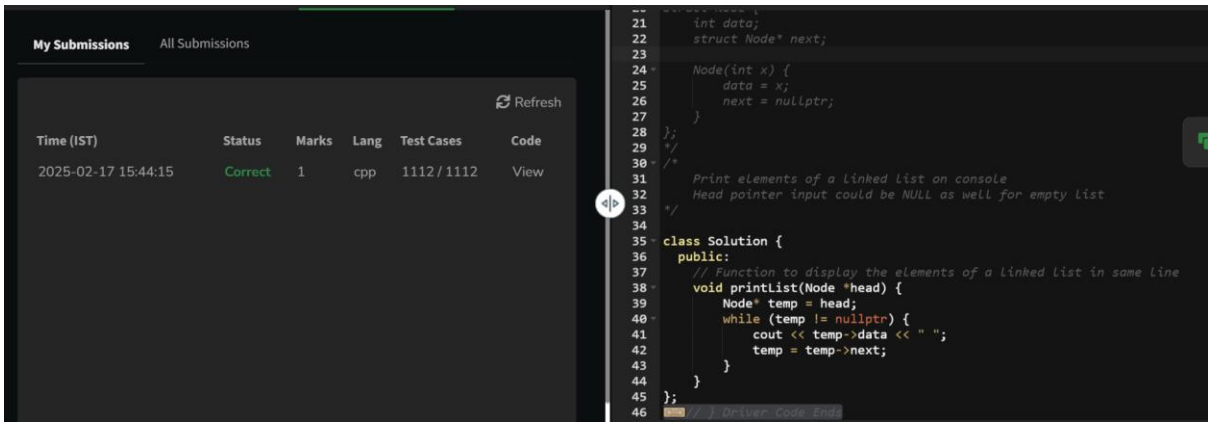
Assignment 3

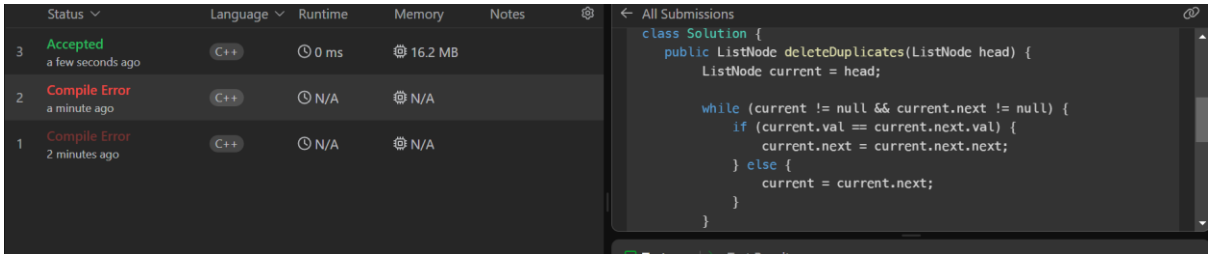
1. Code-

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



2. Code->

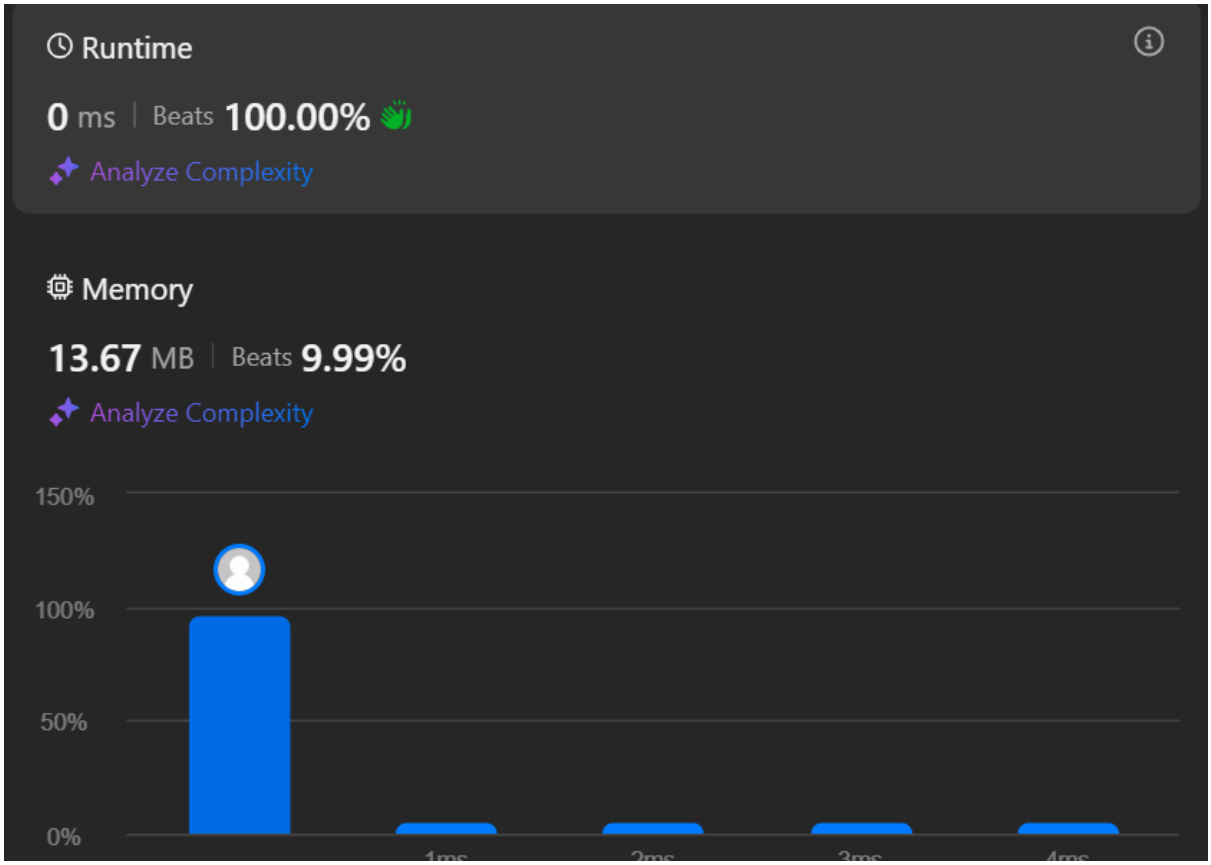
```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* res = head;

        while (head && head->next) {
            if (head->val == head->next->val) {
                head->next = head->next->next;
            } else {
                head = head->next;
            }
        }

        return res;
    }
};
```



3.

Code->

```
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;
    }
};
```

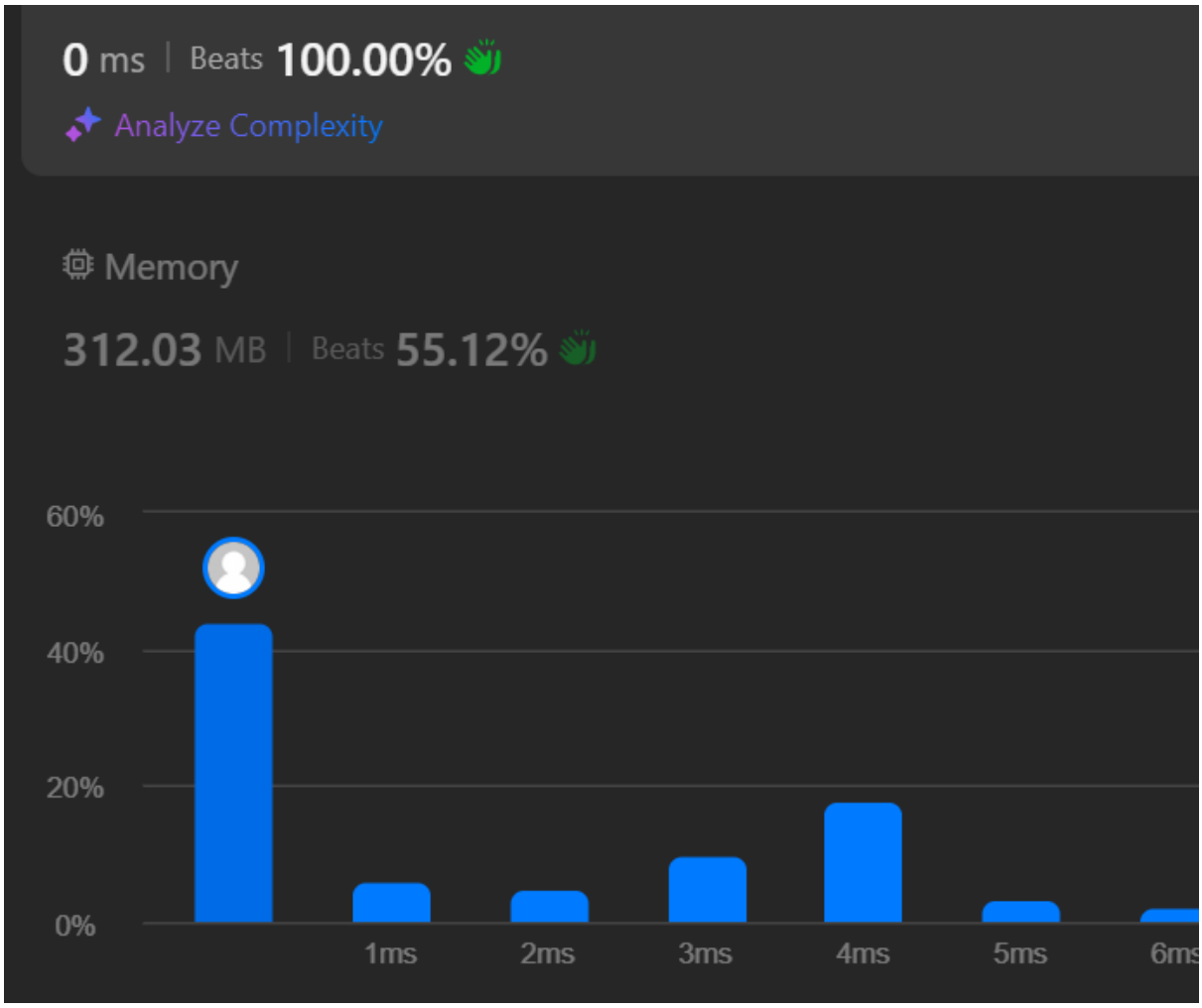


4.

Code-

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

```
        slow->val = slow->next->val;
        slow->next = slow->next->next;
        return head;
    }
};
```

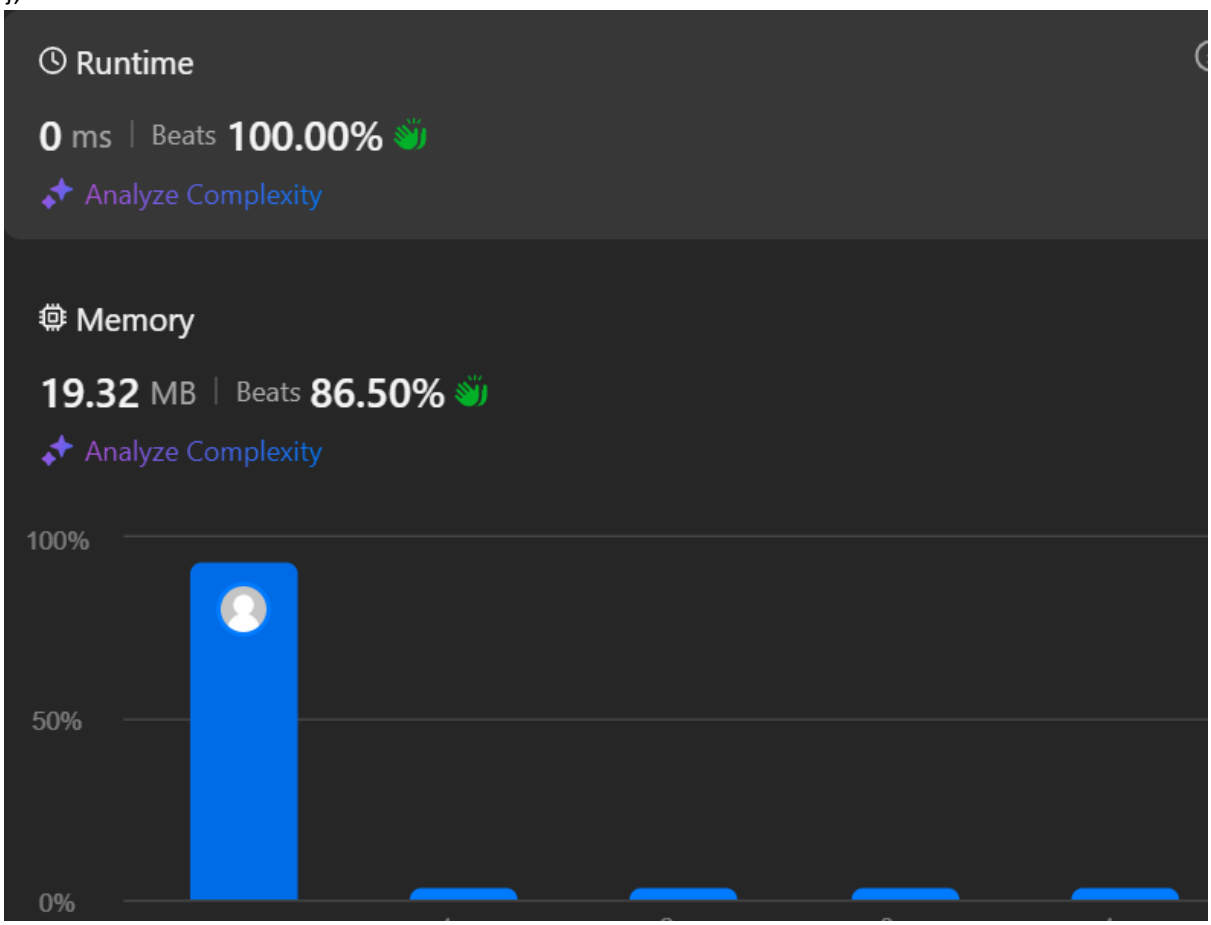


5.

Code->

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if(list1 == NULL || list2 == NULL){
            return list1 == NULL ? list2 : list1;
```

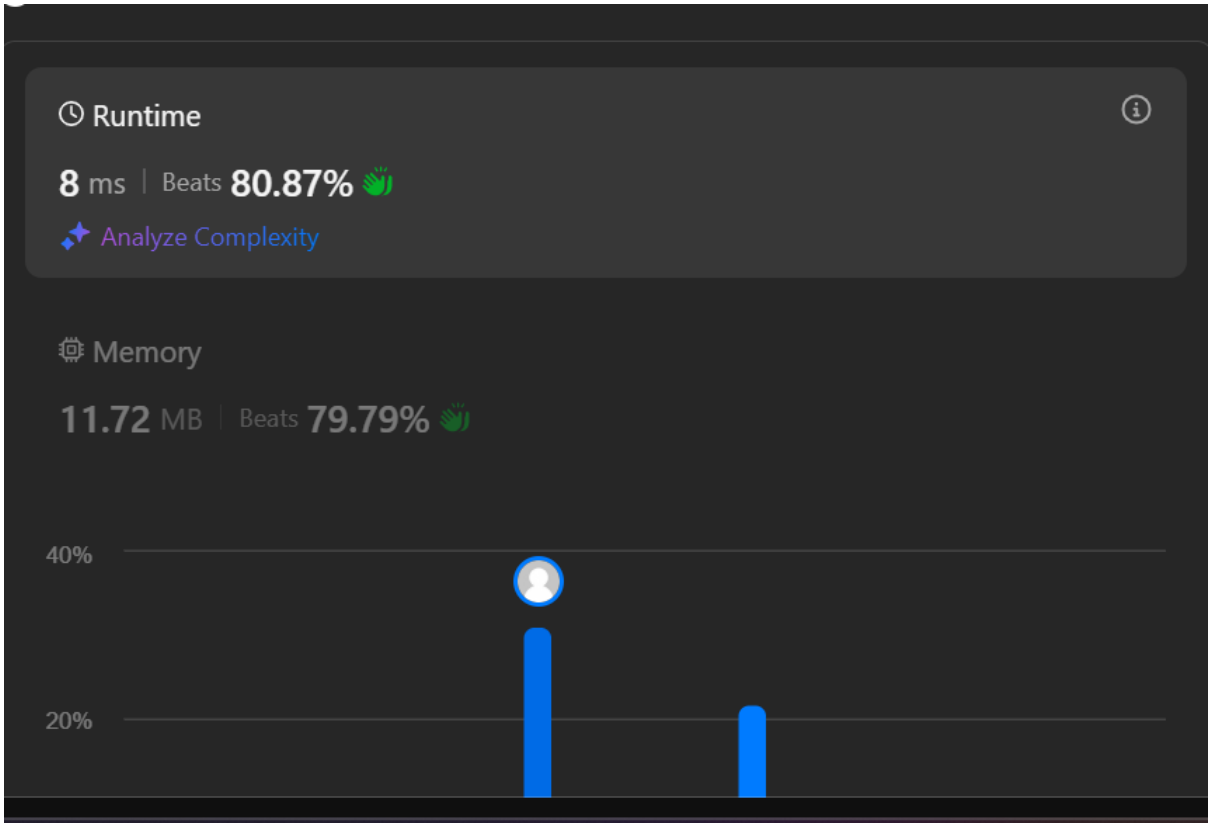
```
        }

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



6.

```
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;
    }
};
```

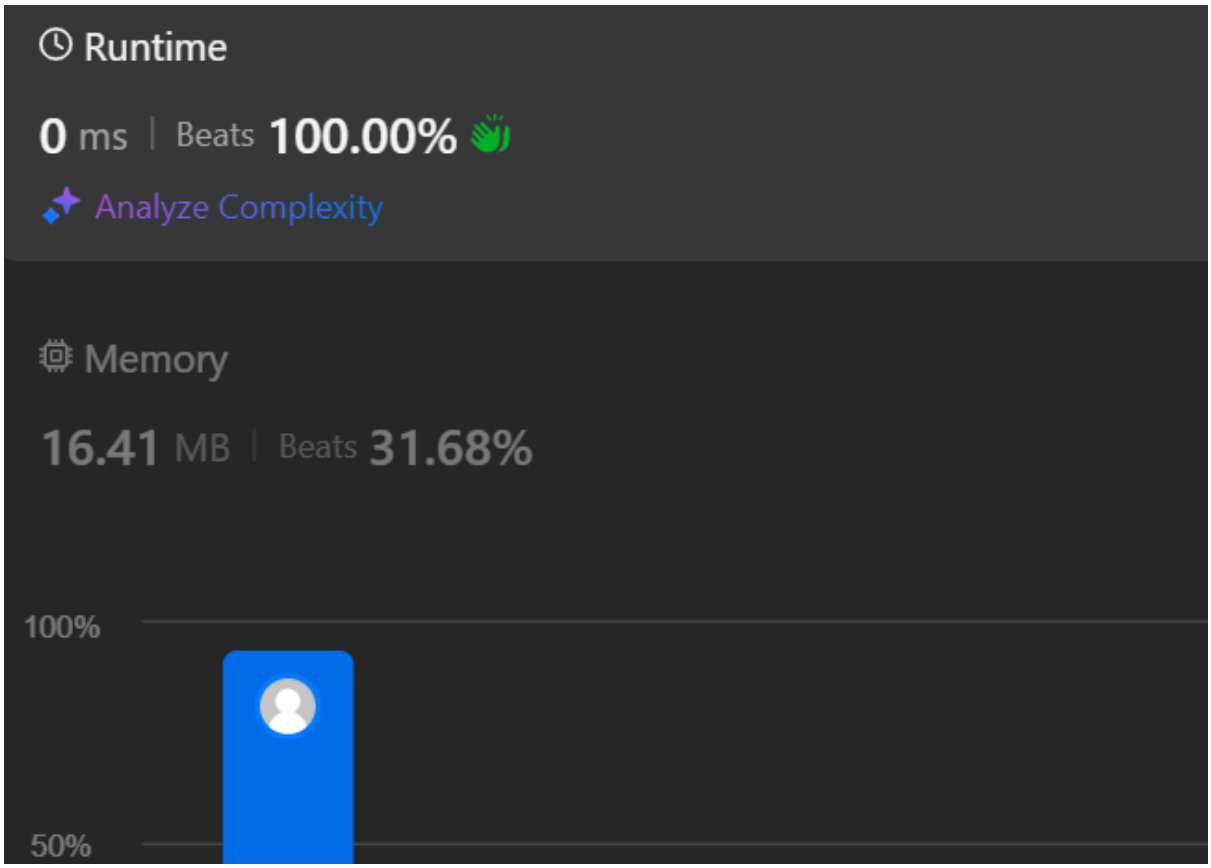


7.

```
/**
```

```
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
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;
    }
};
```

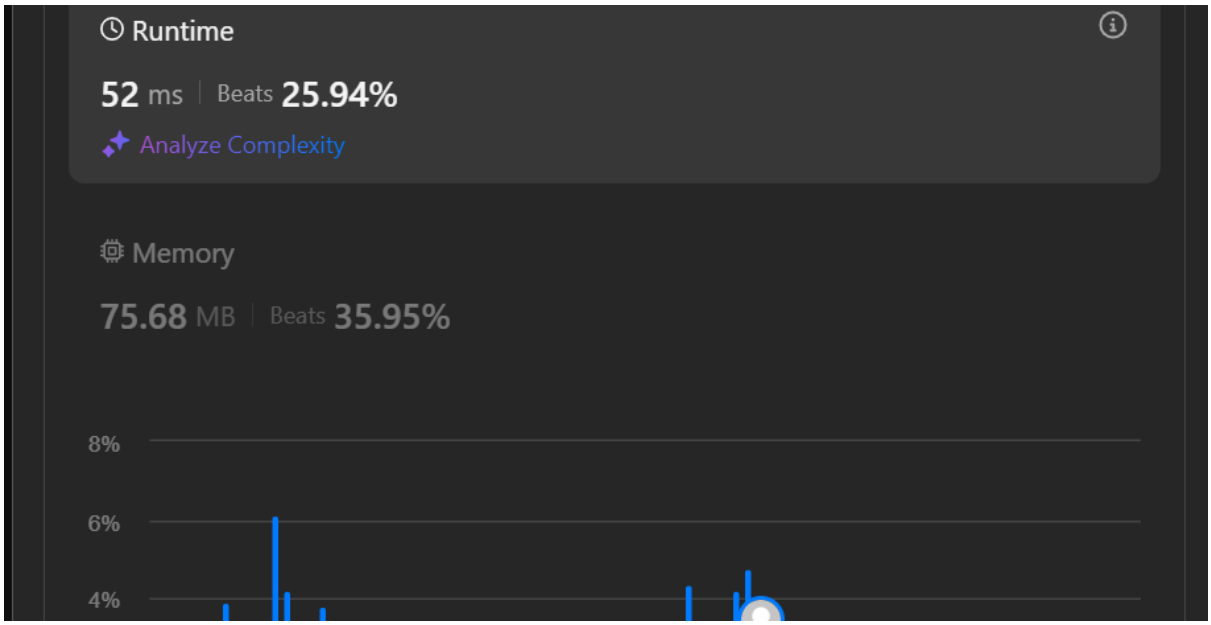


8.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* getmid(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head->next;

        while (fast != NULL && fast->next != NULL) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }

    ListNode* merge(ListNode* left, ListNode* right) {
        if (left == NULL)
            return right;
        if (right == NULL)
            return left;

        ListNode* dummy = new ListNode(0);
```

```
    ListNode* temp = dummy;

    while (left != NULL && right != NULL) {
        if (left->val < right->val) {
            temp->next = left;
            temp = left;
            left = left->next;
        } else {
            temp->next = right;
            temp = right;
            right = right->next;
        }
    }
    while (left != NULL) {
        temp->next = left;
        temp = left;
        left = left->next;
    }
    while (right != NULL) {
        temp->next = right;
        temp = right;
        right = right->next;
    }
    dummy = dummy->next;
    return dummy;
}

ListNode* sortList(ListNode* head) {
    // using merge sort

    // base case
```

```
        if (head == NULL || head->next == NULL)
            return head;

        ListNode* mid = getmid(head);

        ListNode* left = head;
        ListNode* right = mid->next;
        mid->next = NULL;

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

        ListNode* result = merge(left, right);

        return result;
    }
};
```

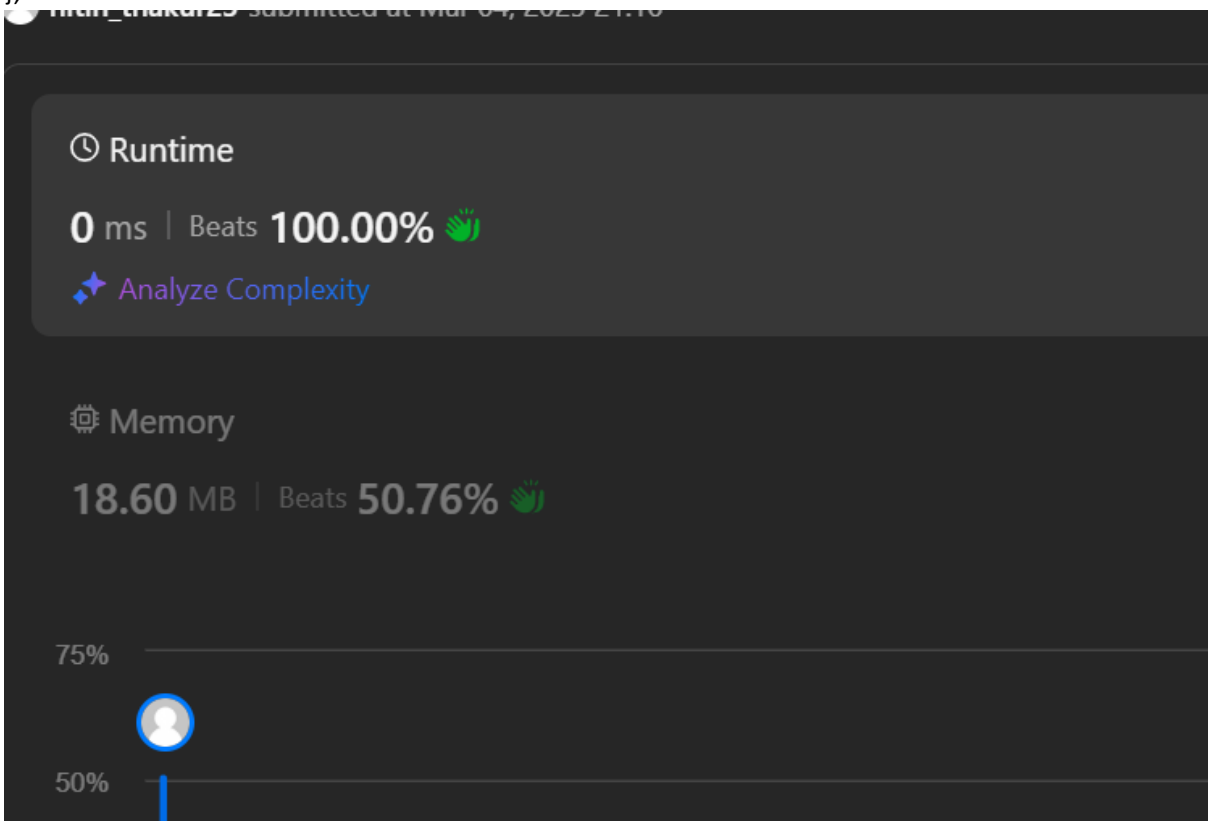


9.

```
#include <vector>
using namespace std;
class Solution {
```

```
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (!l1) return l2;
        if (!l2) return l1;

        if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }

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

    ListNode* divideAndConquer(vector<ListNode*>& lists, int left, int right) {
        if (left == right) return lists[left];

        int mid = left + (right - left) / 2;
        ListNode* l1 = divideAndConquer(lists, left, mid);
        ListNode* l2 = divideAndConquer(lists, mid + 1, right);
        return mergeTwoLists(l1, l2);
    }
```

};


Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
18.60 MB | Beats 50.76%
75%
50%