Problem Description

Objective:

Given the head of a singly linked list, determine the node where a cycle begins. If there is no cycle, return null.

Definitions:

- **Cycle in Linked List:** A cycle exists in a linked list if there is a node in the list that can be revisited by continuously following the next pointers.

- **Node Index (pos):** Internally, pos denotes the index of the node to which the tail's next pointer is connected (0-indexed). If there is no cycle, pos is -1. This pos is used for example explanation but is not passed as a parameter to the function.

Example 1:

- **Input:** head = [3,2,0,-4], pos = 1
- **Output:** tail connects to node index 1
- **Explanation:** The linked list has a cycle, with the tail connecting to the node at index 1.

Example 2:

- **Input:** head = [1,2], pos = 0
- **Output:** tail connects to node index 0
- **Explanation:** The linked list has a cycle, with the tail connecting to the node at index 0.

Example 3:

- **Input:** head = [1], pos = -1
- **Output:** no cycle
- **Explanation:** The linked list has no cycle.

Constraints

- Number of nodes in the list: [0, 10^4].
- Node values: -10^5 <= Node.val <= 10^5.
- pos is either -1 or a valid index in the linked list.

Follow-up: Can you solve the problem using O(1) (constant) memory?

Approach

The solution uses Floyd's Tortoise and Hare algorithm to detect the cycle and find its starting node. This algorithm employs two pointers moving at different speeds to identify the presence of a cycle.

Steps:

1. **Initialization:** Use two pointers, slow and fast, both initially pointing to the head of the linked list.

2. **Cycle Detection:**
   - Move the slow pointer one step at a time.
   - Move the fast pointer two steps at a time.
   - If the two pointers meet, a cycle is detected.

3. **Finding the Cycle Start:**
    - After detecting the cycle, initialize another pointer entry at the head.
    - Move both entry and slow pointers one step at a time until they meet. The meeting point is the start of the cycle.

4. **No Cycle Detected:** If the fast pointer reaches the end of the list (fast or fast.next becomes null), there is no cycle.

Explanation

- **Edge Case Handling:** The function first checks if the list is empty or has only one node, in which case no cycle can exist.

- **Cycle Detection:** The slow pointer moves one step and the fast pointer moves two steps in each iteration. If they meet, a cycle is present.

- **Finding Cycle Start:** After detecting the cycle, a new pointer entry is used to find the start of the cycle by moving both entry and slow pointers one step at a time until they meet.

This solution effectively uses O(1) memory and has a time complexity of O(n), where n is the number of nodes in the list.