



Experiment 3

Student Name: Aniket Gupta

Branch: BE-CSE

Semester: 6th

Subject Name: Advanced programming

Lab II

UID: 22BCS13824

Section/Group: IOT_606/B

Date of Performance: 16/02/25

Subject Code: 22CSP-351

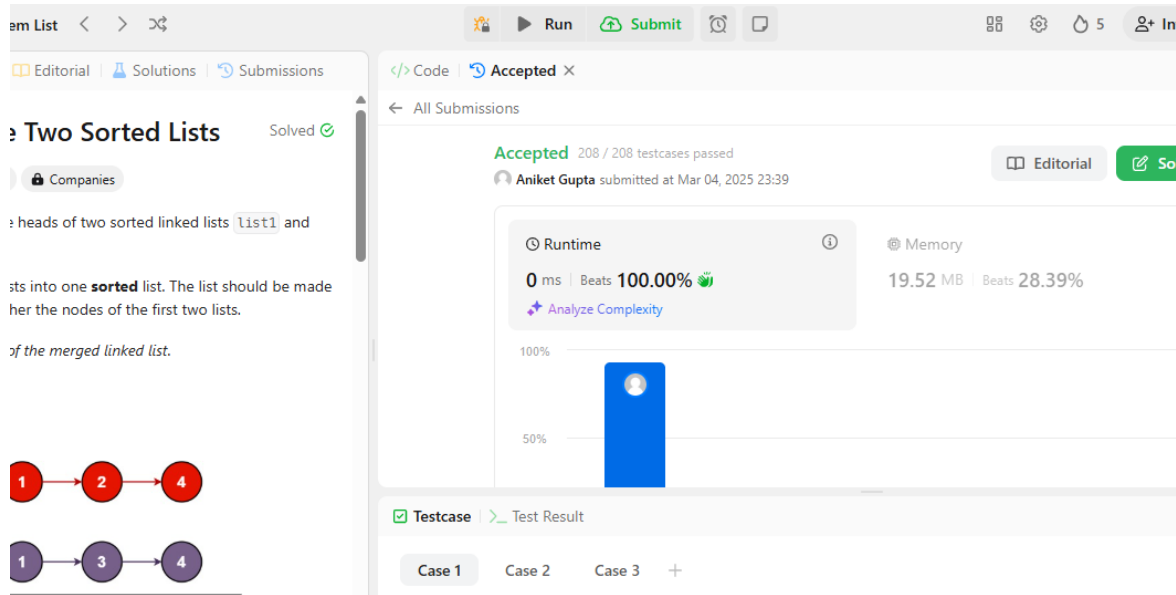
PROBLEM 1:

1. **Aim:** Merge Two Sorted Lists on LeetCode.
2. **Objective:** You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

3. Code:

```
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;  
        } else if (list2 != null) {  
            current.next = list2;  
        }  
  
        return dummy.next;  
    }  
}
```

4. Output:



The screenshot shows a coding interface for the problem "Merge Two Sorted Lists". On the left, the problem description is visible, including a diagram of two linked lists: List 1 (1 → 2 → 4) and List 2 (1 → 3 → 4). The right panel shows the submission status as "Accepted" with 208 / 208 testcases passed. The runtime is 0 ms (Beats 100.00%) and memory is 19.52 MB (Beats 28.39%). A "Testcase" tab is active at the bottom.

5. Time complexity: $O(m+n)$

Space Complexity: $O(1)$

PROBLEM 2:

1. **Aim:** Remove Duplicates from Sorted List II on LeetCode.
2. **Objective:** Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

3. Code:

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode(0, head);
        ListNode prev = dummy;

        while (head != null) {
            if (head.next != null && head.val == head.next.val) {
                while (head.next != null && head.val == head.next.val) {
                    head = head.next;
                }
                prev.next = head.next;
            }
            else {
                prev = head;
            }
        }
    }
}
```

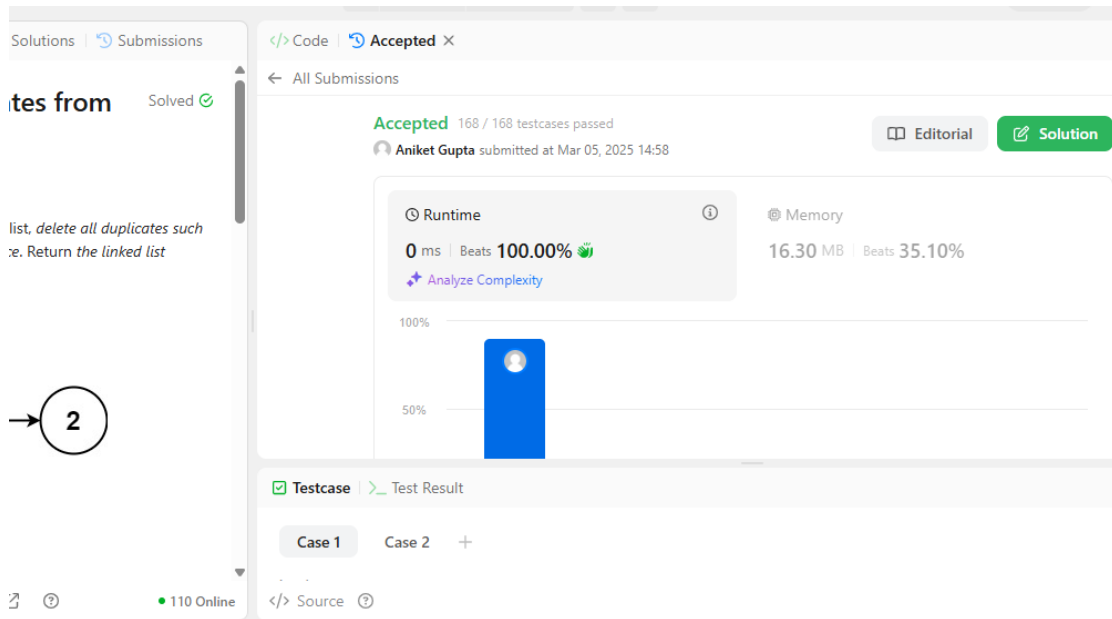
```

        prev = prev.next;
    }
    head = head.next;
}

return dummy.next;
}
}

```

4. Output:



Solutions | Submissions

Remove Duplicates from Sorted List II | Solved ✓

list, delete all duplicates such that no element appears more than once. Return the linked list.

→ 2

Accepted 168 / 168 testcases passed

Aniket Gupta submitted at Mar 05, 2025 14:58

Editorial Solution

Runtime: 0 ms | Beats 100.00% | Memory: 16.30 MB | Beats 35.10%

Analyze Complexity

100% 50%

Testcase | Test Result

Case 1 Case 2 +

110 Online

Source

5. Time Complexity: $O(n)$

Space Complexity: $O(1)$



PROBLEM 3:

1. **Aim:** LRU Cache on LeetCode.
2. **Objective:** Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

3. Code:

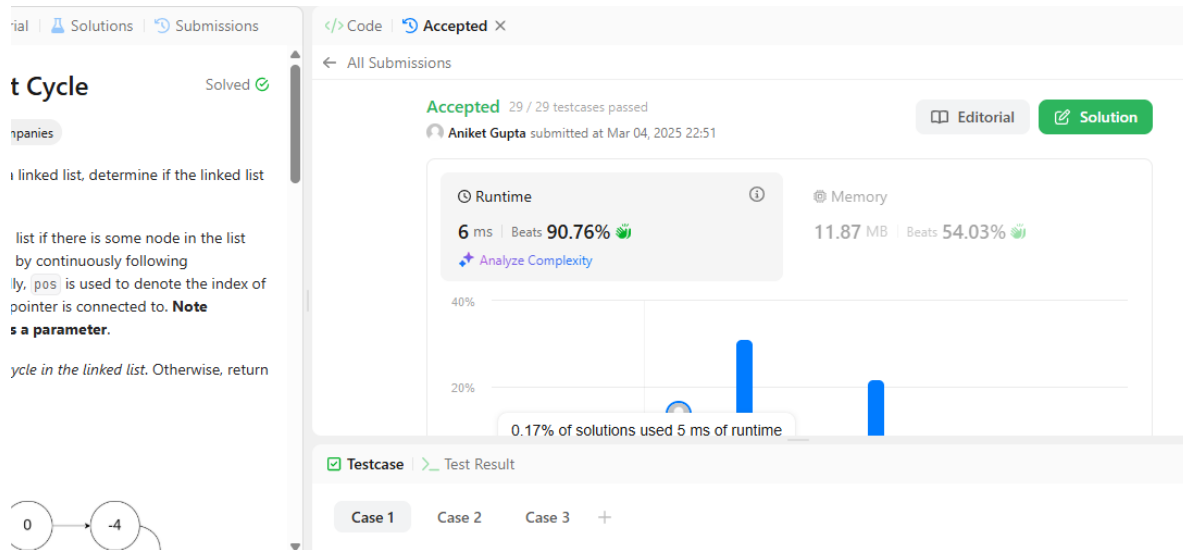
```
import java.util.LinkedHashMap;
import java.util.Map;
class LRUCache extends LinkedHashMap<Integer, Integer> {
    private int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75f, true);
        this.capacity = capacity;
    }
    public int get(int key) {
        return super.getOrDefault(key, -1);
    }

    public void put(int key, int value) {
        super.put(key, value);
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {
        return size() > capacity;
    }
}
```

4. Output:



5. Time Complexity: $O(1)$

Space Complexity: $O(\text{capacity})$

PROBLEM 4:

1. **Aim:** LFU Cache on LeetCode.
2. **Objective:** Design and implement a data structure for a Least Frequently Used (LFU) cache.

3. **Code:**

```
class LFUCache {  
  
    private Map<Integer, Pair<Integer, Integer>> cache;  
  
    private Map<Integer, LinkedHashSet<Integer>> frequencies;  
    private int minf;  
    private int capacity;  
  
    private void insert(int key, int frequency, int value) {  
        cache.put(key, new Pair<>(frequency, value));  
        frequencies.putIfAbsent(frequency, new LinkedHashSet<>());  
        frequencies.get(frequency).add(key);  
    }  
  
    public LFUCache(int capacity) {  
        cache = new HashMap<>();  

```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
frequencies = new HashMap<>();
minf = 0;
this.capacity = capacity;
}

public int get(int key) {
    Pair<Integer, Integer> frequencyAndValue = cache.get(key);
    if (frequencyAndValue == null) {
        return -1;
    }
    final int frequency = frequencyAndValue.getKey();
    final Set<Integer> keys = frequencies.get(frequency);
    keys.remove(key);
    if (keys.isEmpty()) {
        frequencies.remove(frequency);

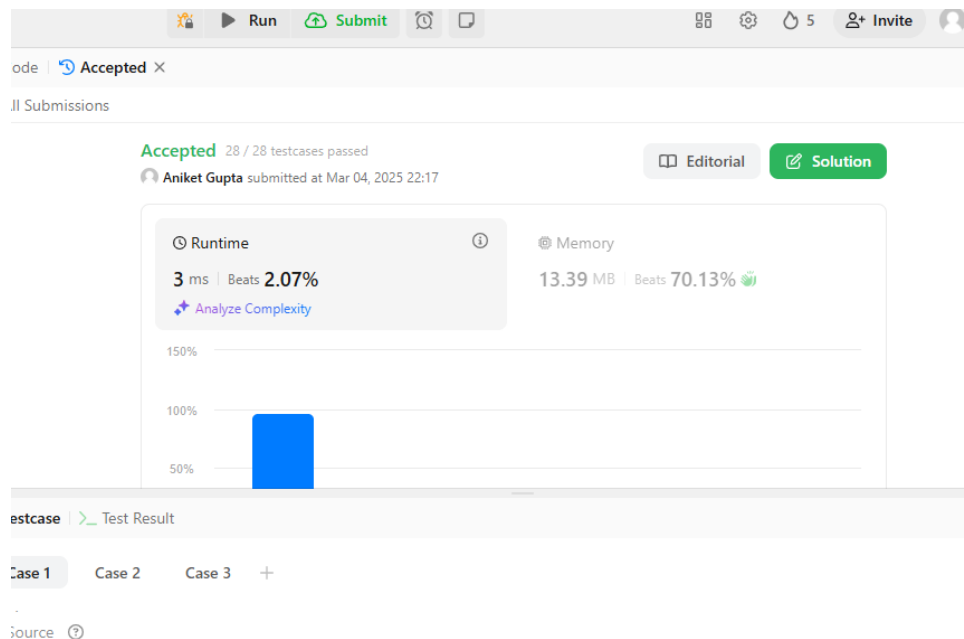
        if (minf == frequency) {
            ++minf;
        }
    }
    final int value = frequencyAndValue.getValue();
    insert(key, frequency + 1, value);
    return value;
}

public void put(int key, int value) {
    if (capacity <= 0) {
        return;
    }
    Pair<Integer, Integer> frequencyAndValue = cache.get(key);
    if (frequencyAndValue != null) {
        cache.put(key, new Pair<>(frequencyAndValue.getKey(), value));
        get(key);
        return;
    }
}
```



```
}  
if (capacity == cache.size()) {  
    final Set<Integer> keys = frequencies.get(minf);  
    final int keyToDelete = keys.iterator().next();  
    cache.remove(keyToDelete);  
    keys.remove(keyToDelete);  
    if (keys.isEmpty()) {  
        frequencies.remove(minf);  
    }  
}  
minf = 1;  
insert(key, 1, value);  
}  
}
```

4. Output:



5. Time Complexity: $O(1)$

Space Complexity: $O(\text{capacity})$

6. Learning Outcome:

- Understanding of Linked Lists: Gained deeper insights into linked list operations, including traversal, insertion, and deletion.
- Two-Pointer Technique: Used two-pointer approach effectively for merging lists and removing duplicates.



- c. Cache Mechanisms & Policies: Understood LRU (Least Recently Used) and LFU (Least Frequently Used) caching strategies.
- d. HashMap + Doubly Linked List Approach: Explored how to maintain $O(1)$ operations for LRU using LinkedHashMap and frequency-based eviction in LFU.
- e. Efficiency Considerations: Realized the importance of balancing time complexity ($O(1)$ for `get()` and `put()`) with space constraints.
- f. Eviction Strategies: Implemented LRU's eviction of least recently used items and LFU's removal of least frequently accessed elements.
- g. Java Data Structures: Learned how LinkedHashMap provides built-in LRU functionality and used HashMap with LinkedHashSet for LFU.