25. Reverse Nodes in k-Group

Hard Recursion Linked List

Problem Description

In this problem, we are given the head of a linked list and an integer `k`. The task is to reverse every consecutive `k` nodes in the linked list. If the number of nodes is not a multiple of `k`, then the remaining nodes at the end of the list should stay in the same order. It is important to note that we can only change the links between nodes, not the node values themselves. This process is similar to reversing the entire linked list, but it's done in chunks of `k` elements at a time.

Intuition

The key to solving this problem lies in breaking it down into smaller, more manageable tasks. Here's how we can think about it:

1. **Divide the list into segments:** We treat the list as a sequence of segments each with `k` nodes, except possibly the last segment which may have less than `k` nodes if the list's length is not a multiple of `k`.
2. **Reverse individual segments:** We reverse each segment of `k` nodes while ensuring to maintain the connection with the rest of the list. This means detaching the segment, reversing it, and then reconnecting it with the main list.
3. **Re-connect segments:** Once a segment is reversed, it is necessary to attach the tail of the segment (which was originally the head) to the next part of the list which may be the head of the next segment to be reversed or the remaining part of the list.
4. **Handle the end case:** When we reach the end of the list and there are fewer than `k` nodes left, we simply retain their order and attach that segment as it is to the previously processed part of the list.

The intuition behind the solution approach comes from recognizing that this problem is a modification of a standard linked list reversal algorithm. In a typical linked list reversal, we reverse the links between nodes until we reach the end of the list. In this case, we apply the same principle but in a more controlled manner where the reversal stops after `k` nodes and we take extra care to attach the reversed segment back to the main list.

Solution Approach

The solution approach can be broken down into the following steps:

1. **Set up a dummy node:** We start by creating a dummy node, which serves as a preceding node to the head of the linked list. This allows us to easily manipulate the head of the list (which may change several times as we reverse segments) without worrying about losing the reference to the start of the list.
2. **Initialize pointers:** We set up two pointers, `pre` and `cur`, that initially point to the dummy node. The `pre` pointer will be used to keep track of the node at the beginning of the segment we're about to reverse, and `cur` will be used to traverse `k` nodes ahead to find the end of the segment.
3. **Find segments to reverse:** We move the `cur` pointer `k` nodes ahead to confirm that we have a full segment to reverse. If we reach the end of the list before moving `k` steps, we know that we're on the last segment, and thus we simply return the list without modifying this last part.
4. **Reverse the segment:** Once a full segment of `k` nodes has been confirmed by the `cur` pointer, we temporarily detach this segment from the rest of the list. We call the `reverseList` helper function, which reverses the segment using the classic three-pointer approach for reversing a linked list (`prev`, `curr`, `next`).
5. **Reconnect segments:** After reversing the segment, we need to reconnect it with the main list. This involves setting the `next` pointer of the last node of the reversed segment (previously the first) to point to `cur`, which is the node following the `cur` node (the end of the segment being reversed). The `next` pointer of `pre` (which was the tail of the previous segment) is then updated to point to the `prev.next = reverseList(start)` which returns the new head of the reversed segment.
6. **Preparation for next iteration:** Finally, `pre` is moved to the end of the reversed segment (which was its starting node), and `cur` is reset to `pre`. This ensures that we are ready to process the next segment.

The pattern used in this solution could be described as a modified two-pointer technique where one pointer (`cur`) is used to identify the segment to be reversed, and another pointer (`pre`) is used to manage connections between reversed segments. The algorithm efficiently manipulates references within the list, never requiring more than a constant amount of additional space, which leads to its space complexity of $O(1)$.

The reversal of each segment is a classic linked list reversal process that is applied repeatedly to each segment determined by our two-pointer setup. The time complexity of the overall algorithm is $O(n)$ because each node is looked at a constant number of times (once when identifying the segment and once when actually reversing it).

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have a linked list and `k = 2`:

```
Original List: 1 -> 2 -> 3 -> 4 -> 5

We are expected to reverse every two nodes:
Desired Outcome: 2 -> 1 -> 4 -> 3 -> 5
```

1. **Set up a dummy node:**
   - We initiate the dummy node and link it to the head of the list: `dummy -> 1 -> 2 -> 3 -> 4 -> 5`.
2. **Initialize pointers:**
   - `pre` points to `dummy`, and `cur` also points to `dummy`.
3. **Find segments to reverse:**
   - We move `cur` two steps, after which `cur` points to the node `2`.
   - We verify that there is a full segment of `k` nodes (`1` and `2`) to reverse.
4. **Reverse the segment:**
   - We detach the segment `1 -> 2`, and we reverse it, such that it becomes `2 -> 1`.
   - The rest of the list is temporarily disconnected, so we have `dummy -> 2 -> 1` and then `3 -> 4 -> 5`.
5. **Reconnect segments:**
   - We connect the node `1` (which is the tail of the reversed segment) to node `3` (which is the next node on the list).
   - Now the list is `dummy -> 2 -> 1 -> 3 -> 4 -> 5`.
6. **Preparation for the next iteration:**
   - We move the `pre` pointer to the end of the reversed segment (node `1`), and reset `cur` to `pre`.
   - The list remains as `dummy -> 2 -> 1 -> 3 -> 4 -> 5`.

We then repeat steps 3 to 6 for the next segment:

- By moving `cur` pointer two steps, we have `cur` at node `4`, confirming a full segment (`3` and `4`).
- We then reverse the `3 -> 4` segment to get `4 -> 3`.
- The list is reconnected to become `dummy -> 2 -> 1 -> 4 -> 3 -> 5`.
- Once we prepare for the next iteration, there is only one node left (node `3`), which doesn't form a full segment of `k=2`, so it remains as it is.
- **Final Reconnecting:**
  - The list is now fully processed and connected: `dummy -> 2 -> 1 -> 4 -> 3 -> 5`.

Since the dummy node was used solely for pointer manipulation, the final list's head is the node following the dummy, which is `2`. Therefore, the final output of the list after the algorithm is applied is `2 -> 1 -> 4 -> 3 -> 5`, which matches our desired outcome.

Python Solution

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        # Helper function to reverse a linked list
        def reverse_list(node):
            prev, current = None, node
            while current:
                next_temp = current.next
                current.next = prev
                prev = current
                current = next_temp
            return prev

        # The dummy node is used to handle edge cases smoothly
        dummy = ListNode()
        dummy.next = head
        pre_group = dummy

        while head:
            # Check if there is a full group to reverse
            count = 0
            group = head
            while count < k and current:
                current = current.next
                count += 1

            # If we have k nodes, proceed to reverse
            if count == k:
                # Detach the k group and reverse it
                k_group_end = head
                for _ in range(k - 1):  # Move to the end of the k group
                    k_group_end = k_group_end.next
                next_group = k_group_end.next
                k_group_end.next = None
                new_group_head = reverse_list(head)

                # Connect the reversed group back to the list
                pre_group.next = new_group_head
                head.next = next_group

                # Move pre_group and head for the next round of reversal
                pre_group = head
                head = next_group
            else:
                # Not enough nodes to fill a k group, so we are done
                break

        # Return the new head of the modified list
        return dummy.next
```

Java Solution

```java
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        // A dummy node with 0 as value and pointing to the head of the list
        ListNode dummyHead = new ListNode(0, head);
        ListNode predecessor = dummyHead, current = dummyHead;

        // Iterate through the list
        while (current.next != null) {
            // Check if there are k nodes to reverse
            for (int i = 0; i < k && current != null; i++) {
                current = current.next;
            }
            // If less than k nodes remain, no more reversing is needed
            if (current == null) {
                return dummyHead.next;
            }

            // Temporarily store the next segment to be addressed after reversal
            ListNode tmp = current.next;
            // Detach the k nodes from the rest of the list
            current.next = null;
            // 'start' will be the new tail after the reversal
            ListNode start = reverseList(predecessor.next);
            // Reverse the k nodes
            predecessor.next = reverseList(start);
            // Reconnect the tail to the next segment stored before
            start.next = tmp;
            // Move the predecessor and current pointers k nodes ahead
            predecessor = start;
            current = predecessor;
        }

        return dummyHead.next;
    }

    /**
     * Helper method to reverse the linked list.
     *
     * @param head The head of the list to be reversed.
     * @return The new head of the reversed list.
     */
    private ListNode reverseList(ListNode head) {
        ListNode previous = null, currentNode = head;

        // Traverse the list and reverse the links
        while (currentNode != null) {
            ListNode nextNode = currentNode.next;
            currentNode.next = previous;
            previous = currentNode;
            currentNode = nextNode;
        }
        // Return the new head of the reversed list
        return previous;
    }
}
```

C++ Solution

```cpp
// Definition for singly-linked list node
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
};

/**
 * Reverses a portion of a linked list from a start node to an end node, both inclusive.
 * @param start - The first node in the sequence to be reversed.
 * @param end - The last node in the sequence to be reversed.
 * @return A pair where the first element is the new head and the second is the new tail.
 */
std::pair<ListNode*, ListNode*> reverseListNode(ListNode* start, ListNode* end) {
    ListNode* prev = end->next;
    ListNode* curr = start;
    // Iteratively reverse nodes until the end is reached
    while (prev != end) {
        ListNode* temp = curr->next;
        curr->next = prev;
        prev = curr;
        curr = temp;
    }
    return {end, start}; // The start becomes the new end, and the end becomes the new start
}

/**
 * Reverses nodes in k-group.
 * @param head - The head of the linked list.
 * @param k - The size of the groups of nodes within.
 * @return The head of the modified linked list after k-group reversals.
 */
ListNode* reverseKGroup(ListNode* head, int k) {
    ListNode dummyNode(0); // Dummy node to simplify edge cases
    dummyNode.next = head;
    ListNode* predecessor = &dummyNode;

    while (head != nullptr) {
        ListNode* tail = predecessor;

        // Find the k-th node from the head or end if there are less than k nodes left
        for (int i = 0; i < k; ++i) {
            tail = tail->next;
            if (tail == nullptr) {
                return dummyNode.next; // Less than k nodes, return the list as is
            }
        }

        ListNode* nextGroupHead = tail->next;
        // Reverse the current group of k nodes
        auto reversed = reverse(head, tail);

        // Connect the reversed group with the rest of the list
        predecessor->next = reversed.first;
        reversed.second->next = nextGroupHead;

        // Move the pointers to the next group of nodes
        predecessor = reversed.second;
        head = nextGroupHead;
    }

    return dummyNode.next; // Return the new head of the list
}
};
```

Typescript Solution

```typescript
// Definition for singly-linked list node
interface ListNode {
    val: number
    next: ListNode | null
}

/**
 * Reverses a portion of a linked list from a start node to an end node, both inclusive.
 * @param start - The first node in the sequence to be reversed.
 * @param end - The last node in the sequence to be reversed.
 * @return A tuple where the first element is the new head and the second is the new tail.
 */
function reverseListNode(start: ListNode, end: ListNode): [ListNode, ListNode] {
    let prev = end.next;
    let curr = start;
    // Iteratively reverse nodes until the end is reached
    while (prev !== end) {
        let temp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = temp;
    }
    return [end, start];
}

/**
 * Reverses nodes in k-group.
 * @param head - The head of the linked list.
 * @param k - The size of the groups of nodes within.
 * @return The head of the modified linked list after k-group reversals.
 */
function reverseKGroup(head: ListNode | null, k: number): ListNode | null {
    let dummyNode = { val: 0, next: head }; // Dummy node to simplify edge cases
    let predecessor = dummyNode;

    while (head != null) {
        let tail = predecessor;

        // Find the k-th node from the head or end if there are less than k nodes left
        for (let i = 0; i < k; ++i) {
            tail = tail.next;
            if (tail == null) {
                return dummyNode.next;
            }
        }

        let nextGroupHead = tail.next;
        // Reverse the current group of k nodes
        [head, tail] = reverseHead, tail]

        // Connect the reversed group with the rest of the list
        predecessor.next = head;
        tail.next = nextGroupHead;

        // Move the pointers to the next group of nodes
        predecessor = tail;
        head = nextGroupHead;
    }

    return dummyNode.next; // Return the new head of the list
};
```

Time and Space Complexity

The time complexity of the given code can be understood by analyzing the two main operations it performs: traversal of the linked list and reversal of each k-sized group.

1. **Traversal Time Complexity:**
   To ascertain whether a k-group can be reversed, the list is traversed in segments up to `k` nodes. This check is carried out `n/k` times where `n` is the total number of nodes in the list. If a full group of `k` nodes is found, a reversal is performed; if not, the remaining nodes are left as is.
2. **Reversal Time Complexity:**
   For each k-group identified, a reversal is performed. The reversal operation within a group of size `k` is O(k). Since there are `n/k` such groups in the list, the total time taken for all reversals amounts to `n * (k/k) = n`.

Therefore, the combined time complexity for traversing and reversing the linked list in k-groups is $O(n)$, where `n` is the total number of nodes in the linked list.

Space Complexity:

The space usage of the algorithm comes from the variables that hold pointers and the recursive call stack when reversing the linked list. Since the reversal is done iteratively here, there is no additional space complexity due to recursion.

The iterative approach only uses a fixed number of pointers (`pre`, `p`, `q`, `cur`, `start`, `t`, and `dummy`), so the space complexity is $O(1)$, which is constant and does not depend on the number of nodes in the linked list or the group size `k`.