Assignment 2

Name: Diksha Kumari UID: 22BCS16005

Section: 606 – B Subject: AP LAB

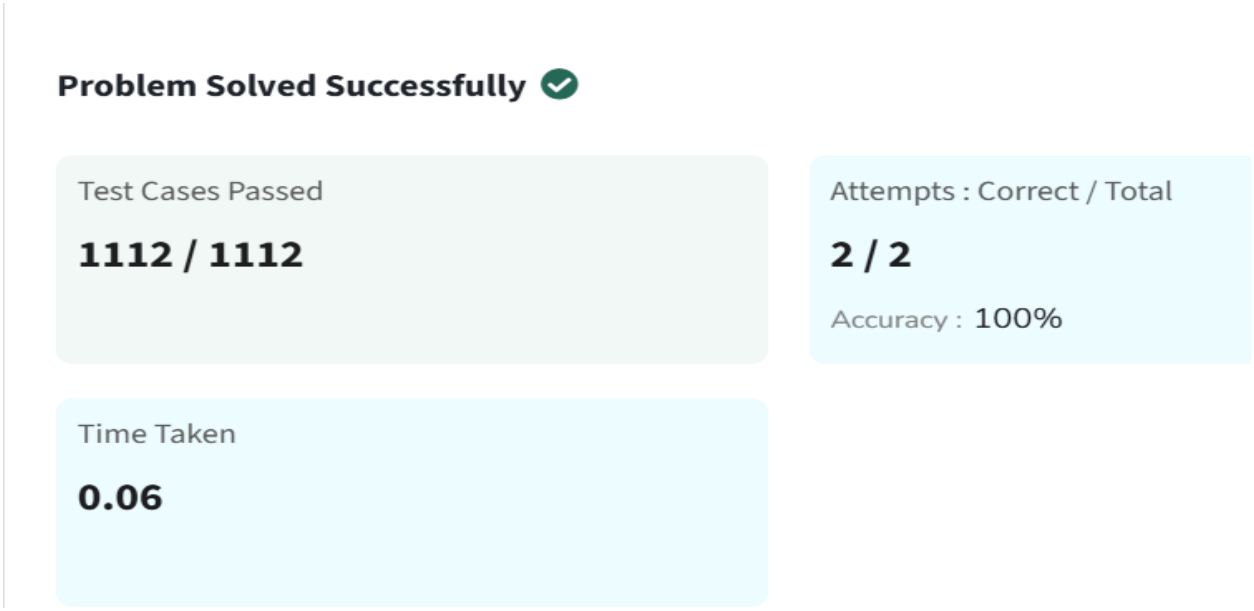
1. Print Linked List

Given a linked list. Print all the elements of the linked list separated by space followed.

CODE:

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

OUTPUT



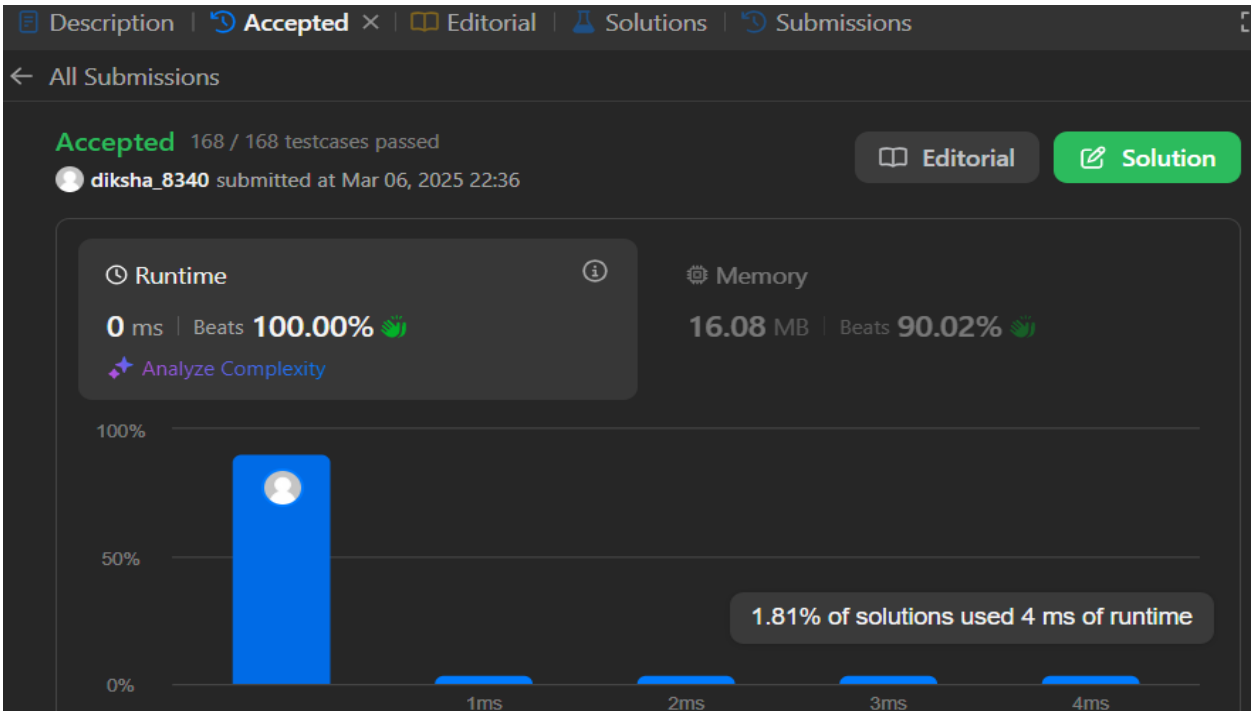
2. <u>Remove Duplicates from Sorted List</u>

Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list **sorted** as well*.

CODE

```
class Solution {
public:
   ListNode* deleteDuplicates(ListNode* head) {
      ListNode* curr = head;
      while(curr && curr->next){
         if(curr->val == curr->next->val){
            curr->next = curr->next->next;
         }
         else{
            curr = curr ->next;
         }
      }
      return head;
   }
};
```

OUTPUT:



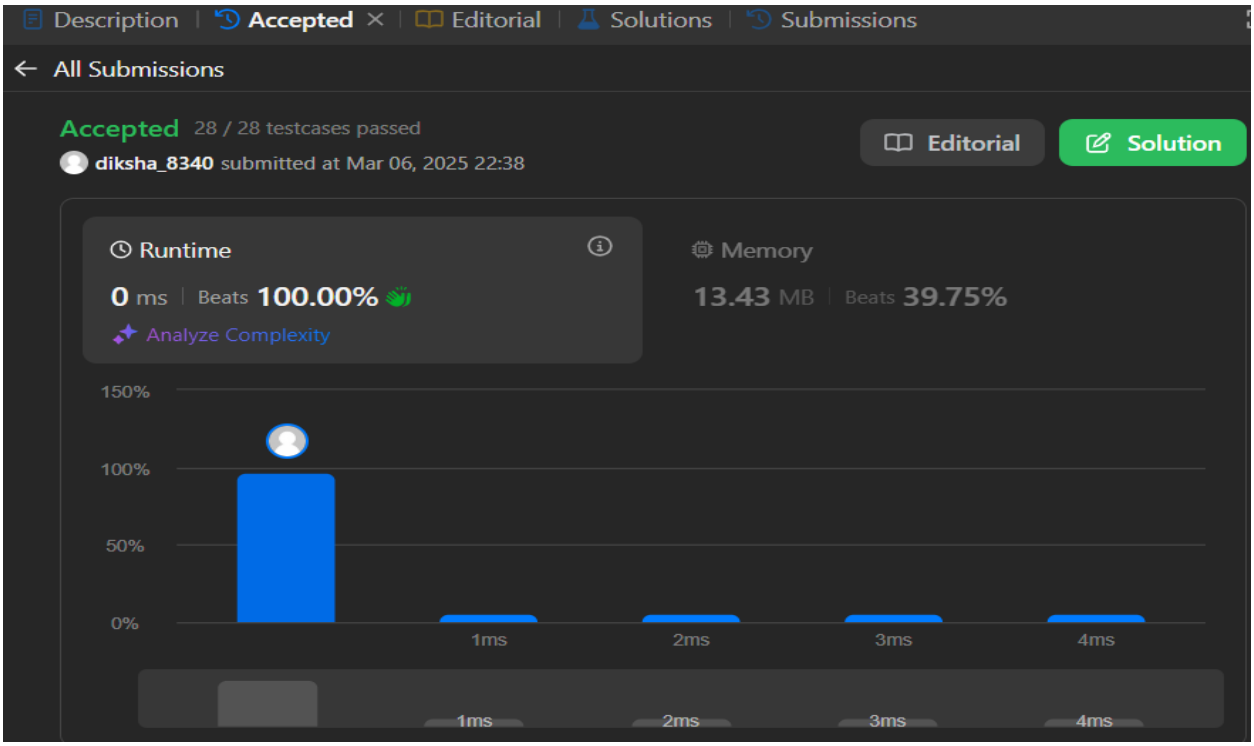
3. Reverse a Linked List

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

CODE:

```
class Solution {
public:
   ListNode* reverseList(ListNode* head) {
      ListNode* prev = NULL;
      ListNode* curr = head;

      while(curr != NULL){
       ListNode* forward = curr->next;
       curr ->next = prev;
       prev = curr;
       curr = forward;
      // forward = curr->next;
      }
      return prev;
   }
};
```

OUTPUT



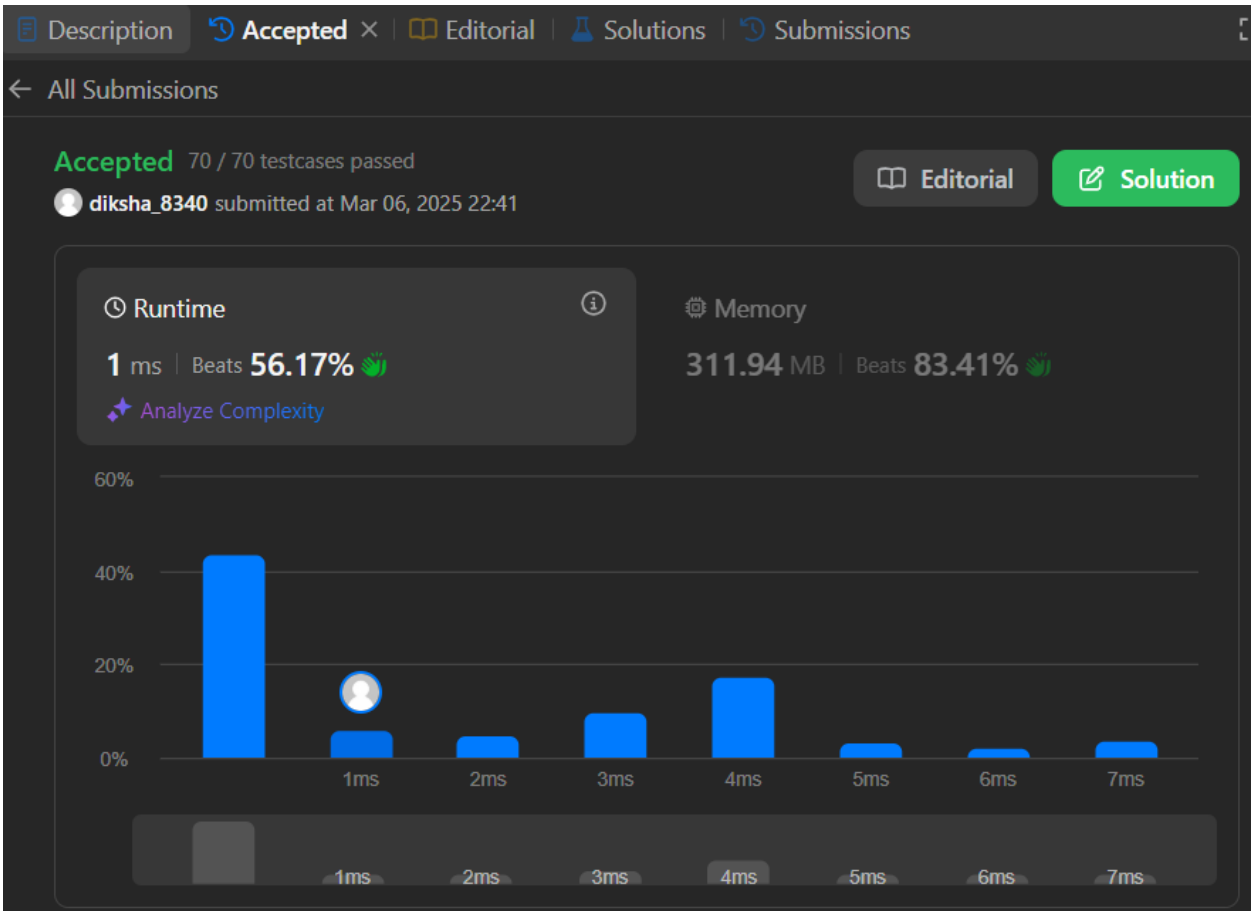
4. Delete middle node of a linked list

You are given the head of a linked list. **Delete** the **middle node**, and return *the* head *of the modified linked list*. The **middle node** of a linked list of size n is the $\lfloor n / 2 \rfloor^{th}$ node from the **start** using **0-based indexing**, where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x.

CODE:

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

OUTPUT:



5. Merge two sorted linked list

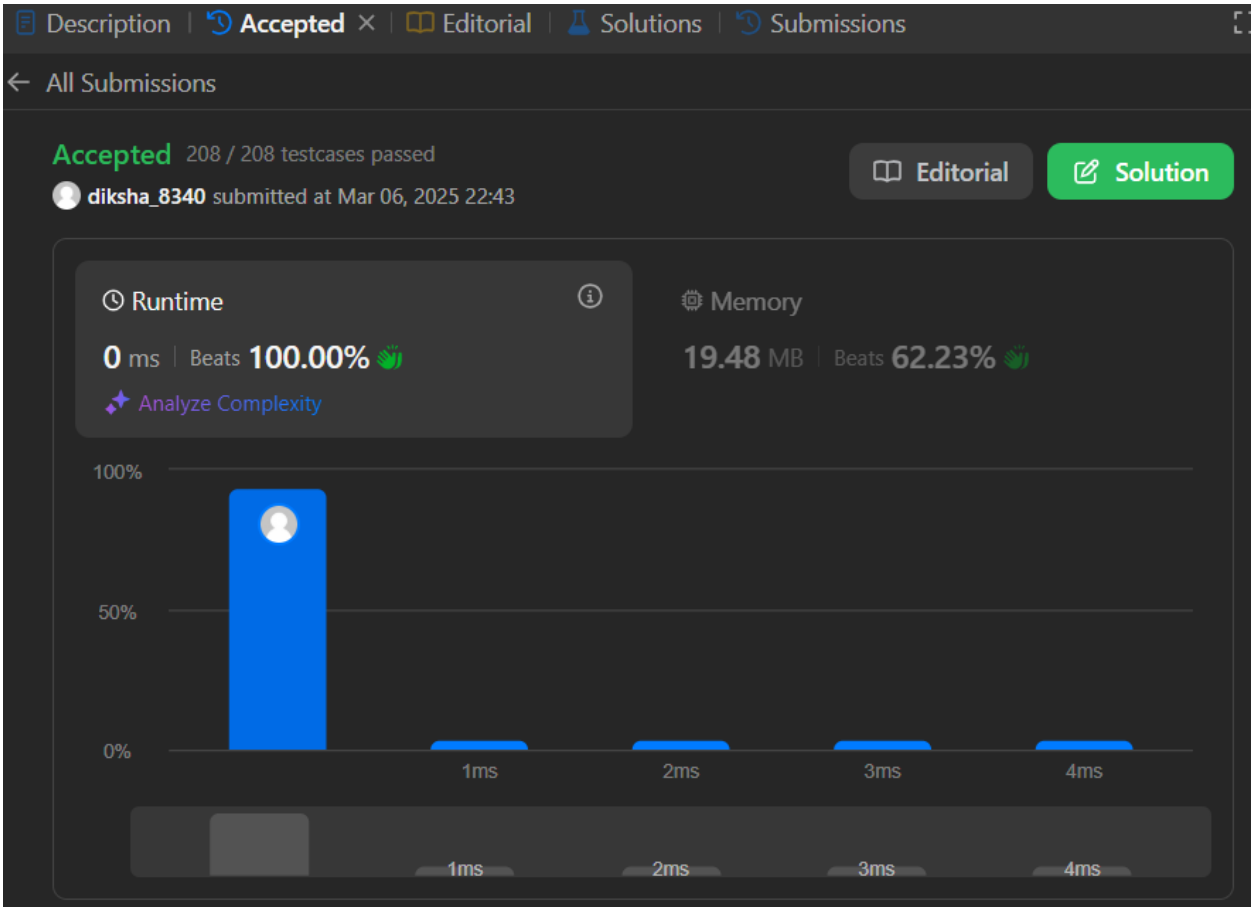
6. You are given the heads of two sorted linked lists list1 and list2.
7. Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.
8. Return *the head of the merged linked list*.

CODE:

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

```
        }
        ListNode* dummy = new ListNode(-1);
        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;
        }
        if(list1){
            tail->next = list1;
        }
        else{
            tail->next = list2;
        }
        return dummy->next;

    }
};
```

OUTPUT:



6. Detect cycle in a linked list:

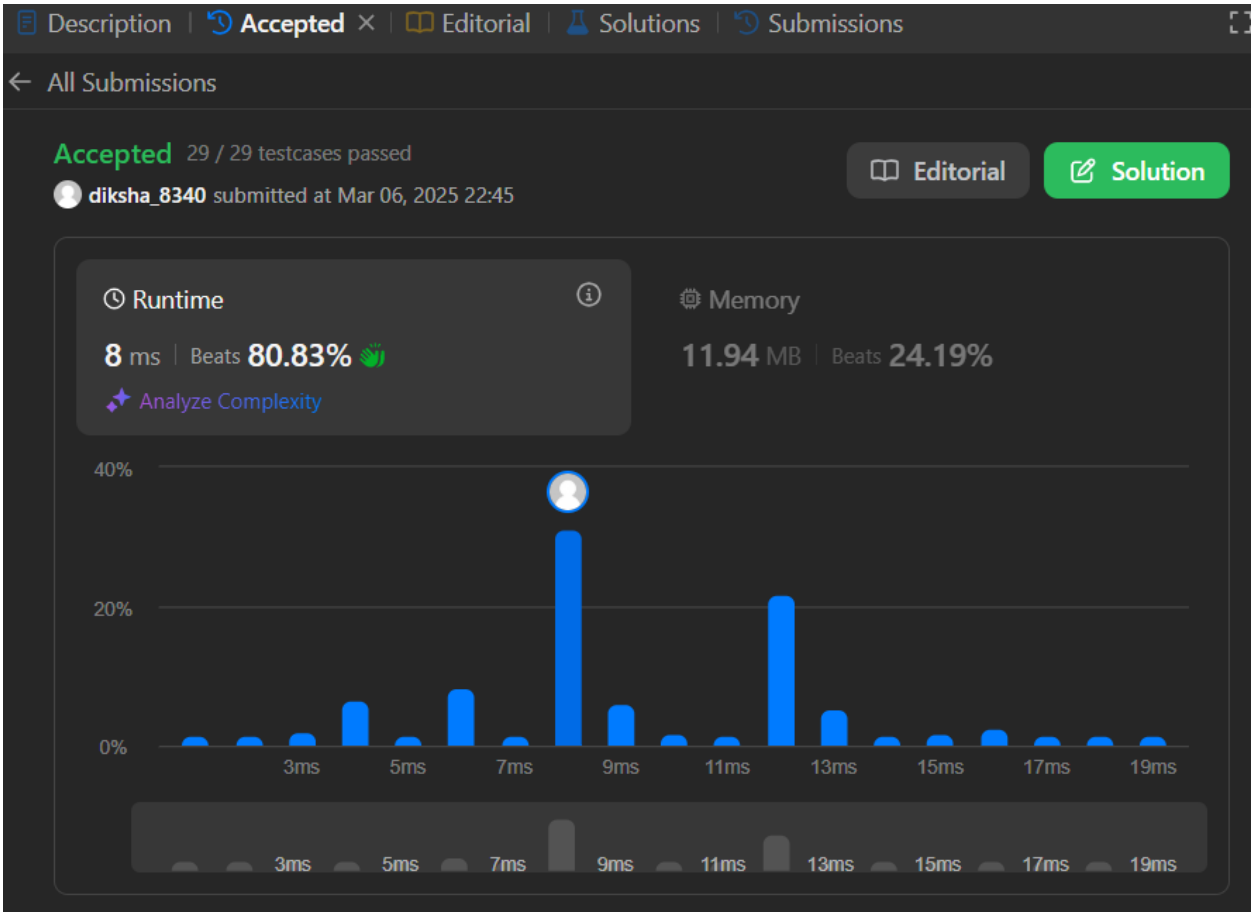
7. Given head, the head of a linked list, determine if the linked list has a cycle in it.
8. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.
9. Return true *if there is a cycle in the linked list*. Otherwise, return false.

CODE:

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

      ListNode *slow = head, *fast = head;

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

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

        return false;
    }
};
```

OUTPUT:



7. Rotate a list:

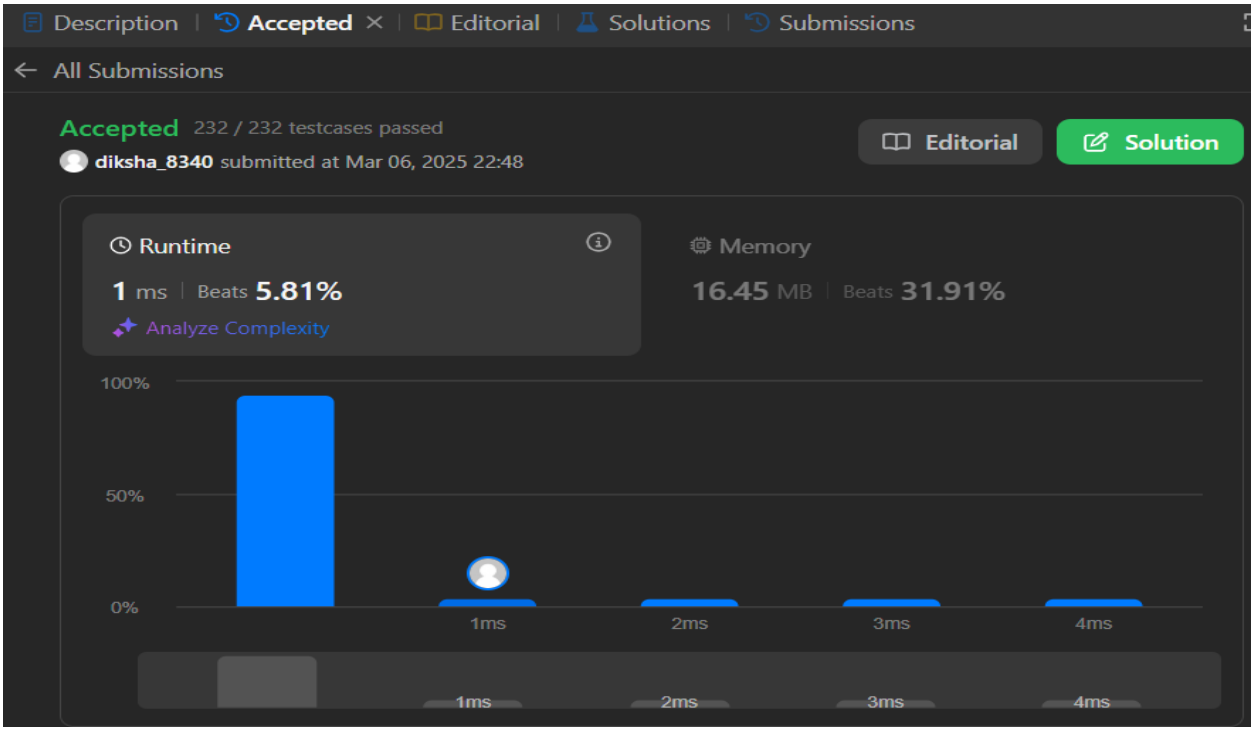
Given the head of a linked list, rotate the list to the right by k places.

CODE:

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

        ListNode* temp = head;
```

```
        int len = 1;
        while (temp->next) {
            temp = temp->next;
            len++;
        }

        temp->next = head;

        k = k % len;
        int stepsToNewTail = len - k;
        temp = head;
        for (int i = 1; i < stepsToNewTail; i++) {
            temp = temp->next;
        }

        head = temp->next;
        temp->next = nullptr;

        return head;
    }
};
```

OUTPUT:



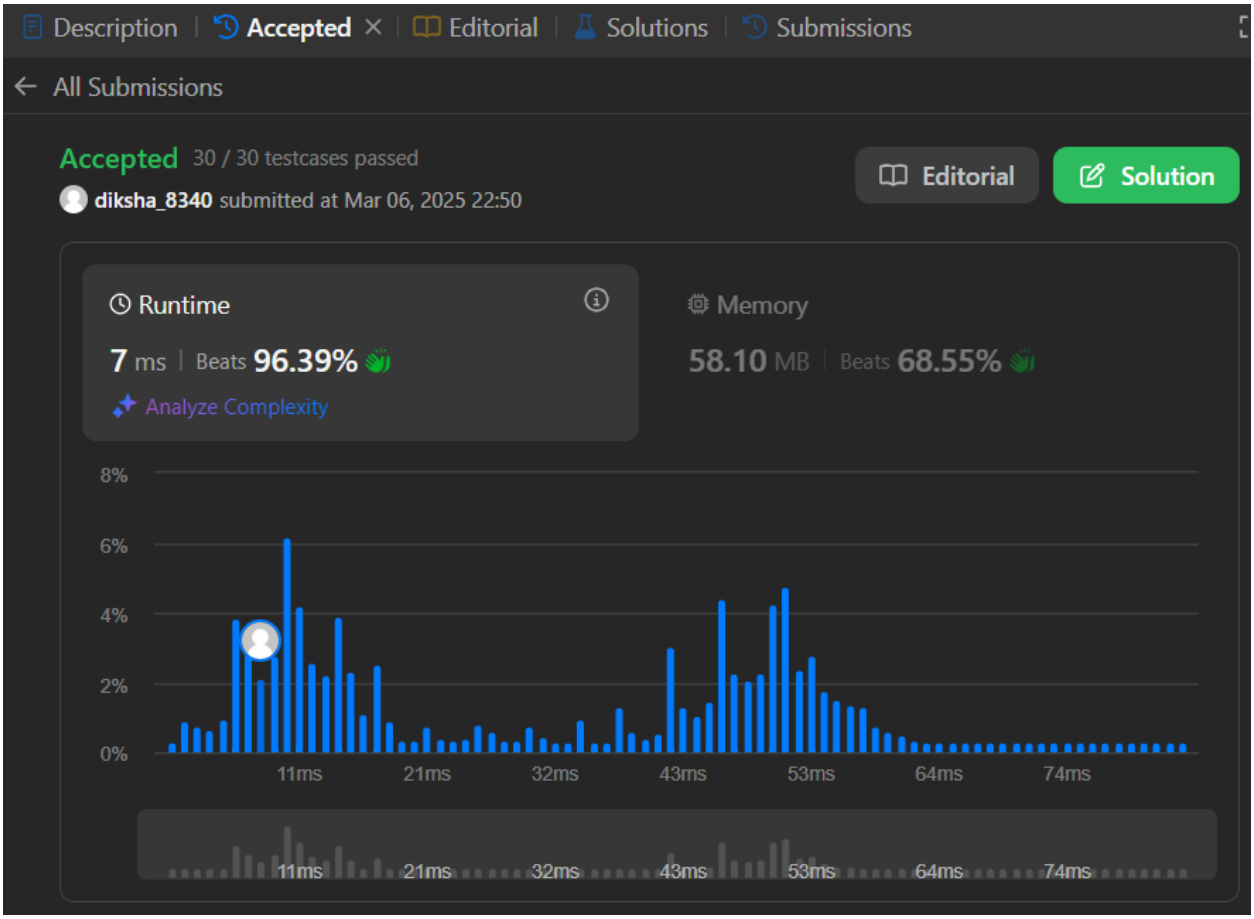
8. Sort a list

Given the head of a linked list, return *the list after sorting it in* ***ascending order***.

CODE:

```
class Solution {
public:
ListNode* getMid(ListNode* head) {
      ListNode *slow = head, *fast = head;
      while (fast && fast->next && fast->next->next) {
         slow = slow->next;
         fast = fast->next->next;
      }
      return slow;
   }

   ListNode* merge(ListNode* l1, ListNode* l2) {
      if (!l1) return l2;
      if (!l2) return l1;

      if (l1->val < l2->val) {
         l1->next = merge(l1->next, l2);
         return l1;
      } else {
         l2->next = merge(l1, l2->next);
         return l2;
      }
   }
   ListNode* sortList(ListNode* head) {
      if (!head || !head->next) return head;
      ListNode* mid = getMid(head);
      ListNode* right = mid->next;
      mid->next = nullptr;

      ListNode* left = sortList(head);
      right = sortList(right);

      return merge(left, right);
   }
};
```

OUTPUT:



9. Merge k sorted list

10. You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.
11. *Merge all the linked-lists into one sorted linked-list and return it.*

CODE:

```
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 (auto 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 = node;
        if (node->next) minHeap.push(node->next);
      }

      return dummy.next;
    }
};
```

OUTPUT: