Assignment-10

Student Name: Anjali
UID: 22BCS10665
Branch: BE- CSE
Section/Group:22BCS_IOT-614/B
Semester: 6
Date of Performance:17-04-25
Subject Name: AP LAB-II
Subject Code: 22CSP-351

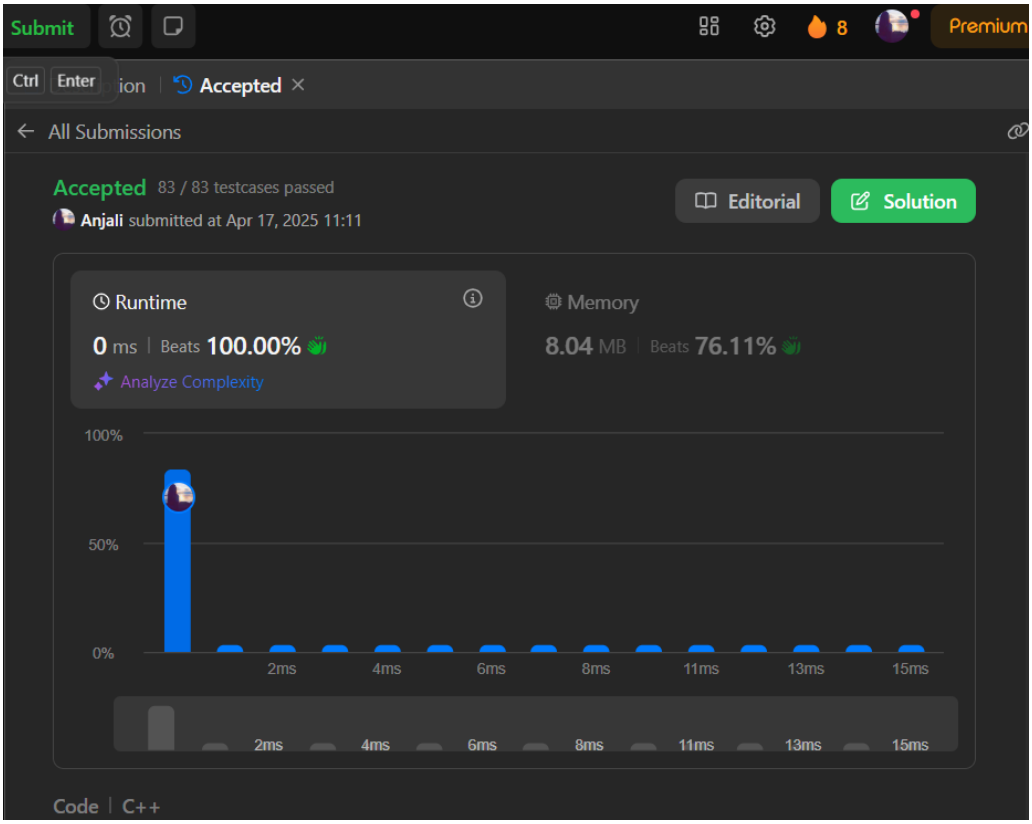
PROBLEM-1: Valid Parenthesis String

- **Objective:** Given a string s containing only three types of characters: '(', ')' and '*', return true *if* s *is valid*.
  The following rules define a valid string:
  Any left parenthesis '(' must have a corresponding right parenthesis ')'.
  Any right parenthesis ')' must have a corresponding left parenthesis '('.
  Left parenthesis '(' must go before the corresponding right parenthesis ')'.
  '*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string "".

- **Implementation/Code:**

```
class Solution {
public:
    bool checkValidString(string s) {
        int low = 0, high = 0;

        for (char c : s) {
            if (c == '(') {
                low++;
                high++;
            } else if (c == ')') {
                low = max(0, low - 1);
                high--;
            } else { // c == '*'
                low = max(0, low - 1);
                high++;
            }

            if (high < 0) return false;
        }

        return low == 0;
    }
};
```

➢ **Output:**



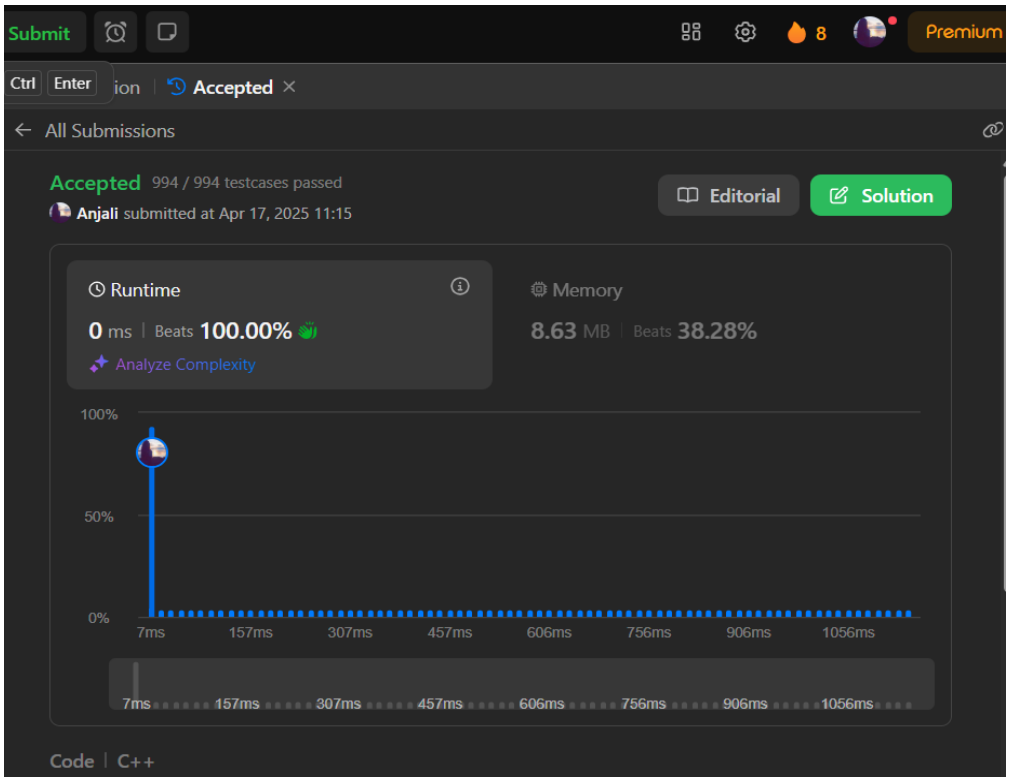
PROBLEM-2: Divide Two Integers

➢ **Objective:** Given two integers dividend and divisor, divide two integers without using multiplication, division, and mod operator.
The integer division should truncate toward zero, which means losing its fractional part. For example, 8.345 would be truncated to 8, and -2.7335 would be truncated to -2.
Return *the quotient after dividing* dividend *by* divisor.
Note: Assume we are dealing with an environment that could only store integers within the 32-bit signed integer range: $[-2^{31}, 2^{31} - 1]$. For this problem, if the quotient is strictly greater than 2^{31} - 1, then return 2^{31} - 1, and if the quotient is strictly less than -2^{31}, then return -2^{31}.

- **Implementation/Code:**

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (dividend == INT_MIN && divisor == -1) return INT_MAX;
        if (divisor == INT_MIN) return (dividend == INT_MIN) ? 1 : 0;
        if (dividend == 0) return 0;
        bool isNegative = (dividend < 0) ^ (divisor < 0);
        long long a = abs((long long)dividend);
        long long b = abs((long long)divisor);
        long long result = 0;
        while (a >= b) {
            long long temp = b, multiple = 1;
            while ((temp << 1) <= a) {
                temp <<= 1;
                multiple <<= 1;
            }
            a -= temp;
            result += multiple;
        }
        return isNegative ? -result : result;
    }
     };
```

- **Output:**



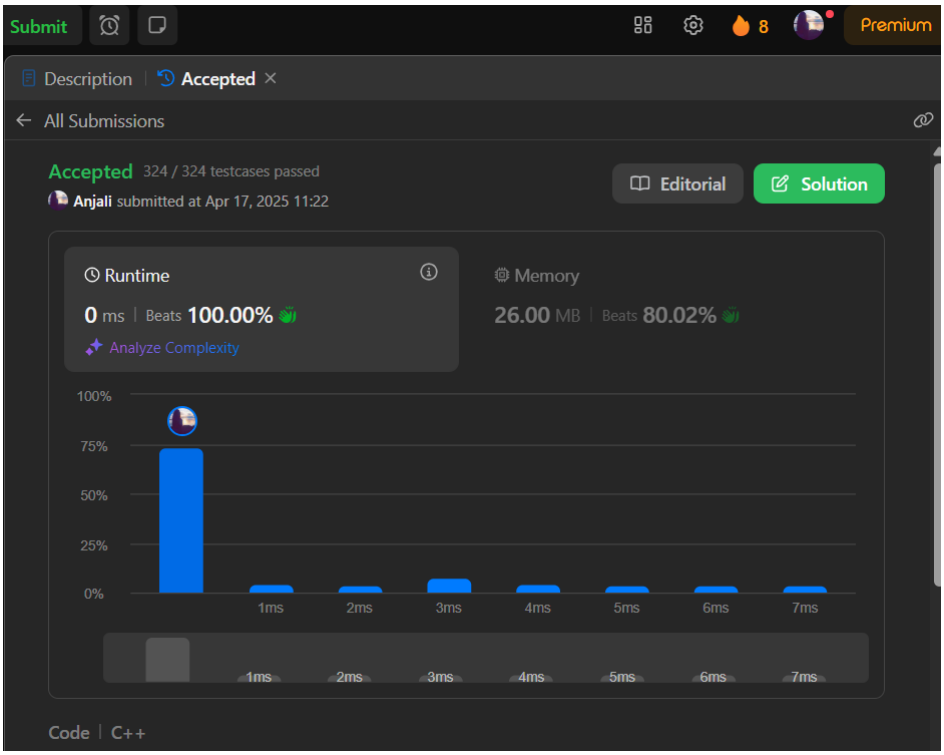
PROBLEM-3: Trapping Rain Water

- **Objective:** Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

- **Implementation/Code:**

```
class Solution {
public:
    int trap(vector<int>& height) {
        if (height.empty()) return 0;

        int left = 0, right = height.size() - 1;
        int leftMax = 0, rightMax = 0, waterTrapped = 0;

        while (left < right) {
            if (height[left] < height[right]) {
                if (height[left] >= leftMax) {
                    leftMax = height[left];
                } else {
                    waterTrapped += leftMax - height[left];
                }
                left++;
            } else {
                if (height[right] >= rightMax) {
                    rightMax = height[right];
                } else {
                    waterTrapped += rightMax - height[right];
                }
                right--;
            }
        }

        return waterTrapped;
    }
};
```

➢ **Output:**



PROBLEM-4: LRU Cache

➢ **Objective:** Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.
Implement the LRUCache class:
LRUCache(int capacity) Initialize the LRU cache with positive size capacity.
int get(int key) Return the value of the key if the key exists, otherwise return -1.
void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.
The functions get and put must each run in O(1) average time complexity.

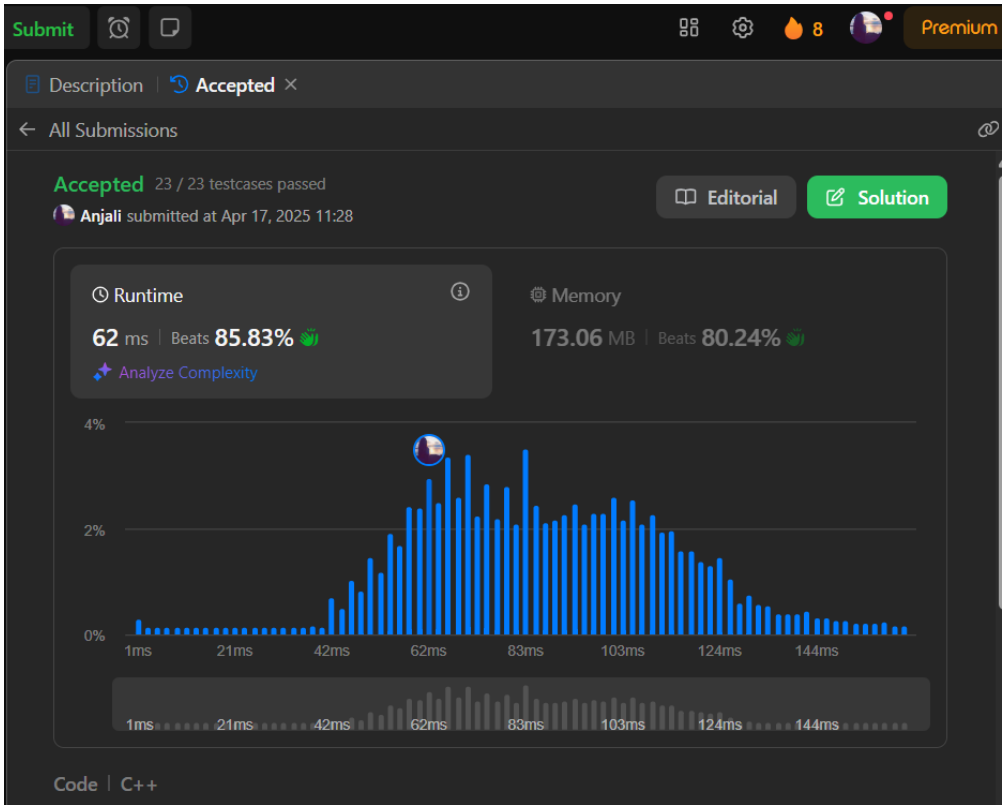
➢ **Implementation/Code:**

```
class LRUCache {
private:
    struct Node {
        int key, value;
        Node* prev;
        Node* next;
        Node(int k, int v) : key(k), value(v), prev(nullptr), next(nullptr) {}
    };
```

```
    unordered_map<int, Node*> cache;
    int capacity;
    Node* head;
    Node* tail;

    void remove(Node* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }

    void insert(Node* node) {
        node->next = head->next;
        node->prev = head;
        head->next->prev = node;
        head->next = node;
    }

public:
    LRUCache(int capacity) {
        this->capacity = capacity;
        head = new Node(0, 0); // dummy head
        tail = new Node(0, 0); // dummy tail
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        if (cache.count(key)) {
            Node* node = cache[key];
            remove(node);
            insert(node);
            return node->value;
        }
        return -1;
    }

    void put(int key, int value) {
        if (cache.count(key)) {
            Node* node = cache[key];
            node->value = value;
            remove(node);
            insert(node);
        } else {
            if (cache.size() == capacity) {
                Node* lru = tail->prev;
                remove(lru);
                cache.erase(lru->key);
                delete lru;
            }
```

```
            Node* newNode = new Node(key, value);
            insert(newNode);
            cache[key] = newNode;
        }
    }
        };
```

- **Output:**



PROBLEM-5: Serialize and Deserialize Binary Tree

- **Objective:** Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.
  Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

- **Implementation/Code:**

```
class Codec {
public:
    string serialize(TreeNode* root) {
        if (!root) return "null,";
        return to_string(root->val) + "," + serialize(root->left) + serialize(root->right);
    }
    TreeNode* deserializeHelper(queue<string>& nodes) {
        string val = nodes.front();
        nodes.pop();
        if (val == "null") return nullptr;
        TreeNode* node = new TreeNode(stoi(val));
        node->left = deserializeHelper(nodes);
        node->right = deserializeHelper(nodes);
        return node;
    }
    TreeNode* deserialize(string data) {
        queue<string> nodes;
        stringstream ss(data);
        string token;
        while (getline(ss, token, ',')) {
            nodes.push(token);
        }
        return deserializeHelper(nodes);
    }
     };
```

- **Output:**