People follow many approaches to detect loop in linked list. Here I'm going to discuss about two approaches. Both approaches can be used on a singly linked list or doubly linked list.

1. Traverse & Mark Visited Nodes
2. Floyd's Cycle Finding Algorithm

Traverse & Mark Visited Nodes: This is the most trivial approach. We need to start from the **head** of the linked list and traverse through it. Each time we visit a node, we **mark** it a s visited and go forward. If any time we hit the end of the list, we conclude there is no loop. If we hit an already **marked** node, we conclude there is a loop in the linked list. This approach has a time complexity of **O(n)** and space complexity **O(n)**.

Floyd's Cycle Finding Algorithm: Floyd's Cycle Finding algorithm is also known as **tortoise and hare** algorithm. It uses two pointer for finding the loop in linked list. One **fast** pointer and one **slow** pointer. Both pointers are used to traverse the linked list starting from the **head**. The **slow** pointer (also known as **tortoise**) moves one step along the linked list. The **fast** pointer (also known as **hare**) moves two steps at a time along the linked list. Any time the **fast** pointer reaches the end of the list, we return "**no loop**". But if at some point both the pointer points to the same node we conclude there is a loop in the linked list. Why is that? Lets dig deeper for explanation.

There are two possible cases here:

- There is no loop in the linked list. Since the **fast** pointer moves faster than the **slow** pointer, it will hit the end of the list. So we conclude there is no loop.
- There is a loop in the linked list. In this case there is no end of the list. Both the **slow** and **fast** pointer will enter the loop and loop there forever. But since the **fast** pointer moves two step at a time, at some point it will try to overtake the **slow** pointer from the back and both will point to same node. It is like the popular **tortoise vs hare** race. If they run in a circular lap, the **hare** will overtake the **tortoise** over and over (check the image below). So any time both the pointers are on the same position, we conclude there is a loop in the linked list.

fast pointer will overtake slow pointer in a loop

Let's see the pseudo code for **Floyd's** algorithm.

1. Initialize **slow** and **fast** with linked list head
2. If **fast** reaches end of the list, return "**no loop"**
3. Else forward **fast** by one node
4. If **fast** reaches end of the list, return "**no loop**"
5. Else move both **slow** and **fast** pointer by one node
6. If **slow** and **fast** points to same node, return "**loop found**"
7. Else go to Step 2.

This approach to detect loop in linked list takes **O(n)** time and **O(1)** space. So in terms of space, it is better than the trivial 1st approach of marking the visited nodes.

Loop Size: So Floyd's algorithm helped us to detect loop in linked list. Now how do we determine the size of the loop? This is also easier and can be achieved in **O(n)** time. Just after the completion of above loop finding algorithm, now we have to keep the **fast** pointer still and move the **slow** pointer by one step until it hit the **fast** pointer again. As the **slow** pointer moves, we'll count the number of nodes touched by it. This count will give the size of the loop in the linked list.

Start of the Loop: We have detected a loop in the linked list, we've found out the size of the loop. But we don't know the start position of the loop. After the loop detection algorithm, if a loop exists, we can find out the start of the loop in **O(n)** time. To detect starting of the loop, we move the **slow** pointer at the beginning of the linked list. The **fast** pointer will remain in its position after the loop detection. Next we move each pointer by one node at a time. If at some point they both point to the same node, this node will be the start node of the loop. We can prove it mathematically.

Let, distance from head to start of loop is X. Length of the loop is Y + Z. When **slow** and **fast** meet, **slow** covers (X + Y) distance and **fast** covers X + (Y+Z) + Y distance

As **fast** is two times faster than **slow** pointer, so $2(X + Y) = X + (Y+Z) + Y$

Solving this equation we get, $X = Z$.

So after loop detection, if **slow** starts traversing from the head and if both of them are moved by one node at a time, both of them will pass $X = Z$ distance and meet at the start node of the loop.