

AP ASSIGNMENT 3

Name – Nikhil Kumar UID – 22BCS12206 Section – 606-B Date – 06/03/2025

1. Print Linked List:

```
class Solution {  
public:  
    // Function to display the elements of a linked list in same line  
    void printList(Node *head) {  
        while(head!=nullptr){  
            cout<<head->data<<" ";  
            head=head->next;  
        }  
    }  
};
```

The screenshot shows a dark-themed 'Output Window' with a title bar containing a close button. Below the title bar are three tabs: 'Compilation Results' (active), 'Custom Input', and 'Y.O.G.I. (AI Bot)'. The main content area displays a green checkmark and the text 'Problem Solved Successfully'. To the right of this text is a link labeled 'Suggest Feedback'. Below this, there are four performance metrics arranged in a 2x2 grid:

| | |
|---|--|
| Test Cases Passed 1112 / 1112 | Attempts : Correct / Total 1 / 1 Accuracy : 100% |
| Points Scored ⓘ 1 / 1 Your Total Score: 28 ↑ | Time Taken 0.06 |

At the bottom left of the window is a button labeled 'Solve Next'.

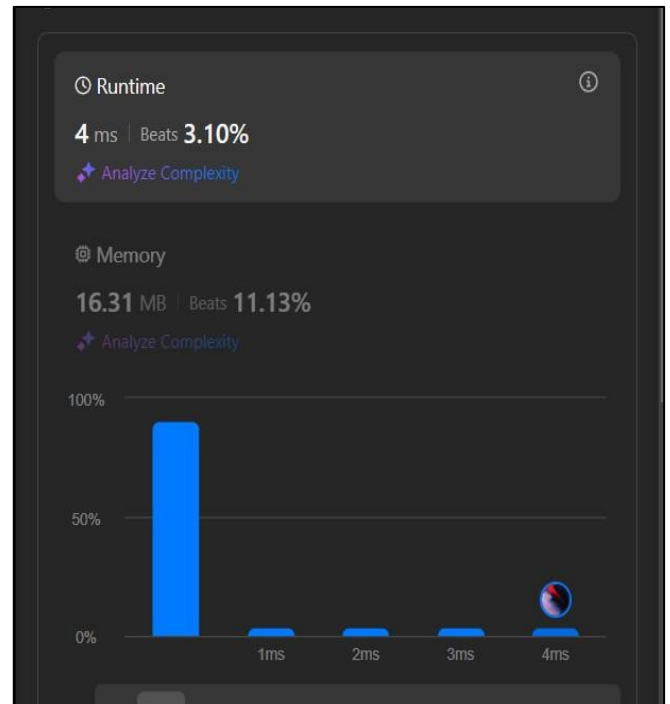
2. Remove duplicates from a sorted list:

```
class Solution {  
public:  
    ListNode* deleteDuplicates(ListNode* head) {  
        if(!head || !head->next){  
            return head;  
        }  
        ListNode* current = head;
```

```

while(current && current->next){
    if(current->val == current->next->val){
        ListNode* duplicate=current->next;
        current->next=current->next->next;
        delete duplicate;
    }
    else{
        current=current->next;
    }
}
return head;
}
};

```

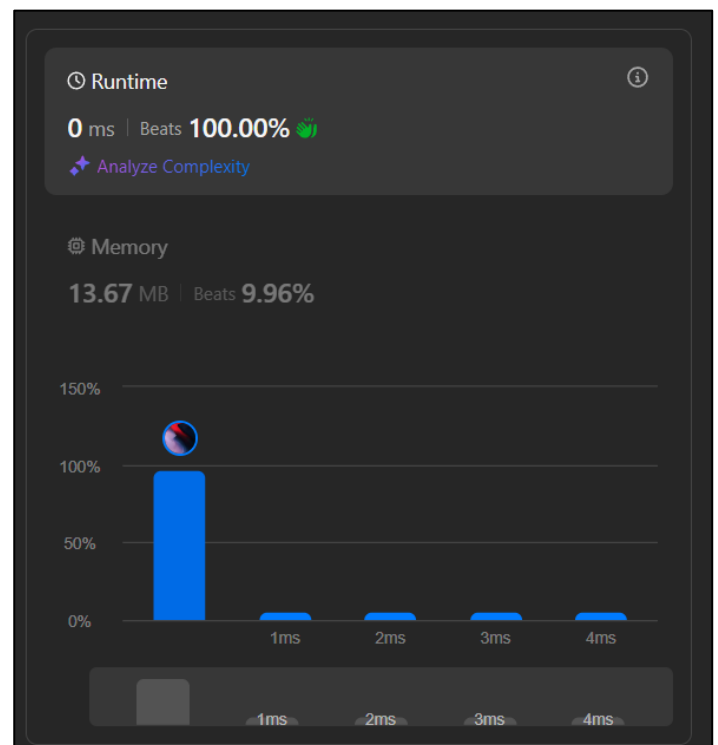


3. Reverse a linked list:

```

class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(!head || !head->next){
            return head;
        }
        ListNode* newn=reverseList(head-
>next);
        head->next->next=head;
        head->next=nullptr;
        return newn;
    }
};

```



4. Delete middle node of a list:

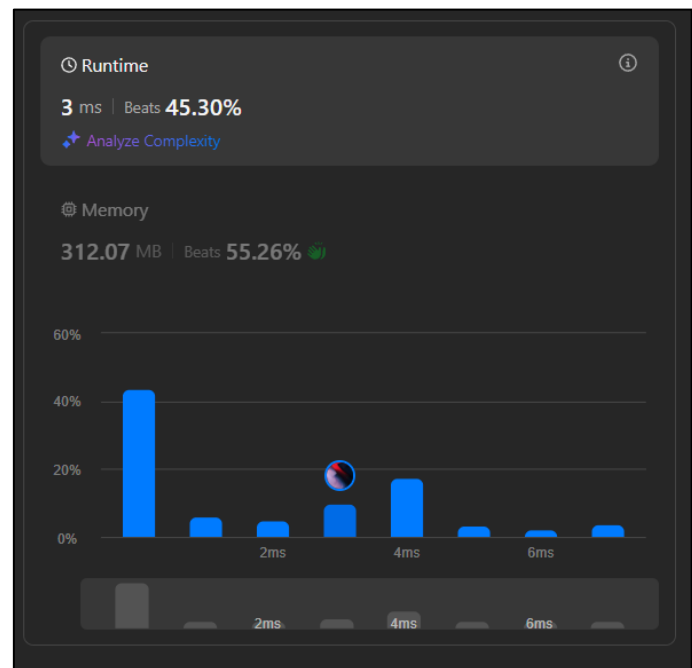
```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if(!head || !head->next){
            return nullptr;
        }

        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;

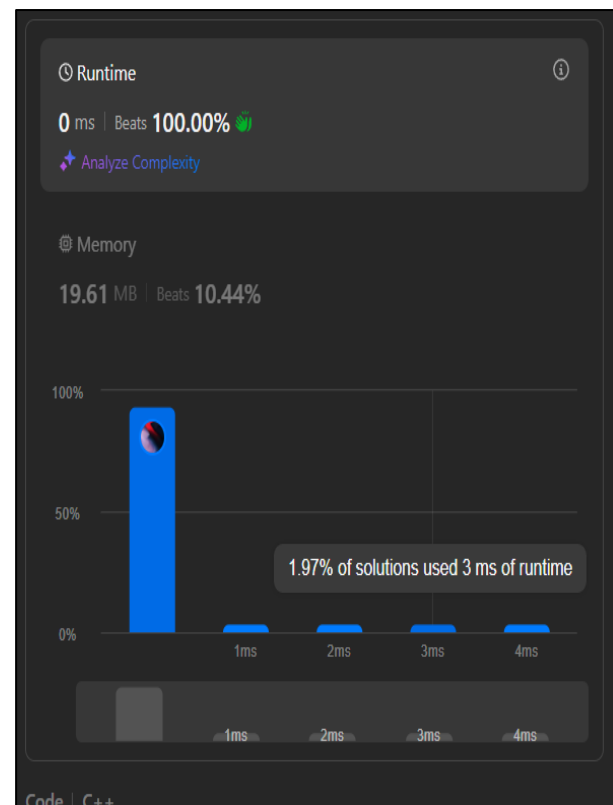
        return head;
    }
};
```



5. Merge two sorted linked lists:

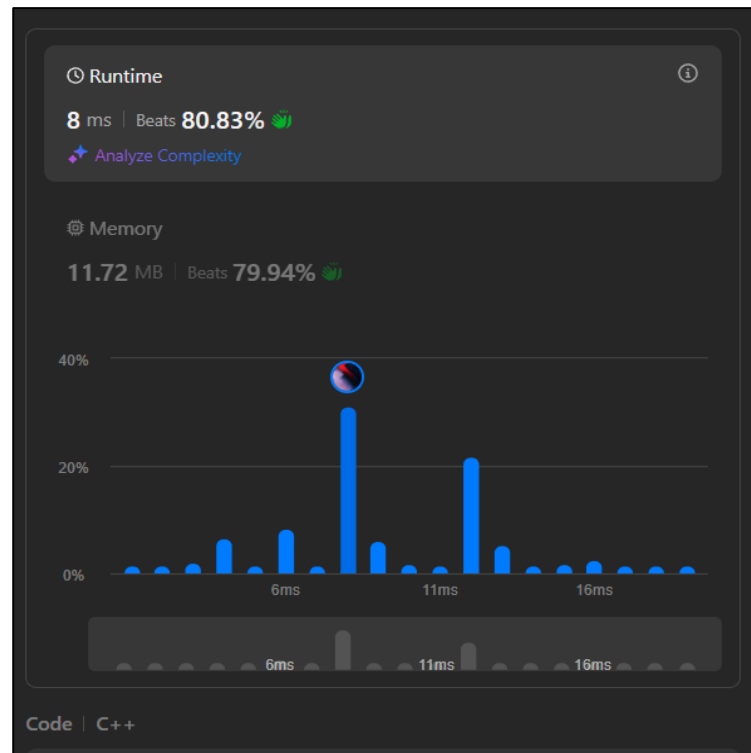
```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1,
        ListNode* list2) {
        if (!list1) return list2;
        if (!list2) return list1;

        if (list1->val <= list2->val) {
            list1->next = mergeTwoLists(list1->next, list2);
            return list1;
        } else {
            list2->next = mergeTwoLists(list1, list2->next);
            return list2;
        }
    }
};
```



6. Detect a cycle in a linked list:

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



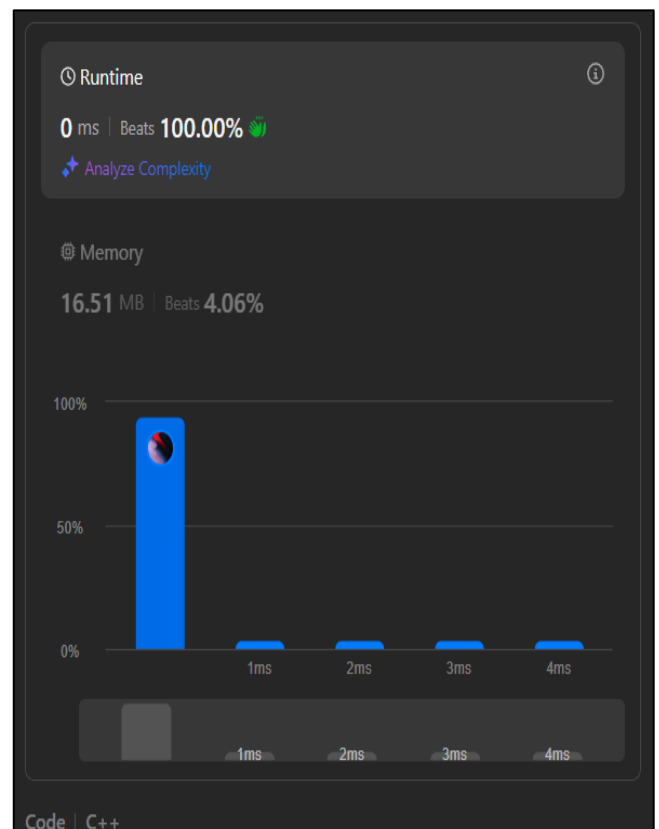
7. Rotate a list:

```
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 List:

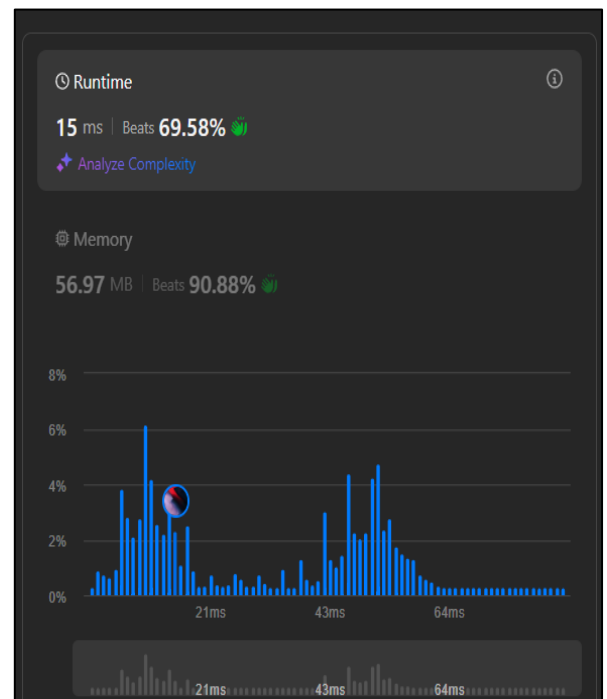
```

class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode* mid = getMiddle(head);
        ListNode* left = head;
        ListNode* right = mid->next;
        mid->next = nullptr;
        left = sortList(left);
        right = sortList(right);
        return merge(left, right);
    }

private:
    ListNode* getMiddle(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }

    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode dummy(0);
        ListNode* tail = &dummy;
        while (l1 && l2) {
            if (l1->val <= l2->val) {
                tail->next = l1;
                l1 = l1->next;
            } else {
                tail->next = l2;
                l2 = l2->next;
            }
        }
    }
}

```



```

        tail = tail->next;
    }
    tail->next = l1 ? l1 : l2;
    return dummy.next;
}
};

```

9. Merge K Sorted List:

```

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> pq;

        for (auto list : lists) {
            if (list) pq.push(list);
        }

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

        while (!pq.empty()) {
            ListNode* node = pq.top();
            pq.pop();
            tail->next = node;
            tail = tail->next;

            if (node->next) pq.push(node->next);
        }

        return dummy.next;
    }
};

```

