<https://leetcode.ca/2023-03-27-2674-Split-a-Circular-Linked-List/>

**Description**

Given a circular linked list list of positive integers, your task is to split it into 2 circular linked lists so that the first one contains the first half of the nodes in list (exactly ceil(list.length / 2) nodes) in the same order they appeared in list, and the second one contains the rest of the nodes in list in the same order they appeared in list.

Return an array answer of length 2 in which the first element is a circular linked list representing the first half and the second element is a circular linked list representing the second half.

A circular linked list is a normal linked list with the only difference being that the last node's next node, is the first node.

**Example 1:**

Input: nums = [1,5,7]

Output: [[1,5],[7]]

Explanation: The initial list has 3 nodes so the first half would be the first 2 elements since ceil(3 / 2) = 2 and the rest which is 1 node is in the second half.

**Example 2:**

Input: nums = [2,6,1,5]

Output: [[2,6],[1,5]]

Explanation: The initial list has 4 nodes so the first half would be the first 2 elements since ceil(4 / 2) = 2 and the rest which is 2 nodes are in the second half.

**Constraints:**

The number of nodes in list is in the range [2, 10^5]

0 <= Node.val <= 10^9

LastNode.next = FirstNode where LastNode is the last node of the list and FirstNode is the first one

**Attempt 1: 2024-05-08**

**Solution 1: Fast and Slow Pointers (30 min)**

class Solution {

// The method splits a circular linked list into two halves.

// If the number of nodes is odd, the extra node goes to the first half.

public ListNode[] splitCircularLinkedList(ListNode head) {

if (head == null) {

return new ListNode[]{null, null};

}

// 'slow' will eventually point to the end of the first half of the list

// 'fast' will be used to find the end of the list to determine the midpoint

ListNode slow = head, fast = head;

// Traverse the list to find the midpoint. Since it's a circular list,

// the conditions check if the traversed list has returned to the head.

while (fast.next != head && fast.next.next != head) {

slow = slow.next; // move slow pointer one step

fast = fast.next.next; // move fast pointer two steps

}

// If there are an even number of elements, move 'fast' to the very end of the list

if (fast.next != head) {

fast = fast.next;

}

// The 'secondHead' points to the start of the second half of the list

ListNode secondHead = slow.next;

// Split the list into two by reassigning the 'next' pointers

fast.next = secondHead; // Complete the second circular list

slow.next = head; // Complete the first circular list

// Return an array of the two new list heads

return new ListNode[]{head, secondHead};

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Step by Step**

**Even number of nodes**

public class LinkedListSolution {

private class ListNode {

int val;

ListNode next;

ListNode() {}

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

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

}

public ListNode[] splitCircularLinkedList(ListNode head) {

if (head == null) {

return new ListNode[]{null, null};

}

// 'slow' will eventually point to the end of the first half of the list

// 'fast' will be used to find the end of the list to determine the midpoint

ListNode slow = head, fast = head;

// Traverse the list to find the midpoint. Since it's a circular list,

// the conditions check if the traversed list has returned to the head.

while (fast.next != head && fast.next.next != head) {

slow = slow.next; // move slow pointer one step

fast = fast.next.next; // move fast pointer two steps

}

// If there are an even number of elements, move 'fast' to the very end of the list

if (fast.next != head) {

fast = fast.next;

}

// The 'secondHead' points to the start of the second half of the list

ListNode secondHead = slow.next;

// Split the list into two by reassigning the 'next' pointers

fast.next = secondHead; // Complete the second circular list

slow.next = head; // Complete the first circular list

// Return an array of the two new list heads

return new ListNode[]{head, secondHead};

}

public static void main(String[] args) {

LinkedListSolution l = new LinkedListSolution();

ListNode node1 = l.new ListNode(1);

ListNode node2 = l.new ListNode(2);

ListNode node3 = l.new ListNode(3);

ListNode node4 = l.new ListNode(4);

ListNode node5 = l.new ListNode(5);

node1.next = node2;

node2.next = node3;

node3.next = node4;

node4.next = node1;

//node4.next = node5;

//node5.next = node1;

ListNode[] result = l.splitCircularLinkedList(node1);

System.out.println(result);

}

}

**Odd number of nodes**

public class LinkedListSolution {

private class ListNode {

int val;

ListNode next;

ListNode() {}

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

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

}

public ListNode[] splitCircularLinkedList(ListNode head) {

if (head == null) {

return new ListNode[]{null, null};

}

// 'slow' will eventually point to the end of the first half of the list

// 'fast' will be used to find the end of the list to determine the midpoint

ListNode slow = head, fast = head;

// Traverse the list to find the midpoint. Since it's a circular list,

// the conditions check if the traversed list has returned to the head.

while (fast.next != head && fast.next.next != head) {

slow = slow.next; // move slow pointer one step

fast = fast.next.next; // move fast pointer two steps

}

// If there are an even number of elements, move 'fast' to the very end of the list

if (fast.next != head) {

fast = fast.next;

}

// The 'secondHead' points to the start of the second half of the list

ListNode secondHead = slow.next;

// Split the list into two by reassigning the 'next' pointers

fast.next = secondHead; // Complete the second circular list

slow.next = head; // Complete the first circular list

// Return an array of the two new list heads

return new ListNode[]{head, secondHead};

}

public static void main(String[] args) {

LinkedListSolution l = new LinkedListSolution();

ListNode node1 = l.new ListNode(1);

ListNode node2 = l.new ListNode(2);

ListNode node3 = l.new ListNode(3);

ListNode node4 = l.new ListNode(4);

ListNode node5 = l.new ListNode(5);

node1.next = node2;

node2.next = node3;

node3.next = node4;

node4.next = node5;

node5.next = node1;

ListNode[] result = l.splitCircularLinkedList(node1);

System.out.println(result);

}

}

**Refer to**

<https://algo.monster/liteproblems/2674>

**Problem Description**

The given problem requires us to take a circular linked list and split it into two separate circular linked lists. The circular linked list is a sequence of nodes interconnected in such a way that each node points to the next node, and the last node points back to the first node, forming a circle. We must divide this list into two halves: the first half should contain the first part of the nodes, while the second half should contain the remaining nodes. Importantly, the split is not necessarily into two equal halves; if the list has an odd number of nodes, the first half should have one more node than the second half (ceiling of half the length of the list). This split must maintain the original order of the nodes.

The resulting sublists should also be circular linked lists, where each one ends by pointing to its own first node. The output should be an array with two elements, each representing one of the halves as a circular linked list.

**Intuition**

To find the solution for splitting the linked list, we need to determine the point where the first list ends and the second list begins. As the linked list is circular, we can utilize the Fast and Slow pointer technique, which involves two pointers moving through the list at different speeds. The slow pointer, a, advances one node at a time, while the fast pointer, b, moves two nodes at a time.

By the time the fast pointer completes its cycle (either by reaching the initial node or the one just before it), the slow pointer, a, will be at the midpoint of the list. This is because when the fast pointer has traveled twice as far, the slow pointer has covered half the distance in the same time. The node where the slow pointer stops is where the first list will end, and the second list will begin.

Once we determine the midpoint, we then create the split. This is done by having the next node after the slow pointer be the beginning of the second linked list (list2). We must then fix the next pointers to maintain the circular nature of both lists. The next pointer of the fast pointer (b) should point to the beginning of the second list (list2), forming the second circular list. The slow pointer (a)'s next pointer should point back to the beginning of the first list, maintaining its circular nature. Finally, we return an array containing the start points of both halves of the list, preserving their circular linked list structure.

**Solution Approach**

In the provided Python solution, the circular linked list is split into two halves using Floyd's Tortoise and Hare approach, which is also known for detecting cycles within linked lists. The algorithm employs two pointers: a slow pointer (a) and a fast pointer (b), as described in the intuition section.

Initially, both pointers are set to the start of the list. Then we use a while loop to continue iterating through the list as long as b (the fast pointer) doesn't meet the following conditions:

b.next is not the start of the list (indicating that it's not run through the entire list yet), and

b.next.next is not the start of the list (checking two steps ahead for the fast pointer).

Inside the loop:

We increment the slow pointer a by one step (a = a.next).

The fast pointer b increments by two steps (b = b.next.next).

When the loop exits, the slow pointer a will be at the midpoint, and the fast pointer b will be at the end of the list (or one before the end if the number of nodes is even). The if condition: if b.next != list: checks if the number of nodes is odd. If it is, then we move the fast pointer b one more node forward.

The line list2 = a.next marks the beginning of the second list by taking the node next to where the slow pointer a stopped. To form the second circular list, we set b.next = list2, linking the end of the first list to the start of the second list.

To complete the first circular list, we need to point a.next back to the head of the list (list), closing the loop on the first half of the split with a.next = list.

Finally, we return both list and list2 within an array, which are the heads of the two split circular linked lists.

To recap, here are the exact steps:

Initialize two pointers a and b to the head of the circular linked list.

Move a one node at a time and b two nodes at a time until b is at the end of the list or one node before the end.

If the list has an odd length, increment b to point to the last node.

Set list2 to the node after a, which will be the head of the second circular linked list.

Link the last node of the original list to the head of the second list, forming a circular structure for the second half.

Update the next pointer of the slow pointer a to the original head, forming the circular structure for the first half.

Return the array [list, list2] where list is the head of the first circular list and list2 is the head of the second.

These steps allow us to achieve the desired splitting of the circular linked list while maintaining the circular nature of both resulting lists.

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach for splitting a circular linked list into two halves. Suppose our circular linked list looks like this:

(Head) A -> B -> C -> D -> A

Here A is the head of the list, and the list contains four nodes. Following the steps outlined in the solution approach:

Initialize two pointers a and b to the head of the circular linked list. So a = A and b = A.

Start moving both pointers forward. a moves one node, and b moves two nodes at a time until b is at the end of the list or one node before the end. After the first iteration, a = B and b = C. After the second iteration, a = C and b = A (since b has moved two nodes from C and we have a circular list, it comes back to A).

The list has an even length, so we don't have to increment b.

Set list2 to the node after a, which will be the head of the second circular linked list. Therefore, list2 = D.

Link the last node of the original list to the head of the second list to maintain the circular structure. This means we'll have C -> D -> C.

Update the next pointer of the slow pointer a to the original head, which closes the loop of the first half. This means A -> B -> C -> A.

Return the array [list, list2] where list is the head of the first circular list and list2 is the head of the second. As a result, we have two circular linked lists:

First list: (Head) A -> B -> C -> A

Second list: (Head) D -> C -> D

These steps demonstrate how the original circular linked list was split into two halves while ensuring that both sublists remained circular.

**Solution Implementation**

class Solution {

// The method splits a circular linked list into two halves.

// If the number of nodes is odd, the extra node goes to the first half.

public ListNode[] splitCircularLinkedList(ListNode head) {

if (head == null) {

return new ListNode[]{null, null};

}

// 'slow' will eventually point to the end of the first half of the list

// 'fast' will be used to find the end of the list to determine the midpoint

ListNode slow = head, fast = head;

// Traverse the list to find the midpoint. Since it's a circular list,

// the conditions check if the traversed list has returned to the head.

while (fast.next != head && fast.next.next != head) {

slow = slow.next; // move slow pointer one step

fast = fast.next.next; // move fast pointer two steps

}

// If there are an even number of elements, move 'fast' to the very end of the list

if (fast.next != head) {

fast = fast.next;

}

// The 'secondHead' points to the start of the second half of the list

ListNode secondHead = slow.next;

// Split the list into two by reassigning the 'next' pointers

fast.next = secondHead; // Complete the second circular list

slow.next = head; // Complete the first circular list

// Return an array of the two new list heads

return new ListNode[]{head, secondHead};

}

}

**Time and Space Complexity**

The given code snippet is designed to split a circular singly linked list into two halves. The algorithm uses the fast and slow pointer technique, also known as Floyd's cycle-finding algorithm, to identify the midpoint of the list.

**Time Complexity:**

The time complexity of this algorithm can be determined by analyzing the while loop, which continues until the fast pointer (b) has either completed a cycle or is at the last node.

In each iteration of the loop, the slow pointer (a) moves one step, and the fast pointer (b) moves two steps.

In the worst case scenario, the fast pointer might traverse the entire list before the loop terminates. This would be the case if the number of elements in the list is odd.

If the list has n nodes, and since the fast pointer moves two steps at a time, it will take O(n/2) steps for the fast pointer to traverse the entire list.

Therefore, the time complexity of the loop is O(n).

**Space Complexity:**

The space complexity of this algorithm refers to the additional space used by the algorithm, not including space for the input itself.

The only extra variables used are the two pointers, a and b. These do not depend on the size of the input list but are rather fixed.

Therefore, the space complexity of the algorithm is O(1).

In summary, the time complexity is O(n) and the space complexity is O(1).

**Refer to**

[L725.Split Linked List in Parts (Ref.L328,L2674)](note://WEB29c5aaaa6ebfc33c3f93ec7197e20c85)