Problem Link:

https://leetcode.com/problems/linked-list-cycle/

Problem Description:

Given `head`, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter**. Return `true` *if there is a cycle in the linked list*. Otherwise, return `false`.

Problem Approach:

We will use **Floyd's Cycle Detection Algorithm** (Tortoise and Hare Algorithm) to check for cycles in the list by employing two pointers that move at different speeds.

Solution:

We start by initializing two pointers slow and fast as the head of the list. Now, we iterate through the list until fast is not none, or fast.next is not none. If fast or fast.next is None, this means we have reached the end of the list, and no cycle exists. The idea is:

- slow moves one node at a time.
- fast moves two nodes at a time.

If there's a cycle, slow and fast will eventually meet. If not, fast will reach the end of the list.
So, we move slow one step by slow = slow.next, while we move fast two steps by fast = fast.next.next. If we find both slow and fast pointers pointing to the same node, we return True as there's a cycle in the linked list.

Otherwise, if we break the loop with any of the conditions, then it means that we've reached the end of the linked list, and hence, we should return False. This algorithm uses O(n) time and O(1) space complexity, and is hence, efficient.

Code (Python):

```
def hasCycle(self, head: Optional[ListNode]) -> bool:
        slow = head
        fast = head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False
```