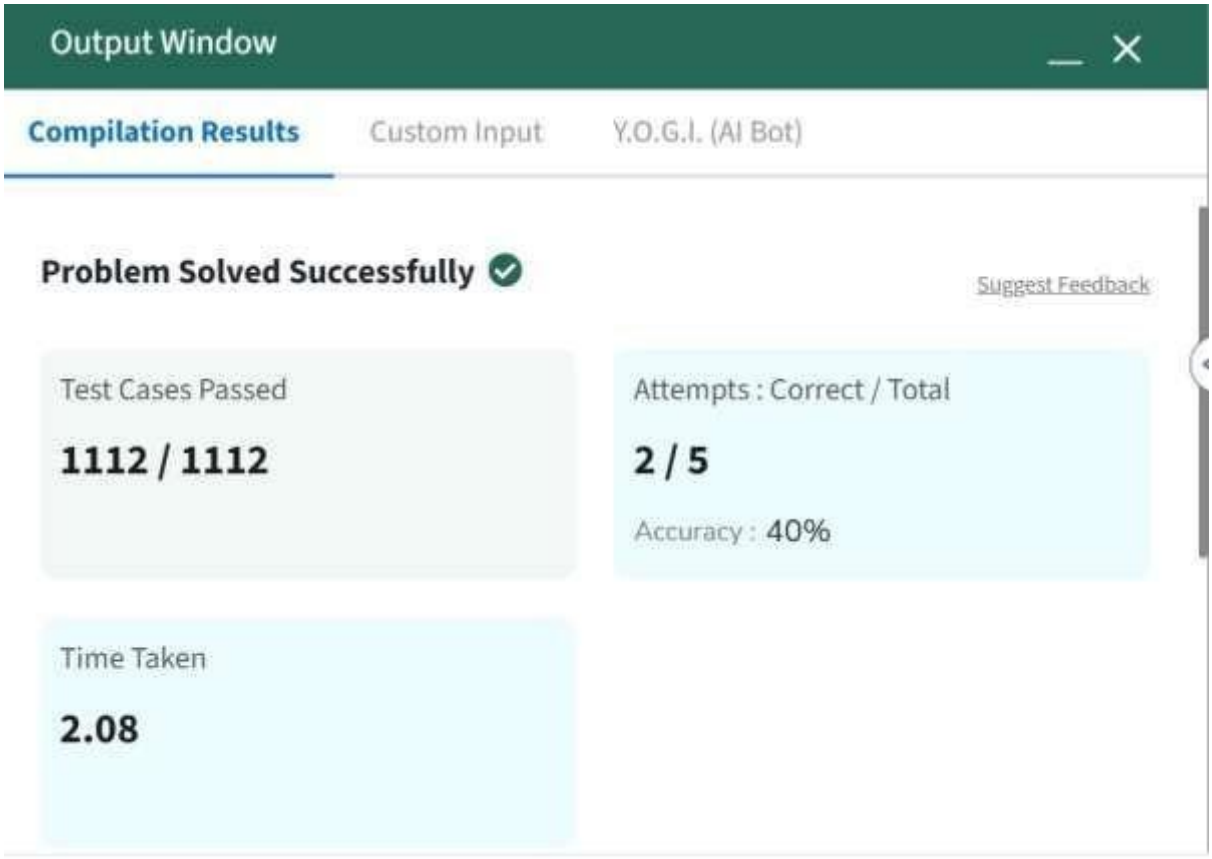
AP Assignment - 03 Name : Abhishek Kumar UID : 22BCS16396 Section : 608-B

Q1-Given a linked list. Print all the elements of the linked list separated by space followed.

CODE-

```
class Solution {     void
printList(Node head) {
Node temp = head;        while
(temp != null) {
          System.out.print(temp.data);
if (temp.next != null)
          {
             System.out.print(" ");
          }
          temp = temp.next;
       }
    }
}
```

Solution:-



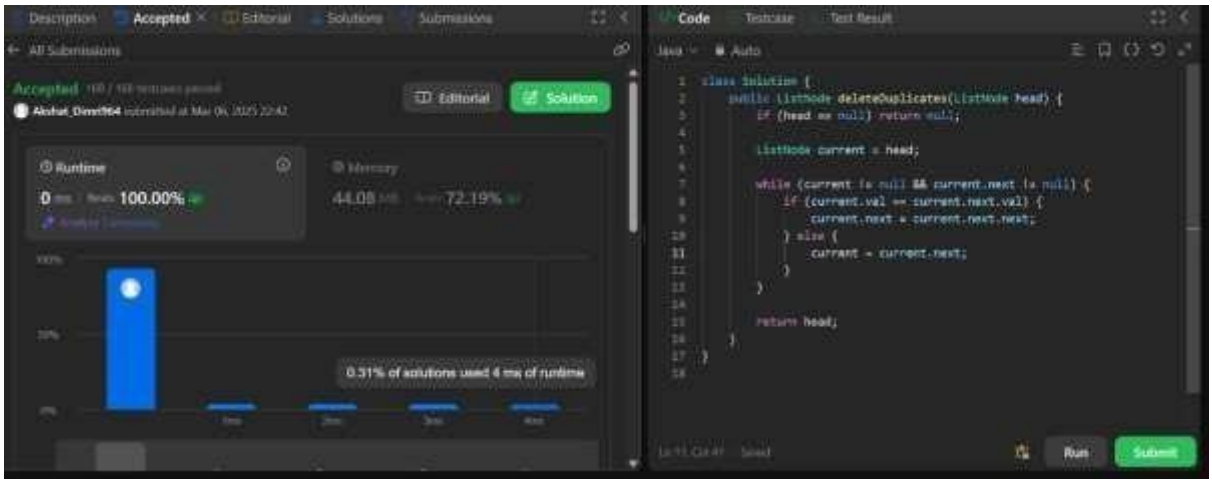
Q2-Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list **sorted** as well*.

Code:-

```
class Solution {     public ListNode deleteDuplicates(ListNode
head) {
        if (head == null) return null;

        ListNode current = head;

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

```
        }
      }

    return head;
  }
}
```

Solution:-



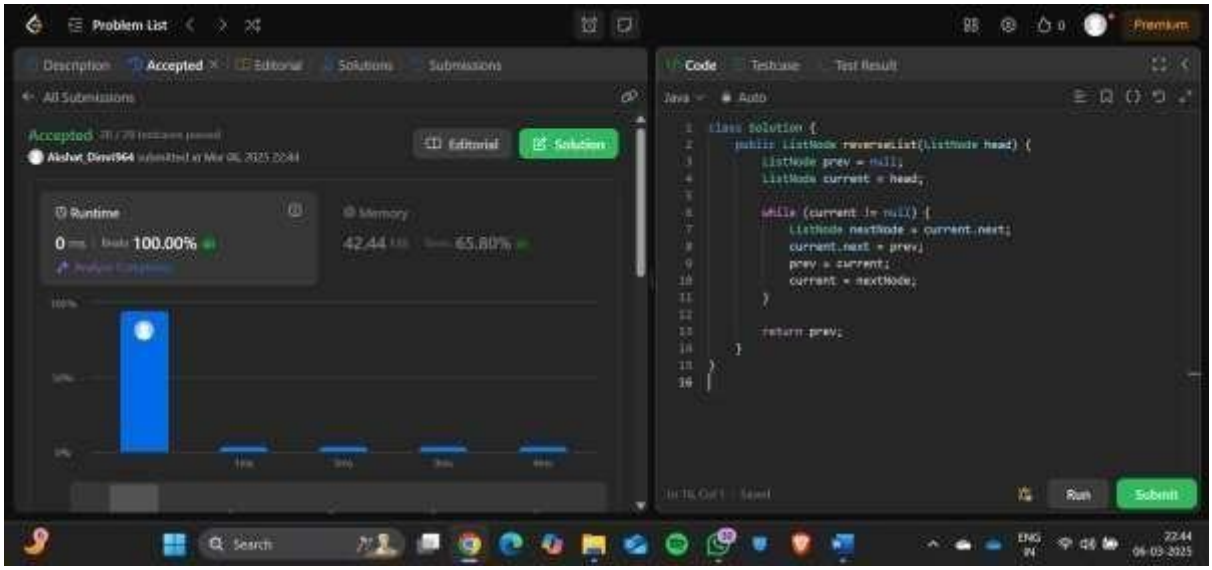
Q3-Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Code:-

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

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

```
    }

    return prev;
  }
}
```

Solution:-



Q4-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 == null
|| head.next == null) {

            return null;

        }
```

ListNode A = head, B = head, prev = null; while

(B != null && B.next != null) {

prev = A;

A = A.next;

B = B.next.next;

}

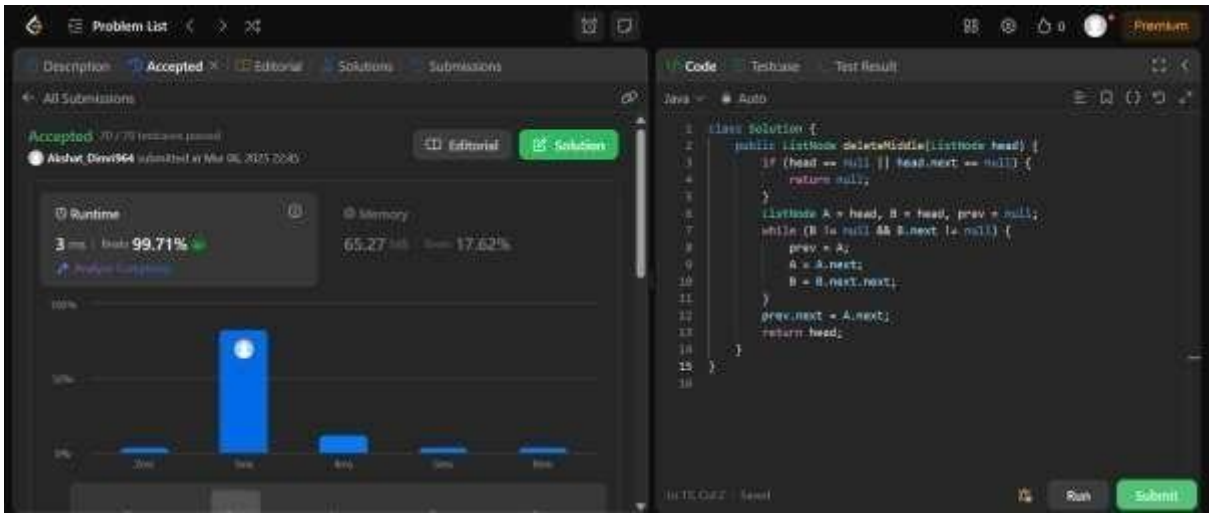
prev.next = A.next;

return head;

}

}

Solution:-



Q5-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*.

Code:-

class Solution { public ListNode mergeTwoLists(ListNode list1,

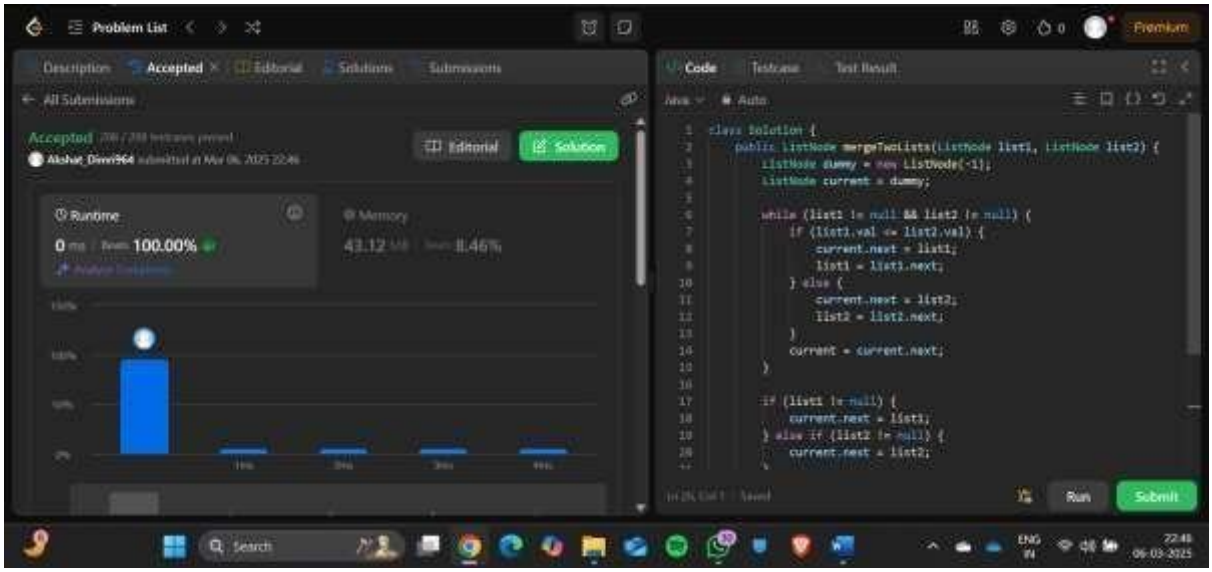
ListNode list2) {

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;
    }
}
```

Solution:-



Q6-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**.

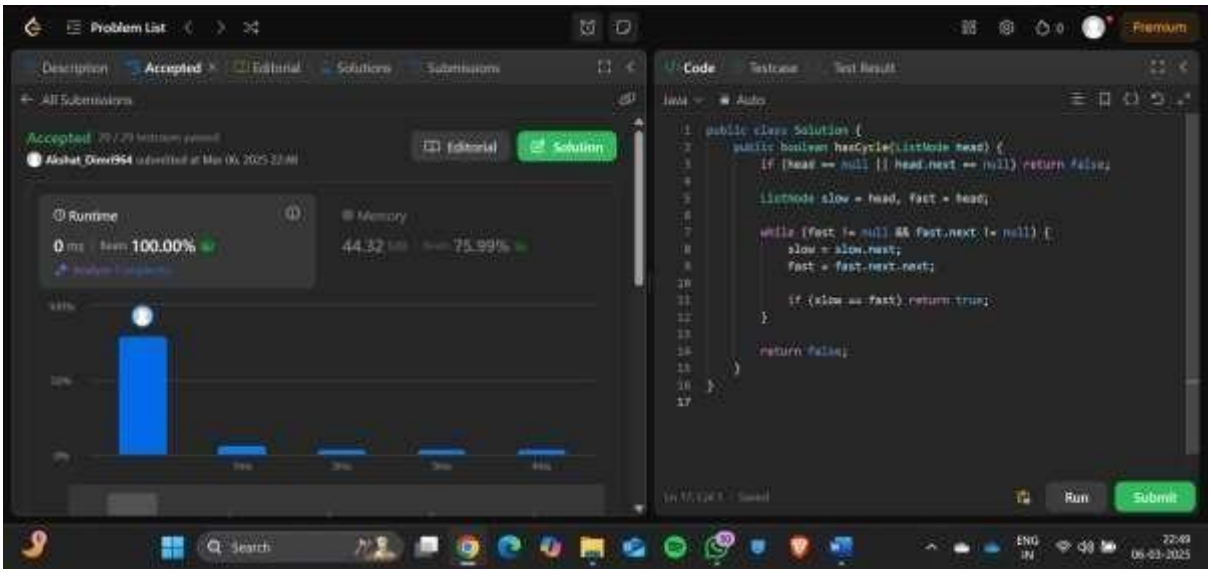
Return true *if there is a cycle in the linked list*. Otherwise, return false.

Code:-

```
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;
    }
```

}

Solution:-



Q7-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 == null || head.next == null || k == 0) return head;

int length = 1;

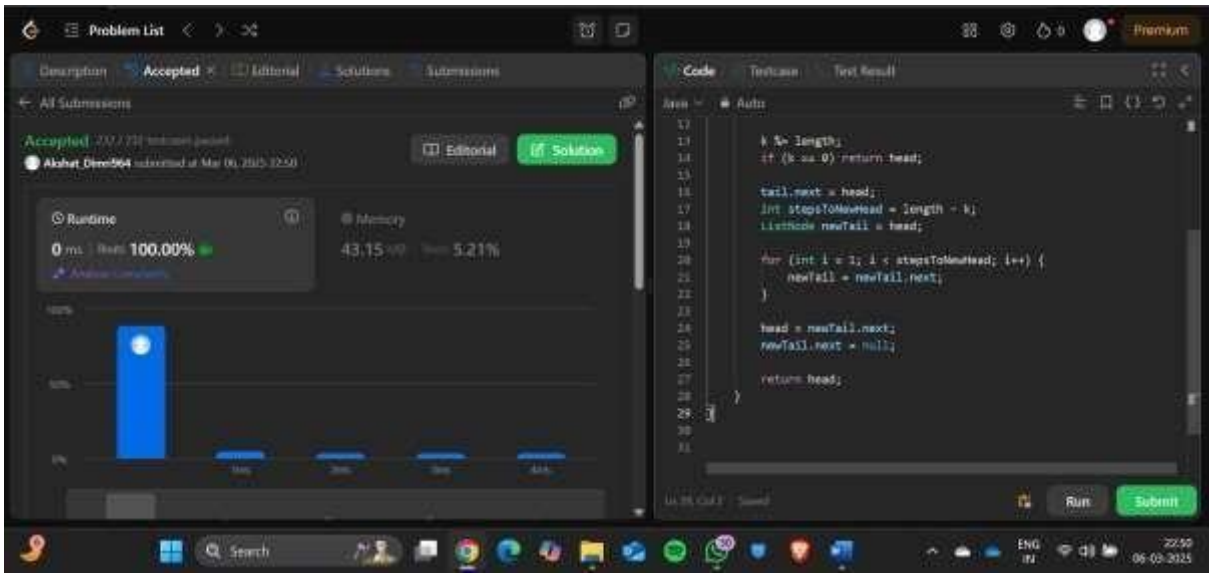
ListNode tail = head;

while (tail.next != null) { tail = tail.next; length++;

}

```
        k %= length;        if (k
== 0) return head;


        tail.next = head;       int
stepsToNewHead = length - k;
        ListNode newTail = head;


        for (int i = 1; i < stepsToNewHead; i++) {
newTail = newTail.next;
        }


        head = newTail.next;
newTail.next = null;


        return head;
    }
}
```

Solution:-



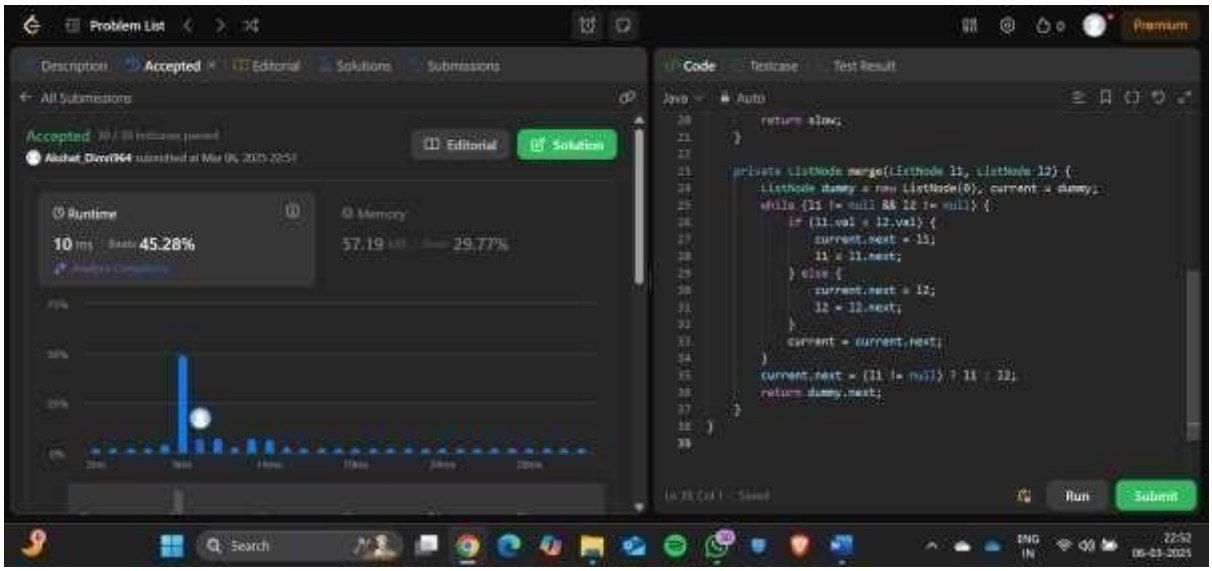
Q8-Given the head of a linked list, return *the list after sorting it in* ***ascending order***.

Code:-

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

        ListNode mid = getMid(head);
        ListNode left = sortList(head);
        ListNode right = sortList(mid);

        return merge(left, right);
    }

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

    private ListNode merge(ListNode l1, ListNode l2) {         ListNode
dummy = new ListNode(0), 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;
        }
        current.next = (l1 != null) ? l1 : l2;
return dummy.next;
    }
}
```

Solution:-



Q9 - 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.

Code:-

```
public class Solution {     public ListNode mergeKLists(ListNode[]
lists) {          PriorityQueue<ListNode> minHeap = new
PriorityQueue<>(Comparator.comparingInt(node -> node.val));
for (ListNode node : lists) {             if (node != null) {
minHeap.add(node);
            }
```

```
        }

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

        while (!minHeap.isEmpty()) {
ListNode node = minHeap.poll();
current.next = node;
current = current.next;

            if (node.next != null) {
minHeap.add(node.next);
            }
        }

        return dummy.next;
    }
}
```

Solution:-