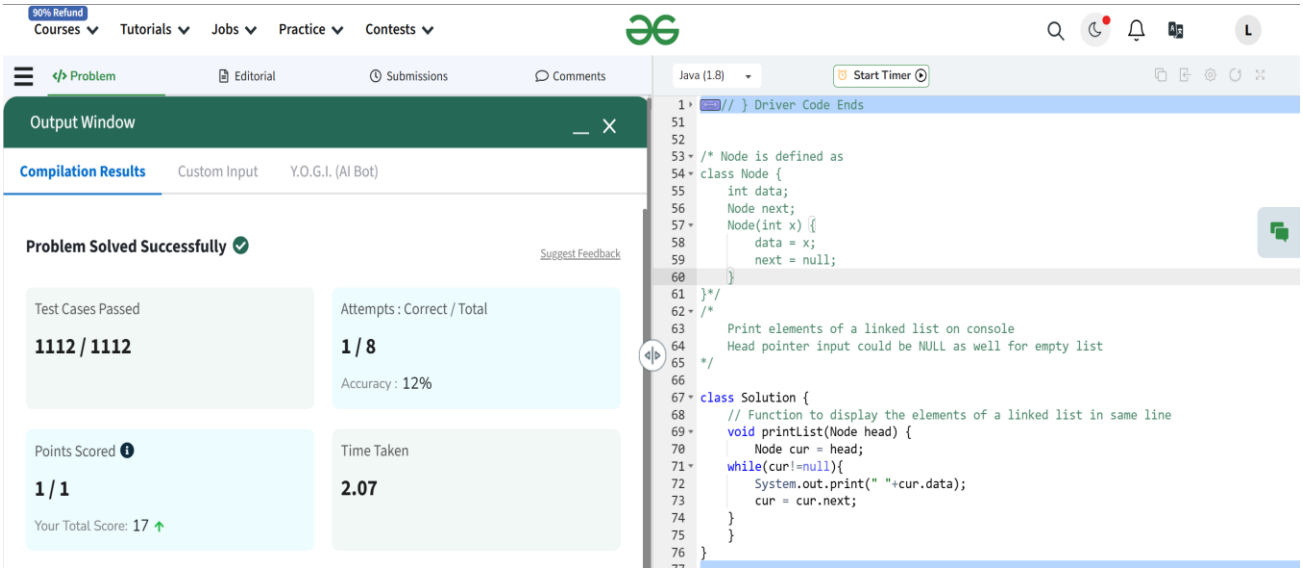
Student Name: Lovely Sharma **UID: 22BCS11001**
Branch: BE-CSE **Section/Group: 22BCS_IOT_610_B**
Semester: 6th **Date of Performance: 05/03/25**
Subject Name: Advanced Programming II **Subject Code: 22CSP-351**

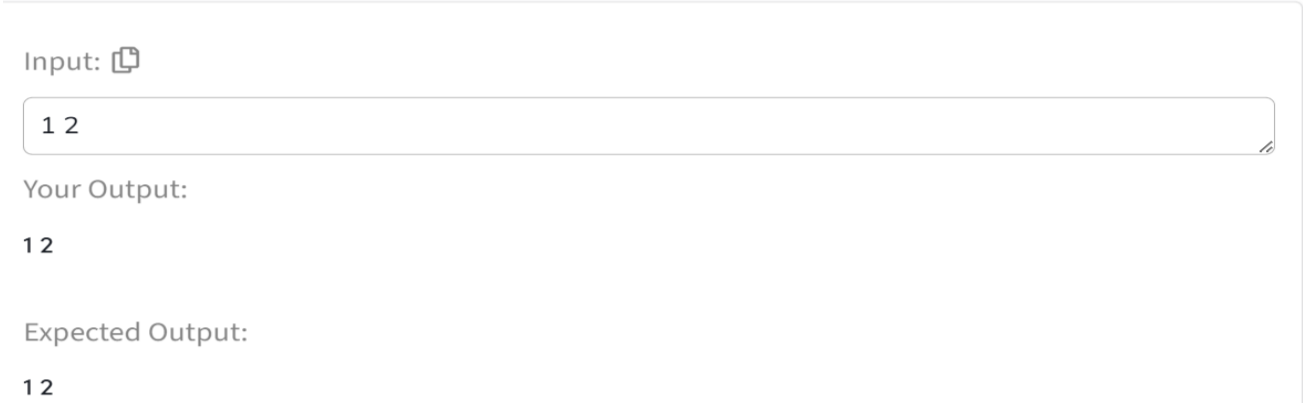
1. Print Linked List:

a) Code:

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

b) Output:



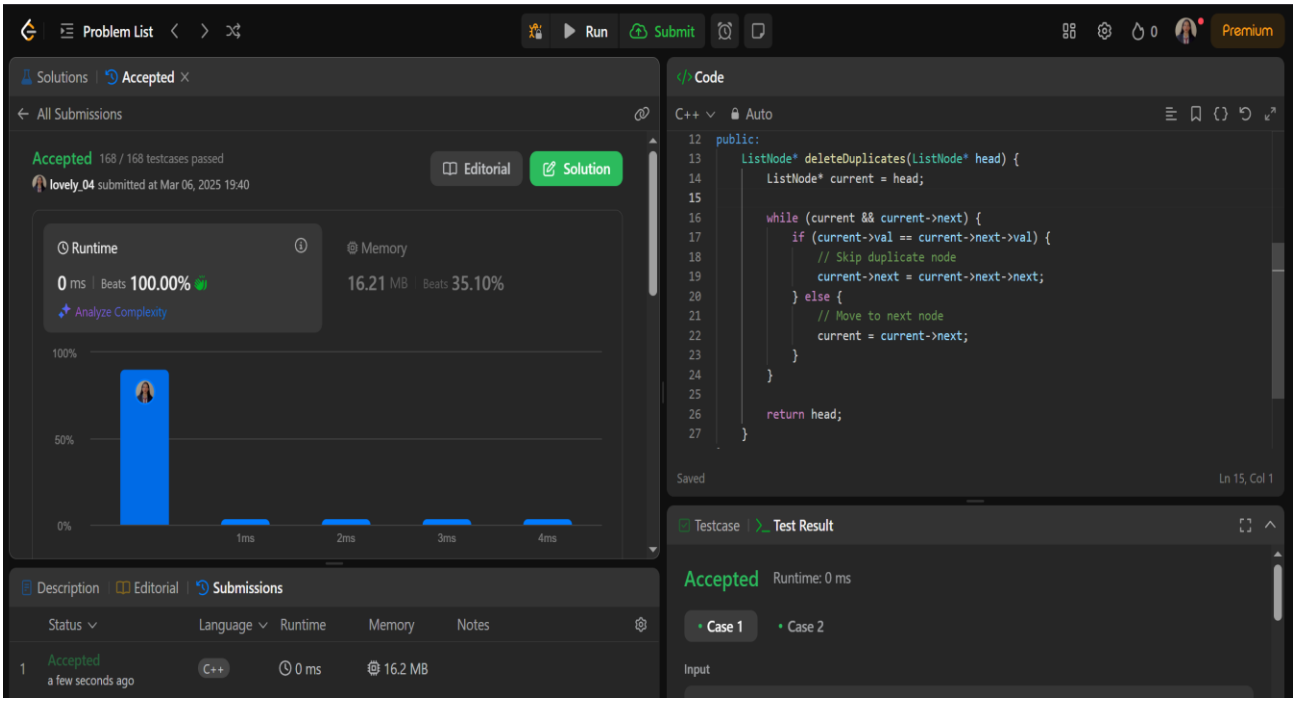


2. Remove duplicates from a sorted list:

a) Code:

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* current = head;

        while (current && current->next) {
            if (current->val == current->next->val) {
                // Skip duplicate node
                current->next = current->next->next;
            } else {
                // Move to next node
                current = current->next;
            }
        }

        return head;
    }
};
```

b) Output:

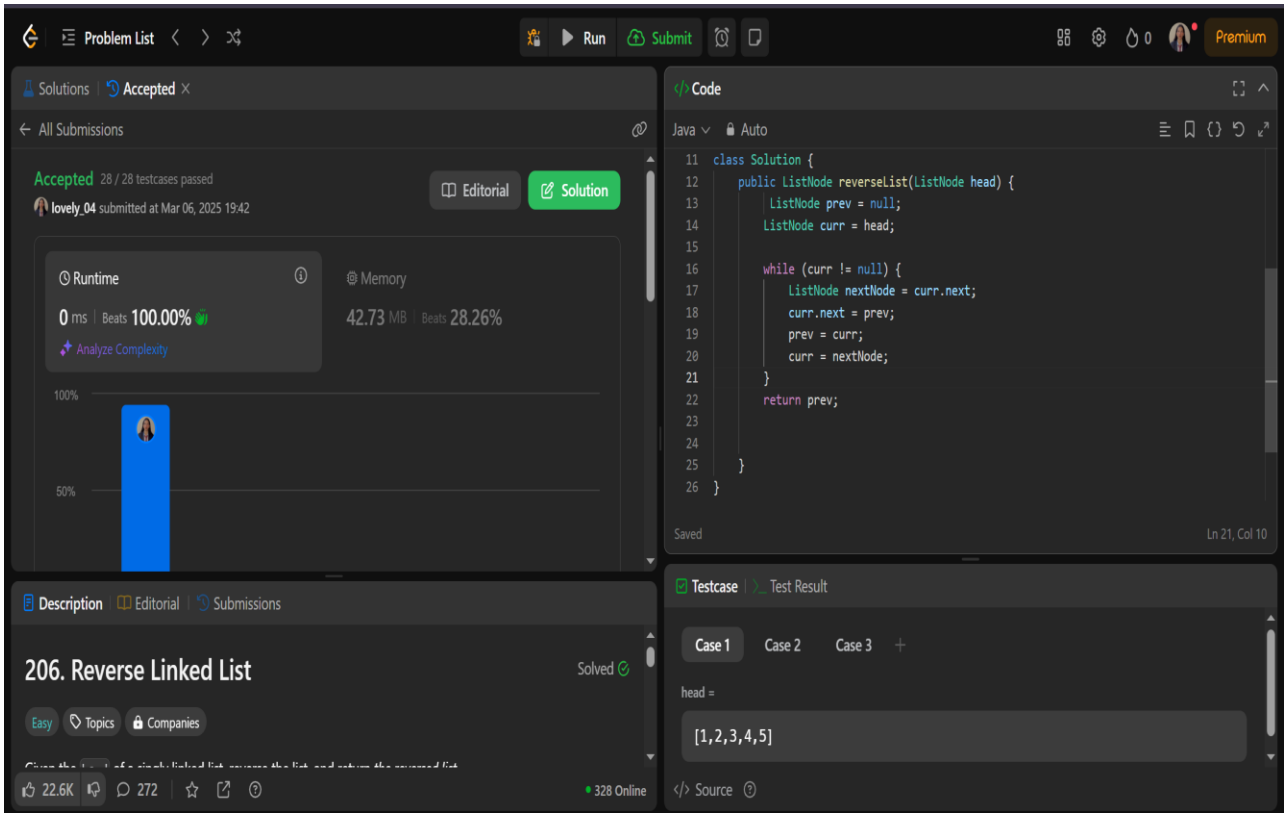


3. Reverse a linked list:

a) Code:

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

      while (curr != null) {
         ListNode nextNode = curr.next;
         curr.next = prev;
         prev = curr;
         curr = nextNode;
      }
      return prev;
   }
}
```

b) Output:

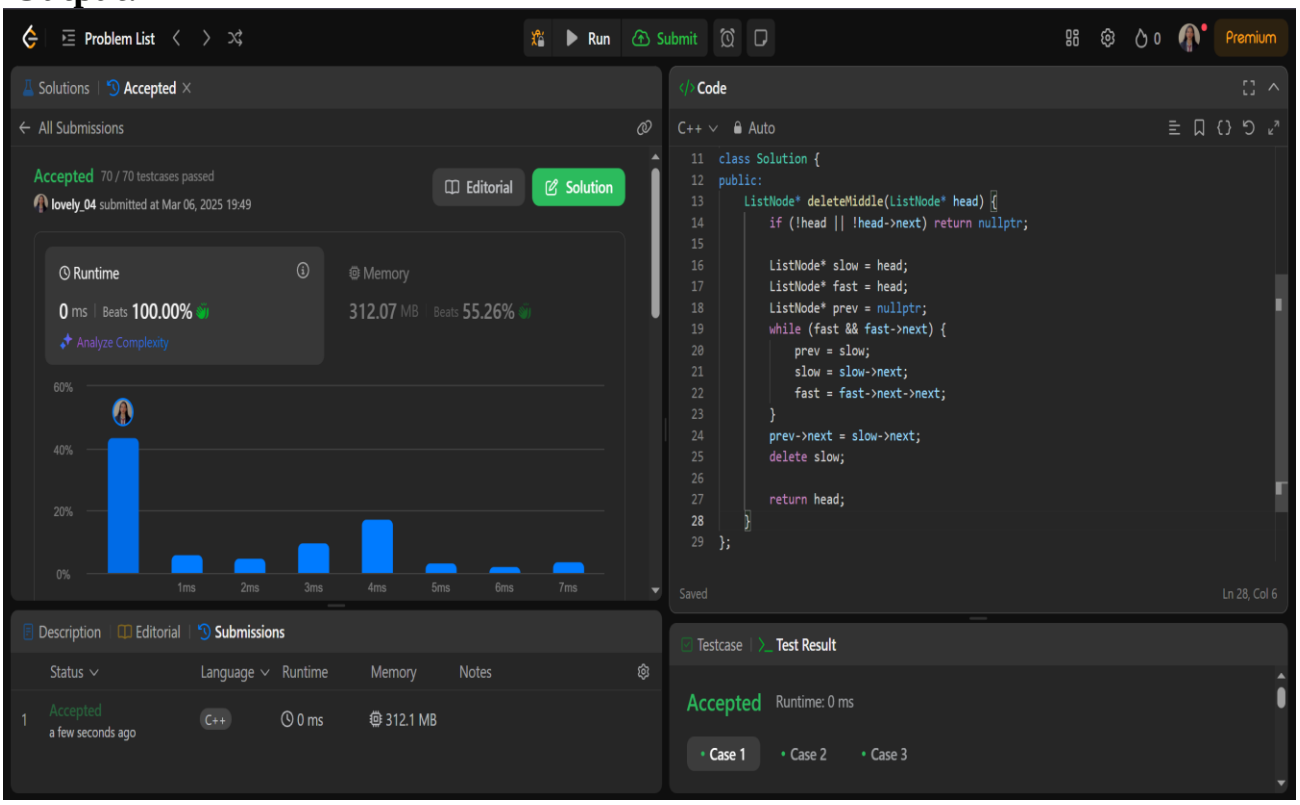


4. Delete middle node of a list:

a) Code:

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;

        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* prev = nullptr;
        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = slow->next;
        delete slow;

        return head;
    }
};
```

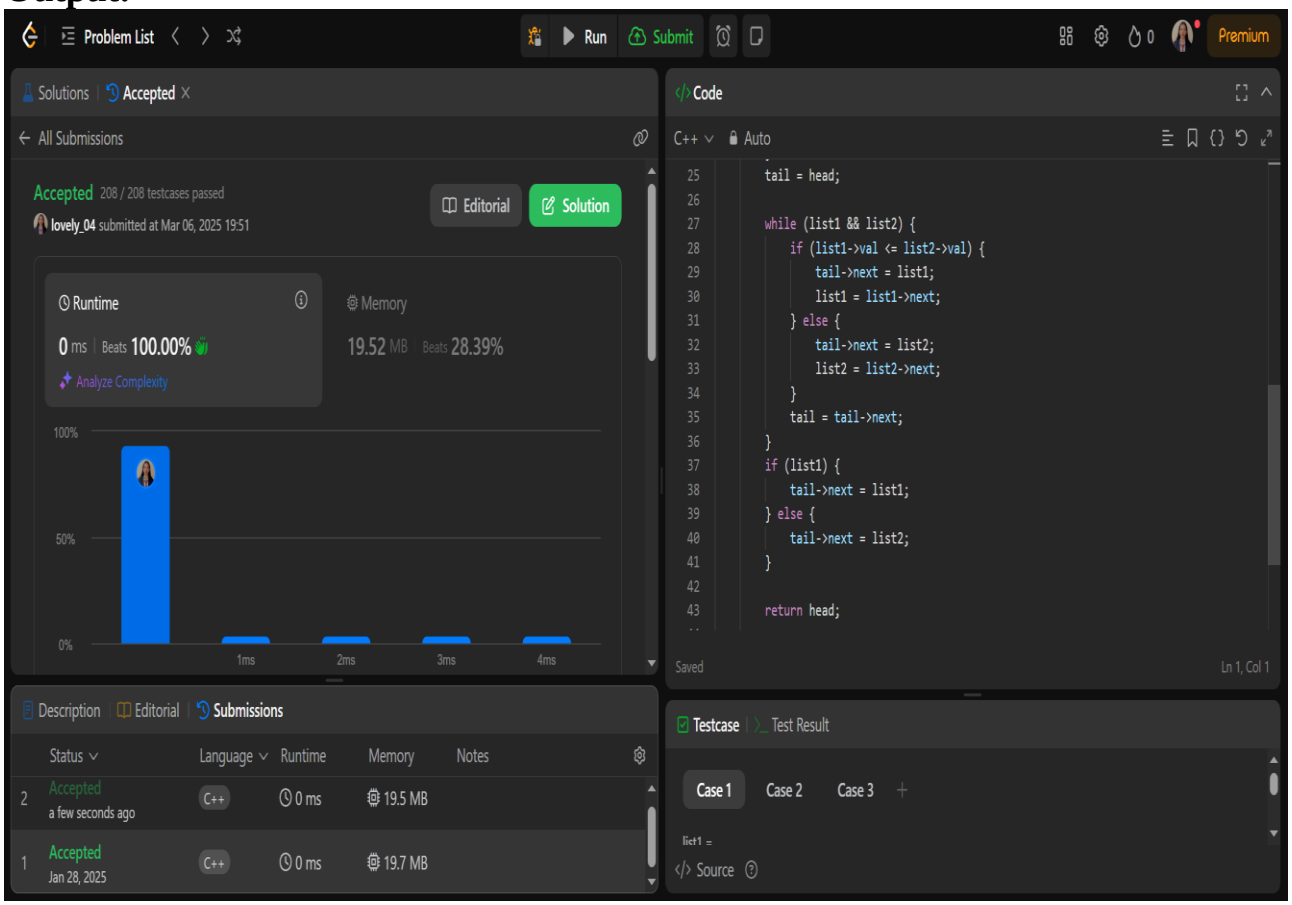
b) Output:



5. **Merge two sorted linked lists:**
   a) **Code:**

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if (!list1) return list2;
        if (!list2) return list1;
        ListNode* head = nullptr;
        ListNode* tail = nullptr;
        if (list1->val <= list2->val) {
            head = list1;
            list1 = list1->next;
        } else {
            head = list2;
            list2 = list2->next;
        }
        tail = head;

        while (list1 && list2) {
            if (list1->val <= list2->val) {
                tail->next = list1;
                list1 = list1->next;
            } else {
                tail->next = list2;
                list2 = list2->next;
            }
            tail = tail->next;
        }
        if (list1) {
            tail->next = list1;
        } else {
            tail->next = list2;
        }

        return head;

    }
};
```

b) Output:

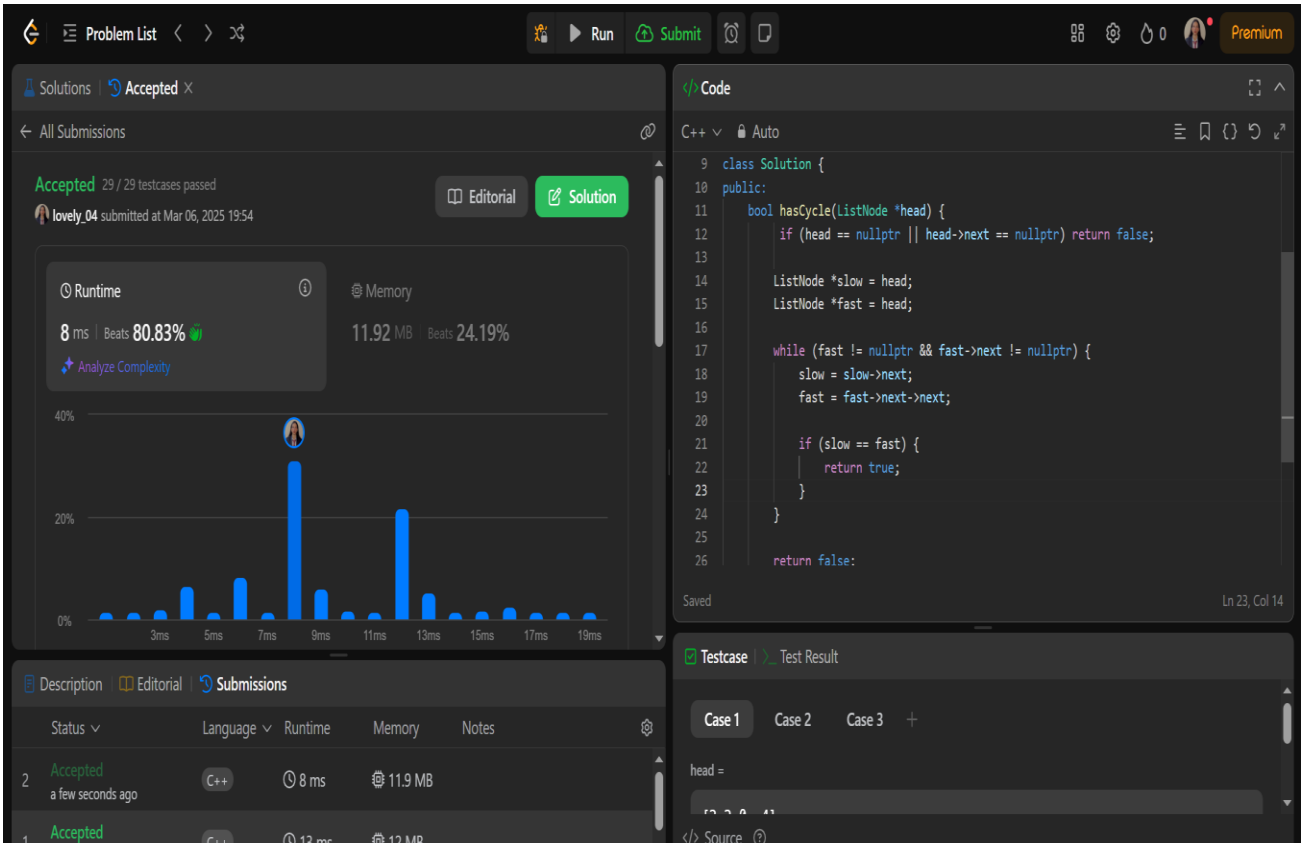


6. Detect a cycle in a linked list:

a) Code:

```
class Solution {
public:
   bool hasCycle(ListNode *head) {
       if (head == nullptr || head->next == nullptr) return false;

      ListNode *slow = head;
      ListNode *fast = head;

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

         if (slow == fast) {
            return true;
         }
      }
      return false;
   }};
```

b) Output:

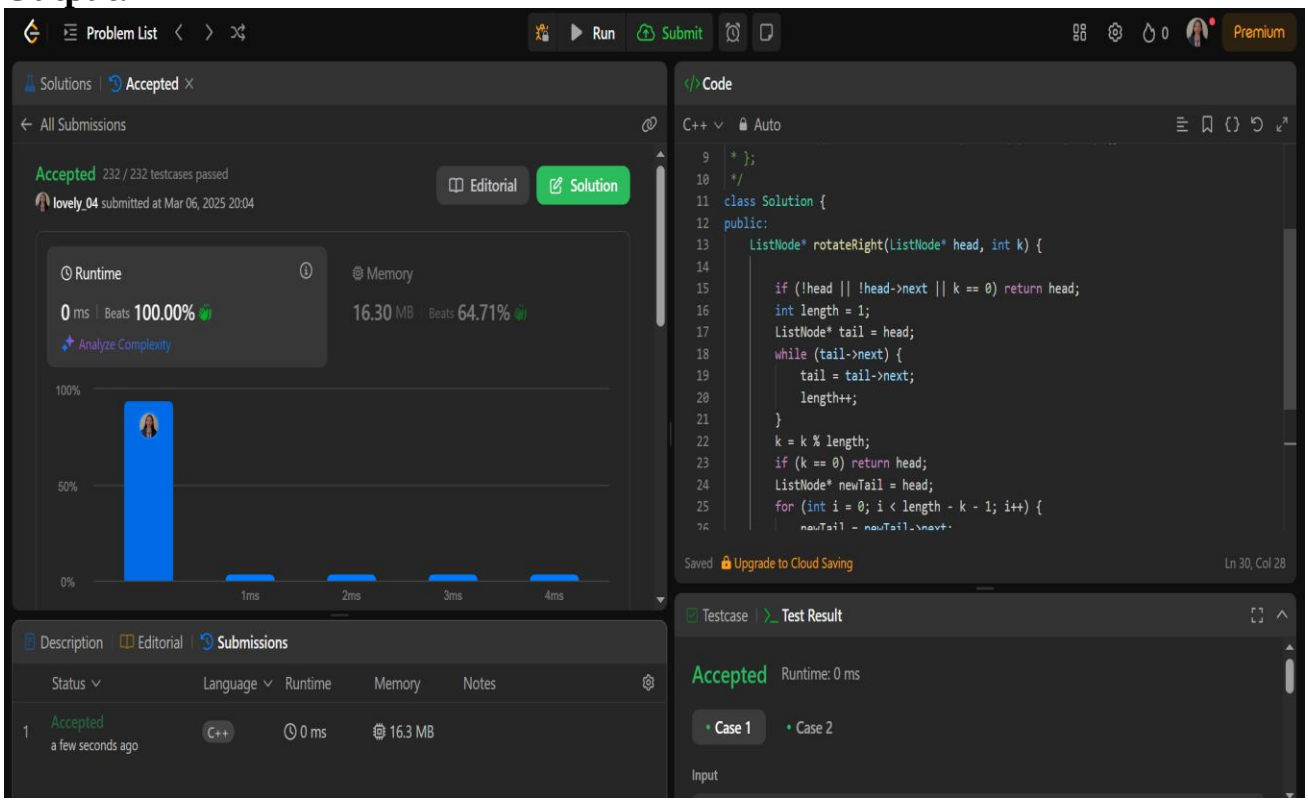


7. Rotate a list:

a) Code:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head;
        int length = 1;
        ListNode* tail = head;
        while (tail->next) {
            tail = tail->next;
            length++;
        }
        k = k % length;
        if (k == 0) return head;
        ListNode* newTail = head;
        for (int i = 0; i < length - k - 1; i++) {
            newTail = newTail->next;
        }
        ListNode* newHead = newTail->next;
        newTail->next = nullptr;
        tail->next = head;
        return newHead;   }};
```

b) Output:



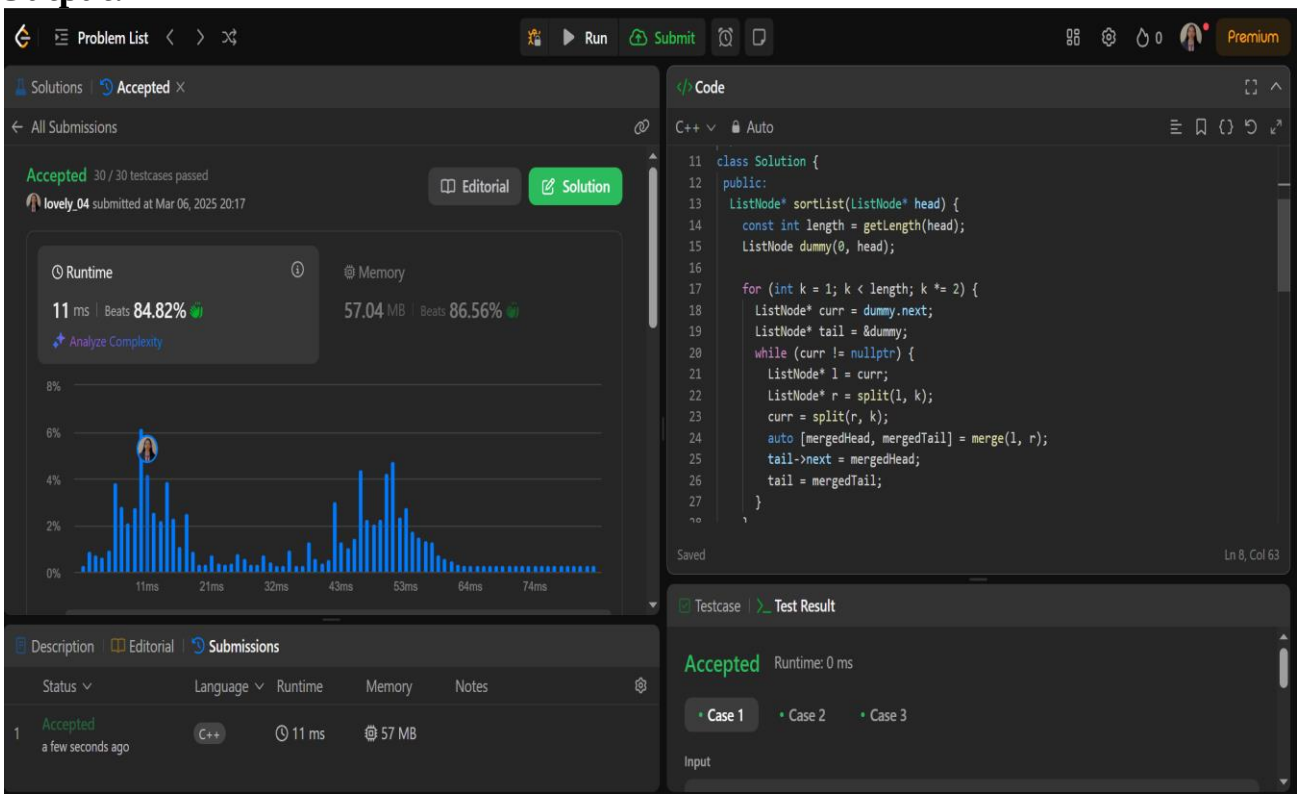
8. Sort List:

a) Code:

```
class Solution {
 public:
  ListNode* sortList(ListNode* head) {
    const int length = getLength(head);
    ListNode dummy(0, head);

    for (int k = 1; k < length; k *= 2) {
      ListNode* curr = dummy.next;
      ListNode* tail = &dummy;
      while (curr != nullptr) {
        ListNode* l = curr;
        ListNode* r = split(l, k);
        curr = split(r, k);
        auto [mergedHead, mergedTail] = merge(l, r);
        tail->next = mergedHead;
        tail = mergedTail;
      }
    }
    return dummy.next;
  }
 private:
  int getLength(ListNode* head) {
    int length = 0;
```

```
  for (ListNode* curr = head; curr; curr = curr->next)
    ++length;
  return length;
}
ListNode* split(ListNode* head, int k) {
  while (--k && head)
    head = head->next;
  ListNode* rest = head ? head->next : nullptr;
  if (head != nullptr)
    head->next = nullptr;
  return rest;
}
pair<ListNode*, ListNode*> merge(ListNode* l1, ListNode* l2) {
  ListNode dummy(0);
  ListNode* tail = &dummy;
  while (l1 && l2) {
    if (l1->val > l2->val)
      swap(l1, l2);
    tail->next = l1;
    l1 = l1->next;
    tail = tail->next;
  }
  tail->next = l1 ? l1 : l2;
  while (tail->next != nullptr)
    tail = tail->next;
  return {dummy.next, tail};}};
```

b) Output:

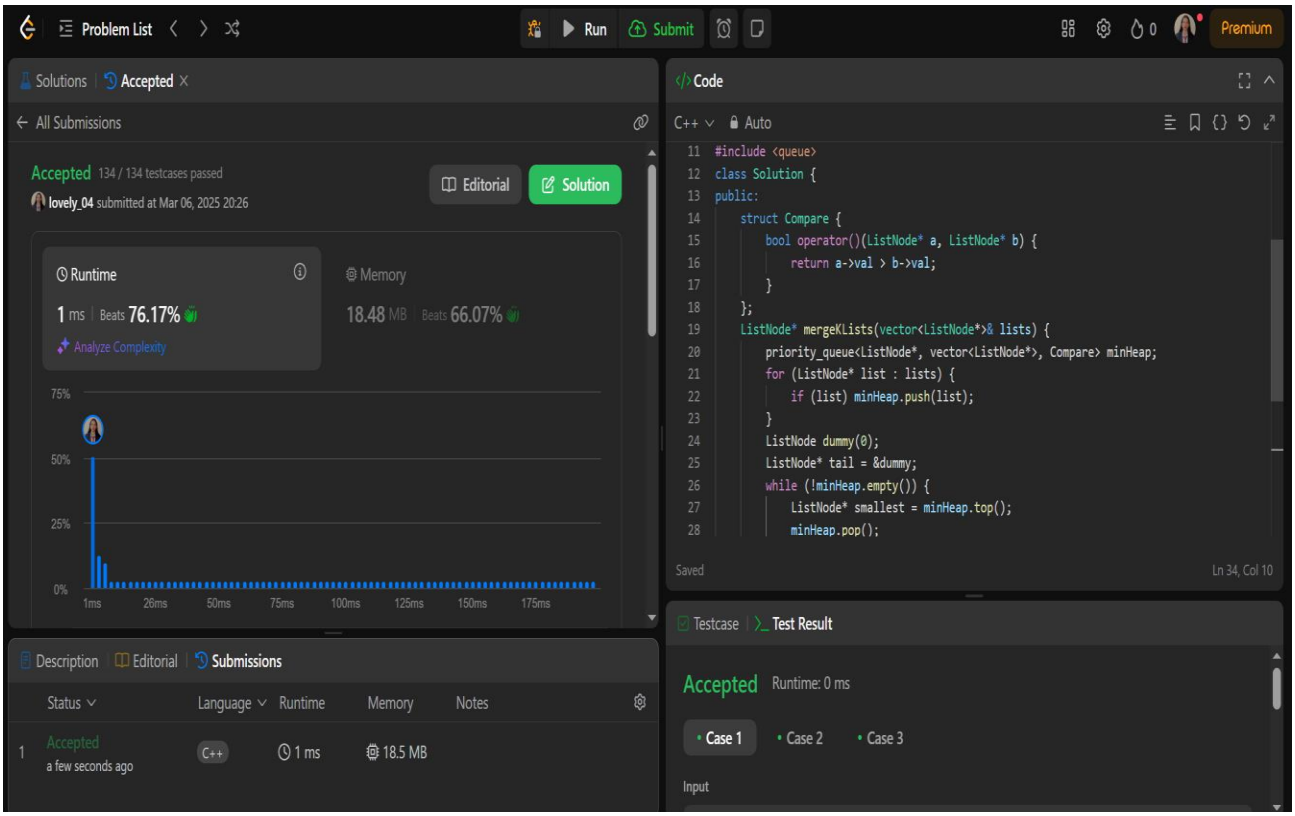


9. Merge k sorted lists:

a) Code:

```
#include <queue>
class Solution {
public:
    struct Compare {
        bool operator()(ListNode* a, ListNode* b) {
            return a->val > b->val;  // Min-Heap based on node values
        }
    };

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, Compare> minHeap;

        // Step 1: Push the head of each non-empty list into the minHeap
        for (ListNode* list : lists) {
            if (list) minHeap.push(list);
        }

        ListNode dummy(0);
        ListNode* tail = &dummy;

        // Step 2: Process the heap
        while (!minHeap.empty()) {
            ListNode* smallest = minHeap.top();
            minHeap.pop();

            tail->next = smallest;
            tail = tail->next;

            if (smallest->next) minHeap.push(smallest->next);
        }
        return dummy.next;
    }
};
```

b) Output:


Problem List
Run
Submit
Premium
Solutions
Accepted
All Submissions
Accepted 134 / 134 testcases passed
lovely_04 submitted at Mar 06, 2025 20:26
Editorial
Solution
Runtime
1 ms Beats 76.17%
Analyze Complexity
Memory
18.48 MB Beats 66.07%
75%
50%
25%
0%
1ms
26ms
50ms
75ms
100ms
125ms
150ms
175ms
Description
Editorial
Submissions
Status
Language
Runtime
Memory
Notes
1
Accepted
a few seconds ago
C++
1 ms
18.5 MB
Code
C++
Auto
#include <queue>
class Solution {
public:
struct Compare {
bool operator()(ListNode* a, ListNode* b) {
return a->val > b->val;
}
};
ListNode* mergeKLists(vector<ListNode*>& lists) {
priority_queue<ListNode*, vector<ListNode*>, Compare> minHeap;
for (ListNode* list : lists) {
if (list) minHeap.push(list);
}
ListNode dummy(0);
ListNode* tail = &dummy;
while (!minHeap.empty()) {
ListNode* smallest = minHeap.top();
minHeap.pop();
Saved
Ln 34, Col 10
Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input