<https://leetcode.com/problems/split-linked-list-in-parts/description/>

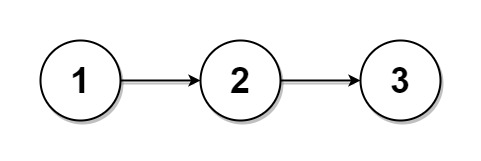
Given the head of a singly linked list and an integer k, split the linked list into k consecutive linked list parts.

The length of each part should be as equal as possible: no two parts should have a size differing by more than one. This may lead to some parts being null.

The parts should be in the order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal to parts occurring later.

Return an array of the k parts.

**Example 1:**



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

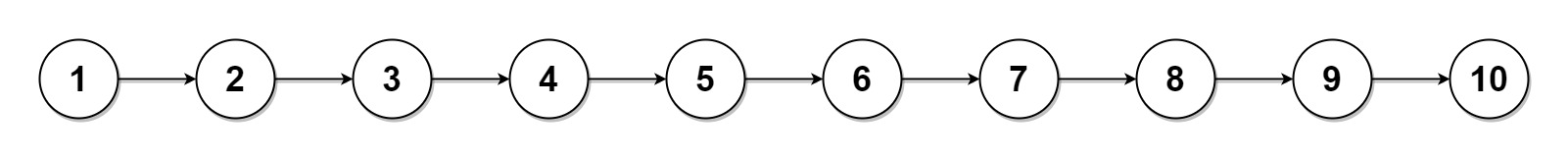
Output: [[1],[2],[3],[],[]]

**Explanation:**

The first element output[0] has output[0].val = 1, output[0].next = null.

The last element output[4] is null, but its string representation as a ListNode is [].

**Example 2:**



**Constraints:**

The number of nodes in the list is in the range [0, 1000].

0 <= Node.val <= 1000

1 <= k <= 50

**Attempt 1: 2024-05-07**

**Solution 1: Linked List (30 min)**

/\*\*

 \* 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[] splitListToParts(ListNode root, int k) {

        ListNode[] result = new ListNode[k];

        if(root == null) {

            return result;

        }

        int total\_nodes = 0;

        // Find given list length

        ListNode iter = root;

        while(iter != null) {

            total\_nodes++;

            iter = iter.next;

        }

        // If there are N nodes in the list, and k parts, then every part has

        // N / k elements, except the first N % k parts have an extra one

        int base\_num = total\_nodes / k;

        int additional\_sections = total\_nodes % k;

        ListNode currStart = root;

        // 'prev' used for tracking as iterator and cut off tailing

        ListNode prev = new ListNode(-1);

        prev.next = currStart;

        for(int i = 0; i < k; i++) {

            result[i] = currStart;

            // For first N % k parts loop 1 more round

            int count = base\_num;

            if(i < additional\_sections) {

                count++;

            }

            while(count > 0) {

                prev = currStart;

                currStart = currStart.next;

                count--;

            }

            // Cut off tailing

            prev.next = null;

        }

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/split-linked-list-in-parts/solutions/109296/java-c-clean-code/>

class Solution {

public ListNode[] splitListToParts(ListNode root, int k) {

ListNode[] parts = new ListNode[k];

int len = 0;

for (ListNode node = root; node != null; node = node.next)

len++;

int n = len / k, r = len % k; // n : minimum guaranteed part size; r : extra nodes spread to the first r parts;

ListNode node = root, prev = null;

for (int i = 0; node != null && i < k; i++, r--) {

parts[i] = node;

for (int j = 0; j < n + (r > 0 ? 1 : 0); j++) {

prev = node;

node = node.next;

}

prev.next = null;

}

return parts;

}

}

**Refer to**

<https://algo.monster/liteproblems/725>

**Problem Description**

The problem presents a scenario where we are given the head of a singly linked list and an integer k. Our task is to divide this linked list into k consecutive parts. The objective here is to distribute the nodes of the linked list as evenly as possible among the k parts, which means each part's length should not differ by more than one compared to the other parts.

Since the size division might not always be perfect (for example, when the number of nodes is not a multiple of k), some parts could end up being empty (null).

It's important to note that the parts need to be split by their original order in the list. And when there's a discrepancy in size due to the division process (which means some parts are allowed to have one more node than the others), the larger parts should appear earlier in the resulting array of linked list parts.

The final goal is to return an array of the k parts created from the original list.

**Intuition**

To arrive at a solution, we can follow the outlined steps:

Count the number of nodes in the given linked list. This step will help us to know how to divide the linked list nodes into k parts as equally as possible.

Compute the width (the number of nodes that each part would have if nodes were evenly distributed) and remainder (the number of parts that will have one extra node) using the division and modulo operators, respectively.

Initialize an array of k elements to hold the resulting linked list parts, setting each element initially as None.

Iterate k times, once for each part we need to create:

In each iteration, we consider the current node (cur) as the new part's head.

Then, we skip width nodes, plus one more if the index i is less than the remainder since those parts are allowed to have one additional node.

After skipping the required number of nodes, we disconnect the current part from the rest of the linked list by setting the next pointer of the current node to None.

Move the cur pointer to the next node in the original list, which will then serve as the head for the next part (if any).

Store the head of the current part in our result array.

By going through this process, we end up dividing the list into k parts satisfying the size requirements stated in the problem, and we return the array containing these parts as our final result.

**Solution Approach**

The implementation of the solution involves a step-by-step traversal and division of the linked list, which uses basic concepts of linked list manipulation. The approach doesn't use any advanced algorithms or patterns but is rather a straightforward application of linked list operations. Here are the details:

Initialization: The number of nodes in the linked list is counted by traversing the list once with a simple while loop. This is stored in the variable n.

Division Logic: The key computation happens with the divmod function, which returns two values: width and remainder.

width is the base size of each part when nodes are evenly distributed.

remainder is the count of initial parts that should have one extra node because the nodes cannot be perfectly divided by k.

Resulting Array Creation: An array res is created and pre-filled with None to hold the final parts. This prepares the groundwork for the later storage of the head nodes of each part.

List Division: Now comes the division of the list into k parts iteratively:

For each of the k parts, the loop starts with the current node cur, which becomes the head of the new part.

A nested loop then runs exactly width times, and if the current part index i is less than the remainder, it runs an additional time. This loop effectively skips a number of nodes corresponding to the size of the current part.

Once the end of the current part is reached, the next node is disconnected from the current part by setting the next property of the last node in the current part to None. This is crucial as it ensures the current part is not linked to the next part.

The head node of the current part is then stored in the res array.

The cur pointer is updated to reference the next node in the original list, which will be the head of the following part.

By the end of these steps, we have an array res with k elements, each referencing the head of a sublist that conforms to the problem's division requirements. The simplicity of this approach lies in its effective use of basic linked list operations such as traversal, node counting, and reassignment of the next pointer to split the list as needed.

**Example Walkthrough**

Let's apply the solution approach to a small example to illustrate how it works in practice.

Assume we have the following singly linked list and integer k:

Linked List: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

k = 3

In this example, we want to split the linked list into 3 parts as evenly as possible.

**Step 1: Initialization**

Count the number of nodes in the linked list:

n = 7 (number of nodes in the list)

**Step 2: Division Logic**

Using the divmod function:

width, remainder = divmod(7, 3)

width = 2 (base size of each part)

remainder = 1 (one part will have one extra node)

**Step 3: Resulting Array Creation**

Create the array to hold the results with None initially:

res = [None, None, None]

**Step 4: List Division**

Now divide the linked list into k parts:

**First iteration (i = 0 < remainder):**

Current node cur is at 1.

Skip width + 1 nodes because i < remainder.

New part: 1 -> 2 -> 3 (since width is 2 but we add 1 for the remainder).

Disconnect third node's next pointer.

Move cur to node 4.

Resulting array receives the first part: [1 -> 2 -> 3, None, None].

**Second iteration (i = 1 < remainder):**

cur is now at node 4.

Skip width + 1 nodes again.

New part: 4 -> 5 -> 6.

Disconnect sixth node's next pointer.

Move cur to node 7.

Resulting array now looks like: [1 -> 2 -> 3, 4 -> 5 -> 6, None].

**Third iteration (i = 2 >= remainder):**

cur is now at node 7.

Skip width nodes only because i >= remainder.

New part is just node 7.

No need to disconnect since it's the end of the list.

All other elements in the array are unchanged.

Final array is: [1 -> 2 -> 3, 4 -> 5 -> 6, 7].

The final output is an array of 3 linked lists as evenly distributed as possible, satisfying the problem's constraints.

The resulting linked list parts contained in our res array will look like:

[

1 -> 2 -> 3,

4 -> 5 -> 6,

7

]

Each step consistently applies the logic of linked list division, accounting for both the base width and the remainder to ensure a fair distribution of nodes across the resulting sublists.

**Solution Implementation**

/\*\*

\* Definition for singly-linked list.

\*/

public class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; }

}

class Solution {

/\*\*

\* Splits the linked list into k consecutive parts.

\*

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

\* @param k The number of parts to split the list into.

\* @return An array of ListNode where each element is the head of a part of the split list.

\*/

public ListNode[] splitListToParts(ListNode root, int k) {

int length = 0;

ListNode current = root;

// Calculate the total length of the linked list.

while (current != null) {

++length;

current = current.next;

}

// Calculate the minimum number of nodes each part should have.

int minPartSize = length / k;

// Calculate how many parts should have an extra node.

int extraNodes = length % k;

ListNode[] result = new ListNode[k];

current = root;

// Divide the list into parts.

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

ListNode partHead = current;

// Determine the size of the current part,

// which is minPartSize plus one if this part is supposed to have an extra node.

int currentPartSize = minPartSize + (i < extraNodes ? 1 : 0);

// Traverse to the end of the current part.

for (int j = 0; j < currentPartSize - 1; ++j) {

if (current != null) {

current = current.next;

}

}

// Detach the current part from the remainder of the list.

if (current != null) {

ListNode nextPartHead = current.next;

current.next = null;

current = nextPartHead;

}

// Add the head of the current part to the result array.

result[i] = partHead;

}

return result;

}

}

**Time and Space Complexity**

The time complexity of the given code is O(n + k), where n is the number of elements in the singly-linked list, and k is the number of parts to split the list into.

The first while loop runs through the entire list to count the number of nodes, which takes O(n) time.

After that, the for loop runs k times to create the k parts. Each part might require up to width + 1 iterations to set the next pointer to None and advance the cur pointer. Since width = n // k, in the worst case, this inner loop can be considered a constant operation because it doesn't scale with n (since the number of iterations within each of the k outer iterations rounds to n total). Thus, the entire for loop still takes O(k) time.

So, total time complexity is the sum of the two primary operations which gives us O(n) + O(k).

The space complexity of the code is O(k) because an array of k elements is created to store the heads of the parts of the linked list. There are no other data structures that scale with the size of the input, and the list slices are created in-place without any extra space, except the output array.

**Refer to**

[L328.Odd Even Linked List (Ref.L138,L725)](note://84C5AF3FAE3F41AA8BD0F50DF718C4DF)