UNIVERSITY INSTITUTE OF ENGINEERING

**Department of Computer Science & Engineering**

**(BE-CSE/IT-6th Sem)**

****

**Subject Name: Advanced Programming**

**Subject Code:** 22CSP-351

**Submitted to: Submitted by:**

Faculty name: Bhumika Manjhi(E18199) Name: Armaan UID:22BCS12007

Section:614 Group:B



**Aim-** Solve leetcode and GFG problems

1. **Print Linked list**

**Code**

class Solution {

void printList(Node head) {

Node temp = head;

while (temp != null) {

System.out.print(temp.data + " ");

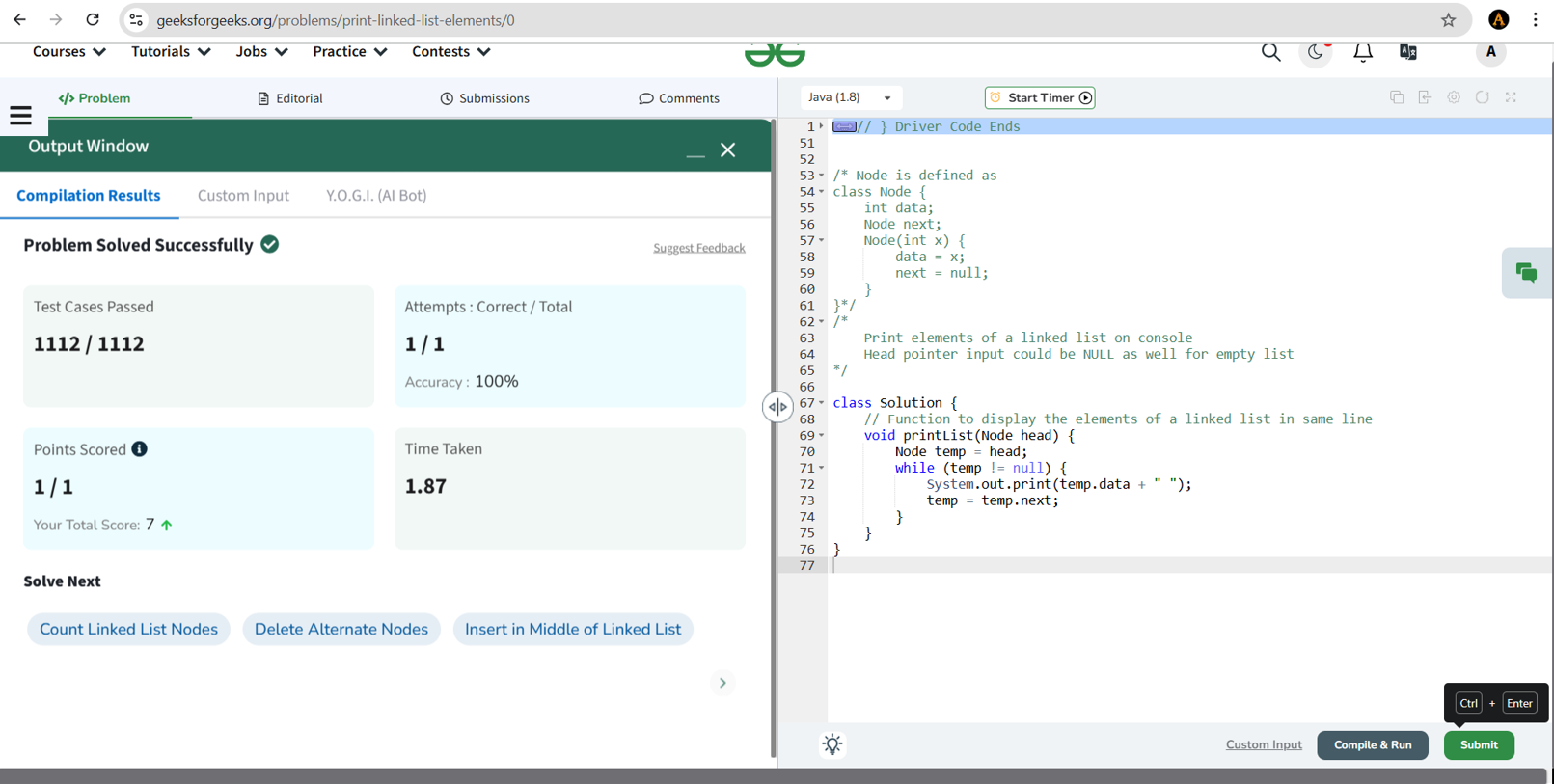
temp = temp.next;

}

}

}

**output**

****



1. **Remove duplicates from a sorted list**

**Code**

class Solution {

public ListNode deleteDuplicates(ListNode head) {

if (head == null) return null; // If the list is empty, return null

ListNode current = head;

while (current != null && current.next != null) {

if (current.val == current.next.val) {

current.next = current.next.next; // Skip the duplicate node

} else {

current = current.next; // Move to the next unique node

}

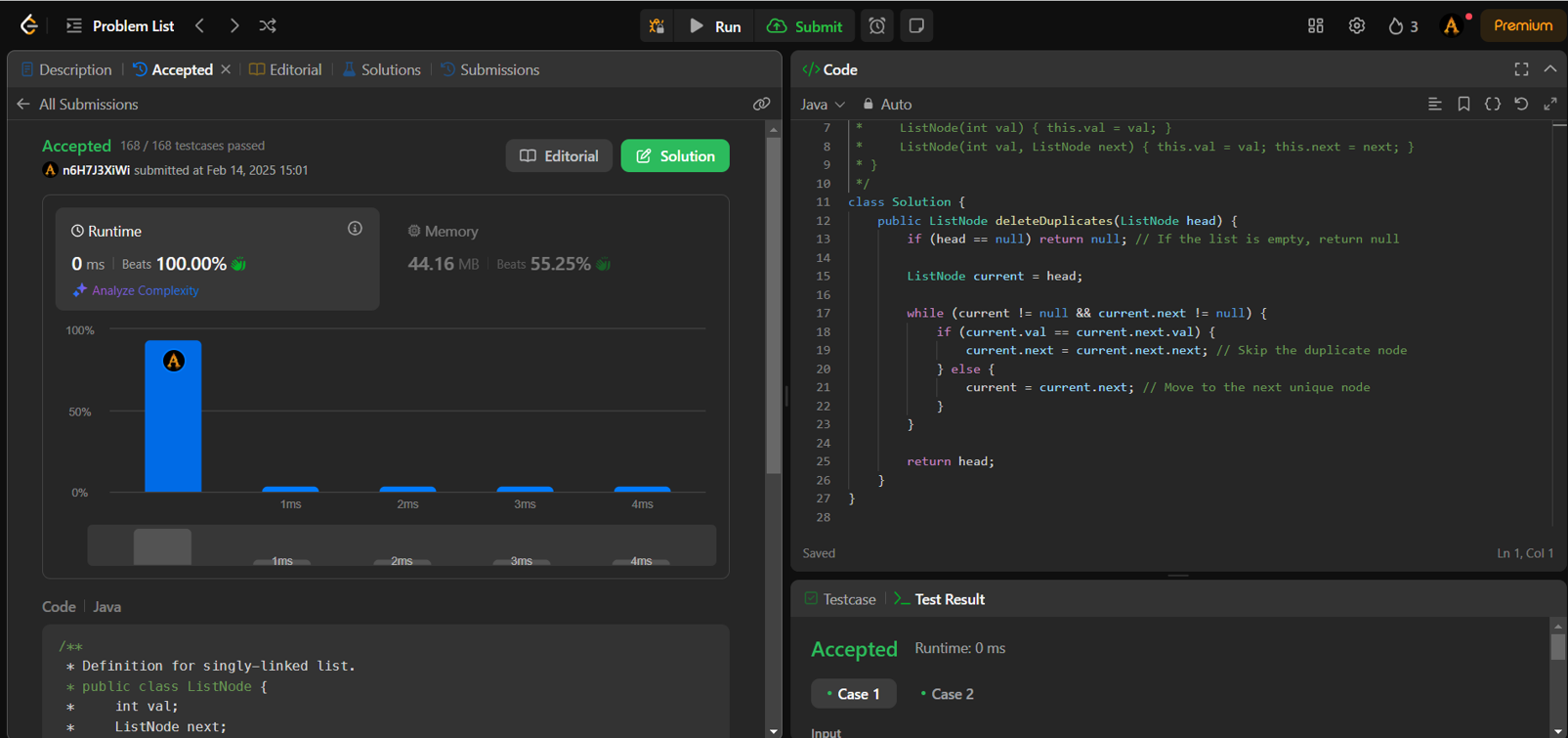
}

return head;

}

}

**output**





1. **Reverse a Linked list**

**Code**

class Solution {

public ListNode reverseList(ListNode head) {

if (head == null || head.next == null) return head; // Base case

ListNode newHead = reverseList(head.next); // Reverse the rest

head.next.next = head; // Make the next node point back

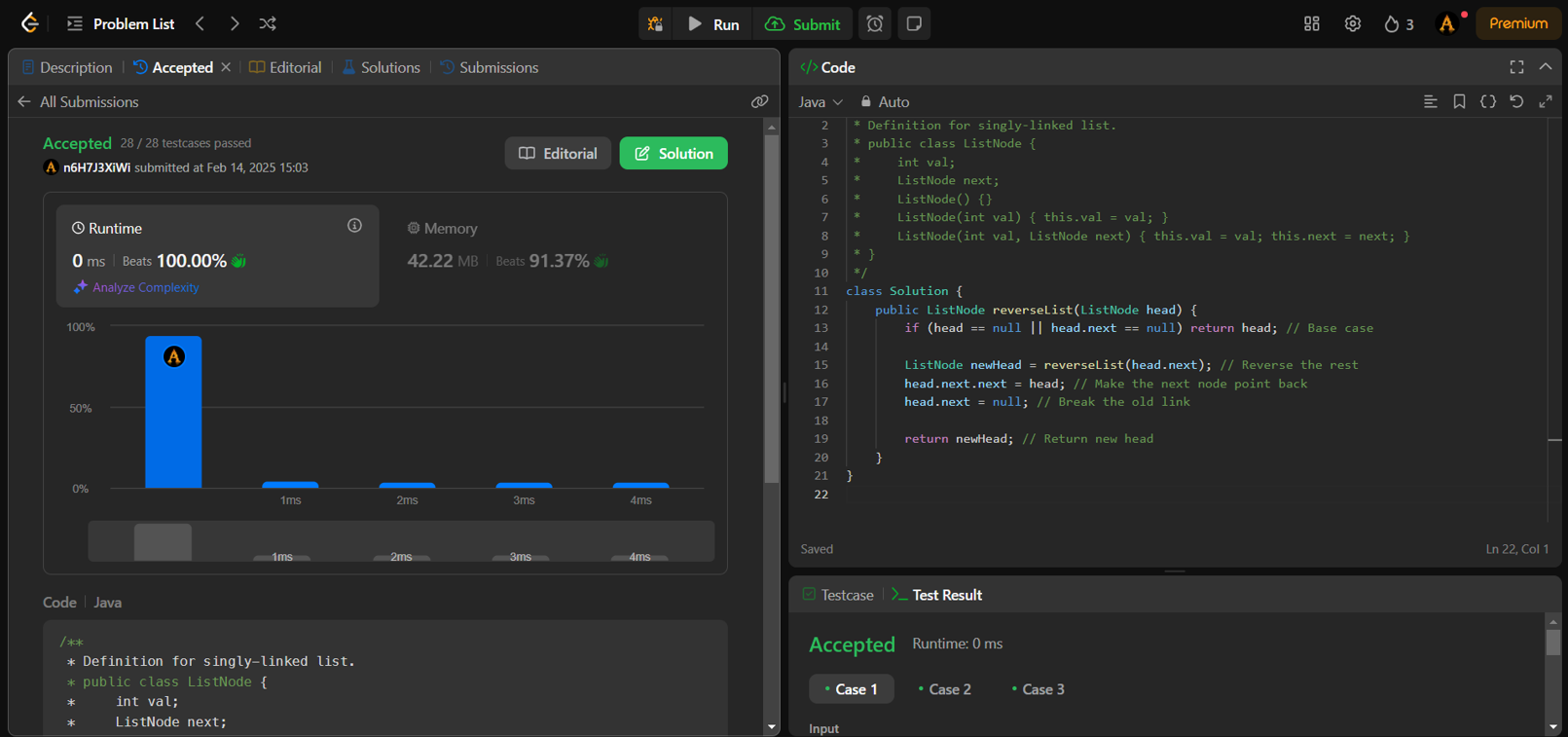
head.next = null; // Break the old link

return newHead; // Return new head

}

}

**Output**

****



1. **Delete middle node of a list**

**Code**

class Solution {

public ListNode deleteMiddle(ListNode head) {

if (head == null || head.next == null) return null; // If only one node, return null

ListNode slow = head, fast = head, prev = null;

while (fast != null && fast.next != null) {

prev = slow; // Store previous node

slow = slow.next; // Move slow pointer one step

fast = fast.next.next; // Move fast pointer two steps

}

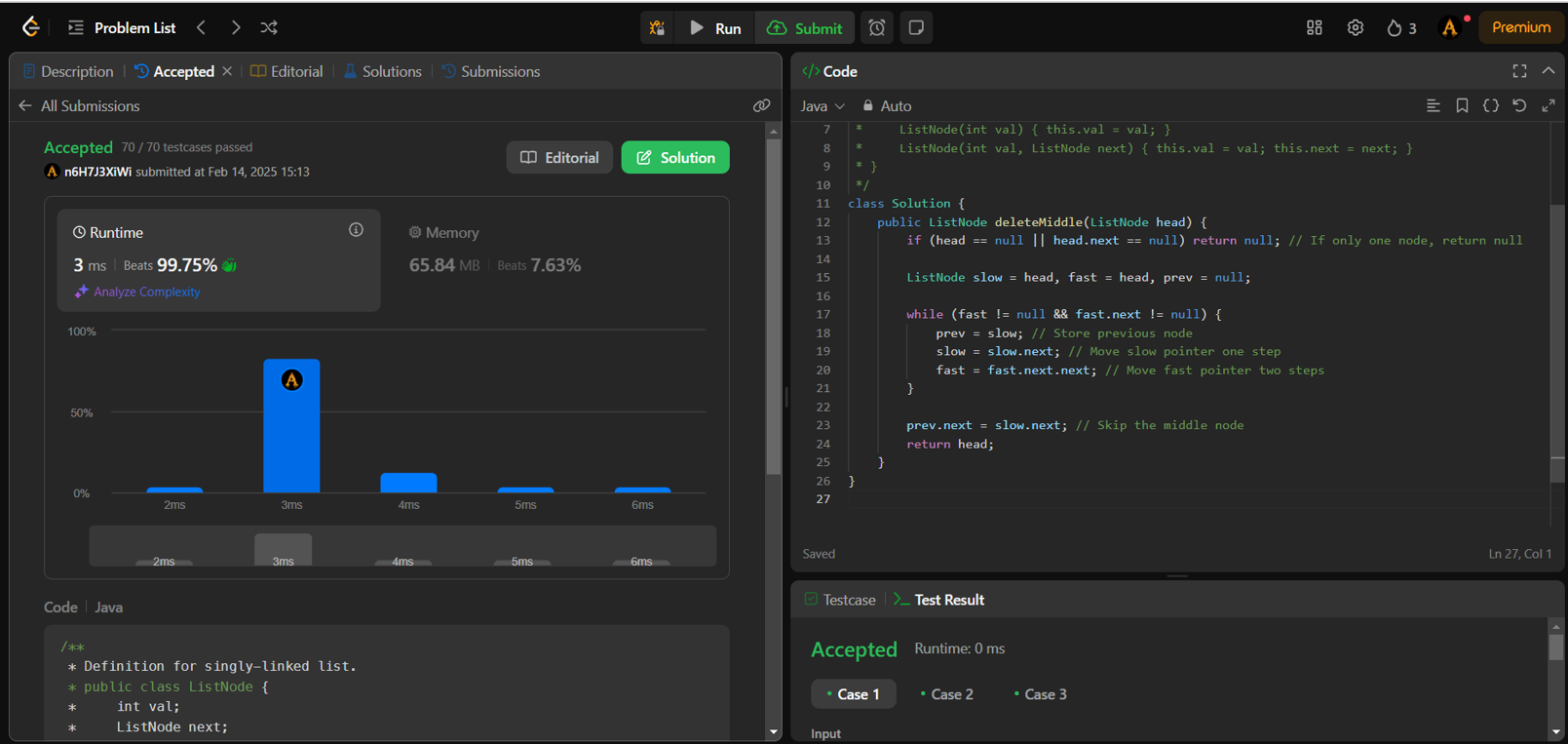
prev.next = slow.next; // Skip the middle node

return head;

}

}

**Output**

****



1. **Merge two sorted linked lists**

**Code**

class Solution {

public ListNode mergeTwoLists(ListNode list1, ListNode list2) {

ListNode dummy = new ListNode(-1); // Dummy node to simplify merging

ListNode current = dummy;

while (list1 != null && list2 != null) {

if (list1.val <= list2.val) {

current.next = list1;

list1 = list1.next;

} else {

current.next = list2;

list2 = list2.next;

}

current = current.next;

}

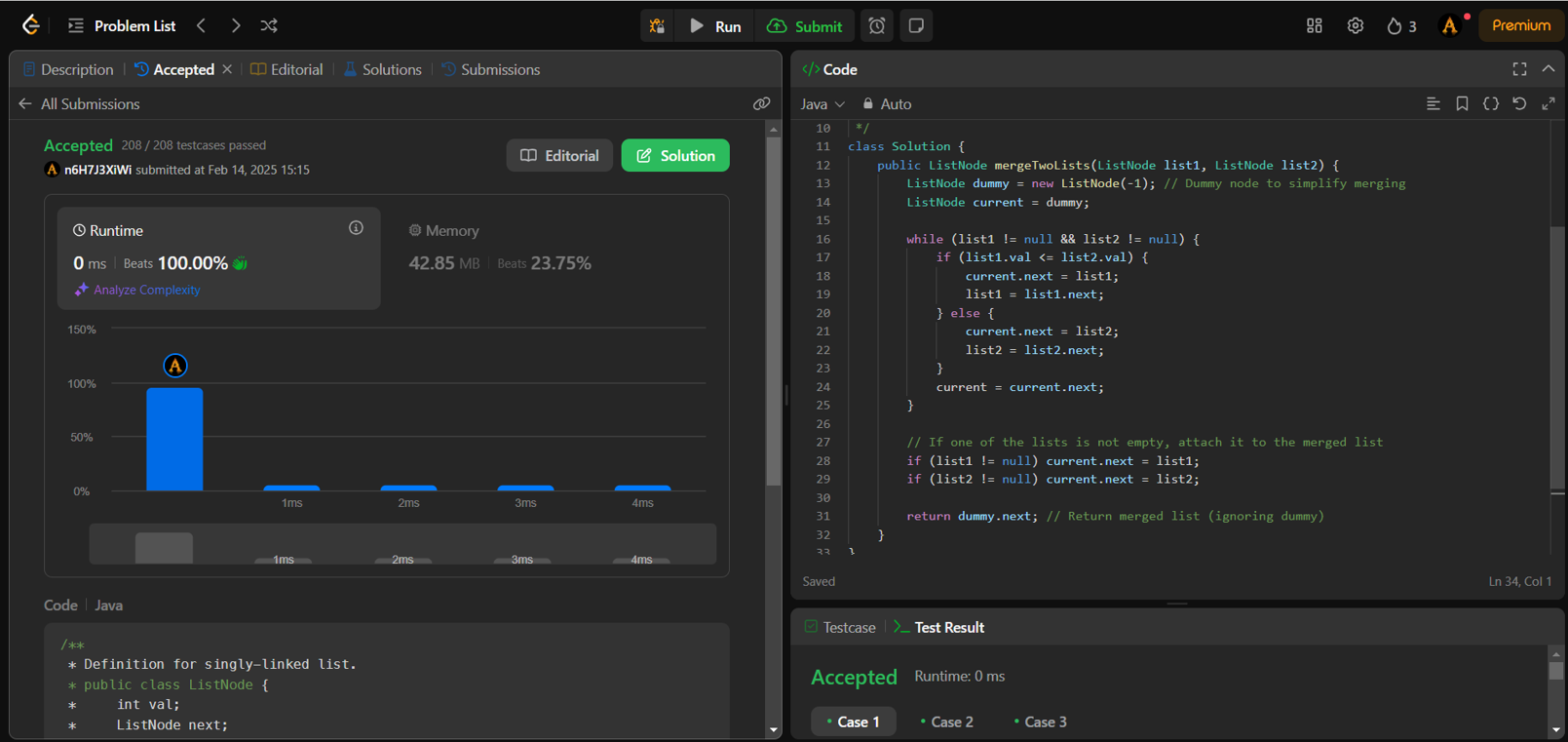
if (list1 != null) current.next = list1;

if (list2 != null) current.next = list2;

return dummy.next; // Return merged list (ignoring dummy)

}

}

**Output**



1. **remove Duplicates from sorted lists**

**Code**

class Solution {

public ListNode deleteDuplicates(ListNode head) {

if (head == null || head.next == null) return head;

if (head.next != null && head.val == head.next.val) {

while (head.next != null && head.val == head.next.val) {

head = head.next; // Skip duplicates

}

return deleteDuplicates(head.next); // Recur with next distinct node

} else {

head.next = deleteDuplicates(head.next); // Recur normally

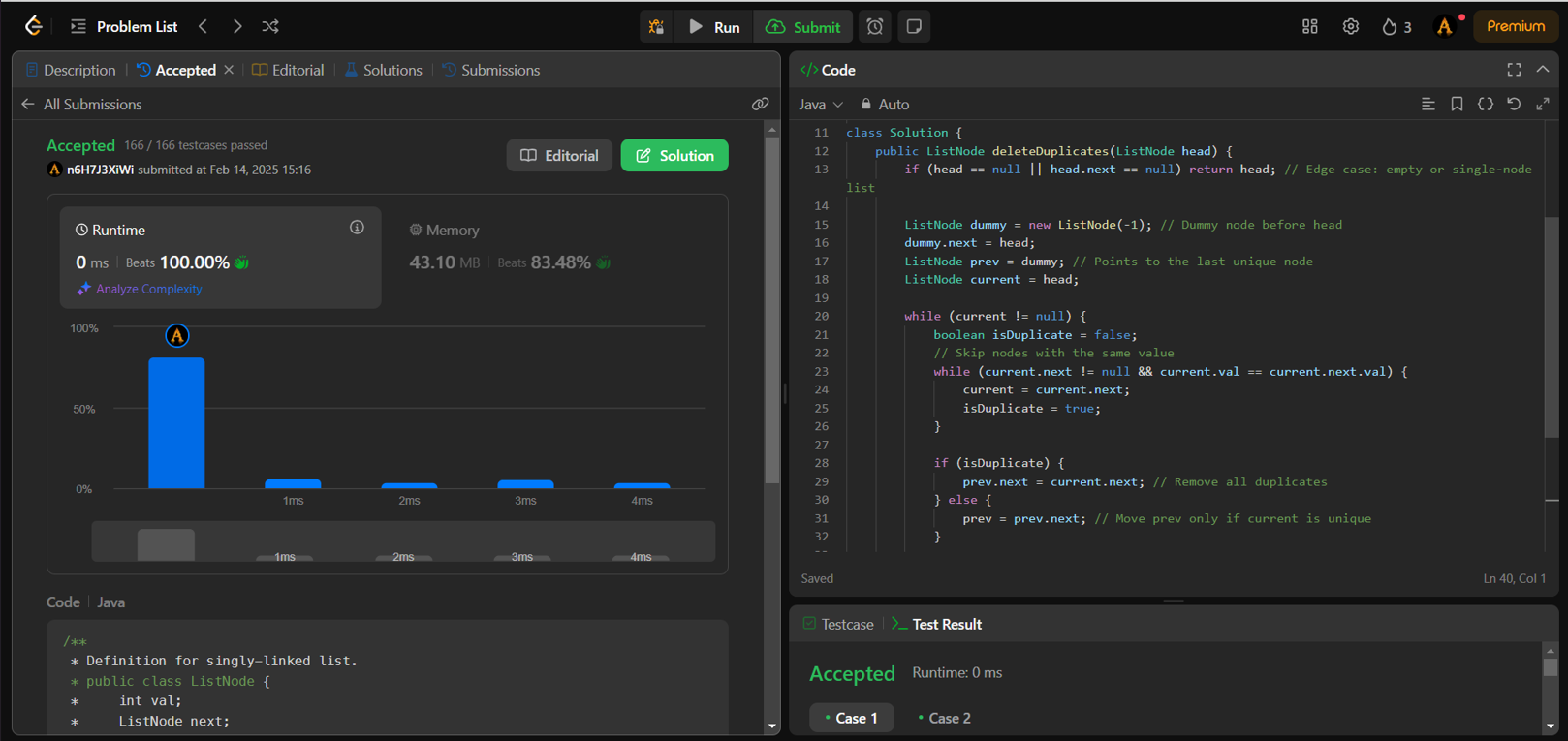
return head;

}

}

}

**Output**

****



1. **Detect a cycle in a linked list**

**Code**

public class Solution {

public boolean hasCycle(ListNode head) {

if (head == null || head.next == null) return false; // No cycle if empty or single-node list

ListNode slow = head;

ListNode fast = head;

while (fast != null && fast.next != null) {

slow = slow.next; // Move one step

fast = fast.next.next; // Move two steps

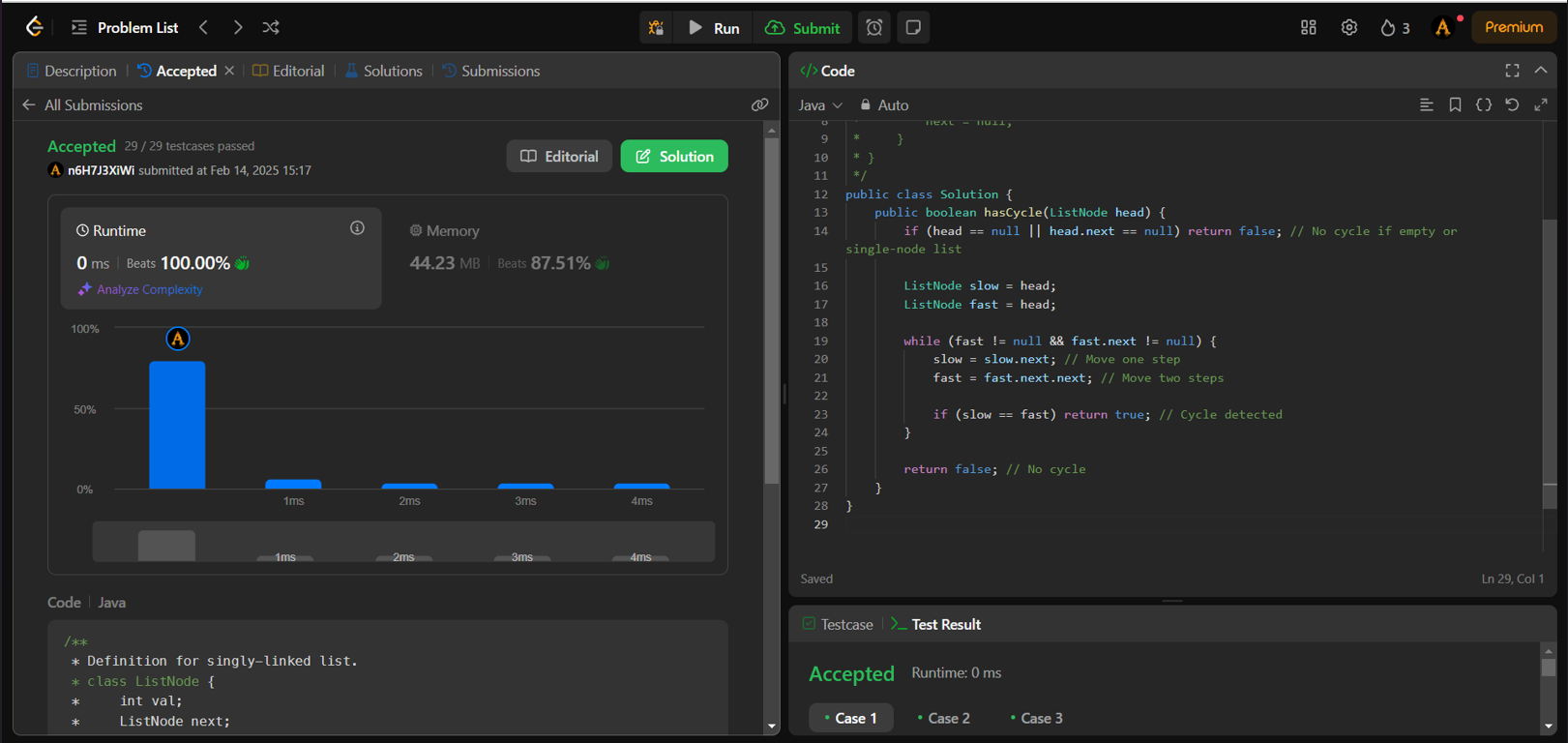
if (slow == fast) return true; // Cycle detected

}

return false; // No cycle

}

}

**Output**



**8. Reverse Linked list 2**

**Code**

class Solution {

public ListNode reverseBetween(ListNode head, int left, int right) {

if (head == null || left == right) return head; // No change needed

ListNode dummy = new ListNode(-1); // Dummy node to handle edge cases

dummy.next = head;

ListNode prev = dummy;

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

prev = prev.next;

}

ListNode curr = prev.next;

ListNode nextNode;

ListNode prevSubList = null;

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

nextNode = curr.next;

curr.next = prevSubList;

prevSubList = curr;

curr = nextNode;

}

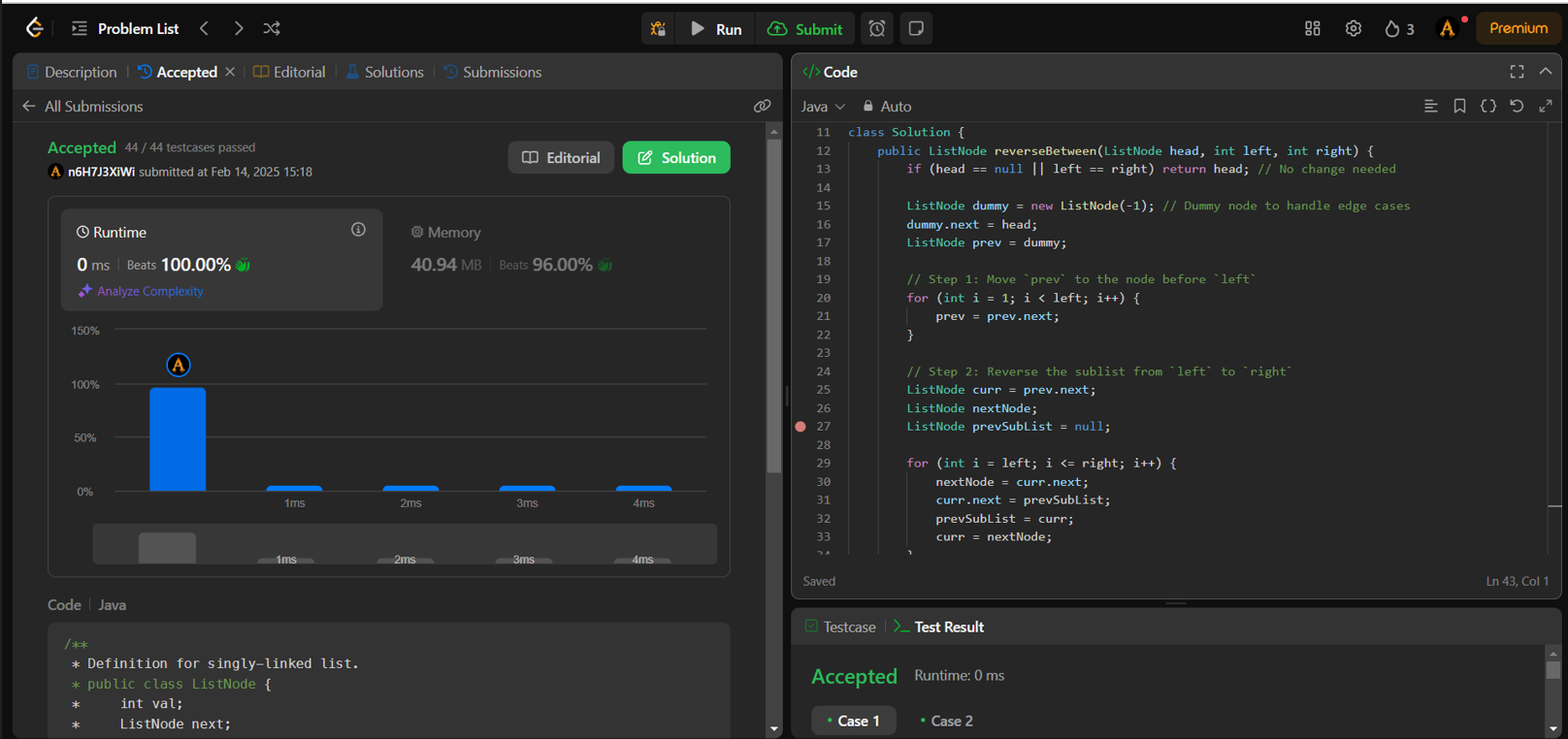
prev.next.next = curr; // Connect end of reversed part to the remaining list

prev.next = prevSubList; // Connect the node before `left` to the new head of reversed part

return dummy.next;

}

}

**output**



1. **Rotate a list**

**Code**

class Solution {

public ListNode rotateRight(ListNode head, int k) {

if (head == null || head.next == null || k == 0) return head; // Edge cases

ListNode temp = head;

int length = 1;

while (temp.next != null) {

temp = temp.next;

length++;

}

k = k % length;

if (k == 0) return head; // No change if k is a multiple of length

temp.next = head; // Connect last node to head (circular list)

ListNode newTail = head;

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

newTail = newTail.next;

}

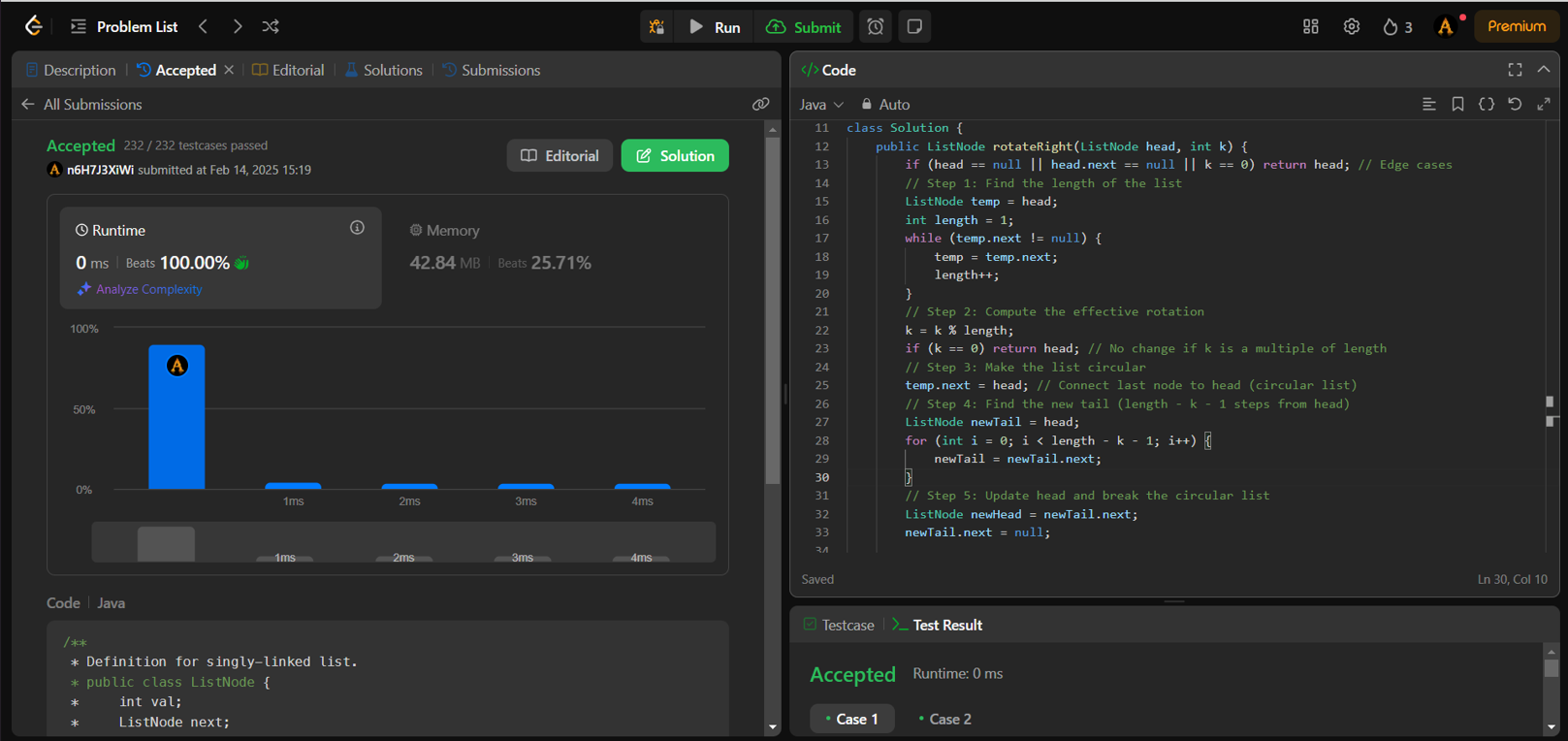
ListNode newHead = newTail.next;

newTail.next = null;

return newHead;

}

}

**Output**



1. **Sort list**

**Code**

class Solution {

public ListNode sortList(ListNode head) {

if (head == null || head.next == null) return head; // Base case

ListNode mid = getMiddle(head);

ListNode left = head;

ListNode right = mid.next;

mid.next = null; // Break the list into two parts

left = sortList(left);

right = sortList(right);

return merge(left, right);

}

private ListNode getMiddle(ListNode head) {

ListNode slow = head, fast = head.next;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow; // Middle node

}

private ListNode merge(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode current = dummy;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

current.next = l1;

l1 = l1.next;

} else {

current.next = l2;

l2 = l2.next;

}

current = current.next;

}

// Attach the remaining elements

if (l1 != null) current.next = l1;

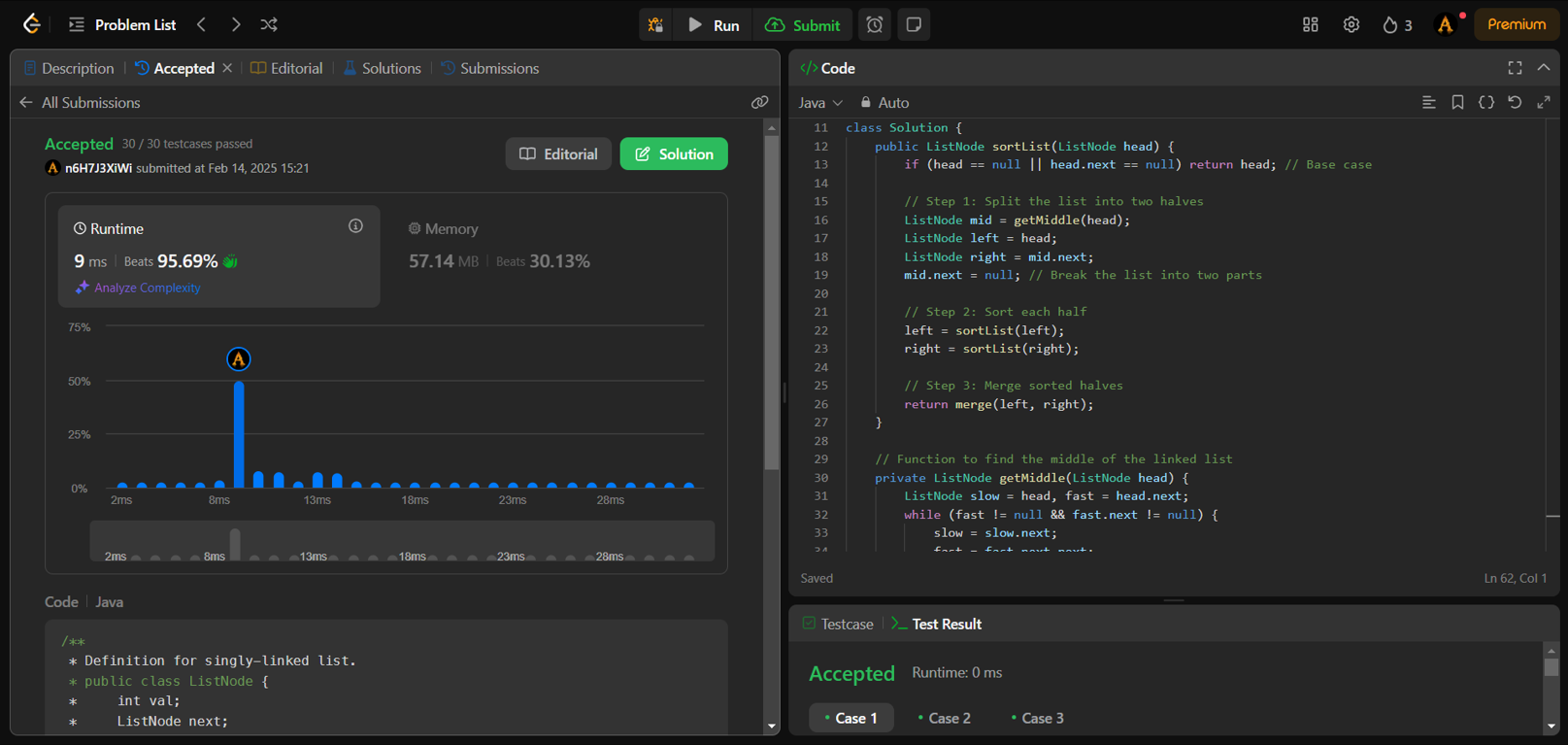
if (l2 != null) current.next = l2;

return dummy.next;

}

}



** Output**

1. **detect a cycle in Linked list**

**code**

public class Solution {

public ListNode detectCycle(ListNode head) {

if (head == null || head.next == null) return null; // Edge case

ListNode slow = head, fast = head;

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

while (fast != null && fast.next != null) {

slow = slow.next; // Move one step

fast = fast.next.next; // Move two steps

if (slow == fast) break; // Cycle detected

}

// If there is no cycle

if (fast == null || fast.next == null) return null;

// Step 2: Find the start of the cycle

slow = head; // Move slow to head

while (slow != fast) { // Move both one step at a time



slow = slow.next;

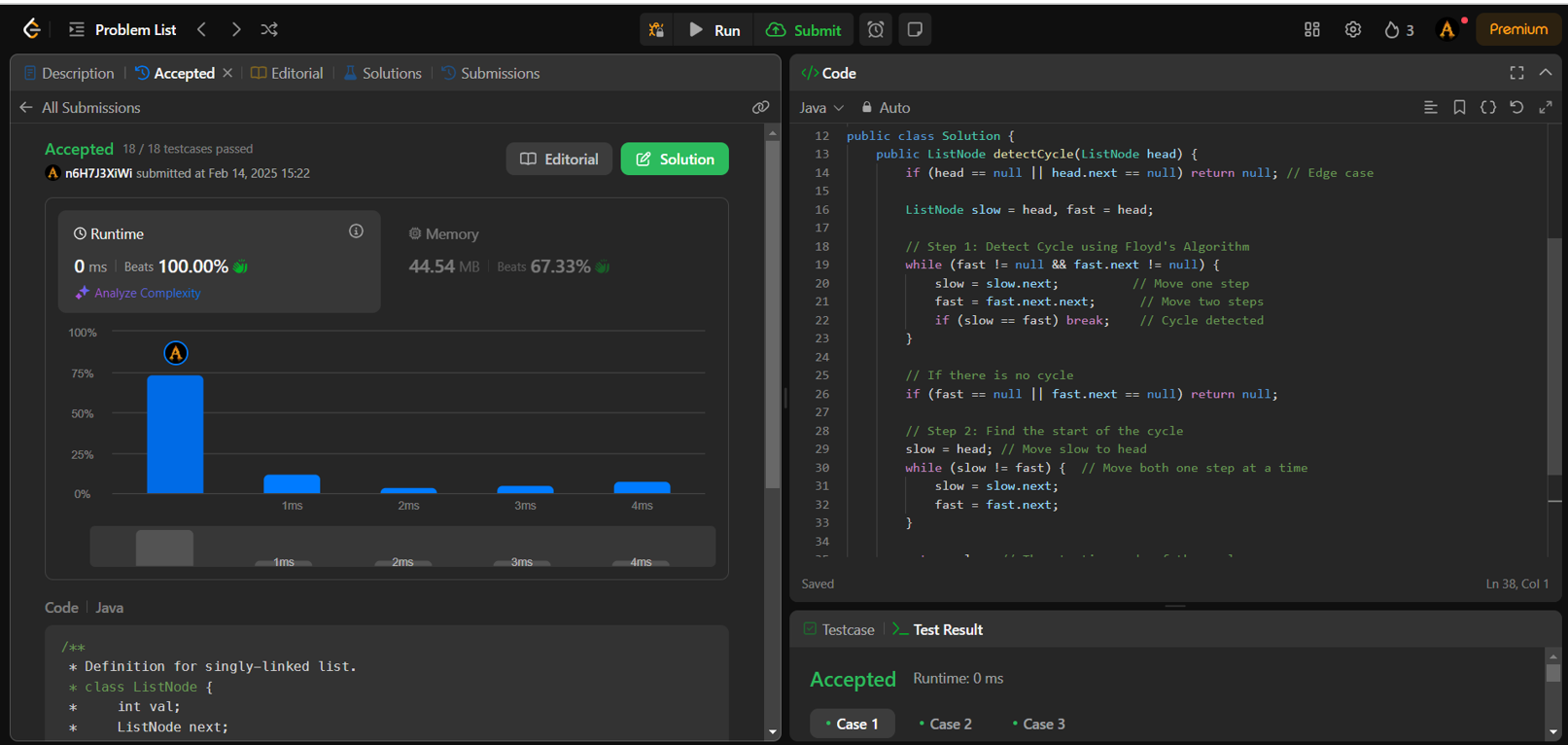
fast = fast.next;

}

return slow; // The starting node of the cycle

}

}

**Output**