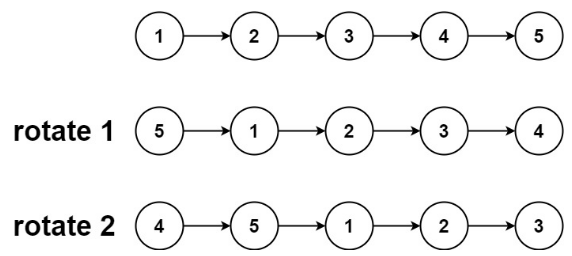
<https://leetcode.com/problems/rotate-list/>

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

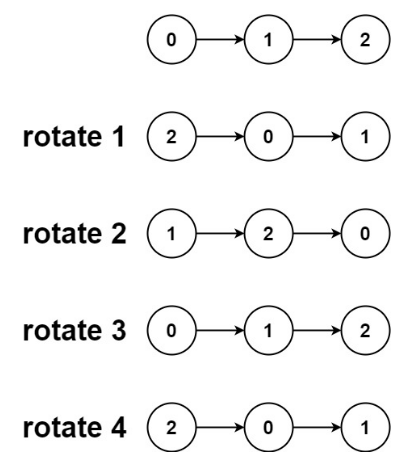
**Example 1:**



Input: head = [1,2,3,4,5], k = 2

Output: [4,5,1,2,3]

**Example 2:**



Input: head = [0,1,2], k = 4

Output: [2,0,1]

**Constraints:**

The number of nodes in the list is in the range [0, 500].

-100 <= Node.val <= 100

0 <= k <= 2 \* 10^9

**Attempt 1: 2023-02-21**

**Solution 1: Find length and new head (10 min)**

**Style 1: Classic way**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\*    int val;

\*    ListNode next;

\*    ListNode() {}

\*    ListNode(int val) { this.val = val; }

\*    ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

    public ListNode rotateRight(ListNode head, int k) {

        if(head == null || head.next == null) {

            return head;

        }

        // Find length

        ListNode iter = head;

        // This way not convenient because besides get length,

        // we also have to get 'tail' node and use 'tail' to

        // connect original 'head' to build rotation first

        //int len = 0;

        //while(iter != null) {

        //    iter = iter.next;

        //    len++;

        //}

        int len = 1;

        while(iter.next != null) {

            iter = iter.next;

            len++;

        }

        // Connect 'iter'(point to tail node now) to original

        // 'head' prepare for rotation cut

        iter.next = head;

        // Find prior node of new head

        ListNode iter1 = head;

        int i = 0;

        while(i < len - k % len - 1) {

            iter1 = iter1.next;

            i++;

        }

        ListNode prev = iter1;

        ListNode newHead = prev.next;

        // Cut between prior node and new head

        prev.next = null;

        return newHead;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/rotate-list/solutions/22735/my-clean-c-code-quite-standard-find-tail-and-reconnect-the-list/>

There is no trick for this problem. Some people used slow/fast pointers to find the tail node but I don't see the benefit (in the sense that it doesn't reduce the pointer move op) to do so. So I just used one loop to find the length first.

class Solution {

public:

    ListNode\* rotateRight(ListNode\* head, int k) {

        if(!head) return head;

        int len=1; // number of nodes

        ListNode \*newH, \*tail;

        newH=tail=head;

        while(tail->next)  // get the number of nodes in the list

        {

            tail = tail->next;

            len++;

        }

        tail->next = head; // circle the link

        if(k %= len)

        {

            for(auto i=0; i<len-k; i++) tail = tail->next; // the tail node is the (len-k)-th node (1st node is head)

        }

        newH = tail->next;

        tail->next = NULL;

        return newH;

    }

};

**Style 2: Fast and Slow pointers**

**Refer to**

<https://algo.monster/liteproblems/61>

**Problem Description**

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

**Intuition**

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

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.

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 places, we calculate k %= n. This simplifies the problem by ensuring that we don't perform unnecessary rotations.

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 forward.

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.

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 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.

**Solution Approach**

The implementation of the solution uses the two-pointer technique along with an understanding of [linked list](/problems/linked_list_cycle) traversal to achieve the rotation. Here is the step-by-step walk-through:

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, so we simply return the head.

Calculate the Length (n): We traverse the entire list to count the number of nodes, storing this count in n. This traversal is done using a while loop that continues until the current node (cur) is None.

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.

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 same. Thus, we can return the head without any modifications.

Initialize Two Pointers: Start with two pointers, fast and slow, both referencing the head of the list. These pointers are going to help us find the new head after the rotations.

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

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 right before the new head of the rotated list.

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).

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

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

**Example Walkthrough**

Let's illustrate the solution approach with a small example:

Suppose we have a linked list [1 -> 2 -> 3 -> 4 -> 5] and we are given k = 2.

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

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

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

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

**Initialize Two Pointers: We set both fast and slow to the head of the list. Currently, fast and slow are pointing to 1.**

**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.**

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

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.**

Perform the Rotation:

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

**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.**

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

**Solution Implementation**

/\*\*

 \* Definition for singly-linked list.

 \* public class ListNode {

 \*     int val;

 \*     ListNode next;

 \*     ListNode() {}

 \*     ListNode(int val) { this.val = val; }

 \*     ListNode(int val, ListNode next) { this.val = val; this.next = next; }

 \* }

 \*/

class Solution {

    public ListNode rotateRight(ListNode head, int k) {

        // If the list is empty or has one node, no rotation needed

        if (head == null || head.next == null) {

            return head;

        }

        // Find the length of the linked list

        ListNode current = head;

        int length = 0;

        while (current != null) {

            length++;

            current = current.next;

        }

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

        k %= length;

        // If k is 0, the list remains unchanged

        if (k == 0) {

            return head;

        }

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

        ListNode fast = head;

        ListNode slow = head;

        while (k > 0) {

            fast = fast.next;

            k--;

        }

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

        while (fast.next != null) {

            fast = fast.next;

            slow = slow.next;

        }

        // 'slow' is now at the node after which the rotation will occur

        ListNode newHead = slow.next;

        // Break the list at the node 'slow' and make 'fast' point to the original head to rotate

        slow.next = null;

        fast.next = head;

        // Return the new head of the rotated list

        return newHead;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Time and Space Complexity**

**Time Complexity**

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

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 list exactly once.

Calculate k %= n: This calculation is done in constant time, O(1).

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 operation, we can say this operation takes at most O(n) time.

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 O(n-k) time. However, in worst-case scenarios where k is 0, this would result in O(n) time. Since the previous steps ensure that k < n, the combined operations will still be O(n).

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 time, O(1).

In all, considering that O(n) is the dominating factor, the overall time complexity of the code is O(n).

**Space Complexity**

The space complexity of the code can also be analyzed:

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.

No additional data structures are used that depend on the size of the input.

All pointer moves and assignments are done in-place.

Hence, the space complexity is O(1), which means it requires constant extra space.