



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment 7

Student Name: Payal Singh

Branch: BE-IT

Semester: 6

Subject Name: AP LAB-II

UID:22BET10347

Section/Group: IOT-702(A)

Date of Performance:20/03/25

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

Discover. Learn. Empower.

Problem List

Description

Editorial

Solutions

Submissions

70. Climbing Stairs

EasyTopicsCompaniesHint

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:

348 Online

RunSubmit

C++Auto

```
1 public:
2     int climbStairs(int n) {
3         // dp[i] := the number of ways to climb to the i-th stair
4         vector<int> dp(n + 1);
5         dp[0] = 1;
6         dp[1] = 1;
7
8         for (int i = 2; i <= n; ++i)
9             dp[i] = dp[i - 1] + dp[i - 2];
10
11         return dp[n];
12     }
13 }
14 ;
```

SavedLn 11, Col 1

TestcaseTest Result

AcceptedRuntime: 0 ms

Case 1Case 2

Input

n =
2

Output

2

Expected



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

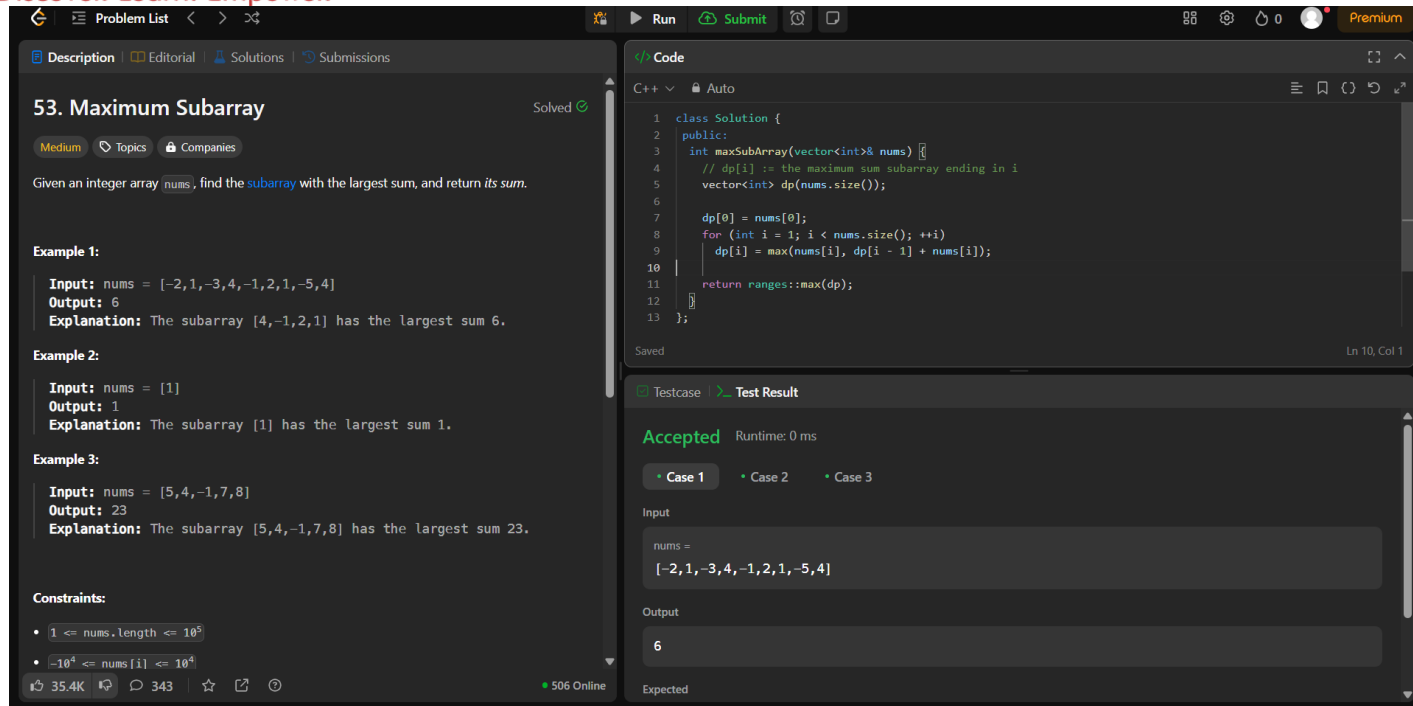
PROBLEM 2:

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:



53. Maximum Subarray Solved

Medium Topics Companies

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 <= 105`
- `-104 <= nums[i] <= 104`

35.4K 343 506 Online

```

1 class Solution {
2 public:
3     int maxSubArray(vector<int>& nums) {
4         // dp[i] := the maximum sum subarray ending in i
5         vector<int> dp(nums.size());
6
7         dp[0] = nums[0];
8         for (int i = 1; i < nums.size(); ++i)
9             dp[i] = max(nums[i], dp[i - 1] + nums[i]);
10
11         return ranges::max(dp);
12     }
13 };

```

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

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]
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
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 Solved

Medium Topics Companies

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.

```
1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         if (nums.empty())
5             return 0;
6         if (nums.size() == 1)
7             return nums[0];
8
9         // dp[i] := the maximum money of robbing nums[0..i]
10        vector<int> dp(nums.size());
11        dp[0] = nums[0];
12        dp[1] = max(nums[0], nums[1]);
13    }
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

nums =

[1,2,3,1]

Output

4

Expected

PROBLEM 4:

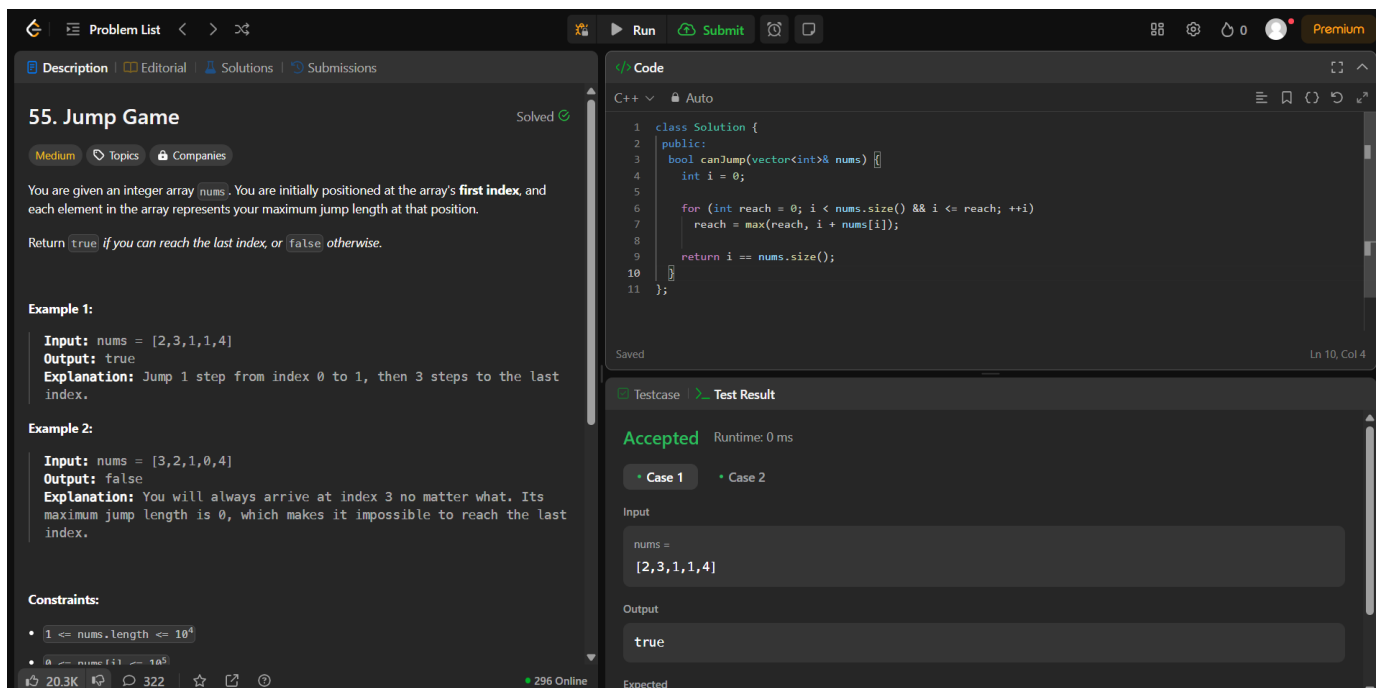
Aim: You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

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:



The screenshot displays a coding interface for the '55. Jump Game' problem. The problem description states: '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:
• $1 \leq \text{nums.length} \leq 10^4$
• $0 \leq \text{nums}[i] \leq 10^5$

The solution code is shown in the 'Code' editor, which matches the code provided in the previous block. The 'Test Result' section shows 'Accepted' with a runtime of 0 ms. The input is `nums = [2,3,1,1,4]` and the output is `true`.

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.

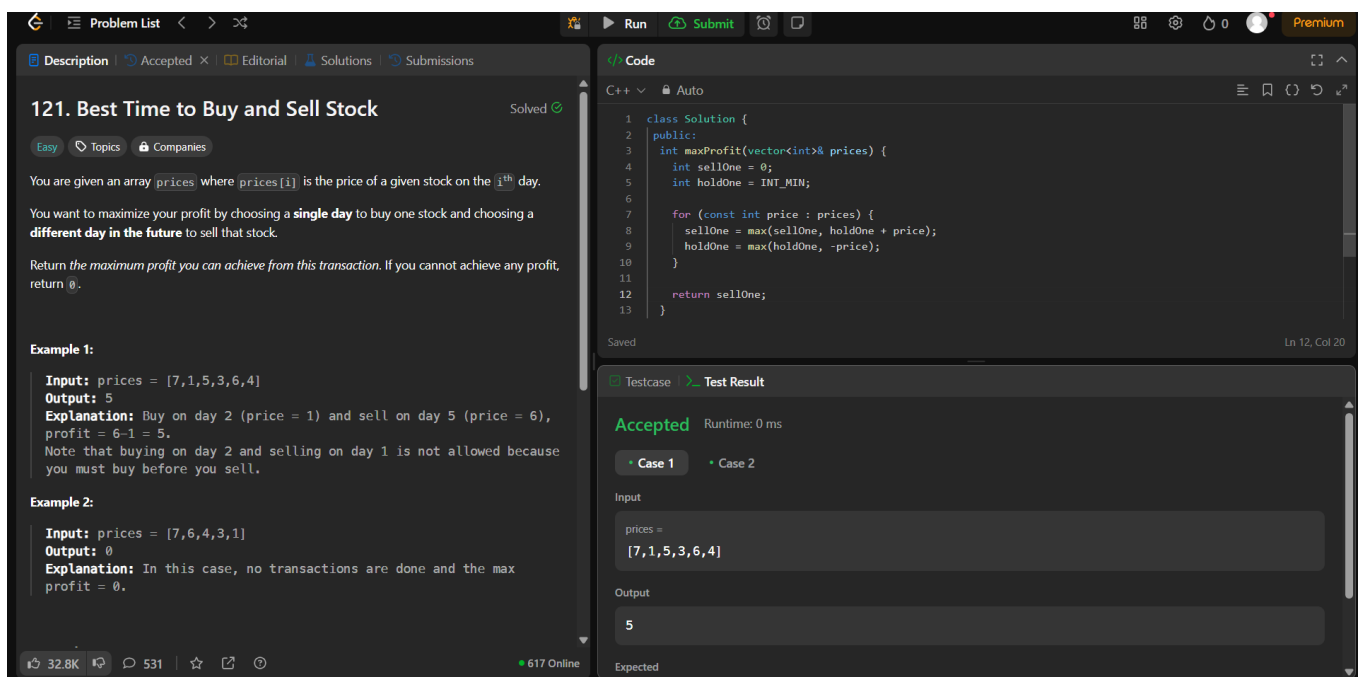
Code:

```
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:



The screenshot displays a coding interface for the problem "121. Best Time to Buy and Sell Stock". The left panel shows the problem description, which states: "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." It includes two examples: Example 1 with input [7,1,5,3,6,4] and output 5, and Example 2 with input [7,6,4,3,1] and output 0. The right panel shows the C++ code for the solution, which is the same as the code provided in the text. Below the code, the test results are shown as "Accepted" with a runtime of 0 ms. The input field contains the array [7,1,5,3,6,4] and the output field contains the value 5.

PROBLEM 6:

Aim: There is a robot on an $m \times n$ grid. The robot is initially located at the **top-left corner** (i.e., $\text{grid}[0][0]$). The robot tries to move to the **bottom-right corner** (i.e., $\text{grid}[m - 1][n - 1]$). 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 $2 * 10^9$.

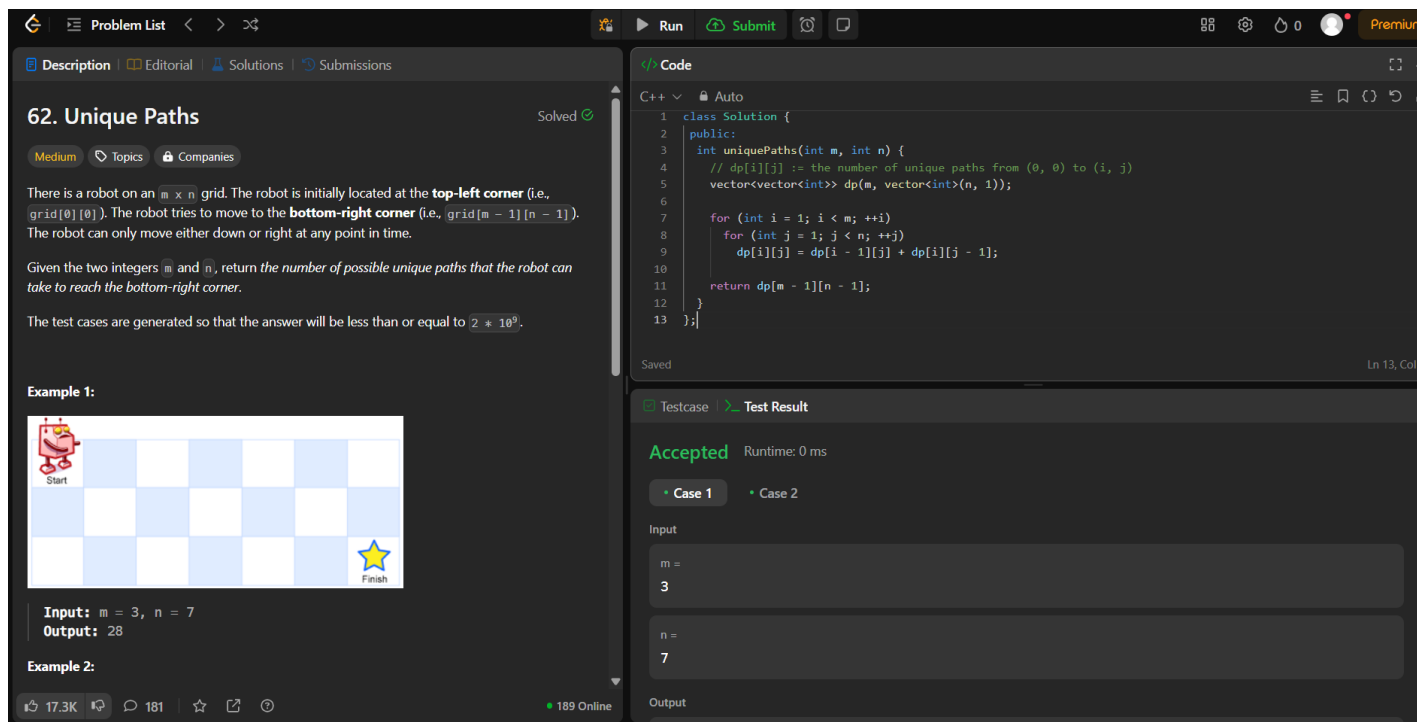
Code:

```
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

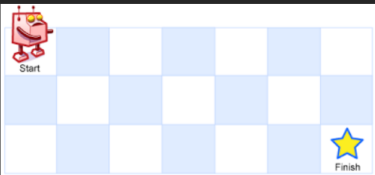
Medium Topics Companies

There is a robot on an $m \times n$ grid. The robot is initially located at the **top-left corner** (i.e., $\text{grid}[0][0]$). The robot tries to move to the **bottom-right corner** (i.e., $\text{grid}[m - 1][n - 1]$). 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:



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

Example 2:

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];
    }
};
```

Testcase Test Result

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.

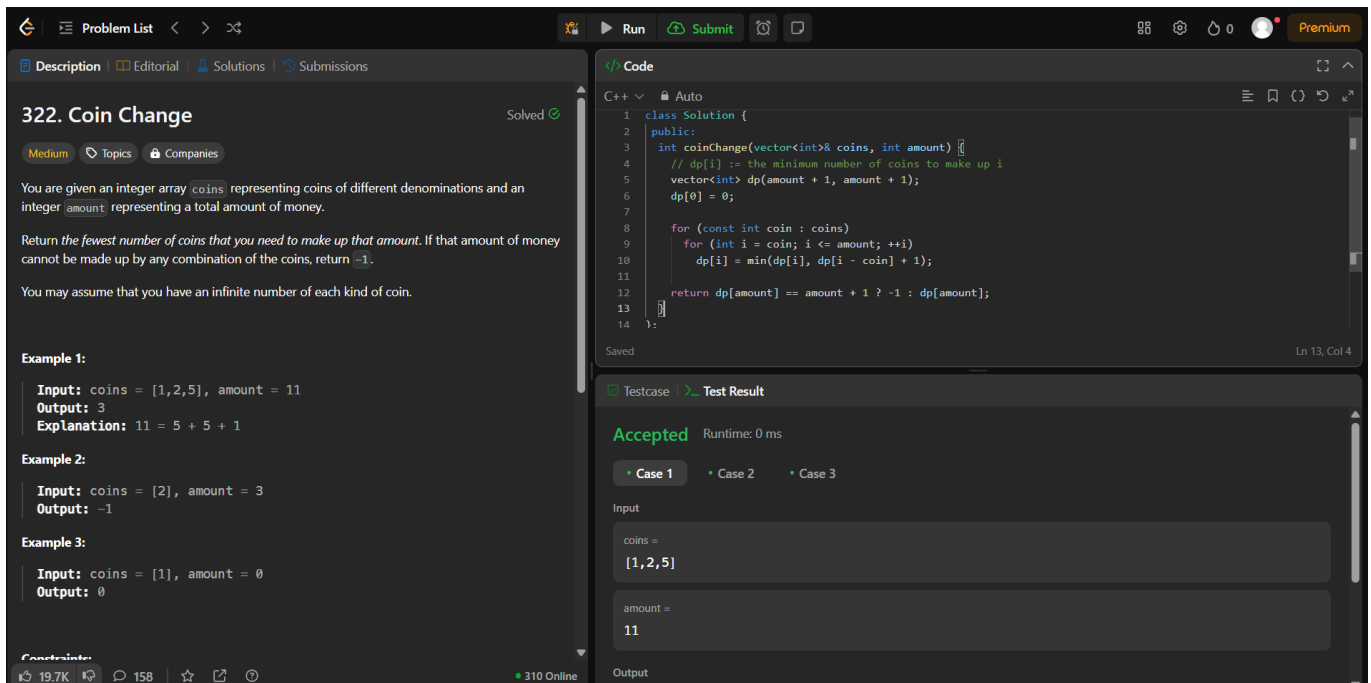
Code:

```
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:



The screenshot displays a coding interface for the '322. Coin Change' problem. The left panel shows the problem description, which asks for the fewest number of coins to make up a given amount. The right panel shows the C++ code implementing a dynamic programming solution. The code uses a 1D array `dp` to store the minimum number of coins for each amount up to `amount`. The initial value for `dp[0]` is 0, and for other amounts, it is initialized to `amount + 1`. The code iterates over each coin and updates the `dp` array. The final result is `dp[amount]`, which is -1 if the amount cannot be made up.

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

Code

```
1 class Solution {
2 public:
3     int coinChange(vector<int>& coins, int amount) {
4         // dp[i] := the minimum number of coins to make up i
5         vector<int> dp(amount + 1, amount + 1);
6         dp[0] = 0;
7
8         for (const int coin : coins)
9             for (int i = coin; i <= amount; ++i)
10                 dp[i] = min(dp[i], dp[i - coin] + 1);
11
12         return dp[amount] == amount + 1 ? -1 : dp[amount];
13     }
14 }
```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

coins =
[1,2,5]

amount =
11

Output



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

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:

The screenshot displays a coding interface for the problem "300. Longest Increasing Subsequence". The problem description states: "Given an integer array `nums`, return the length of the longest *strictly increasing subsequence*." It includes three examples: Example 1 with input `[10,9,2,5,3,7,101,18]` and output 4; Example 2 with input `[0,1,0,3,2,3]` and output 4; and Example 3 with input `[7,7,7,7,7,7,7]` and output 1. Constraints are listed as `1 <= nums.length <= 2500` and `-104 <= nums[i] <= 104`. The code editor shows the provided C++ solution. The test result section shows "Accepted" with a runtime of 0 ms for Case 1, where the input is `[10,9,2,5,3,7,101,18]` and the output is `4`.

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.

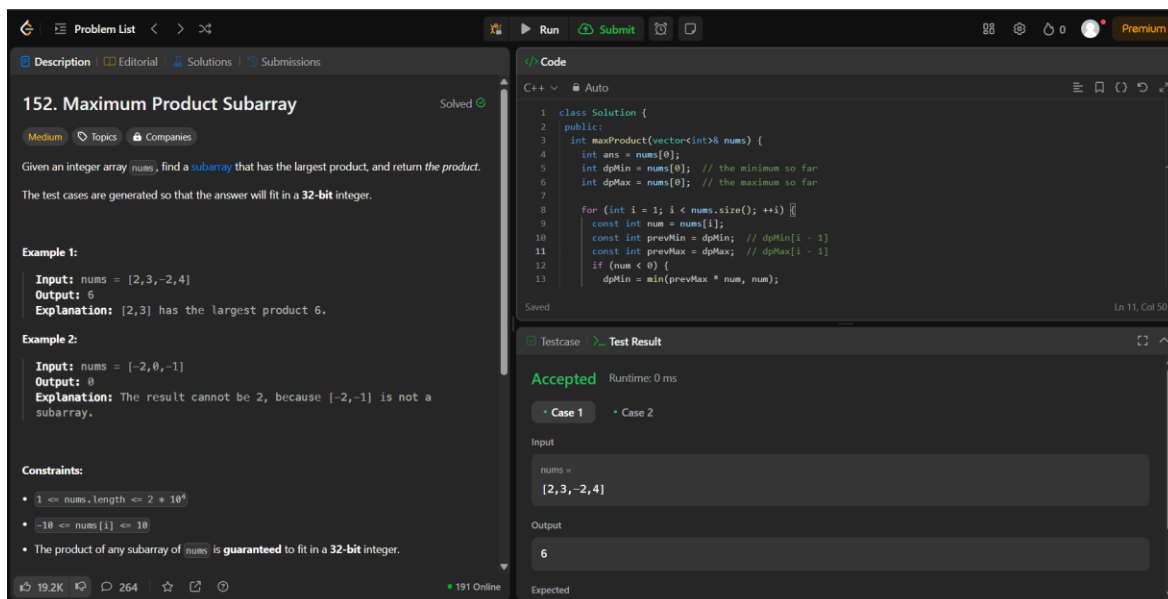
Code:

```
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 Solved

Medium Topics Companies

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:

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

Code

```
1 class Solution {
2 public:
3     int maxProduct(vector<int>& nums) {
4         int ans = nums[0];
5         int dpMin = nums[0]; // the minimum so far
6         int dpMax = nums[0]; // the maximum so far
7
8         for (int i = 1; i < nums.size(); ++i) {
9             const int num = nums[i];
10            const int prevMin = dpMin; // dpMin[i - 1]
11            const int prevMax = dpMax; // dpMax[i - 1]
12            if (num < 0) {
13                dpMin = min(prevMax * num, num);
14            } else {
15                dpMin = min(prevMin * num, num);
16                dpMax = max(prevMax * num, num);
17            }
18            ans = max(ans, dpMax);
19        }
20        return ans;
21    }
22 }
```

Testcase **Test Result**

Accepted Runtime: 0 ms

Case 1 Case 2

Input

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

Output

6

Expected



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

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:

The screenshot displays a coding problem interface. On the left, the problem description for '122. Best Time to Buy and Sell Stock II' is shown, including the input array `prices = [7,1,5,3,6,4]` and the output `7`. The explanation details the optimal trading strategy: buying on day 2 (price 1) and selling on day 3 (price 5) for a profit of 4, then buying on day 4 (price 3) and selling on day 5 (price 6) for a profit of 3, resulting in a total profit of 7. On the right, the C++ code is shown, which implements the solution using a single pass through the array, maintaining a `hold` variable for the current stock price and a `sell` variable for the maximum profit. The code is accepted, and the test result shows the input `prices = [7,1,5,3,6,4]` and the output `7`.

```
122. Best Time to Buy and Sell Stock II
Medium
Topics: Companies
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.

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.

C++ Code
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

Case 1 Case 2 Case 3

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

Output: 7

Expected



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

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 Solved

Medium Topics Companies

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:

- $1 \leq n \leq 10^4$

Code

```
C++  
1 int numSquares(int n) {  
2     vector<int> dp(n + 1, n); // 1^2 x n  
3     dp[0] = 0;                // no way  
4     dp[1] = 1;                // 1^2  
5  
6     for (int i = 2; i <= n; ++i)  
7         for (int j = 1; j * j <= i; ++j)  
8             dp[i] = min(dp[i], dp[i - j * j] + 1);  
9  
10    return dp[n];  
11 }  
12  
13  
14 }
```

Saved Ln 11, Col 1

Testcase **Test Result**

Accepted Runtime: 0 ms

Case 1 Case 2

Input

$n =$
12

Output

3

Expected