



## Experiment-3

**Student Name:** Anshuman Raj

**UID:** 22BET10081

**Branch:** BE-IT

**Section:** 22BET\_IOT-702 'A'

**Semester:** 6<sup>th</sup>

**Date of Performance:** 13/02/25

**Sub Name:** Advanced Programming Lab-2

**Subject Code:** 22ITP-351

### Problem 1

#### 1. Aim:

To implement and comprehend a program that identifies the presence of a cycle in a linked list using Floyd's Cycle Detection Algorithm (Tortoise and Hare approach).

#### 2. Objective:

- To develop skills in identifying and solving problems involving cyclic linked lists.
- To improve proficiency in implementing algorithms in Java for data structures.

#### 3. Code:

```
class ListNode {  
    int val;  
    ListNode next;  
    ListNode(int x) {  
        val = x;  
        next = null;  
    }  
  
    public static ListNode deserialize(String data) {  
        if (data == null || data.isEmpty() || data.equals("[]")) return null;  
        String[] values = data.replace("[", "").replace("]", "").split(",");  
        ListNode dummy = new ListNode(0);
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
ListNode current = dummy;
```

```
for (String value : values) {
```

```
    current.next = new ListNode(Integer.parseInt(value.trim()));
```

```
    current = current.next;
```

```
}
```

```
return dummy.next;
```

```
}
```

```
}
```

```
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;
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        String input = "[3,2,0,-4]";
```

```
        ListNode head = ListNode.deserialize(input);
```

```
        Solution solution = new Solution();
```

```
        boolean result = solution.hasCycle(head);
```

```
        System.out.println("Has Cycle: " + result);
```

```
    }
```

```
}
```

## 4. Output:

Description | Editorial | Solutions | Submissions

### 141. Linked List Cycle

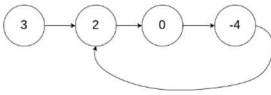
Easy | Topics | Companies

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

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.**

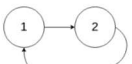
Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

**Example 1:**



**Input:** `head = [3,2,0,-4]`, `pos = 1`  
**Output:** `true`  
**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

**Example 2:**



16.2K | 352 | 192 Online

Code

```

24 }
25
26 public class Solution {
27     public boolean hasCycle(ListNode head) {
28         if (head == null || head.next == null) return false;
29
30         ListNode slow = head, fast = head;
31         while (fast != null && fast.next != null) {
32             slow = slow.next;
33             fast = fast.next.next;
34             if (slow == fast) return true;
35         }
36     }
37 }

```

Saved | Ln 48, Col 1

Testcase | Test Result

**Accepted** Runtime: 0 ms

Case 1 | Case 2 | Case 3

Input

head =  
[3,2,0,-4]

pos =  
1

Output

true

## 5. Learning Outcomes:

- Understand the concept and application of Floyd's Cycle Detection Algorithm for detecting cycles in linked lists.
- Learn to implement the algorithm efficiently and explain its working principle.
- Develop skills to test and debug the code for various cases, including empty lists, single-node lists, and cyclic lists.
- Analyze the algorithm's time and space complexity for better optimization.



## Problem 2

### 1. Aim:

To develop a program that reverses a specified section of a singly linked list between the given positions left and right.

### 2. Objective:

- Gain proficiency in traversing and manipulating a singly linked list using pointers.
- Understand and implement in-place reversal of a specific sublist within a linked list.

### 3. Code:

```
class ListNode {  
  
    int val;  
  
    ListNode next;  
  
    ListNode(int val) {  
  
        this.val = val;  
  
        this.next = null;  
  
    }  
  
    public static ListNode deserialize(String data) {  
  
        if (data == null || data.isEmpty() || data.equals("")) return null;  
  
        String[] values = data.replace("[", "").replace("]", "").split(",");  
  
        ListNode dummy = new ListNode(0);  
  
        ListNode current = dummy;
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
for (String value : values) {

    current.next = new ListNode(Integer.parseInt(value.trim()));

    current = current.next;

}

return dummy.next;

}

public static void printList(ListNode head) {

    while (head != null) {

        System.out.print(head.val + " -> ");

        head = head.next;

    }

    System.out.println("null");

}

}

public class Solution {

    public ListNode reverseBetween(ListNode head, int left, int right) {

        if (head == null || left == right) return head;

        ListNode dummy = new ListNode(0);

        dummy.next = head;

        ListNode prev = dummy;

        for (int i = 1; i < left; i++) {

            prev = prev.next;
```



```
}  
  
ListNode curr = prev.next;  
  
ListNode next;  
  
ListNode prevReversed = null;  
  
for (int i = 0; i <= right - left; i++) {  
  
    next = curr.next;  
  
    curr.next = prevReversed;  
  
    prevReversed = curr;  
  
    curr = next;  
  
}  
  
prev.next.next = curr;  
  
prev.next = prevReversed;  
  
return dummy.next;  
  
}  
  
public static void main(String[] args) {  
  
    String input = "[1,2,3,4,5]"; // Example input  
  
    int left = 2, right = 4;  
  
    ListNode head = ListNode.deserialize(input);  
  
    System.out.println("Original List:");  
  
    ListNode.printList(head);  
  
    Solution solution = new Solution();  
  
    head = solution.reverseBetween(head, left, right);
```

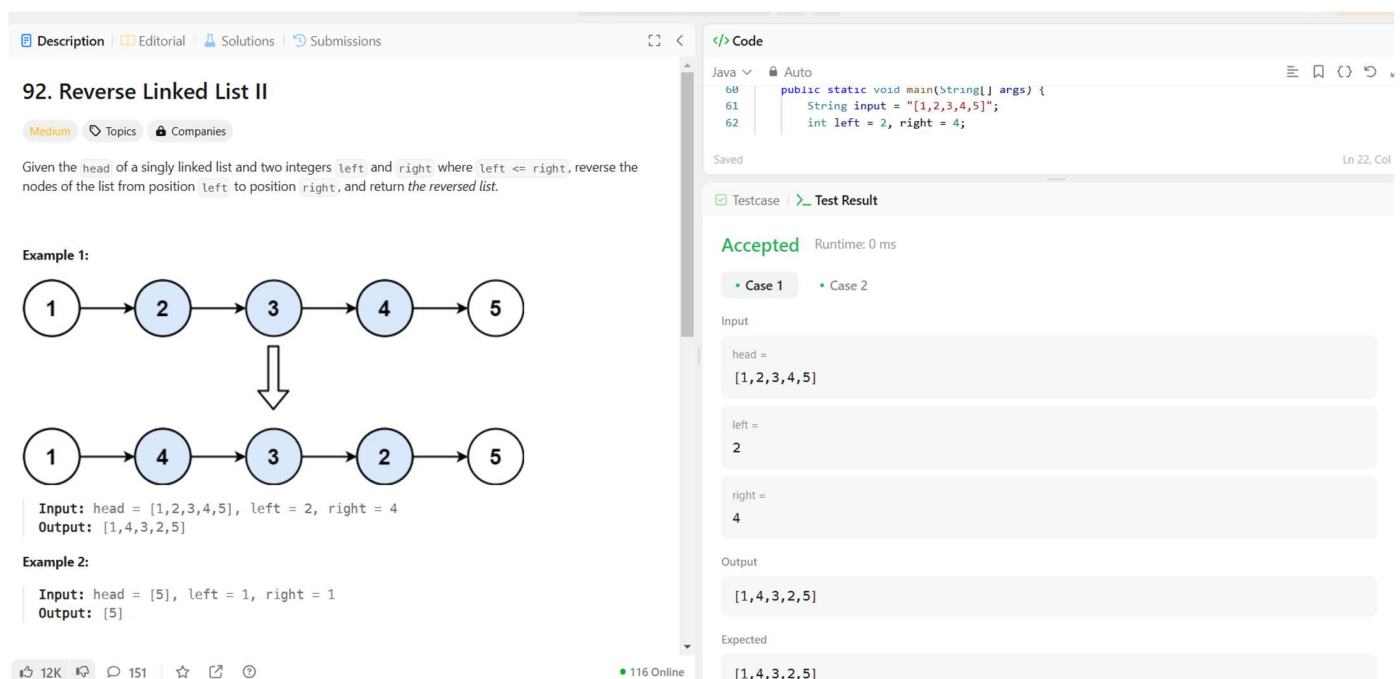
```
System.out.println("Reversed List:");
```

```
ListNode.printList(head);
```

```
}
```

```
}
```

## 4. Output:



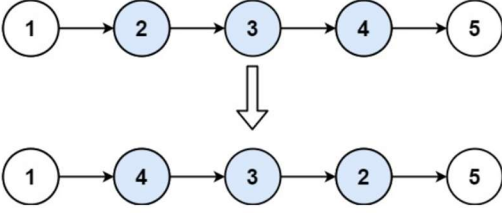
The screenshot displays a coding problem interface. On the left, the problem description for '92. Reverse Linked List II' is shown, including a diagram of a linked list transformation. The diagram illustrates a linked list with nodes 1, 2, 3, 4, 5. An arrow points to the transformed list where nodes 2, 3, and 4 are reversed, resulting in 1, 4, 3, 2, 5. Below the diagram, the input and output are specified: Input: head = [1,2,3,4,5], left = 2, right = 4; Output: [1,4,3,2,5]. Example 2 is also provided: Input: head = [5], left = 1, right = 1; Output: [5]. On the right, the 'Code' editor shows a Java solution. The code defines a main method that takes a string input '[1,2,3,4,5]', parses it into an array, and then calls a function to reverse the linked list between indices 2 and 4. The 'Test Result' section shows the test case 'Case 1' as 'Accepted' with a runtime of 0 ms. The input fields show 'head = [1,2,3,4,5]', 'left = 2', and 'right = 4'. The output field shows '[1,4,3,2,5]', which matches the expected output.

**92. Reverse Linked List II**

Medium Topics Companies

Given the head of a singly linked list and two integers left and right where left ≤ right, reverse the nodes of the list from position left to position right, and return the reversed list.

**Example 1:**



Input: head = [1,2,3,4,5], left = 2, right = 4  
Output: [1,4,3,2,5]

**Example 2:**

Input: head = [5], left = 1, right = 1  
Output: [5]

12K 151 116 Online

```
Java Auto
60 public static void main(String[] args) {
61     String input = "[1,2,3,4,5]";
62     int left = 2, right = 4;
}
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

head =  
[1,2,3,4,5]

left =  
2

right =  
4

Output

[1,4,3,2,5]

Expected

[1,4,3,2,5]

## 5. Learning Outcomes:

- Understand and implement Floyd's Cycle Detection Algorithm for linked list cycle detection.
- Learn to traverse and manipulate a singly linked list efficiently using pointers.
- Develop the ability to reverse a specific portion of a linked list in-place.
- Enhance debugging skills by handling edge cases like empty lists, single-node lists, and cyclic lists.



## Problem 3

### 1.Aim:

To develop a program that efficiently rotates a singly linked list to the right by a specified number of positions, while handling edge cases and large inputs effectively.

### 2. Objective:

- Understand the concept and implementation of rotating a singly linked list.
- Practice in-place modifications, including creating and breaking circular connections in linked lists.

### 3.Code:

```
class ListNode {  
  
    int val;  
  
    ListNode next;  
  
    ListNode(int val) {  
  
        this.val = val;  
  
        this.next = null;  
  
    }  
  
    public static ListNode deserialize(String data) {  
  
        if (data == null || data.isEmpty() || data.equals("")) return null;  
  
        String[] values = data.replace("[", "").replace("]", "").split(",");  
  
        ListNode dummy = new ListNode(0);  
  
        ListNode current = dummy;  
  
        for (String value : values) {  
  
            current.next = new ListNode(Integer.parseInt(value.trim()));  
  
            current = current.next;  

```





# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
}

return dummy.next;

}

public static void printList(ListNode head) {

    while (head != null) {

        System.out.print(head.val + " -> ");

        head = head.next;

    }

    System.out.println("null");

}

}

public class Solution {

    public ListNode rotateRight(ListNode head, int k) {

        if (head == null || head.next == null || k == 0) return head;

        int length = 1;

        ListNode tail = head;

        while (tail.next != null) {

            tail = tail.next;

            length++;

        }

        k = k % length;

        if (k == 0) return head; // No change needed

        ListNode prev = head;

        for (int i = 1; i < length - k; i++) {
```



```
prev = prev.next;

}

ListNode newHead = prev.next;

prev.next = null;

tail.next = head; // Connect old tail to old head

return newHead;

}

public static void main(String[] args) {

String input = "[1,2,3,4,5]";

int k = 2;

ListNode head = ListNode.deserialize(input);

System.out.println("Original List:");

ListNode.printList(head);

Solution solution = new Solution();

head = solution.rotateRight(head, k);

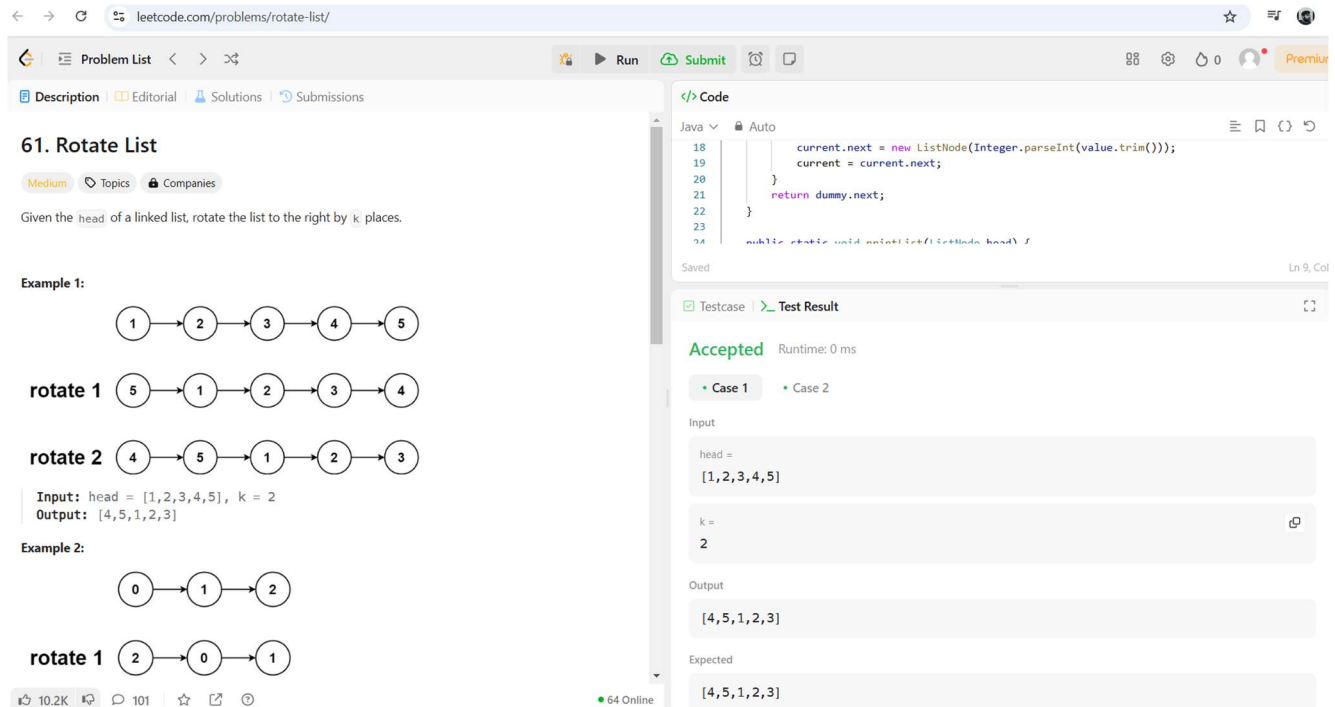
System.out.println("Rotated List:");

ListNode.printList(head);

}

}
```

## 4. Output:



61. Rotate List

Medium

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

Example 1:

1 → 2 → 3 → 4 → 5

rotate 1

5 → 1 → 2 → 3 → 4

rotate 2

4 → 5 → 1 → 2 → 3

Input: head = [1,2,3,4,5], k = 2  
Output: [4,5,1,2,3]

Example 2:

0 → 1 → 2

rotate 1

2 → 0 → 1

10.2K 101 64 Online

```

18  current.next = new ListNode(Integer.parseInt(value.trim()));
19  current = current.next;
20  }
21  return dummy.next;
22  }
23
24  public static void printList(ListNode head) {

```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

head = [1,2,3,4,5]

k = 2

Output

[4,5,1,2,3]

Expected

[4,5,1,2,3]

## 5. Learning Outcomes:

- Understand the concept and implementation of rotating a singly linked list.
- Develop skills in optimizing operations by handling edge cases efficiently.
- Practice in-place modifications, including updating pointers for rotation.
- Enhance debugging abilities by testing various input scenarios.

## Problem 4

### 1.Aim:

To implement an efficient algorithm that merges k sorted linked lists into a single sorted linked list using priority queues or divide-and-conquer techniques.

### 2.Objective:

- Understand the concept of merging multiple sorted linked lists into a single sorted list.
- Explore and implement various approaches to solve the problem, including:
- Utilizing a min-heap (priority queue) for efficient merging.
- Applying a divide-and-conquer strategy to iteratively merge pairs of lists.

### 3.Code:

```
import java.util.PriorityQueue;

public 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.offer(list);
            }
        }
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;
        while (!minHeap.isEmpty()) {
            ListNode smallest = minHeap.poll();
            current.next = smallest;
            current = current.next;
            if (smallest.next != null) {
                minHeap.offer(smallest.next);
            }
        }
        return dummy.next;
    }
}
```

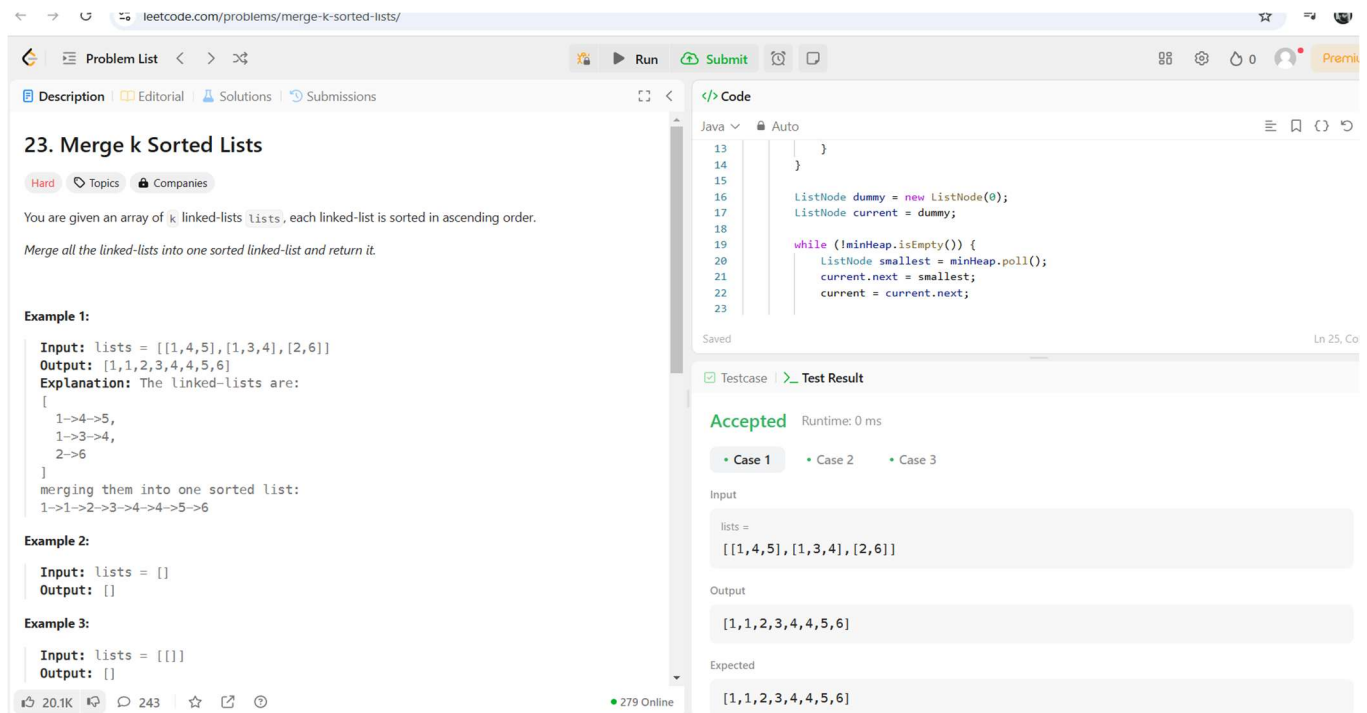
```

    }
}

return dummy.next;
}
}

```

## 4. Output:



23. Merge k Sorted Lists

Hard Topics Companies

You are given an array of  $k$  linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

**Example 1:**

Input: `lists = [[1,4,5],[1,3,4],[2,6]]`  
Output: `[1,1,2,3,4,4,5,6]`  
Explanation: The linked-lists are:  

```

1->4->5,
1->3->4,
2->6

```

merging them into one sorted list:  
1->1->2->3->4->4->5->6

**Example 2:**

Input: `lists = []`  
Output: `[]`

**Example 3:**

Input: `lists = [[]]`  
Output: `[]`

20.1K 243 279 Online

```

13     }
14     }
15
16     ListNode dummy = new ListNode(0);
17     ListNode current = dummy;
18
19     while (!minHeap.isEmpty()) {
20         ListNode smallest = minHeap.poll();
21         current.next = smallest;
22         current = current.next;
23     }

```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

`lists =`  
`[[1,4,5],[1,3,4],[2,6]]`

Output

`[1,1,2,3,4,4,5,6]`

Expected

`[1,1,2,3,4,4,5,6]`

## 5. Learning Outcomes:

- Understand how to merge multiple sorted linked lists into a single sorted list.
- Learn to implement efficient merging using a min-heap (priority queue).
- Explore the divide-and-conquer approach for merging lists.
- Analyze and compare different merging strategies based on time and space complexity.

## Problem 5

### 1.Aim:

To develop an algorithm for sorting a singly linked list in ascending order using efficient techniques such as merge sort or quick sort.

### 2.Objective:

- Understand and implement efficient sorting techniques for singly linked lists.
- Explore merge sort and quick sort for linked list sorting.
- Analyze the time and space complexity of different sorting approaches.
- Develop skills in manipulating linked lists for in-place sorting.

### 3.Code:

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

```
        ListNode mid = getMiddle(head);
```

```
        ListNode left = head;
```

```
        ListNode right = mid.next;
```

```
        mid.next = null; // Break the list
```

```
        left = sortList(left);
```

```
        right = sortList(right);
```

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

```
    }
```

```
    private ListNode getMiddle(ListNode head) {
```

```
        ListNode slow = head, fast = head.next;
```

```
        while (fast != null && fast.next != null) {
```

```
            slow = slow.next;
```



# DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
fast = fast.next.next;
}
return slow;
}
private ListNode merge(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

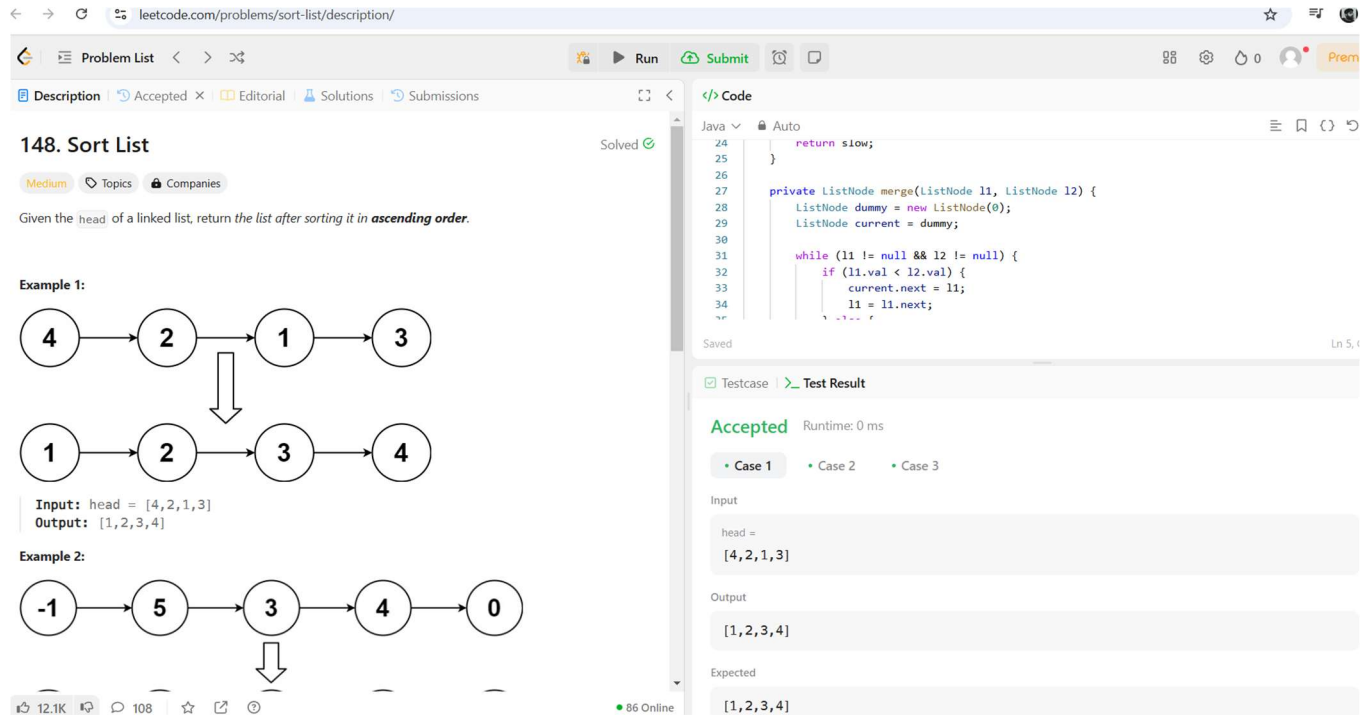
    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {

            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    if (l1 != null) current.next = l1;
    if (l2 != null) current.next = l2;

    return dummy.next;
}
}
```

## 4. Output:



**148. Sort List**

Medium Topics Companies

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

**Example 1:**

Input: `head = [4,2,1,3]`  
Output: `[1,2,3,4]`

**Example 2:**

Input: `head = [-1,5,3,4,0]`  
Output: `[1,2,3,4]`

**Code:**

```
Java
24     return slow;
25 }
26
27 private ListNode merge(ListNode l1, ListNode l2) {
28     ListNode dummy = new ListNode(0);
29     ListNode current = dummy;
30
31     while (l1 != null && l2 != null) {
32         if (l1.val < l2.val) {
33             current.next = l1;
34             l1 = l1.next;
35         } else {
36             current.next = l2;
37             l2 = l2.next;
38         }
39         current = current.next;
40     }
41     current.next = l1 != null ? l1 : l2;
42     return dummy.next;
43 }
```

**Testcase | Test Result**

**Accepted** Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =  
[4,2,1,3]

Output

[1,2,3,4]

Expected

[1,2,3,4]

## 5. Learning Outcomes:

- Gain proficiency in sorting a singly linked list using efficient algorithms.
- Understand the working principles of merge sort and quick sort for linked lists.
- Learn to optimize sorting operations while considering time and space complexity.
- Develop problem-solving skills for handling linked list-based sorting challenges.