**Assignment 7 (Advance Programming)**

**Q-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.

**Solution:**

class Solution {

public:

    int rob(vector<int>& nums) {

        int n = nums.size();

        if (n == 0) return 0;

        if (n == 1) return nums[0];

        int prev2 = 0, prev1 = 0, curr = 0;

        for (int num : nums) {

            curr = max(prev1, num + prev2);

            prev2 = prev1;

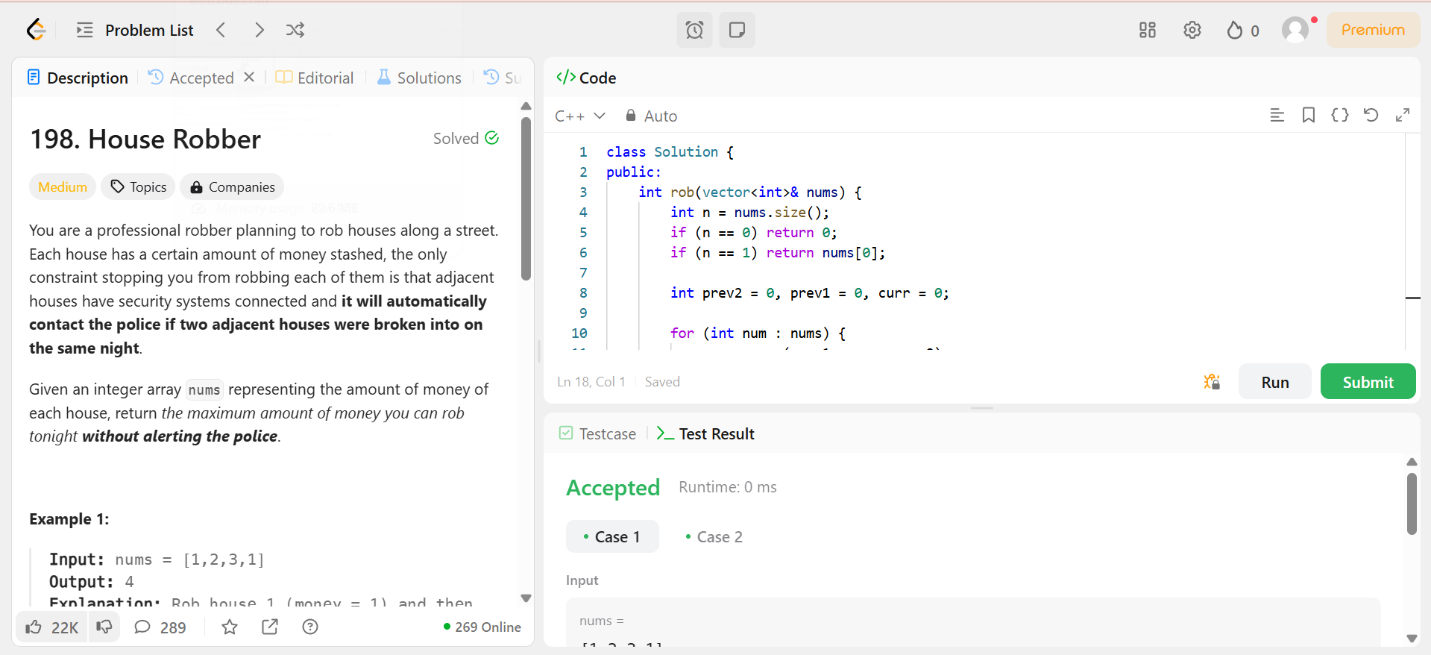
            prev1 = curr;

        }

        return curr;

    }

};



**Q-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.

**Solution:**

class Solution {

public:

    bool canJump(vector<int>& nums) {

        int maxReach = 0; // Tracks the farthest index we can reach

        for (int i = 0; i < nums.size(); i++) {

            if (i > maxReach) return false; // If we are stuck, return false

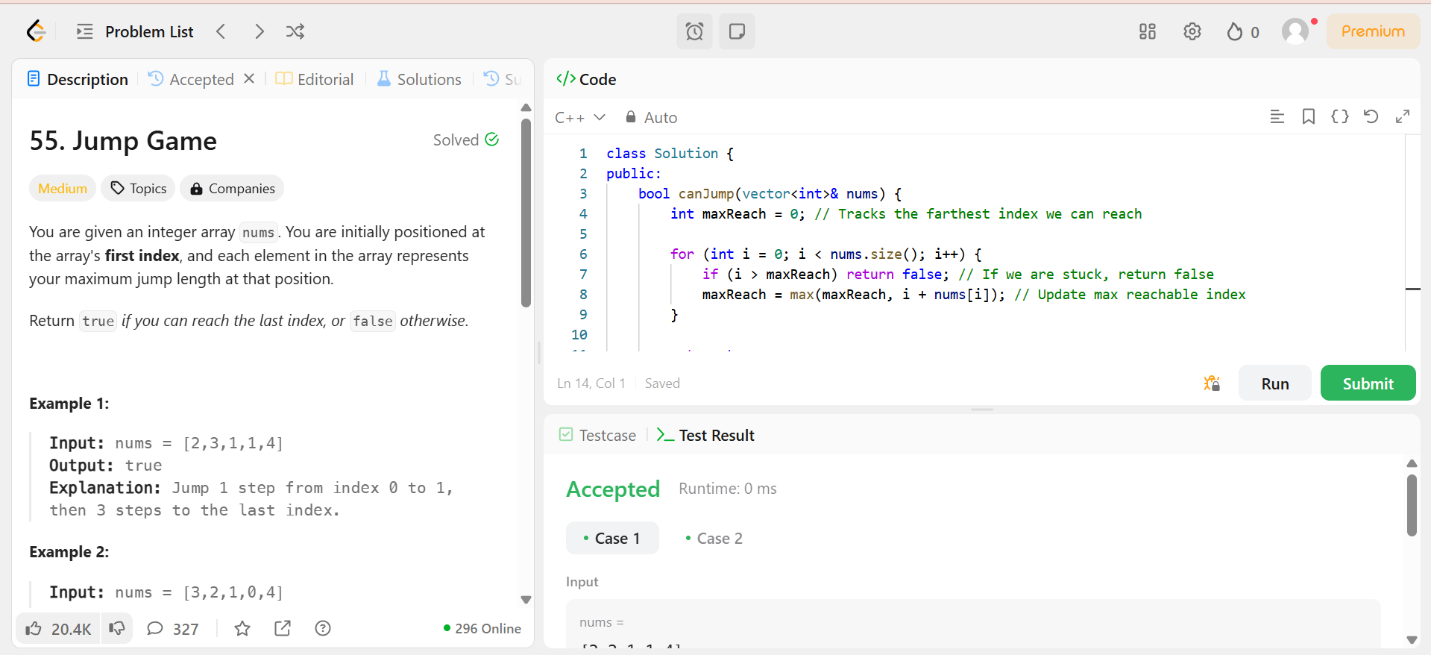
            maxReach = max(maxReach, i + nums[i]); // Update max reachable index

        }

        return true;

    }

};



**Q-152 Maximum Product Sub array**

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.

**Solution:**

class Solution {

public:

    int maxProduct(vector<int>& nums) {

        if (nums.empty()) return 0;

        int maxProd = nums[0], minProd = nums[0], result = nums[0];

        for (int i = 1; i < nums.size(); i++) {

            if (nums[i] < 0) swap(maxProd, minProd); // Swap when negative to handle sign change

            maxProd = max(nums[i], maxProd \* nums[i]);

            minProd = min(nums[i], minProd \* nums[i]);

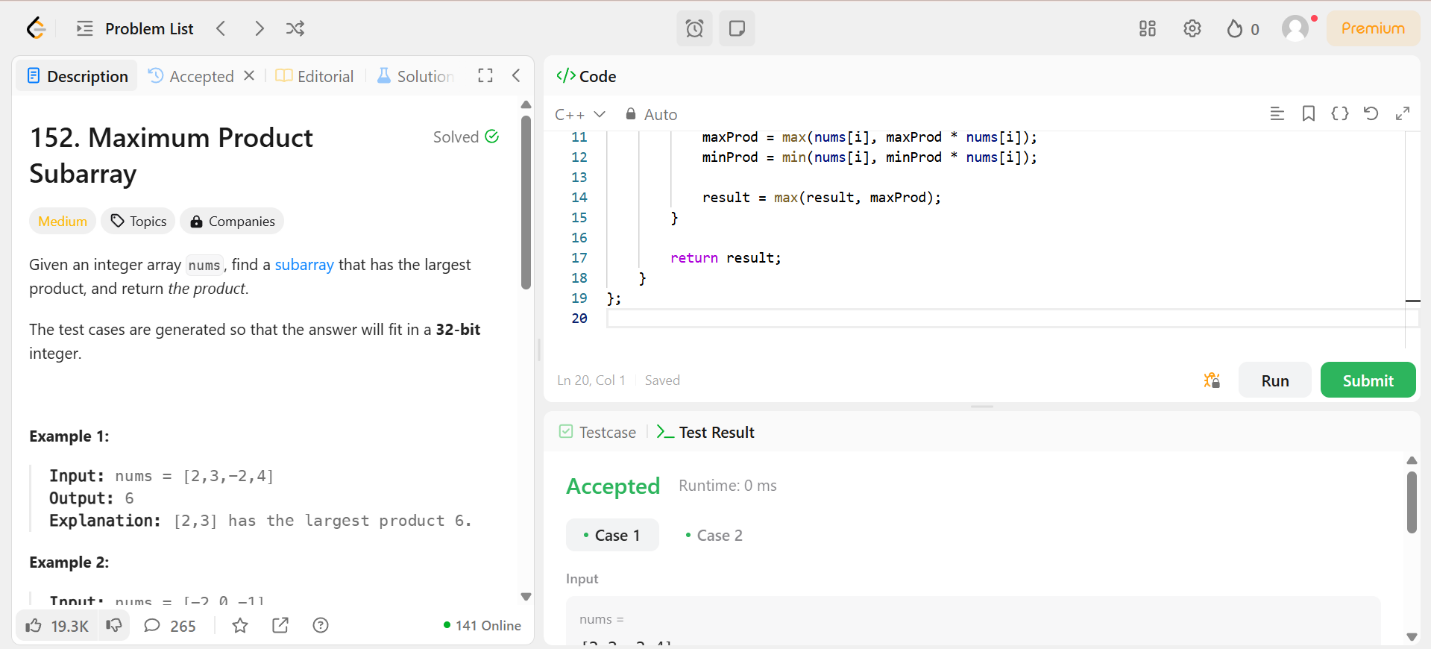
            result = max(result, maxProd);

        }

        return result;

    }

};



**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.

**Solution:**

class Solution {

public:

    int numSquares(int n) {

        vector<int> dp(n + 1, INT\_MAX);

        dp[0] = 0;

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

            for (int j = 1; j \* j <= i; j++) {

                dp[i] = min(dp[i], 1 + dp[i - j \* j]);

            }

        }

        return dp[n];

    }

};

