Name: Aanchal

Negi UID: 22BCS14969

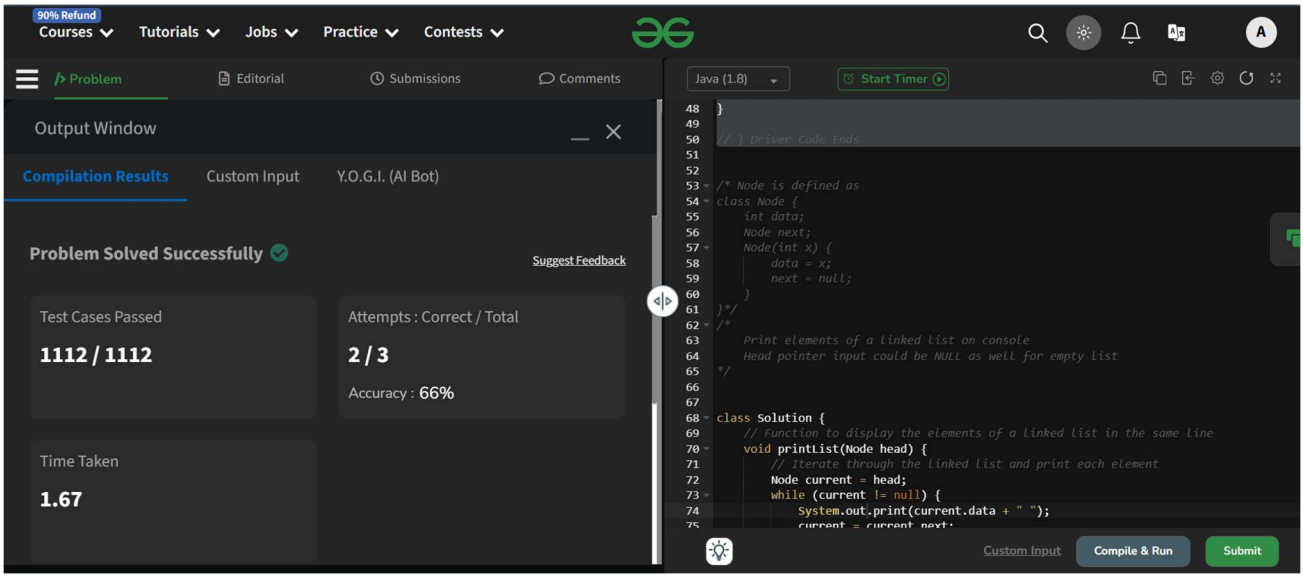
Section/Group: 609(B)

➢ Print Linked List

Code:

```
class Solution {
    // Function to display the elements of a linked list in the same line
    void printList(Node head) {
        // Iterate through the linked list and print each element
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
    }
}
```

Output:

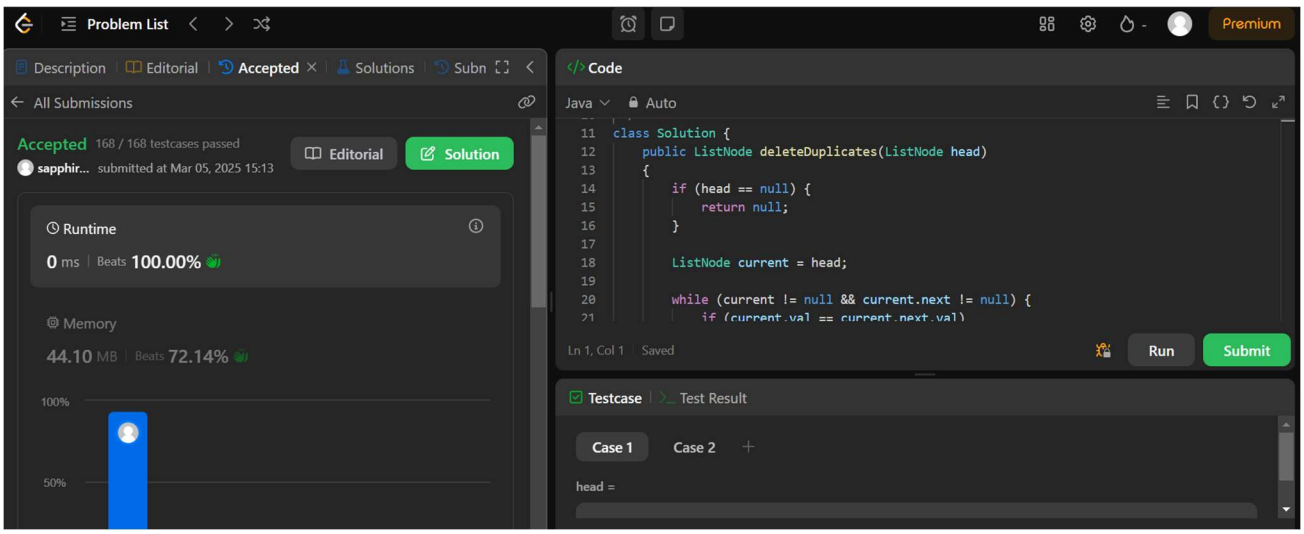


➢ Remove duplicates from a sorted list

Code:

```
class Solution {
    public ListNode deleteDuplicates(ListNode head)
    {
        if (head == null) {
            return null;
        }

        ListNode current = head;

        while (current != null && current.next != null) {
            if (current.val == current.next.val)
            {
                current.next = current.next.next;
            }
            else
            {
                current = current.next;
            }
        }

        return head;
    }
}
```

Output:

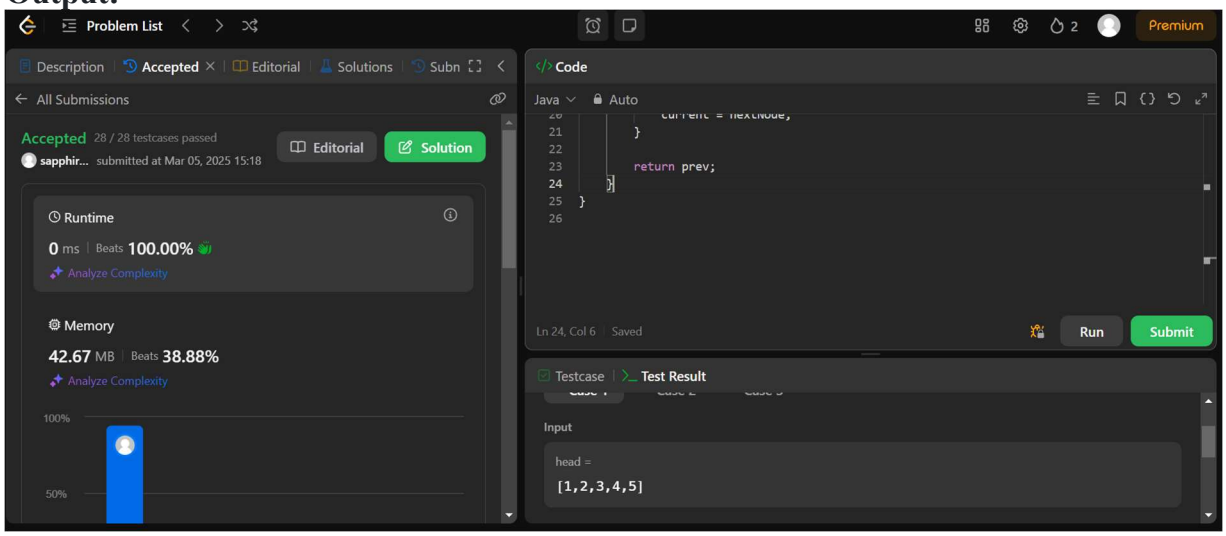


➢ Reverse Linked List

Code:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;

        while (current != null) {
            ListNode nextNode = current.next;
            current.next = prev;
            prev = current;
            current = nextNode;
        }

        return prev;
    }
}
```

Output:

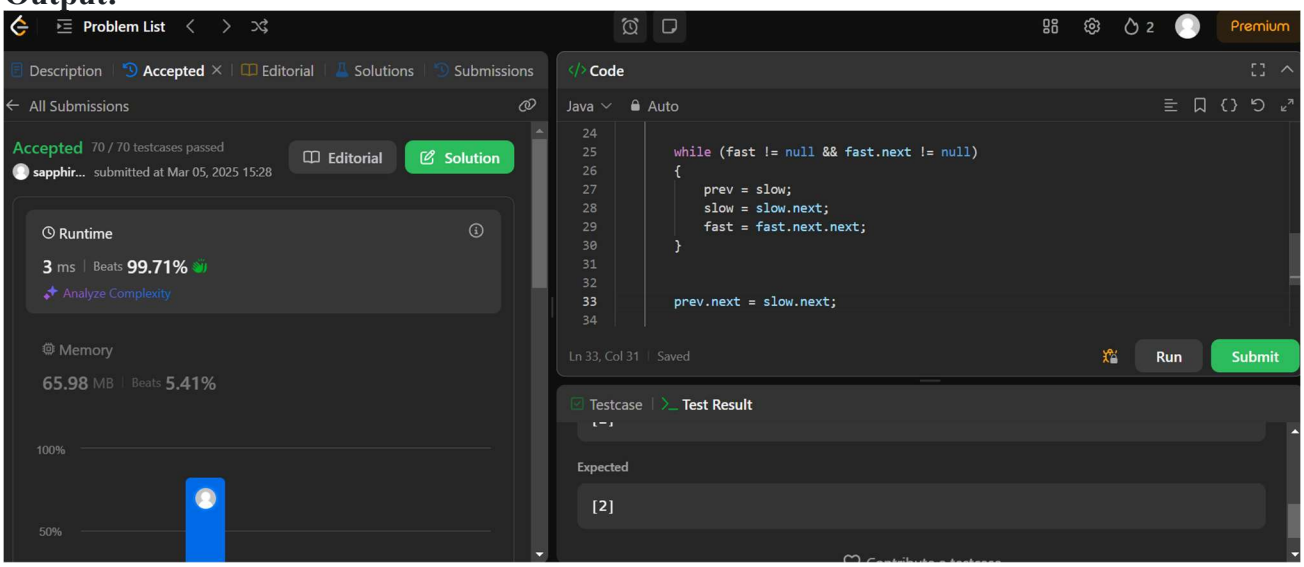


➢ Delete middle node of a list

Code:

```
class Solution {
    public ListNode deleteMiddle(ListNode head)
    {
        if (head == null || head.next == null) {
            return null;
        }


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


        while (fast != null && fast.next != null) {
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }


        prev.next = slow.next;

        return head;
    }
}
```

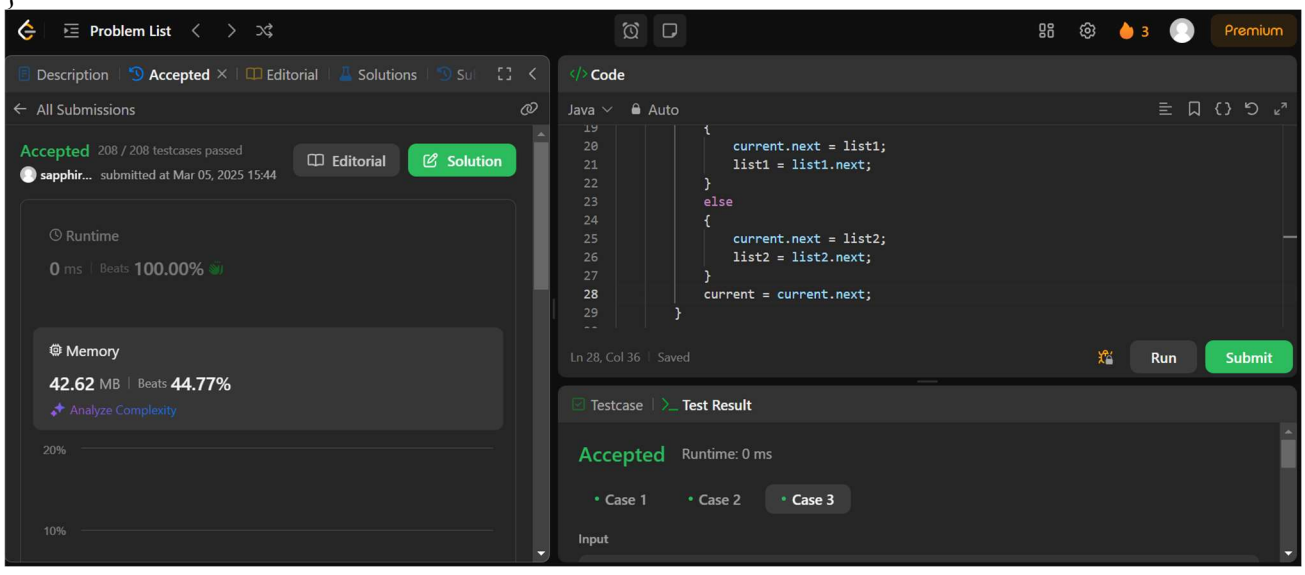
Output:



➢ Merge Two Sorted Lists

Code:

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode dummy = new ListNode();
        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;
        }
        else
        {
            current.next = list2;
        }


        return dummy.next;
    }
}
```

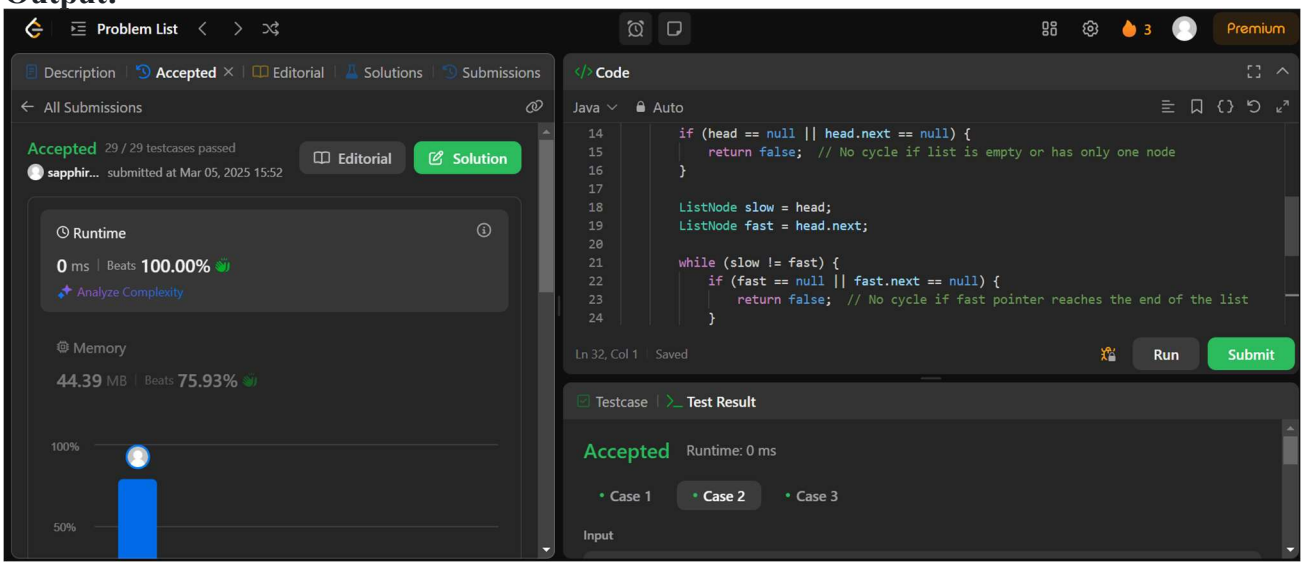


➢ Detect a cycle in a linked list

Code:

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;  // No cycle if list is empty or has only one node
        }

        ListNode slow = head;
        ListNode fast = head.next;

        while (slow != fast) {
            if (fast == null || fast.next == null) {
                return false;  // No cycle if fast pointer reaches the end of the list
            }
            slow = slow.next;
            fast = fast.next.next;
        }

        return true;  // Cycle detected if slow pointer meets fast pointer
    }
}
```

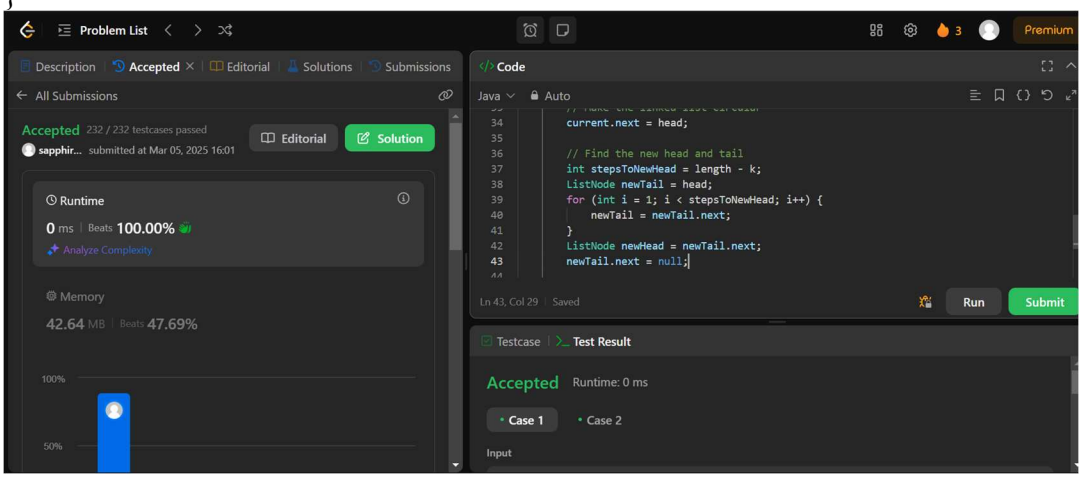
Output:



➢ Rotate a list

Code:

```
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null || head.next == null || k == 0) {
            return head;
        }

        // Calculate the length of the linked list
        ListNode current = head;
        int length = 1;
        while (current.next != null) {
            current = current.next;
            length = length + 1;
        }

        // Adjust k if it is greater than the length
        k = k % length;

        // If k is 0, no rotation is needed
        if (k == 0) {
            return head;
        }

        // Make the linked list circular
        current.next = head;

        // Find the new head and tail
        int stepsToNewHead = length - k;
        ListNode newTail = head;
        for (int i = 1; i < stepsToNewHead; i++) {
            newTail = newTail.next;
        }
        ListNode newHead = newTail.next;
        newTail.next = null;

        return newHead;
    }
}
```



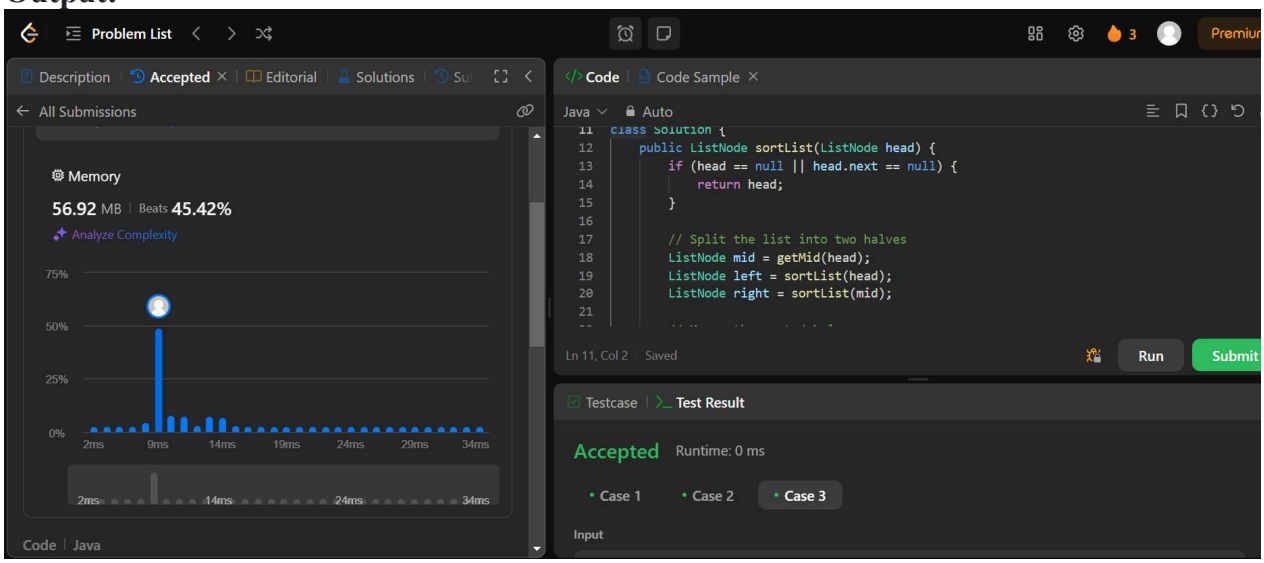
➢ Sort List

Code:

```
class Solution {
    public ListNode sortList(ListNode head) {
        if(head == null || head.next == null)
            return head;

        ListNode left = head;
        ListNode mid = findMid(left);
        ListNode right = mid.next;
        mid.next = null;

        left = sortList(left);
        right = sortList(right);

        return merge(left, right);
    }

    private ListNode findMid(ListNode node) {
        ListNode slow = node;
        ListNode fast = node.next;

        while(fast != null && fast.next!=null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        return slow;
    }

    private ListNode merge(ListNode left, ListNode right) {
        ListNode node = new ListNode();
        ListNode merged = node;
        while(left != null && right != null) {
            if(left.val < right.val) {
                node.next = left;
                left = left.next;
            } else {
                node.next = right;
                right = right.next;
            }
            node = node.next;
        }

        if(left!=null) {
            node.next = left;
            node = node.next;
        }

        if(right!=null) {
            node.next = right;
            node = node.next;
        }
```

```
        return merged.next;
    }
}
```

Output:



➢ Merge k sorted lists :

Code:

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }

        // Create a min-heap (priority queue) to store the nodes
        PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val - b.val);

        // Add the head of each list to the min-heap
        for (ListNode node : lists) {
            if (node != null) {
                minHeap.add(node);
            }
        }

        // Create a dummy node to build the merged list
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        // Extract the smallest node from the min-heap and add it to the merged list
        while (!minHeap.isEmpty()) {
            ListNode smallest = minHeap.poll();
            current.next = smallest;
            current = current.next;

            // If the extracted node has a next node, add it to the min-heap
```

```
            if (smallest.next != null) {
                minHeap.add(smallest.next);
            }
        }

        return dummy.next;
    }
}
```

Output: