**Name:** Pratistha Tewari

**UID:** 22BCS13955

**Section:** 22BCS\_IOT-614-B

1. **Code: (Print linked list)**

class Solution {

public:

// Function to display the elements of a linked list in same line

void printList(Node \*head) {

Node\* temp = head;

while (temp != nullptr) {

cout << temp->data << " ";

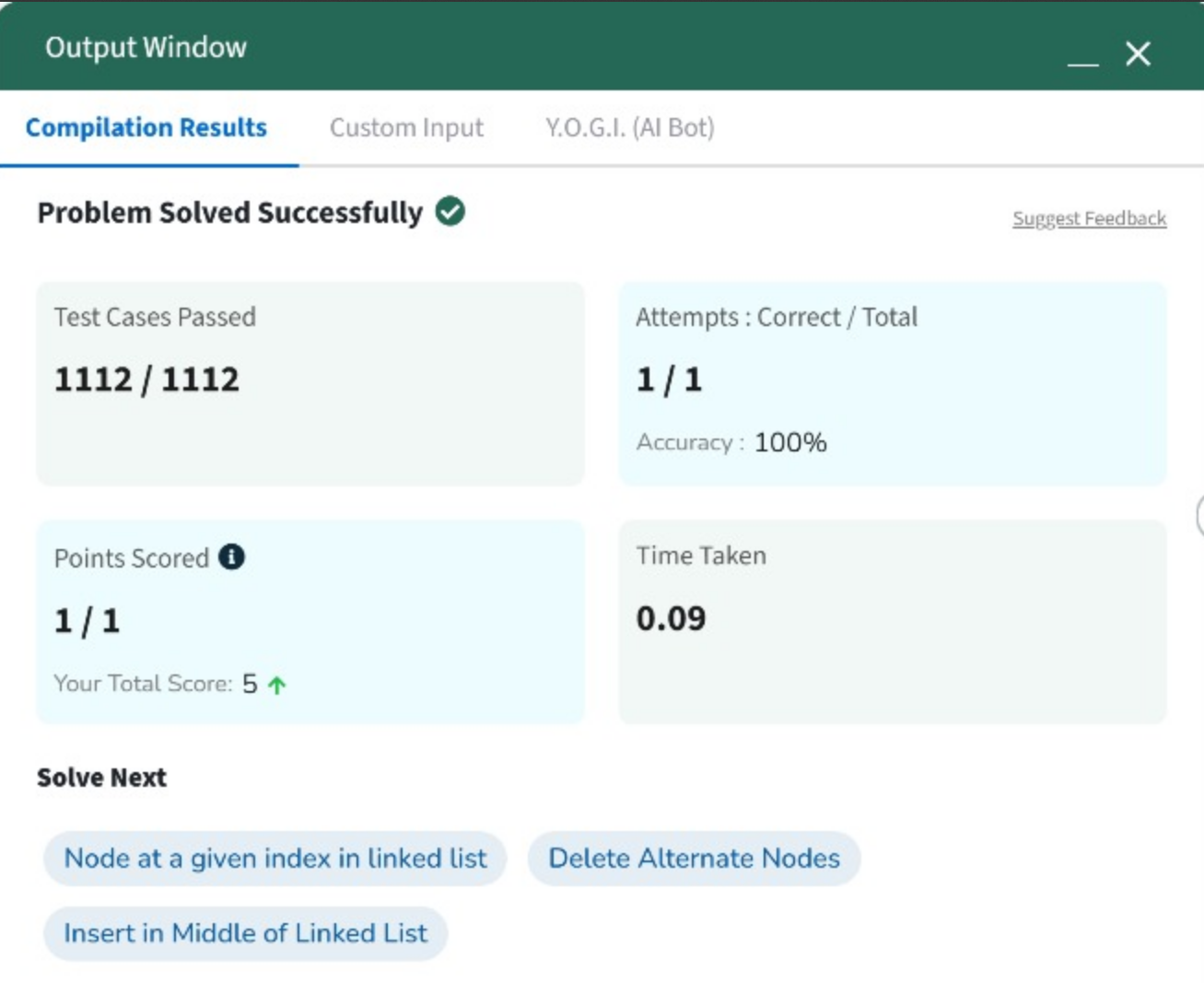
temp = temp->next;

}

}

};

**Output:**

****

1. **Code: (Remove linked list from sorted list)**

class Solution {

public:

ListNode\* deleteDuplicates(ListNode\* head) {

ListNode\* current = head;

while (current != nullptr && current->next != nullptr) {

if (current->val == current->next->val) {

ListNode\* duplicate = current->next;

current->next = current->next->next;

delete duplicate; // Free memory of the removed node

} else {

current = current->next; // Move to the next node

}

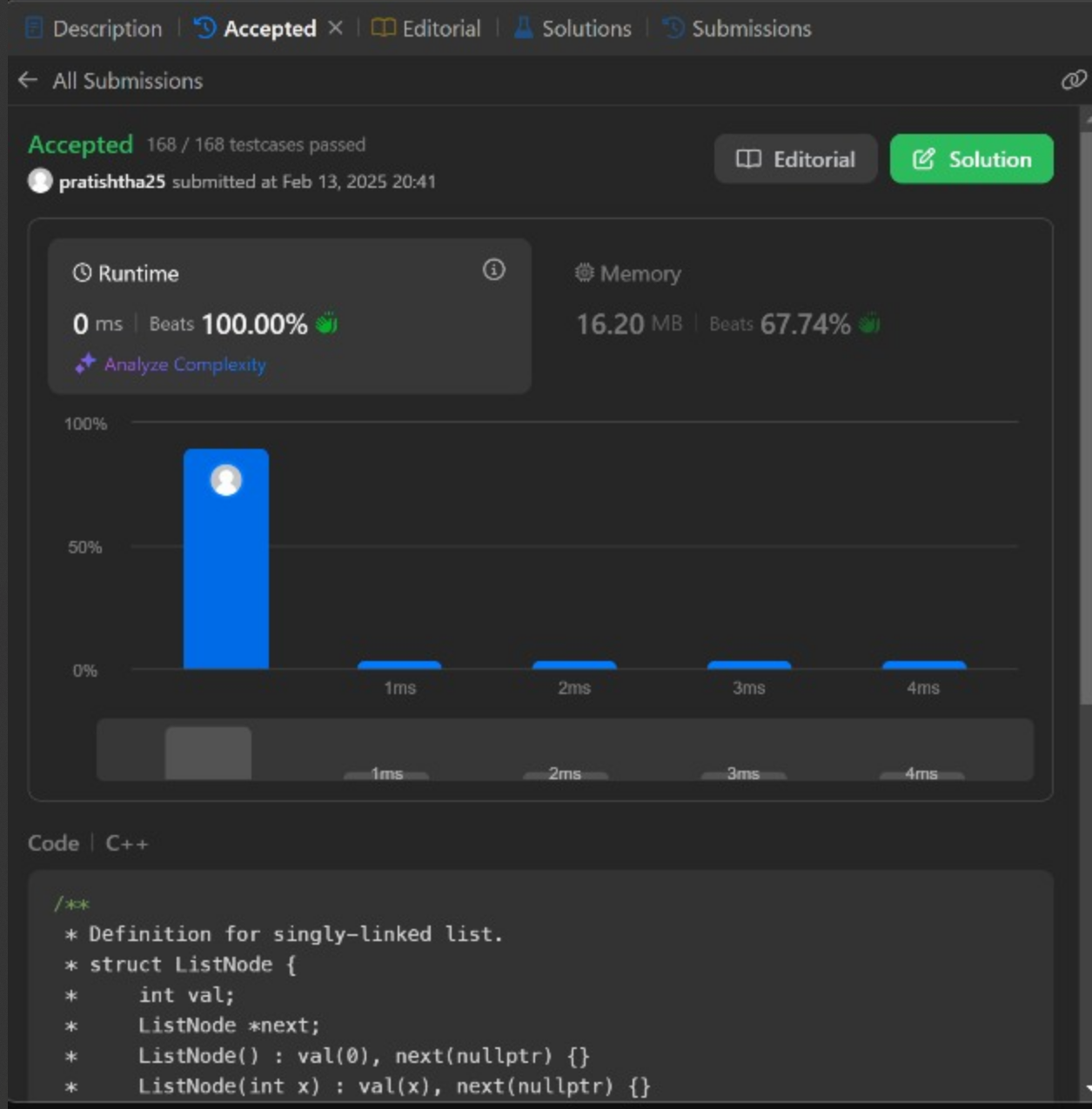
}

return head;

}

};

**OUTPUT:**

****

1. **Code: (Reverse the linked list)**

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

ListNode\* prev = nullptr;

ListNode\* curr = head;

while (curr) {

ListNode\* nextNode = curr->next;

curr->next = prev;

prev = curr;

curr = nextNode;

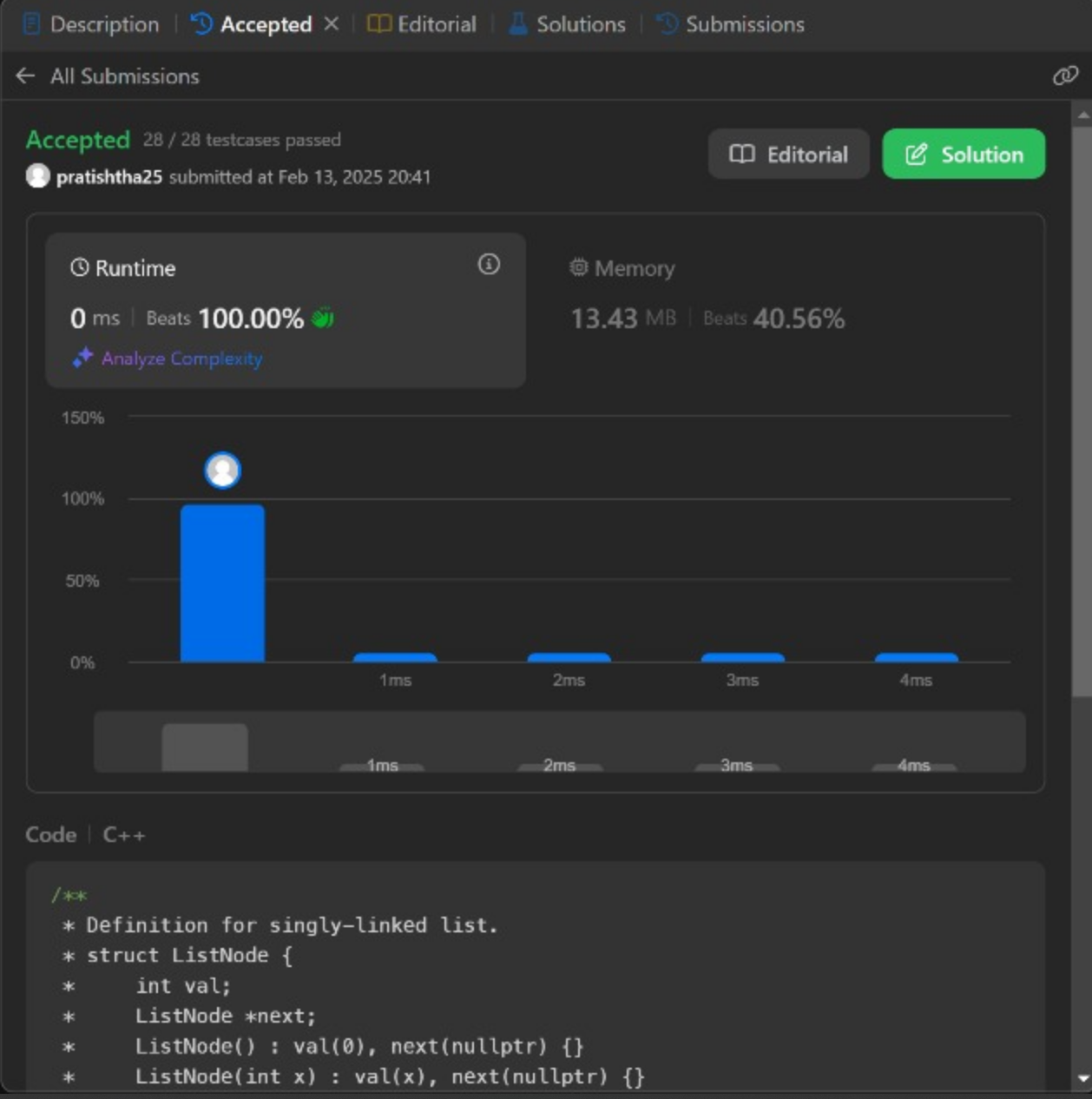
}

return prev;

}

};

**Output:**

****

1. **Code: (Delete middle node of a list)**

class Solution {

public:

ListNode\* deleteMiddle(ListNode\* head) {

// If the list has only one node, return nullptr (empty list)

if (!head || !head->next) {

return nullptr;

}

// Use two pointers: slow and fast

ListNode\* slow = head;

ListNode\* fast = head;

ListNode\* prev = nullptr; // To track the node before the middle node

// Move the fast pointer twice as fast as the slow pointer

while (fast && fast->next) {

prev = slow; // Keep track of the node before slow

slow = slow->next; // Move slow one step

fast = fast->next->next; // Move fast two steps

}

// Remove the middle node

if (prev) {

prev->next = slow->next;

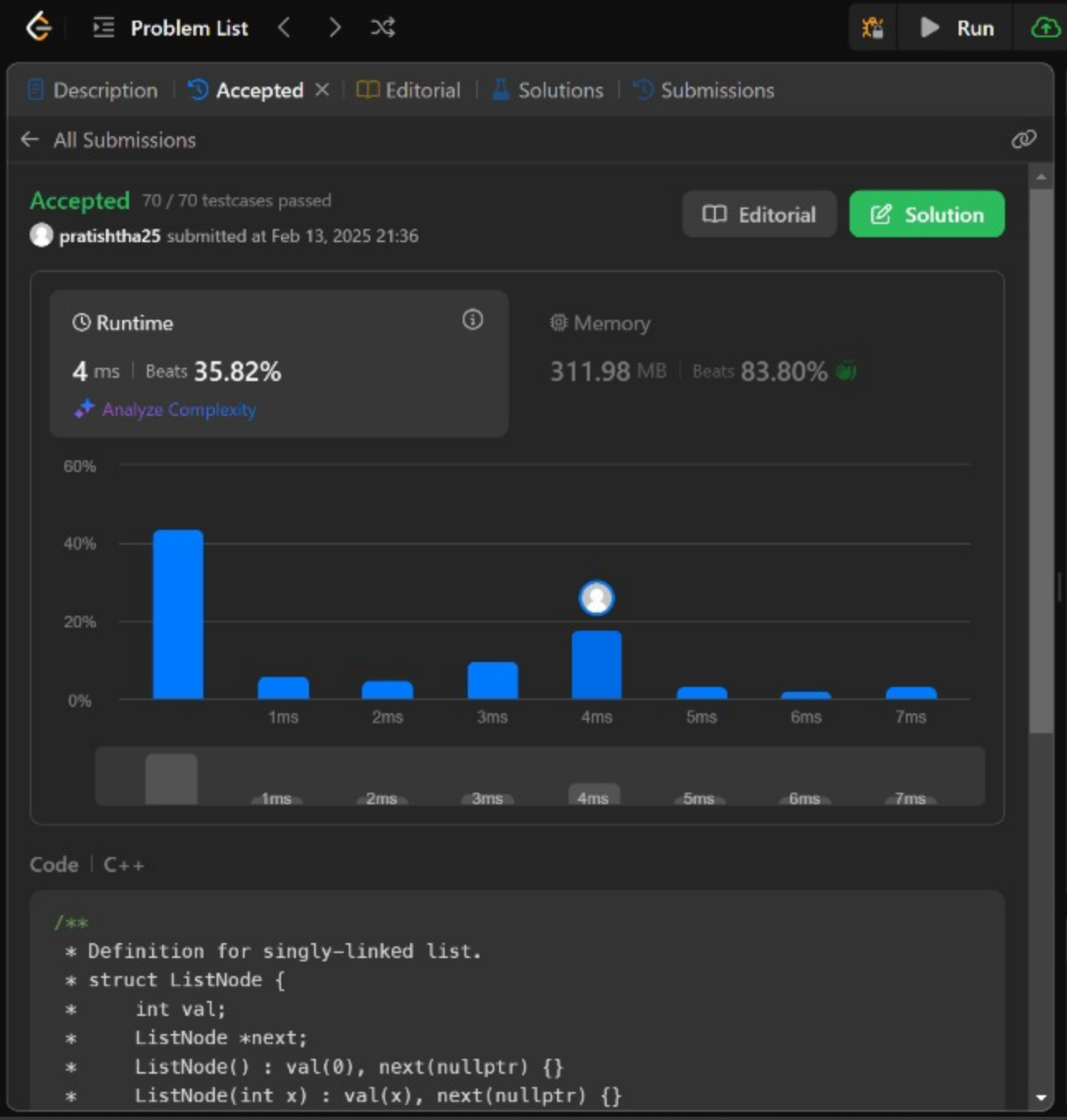
}

return head;

}

};

**Output:**

****

1. **Code: (Merge two sorted list)**

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) {

// Dummy node to simplify handling the head

ListNode dummy(0);

ListNode\* tail = &dummy;

// Traverse both lists

while (list1 && list2) {

if (list1->val < list2->val) {

tail->next = list1;

list1 = list1->next;

} else {

tail->next = list2;

list2 = list2->next;

}

tail = tail->next;

}

// Attach remaining nodes from list1 or list2

if (list1) tail->next = list1;

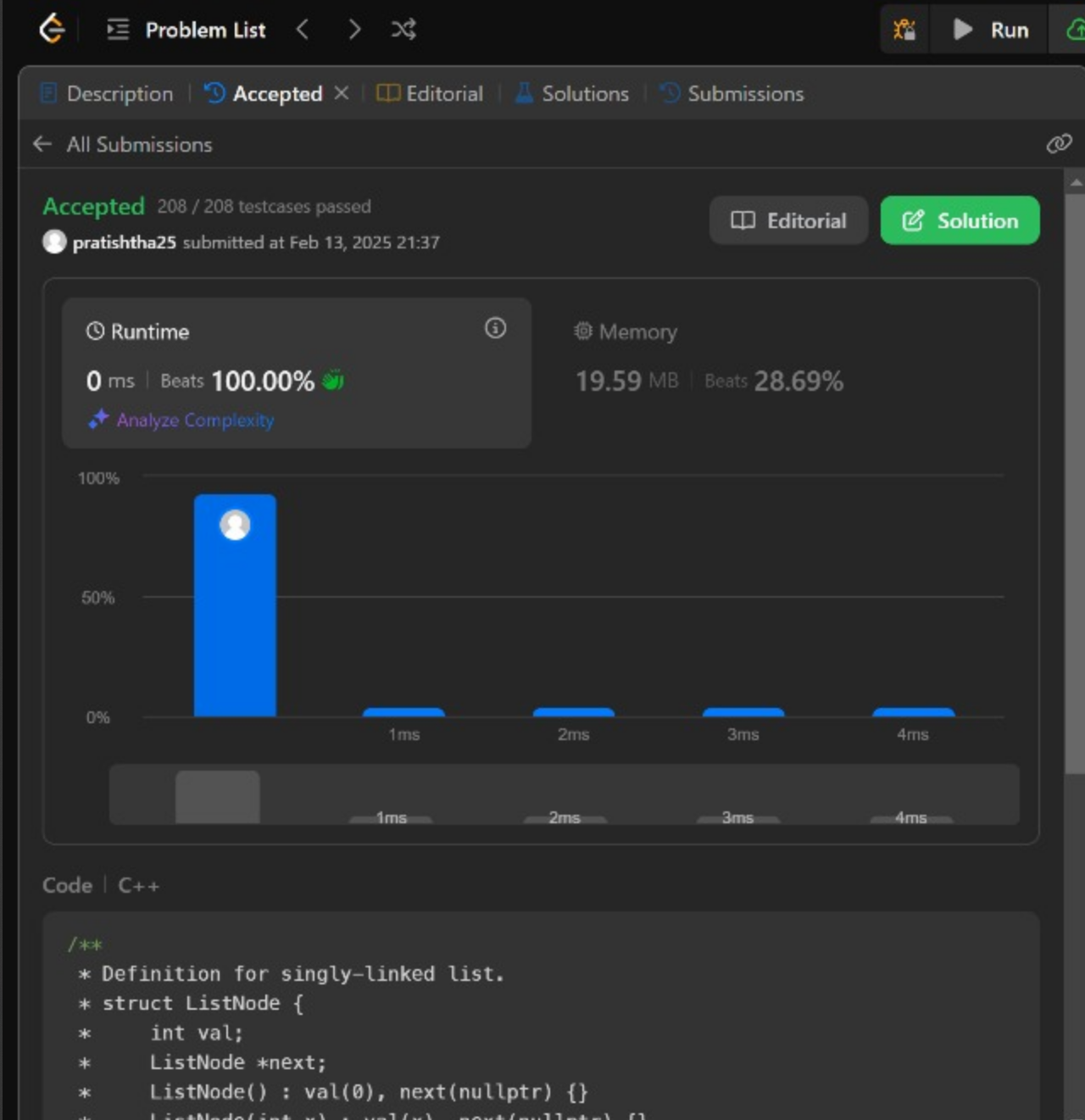
if (list2) tail->next = list2;

return dummy.next; // Return the merged list head

}

};

**Output:**

****

1. **Code: (**[**Remove Duplicates from Sorted List II**](https://leetcode.com/problems/remove-duplicates-from-sorted-list-ii/)**)**

class Solution {

public:

ListNode\* deleteDuplicates(ListNode\* head) {

if (!head) return nullptr;

ListNode dummy(0, head); // Dummy node pointing to head

ListNode\* prev = &dummy; // Previous pointer before potential duplicates

while (head) {

// If duplicates exist, skip all occurrences

if (head->next && head->val == head->next->val) {

while (head->next && head->val == head->next->val) {

head = head->next;

}

prev->next = head->next; // Skip duplicates

} else {

prev = prev->next; // Move prev only if no duplicate

}

head = head->next; // Move forward

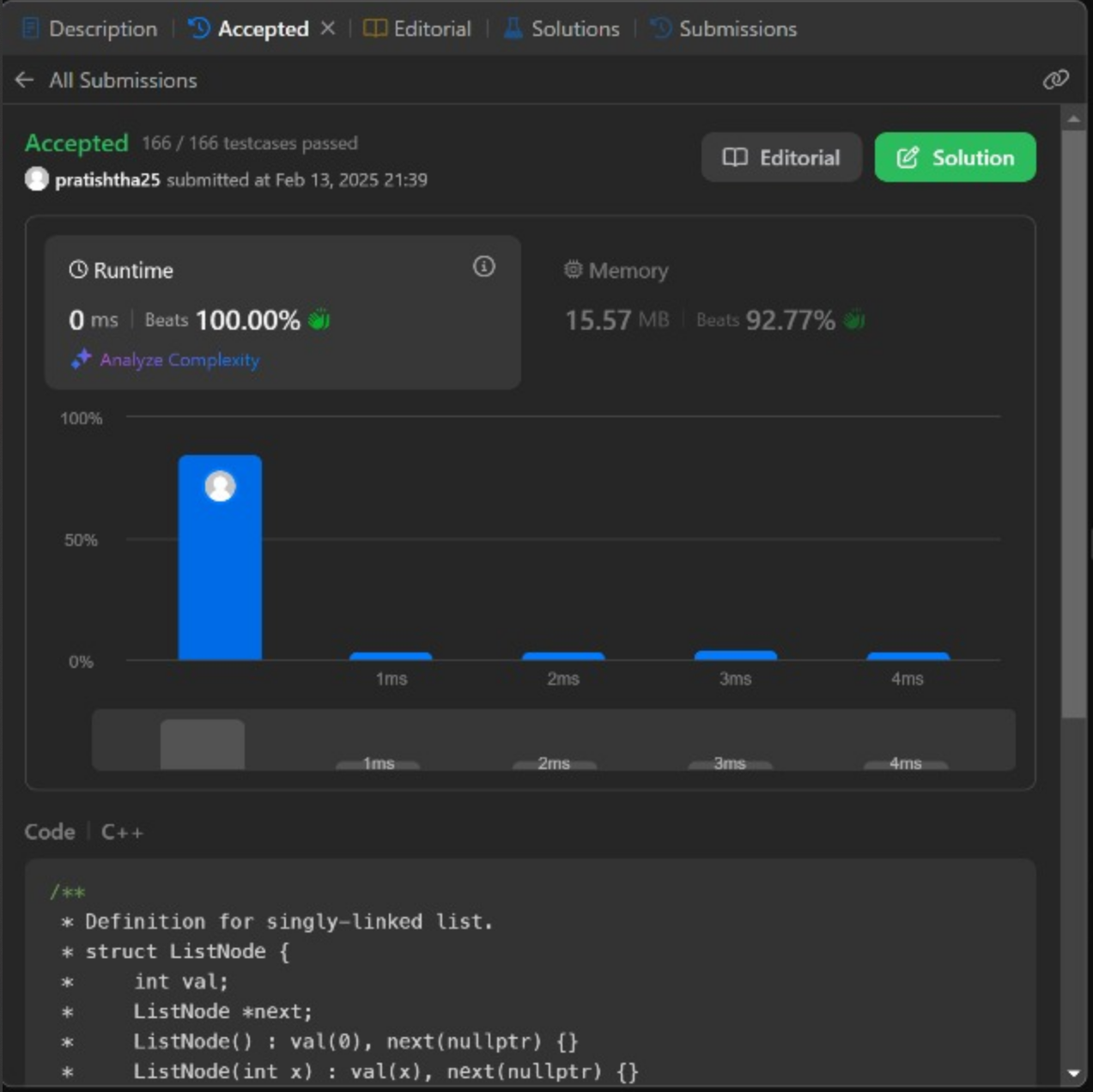
}

return dummy.next;

}

};

**Output:**

****

1. **Code: (**[**Linked List Cycle**](https://leetcode.com/problems/linked-list-cycle/)**)**

class Solution {

public:

bool hasCycle(ListNode \*head) {

if (!head || !head->next) return false;

ListNode \*slow = head;

ListNode \*fast = head;

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast) return true;

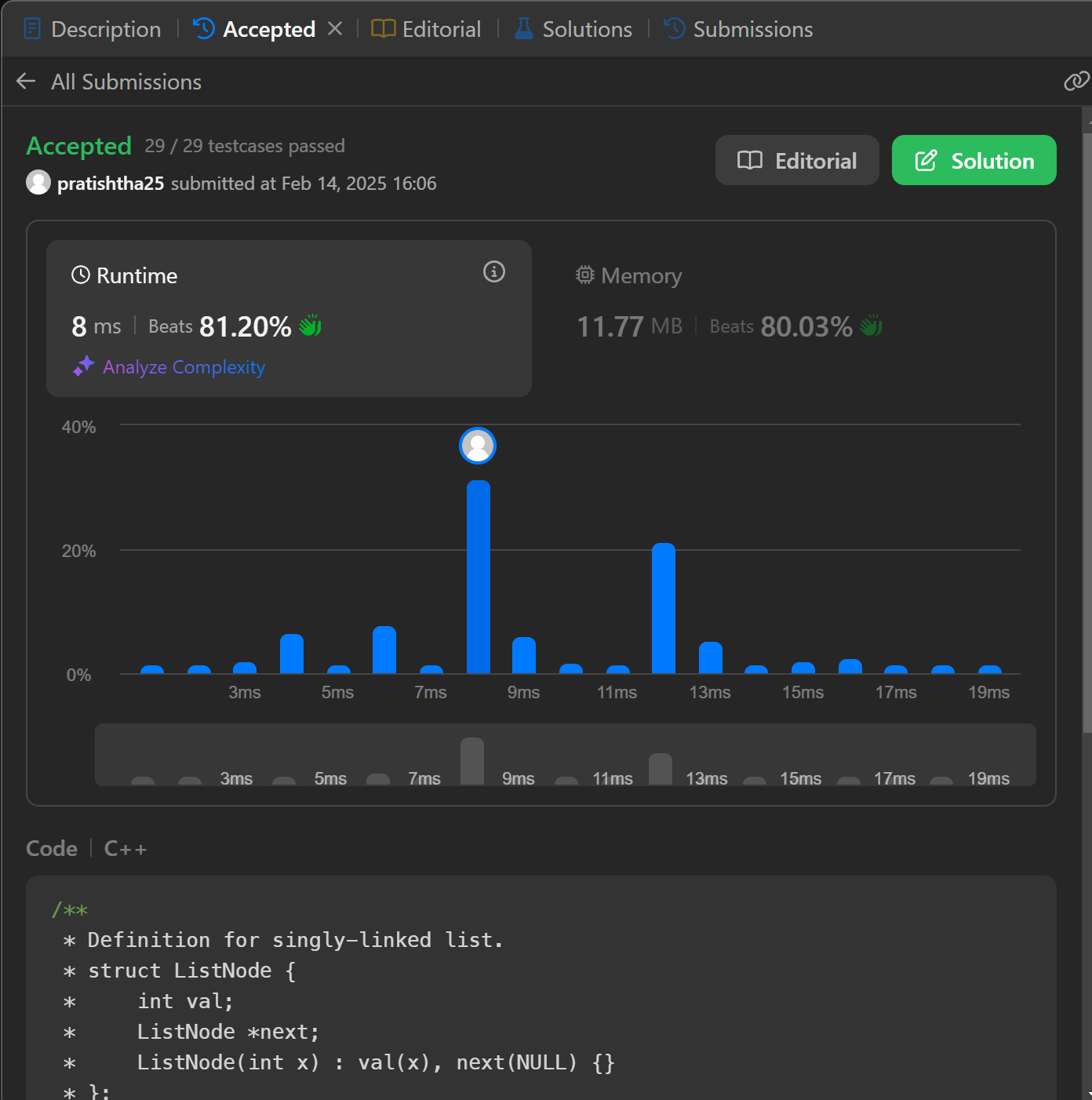
}

return false;

}

};

**Output:**

****

1. **Code: (**[**Reverse Linked List II**](https://leetcode.com/problems/reverse-linked-list-ii/)**)**

class Solution {

public:

ListNode\* reverseBetween(ListNode\* head, int left, int right) {

if (!head || left == right) return head;

ListNode dummy(0);

dummy.next = head;

ListNode\* prev = &dummy;

for (int i = 1; i < left; ++i) {

prev = prev->next;

}

ListNode\* curr = prev->next;

ListNode\* nextNode = nullptr;

for (int i = 0; i < right - left; ++i) {

nextNode = curr->next;

curr->next = nextNode->next;

nextNode->next = prev->next;

prev->next = nextNode;

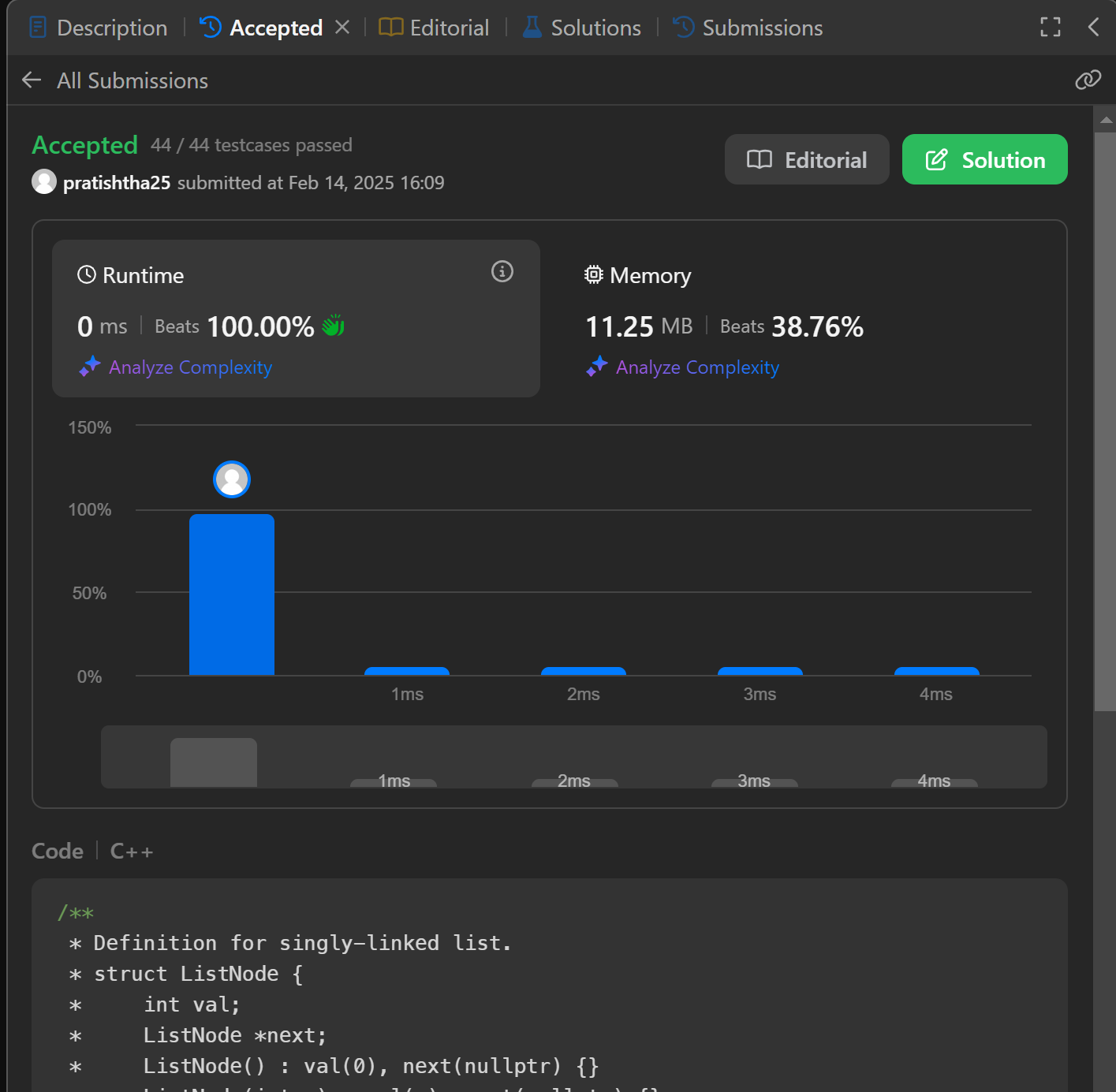
}

return dummy.next;

}

};

**Output:**

****

1. **Code: (**[**Rotate List**](https://leetcode.com/problems/rotate-list/)**)**

class Solution {

public:

ListNode\* rotateRight(ListNode\* head, int k) {

if (!head || !head->next || k == 0) return head;

// Step 1: Find the length of the linked list

int length = 1;

ListNode\* tail = head;

while (tail->next) {

tail = tail->next;

length++;

}

// Step 2: Optimize k

k = k % length;

if (k == 0) return head;

// Step 3: Find the new tail (length - k - 1) steps from the head

ListNode\* newTail = head;

for (int i = 0; i < length - k - 1; i++) {

newTail = newTail->next;

}

// Step 4: Update pointers

ListNode\* newHead = newTail->next;

newTail->next = nullptr;

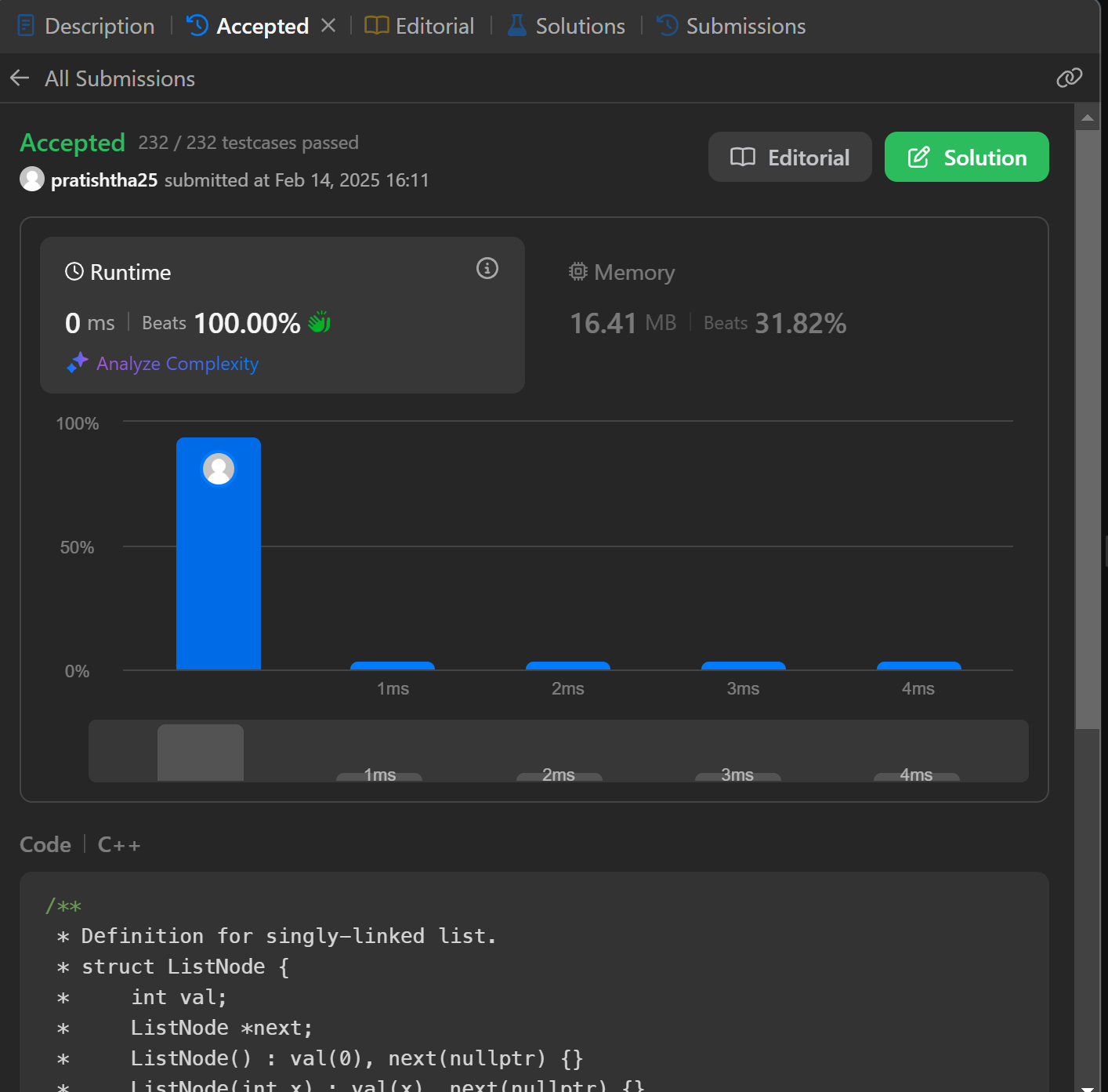
tail->next = head;

return newHead;

}

};

**Output:**

****

1. **Code: (**[**Sort List**](https://leetcode.com/problems/sort-list/)**)**

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

if (!l1) return l2;

if (!l2) return l1;

ListNode\* dummy = new ListNode(-1);

ListNode\* current = dummy;

while (l1 && l2) {

if (l1->val < l2->val) {

current->next = l1;

l1 = l1->next;

} else {

current->next = l2;

l2 = l2->next;

}

current = current->next;

}

if (l1) current->next = l1;

if (l2) current->next = l2;

return dummy->next;

}

// Function to sort the linked list using Merge Sort

ListNode\* sortList(ListNode\* head) {

if (!head || !head->next) return head; // Base case

// Step 1: Find the middle of the list

ListNode\* slow = head;

ListNode\* fast = head;

ListNode\* prev = nullptr;

while (fast && fast->next) {

prev = slow;

slow = slow->next;

fast = fast->next->next;

}

// Step 2: Split the list into two halves

prev->next = nullptr; // Break the list into two halves

// Step 3: Recursively sort both halves

ListNode\* left = sortList(head);

ListNode\* right = sortList(slow);

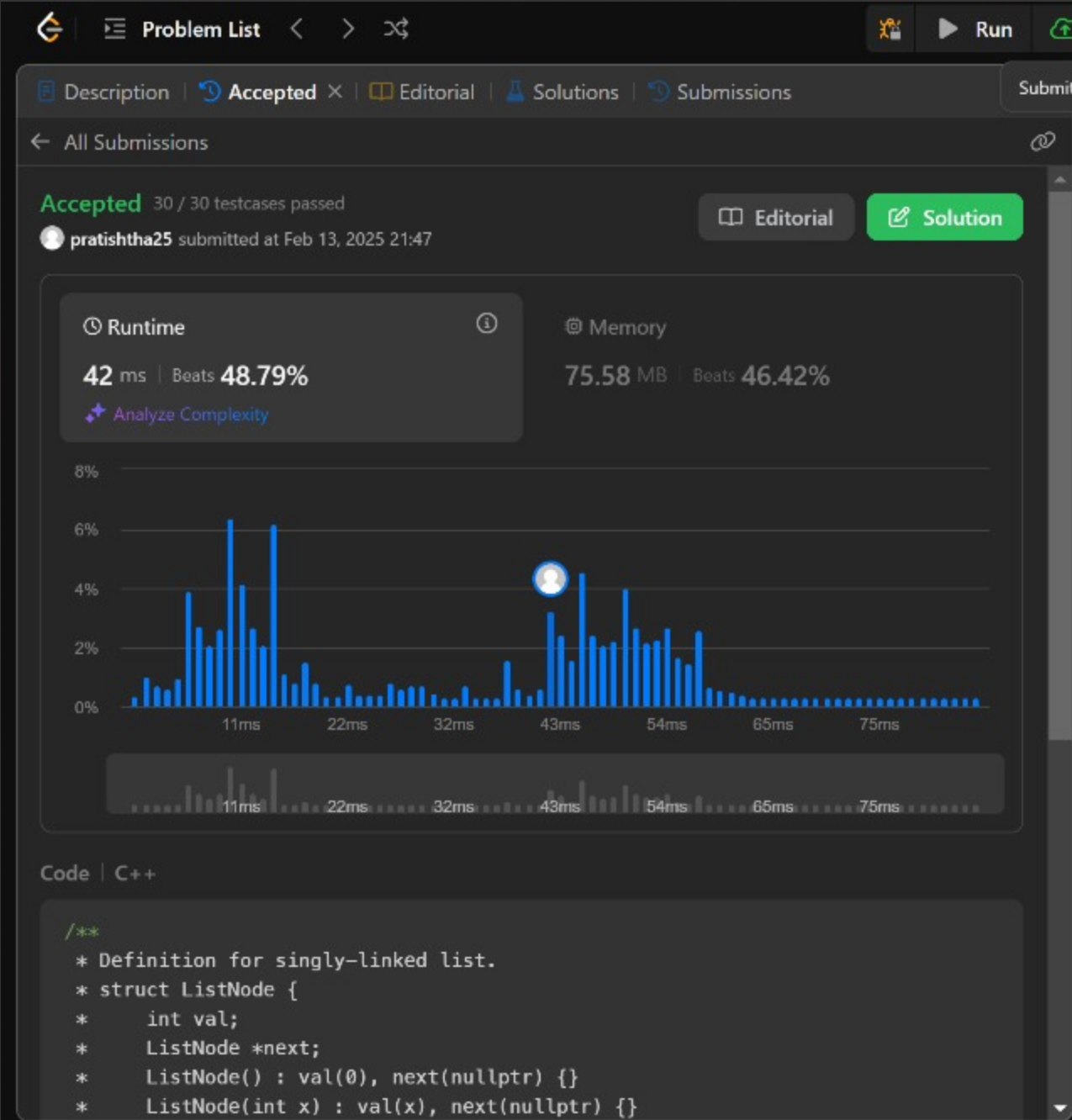
// Step 4: Merge the sorted halves

return mergeTwoLists(left, right);

}

};

**Output:**

****

1. **Code: (**[**Linked List Cycle II**](https://leetcode.com/problems/linked-list-cycle-ii/)**)**

class Solution {

public:

ListNode \*detectCycle(ListNode \*head) {

if (!head || !head->next) return nullptr;

ListNode \*slow = head, \*fast = head;

// Step 1: Detect Cycle

while (fast && fast->next) {

slow = slow->next; // Move slow pointer one step

fast = fast->next->next; // Move fast pointer two steps

if (slow == fast) { // Cycle detected

// Step 2: Find the start of the cycle

slow = head; // Reset slow to head

while (slow != fast) {

slow = slow->next;

fast = fast->next;

}

return slow; // Cycle start node

}

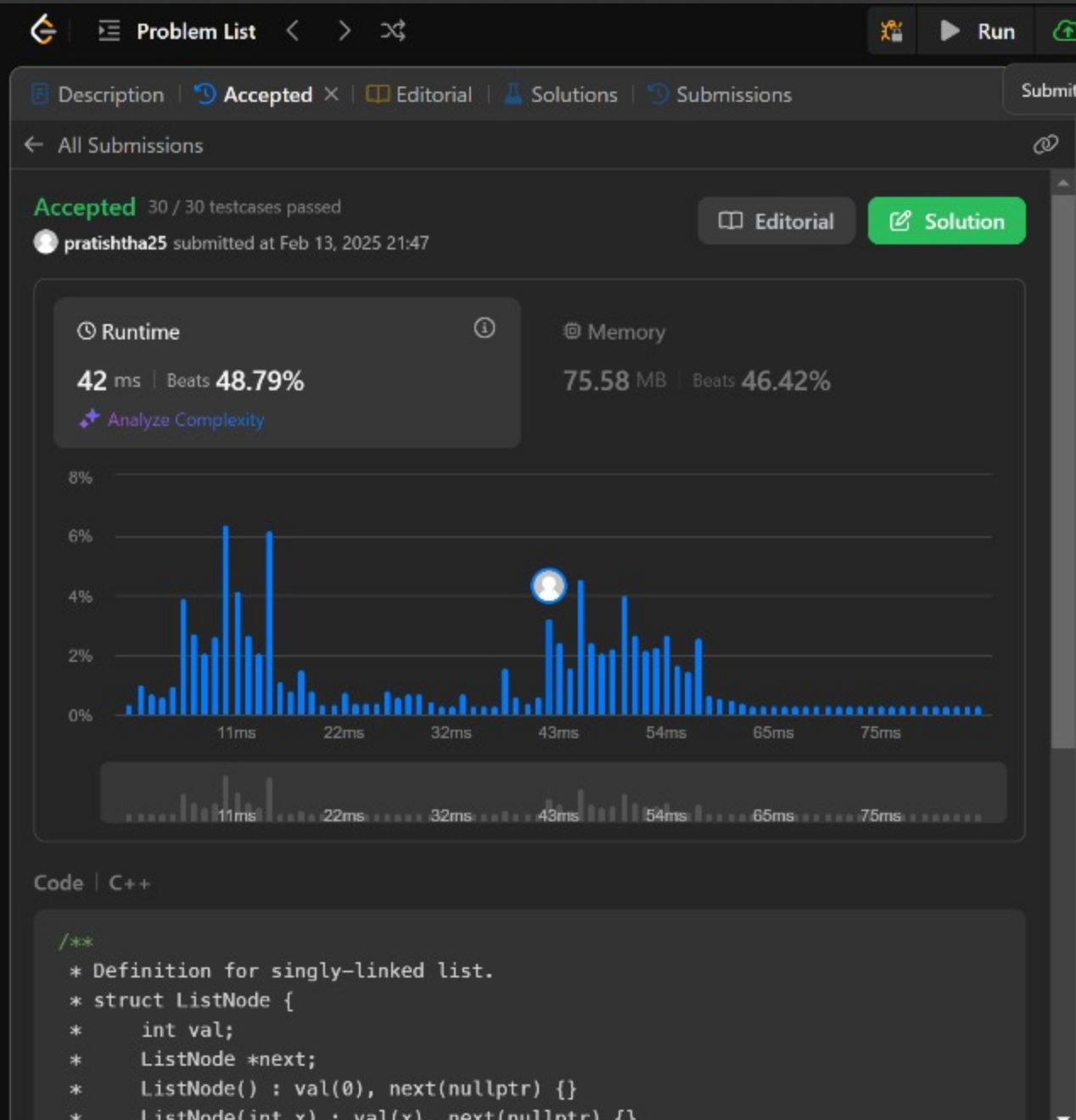
}

return nullptr;

}

};

**Output:**

****