peopleTECH

A Quest Global Company

INTERNSHIP REPORT

WEEK 2 – Leet Code Documentation

SUBMITTED TO

PEOPLE TECH GROUP INC

GOVIND SHARMA

RECRUITMENT LEAD

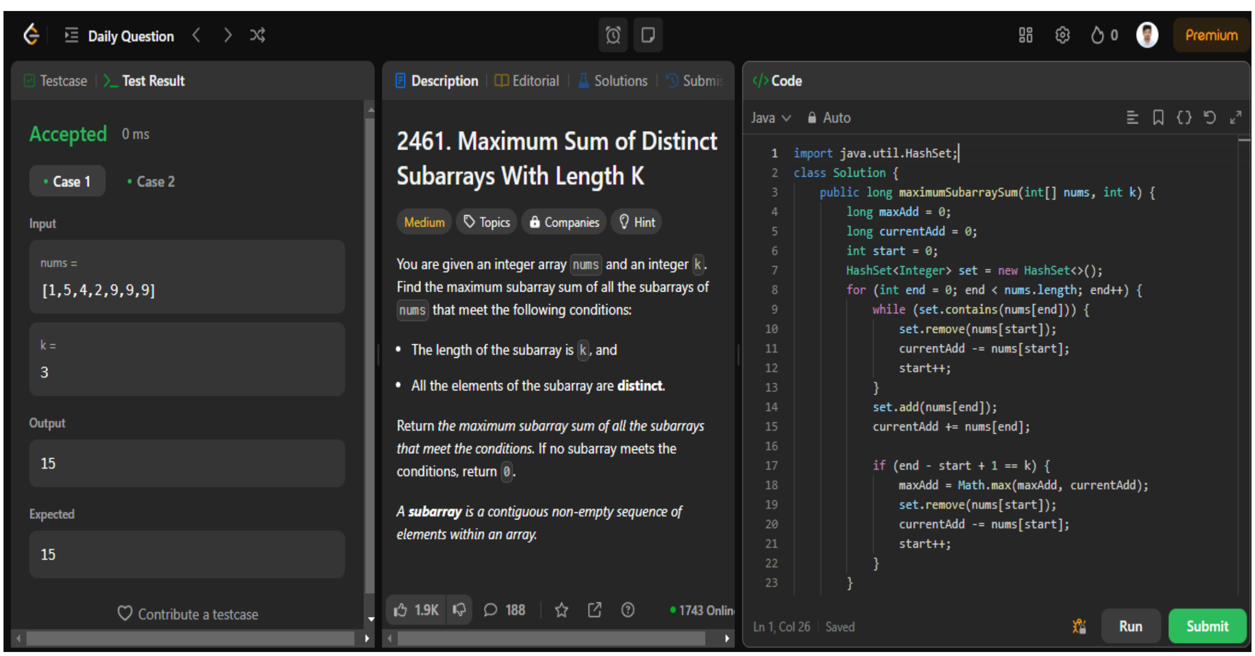
SUBMITTED BY

SANJEEV REDDY SIRIPINANE

All My Submissions

Time Submitted	Question	Status	Runtime	Language
9 hours ago	Minimum Path Sum	Accepted	4 ms	java
13 hours, 35 minutes ago	Unique Paths	Accepted	0 ms	java
1 day, 1 hour ago	Unique Binary Search Trees	Accepted	0 ms	java
1 day, 5 hours ago	Edit Distance	Accepted	5 ms	java
1 day, 8 hours ago	Unique Binary Search Trees II	Accepted	1 ms	java
1 day, 9 hours ago	Maximum Subarray	Accepted	1 ms	java
1 day, 10 hours ago	Generate Parentheses	Accepted	1 ms	java
1 day, 11 hours ago	Longest Path With Different Adjacent Characters	Accepted	86 ms	java
1 day, 13 hours ago	Number of Good Paths	Accepted	115 ms	java
2 days, 7 hours ago	Find Minimum Diameter After Merging Two Trees	Accepted	323 ms	java
2 days, 7 hours ago	Find Minimum Diameter After Merging Two Trees	Wrong Answer	N/A	java
2 days, 7 hours ago	Find Minimum Diameter After Merging Two Trees	Runtime Error	N/A	java
2 days, 8 hours ago	Validate Binary Tree Nodes	Accepted	4 ms	java
2 days, 10 hours ago	Reachable Nodes With Restrictions	Accepted	69 ms	java
2 days, 12 hours ago	Most Profitable Path in a Tree	Accepted	53 ms	java
2 days, 13 hours ago	Minimum Fuel Cost to Report to the Capital	Accepted	52 ms	java
2 days, 14 hours ago	Time Taken to Mark All Nodes	Time Limit Exceeded	N/A	java
3 days, 5 hours ago	Time Taken to Mark All Nodes	Time Limit Exceeded	N/A	java
3 days, 5 hours ago	Time Taken to Mark All Nodes	Time Limit Exceeded	N/A	java
3 days, 8 hours ago	Maximum Sum of Distinct Subarrays With Length K	Accepted	32 ms	java

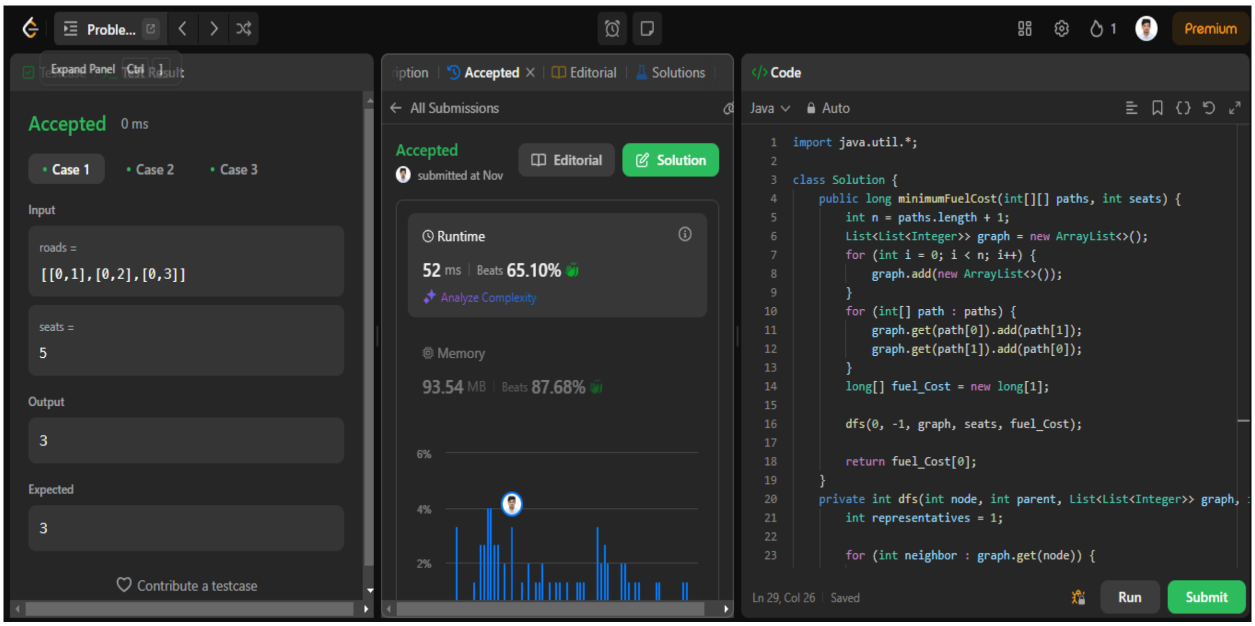
Maximum Sum of Distinct Subarrays with Length K:



Approach:

This algorithm aims to find the maximum sum of a subarray of length k with no duplicate values. It uses a sliding window approach to efficiently explore all potential subarrays. The idea is to maintain a window of size k while ensuring all elements in the window are unique. To achieve this, a HashSet is used to track the elements within the current window. As the end pointer moves through the array, it checks if the current element is already in the set. If it is, the start pointer is moved forward to shrink the window from the left, removing the duplicate and adjusting the sum accordingly. The currentAdd variable tracks the sum of the elements in the current window, and whenever the window reaches the desired length of k, the algorithm compares the current sum to the maximum sum encountered so far (maxAdd). By continuously updating the window and ensuring the uniqueness of its elements, the algorithm efficiently computes the maximum sum of the subarray of length k without duplicates.

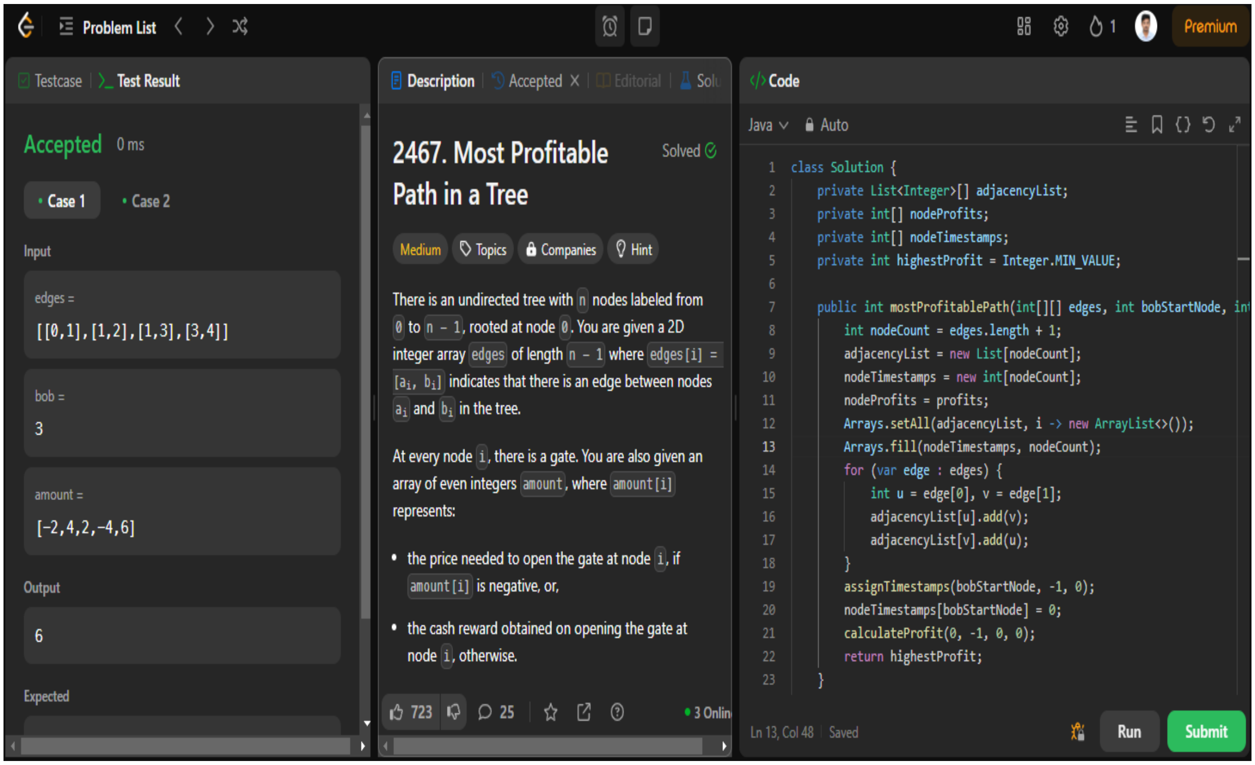
Maximum FuelCost:



Approach:

The algorithm calculates the minimum fuel cost required to transport people across a network of cities connected by roads, where each vehicle can carry a maximum of seats people. It represents the cities and roads as a graph and uses a Depth-First Search (DFS) to explore the cities. For each city, it recursively calculates the number of trips needed to transport people to its neighbors. The number of trips is determined by dividing the number of people in each subtree by the vehicle's capacity, and the fuel cost accumulates by summing the trips across all cities. The final result is the total fuel cost required to transport all people efficiently.

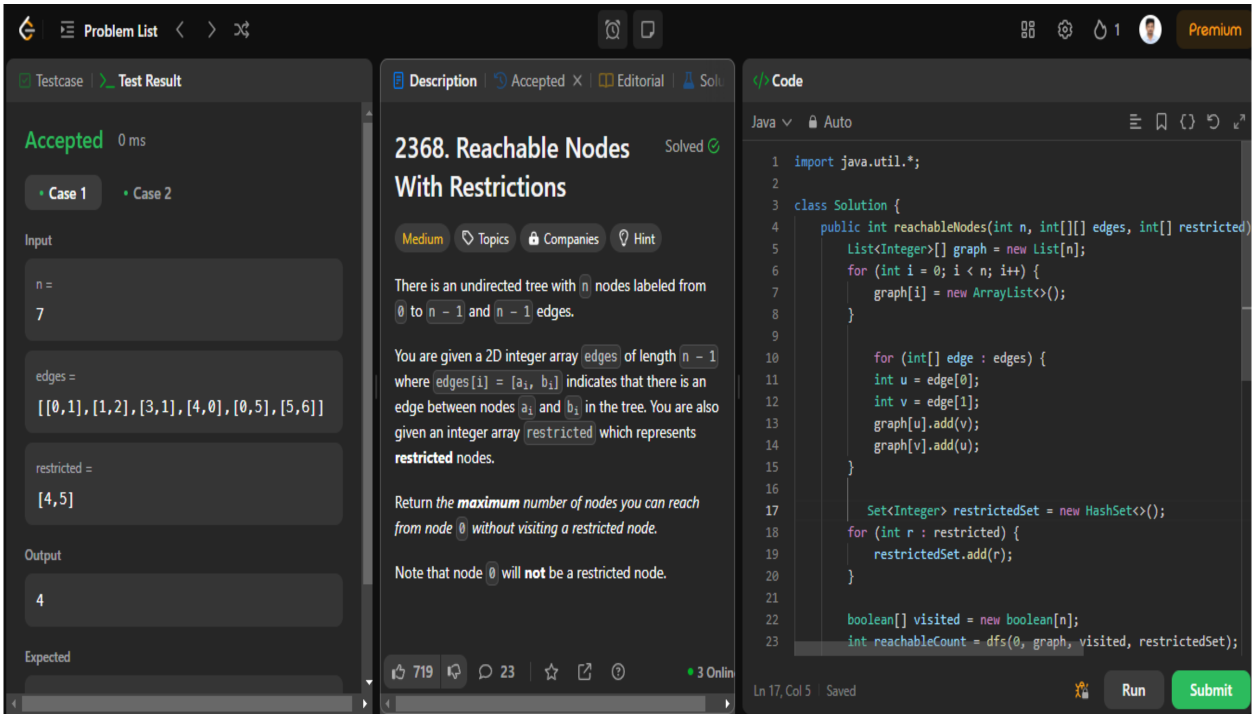
Most Profitable Path in a Tree:



Approach:

This algorithm finds the most profitable path in a tree of cities, starting from the root city (node 0). It considers both the main traveler and Bob, who starts from a different node, with Bob's arrival times affecting the profit. The graph is represented using an adjacency list, and each city has an associated profit. The algorithm uses a depth-first search (DFS) to compute the earliest time Bob reaches each city and adjusts the profit accordingly: if the main traveler arrives before Bob, the full profit is added; if Bob arrives first, the profit is halved. The algorithm recursively explores the graph, accumulating profits and tracking the highest possible profit along the way.

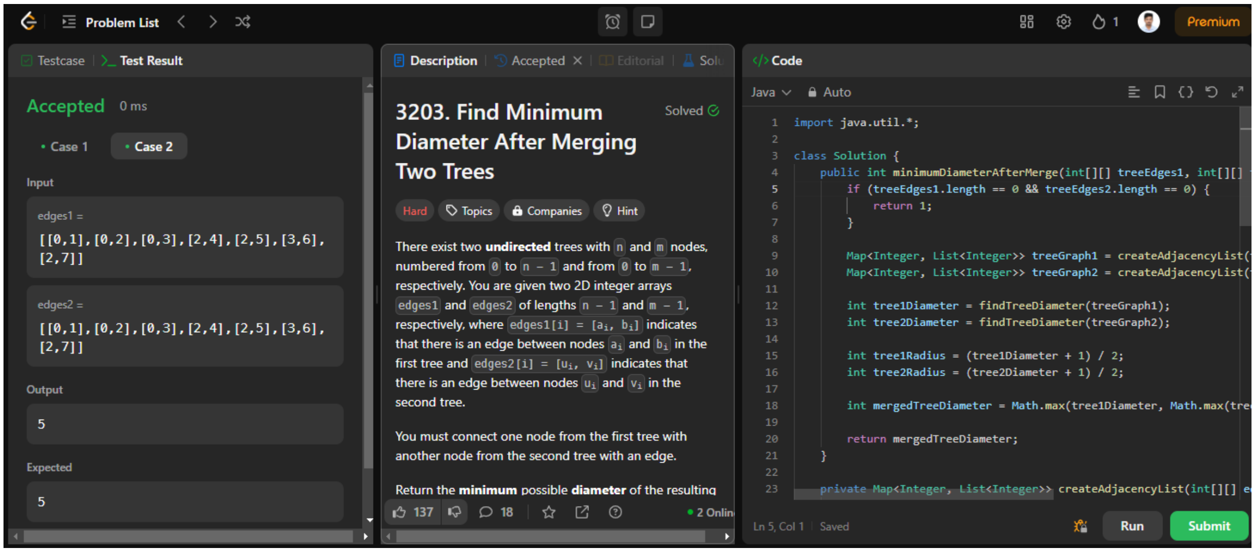
Reachable Nodes with Restrictions:



Approach:

This algorithm calculates the number of reachable nodes in a graph, starting from node 0, while avoiding certain restricted nodes. The graph is represented by an adjacency list where each node has a list of its connected neighbors. The algorithm first builds the graph and converts the list of restricted nodes into a set for fast lookup. Then, it uses Depth-First Search (DFS) to explore the graph from node 0. As it traverses the graph, it avoids visiting restricted nodes and ensures each node is visited only once by marking them as visited. For every node that can be reached (and isn't restricted), it increments a counter. The algorithm returns the total count of reachable nodes from the starting node (0).

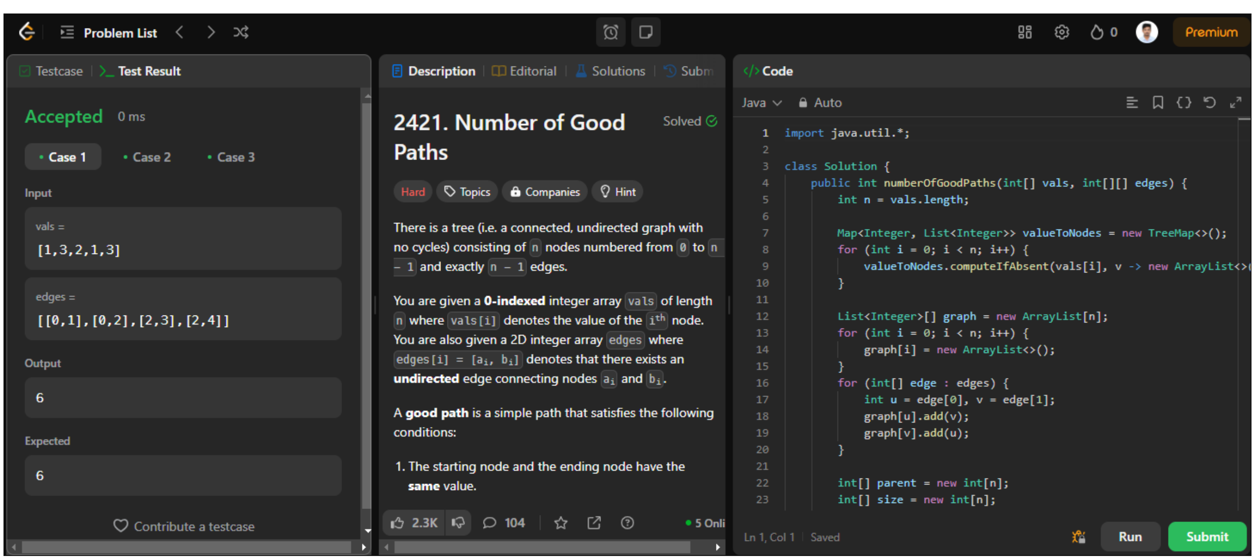
Find Minimum Diameter after Merging two trees:



Approach:

This algorithm calculates the minimum possible diameter of a merged tree formed by connecting two given trees. It starts by constructing adjacency lists for each tree, then calculates the diameter of each tree by finding the longest path between any two nodes using a two-step BFS approach. The diameter of each tree is used to compute the radius, which is half the diameter, rounded up. Finally, the algorithm merges the two trees by adding an edge between them and calculates the new diameter as the maximum of the individual tree diameters or the sum of their radii plus one. This approach ensures that the resulting tree has the smallest possible diameter after the merge.

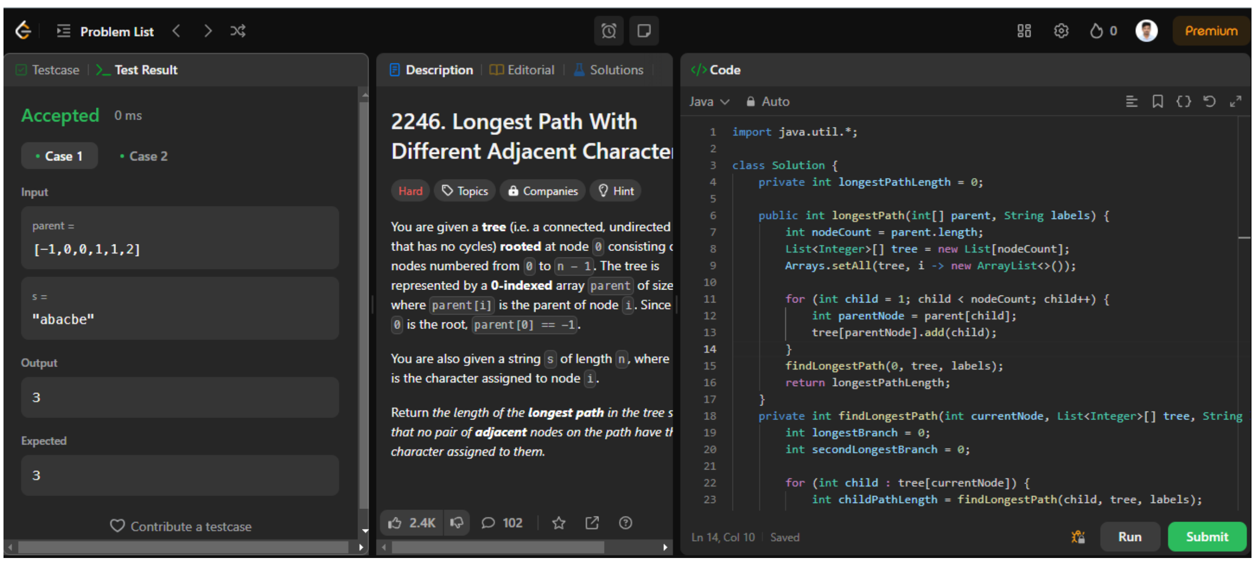
Number Of Good Paths:



Approach:

This solution aims to calculate the number of "good paths" in a graph, where a good path is defined as a path where the values of the nodes along the path are non-decreasing. The approach first groups nodes by their values, then processes them in increasing order of node values. For each value, the algorithm connects nodes that have a smaller or equal value to their neighbors using a union-find (disjoint-set) data structure. This allows us to efficiently track connected components of nodes that are part of a valid path. After processing the nodes with each value, the algorithm counts the number of good paths in each component and accumulates them. The number of good paths in a component of size k is given by k * (k + 1) / 2. The union-find operations help in managing the dynamic connectivity of nodes as the graph is processed by node values.

Longest Path With Different Adjacent Character:



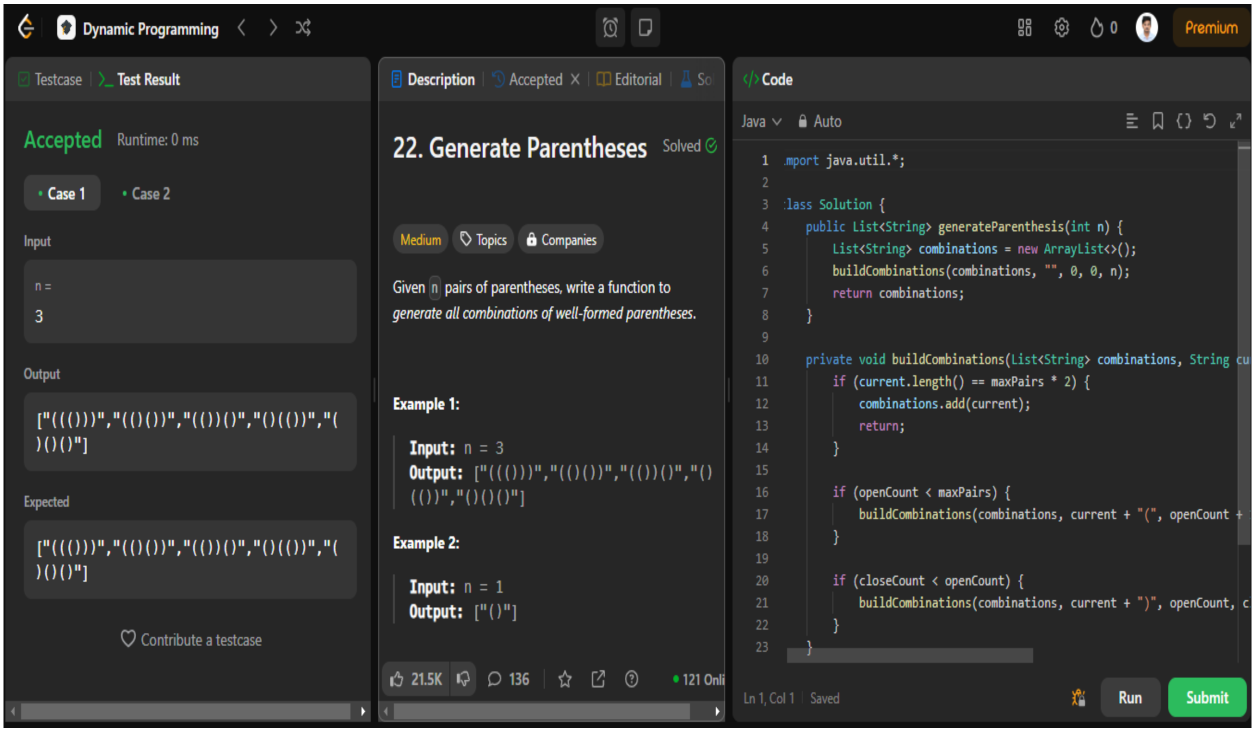
Approach:

This solution computes the longest path in a tree, where a valid path is one that only includes nodes with different labels. The tree is represented by an array parent where each element specifies the parent of a node. The longestPath method builds an adjacency list tree to represent the tree structure and then calls the helper function findLongestPath starting from the root node.

The helper function recursively traverses the tree, calculating the longest path through each node. For each node, it compares the labels of the current node and its children. If the labels are different, it updates the longest and second-longest branches found among the children, ensuring that only paths with different labels are considered. The result for each node is the sum of the two longest branches, plus one for the current node. The longestPathLength variable is updated as the maximum of the current longest path and the calculated path for each node.

Dynamic Programming

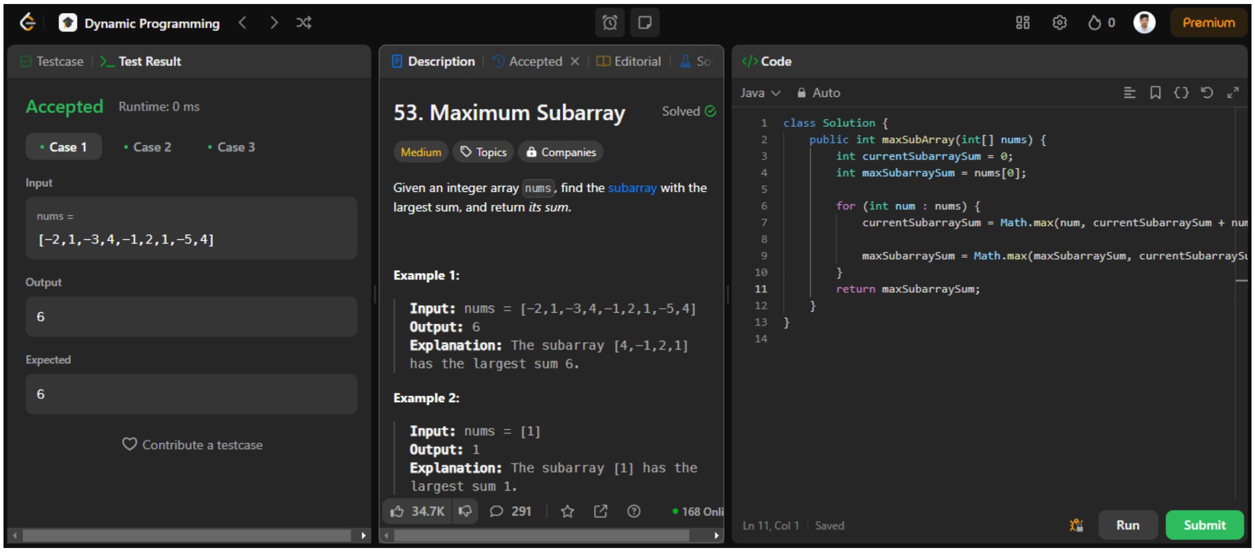
Generate Parentheses



Approach:

This solution generates all possible valid combinations of parentheses for a given number n of pairs. The process starts by recursively building combinations of parentheses. At each step, the algorithm adds an open parenthesis if the number of open parentheses is less than n and adds a close parenthesis if it's allowed (i.e., there are more open parentheses than close ones). Once a combination reaches the length of 2 * n, it's considered valid and added to the result list. By ensuring that the parentheses are balanced at every step, this approach efficiently generates all well-formed parentheses combinations.

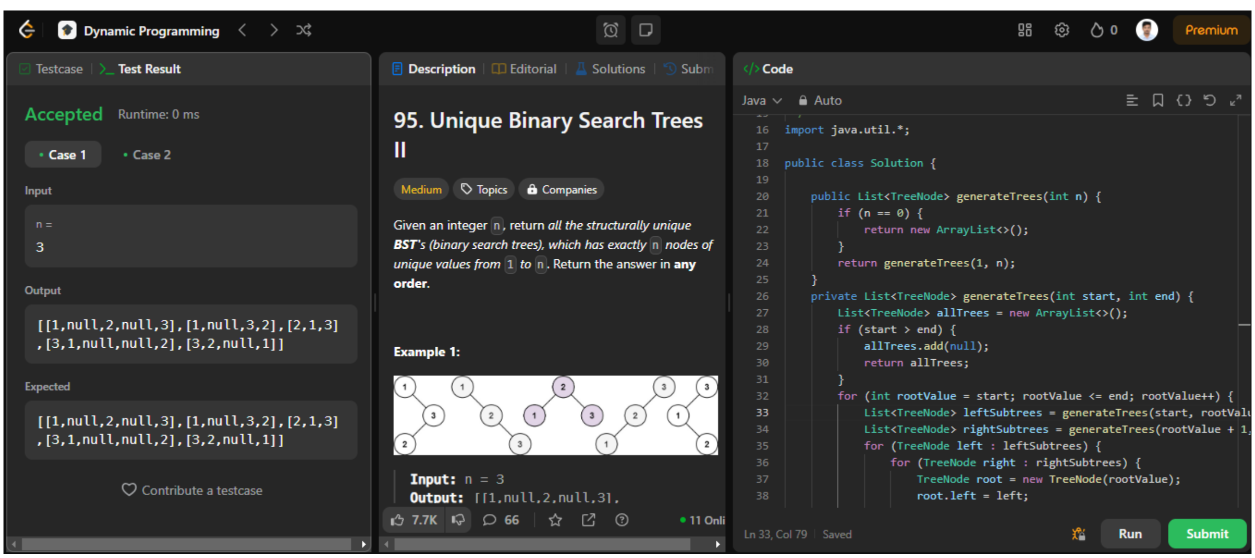
Maximum Subarray:



Approach:

This solution uses Kadane's Algorithm to efficiently find the maximum sum of a contiguous subarray within an integer array. It works by iterating through each element and deciding whether to start a new subarray with the current element or to add the current element to the existing subarray. The algorithm keeps track of the maximum sum encountered so far, updating it as needed. By the end of the loop, it returns the largest sum found. This approach ensures that the solution runs in linear time, making it both fast and effective for this problem.

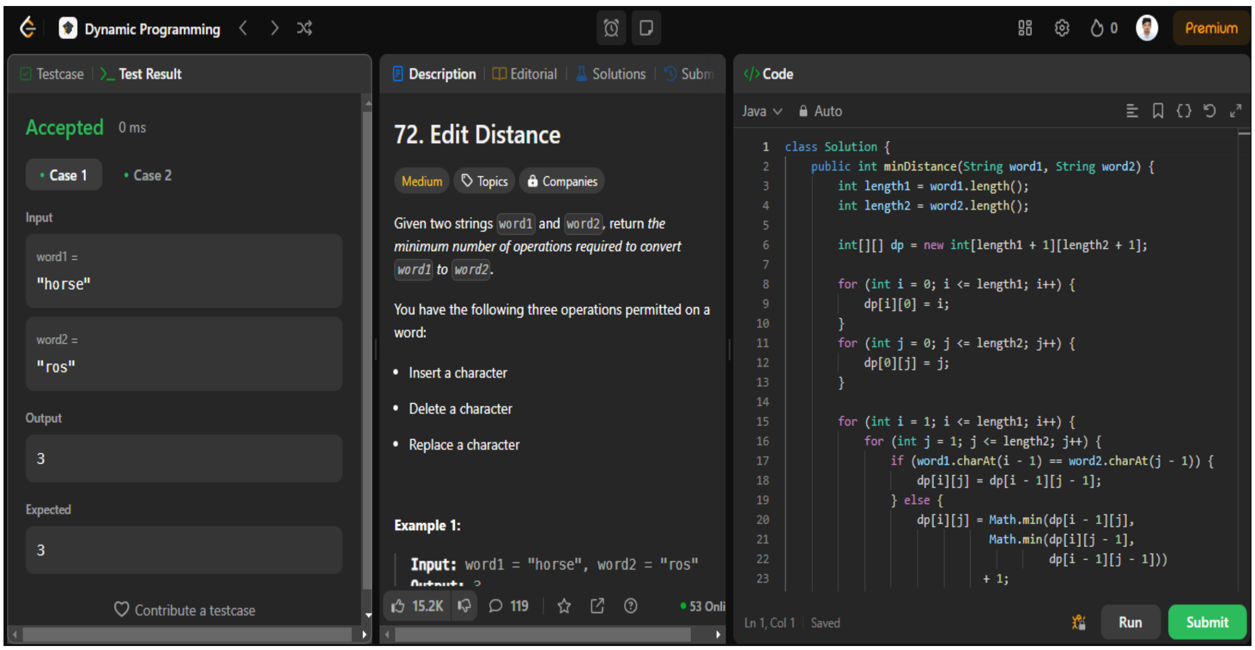
Unique Binary Search Trees:



Approach:

The goal is to construct all unique binary search trees (BSTs) that may be built with integers ranging from 1 to n. The method entails iteratively selecting each integer as the root within the specified range. It produces all feasible left and right subtrees for each root by combining the remaining integers. Once the subtrees have been produced, they are joined with the current root to create various BSTs. If n equals zero, the solution produces an empty list since no trees can be created. The output is a list of all conceivable unique BSTs.

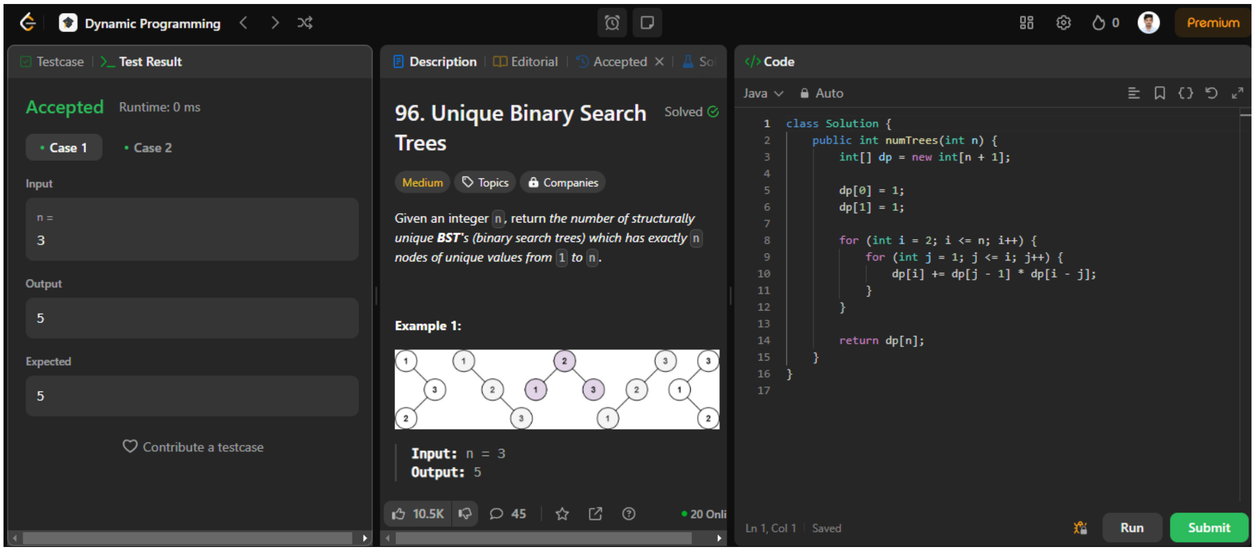
Edit Distance:



Approach:

The aim is to determine the smallest number of operations needed to convert one string to another, where the allowable operations are insertions, deletions, and replacements. The solution use dynamic programming to generate a table, with each entry representing the smallest number of operations required to convert one substring of the first word into a substring of the second word. Starting with base cases (empty strings), the method checks characters at each point, updating the table using the three operations. The final entry in the table specifies the minimum number of operations required for the whole transformation.

Unique Binary Search Trees:



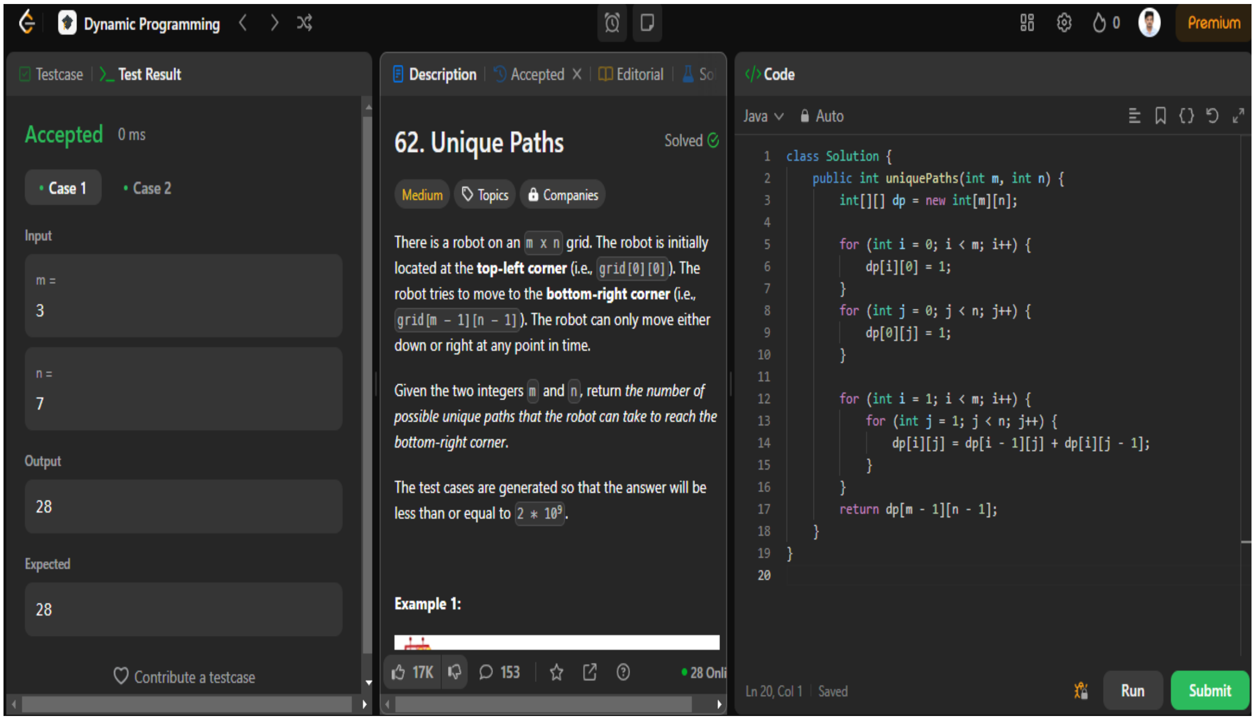
Approach:

This approach uses dynamic programming to determine the number of distinct Binary Sear ch Trees (BSTs) that may be created with n nodes.The goal is to take advantage of the fact th at any node i may serve as the root of a BST, with nodes to its left and right forming smaller subtrees.The dynamic programming array dp[i] contains the number of distinct BSTs that m ay be created with i nodes.Beginning with base instances where dp[0] and dp[1] are both 1 (representing one empty tree and one tree with a single node, respectively), the method rep eatedly computes the number of trees for higher n.
For each i, it evaluates all potential root nodes j and adds the products of the number of left and right subtrees, resulting in the total number of unique BSTs. The final result is stored in dp[n].

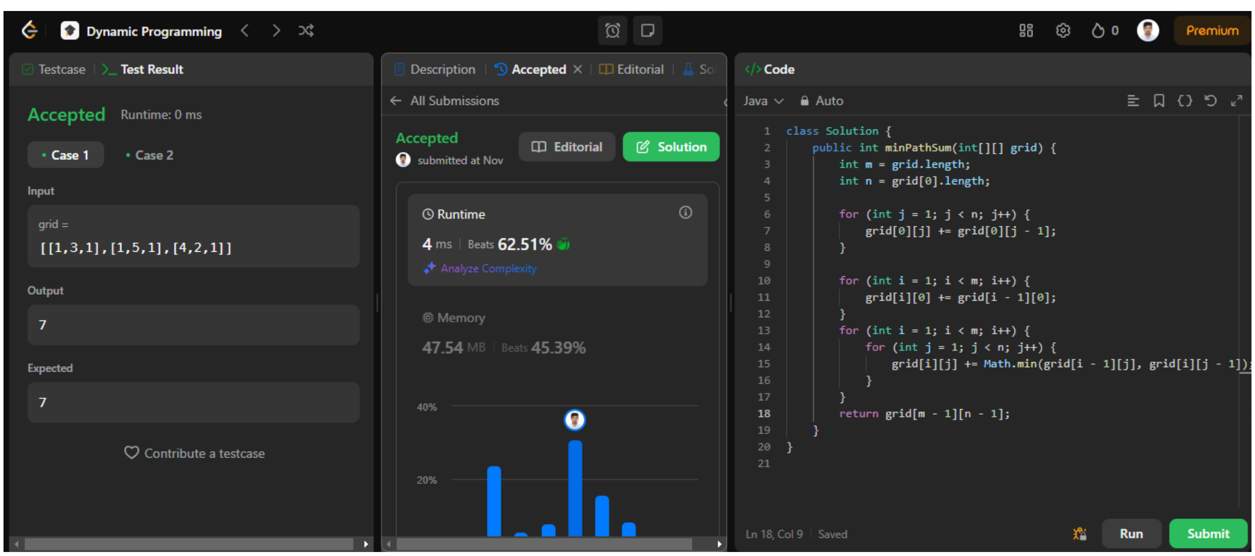
Unique Paths:

Approach:

This solution calculates the number of unique paths in a grid from the top-left corner to the bottom-right corner, where you can only move either down or right. It uses dynamic programming, starting by initializing the first row and the first column to 1, since there's only one way to move along those edges — either by always going right for the first row or always going down for the first column. Then, for each remaining cell in the grid, the number of ways to reach that cell is the sum of the number of ways to reach the cell directly above it and the cell directly to the left of it. This way, the solution builds up the total number of ways to reach each cell, and finally, the number of unique paths is found at the bottom-right corner of the grid.



Minimum Path Sum:



Approach:

This approach calculates the minimal path total from a grid's top-left corner to the bottom-right corner, with each step moving either right or down. The solution is built using dynamic programming. First, it updates the first row and column since there is only one method to access each cell in those areas: go right for the row and down for the column. The value of each remaining cell is then updated by adding the current cell's value to the least of the values from the cell above it or the cell to its left, ensuring that the route sum is reduced at each step. Finally, the minimal path total is in the bottom-right corner.