** **

**UNIVERSITY INSTITUTE OF ENGINEERING**

**Department of Computer Science & Engineering**

**(BE-CSE/IT-5th Sem)**

****

**Subject Name: AP LAB-II**

**Subject Code: 22CSP-351**

**Submitted to: Submitted by:**

Faculty name Name: Ankit Kharb

Bhumika UID: 22BCS16964

Section: 22BCS-IOT-614

Group: B

**Question 1.** Print Linked list.

**CODE:**

1>>>>.

/\* Solution class \*/

class Solution {

public:

void printList(Node\* head) {

while (head) {

cout << head->data << " ";

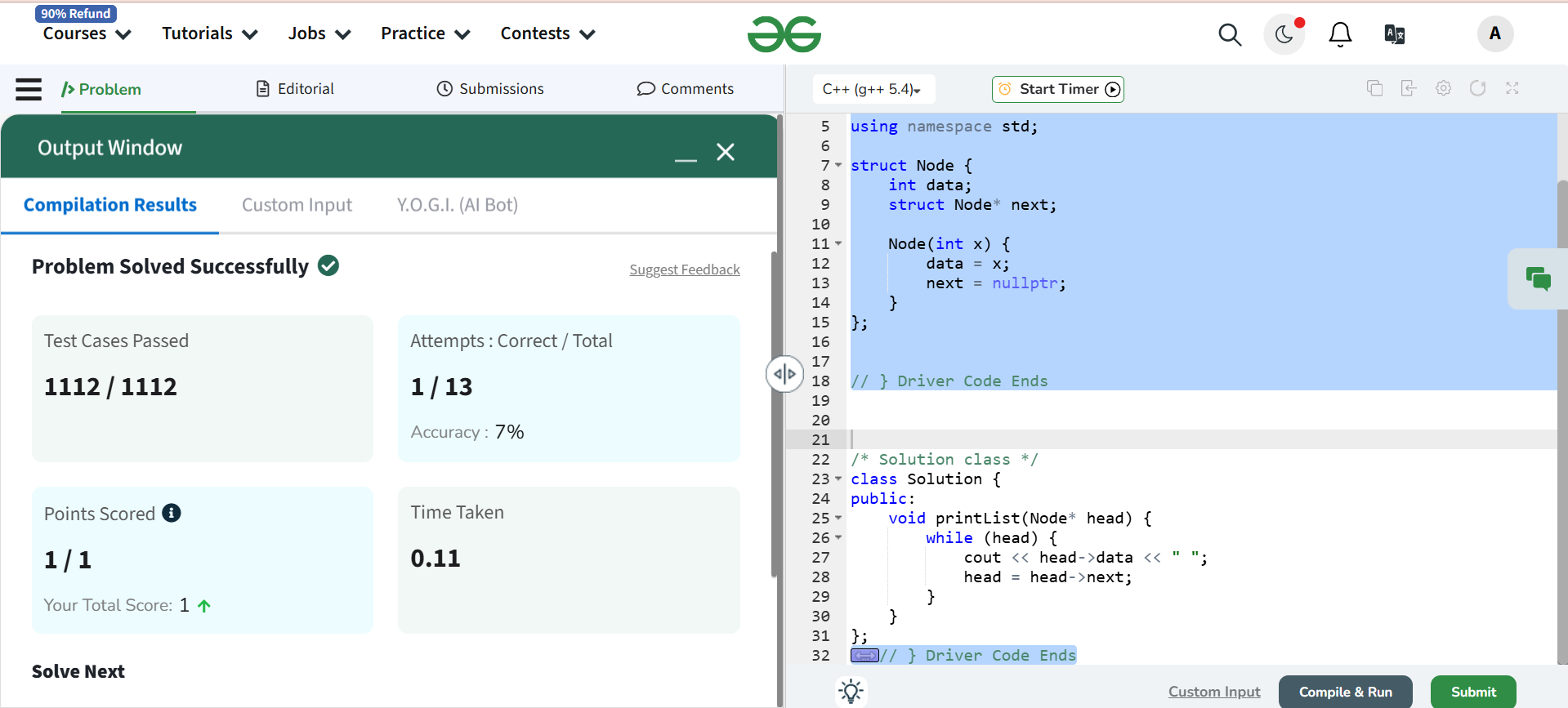
head = head->next;

}

}

};

**OUTPUT:**

****

**Question 2:** [Remove duplicates from a sorted list](https://leetcode.com/problems/remove-duplicates-from-sorted-list/description/)

**CODE:**

/\*\*

 \* Definition for singly-linked list.

class Solution {

public:

    ListNode\* deleteDuplicates(ListNode\* head) {

        if (!head) return nullptr; // Edge case: empty list

        ListNode\* current = head; // Pointer to traverse the lis

        while (current->next) { // Traverse until the last node

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

                current->next = current->next->next; // Skip duplicate node

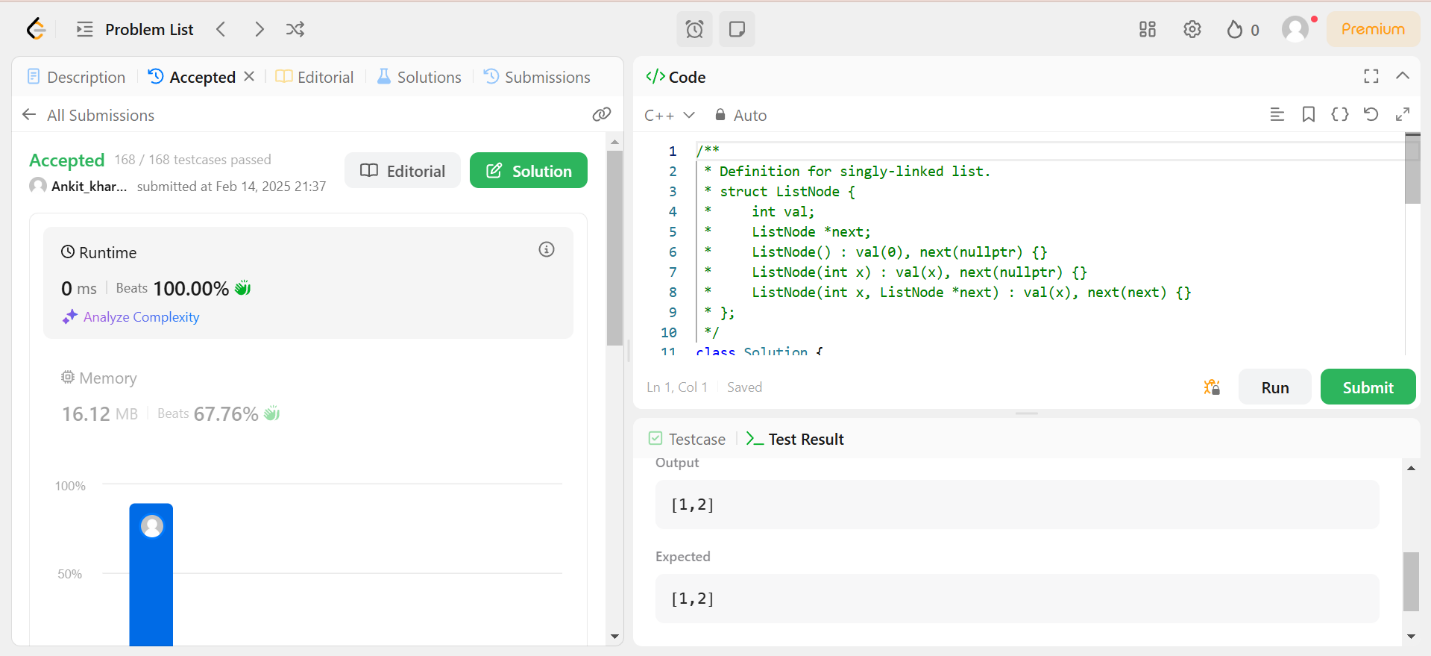
            } else {

                current = current->next; // Move forward

 } }

        return head;}};

**OUTPUT:**



**Question 3:** [ Reverse a linked list] (<https://leetcode.com/problems/reverse-linked-list/>)

**CODE:**

class Solution {

public:

    ListNode\* reverseList(ListNode\* head) {

        ListNode\* prev = nullptr;

        ListNode\* curr = head;

        while (curr != nullptr) {

            ListNode\* nextNode = curr->next; // Save next node

            curr->next = prev; // Reverse the link

            prev = curr; // Move prev forward

            curr = nextNode; // Move curr forward

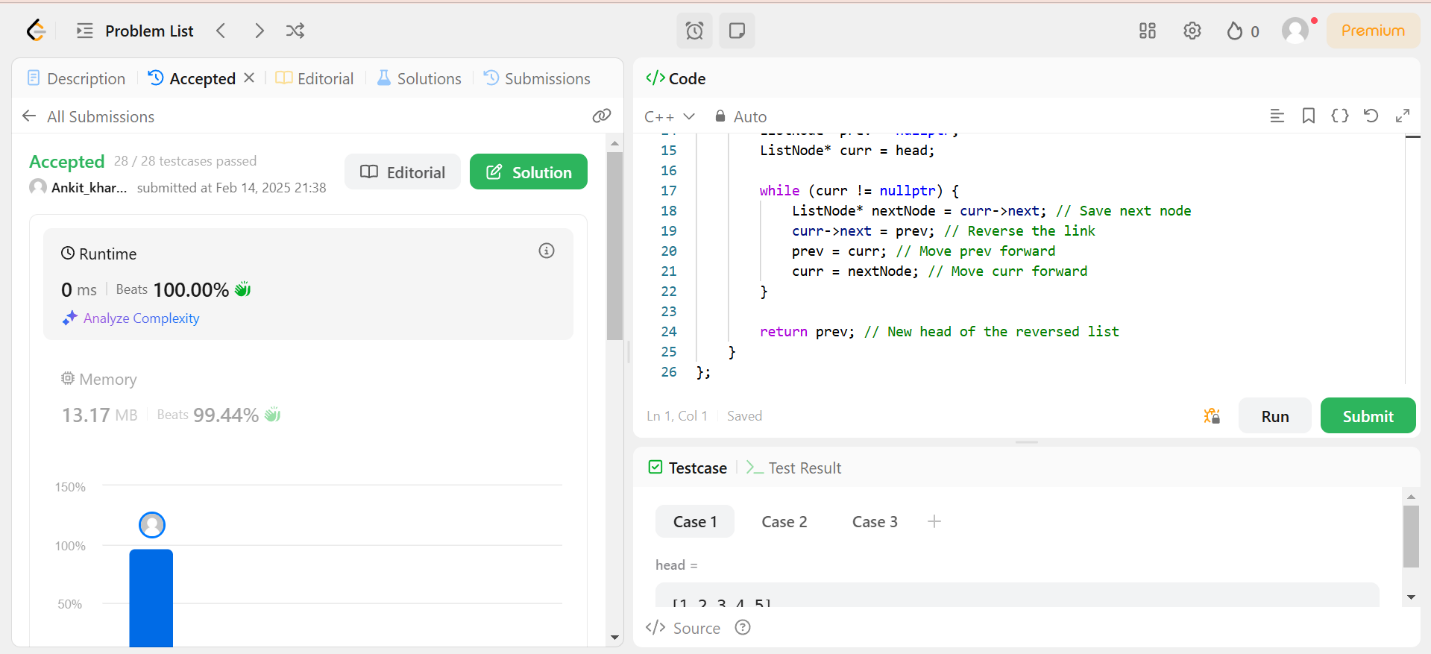
        }

        return prev; // New head of the reversed list

    }

};

**OUTPUT:**



**Question 4:** Delete the middle node of a list.

**CODE:**

class Solution {

public:

    ListNode\* deleteMiddle(ListNode\* head) {

        // Edge case: if the list has only one node, return nullptr

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

        ListNode\* slow = head;

        ListNode\* fast = head;

        ListNode\* prev = nullptr;  // To keep track of the node before slow

        while (fast && fast->next) {

            prev = slow;  // Store previous node

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

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

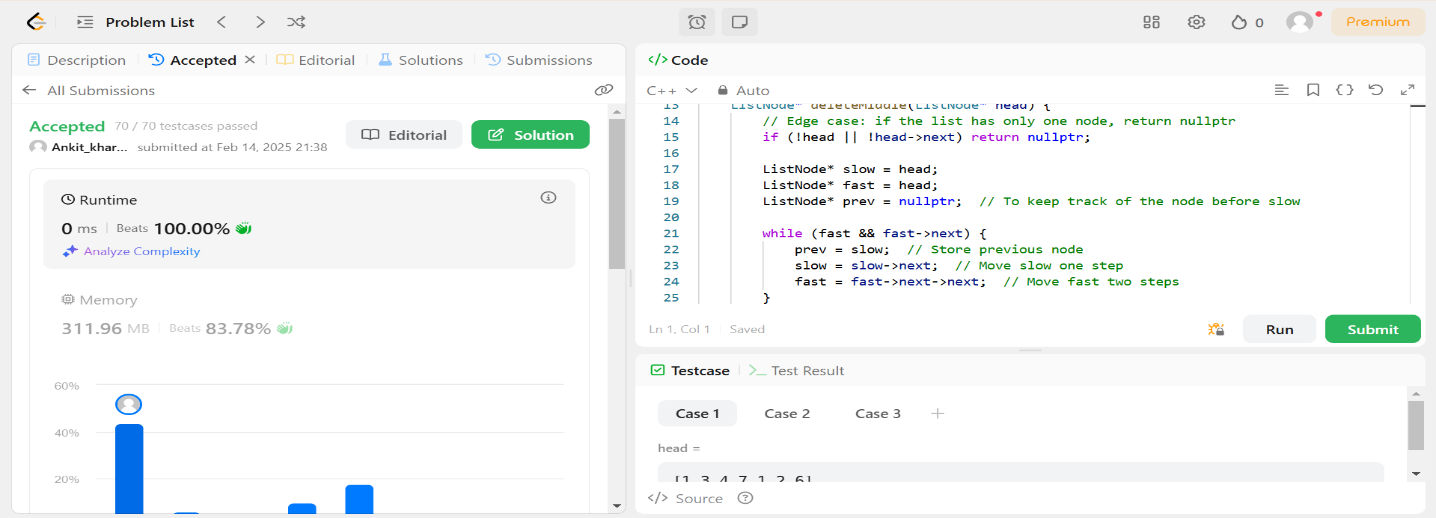
        // Delete the middle node

        prev->next = slow->next;

        delete slow;  // Free memory

return head;}};

**OUTPUT:**

****

**Question 5:** Merge two sorted linked list.

**CODE:**

class Solution {

public:

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

        // Create a dummy node to simplify operations

        ListNode dummy(0);

        ListNode\* current = &dummy;

        // Traverse both lists

        while (list1 && list2) {

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

                current->next = list1;

                list1 = list1->next;

            } else {

                current->next = list2;

                list2 = list2->next;

            }

            current = current->next;

        }

        // If there are remaining nodes in either list1 or list2, attach them

        if (list1) current->next = list1;

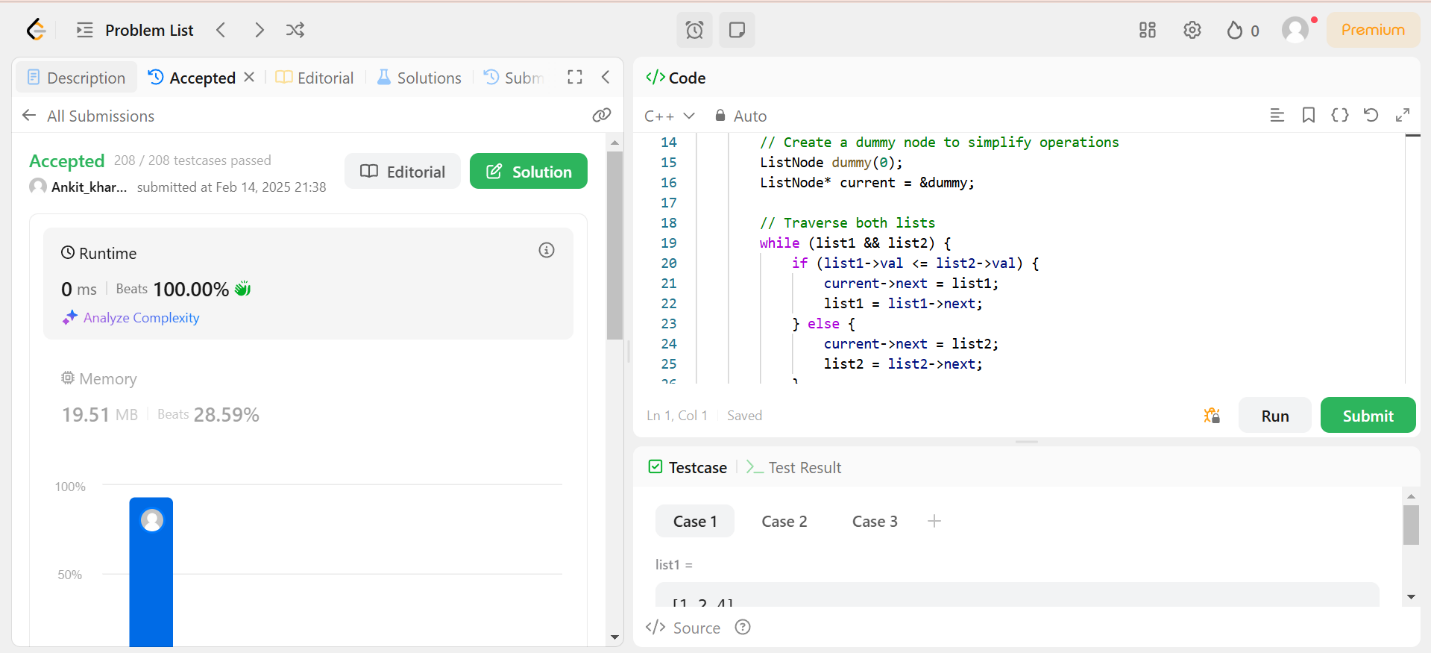
        if (list2) current->next = list2;

        return dummy.next;  // The merged sorted list

    }

};

**OUTPUT:**

****

**Question 6:** Remove duplicates from sorted list two.

**CODE:**

class Solution {

public:

    ListNode\* deleteDuplicates(ListNode\* head) {

        if (!head || !head->next) return head;  // Edge case: Empty or single-node list

        ListNode dummy(0);  // Dummy node before head

        dummy.next = head;

        ListNode\* prev = &dummy;  // Pointer to last unique node

        ListNode\* curr = head;  // Pointer to traverse list

        while (curr) {

            bool isDuplicate = false;

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

                curr = curr->next;  // Skip duplicate nodes

                isDuplicate = true;

            }

            if (isDuplicate) {

                prev->next = curr->next;  // Remove duplicates

            } else {

                prev = prev->next;  // Move prev forward

            }

            curr = curr->next;  // Move curr forward

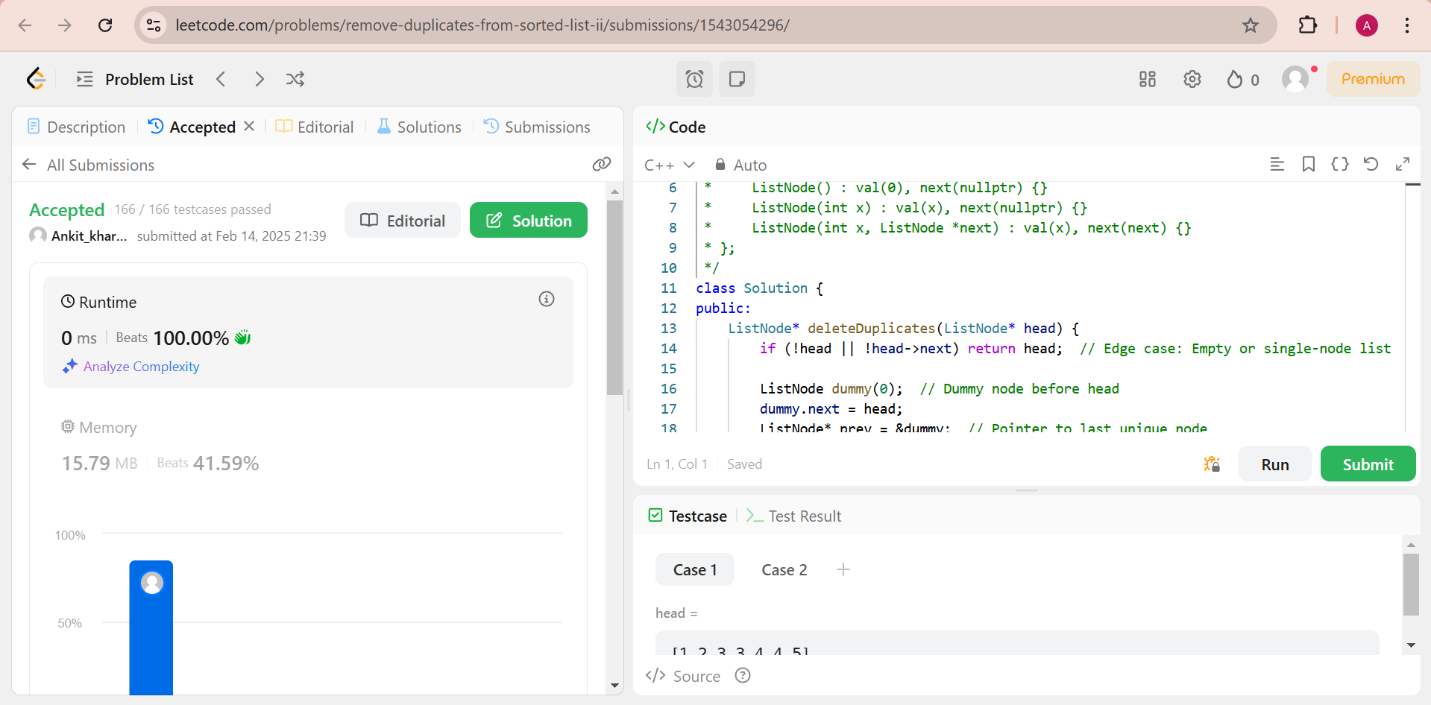
        }

        return dummy.next;

    }

};

**OUTPUT:**



**Question 7:** Detect a cycle in Linked List

**CODE:**

class Solution {

public:

    bool hasCycle(ListNode \*head) {

        ListNode\* slow = head;

        ListNode\* fast = head;

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

            slow = slow->next;       // Move slow by 1 step

            fast = fast->next->next; // Move fast by 2 steps

            if (slow == fast) { // If they meet, cycle exists

                return true;

            }

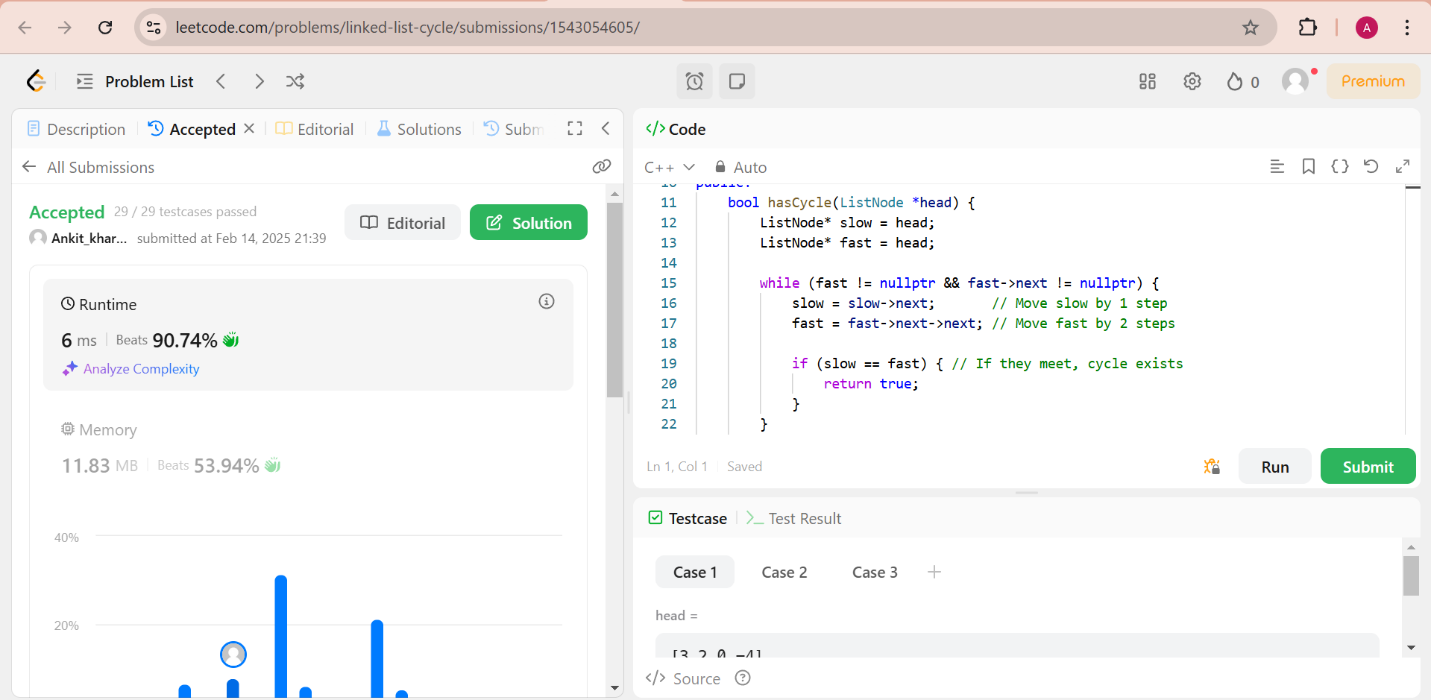
        }

        return false; // If fast reaches the end, no cycle

    }

};

**OUTPUT:**



**Question 8:** Reverse a linked list 2

**CODE:**

class Solution {

public:

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

        if (!head || left == right) return head;  // Edge case: No change needed

        ListNode dummy(0);  // Dummy node before head

        dummy.next = head;

        ListNode\* prev = &dummy;

        // Step 1: Move `prev` to the node just before `left`

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

            prev = prev->next;

        }

        // Step 2: Reverse the sublist between `left` and `right`

        ListNode\* curr = prev->next;

        ListNode\* nextNode = nullptr;

        ListNode\* reversePrev = nullptr;

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

            nextNode = curr->next;

            curr->next = reversePrev;

            reversePrev = curr;

            curr = nextNode;

        }

        // Step 3: Reconnect the reversed sublist to the original list

        prev->next->next = curr;  // Connect end of reversed part to next part

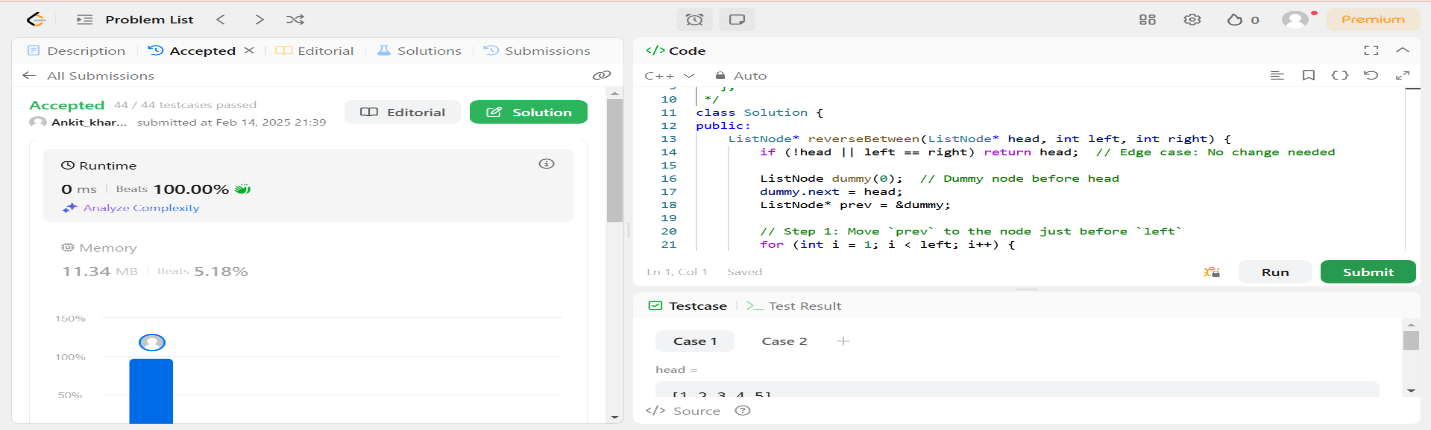
        prev->next = reversePrev; // Connect start of reversed part

        return dummy.next;

    }

};

**OUTPUT:**

****

**Question 9:** Rotate a List:

**CODE:**

class Solution {

public:

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

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

        // Step 1: Compute the length of the list

        int length = 1;

        ListNode\* tail = head;

        while (tail->next) {

            tail = tail->next;

            length++;

        }

        // Step 2: Make it a circular linked list

        tail->next = head;

        // Step 3: Find the new tail at (length - k % length) position

        k = k % length;

        int stepsToNewTail = length - k;

        ListNode\* newTail = head;

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

            newTail = newTail->next;

        }

        // Step 4: Break the cycle and set new head

        ListNode\* newHead = newTail->next;

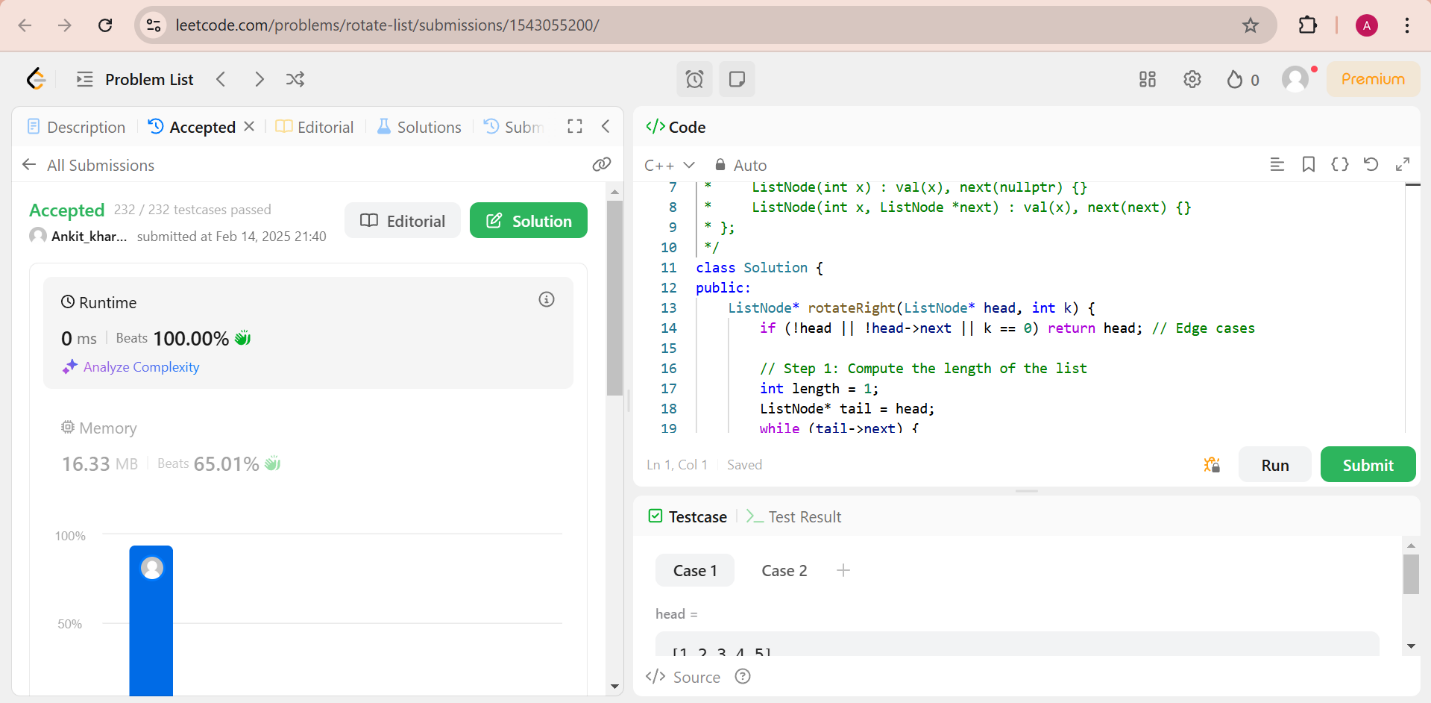
        newTail->next = nullptr;

        return newHead;

    }

};

**OUTPUT:**



**Question 10:** Sort List

**CODE:** class Solution {

public:

    // Function to merge two sorted linked lists

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

        if (!l1) return l2;

        if (!l2) return l1;

        // Dummy node to simplify merging process

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

        ListNode\* tail = dummy;

        while (l1 && l2) {

            if (l1->val < l2->val) {

                tail->next = l1;

                l1 = l1->next;

            } else {

                tail->next = l2;

                l2 = l2->next;

            }

            tail = tail->next;

        }

        // Append remaining nodes

        tail->next = l1 ? l1 : l2;

        return dummy->next;

    }

    // Function to find the middle node

    ListNode\* getMiddle(ListNode\* head) {

        ListNode\* slow = head;

        ListNode\* fast = head->next; // Ensures left half is smaller when splitting even-sized lists

        while (fast && fast->next) {

            slow = slow->next;

            fast = fast->next->next;

        }

        return slow;

    }

    ListNode\* sortList(ListNode\* head) {

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

        // Step 1: Find the middle node

        ListNode\* mid = getMiddle(head);

        ListNode\* rightHead = mid->next;

        mid->next = nullptr; // Split the list

        // Step 2: Sort both halves recursively

        ListNode\* left = sortList(head);

        ListNode\* right = sortList(rightHead);

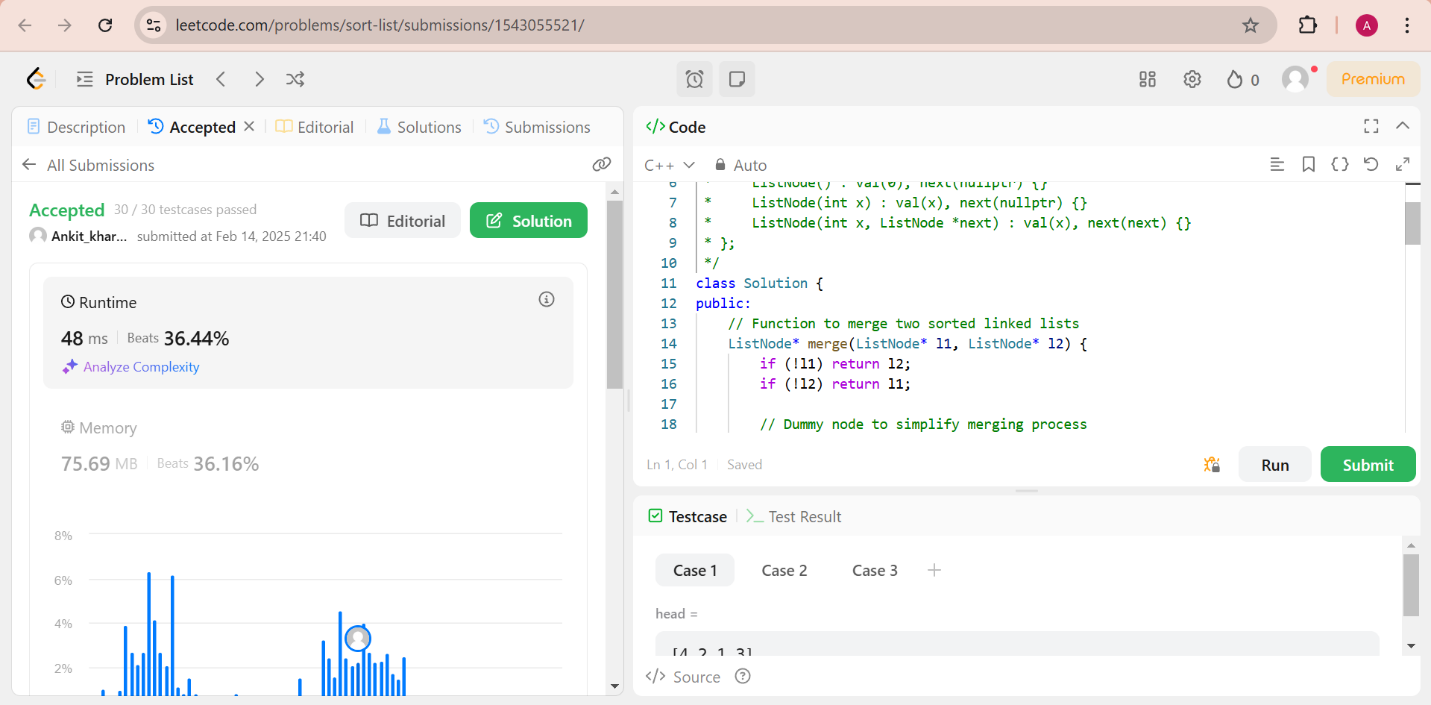
        // Step 3: Merge the two sorted halves

        return merge(left, right);

    }

};

**OUTPUT:**

****

**Question 11:** Linked list cycle 2

**CODE:**

class Solution {

public:

    ListNode \*detectCycle(ListNode \*head) {

        if (!head || !head->next) return nullptr; // Edge case: Empty list or single node

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

        // Step 1: Detect cycle using Floyd's Algorithm

        while (fast && fast->next) {

            slow = slow->next;

            fast = fast->next->next;

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

                ListNode \*entry = head;

                // Step 2: Find the entry point of the cycle

                while (entry != slow) {

                    entry = entry->next;

                    slow = slow->next;

                }

                return entry; // Cycle starts here

            }

        }

        return nullptr; // No cycle found

    }

};

**OUTPUT:**

