2674. Split a Circular Linked List

# **Problem Description**

Medium

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.

The given problem requires us to take a circular linked list and split it into two separate circular linked lists. The 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. 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),

## 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:

as described in the intuition section.

• 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).

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

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

- 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.

form the second circular list, we set b.next = list2, linking the end of the first list to the start of the second list.

the split with a.next = list.

To recap, here are the exact steps:

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

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

1. Initialize two pointers a and b to the head of the circular linked list. 2. 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.

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

6. Update the next pointer of the slow pointer a to the original head, forming the circular structure for the first half. 7. Return the array [list, list2] where list is the head of the first circular list and list2 is the head of the second.

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

- resulting lists.
  - 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

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

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

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

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

# 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**.

-> D -> C.

-> C -> A.

result, we have two circular linked lists:

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

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

Solution Implementation

# Definition for singly-linked list.

fast pointer = head

slow\_pointer = head

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

if fast pointer.next != head:

slow\_pointer.next = head

return [head, second\_half\_head]

# Slow pointer for moving one step at a time

slow pointer = slow pointer.next

fast\_pointer = fast\_pointer.next

fast\_pointer = fast\_pointer.next.next

# The slow pointer now points to the middle of the list,

# Return the two halves, both are now circular linked lists

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

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

public ListNode[] splitCircularLinkedList(ListNode head) {

// Function to split a circular linked list into two halves.

// 'slow ptr' will be in the middle.

fast\_ptr = fast\_ptr->next->next;

slow ptr = slow ptr->next;

fast\_ptr = fast\_ptr->next;

ListNode\* head2 = slow ptr->next:

// and 'head' as the head of the first list.

// It returns an array containing the two halves of the list.

// Return the heads of the two halves in a vector.

if (fast ptr->next != head) {

return {head, head2};

return [null, null];

std::vector<ListNode\*> splitCircularLinkedList(ListNode\* head) {

while (fast ptr->next != head && fast\_ptr->next->next != head) {

ListNode \*slow ptr = head; // 'slow ptr' will eventually point to the mid-point of the list.

// Move 'fast ptr' twice as fast as 'slow\_ptr'. When 'fast\_ptr' reaches the end of the list,

// 'slow ptr' is now at the splitting point so we create the second list starting after 'slow\_ptr'.

fast ptr->next = head2; // End of first list is now connected to the start of the second.

slow\_ptr->next = head; // End of second list is now connected to the start of the first.

ListNode \*fast\_ptr = head; // 'fast\_ptr' will be used to find the end of the list.

// If there are an even number of nodes, move 'fast\_ptr' to the end of the list

// The list is now split into two, with 'head2' as the head of the second list

// This function takes a circular singly—linked list and splits it into two halves.

// If the list is empty, just return an array with two null elements.

let slowPointer: ListNode | null = head; // This will move one step at a time.

function splitCircularLinkedList(head: ListNode | null): Array<ListNode | null> {

# so we will start the second half from the next node

# Move pointers until the fast pointer reaches the end of the list

while fast pointer.next != head and fast\_pointer.next.next != head:

# If the fast pointer is one step away from completing the cycle, move it once more

# The slow pointer should point to the head of the first half, completing the first circular list

**Python** 

class ListNode:

**Example Walkthrough** 

our circular linked list looks like this:

(Head)  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$ 

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

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

- These steps demonstrate how the original circular linked list was split into two halves while ensuring that both sublists remained circular.
- self.val = val self.next = next class Solution: def splitCircularLinkedList(self, head: Optional[ListNode]) -> List[Optional[ListNode]]: # Fast pointer for moving two steps at a time

#### second\_half\_head = slow\_pointer.next # The fast pointer should now point to the second half, making it a circular list fast\_pointer.next = second\_half\_head

Java

class Solution {

class Solution {

public:

```
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};
C++
#include <vector>
// Definition for a node in a singly-linked list.
struct ListNode {
                       // Value of the node.
    int val;
   ListNode *next; // Pointer to the next node.
   ListNode(): val(0), next(nullptr) {}
                                                                 // Default constructor.
                                                                // Constructor initializing with a value.
   ListNode(int x) : val(x), next(nullptr) {}
   ListNode(int x, ListNode *next) : val(x), next(next) {} // Constructor initializing with a value and the next node.
```

# let fastPointer: ListNode | null = head; // This will move two steps at a time.

**TypeScript** 

if (!head) {

```
// Move through the list to find the middle
   // Fast pointer moves twice as fast as the slow pointer.
   // When the fast pointer reaches the end, slow pointer will be at the middle.
   while (fastPointer.next !== head && fastPointer.next.next !== head) {
        slowPointer = slowPointer.next;
        fastPointer = fastPointer.next.next;
   // In case of an even number of elements
   // move the fast pointer one more step to reach the end of the list.
   if (fastPointer.next !== head) {
        fastPointer = fastPointer.next;
   // Now, slowPointer is at the end of the first half of the list.
   // The node following slowPointer starts the second half.
   const secondHalfHead: ListNode | null = slowPointer.next;
   // Split the list into two halves.
   // The first half will end at the slowPointer and should circle back to the head
    slowPointer.next = head;
   // The second half starts at secondHalfHead and will end at the fastPointer.
   fastPointer.next = secondHalfHead;
   // Return the two halves of the list.
   return [head, secondHalfHead];
// Definition for singly-linked list (provided for context).
class ListNode {
   val: number;
   next: ListNode | null:
    constructor(val?: number, next?: ListNode | null) {
       this.val = (val === undefined ? 0 : val);
       this.next = (next === undefined ? null : next);
# Definition for singly-linked list.
class ListNode:
   def init (self, val=0, next=None):
       self.val = val
       self.next = next
class Solution:
   def splitCircularLinkedList(self, head: Optional[ListNode]) -> List[Optional[ListNode]]:
       # Fast pointer for moving two steps at a time
       fast pointer = head
       # Slow pointer for moving one step at a time
       slow_pointer = head
       # Move pointers until the fast pointer reaches the end of the list
       while fast pointer.next != head and fast_pointer.next.next != head:
            slow pointer = slow pointer.next
            fast_pointer = fast_pointer.next.next
       # If the fast pointer is one step away from completing the cycle, move it once more
       if fast pointer.next != head:
            fast_pointer = fast_pointer.next
       # The slow pointer now points to the middle of the list,
       # so we will start the second half from the next node
```

# 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 and Space Complexity

slow\_pointer.next = head

return [head, second\_half\_head]

second\_half\_head = slow\_pointer.next

fast\_pointer.next = second\_half\_head

# Return the two halves, both are now circular linked lists

# The fast pointer should now point to the second half, making it a circular list

# The slow pointer should point to the head of the first half, completing the first circular list

**Time Complexity:** The time complexity of this algorithm can be determined by analyzing the while loop, which continues until the fast pointer (b)

## • 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.

**Space Complexity:** 

• If the list has n nodes, and since the fast pointer moves two steps at a time, it will take 0(n/2) steps for the fast pointer to traverse the entire list. Therefore, the time complexity of the loop is O(n).

- 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 0(1).

has either completed a cycle or is at the last node.