Experiment 7

Student Name: Shubhang Deep Singh	**UID:22BET10325**
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?

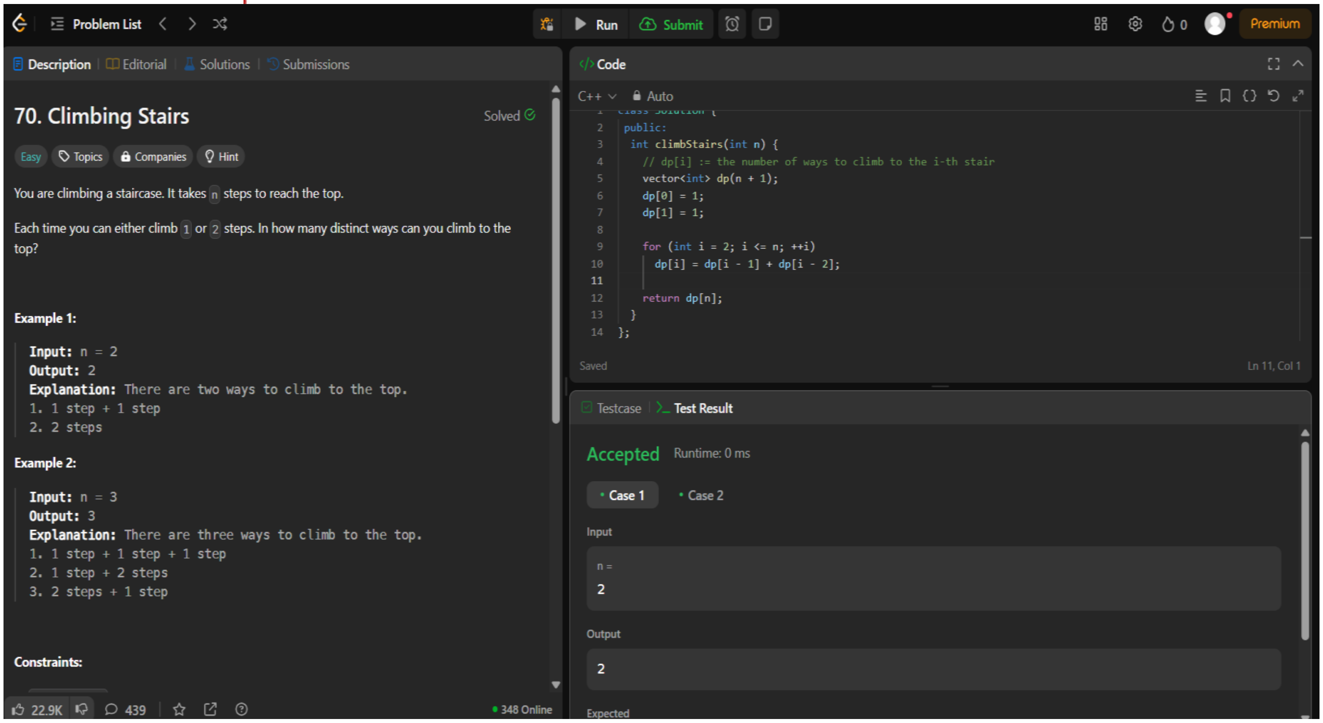
Code:

```
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:



70. Climbing Stairs

Solved ✓

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?

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Constraints:

```cpp
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];
  }
};
```

Saved

Ln 11, Col 1

Accepted Runtime: 0 ms

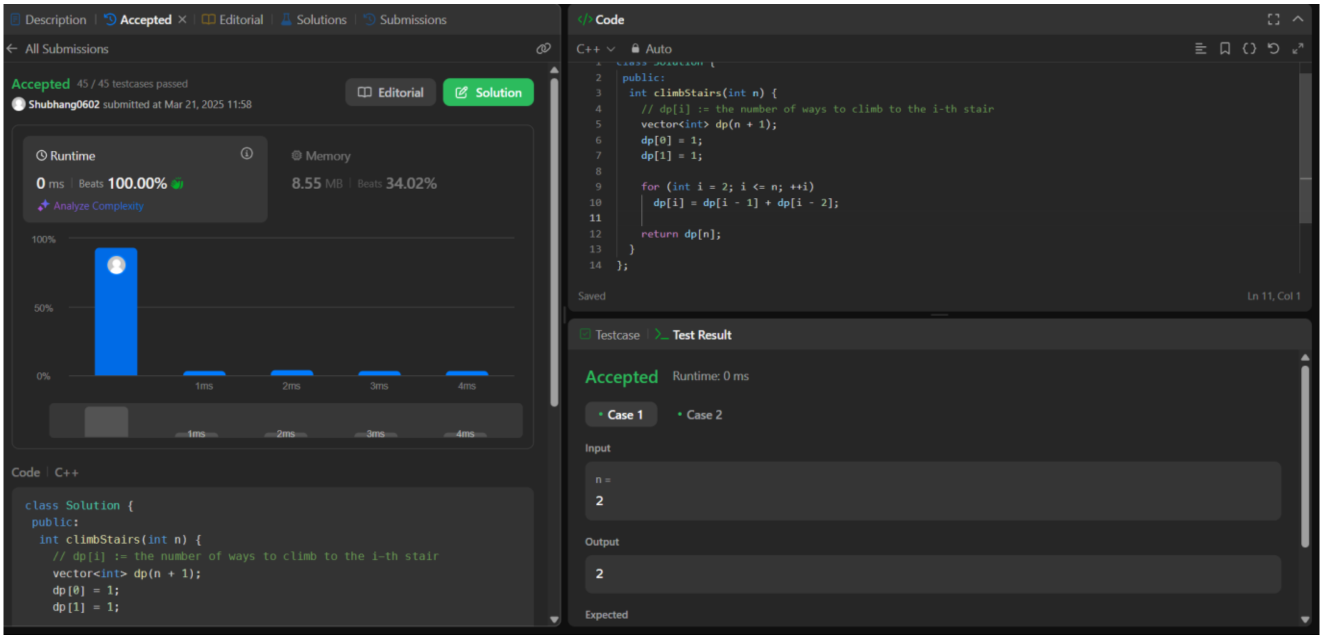
Input

n =
2

Output

2

Expected



Accepted 45 / 45 testcases passed

Shubhang0602 submitted at Mar 21, 2025 11:58

Runtime

0 ms | Beats 100.00%

Memory

8.55 MB | Beats 34.02%

Code | C++

```cpp
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;
```

```cpp
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];
  }
};
```

Saved

Ln 11, Col 1

Accepted Runtime: 0 ms

Input

n =
2

Output

2

Expected

PROBLEM 2:

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

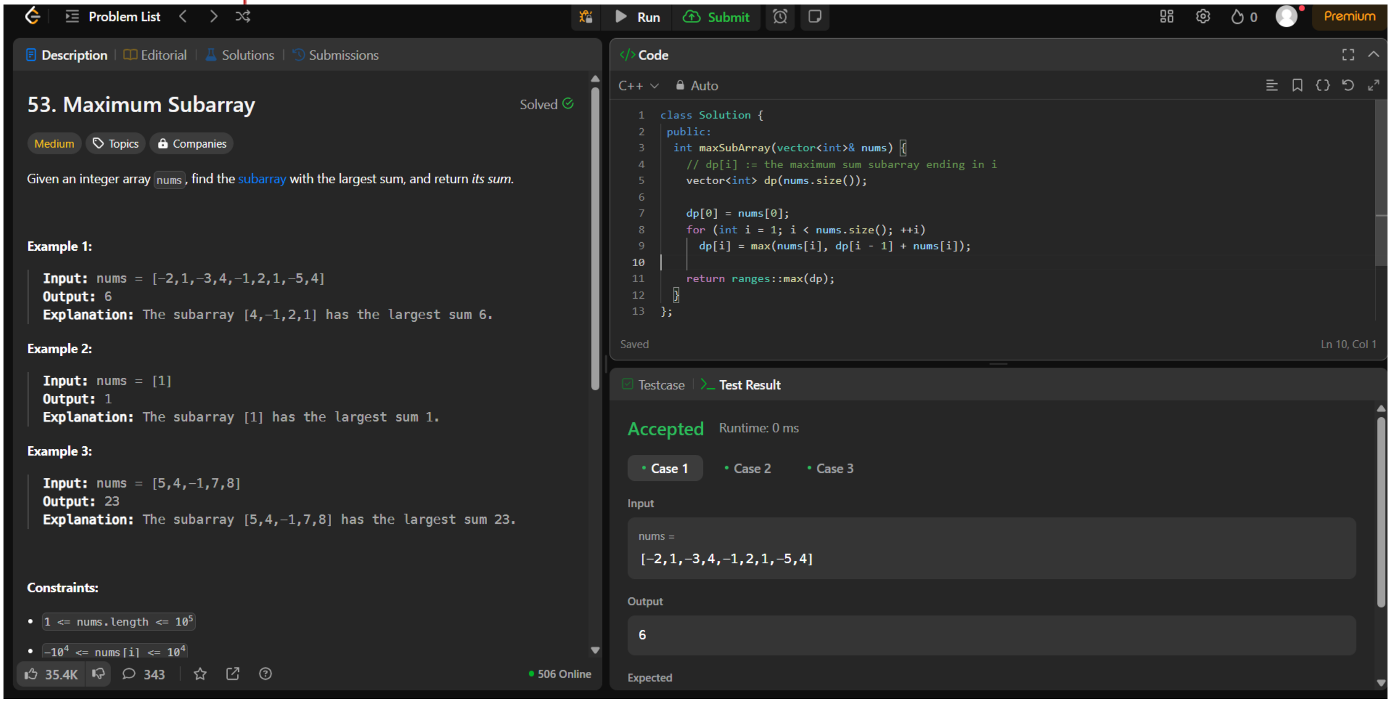
Code:

```
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:



53. Maximum Subarray

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

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
```

Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
```

Example 3:

```
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.
```

Constraints:

- $1 <= nums.length <= 10^5$
- $-10^4 <= nums[i] <= 10^4$

```cpp
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);
  }
};
```

Saved Ln 10, Col 1

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

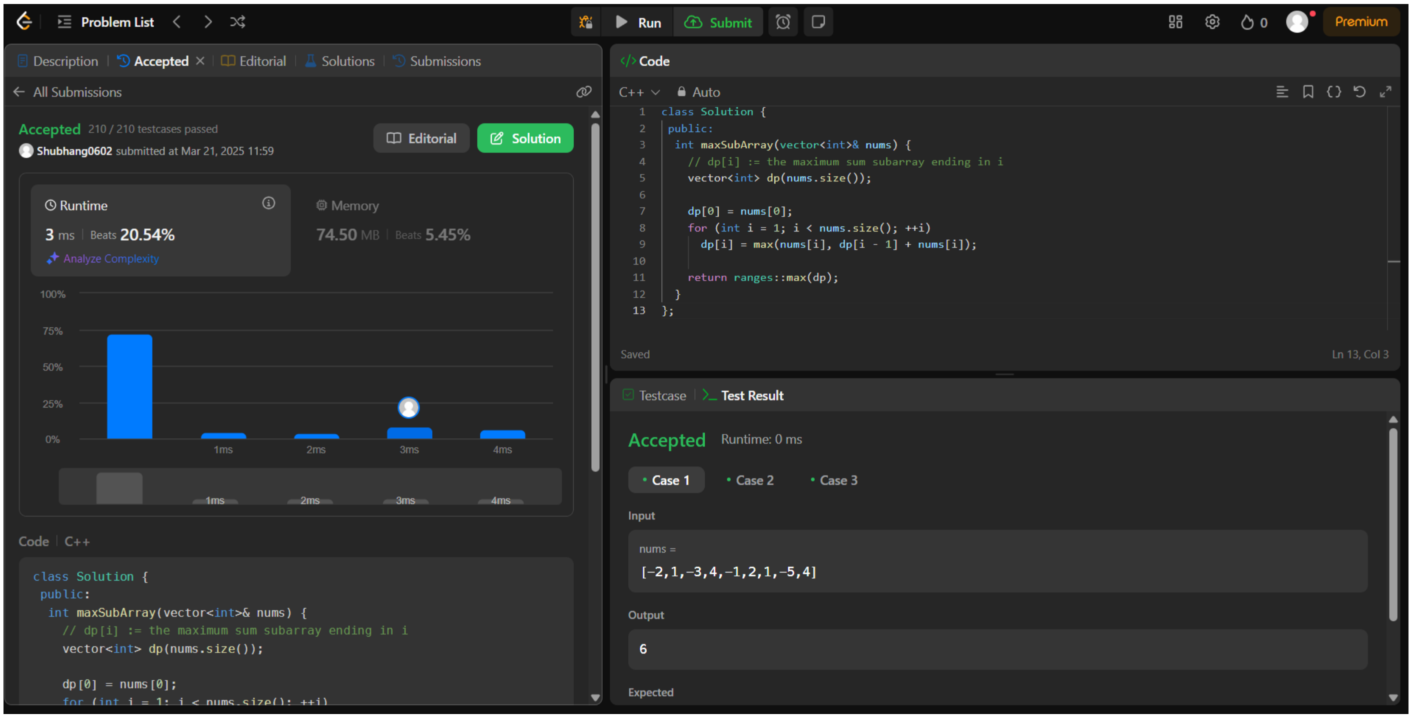
Input

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

Output

6

Expected



Accepted 210 / 210 testcases passed
Shubhang0602 submitted at Mar 21, 2025 11:59

Runtime 3 ms | Beats 20.54%

Memory 74.50 MB | Beats 5.45%

Code | C++

```cpp
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)
```

```cpp
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);
  }
};
```

Saved Ln 13, Col 3

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

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

Output

6

Expected

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:

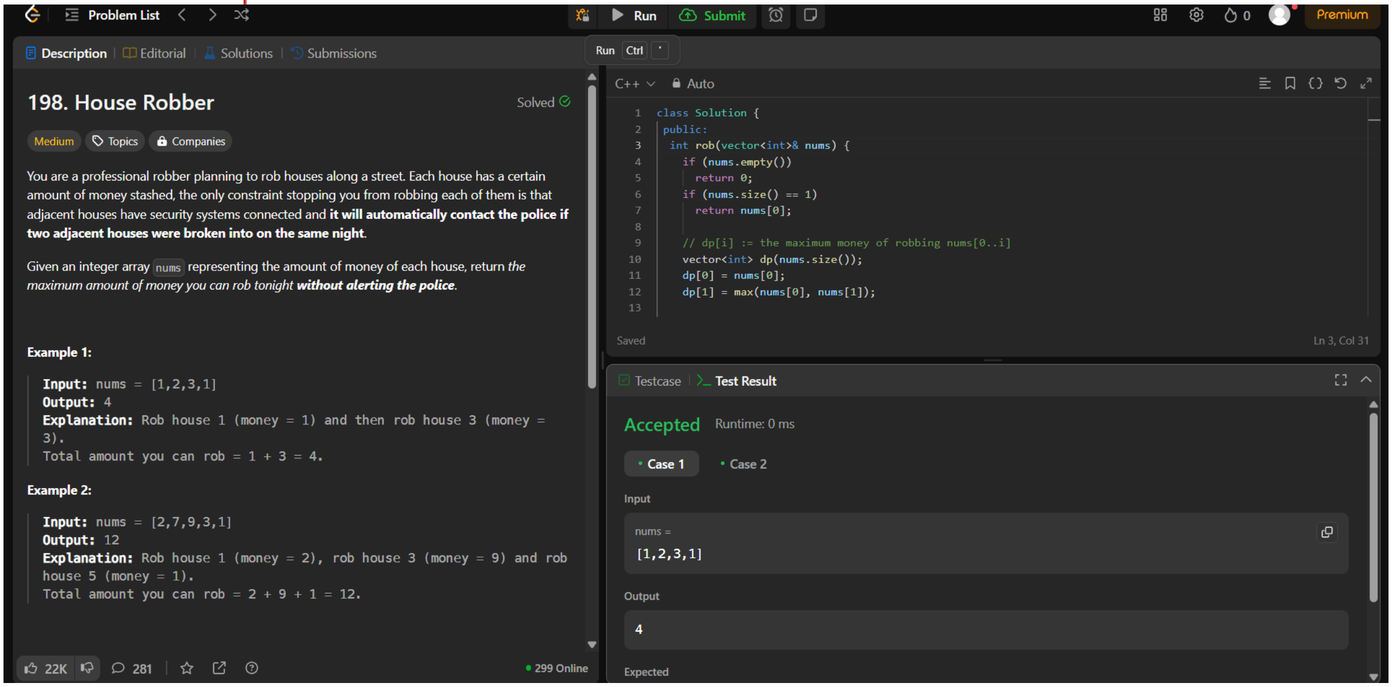
```
class Solution {
 public:
  int rob(vector<int>& nums) {
    if (nums.empty())
      return 0;
    if (nums.size() == 1)
      return nums[0];

    // 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:



198. House Robber

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***.

Example 1:

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money =
3).
Total amount you can rob = 1 + 3 = 4.
```

Example 2:

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob
house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.
```

```cpp
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]);

```

Saved Ln 3, Col 31

Accepted Runtime: 0 ms

Case 1 Case 2

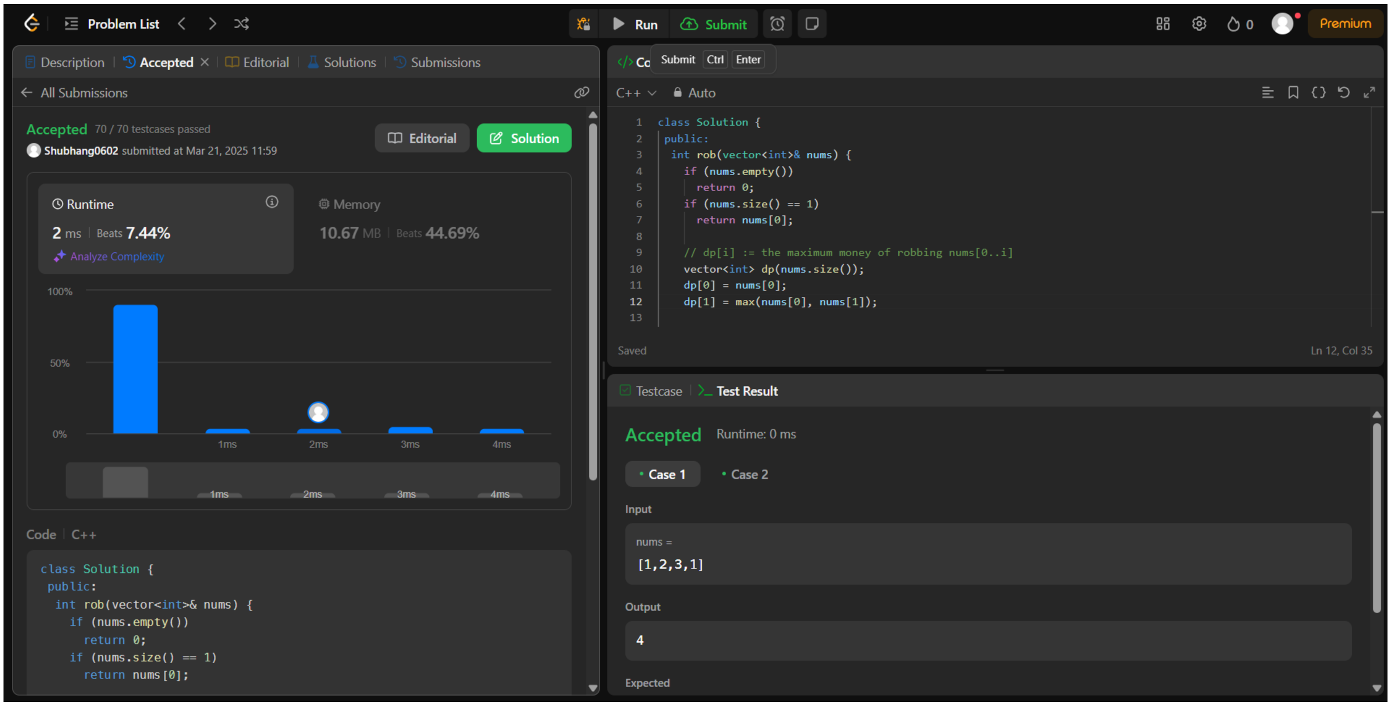
Input

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

Output

```
4
```

Expected



Accepted 70 / 70 testcases passed

Shubhang0602 submitted at Mar 21, 2025 11:59

Runtime 2 ms | Beats 7.44%

Memory 10.67 MB | Beats 44.69%

Code | C++

```cpp
class Solution {
 public:
  int rob(vector<int>& nums) {
    if (nums.empty())
      return 0;
    if (nums.size() == 1)
      return nums[0];
```

```cpp
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]);

```

Saved Ln 12, Col 35

Accepted Runtime: 0 ms

Case 1 Case 2

Input

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

Output

```
4
```

Expected

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*.

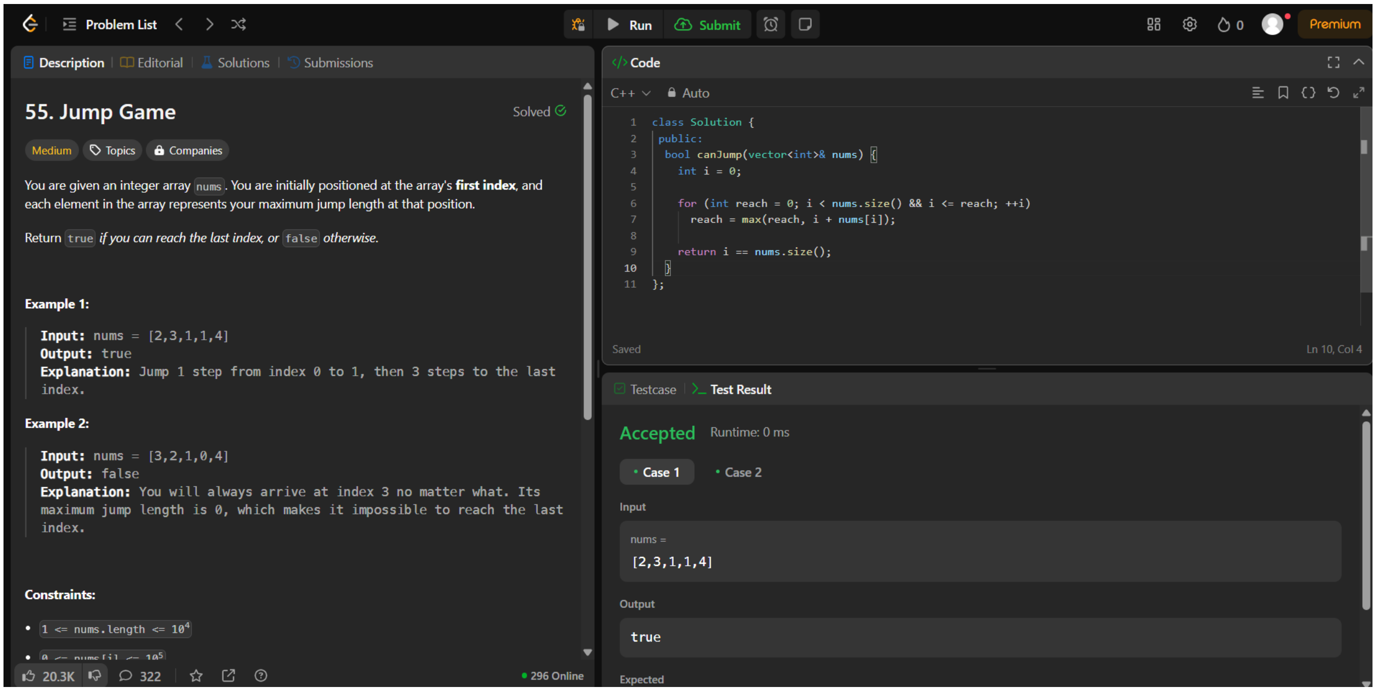
Code:

```
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:



55. 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.

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

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last
index.
```

Example 2:

```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its
maximum jump length is 0, which makes it impossible to reach the last
index.
```

Constraints:

- $1 <= nums.length <= 10^4$
- $0 <= nums[i] <= 10^5$

```
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();
  }
};
```

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

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

Output

true

Expected

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.

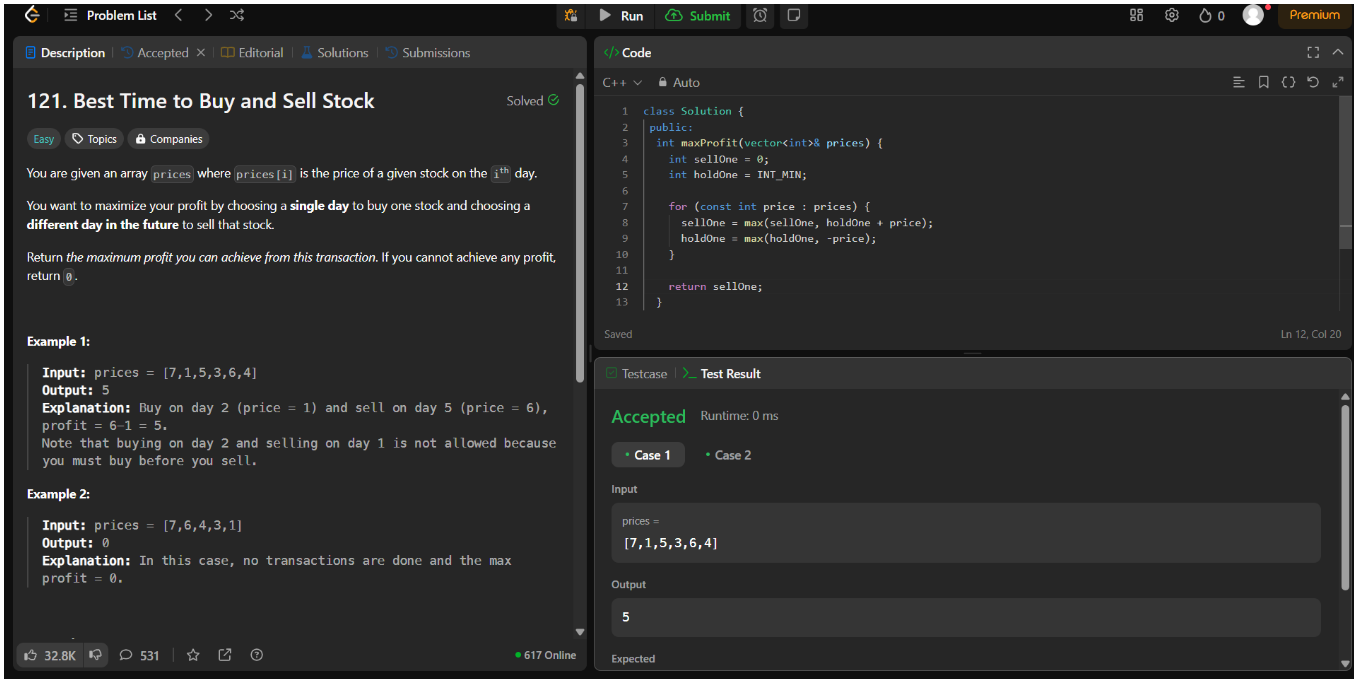
Code:

```
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:



121. Best Time to Buy and Sell Stock

Solved

Easy   Topics   Companies

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`.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6),
profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because
you must buy before you sell.
```

Example 2:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max
profit = 0.
```

```
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;
  }
```

Saved   Ln 12, Col 20

Accepted   Runtime: 0 ms

Case 1   Case 2

Input

prices =
[7,1,5,3,6,4]

Output

5

Expected

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.

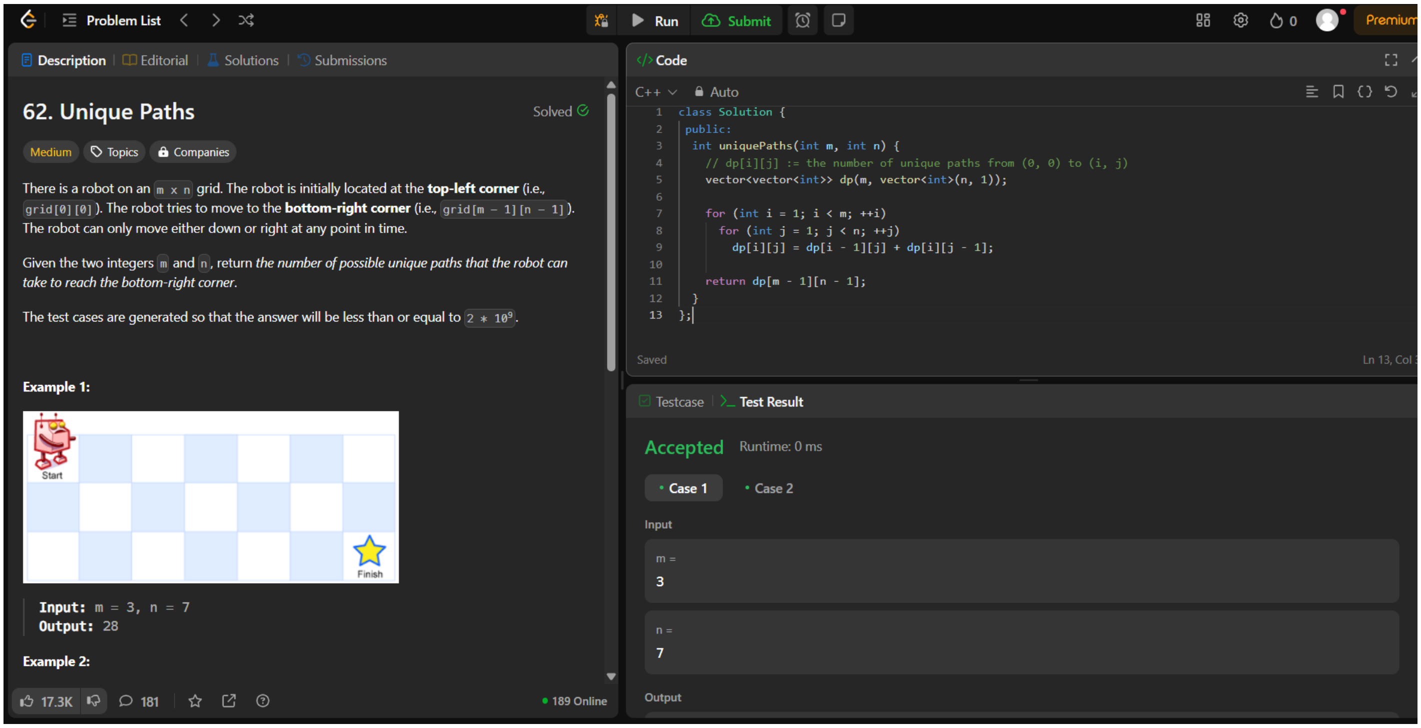
Code:

```
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:



62. Unique Paths

Solved

Medium | Topics | Companies

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`.

Example 1:

Start

Finish

```
Input: m = 3, n = 7
Output: 28
```

Example 2:

```
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];
  }
};
```

Saved

Ln 13, Col 3

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

m =
3

n =
7

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.

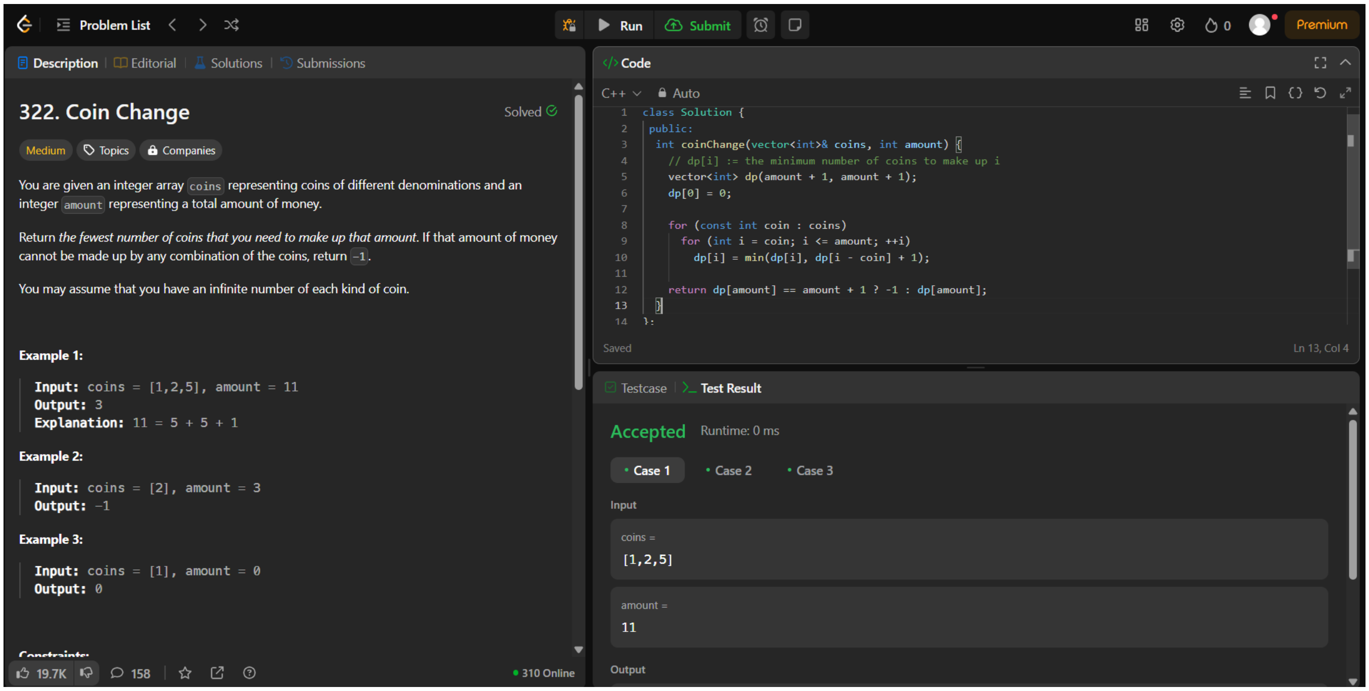
Code:

```
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:



322. Coin Change

Solved ✓

Medium · Topics · Companies

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.

Example 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```

Example 3:

```
Input: coins = [1], amount = 0
Output: 0
```

Constraints:

```
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];
  }
};
```

Saved   Ln 13, Col 4

Accepted   Runtime: 0 ms

• Case 1   • Case 2   • Case 3

Input

coins =
[1,2,5]

amount =
11

Output

PROBLEM 8:

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

Code:

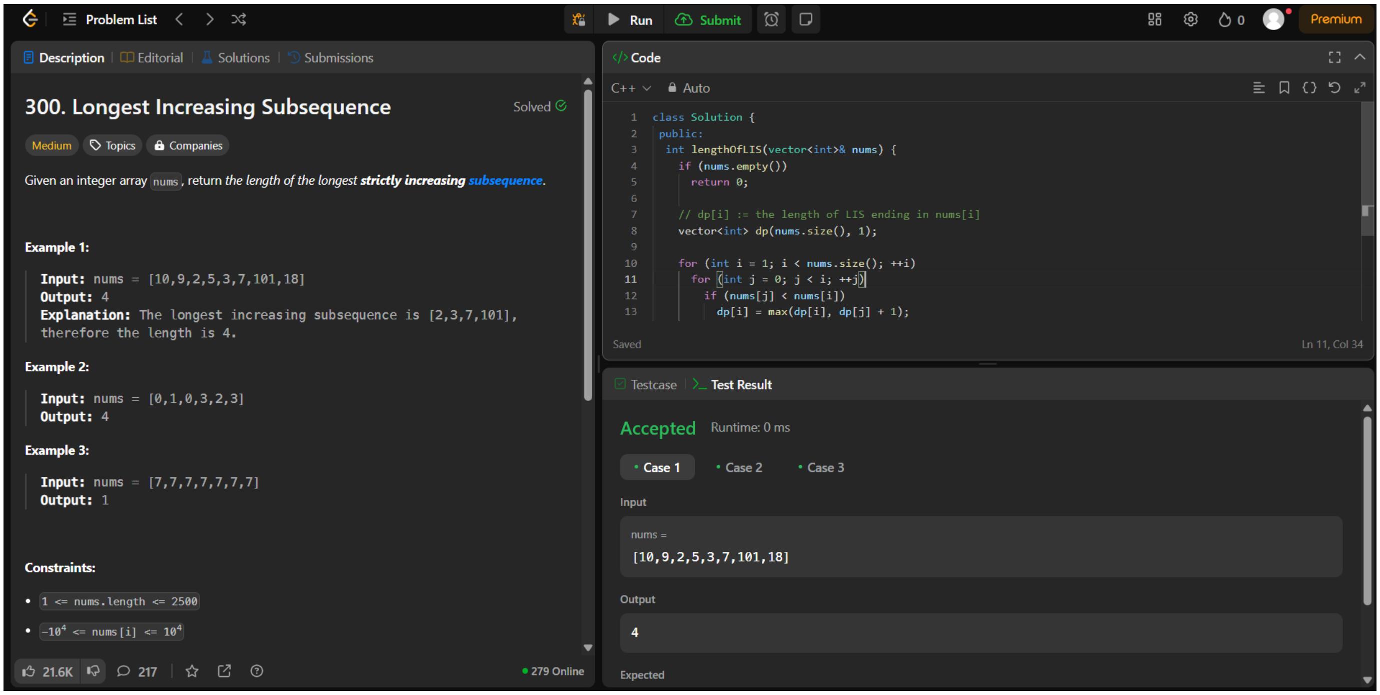
```
class Solution {
 public:
  int lengthOfLIS(vector<int>& nums) {
    if (nums.empty())
      return 0;

    // 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:



300. Longest Increasing Subsequence

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

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101],
therefore the length is 4.
```

Example 2:

```
Input: nums = [0,1,0,3,2,3]
Output: 4
```

Example 3:

```
Input: nums = [7,7,7,7,7,7,7]
Output: 1
```

Constraints:

- `1 <= nums.length <= 2500`
- -10^4 `<= nums[i] <=` 10^4

```
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);
```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
nums =
[10,9,2,5,3,7,101,18]
```

Output

```
4
```

Expected

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.

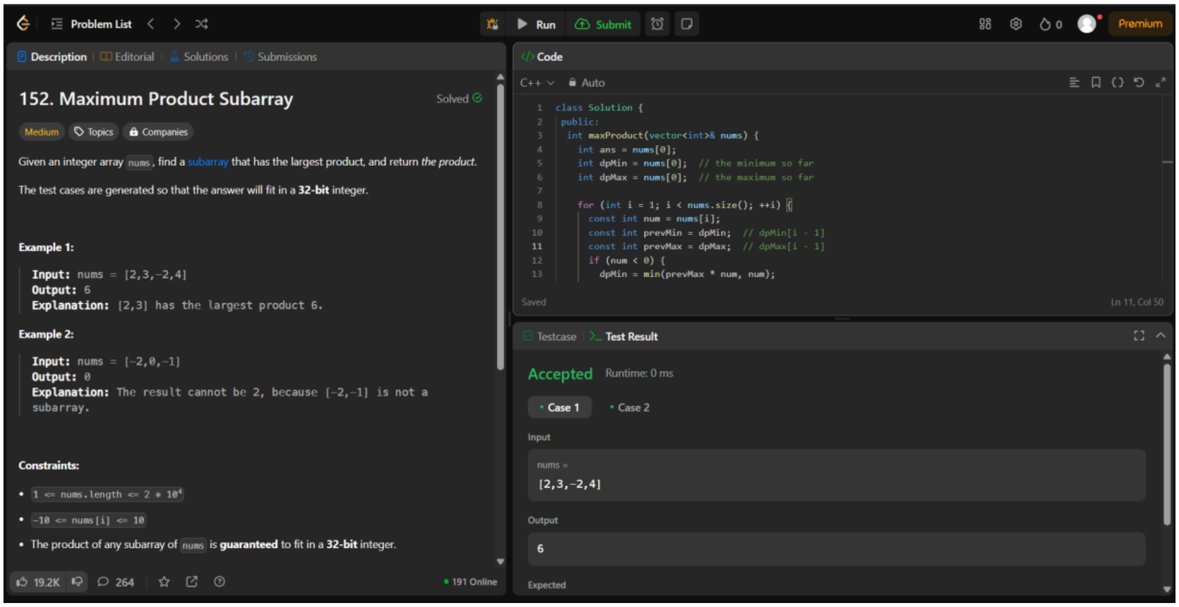
Code:

```
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:



152. Maximum Product Subarray

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.

Example 1:

```
Input: nums = [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
```

Example 2:

```
Input: nums = [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a
subarray.
```

Constraints:

- `1 <= nums.length <= 2 * 10`^{4}
- `-10 <= nums[i] <= 10`
- The product of any subarray of `nums` is **guaranteed** to fit in a **32-bit** integer.

```
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);
```

Saved Ln 11, Col 50

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums =
[2,3,-2,4]
```

Output

```
6
```

Expected

PROBLEM 10:

Aim: You are given an integer array prices where prices[i] is the price of a given stock on the ith 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*.

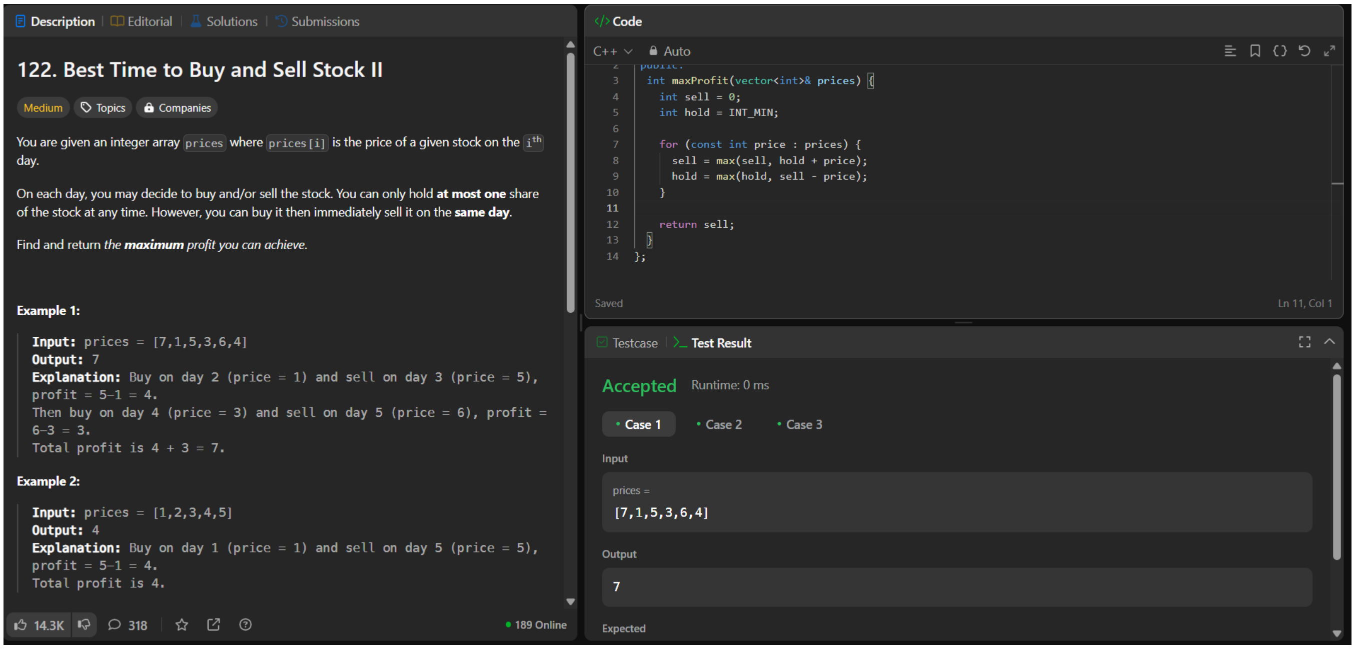
Code:

```
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:



122. Best Time to Buy and Sell Stock II

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*.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5),
profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit =
6-3 = 3.
Total profit is 4 + 3 = 7.
```

Example 2:

```
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5),
profit = 5-1 = 4.
Total profit is 4.
```

```
 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;
  }
};
```

Accepted Runtime: 0 ms

Input

```
prices =
[7,1,5,3,6,4]
```

Output

```
7
```

Expected

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.

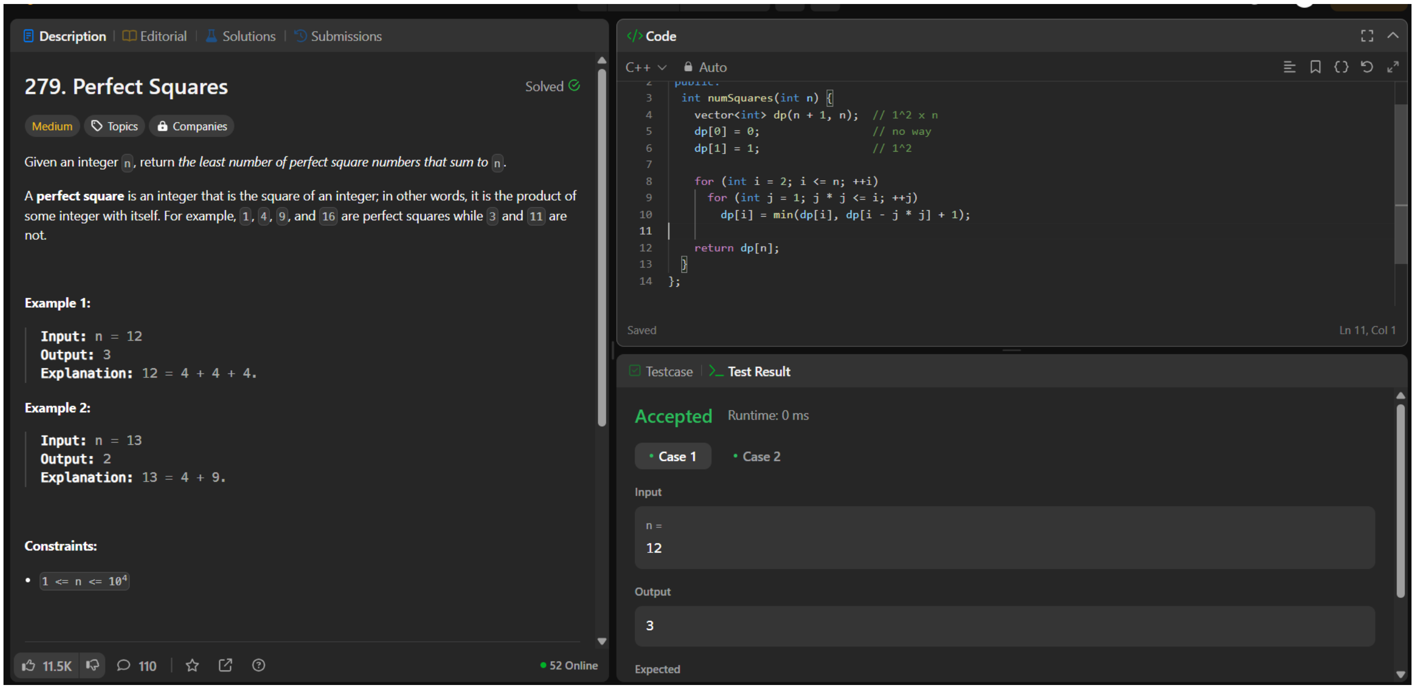
Code:

```
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:



279. Perfect Squares

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.

Example 1:

```
Input: n = 12
Output: 3
Explanation: 12 = 4 + 4 + 4.
```

Example 2:

```
Input: n = 13
Output: 2
Explanation: 13 = 4 + 9.
```

Constraints:

- `1 <= n <= 10`^{4}

```
 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];
  }
};
```

Accepted Runtime: 0 ms

Input

```
n =
12
```

Output

```
3
```

Expected