25. Reverse Nodes in k-Group Recursion **Linked List Leetcode Link** Hard

## **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

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

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

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.

which may have less than k nodes if the list's length is not a multiple of k.

- 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

**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

space complexity of 0(1).

**Example Walkthrough** 

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

4 Desired Outcome: 2 -> 1 -> 4 -> 3 -> 5

3 We are expected to reverse every two nodes:

the reversed segment back to the main list.

about losing the reference to the start of the list.

pointer approach for reversing a linked list (pre, p, q).

reset to pre. This ensures that we are ready to process the next segment.

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

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

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.

segment from the rest of the list. We call the reverseList helper function, which reverses the segment using the classic three-

4. Reverse the segment: Once a full segment of k nodes has been confirmed by the cur pointer, we temporarily detach this

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 t, 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 pre.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

The pattern used in this solution could be described as a modified two-pointer technique where one pointer (cur) is used to identify

efficiently manipulates references within the list, never requiring more than a constant amount of additional space, which leads to its

the segment to be reversed, and another pointer (pre) is used to manage connections between reversed segments. The algorithm

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).

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

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.

### ○ The rest of the list is temporarily disconnected, so we have dummy -> 2 -> 1 and then 3 -> 4 -> 5. 5. Reconnect segments:

remains as it is.

7. Final Reconnecting:

**Python Solution** 

class Solution:

9

10

11

12

13

14

16

22

23

24

25

26

28

29

30

31

32

33

38

39

40

41

42

43

44

45

9

10

11

12

13

14

15

16

17

18

20

21

23

24

26

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

48

49

50

51

52

53

55

54 }

6 };

13

16

19

20

21

22

23

24

26

25 }

/\*\*

\*/

/\*\*

self.next = next

def reverse\_list(node):

while current:

return prev

while head:

count = 0

current = head

if count == k:

count += 1

6. Preparation for the next iteration:

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

We then reverse the 3 -> 4 segment to get 4 -> 3.

4. Reverse the segment:

• 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.

• We move the pre pointer to the end of the reversed segment (node 1), and reset cur to pre.

By moving cur pointer two steps, we have cur at node 4, confirming a full segment (3 and 4).

○ The list is now fully processed and connected: dummy -> 2 -> 1 -> 4 -> 3 -> 5.

• 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 5), which doesn't form a full segment of k=2, so it

○ The list remains as dummy -> 2 -> 1 -> 3 -> 4 -> 5.

• We verify that there is a full segment of k nodes (1 and 2) to reverse.

○ We detach the segment 1 → 2, and we reverse it, such that it becomes 2 → 1.

class ListNode: def \_\_init\_\_(self, val=0, next=None): self.val = val

def reverseKGroup(self, head: ListNode, k: int) -> ListNode:

# Check if there is a full group to reverse

# If we have k nodes, proceed to reverse

next\_group = k\_group\_end.next

k\_group\_end.next = None

head.next = next\_group

// Iterate through the list

while (current.next != null) {

if (current == null) {

current.next = null;

// Reverse the k nodes

start.next = temp;

return dummyNode.next;

predecessor = start;

current = predecessor;

\* Helper method to reverse the linked list.

\* @return The new head of the reversed list.

private ListNode reverseList(ListNode head) {

currentNode.next = previous;

while (currentNode != null) {

previous = currentNode;

currentNode = nextNode;

// Definition for singly-linked list node

ListNode\* current = start;

while (prev != end) {

prev = current;

current = temp;

ListNode\* prev = end->next;

current->next = prev;

ListNode(int x) : val(x), next(nullptr) {}

ListNode\* temp = current->next;

\* @param start - The first node in the sequence to be reversed.

std::pair<ListNode\*, ListNode\*> reverse(ListNode\* start, ListNode\* end) {

\* @param end - The last node in the sequence to be reversed.

// Iteratively reverse nodes until the end is reached

return previous;

\* @param head The head of the list to be reversed.

ListNode previous = null, currentNode = head;

ListNode nextNode = currentNode.next;

// Return the new head of the reversed list

// Traverse the list and reverse the links

// Check if there are k nodes to reverse

current = current.next;

return dummyNode.next;

ListNode temp = current.next;

ListNode start = predecessor.next;

predecessor.next = reverseList(start);

for (int i = 0; i < k && current != null; i++) {</pre>

// Detach the k nodes from the rest of the list

// 'start' will be the new tail after the reversal

// Connect the new tail to the temp segment stored before

// Move the predecessor and current pointers k nodes ahead

\* Reverses a portion of a linked list from a start node to an end node, both inclusive.

\* @returns A pair where the first element is the new head and the second is the new tail.

return {end, start}; // The start becomes the new end, and the end becomes the new start

// If less than k nodes remain, no more reversing is needed

// Temporarily store the next segment to be addressed after reversal

# Detach the k group and reverse it

new\_group\_head = reverse\_list(head)

prev\_group.next = new\_group\_head

k\_group\_end = k\_group\_end.next

# Connect the reversed group back to the list

for \_ in range(k - 1): # Move to the end of the k group

# Move prev\_group and head for the next round of reversal

# Helper function to reverse a linked list

next\_temp = current.next

prev, current = None, node

current.next = prev

current = next\_temp

while count < k and current:</pre>

k\_group\_end = head

current = current.next

prev = current

17 # The dummy node is used to handle edge cases smoothly 18 19 dummy = ListNode() 20 dummy.next = head 21 prev\_group = dummy

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.

### 34 35 36 37

```
prev_group = head
46
                   nead = next_group
48
               else:
49
                   # Not enough nodes to fill a k group, so we are done
50
                   break
51
52
           # Return the new head of the modified list
53
           return dummy.next
54
Java Solution
   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 dummyNode = new ListNode(0, head);
           ListNode predecessor = dummyNode, current = dummyNode;
```

# C++ Solution

struct ListNode {

int val;

ListNode \*next;

```
27 /**
    * Reverses nodes in k-group.
    * @param head - The head of the linked list.
    * @param k - The size of the group to reverse nodes within.
    * @returns The head of the modified linked list after k-group reversals.
32
    */
   ListNode* reverseKGroup(ListNode* head, int k) {
       ListNode dummyNode(0); // Dummy node to simplify edge cases
34
       dummyNode.next = head;
35
       ListNode* predecessor = &dummyNode;
36
37
38
       while (head != nullptr) {
39
           ListNode* tail = predecessor;
40
           // Find the k-th node from the head or end if there are less than k nodes left
41
           for (int i = 0; i < k; ++i) {
42
               tail = tail->next;
43
               if (tail == nullptr) {
44
45
                    return dummyNode.next; // Less than k nodes, return the list as is
46
47
48
49
           ListNode* nextGroupHead = tail->next;
           // Reverse the current group of k nodes
50
51
           auto reversed = reverse(head, tail);
52
53
           // Connect the reversed group with the rest of the list
           predecessor->next = reversed.first;
54
55
            reversed.second->next = nextGroupHead;
56
57
           // Move the pointers to the next group of nodes
58
           predecessor = reversed.second;
59
           head = nextGroupHead;
60
61
62
       return dummyNode.next; // Return the new head of the list
63 }
64
Typescript Solution
  // Definition for singly-linked list node
   interface ListNode {
       val: number;
       next: ListNode | null;
 5
 6
    * 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.
    * @returns A tuple where the first element is the new head and the second is the new tail.
12
    */
   function reverse(start: ListNode, end: ListNode): [ListNode, ListNode] {
       let current = start;
14
```

## list and reversal of each k-sized group. 1. Traversal Time Complexity:

remaining nodes are left as is.

2. Reversal Time Complexity:

let prev = end.next;

while (prev !== end) {

prev = current;

current = temp;

return [end, start];

\* Reverses nodes in k-group.

let temp = current.next;

\* @param head - The head of the linked list.

current.next = prev;

let predecessor = dummyNode;

let tail = predecessor;

for (let i = 0; i < k; ++i) {

let nextGroupHead = tail.next;

predecessor.next = head;

predecessor = tail;

head = nextGroupHead;

Time and Space Complexity

tail.next = nextGroupHead;

return dummyNode.next;

// Reverse the current group of k nodes

// Connect the reversed group with the rest of the list

// Move the pointers to the next group of nodes

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

[head, tail] = reverse(head, tail);

tail = tail.next!;

if (tail == null) {

while (head !== null) {

// Iteratively reverse nodes until the end is reached

\* @param k - The size of the group to reverse nodes within.

\* @returns 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

// Find the k-th node from the head or end if there are less than k nodes left

15

16

17

18

20

21

22

23

25

26

31

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

62

24 }

/\*\*

\*/

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 k \* (n/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:** 

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 0(1),

which is constant and does not depend on the number of nodes in the linked list or the group size k.

The time complexity of the given code can be understood by analyzing the two main operations it performs: traversal of the linked

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

The space usage of the algorithm comes from the variables that hold pointers and the recursive call stack when reversing the linked