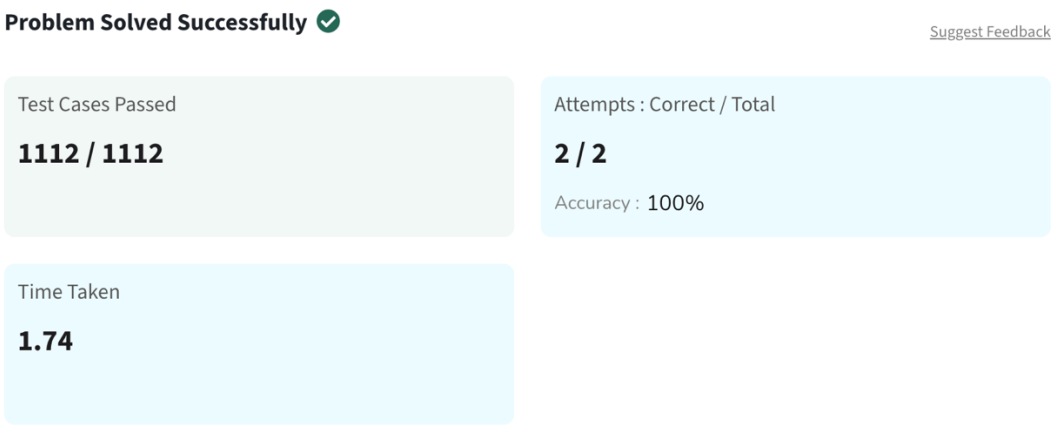
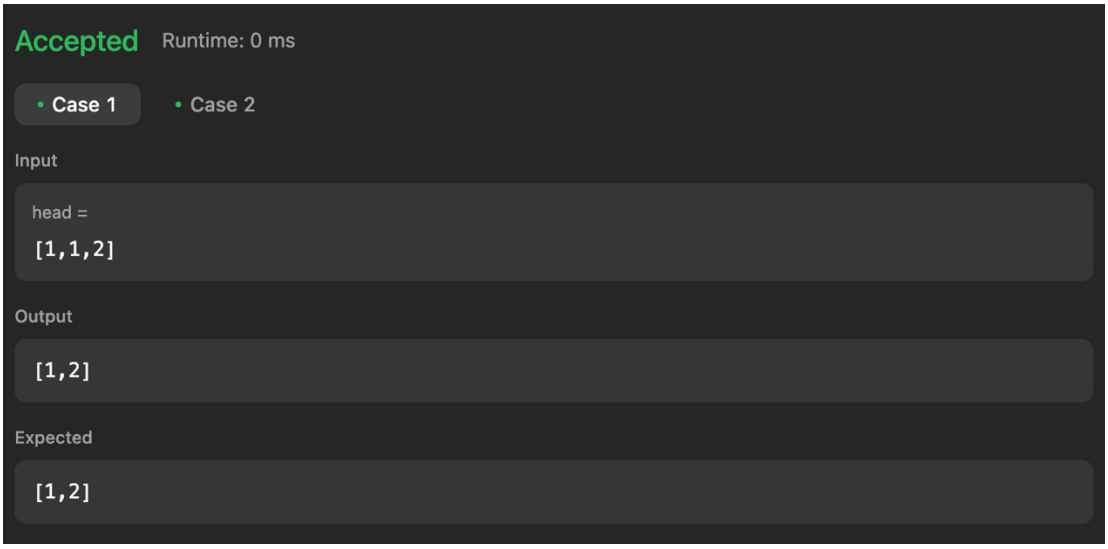
1. **Print Linked List:**

```
class Solution {
  // Function to display the elements of a linked list in same line
  void printList(Node head) {
    Node temp = head;

    while (temp != null) {
      System.out.print(temp.data + " ");
      temp = temp.next;
    }
  }
}
```



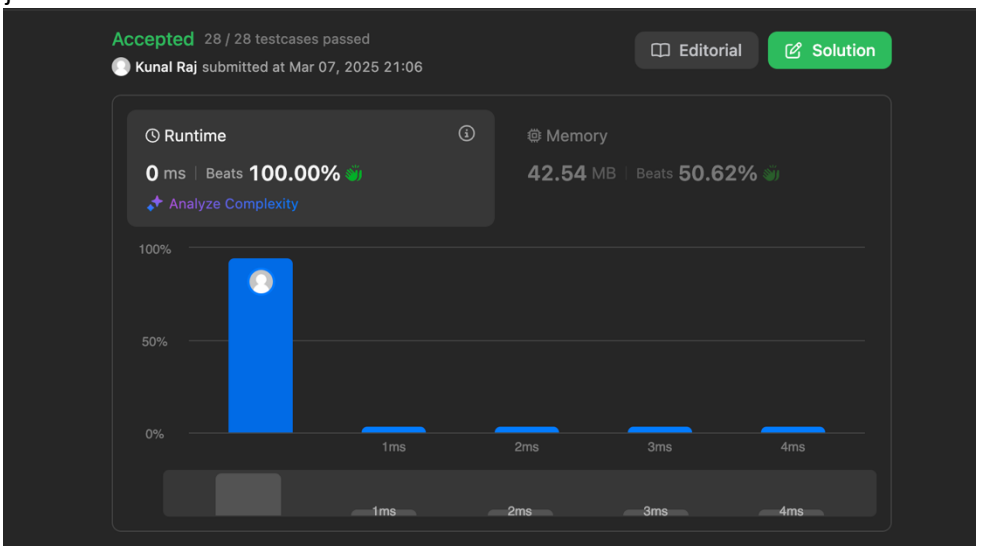
2. **Remove duplicates from a sorted list:**

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

    while (current != null && current.next != null) {
      if (current.val == current.next.val) {
        current.next = current.next.next; // Skip duplicate node
      } else {
        current = current.next; // Move to next node
      }
    }
    return head;
  }
}
```



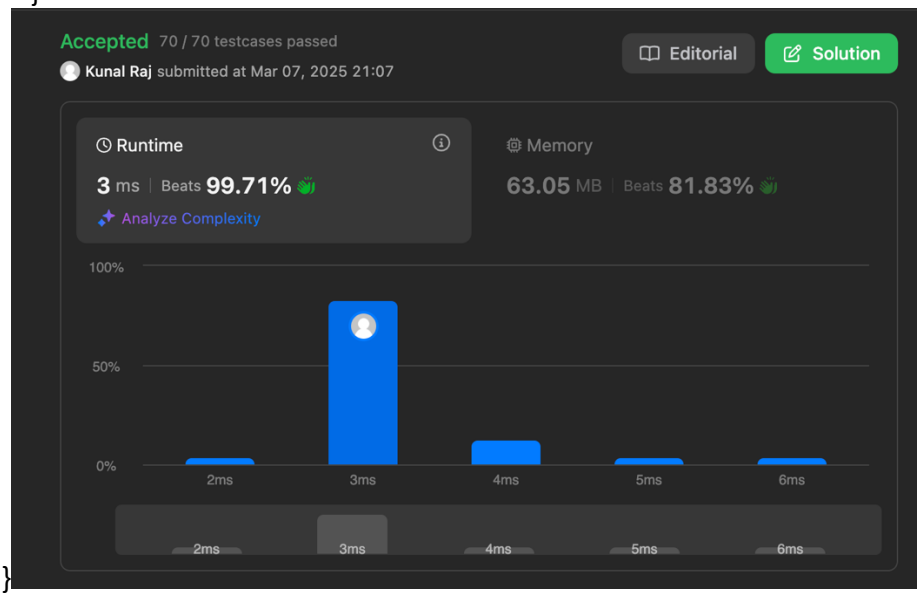
3. Reverse a linked list:

```
class Solution {
  public ListNode reverseList(ListNode head) {
    ListNode prev = null, current = head;

    while (current != null) {
      ListNode nextNode = current.next;
      current.next = prev;
      prev = current;
      current = nextNode;
    }
    return prev;
  }
}
```



4. **Delete middle node of a list:**

```
public class Solution {
  public ListNode deleteMiddle(ListNode head) {
    if (head == null || head.next == null) {
      return null;
    }

    ListNode slow = head, fast = head;
    ListNode prev = null;

    while (fast != null && fast.next != null) {
      prev = slow;
      slow = slow.next;
      fast = fast.next.next;
    }

    prev.next = slow.next;

    return head;
  }
```

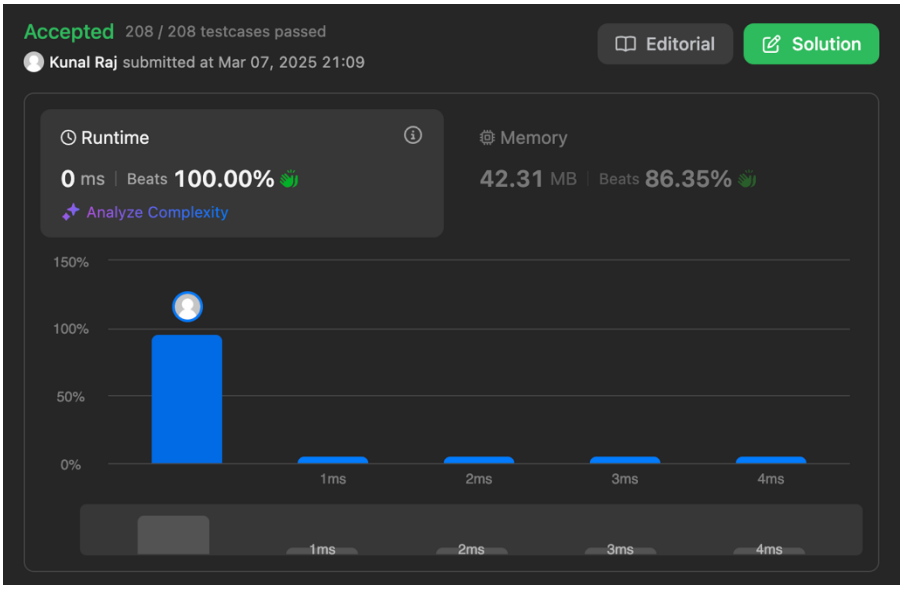



```
}
```

5. **Merge two sorted linked lists:**

```
class Solution {
  public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
    ListNode dummy = new ListNode(-1);
    ListNode current = dummy;

    while (list1 != null && list2 != null) {
      if (list1.val < list2.val) {
```

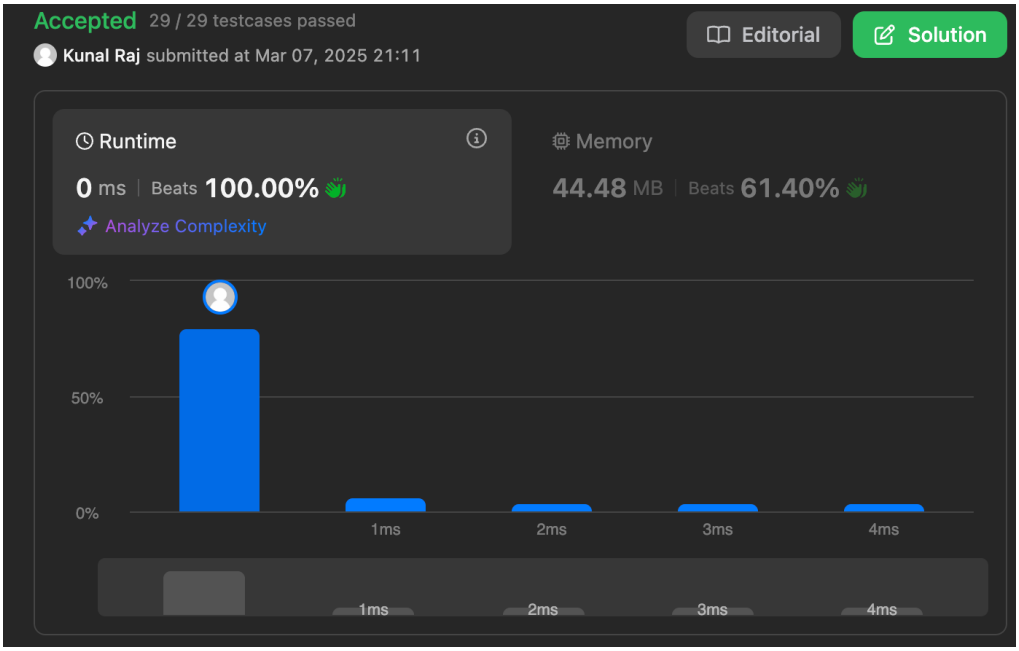
```
            current.next = list1;
            list1 = list1.next;
        } else {
            current.next = list2;
            list2 = list2.next;
        }
        current = current.next;
    }

    if (list1 != null) current.next = list1;
    if (list2 != null) current.next = list2;

    return dummy.next;
  }
}
```



6. **Detect a cycle in a linked list:**

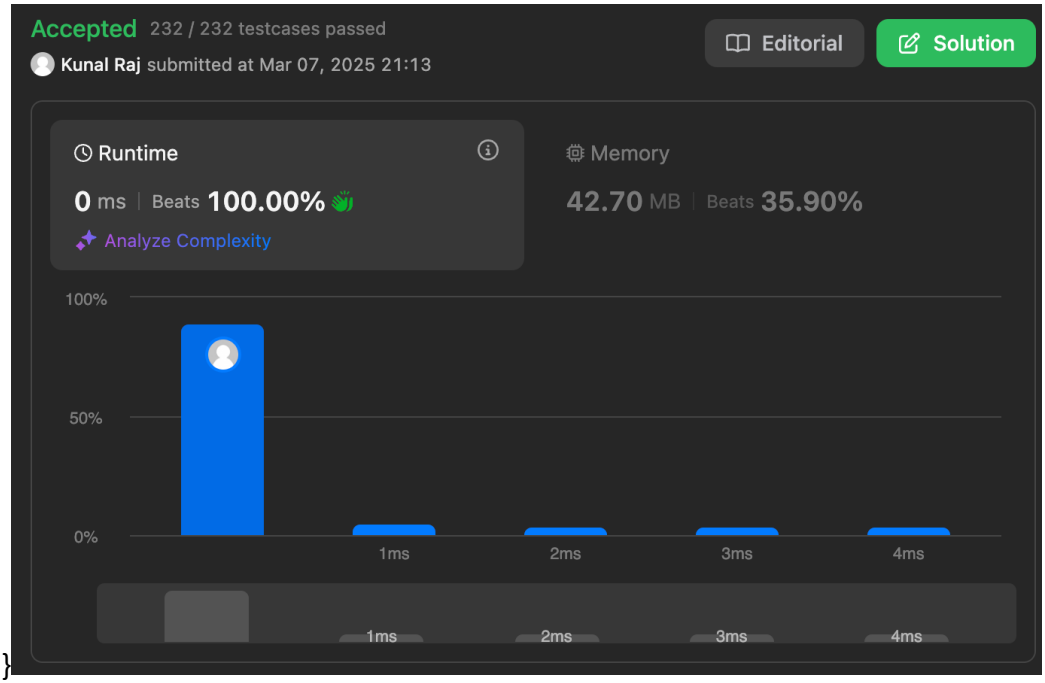
```
public class Solution {
  public boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) return false;

    ListNode slow = head, fast = head;

    while (fast != null && fast.next != null) {
      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 == null || head.next == null || k == 0) return head;

        ListNode temp = head;
        int length = 1;
        while (temp.next != null) {
            temp = temp.next;
            length++;
        }

        temp.next = head;

        int newTailIndex = length - k % length - 1;
        ListNode newTail = head;
        for (int i = 0; i < newTailIndex; i++) {
            newTail = newTail.next;
        }
        head = newTail.next;
        newTail.next = null;

        return head;
    }
```

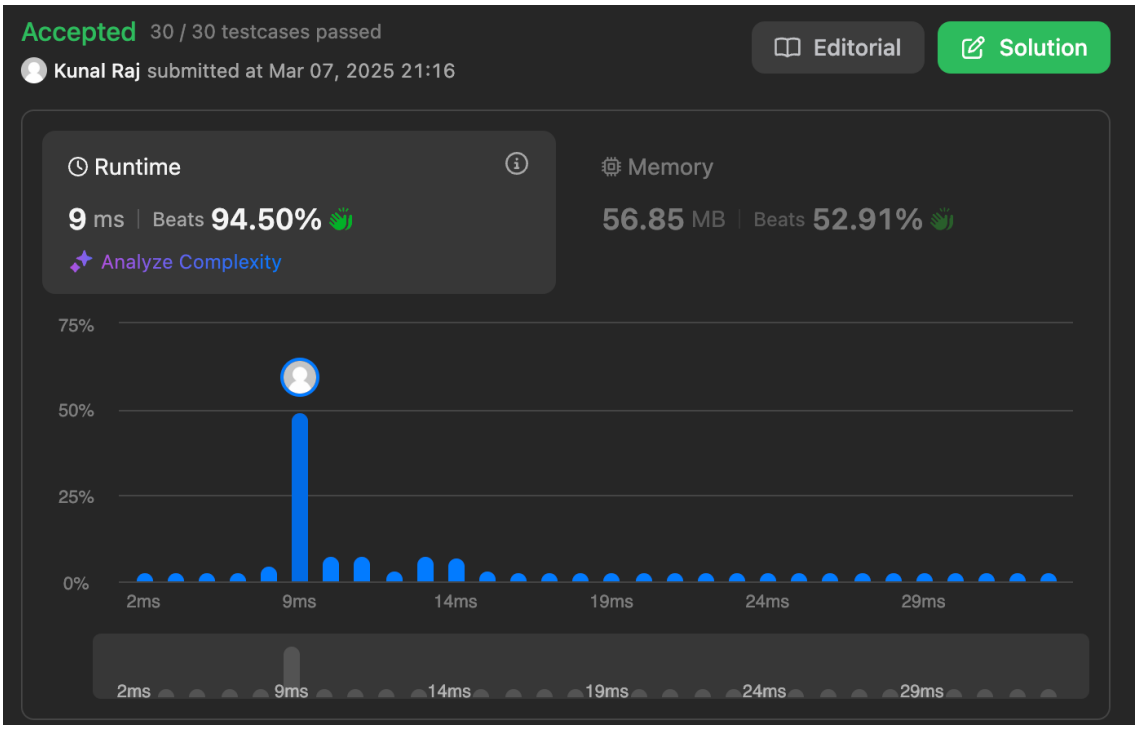


}

8. **Sort List:**

```
class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode mid = getMiddle(head);
        ListNode rightHead = mid.next;
        mid.next = null;

        ListNode left = sortList(head);
        ListNode right = sortList(rightHead);
        return merge(left, right);
    }

    private ListNode getMiddle(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

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

        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;
    }

    curr.next = (l1 != null) ? l1 : l2;
    return dummy.next;
  }
}
```



9. **Merge k sorted lists:**

```
import java.util.PriorityQueue;

class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) return null;

        PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val - b.val);
        for (ListNode list : lists) {
            if (list != null) minHeap.add(list);
        }

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

```
        ListNode curr = dummy;
        while (!minHeap.isEmpty()) {
            ListNode smallest = minHeap.poll();
            curr.next = smallest;
            curr = curr.next;
            if (smallest.next != null) {
                minHeap.add(smallest.next);
            }
        }
        return dummy.next;
    }
}
```