ASSIGNMENT -3

Student Name: Tanisha Mahajan **UID: 22BCS11132**

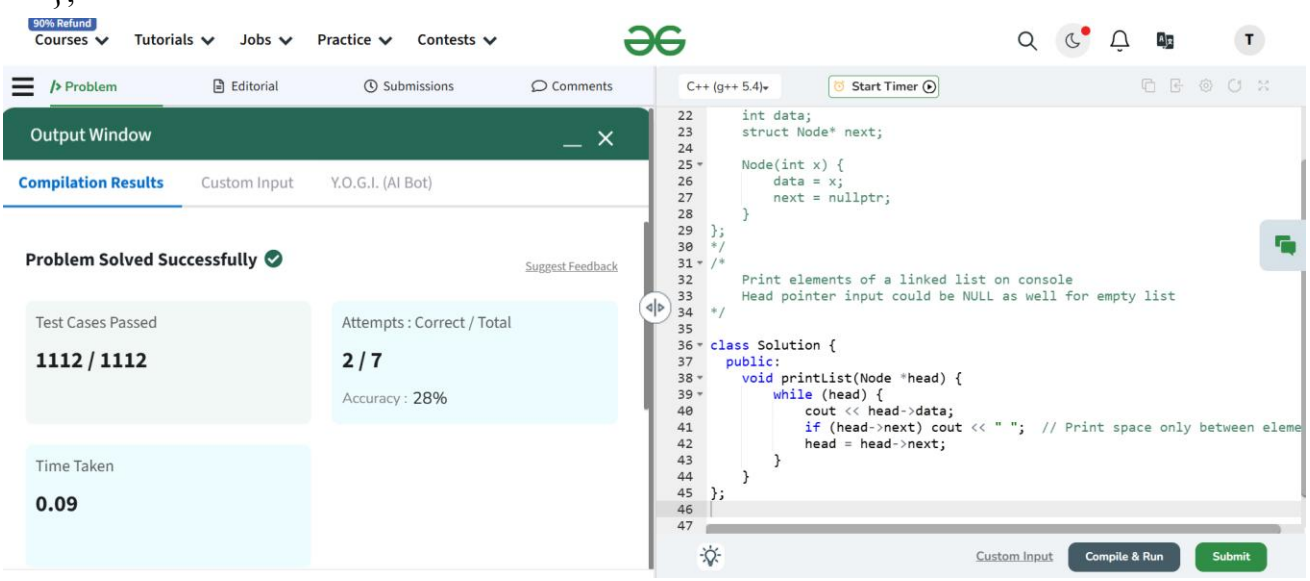
Branch: BE-CSE **Section/Group: 22BCS_TPP_607-B**

Semester: 6TH **Subject Name : AP-11**

❍ Linked List :

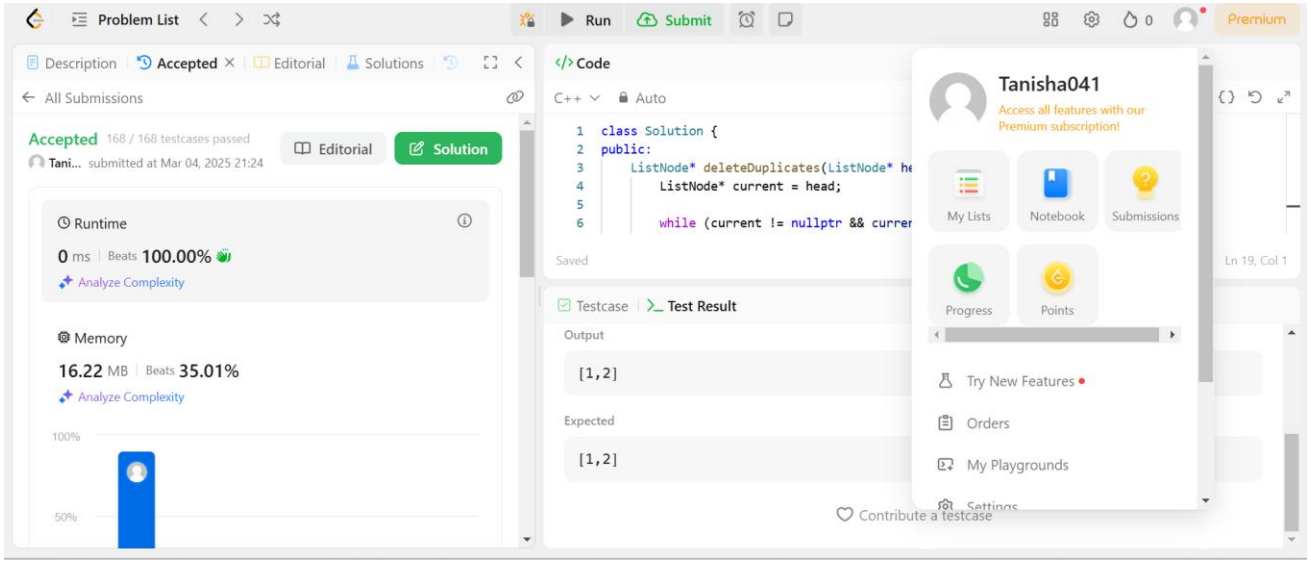
1. Print linked list :

```
class Solution {
  public:
      void printList(Node *head) {
      Node* temp = head;
      while (temp != nullptr) {
         cout << temp->data << " ";
         temp = temp->next;
      }
      cout << endl; }
  };
```



2. Remove duplicates from the sorted list :

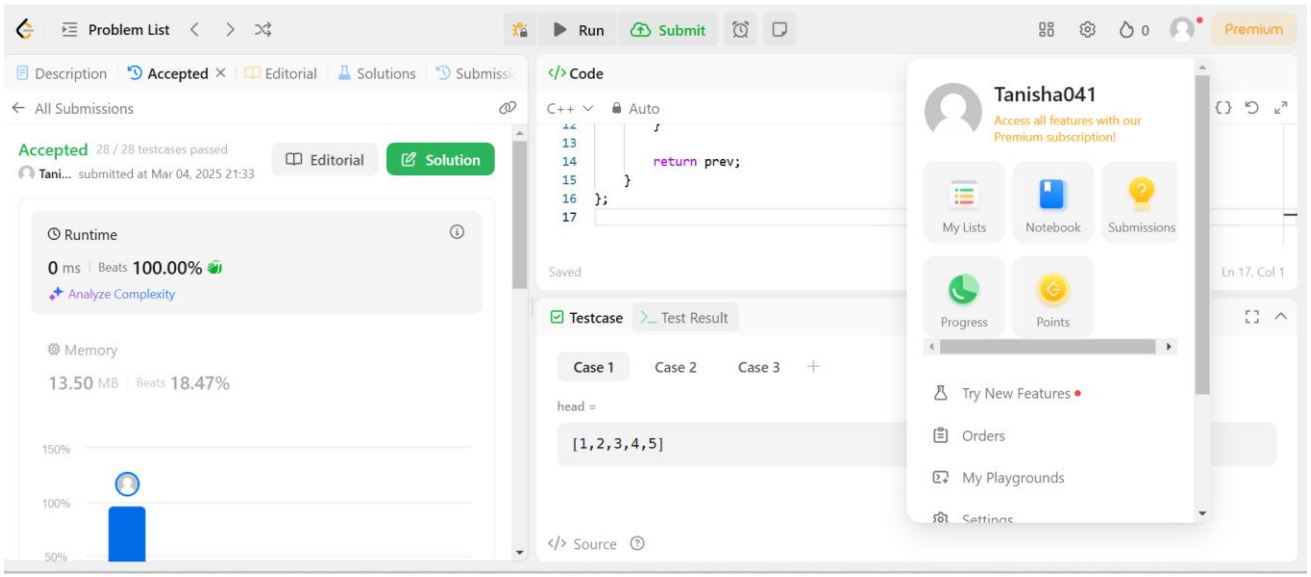
```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* current = head;

        while (current != nullptr && current->next != nullptr) {
            if (current->val == current->next->val) {
                ListNode* temp = current->next;
                current->next = current->next->next;
                delete temp;
            } else {
                current = current->next;            }
        }

        return head;
    }
};
```



3. Reverse a Linked list :

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
```

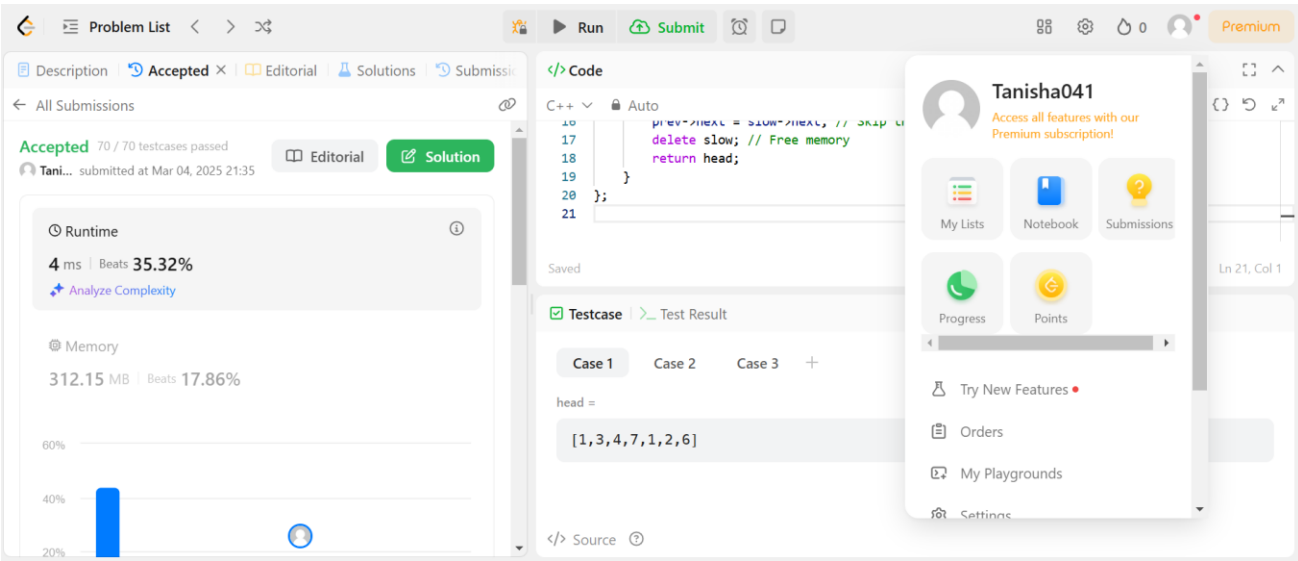
```
        ListNode* prev = nullptr;
        ListNode* curr = head;
        while (curr != nullptr) {
            ListNode* nextNode = curr->next;
            curr->next = prev;
            prev = curr;
            curr = nextNode;
        }
        return prev;
    }
};
```



4. Delete middle node of linked list :

```
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;
```

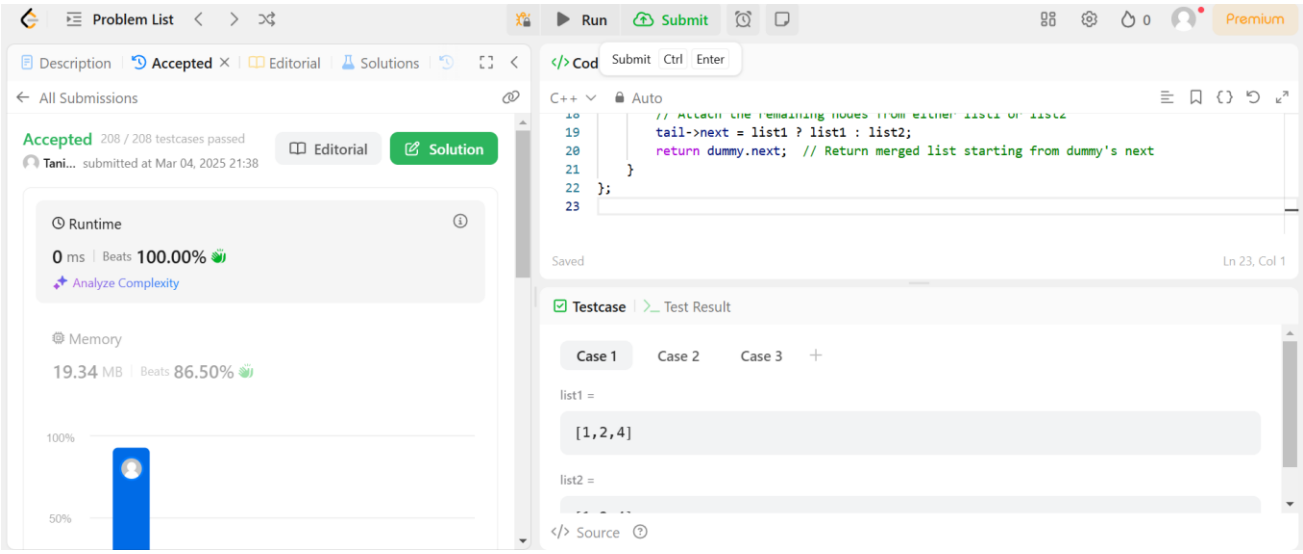
```
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = slow->next;
        delete slow;
        return head;
    }
};
```



5. Merge two sorted linked lists :

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode dummy(0);
        ListNode* tail = &dummy;
        while (list1 && list2) {
            if (list1->val < list2->val) {
                tail->next = list1;
                list1 = list1->next;
            }
            else {
                tail->next = list2;
```

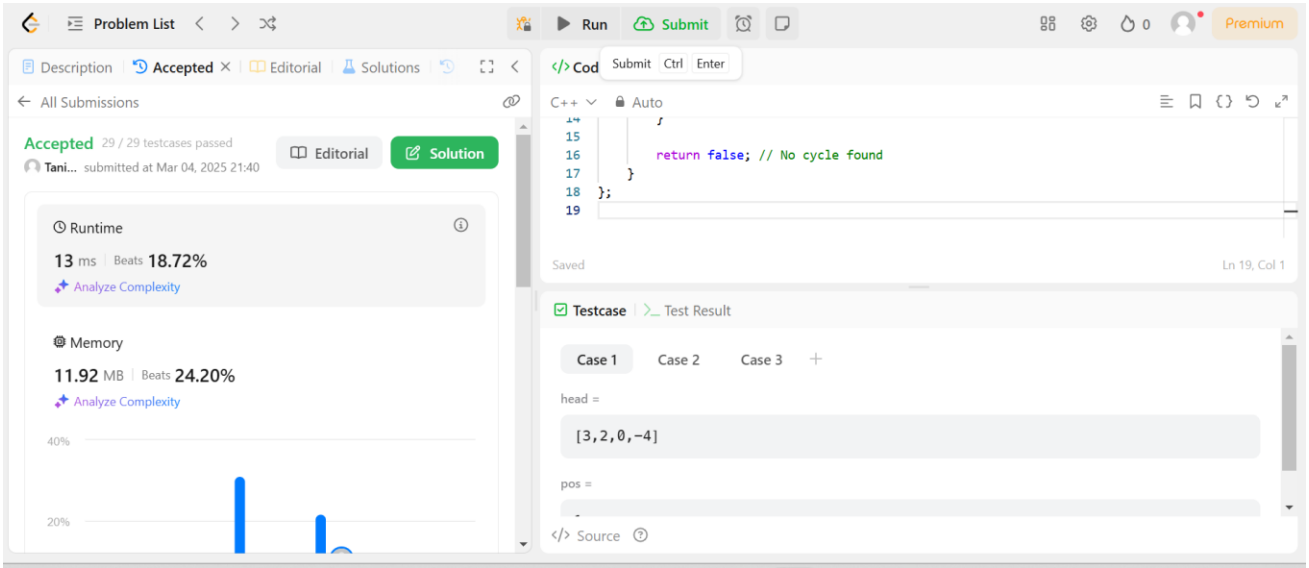
```
            list2 = list2->next;
        }
        tail = tail->next;
    }
    tail->next = list1 ? list1 : list2;
    return dummy.next;
  }
};
```



6. Detect a cycle in a linked list::

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

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

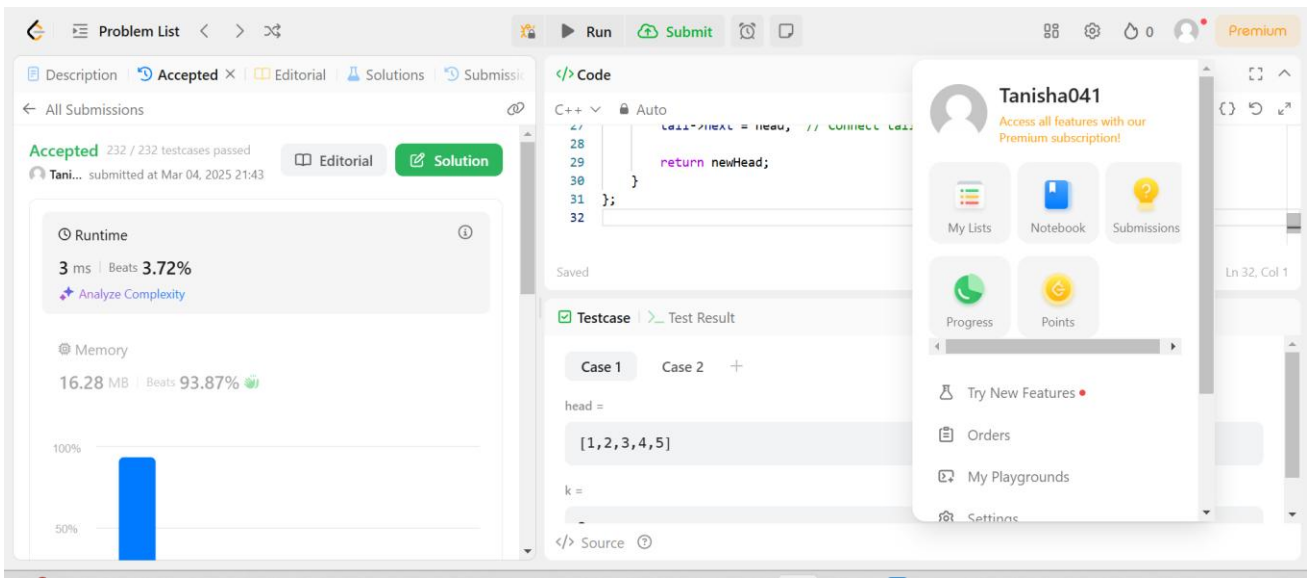
```
            if (slow == fast) return true;
        }
        return false;
    }
};
```



7. Rotate a list:

```
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;
```

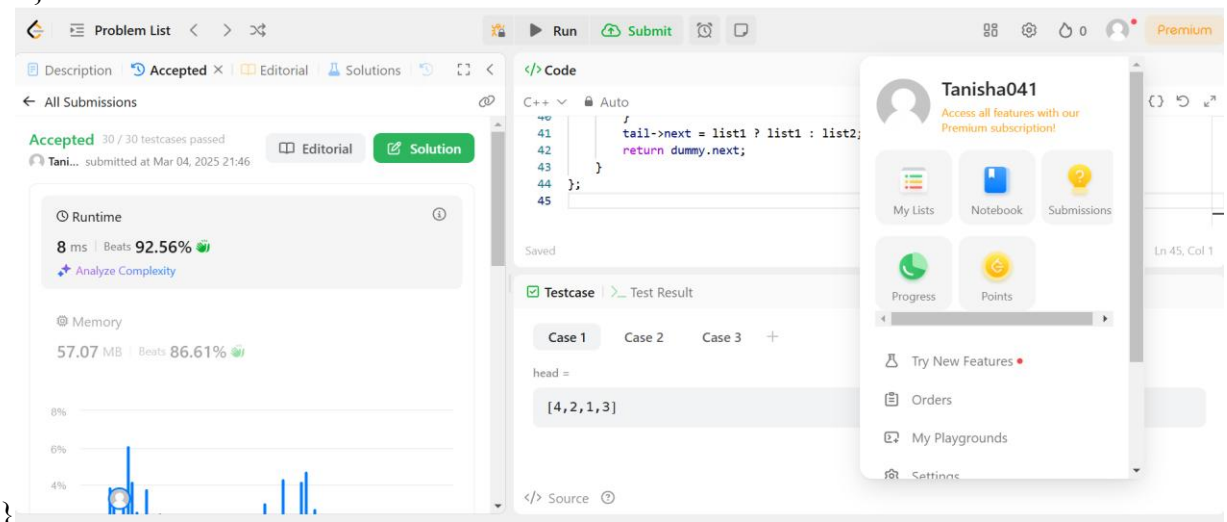
```
        for (int i = 0; i < length - k - 1; i++) {
            newTail = newTail->next;
        }
        ListNode* newHead = newTail->next;
        newTail->next = nullptr;
        tail->next = head;
        return newHead;
    }
};
```



8. Sort a list:

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* prev = nullptr;
        while (fast && fast->next) {
            prev = slow;
```

```
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = nullptr;
        ListNode* left = sortList(head);
        ListNode* right = sortList(slow);
        return merge(left, right);
    }
private:
    ListNode* merge(ListNode* list1, ListNode* list2) {
        ListNode dummy(0);
        ListNode* tail = &dummy;
        while (list1 && list2) {
            if (list1->val < list2->val) {
                tail->next = list1;
                list1 = list1->next;
            } else {
                tail->next = list2;
                list2 = list2->next;
            }
            tail = tail->next;
        }
        tail->next = list1 ? list1 : list2;
        return dummy.next;
    }
```



}

9. Merge k sorted lists :

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        auto compare = [](ListNode* a, ListNode* b) { return a->val > b->val; };
        priority_queue<ListNode*, vector<ListNode*>, decltype(compare)> minHeap(compare);

        for (ListNode* list : lists) {
            if (list) minHeap.push(list);
        }
        ListNode dummy(0);
        ListNode* tail = &dummy;
        while (!minHeap.empty()) {
            ListNode* node = minHeap.top();
            minHeap.pop();
            tail->next = node;
            tail = tail->next;
            if (node->next) minHeap.push(node->next);
        }
        return dummy.next;
    }
};
```