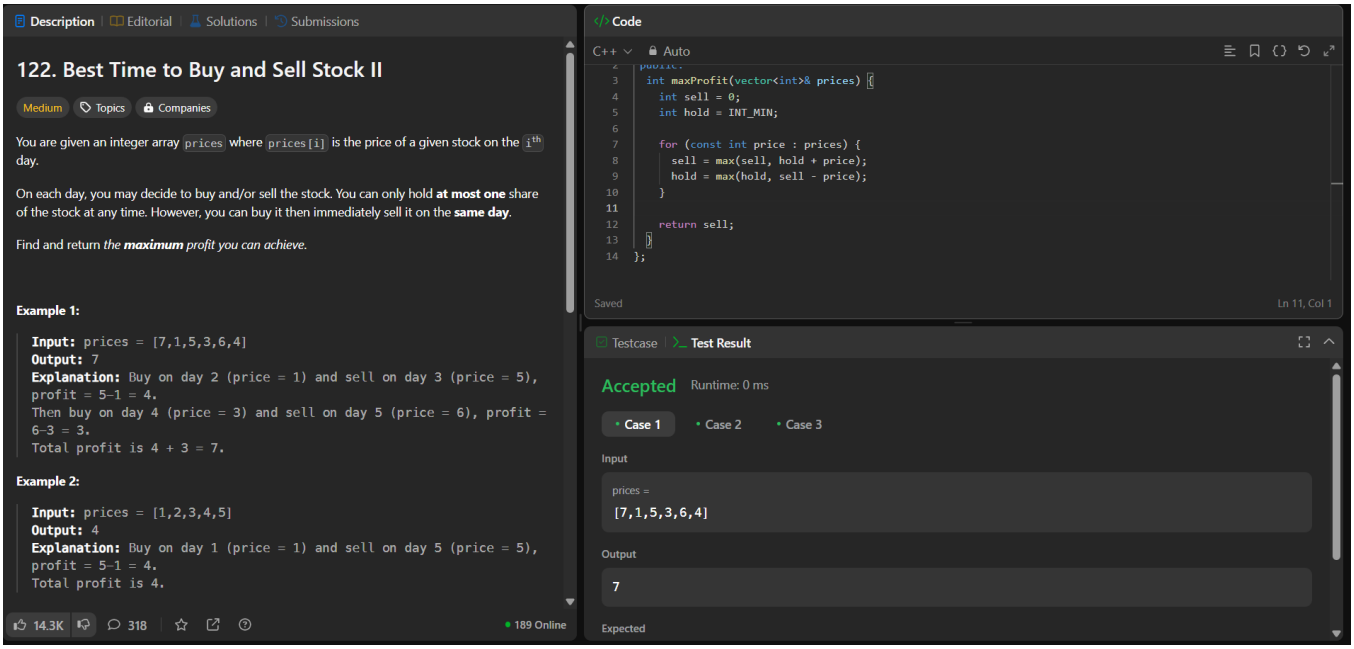
PROBLEM 10:

Aim: You are given an integer array prices where prices[i] is the price of a given stock on the i^{th} day.
On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.
Find and return *the **maximum** profit you can achieve.*

Code:

```
class Solution {
 public:
  int maxProfit(vector<int>& prices) {
    int sell = 0;
    int hold = INT_MIN;

    for (const int price : prices) {
      sell = max(sell, hold + price);
      hold = max(hold, sell - price);
    }

    return sell;
  }
};
```

OUTPUT:



PROBLEM 11:

Aim: Given an integer n, return *the least number of perfect square numbers that sum to* n.

A **perfect square** is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

Code:

```
class Solution {
 public:
  int numSquares(int n) {
    vector<int> dp(n + 1, n);  // 1^2 x n
    dp[0] = 0;                 // no way
    dp[1] = 1;                 // 1^2

    for (int i = 2; i <= n; ++i)
      for (int j = 1; j * j <= i; ++j)
        dp[i] = min(dp[i], dp[i - j * j] + 1);

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

OUTPUT: