# **Cycles and Linked Lists**

* Cycle Problems are common when dealing with linked lists.
* Most often, we will use the two-pointer slow-fast (“Hare-Tortoise) technique to solve them.
* We have two pointers:

1. A slow pointer
2. A fast pointer

* We check for equality of the two to determine if there is a loop.
* Usually we check if there is some terminating condition, such as the fast pointer being null to determine if there is no loop.

# **Check IF Cycle Exists in Linked List**

**Floyd’s Cycle Finding Algorithm (“Hare-Tortoise Algorithm”)**

* Floyd’s Cycle detection algorithm or Hair Tortoise algorithm is used to detect if there is a **cycle** (loop) in a linked list.
* Floyd’s Cycle Finding Algorithm is a two-pointer algorithm
  + One pointer: slow that moves 1 step
  + One pointer: fast that moves 2 steps
* While traversing the linked list one of these things will occur:

1. The Fast pointer may reach the end of the list.
   1. The fast pointer reaching the end of list means that it itself is null or its next pointer is null.
   2. This shows that there is no loop in the linked list.
2. The Fast pointer catches the slow pointer at some time therefore a loop exists in the linked list.

**Example 1**

Diagram

Description automatically generated

**Input:** head = [3,2,0,-4], pos = 1

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

This linked list very clearly has a cycle because there is no node pointing to null, and we can see the last node loop back earlier into the list.

**Example 2**

A picture containing diagram

Description automatically generated

**Input:** head = [1], pos = -1

**Output:** false

**Explanation:** There is no cycle in the linked list.

A linked list with a single node cannot have any cycles.

**Code**

The code looks very similar to if we were trying to find the middle of a list.

Since the fast pointer moves twice as fast as the slow pointer, there will come a time where they will overlap.

ListNode slow = head;

ListNode fast = head;

while(fast != null && fast.next != null) // accounts for even and odd size

{

slow = slow.next;

fast = fast.next.next;

if(slow == fast) return true;

}

return false;

If we satisfy the loop condition, that means we have found a null pointer, which means is there a point in the list that terminates (meaning there is no loop).

# **Check WHERE Cycle is in Linked List**

**Problem Statement**

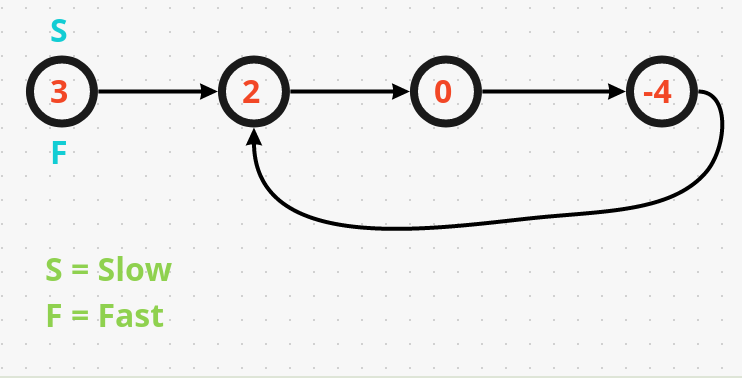
Given the head of a linked list, return the node *where* the cycle begins.

If there is no cycle, return null

**Do not modify** the linked list.

**Example 1:**

**head = [3, 2, 0, -4], pos = 1**



**Algorithm**

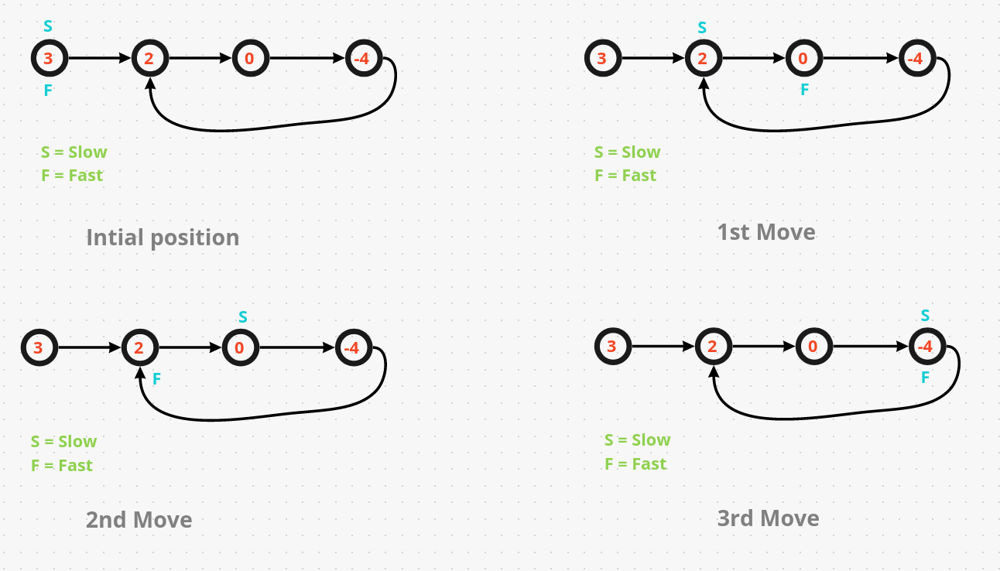
**Step 1: Determine if there is a cycle**

1.1) Using a slow pointer that move forward 1 step each time

1.2) Using a fast pointer that move forward 2 steps each time

1.3) If the slow pointer and fast pointer both point to the same location after several moving steps, there is a cycle.

1.4) Otherwise, if (fast->next == NULL || fast->next->next == NULL), there has no cycle.



**Step 2: If there is a cycle, return the entry location of the cycle**

2.1) **L1** is defined as the distance between

the head pointer

and

the start of the cycle

2.2) **L2** is defined as the distance between

the start of the cycle

and

the meeting point of the slow and fast pointers

2.3) **L3** is defined as the distance between

the meeting point of the slow and fast pointers

and

the start of the cycle



Diagram

Description automatically generated



**According to the definition of L1, L2 and L3, we can obtain:**

2.4) Distance traveled by the slow pointer when the slow and fast pointers meet:

**L1+L2**

2.6) Distance traveled by the fast pointer when the slow and fast pointers meet:

**L1+L2+L3+L2**

2.7) We can set these two equations equal to each other to find a relationship between L1 and L3.

We know the total distance traveled by the fast pointer is **L1+L2+L3+L2**

We also know that the total distance traveled by the slow pointer is **L1+L2**

Remember, that we had two pointers. A slow pointer that moves one step and fast pointer that moves two steps at a time. This means that the fast pointer traveled twice the distance of the slow pointer.

Because we also know that the fast pointer traveled twice as fast as the slow pointer, we can say that if we double the distance the slow pointer traveled, it should be equal to the total distance the fast pointer traveled.

Therefore, we can derive:

**L1+L2+L3+L2 = 2(L1+L2)**

Now, we can simplify:

**L1+L2+L3+L2 = 2L1+2L2** (take away the L2’s from each side)

**L1+L3 = L1** (take away the L1’s from each side)

**L1 = L3**

We have found that

the distance between the head pointer and start of the cycle

is **equal to**

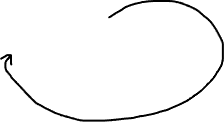
the distance between the slow and fast pointer meeting point and the start of the cycle

**For Example:**



Diagram

Description automatically generated



Fast Pointer Distance Traveled = **L1+L2+L3+L2**

Fast Pointer Distance Traveled = 6



Diagram

Description automatically generated

Slow Pointer Distance Traveled = **L1+L2**

Slow Pointer Distance Traveled = 3

We can see that the distance traveled by the fast pointer is double the distance traveled by the slow pointer.

**Keeping a third pointer to the head**

We can define a pointer "entry" that point to the head of the linked list.

When the slow pointer and the fast pointer encounter in the cycle, this "entry" pointer and either the slow or fast pointer move one step at a time at the same time.

entry = entry.next;

slow = slow.next;

This works because we know that

the distance between the head pointer and start of the cycle

is **equal to**

the distance between the slow and fast pointer meeting point and the start of the cycle

Therefore, if we check for when the entry and slow pointer meet

while(slow != entry)

Then we know that they will meet at the start of the cycle

**Code**

public ListNode detectCycle(ListNode head) {

ListNode slow = head;

ListNode fast = head;

ListNode entry = head;

while(fast != null && fast.next != null)

{

slow = slow.next;

fast = fast.next.next;

if(slow == fast)

{

while(slow != entry)

{

slow = slow.next;

entry = entry.next;

}

return entry;

}

}

return null;

}

**Runtime**

This solution runs in O(N) time with O(1) space.

# **Happy Numbers**

* Cycles are not allows associated with linked list traversal.
* The Happy Numbers problem, for example, has a different KIND of loop.
* We are not traversing over nodes, but rather, traversing over the cycle of values for the sum of the squares of a number.
* Surprisingly, we can also apply Floyd’s Cycle Detection Algorithm in this problem.
* The format of the code looks similar:

public ListNode detectCycle(int num) {

int slow = num, fast = num;

while(true) {

slow = squareSum(slow);

fast = squareSum(squareSum(fast));

if(slow == fast) { // found cycle

break;

}

}

return slow == 1;

}