Experiment-6

Student Name: Sabir Ali
UID: 22BET10033
Branch: BE-CSE
Section/Group: TPP_IOT_701-A
Semester: 06
Date of Performance: 27-02-2025

Subject Name: AP LAB-II
Subject Code: 22ITP-351

1. **Aim:**
   a. **Maximum Depth of Binary Tree.**
   b. **Binary Tree Level Order Traversal.**
   c. **Convert Sorted Array to Binary Search Tree.**

2. **Introduction to Binary Trees Problem:**
   A Binary Search Tree (BST) is a special type of binary tree where:
   - The left subtree contains values less than the node.
   - The right subtree contains values greater than the node.

   This structure enables efficient searching, insertion, and deletion in O(log N) time on average, making BSTs essential in algorithms, databases, and real-world applications.

3. **Implementation/Code:**

A. Maximum Depth of Binary Tree

```
class Solution { public:
   int maxDepth(TreeNode* root) {           if
(root == nullptr) return 0;          int leftDepth
= maxDepth(root->left);         int rightDepth
= maxDepth(root->right);         return
max(leftDepth, rightDepth) + 1;

    }
};
```

B. Binary Tree Level Order Traversal

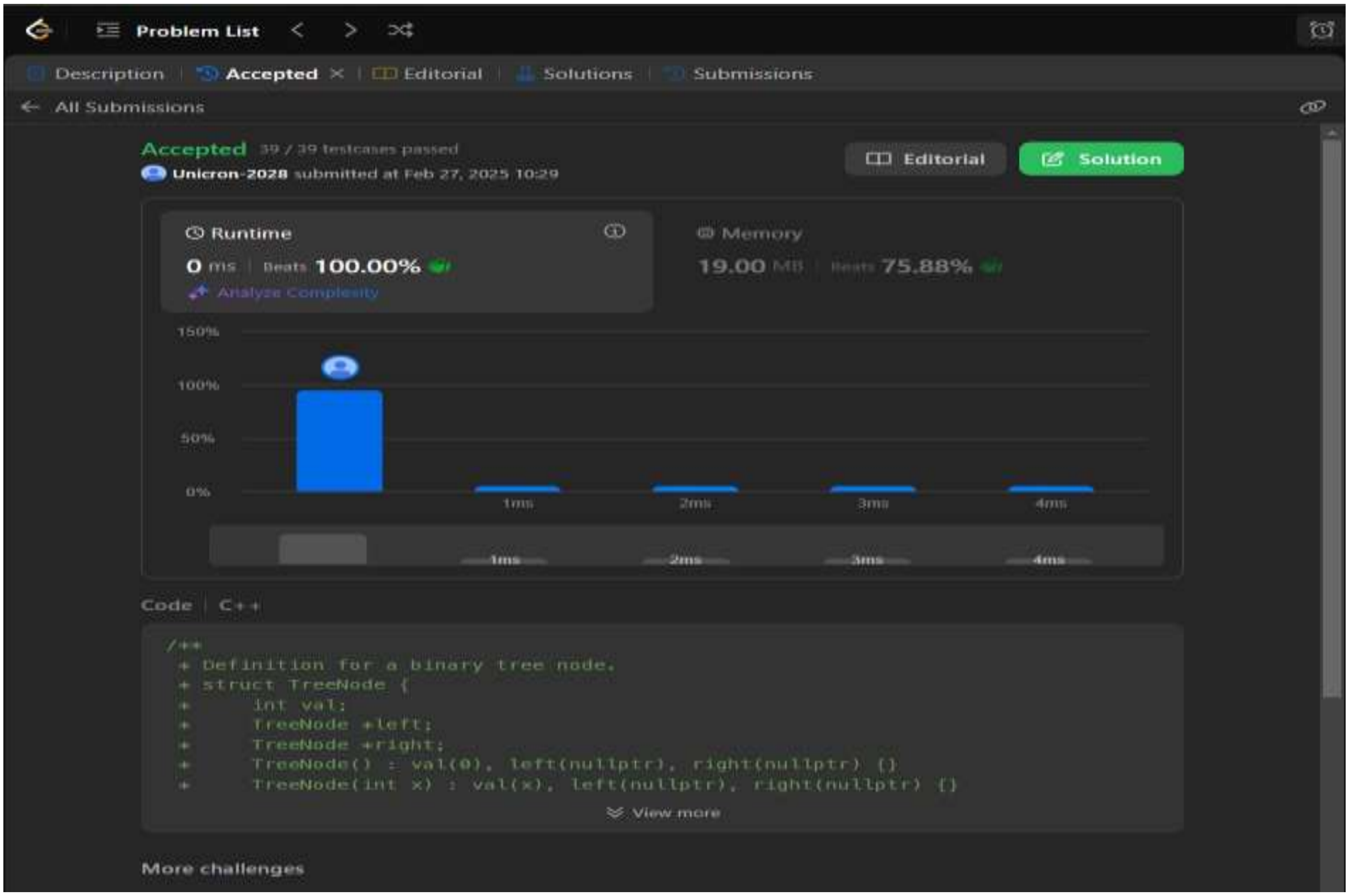
```
class Solution { public:     vector<vector<int>>
levelOrder(TreeNode* root) {
vector<vector<int>> result;
        if (!root) return result; // If tree is empty, return empty
       vector

        queue<TreeNode*> q;
q.push(root);

        while (!q.empty()) {
            int size = q.size(); // Number of nodes at current level
vector<int> level;

            for (int i = 0; i < size; i++) {
TreeNode* node = q.front();                q.pop();
                level.push_back(node->val);

                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }

            result.push_back(level);
        }

        return result;
}
       };
```

C. Convert Sorted Array to Binary Search Tree

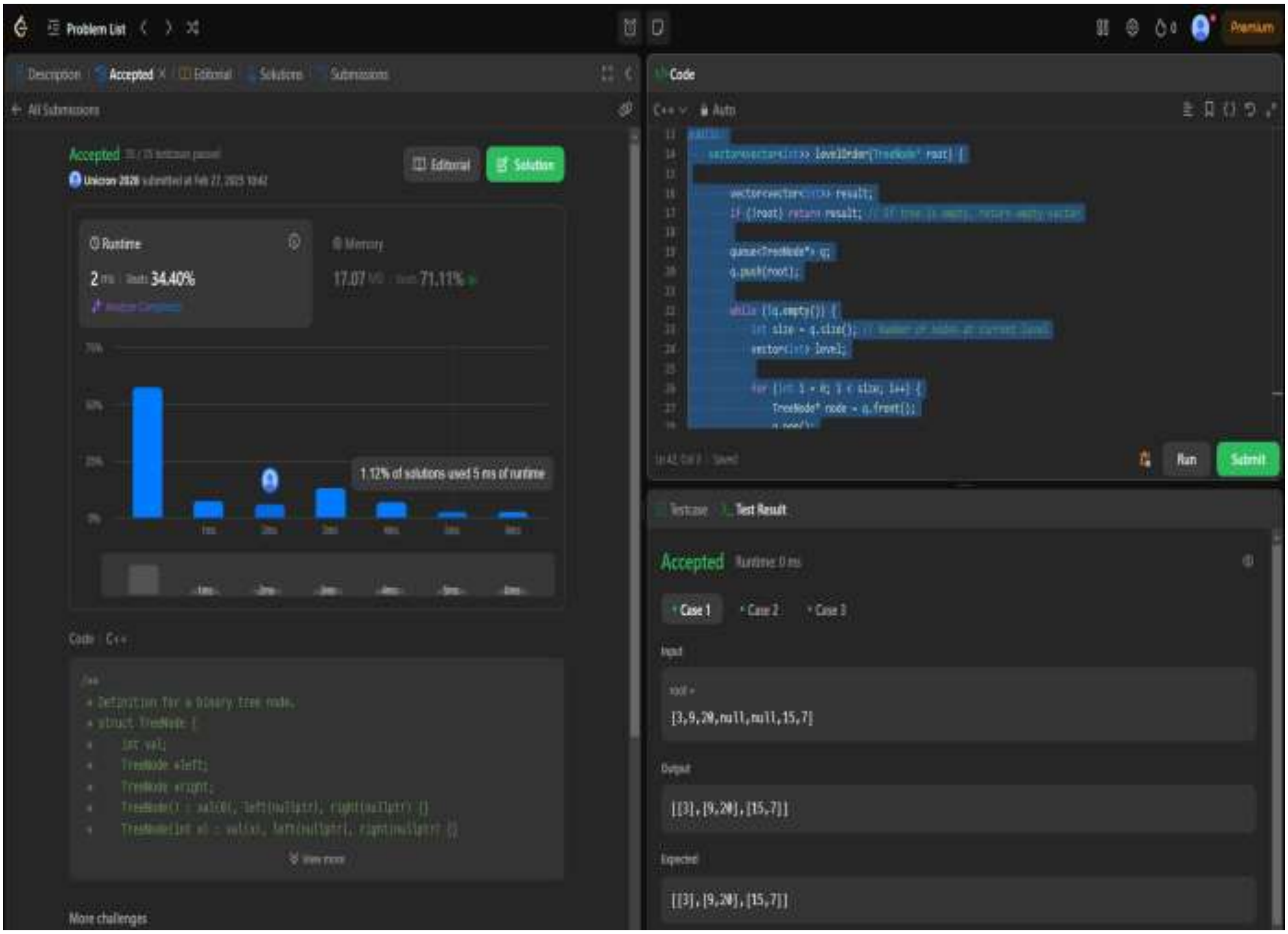
```
class Solution { public:
   TreeNode* sortedArrayToBST(vector<int>& nums) {
return buildBST(nums, 0, nums.size() - 1);
   }

private:
   TreeNode* buildBST(vector<int>& nums, int left, int right) {
if (left > right) return nullptr; // Base case

      int mid = left + (right - left) / 2; // Find middle element
      TreeNode* root = new TreeNode(nums[mid]); // Create root node

      root->left = buildBST(nums, left, mid - 1);  // Recursively build left
     subtree
      root->right = buildBST(nums, mid + 1, right); // Recursively build right
     subtree

      return root;
   } };
```

4. Output

A. Maximum Depth of Binary Tree

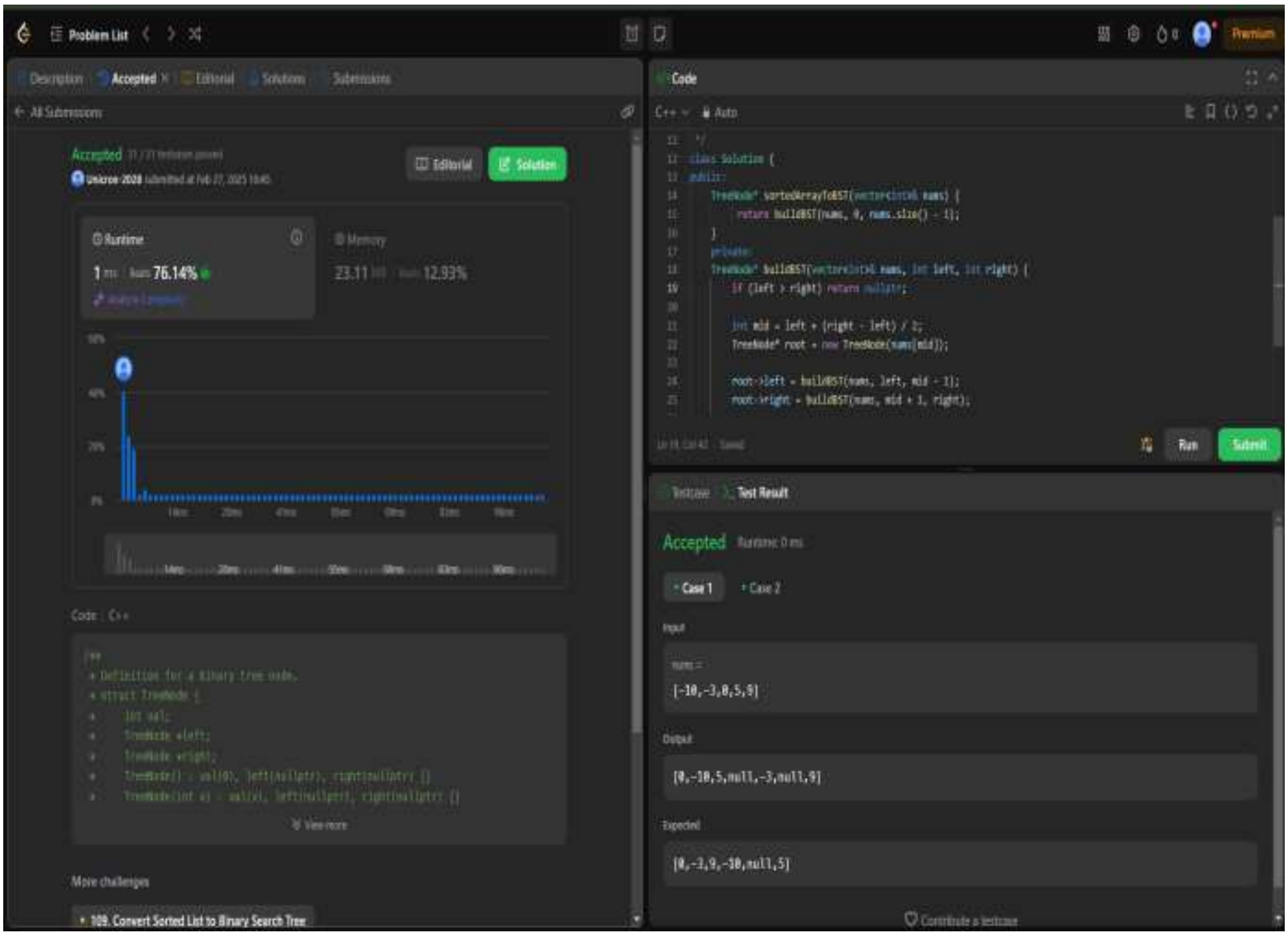


B. Binary Tree Level Order Traversal



Problem List
Description
Accepted
Editorial
Solutions
Submissions
All Submissions
Accepted
Editorial
Solution
Runtime
2 ms
34.40%
Memory
17.07
71.11%
1.12% of solutions used 5 ms of runtime
Code
C++
More challenges
Code
Auto
vector<vector<int>> levelOrder(TreeNode* root) {
vector<vector<int>> result;
queue<TreeNode*> q;
q.push(root);
vector<int> level;
TreeNode* node = q.front();
Run
Submit
Testcase
Test Result
Accepted
Case 1
Case 2
Case 3
Input
root =
[3,9,20,null,null,15,7]
Output
[[3],[9,20],[15,7]]
Expected
[[3],[9,20],[15,7]]
Premium

C. Convert Sorted Array to Binary Search Tree



5. Learning Outcomes:

Understanding Binary Search Trees (BSTs)

- Learn how BSTs maintain the ordering property (left < root < right).
- Understand the importance of height-balancing in BSTs for optimal performance.

Divide and Conquer Approach

- Learn how recursion can be used to break a problem into smaller subproblems. Understand how selecting the middle element ensures balanced tree formation.

Recursive Tree Construction

- Develop the ability to construct trees dynamically using recursive functions.
- Learn how to create and link TreeNode objects in C++.

Time & Space Complexity Analysis

- Understand why O(N) time complexity is achieved (each element processed once). Learn about O(log N) space complexity due to recursion depth.

Real-World Applications

- BSTs are used in databases, search operations, and hierarchical data storage.
- Understanding balanced trees helps in optimizing search, insert, and delete operations.