Experiment 3

Student Name: Arshdeep Singh **UID: 22BET10026**
Branch: BE-IT **Section/Group: 22BET_IOT_701/A**
Semester: 6th **Date of Performance: 5-02-25**
Subject Name: Advanced Programming Lab-2 **Subject Code: 22ITP-351**

Problem 1. Given a linked list. Print all the elements of the linked list separated by space followed.

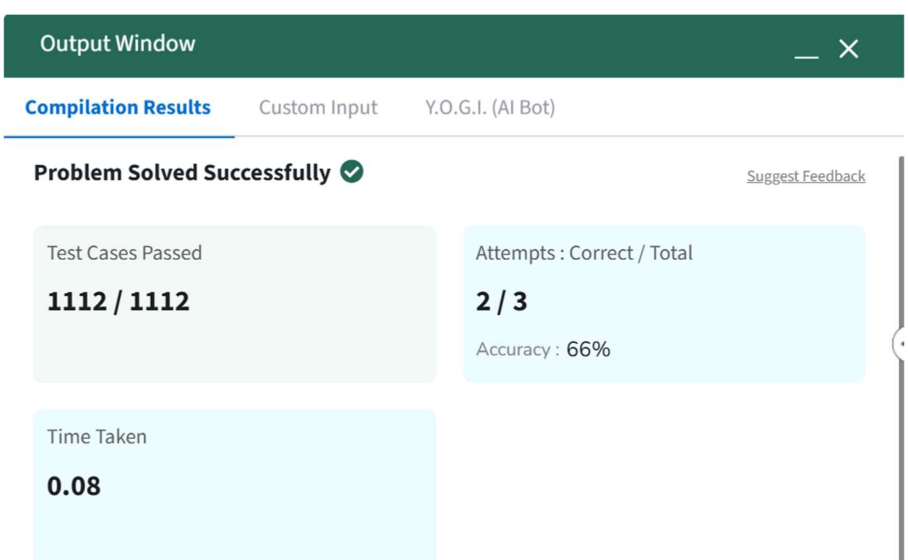
Algorithm:

1. Initialize a pointer temp and set it to head.
2. Loop while temp is not nullptr:
   - Print the data of the current node.
   - Move temp to the next node.
3. End loop when temp reaches nullptr.
4. Exit function (No return value since it prints to the console).

Code:

```
class Solution {
 public:

   void printList(Node *head) {
   Node* temp = head;
   while (temp != nullptr) {
       cout << temp->data << " ";
       temp = temp->next;
   }
   }
};
```

Output:



Compilation Completed

For Input:

1 2

Your Output:

1 2

Expected Output:

1 2

Problem 2. 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*.

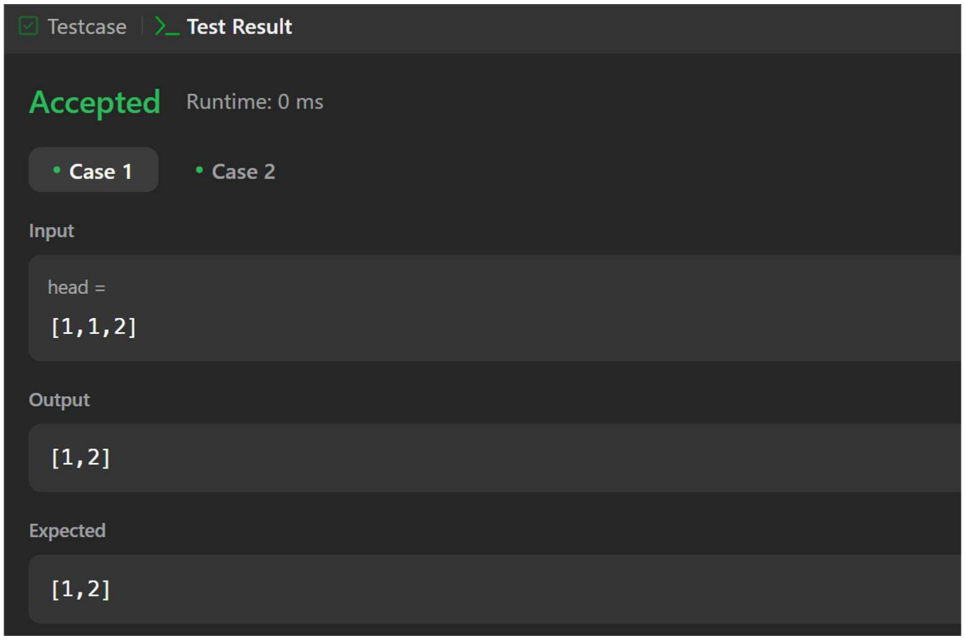
Algorithm:

1. **Initialize** a pointer current and set it to head.
2. **Traverse** the linked list while current and current->next exist:
   - If current->val == current->next->val:
     - Store the duplicate node in a temporary pointer temp.
     - Update current->next to current->next->next (skipping the duplicate node).
     - Delete temp to free memory.
   - Else, move current to current->next (proceed to the next distinct node).
3. **Return** the modified head pointer after processing.

Code:

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

        while (current && current->next) {
            if (current->val == current->next->val) {

                ListNode* temp = current->next;
                current->next = current->next->next;
                delete temp;
            } else {
                current = current->next;
            }
        }

        return head;
    }
```

Output:



Problem 3. Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Algorithm:

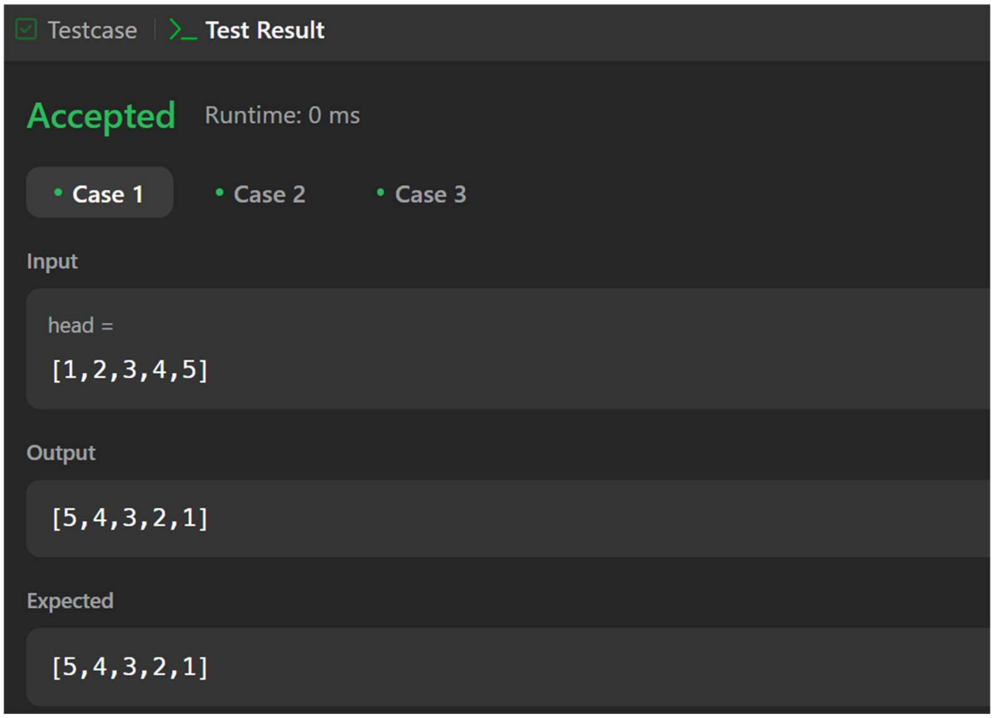
1. **Initialize:**
   - prev = nullptr (This will hold the new head of the reversed list).
   - current = head (Starting point for traversal).
2. **Iterate through the list** while current is not nullptr:
   - Store current->next in a temporary variable nextNode to preserve the next node.
   - Reverse the link by setting current->next = prev.
   - Move prev forward to current.
   - Move current forward to nextNode.
3. **Return prev**, which now holds the new head of the reversed list.

Code:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* current = head;

        while (current != nullptr) {
            ListNode* nextNode = current->next;
            current->next = prev;
            prev = current;
            current = nextNode;
        }

        return prev;
```

```
    }
};
```

Output:



Problem 4. You are given the head of a linked list. Delete the middle node, and return *the* head *of the modified linked list*.

Algorithm:

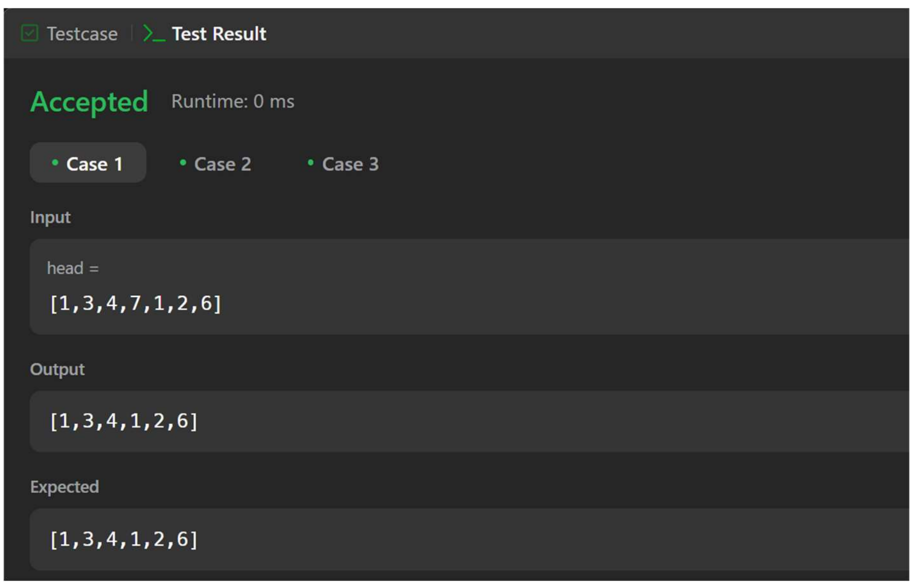
1. **Handle edge case:**
   - If head == nullptr (empty list) or head->next == nullptr (only one node), return nullptr because the middle node is the only node.
2. **Use the slow and fast pointer approach:**
   - Initialize slow = head, fast = head, and prev = nullptr (to keep track of the node before slow).
   - Move slow one step and fast two steps at a time.
   - When fast reaches the end (nullptr or nullptr->next), slow will be at the middle.
3. **Remove the middle node:**
   - Update prev->next = slow->next, skipping the middle node.
   - Delete the middle node to free memory.
4. **Return head**, as the modified list.

Code:

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return nullptr;
        }
```

```
            ListNode* slow = head;
            ListNode* fast = head;
            ListNode* prev = nullptr;

            while (fast != nullptr && fast->next != nullptr) {
                prev = slow;
                slow = slow->next;
                fast = fast->next->next;
            }


            prev->next = slow->next;
            delete slow;

            return head;
        }
    };
```

Output:



Problem 5. You are given the heads of two sorted linked lists list1 and list2.

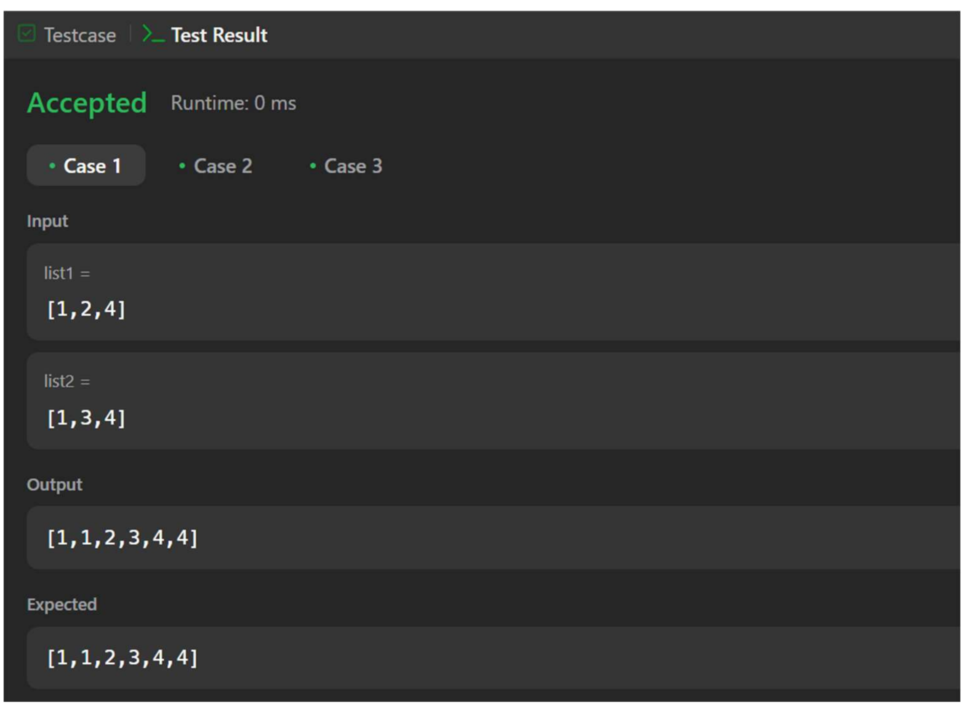
Algorithm:

1. **Create a dummy node** to simplify merging (dummy(0)) and initialize a tail pointer pointing to it.
2. **Traverse both lists** while list1 and list2 are not nullptr:
   - Compare list1->val and list2->val:
     - If list1->val <= list2->val, attach list1 to tail and move list1 forward.
     - Else, attach list2 to tail and move list2 forward.
   - Move tail forward.
3. **Append the remaining nodes** from either list1 or list2 if one list is exhausted.
4. **Return dummy.next**, the merged list starting after the dummy node.

Code:

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode dummy(0);
        ListNode* tail = &dummy;


        while (list1 != nullptr && list2 != nullptr) {
            if (list1->val <= list2->val) {
                tail->next = list1;
                list1 = list1->next;
            } else {
                tail->next = list2;
                list2 = list2->next;
            }
            tail = tail->next;
        }


        tail->next = (list1 != nullptr) ? list1 : list2;

        return dummy.next;
    }
};
```

Output:



Problem 6. 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*.

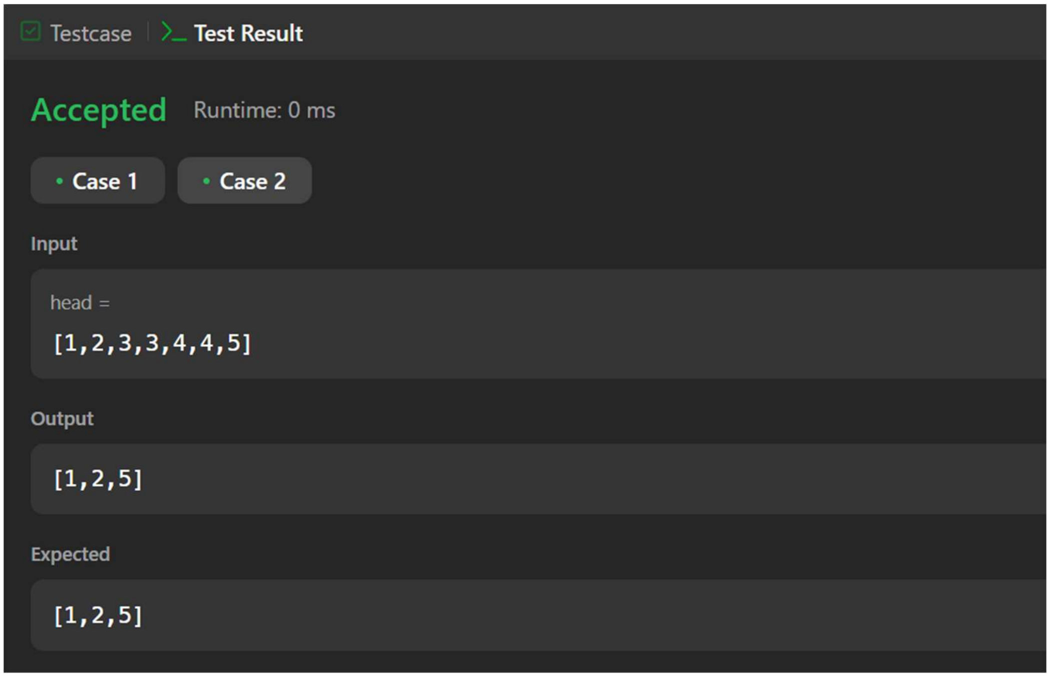
Algorithm:

1. Create a dummy node pointing to head. This helps handle edge cases where the first node gets deleted.
2. Initialize prev pointer to dummy, which will track the last unique node.
3. Traverse the list while head is not nullptr:
   - If head->val == head->next->val, move head forward until all duplicates are skipped.
   - Update prev->next to head->next to remove duplicates.
   - Otherwise, move prev forward.
4. Return dummy->next, as the modified linked list.

Code:

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

        while (head != nullptr) {

            if (head->next != nullptr && head->val == head->next->val) {

                while (head->next != nullptr && head->val == head->next->val) {
                    head = head->next;
                }

                prev->next = head->next;
            } else {
                prev = prev->next;
            }
            head = head->next;
        }

        return dummy->next;
    }
};
```

Output:



Problem 7. Given head, the head of a linked list, determine if the linked list has a cycle in it.

Algorithm:

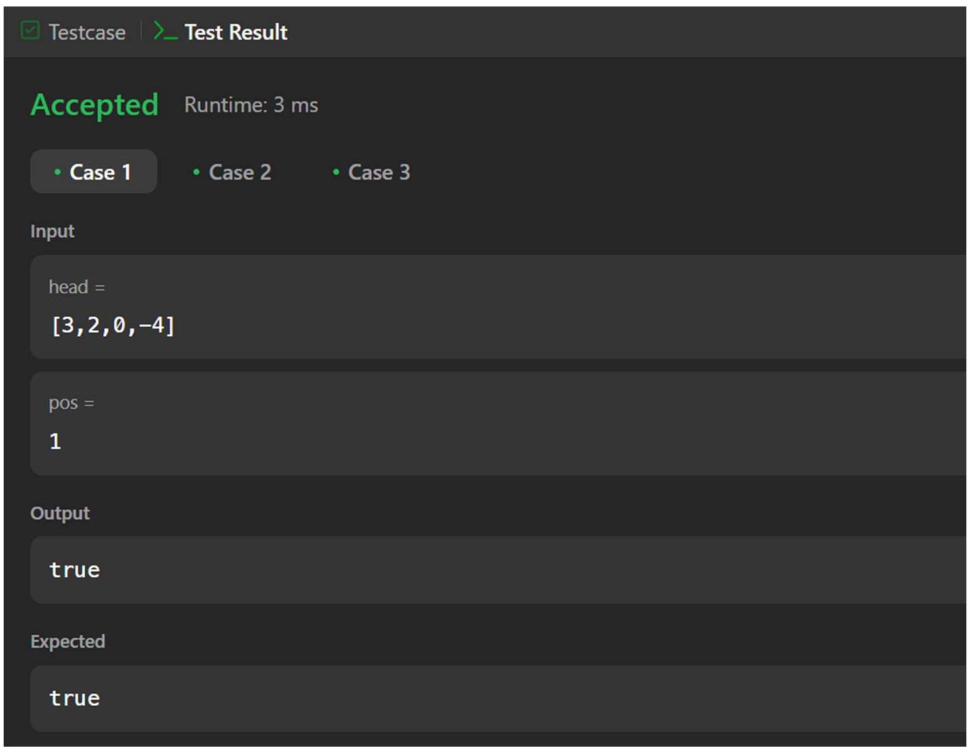
1. **Initialize two pointers:**
   - slow moves **one step** at a time.
   - fast moves **two steps** at a time.
2. **Traverse the linked list** while fast and fast->next are not nullptr:
   - Move slow forward by one step.
   - Move fast forward by two steps.
   - If slow == fast, a cycle is detected → **Return true**.
3. If the loop exits (fast reaches nullptr), return **false** (no cycle).

Code:

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;

        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
```

```
        if (slow == fast) {
                return true;
            }
        }

        return false;
    }
};
```

Output:



Problem 8. 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*.

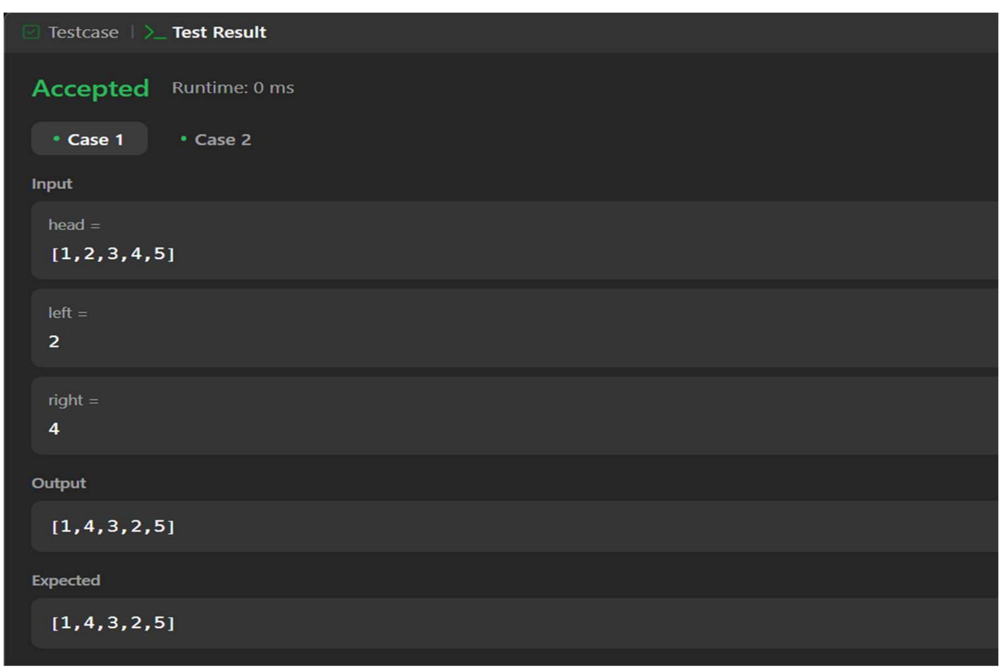
Algorithm:

1. Edge case: If the list is empty or left == right, no need to reverse anything, so return the list as is.
2. Create a dummy node and set it to point to head. This simplifies handling the edge case when the sublist starts at the head.
3. Move the prev pointer to the node just before the left-th position.
4. Reverse the sublist:
   - Start from the left-th position, iteratively reverse the nodes between left and right.
   - For each iteration, adjust pointers so that the nodes between left and right are reversed.
5. Return the modified list starting from dummy->next.

Code:

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        if (head == nullptr || 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 = nullptr;


        for (int i = 0; i < right - left; ++i) {
            next = curr->next;
            curr->next = next->next;
            next->next = prev->next;
            prev->next = next;
        }

        return dummy->next;
    }
};
```

Output:



Problem 9. Given the head of a linked list, rotate the list to the right by k places.

Algorithm:

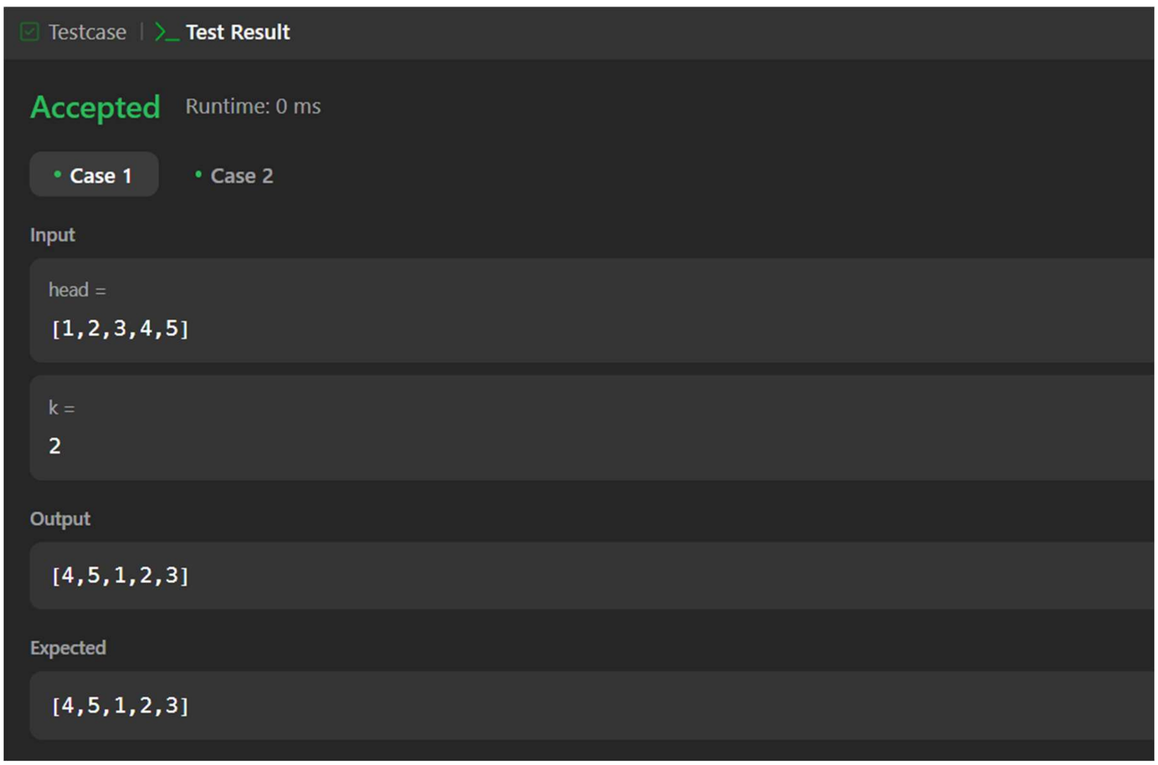
1. Edge case check:
   - If head is nullptr, the list is empty, so return head.
   - If the list has only one node or k == 0, return head as no rotation is needed.
2. Find the length of the list:
   - Traverse the list to calculate its length and find the last node.
3. Form a circular list:
   - Connect the last node's next pointer to the head to form a circular linked list.
4. Calculate the effective k:
   - Since rotating the list by k positions is the same as rotating it by k % length, compute the effective number of rotations.
5. Find the new head:
   - Traverse the list to find the new tail, which is at position length - k - 1, and set the new head to newTail->next.
6. Break the circular list:
   - Set newTail->next = nullptr to disconnect the circular connection and finalize the list rotation.
6. Return the new head.

Code:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (head == nullptr || head->next == nullptr || k == 0) {
            return head;
        }


        ListNode* current = head;
        int length = 1;
        while (current->next != nullptr) {
            current = current->next;
            length++;
        }

        current->next = head;

        k = k % length;
        if (k == 0) {
            current->next = nullptr;
            return head;
        }


        ListNode* newTail = head;
        for (int i = 1; i < length - k; i++) {
```

```
            newTail = newTail->next;
        }


        ListNode* newHead = newTail->next;
        newTail->next = nullptr;

        return newHead;
    }
};
```

Output:



Problem 10. You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Algorithm:

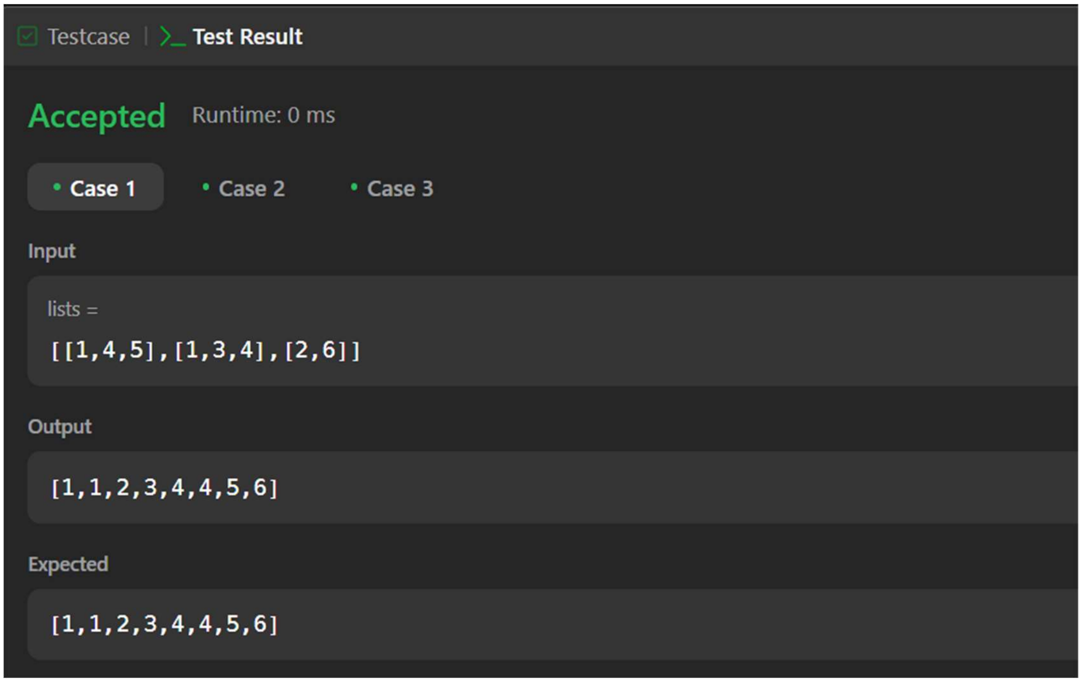
1. Edge case check:
   - If the input lists is empty, return nullptr.
2. Min-Heap Initialization:
   - Use a min-heap (priority queue) to efficiently get the smallest node.
   - Define a custom comparator to compare nodes based on their values (a->val > b->val).
3. Push heads into the min-heap:
   - Loop through each linked list and push the head node of each list into the min-heap.
4. Merge the lists:
   - Create a dummy node to simplify the merging process.
   - Pop the minimum node from the heap and add it to the merged list.
   - If the popped node has a next node, push the next node from the same list into the heap.

5. Return the merged list:
   - Skip the dummy node and return the merged list starting from dummy->next.

Code:

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return nullptr;

        auto compare = [](ListNode* a, ListNode* b) {
            return a->val > b->val;
        };

        priority_queue<ListNode*, vector<ListNode*>, decltype(compare)> minHeap(compare);


        for (ListNode* list : lists) {
            if (list) {
                minHeap.push(list);
            }
        }

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


        while (!minHeap.empty()) {
            ListNode* node = minHeap.top();
            minHeap.pop();
            current->next = node;
            current = current->next;
            if (node->next) {
                minHeap.push(node->next);
            }
        }

        return dummy->next;
    }
};
```

Output:


Testcase
Test Result
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]