61. Rotate List

Medium Linked List Two Pointers

Problem Description

n moves. If a list is empty or has a single node, it remains unchanged after rotation. Intuition

rotating the list to the right by k places. If the list has n nodes and k is greater than n, the rotation should be effective after k modulo

The problem presents a singly linked list and an integer k. The task is to move the last k nodes of the list to the front, essentially

1. Determine the length of the list. This helps to understand how many rotations actually need to be performed since rotating a list by its length n, or multiples of n, results in the same list.

To address this problem, we can follow a series of logical steps:

places, we calculate k = n. This simplifies the problem by ensuring that we don't perform unnecessary rotations. 3. Identify the k-th node from the end (or (n - k)-th from the beginning after adjusting k) which after rotation will become the new head of the list. We use two pointers, fast and slow. We initially set both to the head of the list and move fast k steps

2. Since rotating by k places where k is greater than the length of the list (let's call it n) is the same as rotating by k modulo n

- forward. 4. We then advance both fast and slow pointers until fast reaches the end of the list. At this point, slow will be pointing to the
- node right before the k-th node from the end.
- slow's next becomes the new head of the rotated list, and we also need to set slow's next to None to indicate the new end of the list. Following this approach leads us to the rotated list which is required by the problem.

5. We update the pointers such that fast's next points to the old head, making the old tail the new neighbor of the old head. The

Solution Approach The implementation of the solution uses the two-pointer technique along with an understanding of [linked list]

1. Check for Edge Cases: If the head of the list is None or if there is only one node (head.next is None), there is nothing to rotate,

2. Calculate the Length (n): We traverse the entire list to count the number of nodes, storing this count in n. This traversal is done

so we simply return the head.

(/problems/linked_list_cycle) traversal to achieve the rotation. Here is the step-by-step walk-through:

using a while loop that continues until the current node (cur) is None.

help us find the new head after the rotations.

right before the new head of the rotated list.

3. Adjust k by Modulo: Since rotating the list n times results in the same list, we can reduce k by taking k modulo n (k 🞏 n). This simplifies the number of rotations needed to the minimum effective amount.

same. Thus, we can return the head without any modifications. 5. Initialize Two Pointers: Start with two pointers, fast and slow, both referencing the head of the list. These pointers are going to

4. Early Exit for k = 0: If k becomes 0 after the modulo operation, this means the list should not be rotated as it would remain the

7. Move Both Pointers Until Fast Reaches the End: Now, move both fast and slow pointers simultaneously one step at a time until fast is at the last node of the list. Due to the initial k steps taken, slow will now be pointing to the (n-k-1)-th node or the one

6. Move Fast Pointer: Forward the fast pointer by k steps. Doing this will place fast exactly k nodes ahead of slow in the list.

8. Perform the Rotation: The node following slow (slow.next) is the new head of the rotated list (ans). To complete the rotation, we set the next node of the current last node (fast.next) to the old head (head).

By following the above steps, we have rotated the list to the right by k places effectively and efficiently. As a result, the ans pointer,

To mark the new end of the list, we assign None to the next of slow (slow.next = None).

1. Check for Edge Cases: The list is not empty and has more than one node, so we can proceed.

now referring to the new head of the list, is then returned as the final rotated list.

2. Calculate the Length (n): By traversing the list, we determine the length n = 5.

• The node after slow (slow.next), which is 4, will become the new head.

def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:

Use two pointers, fast and slow. Start both at the beginning of the list

At this point, slow is at the node before the new head after rotation

slow.next = None # The next pointer of the new tail should point to None

fast.next = head # The next pointer of the old tail should point to the old head

We can now adjust the pointers to complete the rotation

return new_head # Return the new head of the list

If the list is empty or has just one element, no rotation is needed

4. Early Exit for k = 0: k is not 0, so we do need to perform a rotation.

3. Adjust k by Modulo: Since k = 2 is not greater than n, k % n is still k. Thus, k remains 2.

- Let's illustrate the solution approach with a small example: Suppose we have a linked list $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5]$ and we are given k = 2.
- 5. Initialize Two Pointers: We set both fast and slow to the head of the list. Currently, fast and slow are pointing to 1. 6. Move Fast Pointer: We advance fast by k steps: from 1 to 2, then 2 to 3. Now, fast is pointing to 3, and slow is still at 1.

8. Perform the Rotation:

Python Solution

2 class ListNode:

class Solution:

9

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

35

36

37

38

39

40

41

15

16

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

53

54

55

56

58

55

56

57

58

60

10

13

17

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

47

48

49

50

53

54

55

56

57

59

60

61

62

65

64 }

*/

11 /**

59 };

57 }

1 # Definition for singly-linked list.

self.val = val

while current:

k %= length

length += 1

return head

fast = slow = head

for _ in range(k):

new_head = slow.next

fast = fast.next

current = current.next

if k == 0: # If no rotation is needed

Move the fast pointer k steps ahead

// Find the length of the linked list

// If k is 0, the list remains unchanged

// Return the new head of the rotated list

// Return the new head of the rotated list

constructor(val: number = 0, next: ListNode | null = null) {

* @param {ListNode | null} head - The head of the linked list.

* @return {ListNode | null} - The head of the rotated linked list.

// Set two pointers, fast and slow, initially at the head.

// The fast pointer points to the last node in the list.

// Return the new head of the rotated list.

function rotateRight(head: ListNode | null, k: number): ListNode | null {

// If the head is null or there is only one node, return the head as no rotation is needed.

// Normalize k to prevent unnecessary rotations if k is greater than or equal to length.

// If k is 0 after the modulo operation, no rotation is needed; return the original head.

// Move both slow and fast pointers until the fast reaches the end of the list.

fast.next = head; // Connect the original end of the list to the original head.

The time complexity of the given Python code can be broken down into the following steps:

// The slow pointer now points to the node just before the rotation point.

let newHead = slow.next; // This will be the new head after rotation.

slow.next = null; // Break the chain to form a new end of the list.

* @param {number} k - Number of places to rotate the list.

return newHead;

1 // Definition for singly-linked list.

next: ListNode | null;

this.val = val;

this.next = next;

if (!head || !head.next) {

return head;

let length: number = 0;

while (current) {

length++;

* Rotates the list to the right by k places.

// Find the length of the linked list.

let current: ListNode | null = head;

current = current.next;

Typescript Solution

class ListNode {

val: number;

// Normalize k in case it's larger than the list's length

// Use two pointers: 'fast' will lead 'slow' by 'k' nodes

ListNode current = head;

while (current != null) {

current = current.next;

int length = 0;

k %= length;

if (k == 0) {

return head;

ListNode fast = head;

ListNode slow = head;

fast = fast.next;

while (fast.next != null) {

fast = fast.next;

slow = slow.next;

while (k > 0) {

return newHead;

k--;

length++;

self.next = next

def __init__(self, val=0, next=None):

Example Walkthrough

 Move slow to 2 and fast to 4. Move slow to 3 and fast to 5.

Now fast is at 5, the end of the list, and slow is at 3.

• We set fast.next (which is 5.next) to the old head (1), forming a connection from 5 to 1. • We update slow.next to None to indicate the new end of the list.

7. Move Both Pointers Until Fast Reaches the End: We advance both pointers until fast is at the last node:

front of the list. By returning the new head (4 in this case), we conclude the rotation process and the modified list is correctly rotated by k places.

After performing rotation, the list now becomes $[4 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 3]$ because the last two nodes (4 and 5) have been moved to the

if head is None or head.next is None: 10 11 return head 12 # Count the number of elements in the linked list 13 current, length = head, 0 14

Compute the actual number of rotations needed as k could be larger than the length of the list

31 # Move both pointers at the same speed until fast reaches the end of the list 32 while fast.next: 33 fast, slow = fast.next, slow.next 34

```
42
 43 # Note: The Optional[ListNode] type hint should be imported from typing
 44 # if you want to use it for type checking in Python 3.
 45 # Otherwise, you can omit it from the function signatures.
 46
Java Solution
  // Definition for singly-linked list.
   class ListNode {
       int val;
       ListNode next;
       ListNode() {}
       ListNode(int val) { this.val = val; }
       ListNode(int val, ListNode next) { this.val = val; this.next = next; }
8 }
 9
   class Solution {
       public ListNode rotateRight(ListNode head, int k) {
11
           // If the list is empty or has one node, no rotation needed
12
13
           if (head == null || head.next == null) {
14
               return head;
```

// Move both at the same pace. When 'fast' reaches the end, 'slow' will be at the k-th node from the end

47 // 'slow' is now at the node after which the rotation will occur ListNode newHead = slow.next; 48 49 // Break the list at the node 'slow' and make 'fast' point to the original head to rotate 50 slow.next = null; 51 52 fast.next = head;

```
C++ Solution
1 /**
    * Definition for singly-linked list.
    * struct ListNode {
          int val;
          ListNode *next;
          ListNode(): val(0), next(nullptr) {}
          ListNode(int x) : val(x), next(nullptr) {}
          ListNode(int x, ListNode *next) : val(x), next(next) {}
    * };
    */
   class Solution {
  public:
       ListNode* rotateRight(ListNode* head, int k) {
           // If the head is null or there is only one node, return the head as rotation isn't needed
14
           if (!head || !head->next) {
16
               return head;
17
18
19
           // find the length of the linked list
20
           ListNode* current = head;
           int length = 0;
22
           while (current) {
23
               ++length;
24
               current = current->next;
25
26
           // Normalize k to prevent unnecessary rotations if k >= length
28
           k %= length;
29
30
           // If k is 0 after modulo operation, no rotation is needed; return the original head
31
           if (k == 0) {
32
               return head;
33
34
           // Set two pointers, fast and slow initially at head
35
36
           ListNode* fast = head;
37
           ListNode* slow = head;
38
           // Move fast pointer k steps ahead
39
           while (k--) {
40
41
               fast = fast->next;
42
43
44
           // Move both slow and fast pointers until fast reaches the end of the list
           while (fast->next) {
45
               fast = fast->next;
46
47
               slow = slow->next;
48
49
50
           // The slow pointer now points to the node just before rotation point
51
           // The fast pointer points to the last node of the list
52
           ListNode* newHead = slow->next; // This will be the new head after rotation
53
           slow->next = nullptr; // Break the chain to form a new end of the list
           fast->next = head; // Connect the original end of the list to the original head
54
```

let slow: ListNode = head; 42 43 44 // Move the fast pointer k steps ahead. 45 while (k > 0) { 46 fast = fast.next!;

k--;

while (fast.next) {

return newHead;

fast = fast.next;

slow = slow.next!;

Time and Space Complexity

k %= length;

if (k === 0) {

return head;

let fast: ListNode = head;

2. Calculate k = n: This calculation is done in constant time, 0(1). 3. Moving the fast pointer k steps ahead: This again takes O(k) time, but since k in this case is always less than n after the modulo

Space Complexity

list exactly once.

Time Complexity

operation, we can say this operation takes at most 0(n) time. 4. Moving both fast and slow pointers to find the new head of the rotated list: This has to traverse the remainder of the list, which takes at most 0(n-k) time. However, in worst-case scenarios where k is 0, this would result in 0(n) time. Since the previous steps

ensure that k < n, the combined operations will still be O(n). 5. Re-link the end of the list to the previous head and set the next of the new tail to None: These operations are done in constant

1. Iterate through the linked list to find out the length n: This process takes O(n) time as it goes through all elements of the linked

- time, 0(1). In all, considering that O(n) is the dominating factor, the overall time complexity of the code is O(n).
- The space complexity of the code can also be analyzed: 1. The given algorithm only uses a fixed amount of extra space for variables cur, n, fast, and slow, regardless of the size of the input linked list.
- 2. No additional data structures are used that depend on the size of the input. 3. All pointer moves and assignments are done in-place.
- Hence, the space complexity is 0(1), which means it requires constant extra space.