Gaurav Jangir

22BCS50153

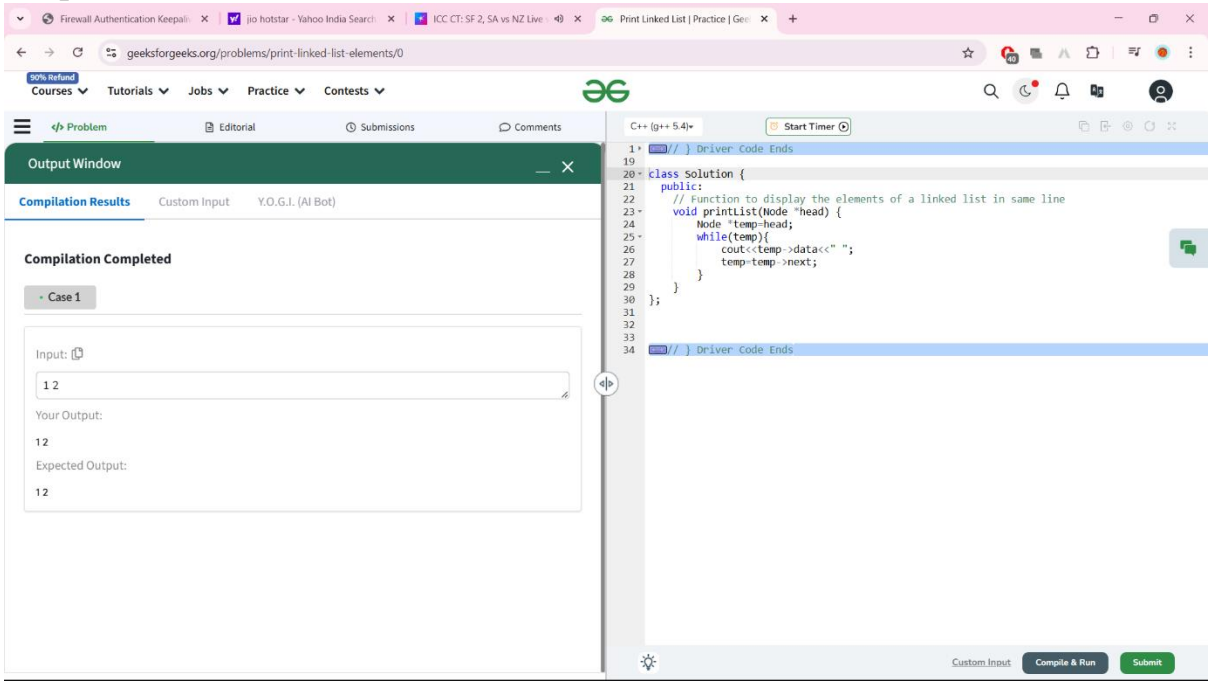
Assignment 03

Advanced Programming

1. 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){
            cout<<temp->data<<" ";
            temp=temp->next;
        }
    }
};
```

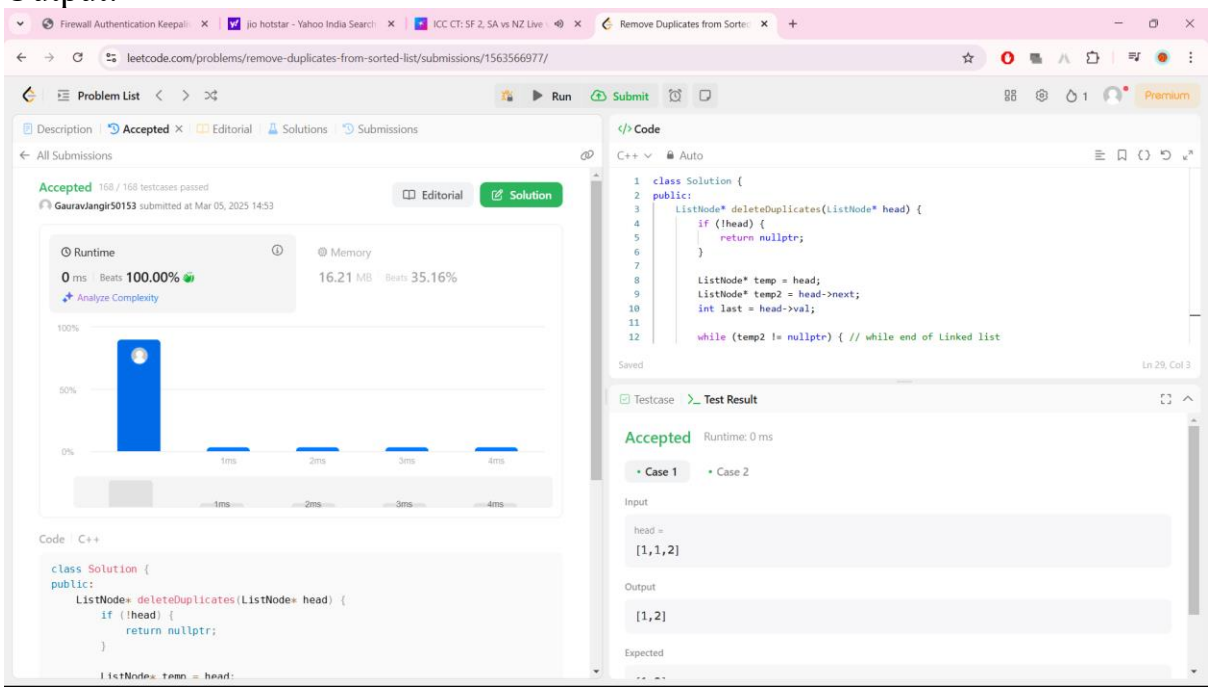
Output:



2. Remove duplicates from a sorted list:

```
class Solution {
public:
   ListNode* deleteDuplicates(ListNode* head) {
      if (!head) {
         return nullptr;
      }

      ListNode* temp = head;
      ListNode* temp2 = head->next;
      int last = head->val;

      while (temp2 != nullptr) { // while end of Linked list
         if (temp2->val == last) { // Current number same as last number
            if (temp2->next == nullptr) { // If last element, just delete and break
loop
               temp->next = nullptr;
               break;
            }
            temp2 = temp2->next; // Not last, then delete that element
            temp->next = temp2; // and move to next element
         } else { // If not the same as last element, jump to next node
            temp = temp2;
            last = temp->val;
            temp2 = temp2->next;
         }
      }

      return head; // return the head back
   }
};
```

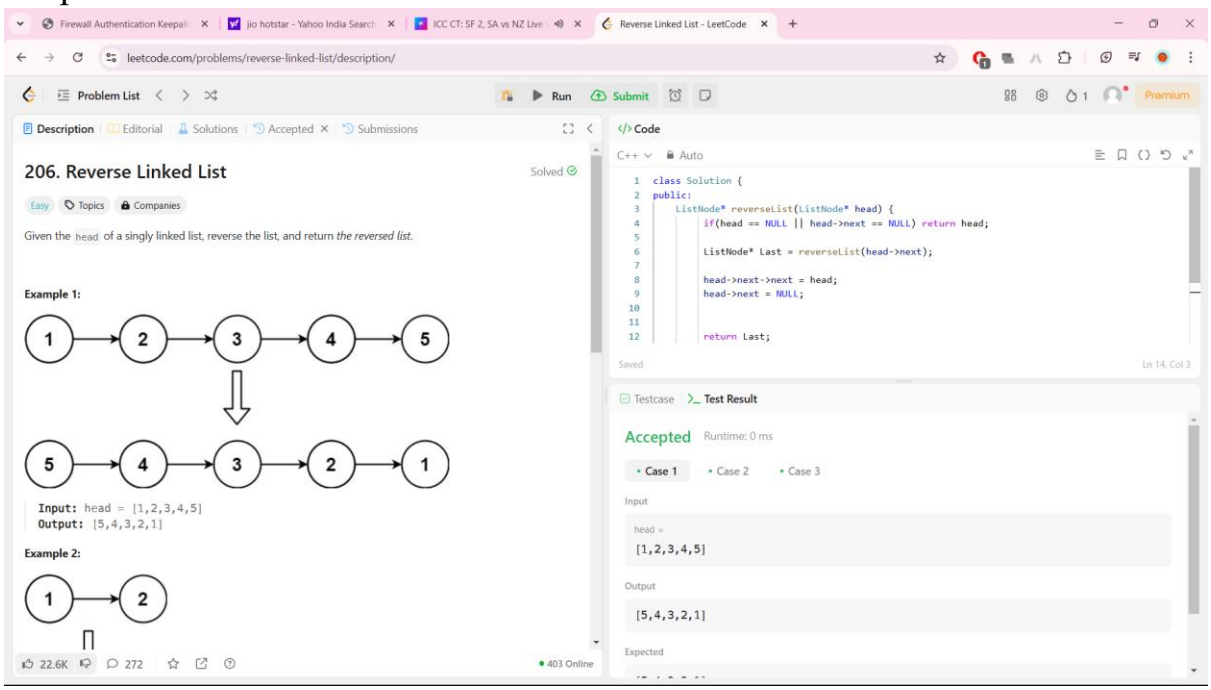
Output:



3. Reverse a linked list:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(head == NULL || head->next == NULL) return head;

        ListNode* Last = reverseList(head->next);

        head->next->next = head;
        head->next = NULL;

        return Last;
    }
};
```

Output:

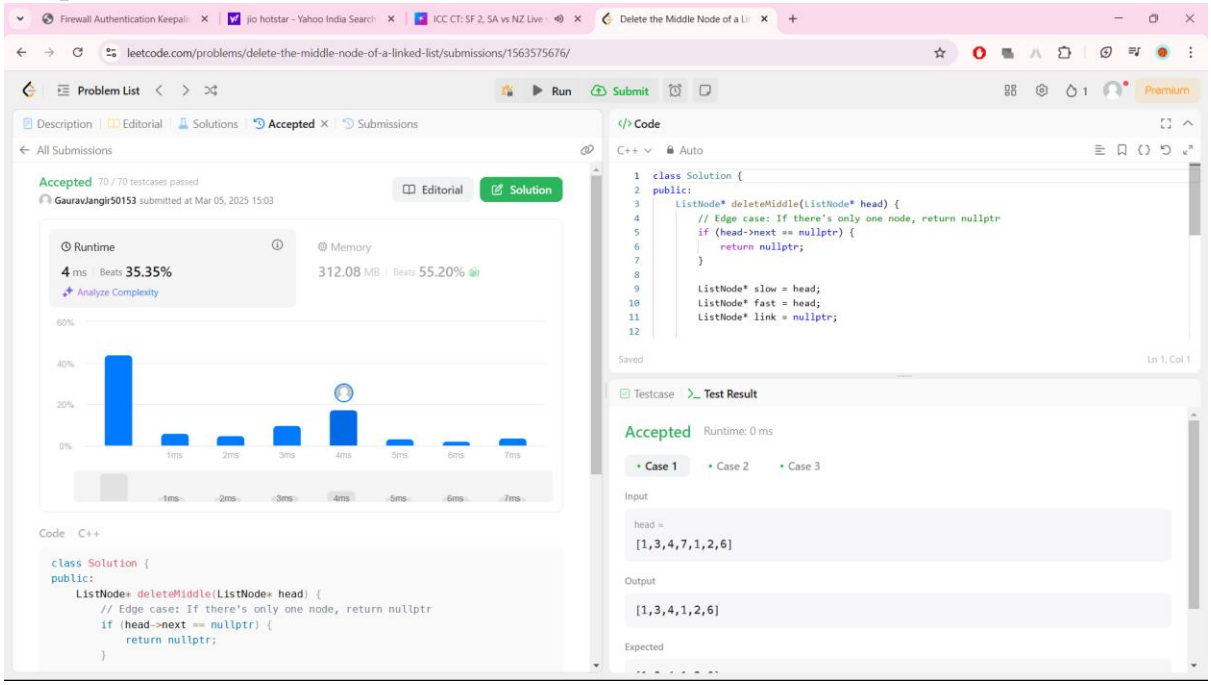


4. Delete middle node of a list:

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        // Edge case: If there's only one node, return nullptr
        if (head->next == nullptr) {
            return nullptr;
        }

        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* link = nullptr;

        // Find the middle node using the slow and fast pointer technique
        while (fast != nullptr && fast->next != nullptr) {
            link = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
```

```
        // Deleting the middle node by skipping over it
        link->next = slow->next;

        return head;
    }
};
```

Output:

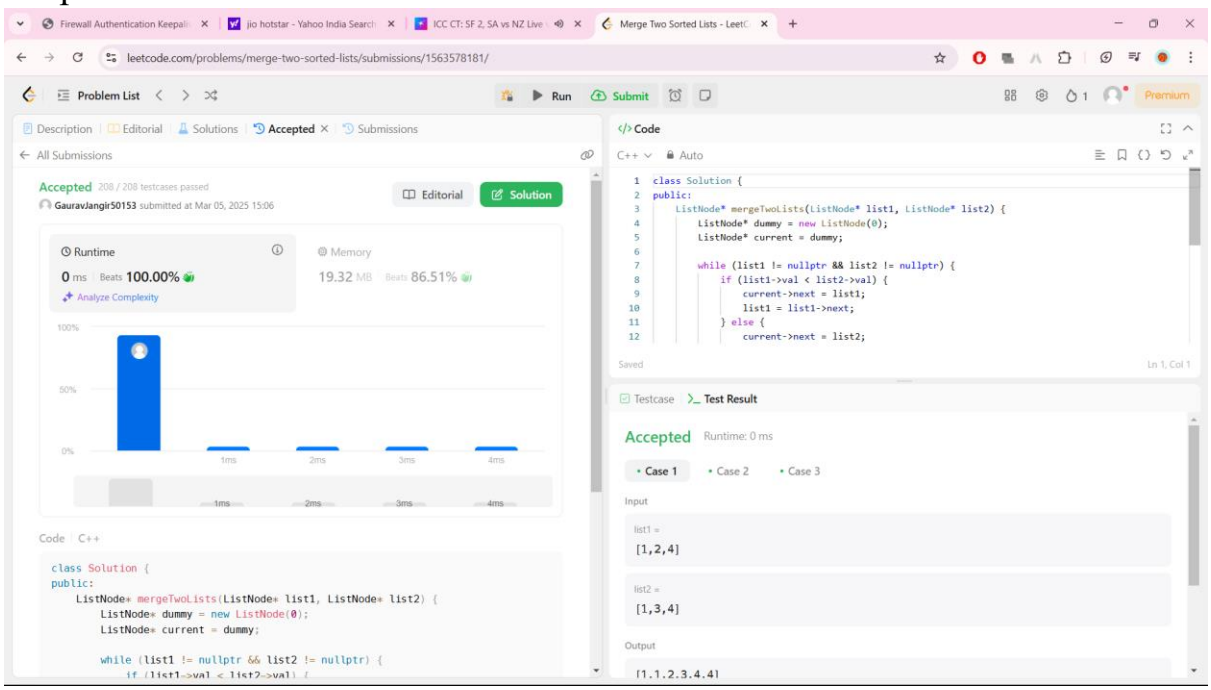


5. Merge two sorted lists:

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode* dummy = new ListNode(0);
        ListNode* current = dummy;

        while (list1 != nullptr && list2 != nullptr) {
            if (list1->val < list2->val) {
                current->next = list1;
                list1 = list1->next;
            } else {
                current->next = list2;
                list2 = list2->next;
            }
            current = current->next;
```

```
        }

        // Attach remaining nodes
        current->next = list1 != nullptr ? list1 : list2;

        return dummy->next;  // Return the head of the merged list
    }
};
```

Output:

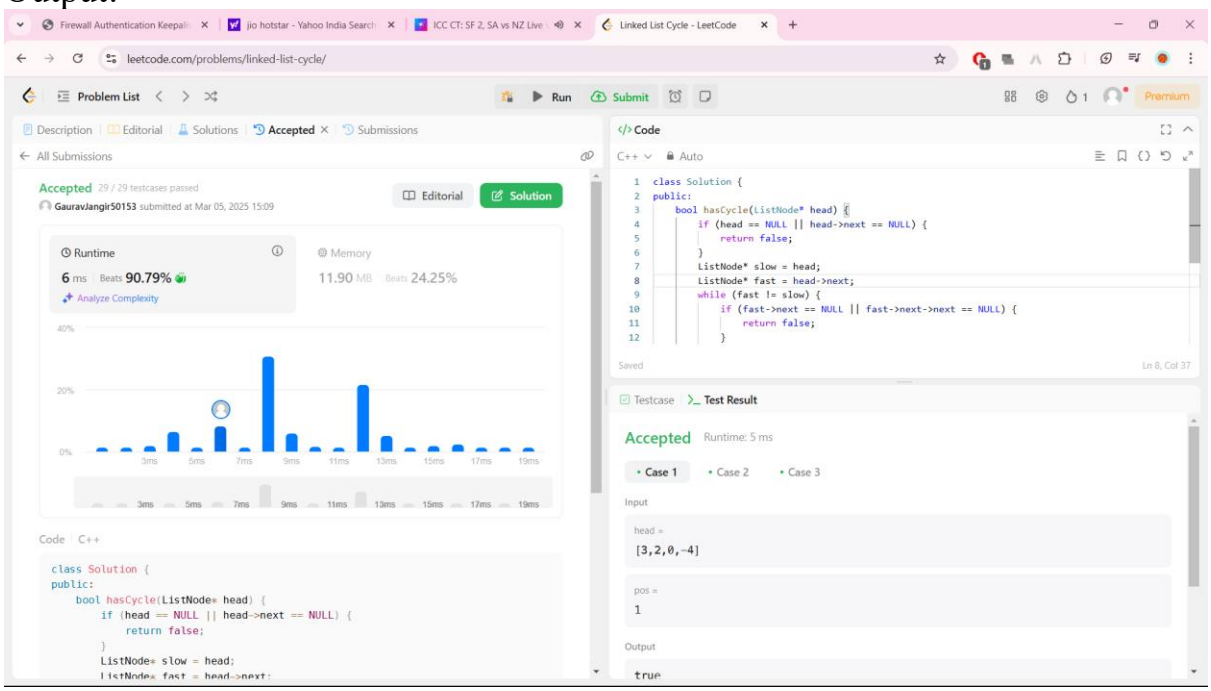


6. Detect a cycle in a linked list:

```
class Solution {
public:
    bool hasCycle(ListNode* head) {
        if (head == NULL || head->next == NULL) {
            return false;
        }
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast != slow) {
            if (fast->next == NULL || fast->next->next == NULL) {
                return false;
            }
            slow = slow->next;
```

```
            fast = fast->next->next;
        }
        return true;
    }
};
```

Output:

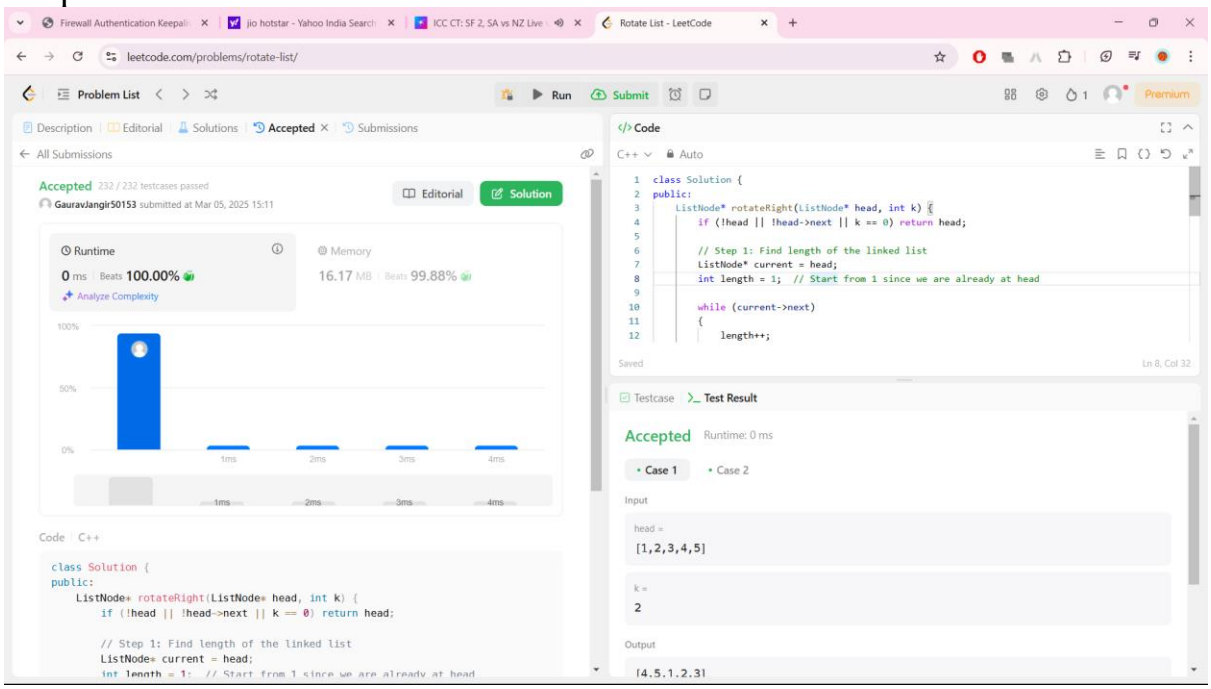


7. Rotate a list:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head;

        // Step 1: Find length of the linked list
        ListNode* current = head;
        int length = 1;  // Start from 1 since we are already at head

        while (current->next)
        {
            length++;
            current = current->next;
        }

        // Step 2: Optimize k
```

```
        k %= length;
        if (k == 0) return head;  // No rotation needed

        // Step 3: Connect last node to head to make it circular
        current->next = head;

        // Step 4: Find the new tail (length - k - 1 moves from start)
        int newTailPos = length - k;
        current = head;

        for (int i = 1; i < newTailPos; i++)
        {
            current = current->next;
        }

        // Step 5: Update head and break the circular link
        head = current->next;  // New head
        current->next = nullptr;  // Break the circular link

        return head;
    }
};
```

Output:

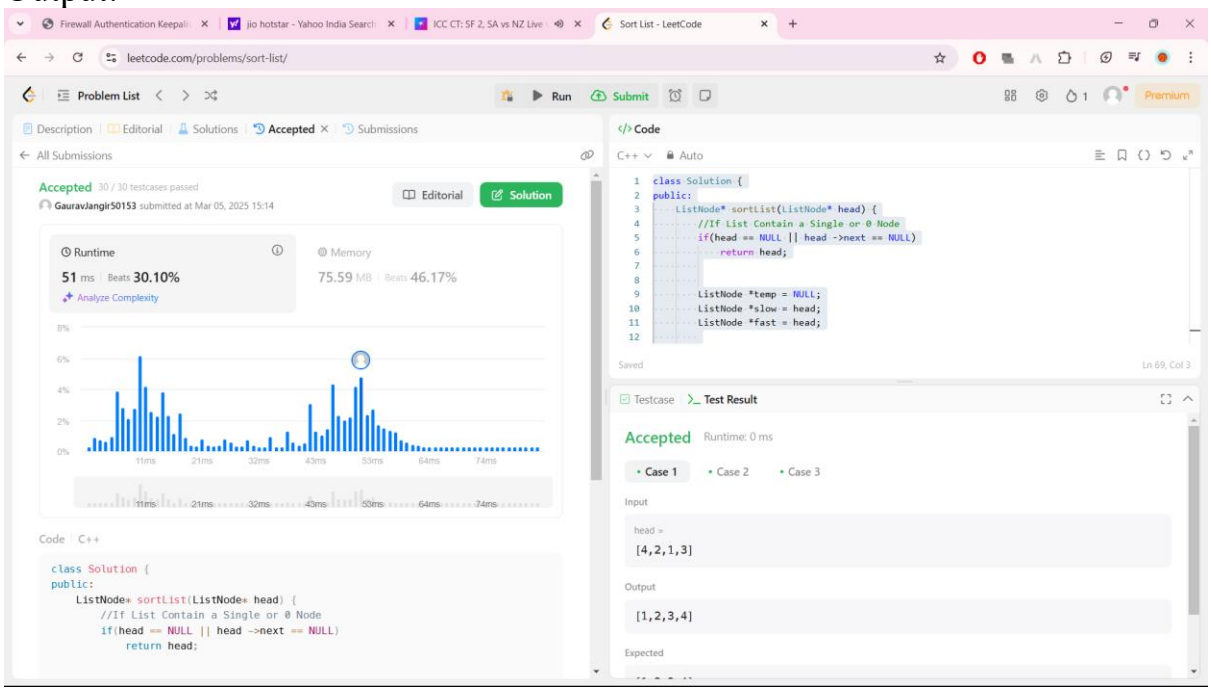


8. Sort List:

```
class Solution {
public:
   ListNode* sortList(ListNode* head) {
      //If List Contain a Single or 0 Node
      if(head == NULL || head ->next == NULL)
         return head;
      ListNode *temp = NULL;
      ListNode *slow = head;
      ListNode *fast = head;
        // 2 pointer appraoach / turtle-hare Algorithm (Finding the middle
element)
      while(fast !=  NULL && fast -> next != NULL)
      {
         temp = slow;
         slow = slow->next;          //slow increment by 1
         fast = fast ->next ->next;  //fast incremented by 2
      }
      temp -> next = NULL;           //end of first left half

      ListNode* l1 = sortList(head);    //left half recursive call
      ListNode* l2 = sortList(slow);    //right half recursive call
      return mergelist(l1, l2);         //mergelist Function call
   }
   //MergeSort Function O(n*logn)
   ListNode* mergelist(ListNode *l1, ListNode *l2)
   {
      ListNode *ptr = new ListNode(0);
      ListNode *curr = ptr;
      while(l1 != NULL && l2 != NULL)
      {
         if(l1->val <= l2->val)
         {
            curr -> next = l1;
            l1 = l1 -> next;
         }
         else
         {
```

```
            curr -> next = l2;
            l2 = l2 -> next;
        }
    curr = curr ->next;
    }
    //for unqual length linked list
    if(l1 != NULL)
    {
        curr -> next = l1;
        l1 = l1->next;
    }
    if(l2 != NULL)
    {
        curr -> next = l2;
        l2 = l2 ->next;
    }
    return ptr->next;
  }
};
```

Output:

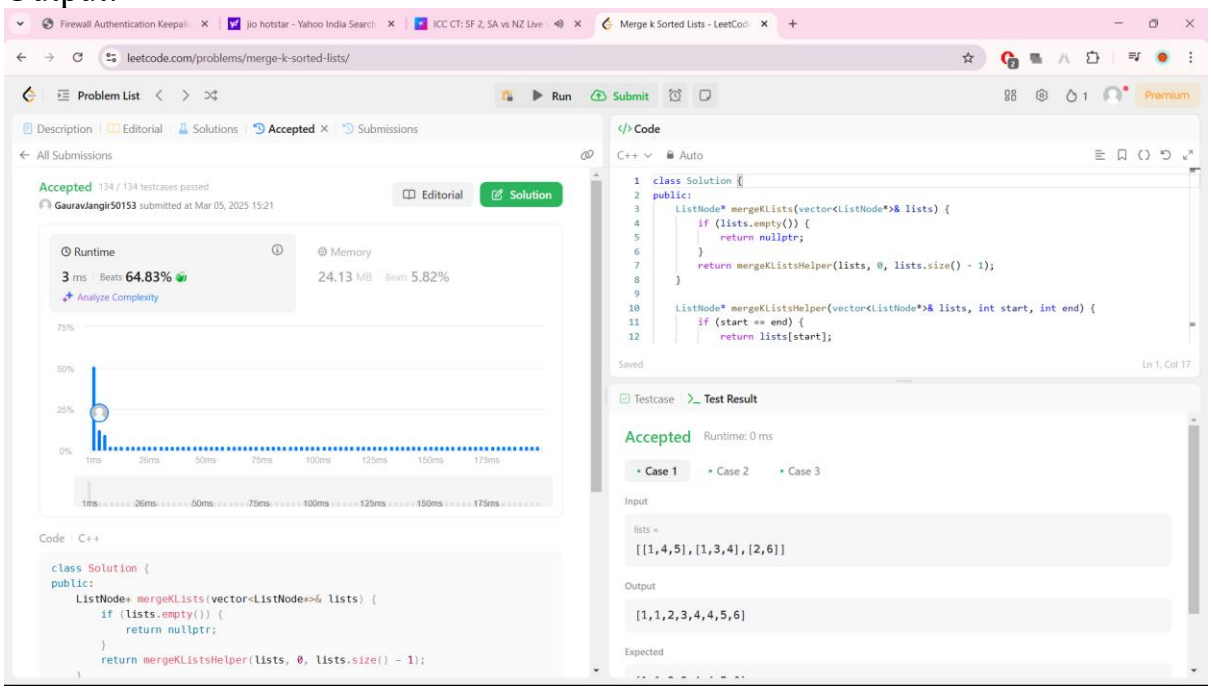


9. Merge k sorted lists:

```
class Solution {
public:
```

```
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) {
            return nullptr;
        }
        return mergeKListsHelper(lists, 0, lists.size() - 1);
    }

    ListNode* mergeKListsHelper(vector<ListNode*>& lists, int start, int end)
{
        if (start == end) {
            return lists[start];
        }
        if (start + 1 == end) {
            return merge(lists[start], lists[end]);
        }
        int mid = start + (end - start) / 2;
        ListNode* left = mergeKListsHelper(lists, start, mid);
        ListNode* right = mergeKListsHelper(lists, mid + 1, end);
        return merge(left, right);
    }

    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val < l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        curr->next = l1 ? l1 : l2;

        return dummy->next;
    }
};
```

Output:


leetcode.com/problems/merge-k-sorted-lists/
Problem List
Run
Submit
Premium
Description
Editorial
Solutions
Accepted
Submissions
All Submissions
Accepted 134 / 134 testcases passed
GauravJangir50153 submitted at Mar 05, 2025 15:21
Editorial
Solution
Runtime
3 ms | Beats 64.83%
Analyze Complexity
Memory
24.13 MB | Beats 5.82%
Code | C++
class Solution {
public:
ListNode* mergeKLists(vector<ListNode*>& lists) {
if (lists.empty()) {
return nullptr;
}
return mergeKListsHelper(lists, 0, lists.size() - 1);
Code
C++
Auto
class Solution {
public:
ListNode* mergeKLists(vector<ListNode*>& lists) {
if (lists.empty()) {
return nullptr;
}
return mergeKListsHelper(lists, 0, lists.size() - 1);
}
ListNode* mergeKListsHelper(vector<ListNode*>& lists, int start, int end) {
if (start == end) {
return lists[start];
Saved
Ln 1, Col 17
Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
lists =
[[1,4,5],[1,3,4],[2,6]]
Output
[1,1,2,3,4,4,5,6]
Expected