Solution Review: Detect Loop in a Linked List

This review provides a detailed analysis of the different ways to solve the Detect a Loop in a Linked List challenge.

We'll cover the following

Solution: Floyd's Cycle-Finding Algorithm

main.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList
# Floyd's Cycle Finding Algorithm
def detect_loop(lst):
    # Keep two iterators
    onestep = lst.get_head()
    twostep = lst.get_head()
    while onestep and twostep and twostep.next_element:
        onestep = onestep.next_element  # Moves one node at a time
        twostep = twostep.next_element.next_element  # Skips a node
        if onestep == twostep:  # Loop exists
            return True
```

```
12      return False
13
14  # ----------------------
15
16
17  lst = LinkedList()
18
19  lst.insert_at_head(21)
20  lst.insert_at_head(14)
21  lst.insert_at_head(7)
22
23  # Adding a loop
24  head = lst.get_head()
25  node = lst.get_head()
26
27  for i in range(4):
28      if node.next_element is None:
```

This is perhaps the fastest algorithm for detecting a linked list loop. We keep track of two iterators, `onestep` and `twostep`.

`onestep` moves forward one node at a time, while `twostep` iterates over two nodes. In this way, `twostep` is the faster iterator.

By principle, if a loop exists, the two iterators will meet. Whenever this condition is fulfilled, the function returns `True`.

Time Complexity#

We iterate the list once, which makes the total running time of this solution O(n).

> **Note:** The solution above has another approach that uses sets. We will cover that approach in Hashing Chapter: Challenge 10

In the next lesson, we'll figure out a way to find the middle n[illegible] list.