DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment-6

Student Name: Gurleen Kaur **UID:** 22BCS15915
Branch: BE-CSE **Section/Group:** KPIT-901(B)
Semester: 6TH **Date of Performance:**13-03-25
Subject Name: Advanced Programming Lab - II **Subject Code:** 22CSP-351

1. Aim:

a. Find the number of ways to climb n stairs when you can take 1 or 2 steps at a time. This follows the Fibonacci sequence and can be solved using dynamic programming (DP) in O(n) time and O(1) space. Edge cases include n = 1 and n = 2.

b. Maximum Subarray Find the contiguous subarray with the maximum sum using Kadane's Algorithm. This runs in O(n) time and O(1) space. Edge cases include an array of all negative numbers.

c. You are given an array representing the amount of money in each house on a street. You cannot rob two adjacent houses. Determine the maximum amount you can rob without alerting the police.

d. Jump Game Given an array where each element represents the maximum jump length from that position, determine if you can reach the last index starting from the first index.

e. Unique Paths You are given an m x n grid where a robot starts at the top-left and can only move right or down. Determine the number of unique paths the robot can take to reach the bottom-right corner.

f. Coin Change Given an array of coin denominations and an amount, determine the minimum number of coins needed to make up that amount. If it is impossible, return -1.

g. Longest Increasing Subsequence Given an array of integers, find the length of the longest subsequence where the elements are strictly increasing.

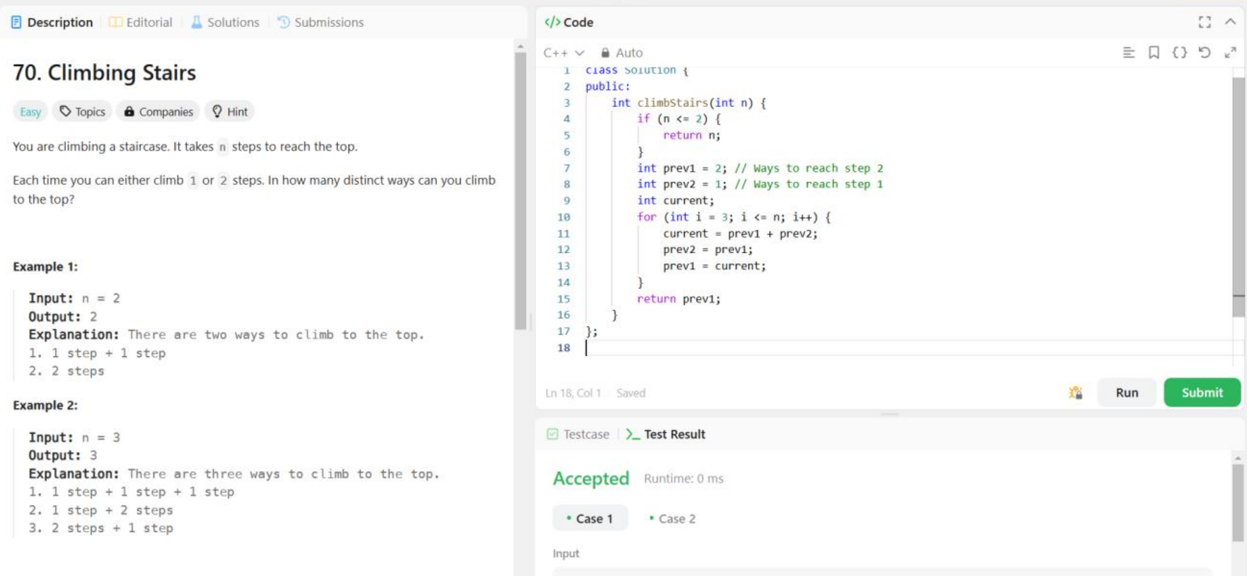
h. Maximum Product Subarray Given an array of integers, find the contiguous subarray (of at least one element) that has the largest product and return its product.

i. Word Break Given a string s and a dictionary of words, determine if s can be segmented into one or more dictionary words.

2. Implementation/Code:

a)

```
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 2) {
            return n;
        }
        int prev1 = 2; // Ways to reach step 2
        int prev2 = 1; // Ways to reach step 1
        int current;
        for (int i = 3; i <= n; i++) {
            current = prev1 + prev2;
            prev2 = prev1;
            prev1 = current;
        }
        return prev1;
    }
};
```



70. Climbing Stairs

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
```

```
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 2) {
            return n;
        }
        int prev1 = 2; // Ways to reach step 2
        int prev2 = 1; // Ways to reach step 1
        int current;
        for (int i = 3; i <= n; i++) {
            current = prev1 + prev2;
            prev2 = prev1;
            prev1 = current;
        }
        return prev1;
    }
};

```

Ln 18, Col 1 Saved

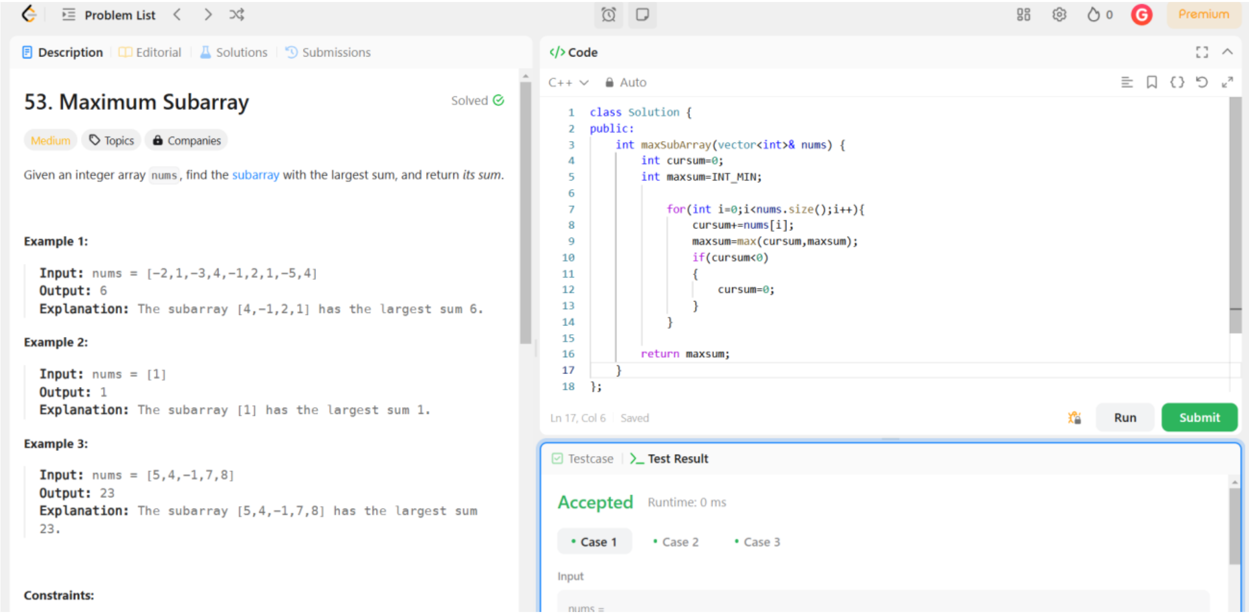
Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

b)

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int cursum=0;
        int maxsum=INT_MIN;

            for(int i=0;i<nums.size();i++){
                cursum+=nums[i];
                maxsum=max(cursum,maxsum);
                if(cursum<0)
                {
                    cursum=0;
                }
            }
        return maxsum;
    }
};
```



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:

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int cursum=0;
        int maxsum=INT_MIN;

          for(int i=0;i<nums.size();i++){
              cursum+=nums[i];
              maxsum=max(cursum,maxsum);
              if(cursum<0)
              {
                  cursum=0;
              }
          }

        return maxsum;
    }
};
```

Ln 17, Col 6   Saved

Accepted   Runtime: 0 ms

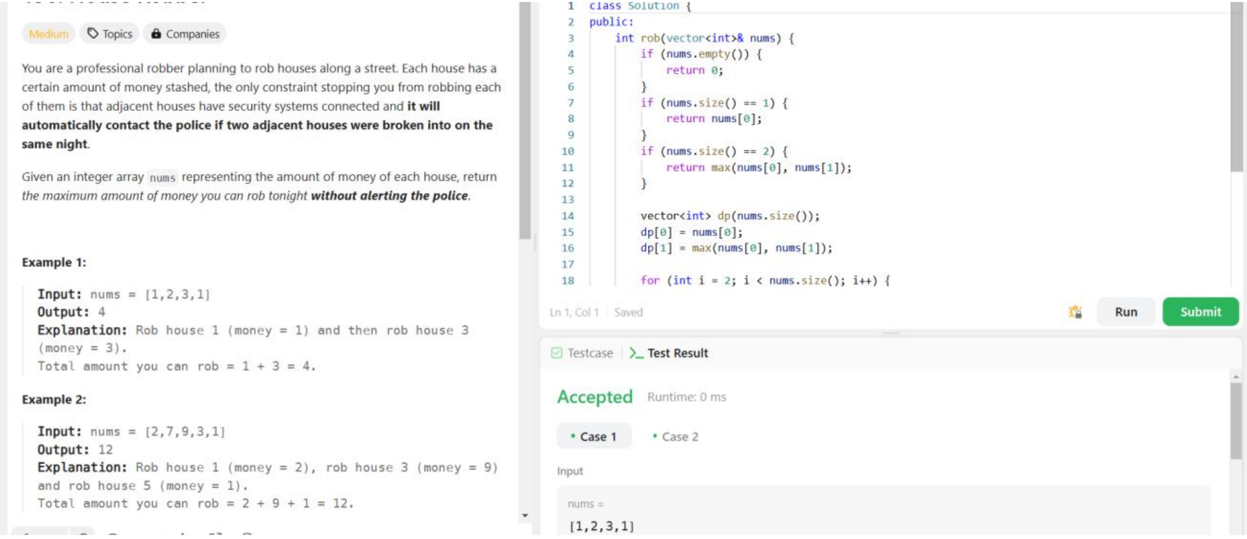
• Case 1   • Case 2   • Case 3

Input

nums =

c)

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }
        if (nums.size() == 2) {
            return max(nums[0], nums[1]);
        }

        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[nums.size() - 1];
    }
};
```



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.
```

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }
        if (nums.size() == 2) {
            return max(nums[0], nums[1]);
        }

        vector<int> dp(nums.size());
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);

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

Accepted Runtime: 0 ms

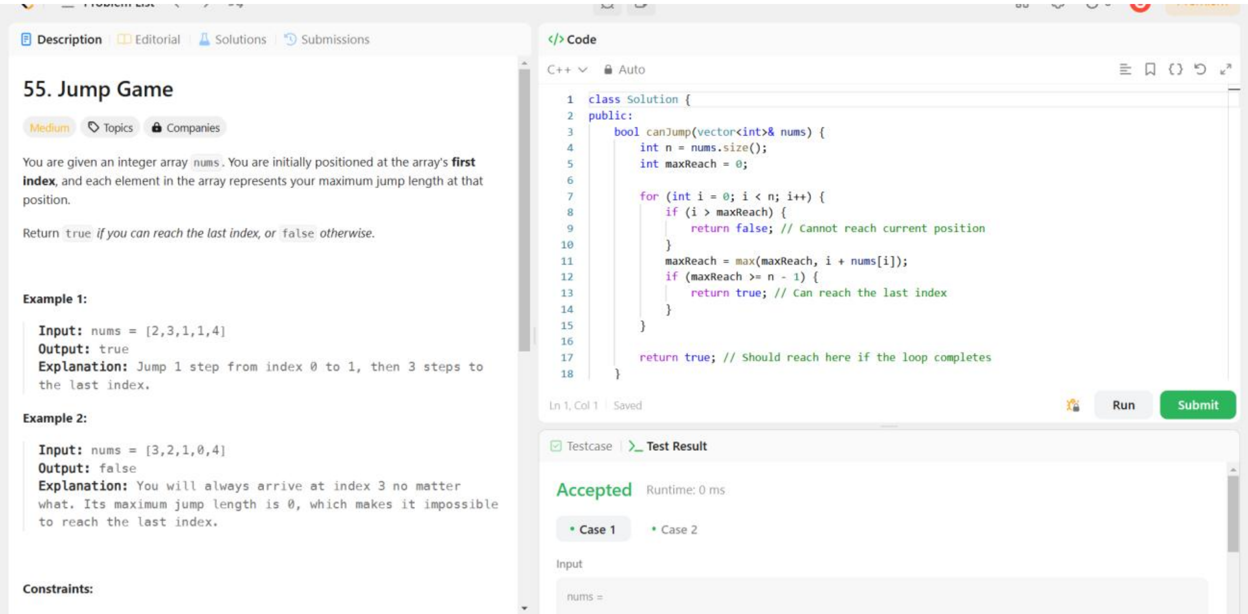
Input

nums =

[1,2,3,1]

d)

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n = nums.size();
        int maxReach = 0;

        for (int i = 0; i < n; i++) {
            if (i > maxReach) {
                return false; // Cannot reach current position
            }
            maxReach = max(maxReach, i + nums[i]);
            if (maxReach >= n - 1) {
                return true; // Can reach the last index
            }
        }

        return true; // Should reach here if the loop completes
    }
};
```



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:

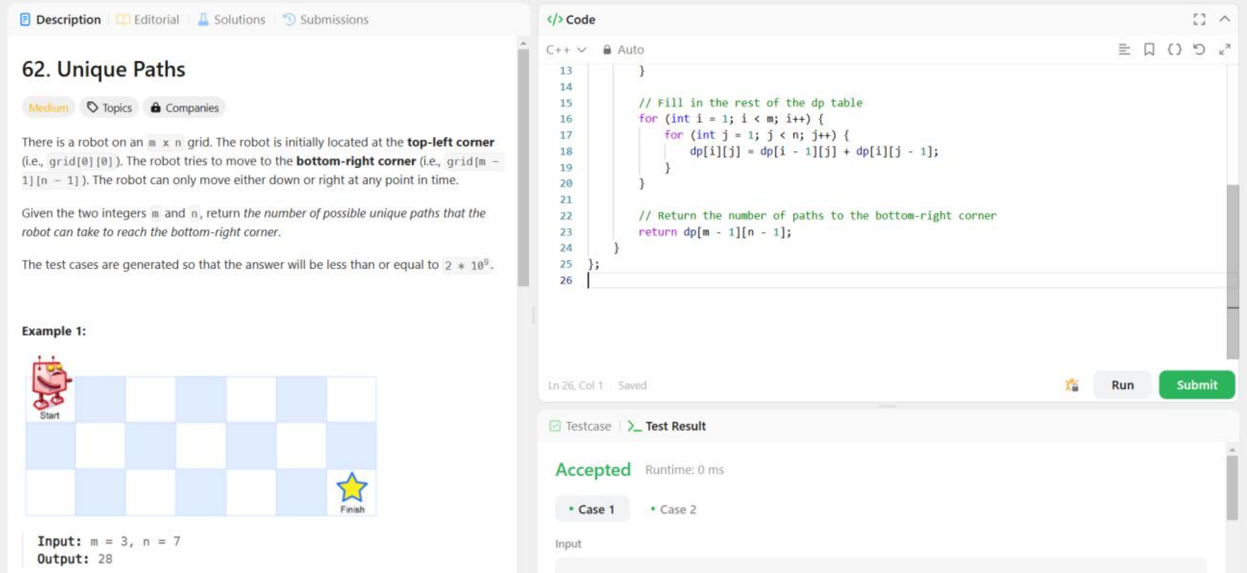
```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter
what. Its maximum jump length is 0, which makes it impossible
to reach the last index.
```

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n = nums.size();
        int maxReach = 0;

        for (int i = 0; i < n; i++) {
            if (i > maxReach) {
                return false; // Cannot reach current position
            }
            maxReach = max(maxReach, i + nums[i]);
            if (maxReach >= n - 1) {
                return true; // Can reach the last index
            }
        }

        return true; // Should reach here if the loop completes
    }
```

Accepted Runtime: 0 ms

e)

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        // Create a 2D vector to store the number of paths
        vector<vector<int>> dp(m, vector<int>(n, 0));

        // Initialize the first row and first column to 1
        for (int i = 0; i < m; i++) {
            dp[i][0] = 1;
        }
…
        // Return the number of paths to the bottom-right corner
        return dp[m - 1][n - 1];
    }
};
```



62. Unique Paths

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:

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

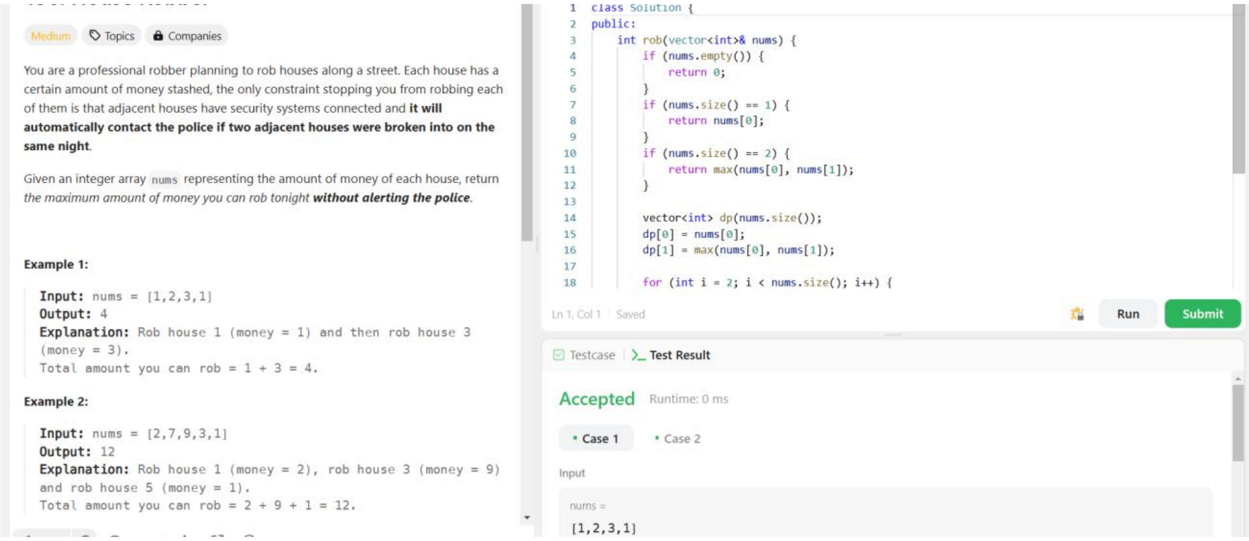
```
        }

        // Fill in the rest of the dp table
        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 the number of paths to the bottom-right corner
        return dp[m - 1][n - 1];
    }
};
```

Accepted Runtime: 0 ms

Input

f)

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }
        if (nums.size() == 2) {
            return max(nums[0], nums[1]);
        }

        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[nums.size() - 1];
    }
};
```



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.
```

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }
        if (nums.size() == 2) {
            return max(nums[0], nums[1]);
        }

        vector<int> dp(nums.size());
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);

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

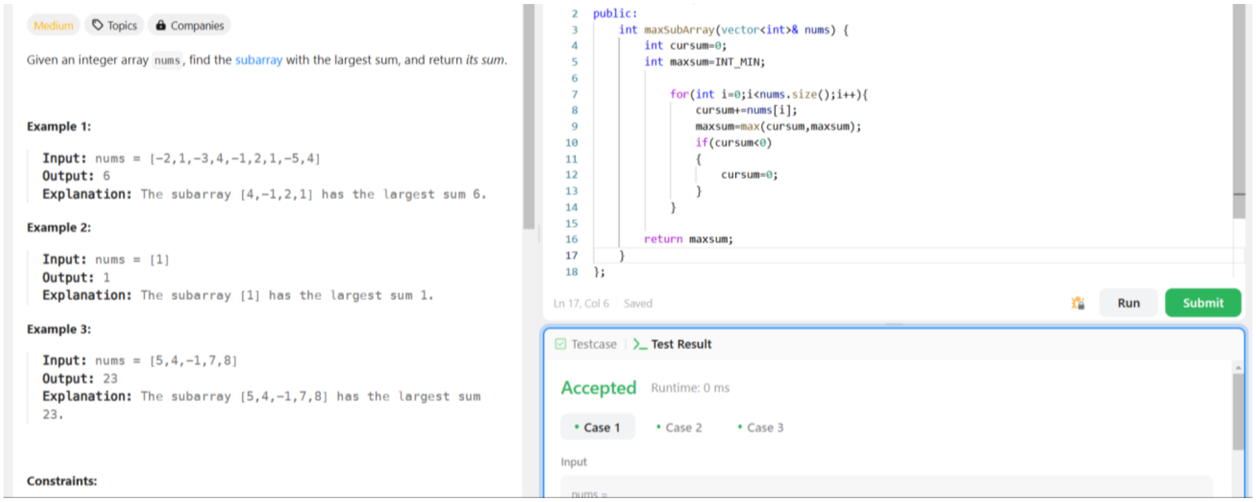
Accepted Runtime: 0 ms

Input

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

g)

```
 class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int cursum=0;
        int maxsum=INT_MIN;

            for(int i=0;i<nums.size();i++){
                cursum+=nums[i];
                maxsum=max(cursum,maxsum);
                if(cursum<0)
                {
                    cursum=0;
                }
            }

        return maxsum;
    }
};
```



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:

```
public:
    int maxSubArray(vector<int>& nums) {
        int cursum=0;
        int maxsum=INT_MIN;

            for(int i=0;i<nums.size();i++){
                cursum+=nums[i];
                maxsum=max(cursum,maxsum);
                if(cursum<0)
                {
                    cursum=0;
                }
            }

        return maxsum;
    }
};
```

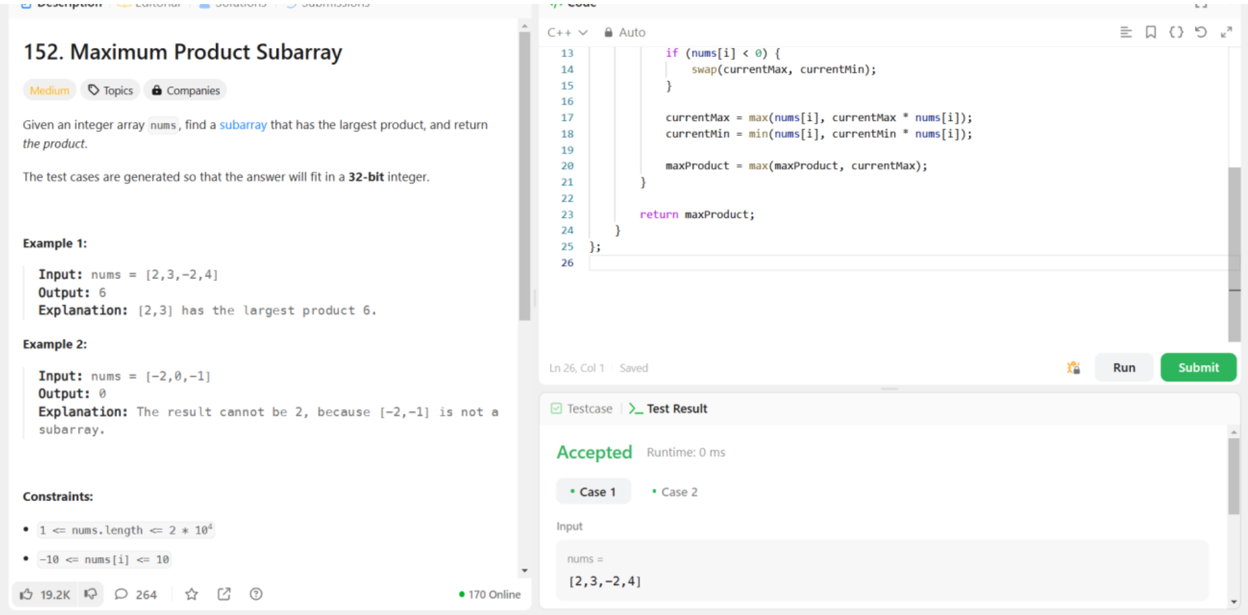
Accepted Runtime: 0 ms

Input

nums =

h)

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

        int maxProduct = nums[0];
        int currentMax = nums[0];
        int currentMin = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] < 0) {
                swap(currentMax, currentMin);
            }

            currentMax = max(nums[i], currentMax * nums[i]);
            currentMin = min(nums[i], currentMin * nums[i]);

            maxProduct = max(maxProduct, currentMax);
        }

        return maxProduct;
    }
};
```



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`

```
            if (nums[i] < 0) {
                swap(currentMax, currentMin);
            }

            currentMax = max(nums[i], currentMax * nums[i]);
            currentMin = min(nums[i], currentMin * nums[i]);

            maxProduct = max(maxProduct, currentMax);
        }

        return maxProduct;
    }
};
```

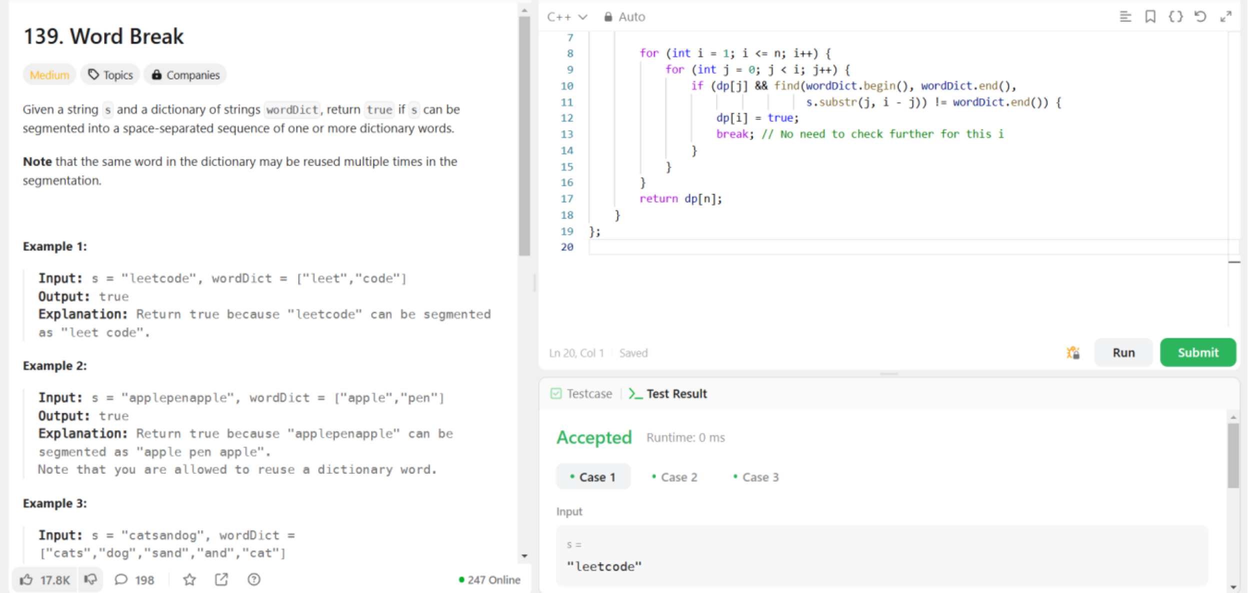
Accepted Runtime: 0 ms

Input

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

i)

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        int n = s.length();
        vector<bool> dp(n + 1, false);
        dp[0] = true; // Empty string is always breakable

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && find(wordDict.begin(), wordDict.end(), s.substr(j, i - j)) !=
        wordDict.end()) {
                    dp[i] = true;
                    break; // No need to check further for this i
                }
            }
        }
        return dp[n];
    }
};
```



139. Word Break

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented
as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be
segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input: s = "catsandog", wordDict =
["cats","dog","sand","and","cat"]
```

```cpp

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && find(wordDict.begin(), wordDict.end(),
                                  s.substr(j, i - j)) != wordDict.end()) {
                    dp[i] = true;
                    break; // No need to check further for this i
                }
            }
        }
        return dp[n];
    }
};

```

Accepted Runtime: 0 ms

Input

s =

"leetcode"

3. Learning Outcome:

- Practice using C++ syntax for defining classes, functions, and vectors.
- Learnt how to iterate through strings and vectors using loops.
- Using the find function to search inside of a vector.
- Understand how to break down a larger problem into smaller, overlapping subproblems and store the results to avoid redundant computations.