**Experiment 3**

**Student Name:** Dixit Balotra  **UID:** 22BET10238

**Branch:** BE -IT  **Section/Group:**22BET\_IOT-702/B

**Semester:** 6th  **Date of Performance:**31/1/2025

**Subject Name:** Advanced Programming Lab-2  **Subject Code:** 22ITP-351

**1. Remove duplicates from a sorted list**

## Aim :

To develop an algorithm that removes duplicate elements from a sorted linked list and returns a list containing only unique elements.

## Objective :

* Understand how duplicate elements occur in a sorted linked list.
* Implement an efficient approach to remove duplicates.
* Ensure that the relative order of elements is maintained.
* Optimize the solution to run in O(n) time complexity.

## Implementation/Code :

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

class Solution {

public ListNode deleteDuplicates(ListNode head) {

if (head == null || head.next == null) {

return head; // If list is empty or has only one node, return as is

}

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 distinct node

}

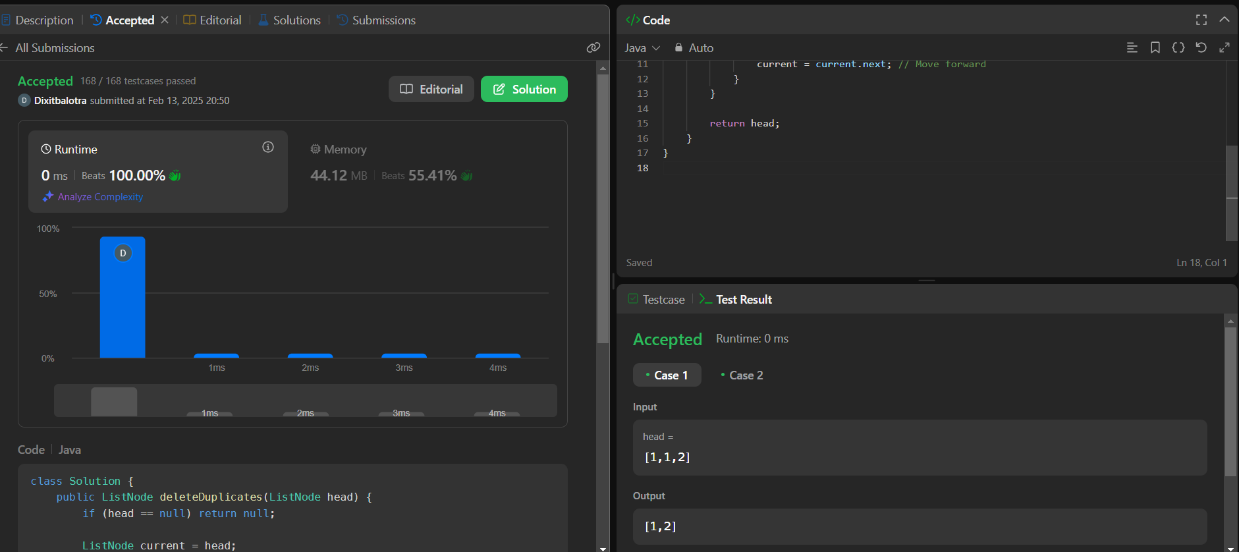
}

return head;

    }

}

## Output :



## 2. Detect a cycle in a linked list

**Aim :**

To implement an algorithm that detects whether a given singly linked list contains a cycle using Floyd’s Cycle Detection Algorithm (Tortoise and Hare approach).

**Objective :**

* Understand the concept of cycles in linked lists.
* Implement the Floyd’s Cycle Detection Algorithm.
* Efficiently determine if a cycle exists in a given linked list.
* Analyze the time and space complexity of the algorithm.

**Implementation/Code :**

public class Solution {

public boolean hasCycle(ListNode head) {

if (head == null || head.next == null) {

return false; // No cycle if list is empty or has only one node

}

ListNode slow = head; // Moves one step at a time

ListNode fast = head; // Moves two steps at a time

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

slow = slow.next; // Move slow pointer one step

fast = fast.next.next; // Move fast pointer two steps

if (slow == fast) { // If they meet, cycle exists

return true;

}

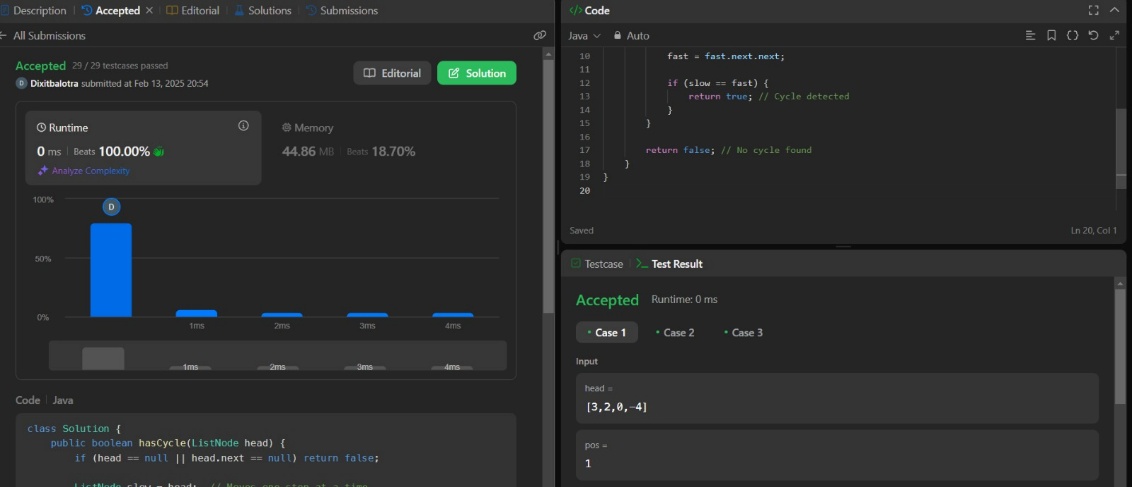
}

return false; // No cycle found

}

}

**Output :**



**3. Reverse linked list**

**Aim :**

To reverse a given singly linked list such that the last node becomes the first and the first node becomes the last.

**Objective :**

* Understand the structure and traversal of a singly linked list.
* Implement an iterative and recursive approach to reverse a linked list.
* Analyze and compare the time complexity of different approaches.
* Optimize memory usage while reversing the linked list.

**Implementation/Code :**

public class Solution {

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

if (head == null || left == right) {

return head; // No need to reverse if the list is empty or left == right

}

// Step 1: Create a dummy node to simplify edge cases

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy;

// Step 2: Move `prev` to the node just before `left`

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

prev = prev.next;

}

// Step 3: Reverse the sublist from `left` to `right`

ListNode current = prev.next;

ListNode nextNode;

ListNode previous = null;

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

nextNode = current.next;

current.next = previous;

previous = current;

current = nextNode;

}

// Step 4: Connect the reversed sublist with the remaining list

prev.next.next = current;

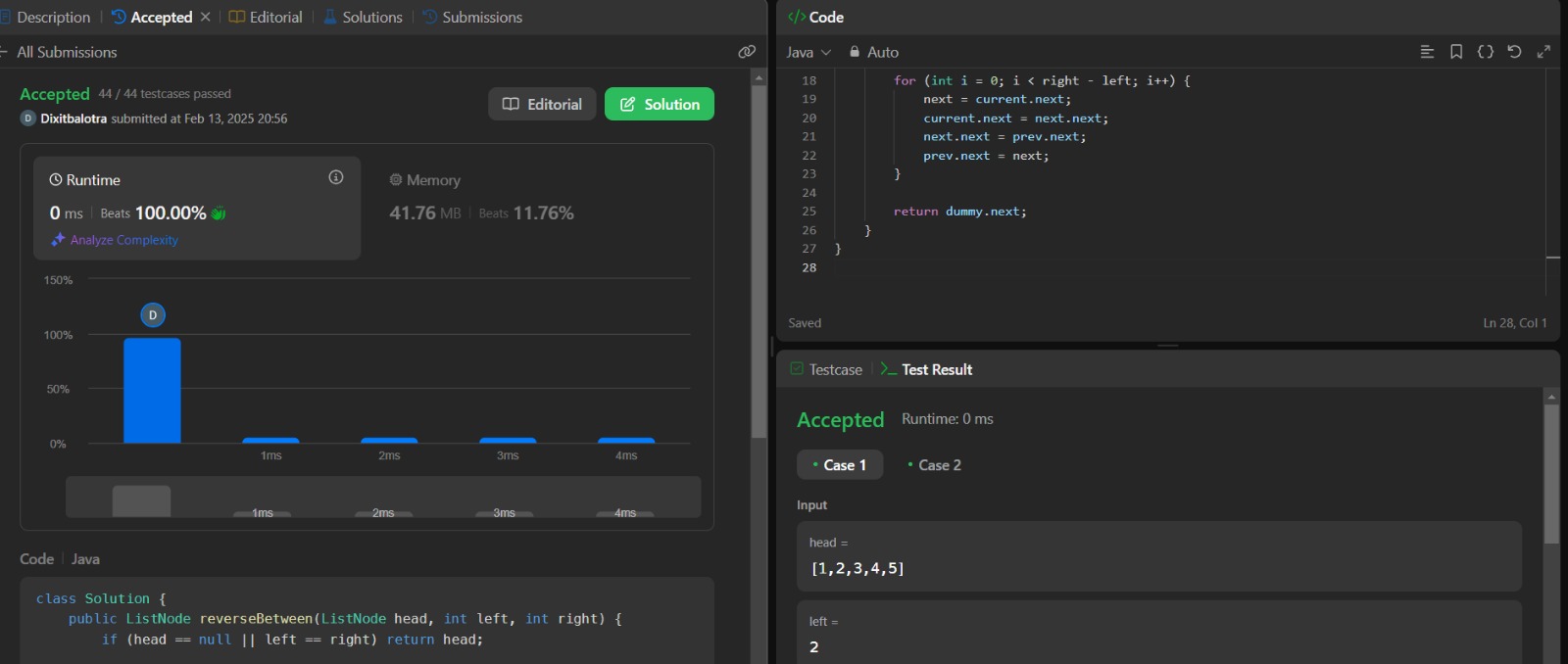
prev.next = previous;

return dummy.next;

}

}

**Output :**



**4. Rotate a list**

**Aim :**

To rotate a singly linked list by a given number of positions either to the left or right.

**Objective :**

* Understand how rotating a linked list affects its structure.
* Implement an efficient algorithm to rotate a linked list by k positions.
* Optimize the solution to handle large values of k efficiently.
* Ensure edge cases such as empty lists and k being greater than the length are handled properly.

**Implementation/Code :**

public class Solution {

public ListNode rotateRight(ListNode head, int k) {

if (head == null || head.next == null || k == 0) {

return head; // No rotation needed

}

// Step 1: Find the length of the linked list

ListNode current = head;

int length = 1; // Start counting from 1 (head exists)

while (current.next != null) {

current = current.next;

length++;

}

// Step 2: Connect the tail to the head to form a circular list

current.next = head;

// Step 3: Find the new tail (length - k % length - 1) and new head (length - k % length)

k = k % length; // Handle cases where k > length

int newTailPosition = length - k;

ListNode newTail = head;

for (int i = 1; i < newTailPosition; i++) { // Find the new tail

newTail = newTail.next;

}

// Step 4: Break the cycle and set new head

ListNode newHead = newTail.next;

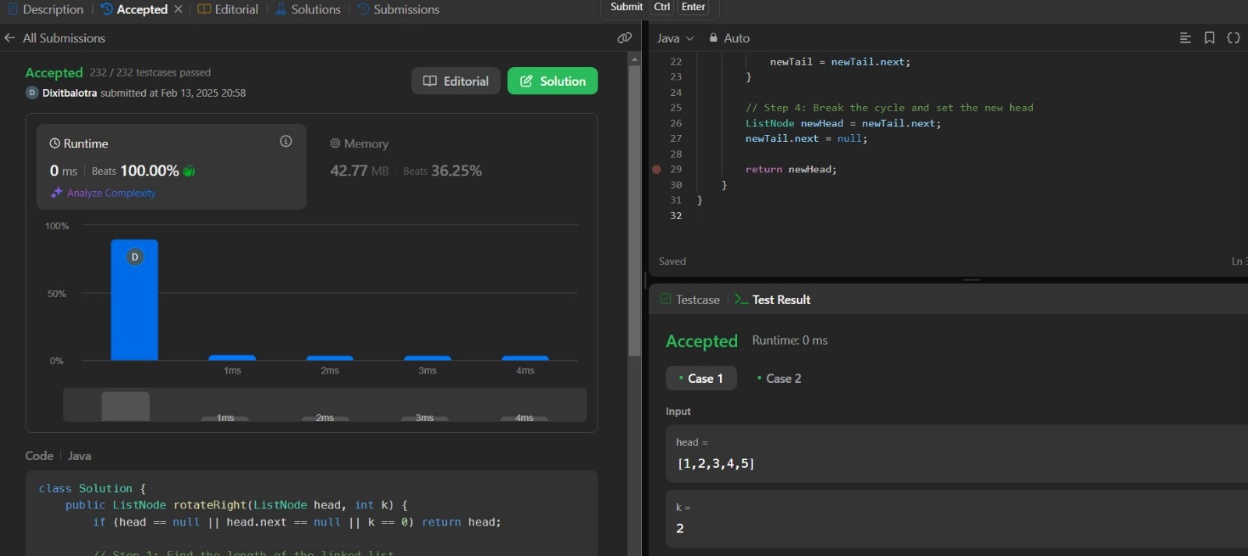
newTail.next = null;

return newHead;

}

}

**Output :**



**5. Merge k sorted lists**

**Aim :**

To merge k sorted linked lists into a single sorted linked list using an efficient algorithm.

**Objectives :**

* Understand the concept of merging multiple sorted lists.
* Implement an efficient approach using priority queues (min-heaps) or divide and conquer.
* Optimize the solution for minimal time complexity.
* Analyze space complexity when merging multiple lists.

**Implementation/Code :**

import java.util.PriorityQueue;

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode mergeKLists(ListNode[] lists) {

if (lists == null || lists.length == 0) {

return null; // Edge case: If the list array is empty, return null.

}

// Min-heap to store ListNode elements, ordered by value.

PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val - b.val);

// Add the first node of each non-empty linked list to the heap.

for (ListNode list : lists) {

if (list != null) {

minHeap.add(list);

}

}

// Dummy node to simplify the list building process.

ListNode dummy = new ListNode(-1);

ListNode tail = dummy;

// Extract the smallest element from the heap and add its next node back.

while (!minHeap.isEmpty()) {

ListNode minNode = minHeap.poll(); // Get the node with the smallest value.

tail.next = minNode; // Add it to the merged list.

tail = tail.next; // Move the tail forward.

if (minNode.next != null) {

minHeap.add(minNode.next); // Add the next node from the list to the heap.

}

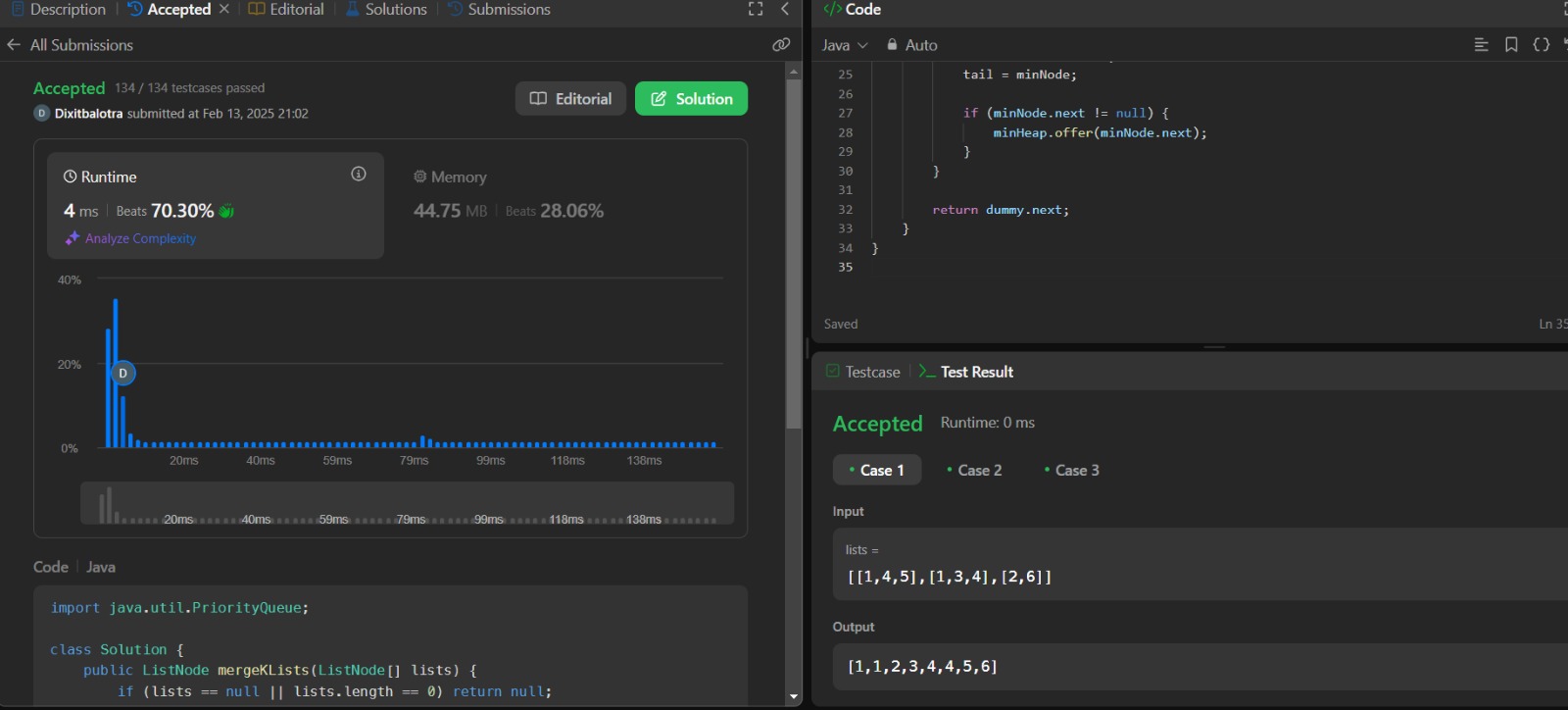
}

return dummy.next; // The merged sorted list.

    }

}

**Output :**



**6. Sort List HW**

**Aim :**

To implement an efficient sorting algorithm (Merge Sort or Quick Sort) to sort a given singly linked list.

**Objectives :**

* Understand sorting techniques applicable to linked lists.
* Implement Merge Sort or Quick Sort for sorting linked lists.
* Optimize the sorting algorithm to run in O(n log n) time complexity.
* Compare sorting approaches in terms of efficiency and space complexity.

**Implementation/Code :**

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode sortList(ListNode head) {

if (head == null || head.next == null) {

return head; // Base case: If the list is empty or has only one node, it's already sorted.

}

// Step 1: Split the list into two halves

ListNode mid = getMid(head);

ListNode left = head;

ListNode right = mid.next;

mid.next = null; // Break the list into two halves

// Step 2: Recursively sort each half

left = sortList(left);

right = sortList(right);

// Step 3: Merge the sorted halves

return merge(left, right);

}

// Helper function to find the middle of the linked list (Slow & Fast pointer technique)

private ListNode getMid(ListNode head) {

ListNode slow = head, fast = head.next;

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

slow = slow.next;

fast = fast.next.next;

}

return slow; // Midpoint of the list

}

// Helper function to merge two sorted linked lists (Merge operation of Merge Sort)

private ListNode merge(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(-1);

ListNode tail = dummy;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

tail.next = l1;

l1 = l1.next;

} else {

tail.next = l2;

l2 = l2.next;

}

tail = tail.next;

}

// Attach the remaining nodes from either list

if (l1 != null) tail.next = l1;

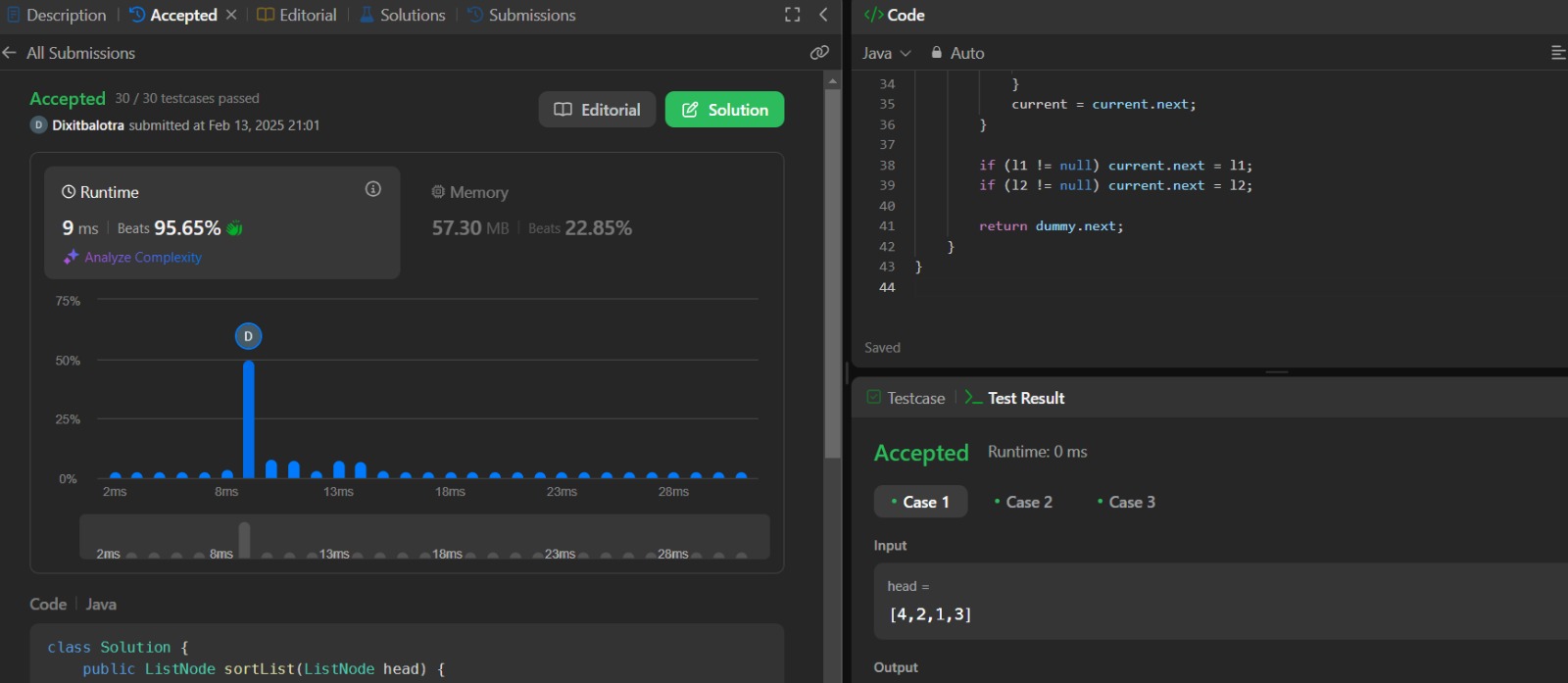
if (l2 != null) tail.next = l2;

return dummy.next; // The sorted merged list

    }

}

**Output :**



**Learning Outcomes :**

* Gain in-depth knowledge of singly linked lists, their structure, and traversal techniques.
* Differentiate between singly, doubly, and circular linked lists.
* Detect and handle cycles in a linked list using Floyd’s Cycle Detection Algorithm.
* Remove duplicate elements from a sorted linked list while maintaining order.
* Reverse a singly linked list using both iterative and recursive approaches.