Topic Introduction

Welcome back to your daily PrepLetter! Today, we're diving into a deceptively simple yet powerful algorithmic technique: **Floyd's Cycle Detection Algorithm**, also known as the **Tortoise and Hare Algorithm**.

What is Floyd's Cycle Detection?

At its heart, Floyd's algorithm is a clever way to detect cycles in sequences without using extra space. Whether it's a linked list, an array of numbers, or even a chain of mathematical operations, this approach uses two pointers moving at different speeds to sniff out cycles. The "tortoise" moves one step at a time, while the "hare" hops two steps. If there's a cycle, they'll eventually meet inside it—like two runners on a circular track.

Why is this useful in coding interviews?
Cycle detection shows up in many disguises: linked lists, number transformations, and arrays where elements point to other indices. Recognizing when to use this trick can save you time, space, and headaches.

Simple example:
Imagine you have a linked list. You want to check: "Does this list loop back on itself?" Instead of keeping a set of visited nodes (which uses extra memory), just use our two pointers. If they ever meet, you found a cycle!

Now, let's see this technique in action through three classic LeetCode problems. We'll start with the most direct use case, then build up to more abstract applications.

Problem 1: Linked List Cycle

[LeetCode 141: Linked List Cycle]

Problem Statement (in plain English):
Given the head of a singly linked list, determine if the list contains a cycle. In other words, does any node in the list point back to a previous node, forming a loop?

Example:
Suppose your linked list looks like this:

```
3 -> 2 -> 0 -> -4
     ^         |
     |--------<-|
```

Here, the next pointer of `-4` points back to node `2`.
Your function should return `True` (cycle detected).

Another example:

```
1 -> 2 -> 3 -> None
```

No cycles here, so return `False`.

Take a moment to try solving this on your own before reading the solution.

Solution: Floyd's Tortoise and Hare

Instead of tracking all visited nodes, use two pointers:

- **slow** (moves 1 step at a time)
- **fast** (moves 2 steps at a time)

If there's a cycle, they *must* meet. If fast reaches the end (`None`), there's no cycle.

Python Code

```
class Solution:
    def hasCycle(self, head):
        # Initialize two pointers, both at the head
        slow = head
        fast = head

        while fast and fast.next:
            slow = slow.next          # move slow by 1
            fast = fast.next.next     # move fast by 2

            # If slow and fast meet, there is a cycle
            if slow == fast:
                return True

        # If we reach here, fast hit the end — no cycle
        return False
```

Complexity:

- **Time:** O(N), where N is the number of nodes. Each pointer traverses at most N nodes.
- **Space:** O(1), no extra space used.

Let's see it in action:

- For this list: `1 -> 2 -> 3 -> None`, fast will hit the end.
- For this list: `1 -> 2 -> 3 -> 2 ... (cycle)`, slow and fast will eventually meet at node `2`.

Try this test case yourself:

`head = [1]` where `head.next = head` (a single node that points to itself). What would the function return?

Problem 2: Find the Duplicate Number

[LeetCode 287: Find the Duplicate Number]

Problem Statement (in plain English):

Given an array of `n+1` integers where each integer is between `1` and `n` (inclusive), and only one repeated number exists, find that number. The catch: you must not modify the array, and you can only use constant extra space.

Example:

Input: `[1, 3, 4, 2, 2]`

Output: `2`

Another example:

Input: `[3, 1, 3, 4, 2]`

Output: `3`

What's different from the previous problem?

Although it's an array, think of `nums[i]` as pointing to the next index. The duplicate creates a "cycle" in these pointers. The job is to find the entry point of this cycle!

Take a moment to try solving this on your own before reading the solution.

Solution: Floyd's Cycle Detection in Arrays

- Treat each value as a pointer to the next index. Unlike the previous problem, we move the slow pointer to `nums[slow]` and the fast pointer to `nums[nums[fast]]`.
- Start both `slow` and `fast` at index 0.
- Move `slow` by one step (`slow = nums[slow]`).
- Move `fast` by two steps (`fast = nums[nums[fast]]`).
- When they meet, reset one pointer to start, and move both by one step at a time. The intersection point is the duplicate.

Why does this work?

The duplicate number causes a cycle in the sequence of jumps. Just like finding a cycle in a linked list!

Python Code

```
class Solution:
    def findDuplicate(self, nums):
        # Phase 1: Find the intersection point
        slow = nums[0]
        fast = nums[0]
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break
```

```
        # Phase 2: Find the entrance to the cycle
        slow = nums[0]
        while slow != fast:
            slow = nums[slow]
            fast = nums[fast]
        return slow
```

Complexity:

- **Time:** O(N)
- **Space:** O(1)

Let's walk through example `[1, 3, 4, 2, 2]`:

- Start: slow = 1, fast = 1
- Move: slow = 3, fast = 2
- Move: slow = 2, fast = 2
- They meet at index 2 (value 4 or 2 depending on moves), then reset slow to start and move both one step at a time to find the duplicate.

Try this test case yourself:
`[1, 3, 4, 5, 1]`
What will your solution return?

Now think about the following
- What would be a better solution if you could modify the array?
- What would be your solution if you could use extra space?
- What would be your solution for Poverty? (Ahh, Off-Topic, Sorry!)

Problem 3: Happy Number

LeetCode 202: Happy Number

Problem Statement (in plain English):
A number is "happy" if, by repeatedly replacing it with the sum of the squares of its digits, you eventually reach 1. If you fall into a loop that never reaches 1, the number is "unhappy". Write a function to decide if a number is happy.

Example:
Input: `19`
Sequence: $1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$
So, return `True`.

Another example:

Input: `2`

You'll get stuck in a loop: 2 -> 4 -> 16 -> 37 -> 58 -> 89 -> 145 -> 42 -> 20 -> 4 ...

How does this differ from the previous problems?

Instead of nodes or array indices, the "next" value is built from digit operations. But the logic is the same: if you see a value again, you're in a cycle.

Take a moment to try solving this on your own before reading the solution.

Solution: Floyd's Cycle Detection on Number Transformations

- Use two variables, both starting at `n`.
- At each step, move `slow` by one transformation, `fast` by two.
- If `fast` reaches 1, return True.
- If `slow` equals `fast` (and not 1), a cycle exists — return False.

Python Code

```
class Solution:
    def isHappy(self, n):
        def get_next(num):
            # Returns the sum of the squares of the digits of num
            total = 0
            while num > 0:
                digit = num % 10
                total += digit * digit
                num //= 10
            return total

        slow = n
        fast = get_next(n)
        while fast != 1 and slow != fast:
            slow = get_next(slow)
            fast = get_next(get_next(fast))
        return fast == 1
```

Complexity:

- **Time:** The sequence always falls into a cycle or reaches 1, so it's O(1) in practice (since numbers get small quickly).
- **Space:** O(1) (no extra data structures).

Let's dry run on `n = 2`:

- slow = 2, fast = get_next(2) = 4
- slow = get_next(2) = 4, fast = get_next(get_next(4)) = get_next(16) = 37
- slow = 16, fast = get_next(get_next(37)) = get_next(58) = 89
- Continue... they will eventually meet in the unhappy cycle.

Try this test case yourself:

Is `7` a happy number?

Is `2025` a happy number?

What about `0`? (Although the LC problem doesn't have that constraint but what stops us from thinking outside the box? We can even think about how can a number be happy? Do they even have Feelings?!)

As you may have guessed, this could also be solved using a set to track seen numbers. Try implementing that after you finish!

Summary and Next Steps

Today, we explored how **Floyd's Cycle Detection Algorithm** elegantly solves problems involving cycles in linked lists, arrays, and even number operations. The magic is in using two pointers at different speeds—no extra memory required!

Key takeaways:

- Always ask: "Is there a cycle here?" in disguise.
- This pattern appears in many forms: pointers, array indices, mathematical sequences.
- Avoid common mistakes:
  - Not moving the fast pointer by two steps.
  - Forgetting to check for termination conditions (e.g., reaching `None` in a linked list).
  - Assuming only linked lists can have cycles—cycles can be hidden in arrays and number transformations too!
- When optimal, try alternate methods (like sets or binary search), but understand the pattern first.
- Off-Topic: Looking at this line of the code- `get_next(get_next(fast))`, I think if Sheldon would write this code, he would write it 3 times!

Action list:

- Try solving all three problems yourself, using both the Floyd's technique and alternate approaches.
- Explore similar problems—search for "cycle detection" or "Floyd's algorithm" on LeetCode.
- Read other people's solutions for different perspectives and edge cases.
- If you get stuck, don't worry! Take a break, review the pattern, and try again tomorrow.

With every PrepLetter, you're leveling up. Keep practicing whether like a tortoise or a hare, stay curious, and remember: the tortoise and the hare are your friends in finding cycles wherever they hide!

Happy coding!