**1. Understanding Binary Trees and Binary Search Trees**

**Binary Trees:**

Binary trees are a type of data structure where each node can have at most two children, called the left and right child. I learned that there are different types of binary trees:

* **Full Binary Tree**: Every node has either no children or exactly two children.
* **Complete Binary Tree**: All levels are fully filled except the last one, where nodes are as far left as possible.

**Tree Traversal Methods**

There are several ways to go through a binary tree, and I learned about three main ones:

* **Pre-order Traversal**: You visit the root node first, then the left subtree, and finally the right subtree.
* **In-order Traversal**: The left subtree is visited first, then the root, and then the right subtree. This is useful in binary search trees because it gives the values in sorted order.
* **Post-order Traversal**: You visit both subtrees first, and then the root. This is helpful when you need to delete or process nodes from the bottom up.

**Binary Search Trees (BST)**

A Binary Search Tree (BST) is a special type of binary tree where all nodes in the left subtree are smaller than the root, and all nodes in the right subtree are greater. I also learned how efficient BSTs can be for searching, inserting, and deleting data.

**BST Operations:**

* **Insertion**: To add a new node, you compare its value with the current node’s value and navigate left or right until you find an empty spot.
* **Search**: Similar to insertion, you move left or right depending on whether the value you’re looking for is smaller or greater than the current node.
* **Deletion**: Deleting nodes was a bit tricky, especially when the node had two children. In that case, you have to replace it with either the in-order predecessor or successor.

**Applications**

I also discovered that binary trees and BSTs are used in real-life applications like:

* **Database indexing** for faster search.
* **File systems** to store hierarchical data.
* **Autocomplete systems** that handle large datasets for quick suggestions.

**2. Challenges Faced**

**Challenge 1: Understanding Tree Traversals**

It was a bit hard at first to fully understand the difference between pre-order, in-order, and post-order traversal. It was especially confusing when I tried to imagine how the recursion works.

* **Solution**: I worked through several small examples on paper to understand the process better. Drawing the trees and following each step helped me see how the traversal works.

**Challenge 2: Keeping the Tree Balanced**

I realized that if a BST gets unbalanced, it can slow down operations to O(n). I hadn’t thought much about this before, but an unbalanced tree can become inefficient.

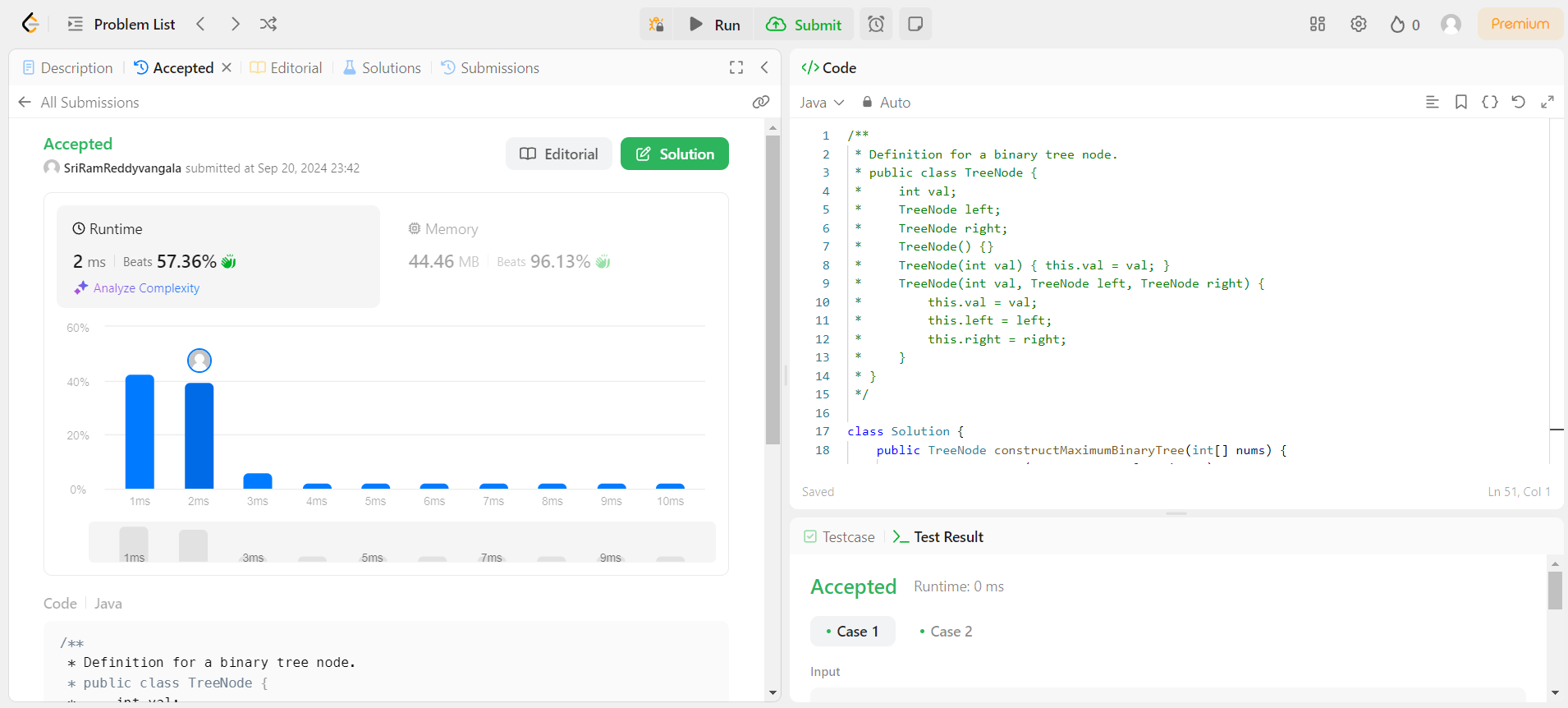
* **Solution**: I looked into advanced topics like AVL trees and Red-Black trees to understand how balancing works. Even though it wasn’t part of the original study, learning about these concepts made a big difference.

**Leet Code:  
  
Question 1:**

**You are given an integer array nums with no duplicates. A maximum binary tree can be built recursively from nums using the following algorithm:**

1. **Create a root node whose value is the maximum value in nums.**
2. **Recursively build the left subtree on the subarray prefix to the left of the maximum value.**
3. **Recursively build the right subtree on the subarray suffix to the right of the maximum value.**

**Return *the maximum binary tree built from*nums.**



**Challenge 1: Recursive Tree Building and Subarray Handling**

* **Issue:** Managing the recursive tree building process, especially ensuring that subarrays are passed correctly during recursive calls without excessive memory consumption.
* **Solution:** Instead of creating new subarrays for every recursive call, the approach uses index boundaries (left and right) to refer to subarrays in the original array. This avoids memory overhead from creating new arrays at every step, keeping memory usage efficient.

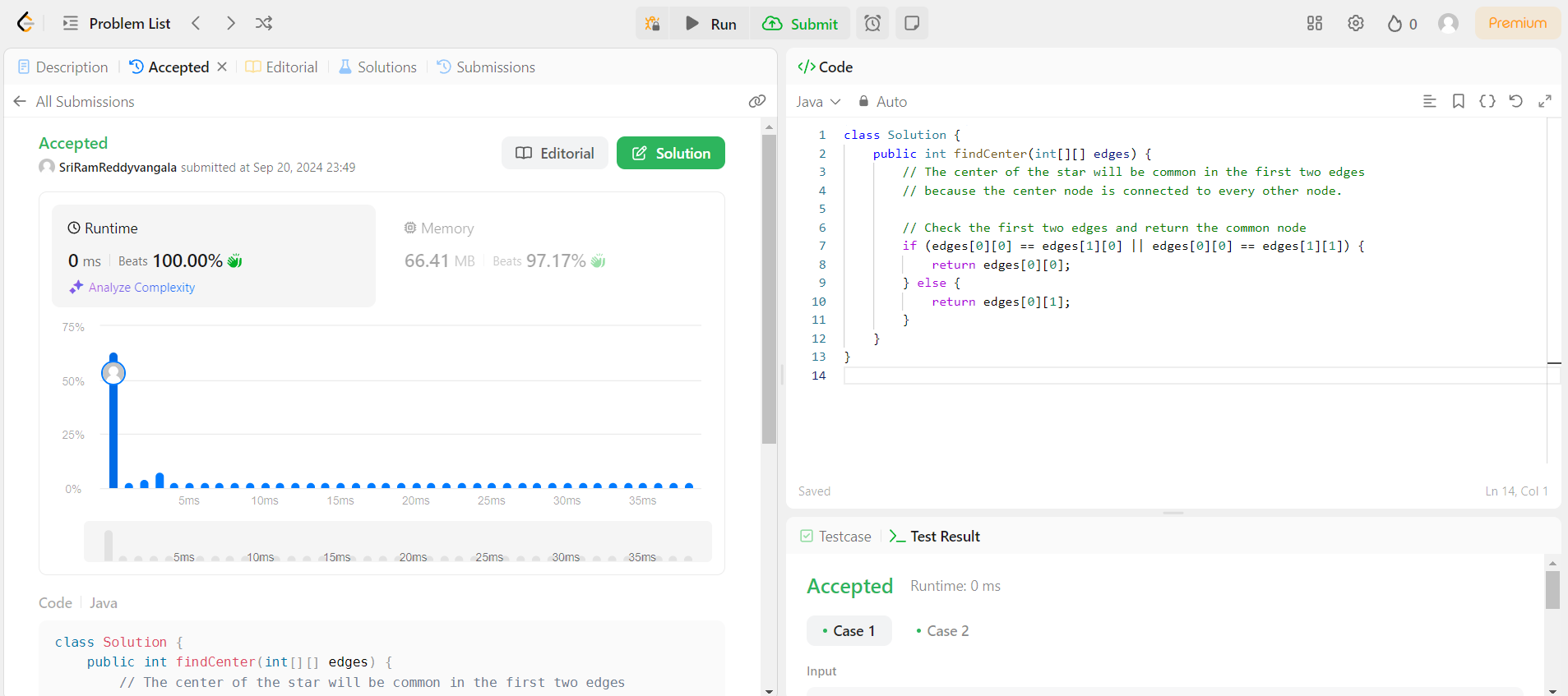
**Challenge 2: Finding Maximum Element in Subarrays Efficiently**

* **Issue:** The brute-force approach to finding the maximum value in each subarray during every recursive call leads to O(n²) complexity, which can be slow for large input arrays.
* **Solution:** A potential optimization involves using a **monotonic stack** to track the maximum elements during a single traversal, reducing the complexity to O(n). However, within problem constraints, a simpler approach of iterating through the subarray using the findMaxIndex method was chosen, which efficiently finds the maximum without memory overhead.

**Question 2:**

**There is an undirected star graph consisting of n nodes labeled from 1 to n. A star graph is a graph where there is one center node and exactly n - 1 edges that connect the center node with every other node.**

**You are given a 2D integer array edges where each edges[i] = [ui, vi] indicates that there is an edge between the nodes ui and vi. Return the center of the given star graph.**



**Challenge 1: Efficient Center Detection**

* **Issue:** Identifying the center of a star graph should be done in constant time. Traversing all edges to count occurrences would be inefficient, especially for large graphs with up to 10510^5105 edges.
* **Solution:** The center node appears in every edge, so we only need to check the first two edges. The node common to both edges is the center of the star graph. This solution operates in O(1) time complexity, as we don’t need to check more than two edges to find the center.

**Challenge 2: Handling Edge Cases with Small Graphs**

* **Issue:** For very small star graphs (n = 3), the logic should still work without additional handling, but it's crucial to ensure that the comparison logic in findCenter correctly identifies the center even when only 2 edges are present.
* **Solution:** Since the problem guarantees a valid star graph, checking the first two edges will always work, even for the smallest possible graph with 3 nodes. Thus, no additional handling is required for edge cases.

**2.** **Understanding Dynamic Programming**

**What is Dynamic Programming?**

Dynamic Programming (DP) is a technique used to solve problems by breaking them down into smaller, overlapping subproblems and storing the results to avoid recalculating them. It’s useful for optimization problems where repeated calculations are common. The two key ideas are:

* **Overlapping Subproblems**: When the same subproblem is solved multiple times.
* **Optimal Substructure**: When the solution to a problem can be constructed from the solutions of its subproblems.

**Key Concepts in Dynamic Programming**

* **Memoization (Top-down)**: In this approach, I learned that we solve the problem recursively but store the results of subproblems so we don’t have to recompute them.

**Example**: Solving the Fibonacci sequence using memoization:

scss

fib(n) = fib(n-1) + fib(n-2)

We store previously computed values, so we don’t need to calculate them again.

* **Tabulation (Bottom-up)**: This is a non-recursive approach where we solve smaller subproblems first and store the results in a table, using the table to build up to the final solution.

**Example**: Fibonacci sequence using tabulation:

less

fib[0] = 0, fib[1] = 1

for i = 2 to n:

fib[i] = fib[i-1] + fib[i-2]

**Common Dynamic Programming Problems**

* **Fibonacci Sequence**: Classic example to understand overlapping subproblems.
* **Knapsack Problem**: Choosing items to maximize value without exceeding a weight limit.
* **Coin Change Problem**: Finding the minimum number of coins for a given amount.
* **Longest Common Subsequence**: Finding the longest sequence common between two strings.

**2. Challenges Faced**

**Challenge 1: Recognizing Dynamic Programming Problems**

At first, I found it difficult to know when a problem could be solved using dynamic programming. Simple problems like Fibonacci were easy to spot, but more complex ones like Knapsack or Coin Change weren’t as obvious to me.

* **Solution**: I started to look for repetitive calculations in the problem. Whenever I noticed a problem where the solution to smaller parts of the problem could be reused, it usually meant dynamic programming could be applied.

**Challenge 2: Choosing Memoization or Tabulation**

I was unsure about when to use memoization (top-down) versus tabulation (bottom-up). Memoization felt easier at first because I could think of it in terms of recursion, but I read that tabulation can be more efficient.

* **Solution**: I practiced using both methods. I found memoization easier for starting out, but tabulation helped me understand how to solve the problem from the ground up. In some cases, like for larger datasets, tabulation turned out to be faster.

**Challenge 3: Defining States and Transitions**

Defining the "state" of the problem in dynamic programming was tricky for me. For example, in the Knapsack problem, it wasn’t clear at first how I should structure the subproblems and transition between them.

* **Solution**: I broke the problem down step by step, writing out what each state should represent. In the Knapsack problem, for example, the state represents the maximum value I can get for a given capacity, and the transition is deciding whether to include an item or not.

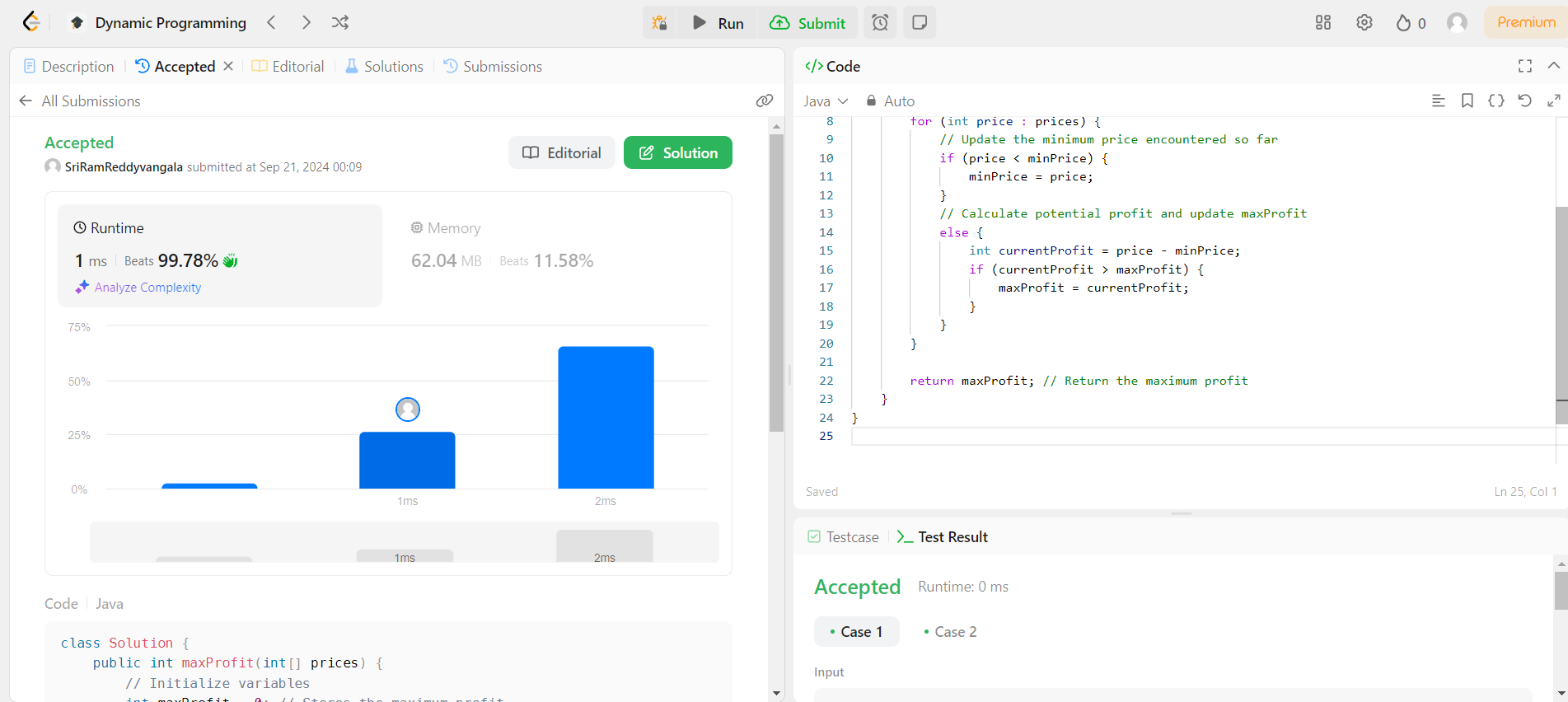
**Leet Code:**

**Question 1:**

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

**  
  
Challenge 1: Tracking Minimum Price**

* **Issue**: We need to track the minimum price seen so far as we iterate through the prices array because the optimal buy day must be before the sell day.
* **Solution**: Keep a running minimum price (minPrice) as we traverse the array. At each step, calculate the potential profit by subtracting the current price from this minimum price.

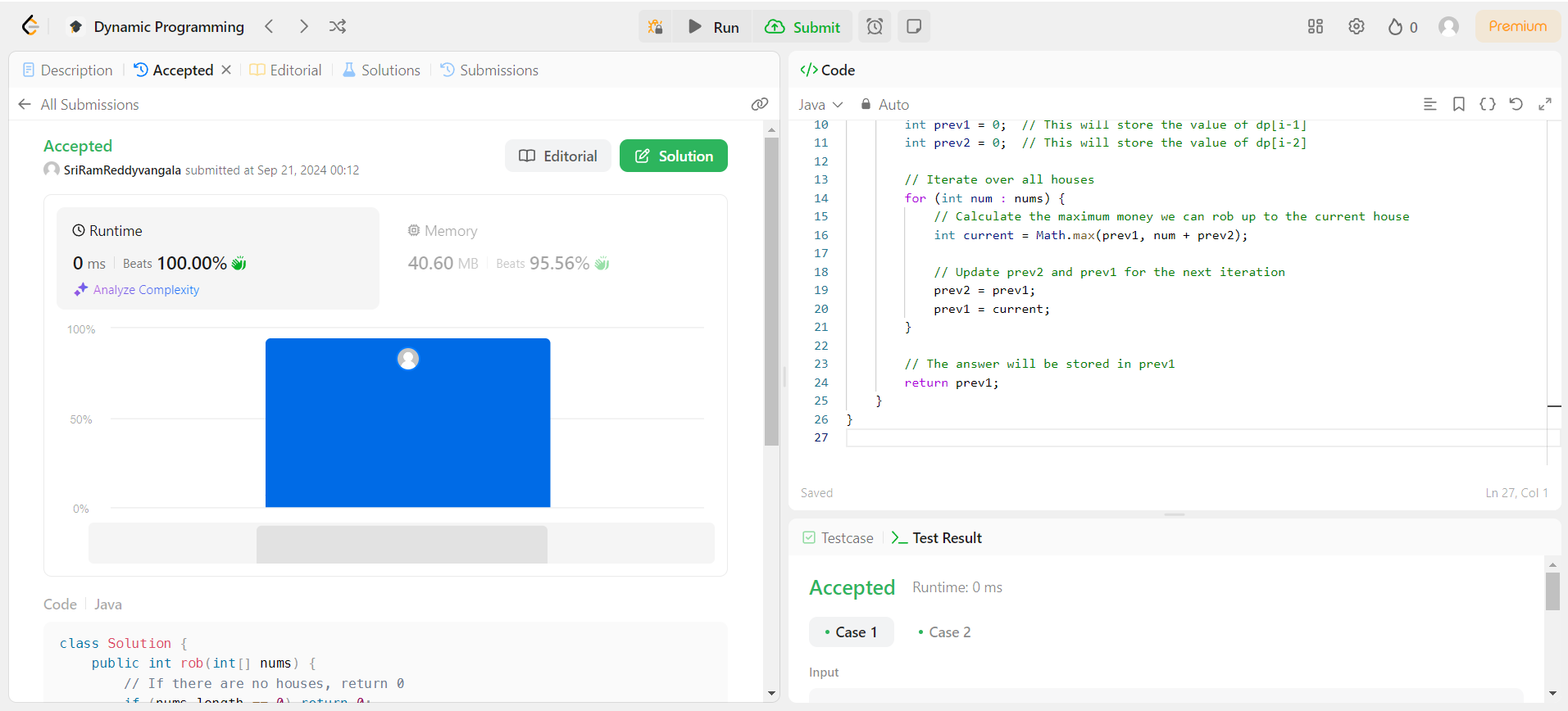
**Challenge 2: Calculating Maximum Profit**

* **Issue**: We need to find the maximum profit by checking the difference between the current price and the minimum price so far.
* **Solution**: At each step, compute the current profit (currentProfit = prices[i] - minPrice) and update the maximum profit if this value is greater than the current maximum.

**Question 2:**

**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.**

****

**Challenge 1: Making Decisions About Robbing Houses**

* **Issue**: For each house, you have two choices:
  1. Rob the current house and skip the previous house.
  2. Skip the current house and take the maximum profit from the previous houses.
* **Solution**: We can keep track of the maximum amount of money that can be robbed up to each house using a dynamic programming table or two variables. At each step, the decision depends on whether you rob the current house or not.

**Challenge 2: Avoiding Adjacent Robberies**

* **Issue**: If you rob two consecutive houses, the alarm will be triggered.
* **Solution**: Use the recurrence relation: for each house i, the maximum money you can rob is either:
  + The money from robbing the current house (nums[i]) plus the money from robbing up to the house two places before (dp[i-2]).
  + The money from skipping the current house and taking the maximum from the previous house (dp[i-1]).