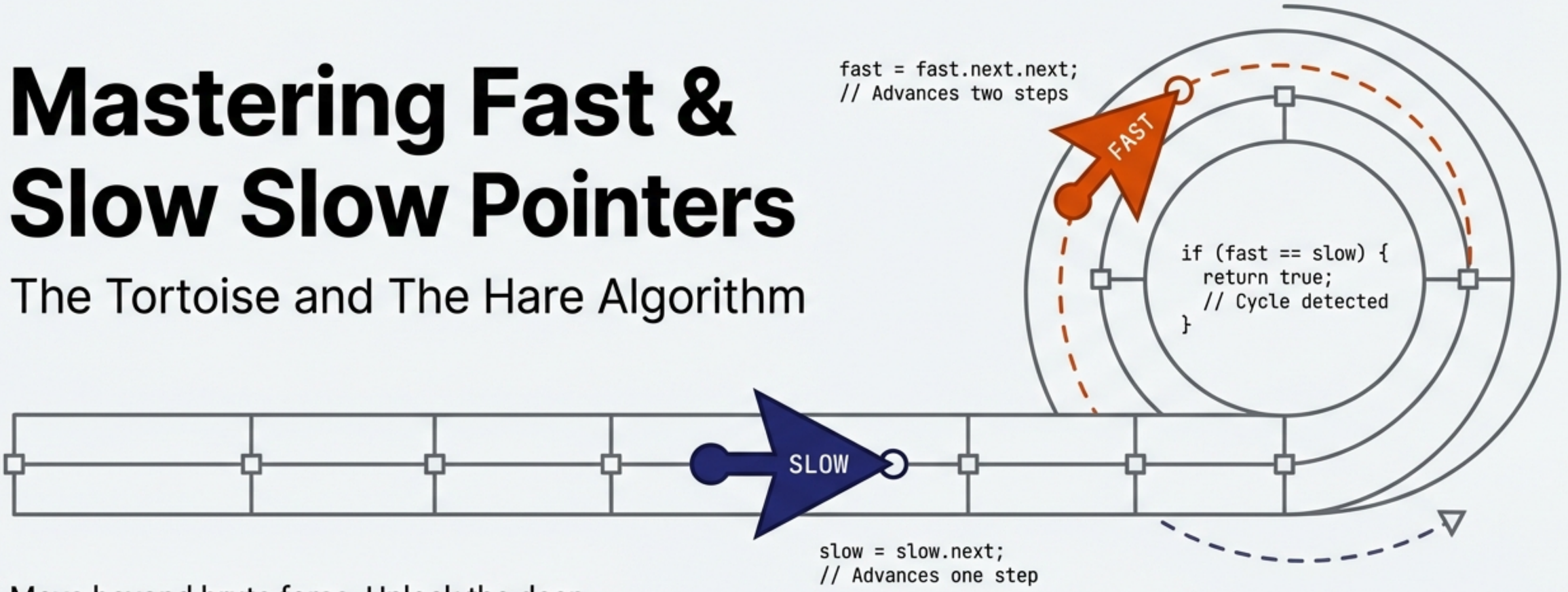
DAY 8/90 DSA PATTERN SERIES

Mastering Fast & Slow Slow Pointers

The Tortoise and The Hare Algorithm



Move beyond brute force. Unlock the deep mathematical proofs for Cycle Detection, Happy Numbers, and Middle Node finding to ace technical interviews.

Let L be the length of the linear part, C be the length of the cycle. Let *fast* and *slow* meet after T steps. Then *slow* has traveled $L + k_1 C + x$, and *fast* has traveled $L + k_2 C + x$, where k_1 and k_2 are number of full cycles and x is the distance inside the cycle. Since *fast* travels twice the speed,

$$2(L + k_1 C + x) = L + k_2 C + x,$$

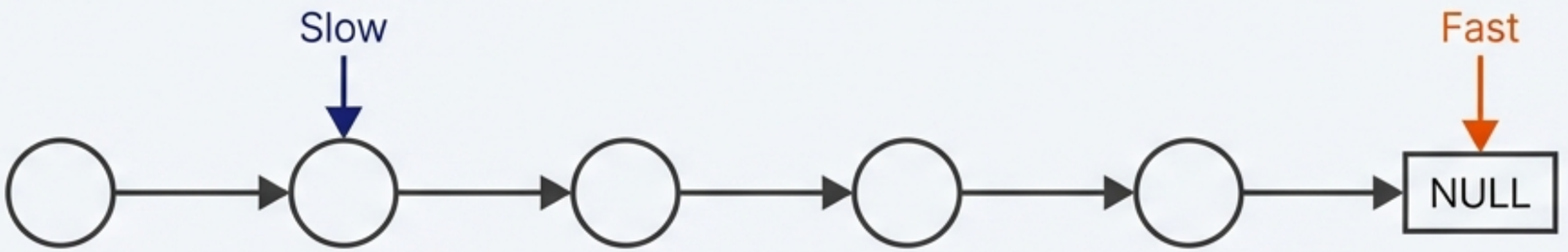
$$(L + x) = (k_2 - 2k_1)C$$

This proves they meet within the cycle if one exists.

Based on the technical analysis by CTO Bhaiya

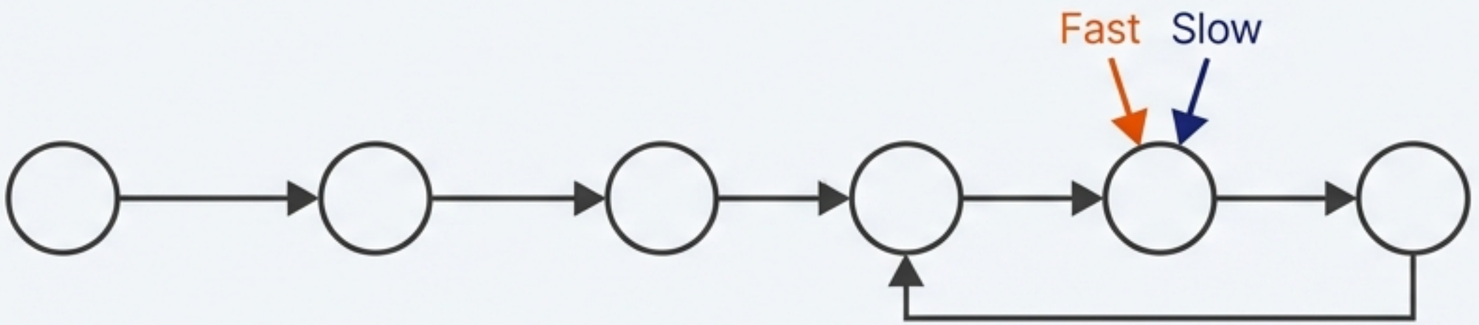
The Core Pattern: Floyd's Cycle-Finding Algorithm

Scenario A: Linear List (No Cycle)



Fast reaches NULL before Slow.
Termination guaranteed.

Scenario B: Cyclic List (Cycle Exists)



Fast laps Slow.
Distance shrinks by 1 node per step.
Collision guaranteed.

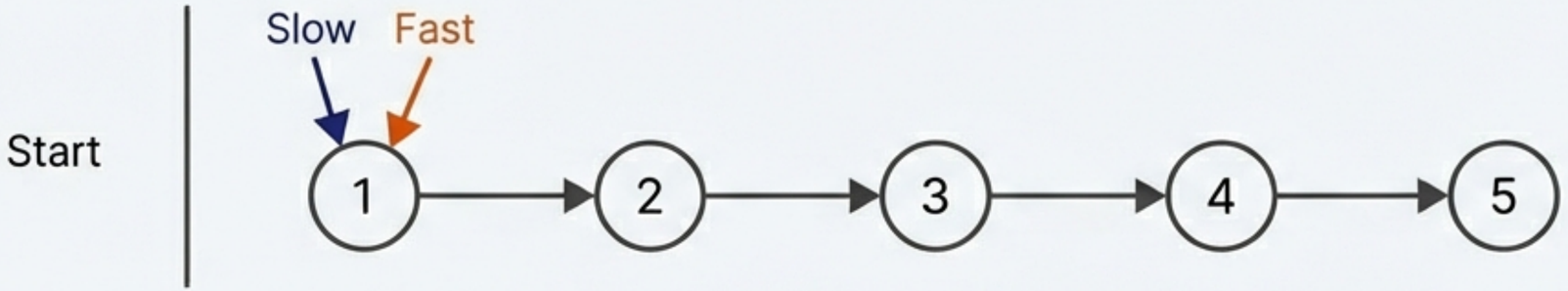
The Golden Rule (Safety Check):

```
while(fast != null && fast.next != null) { ...
}
```

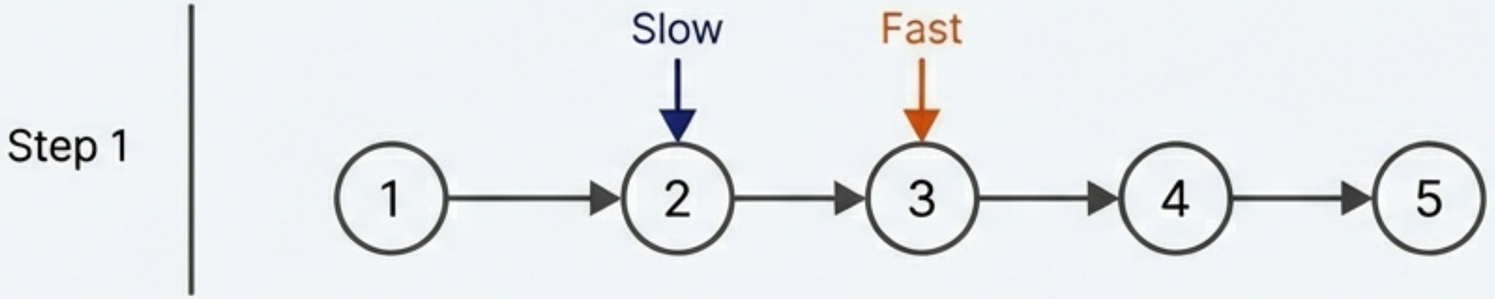
Prevents segmentation faults by ensuring the runway exists before sprinting.

Application 1: Middle of the Linked List (LeetCode 876)

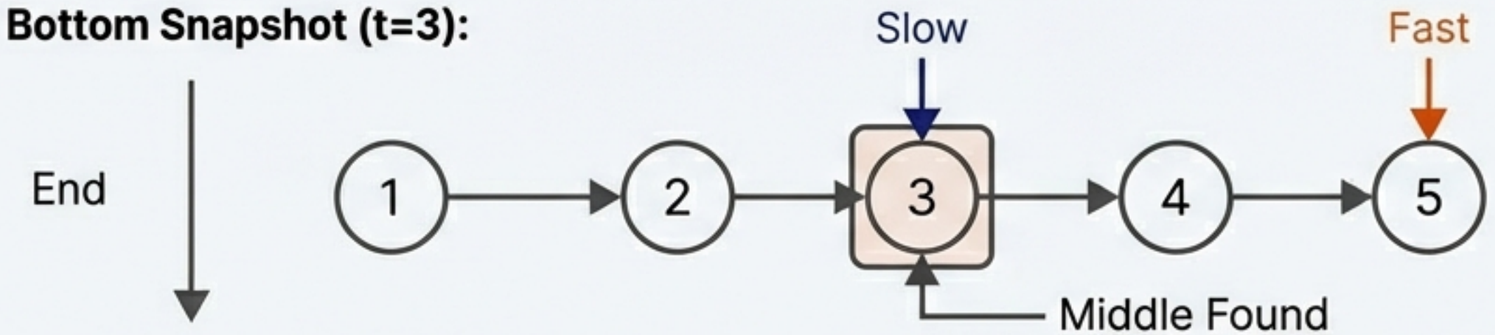
Top Snapshot (t=1):



Middle Snapshot (t=2):



Bottom Snapshot (t=3):



The Insight:

Speed Ratio: Fast = 2x, Slow = 1x.

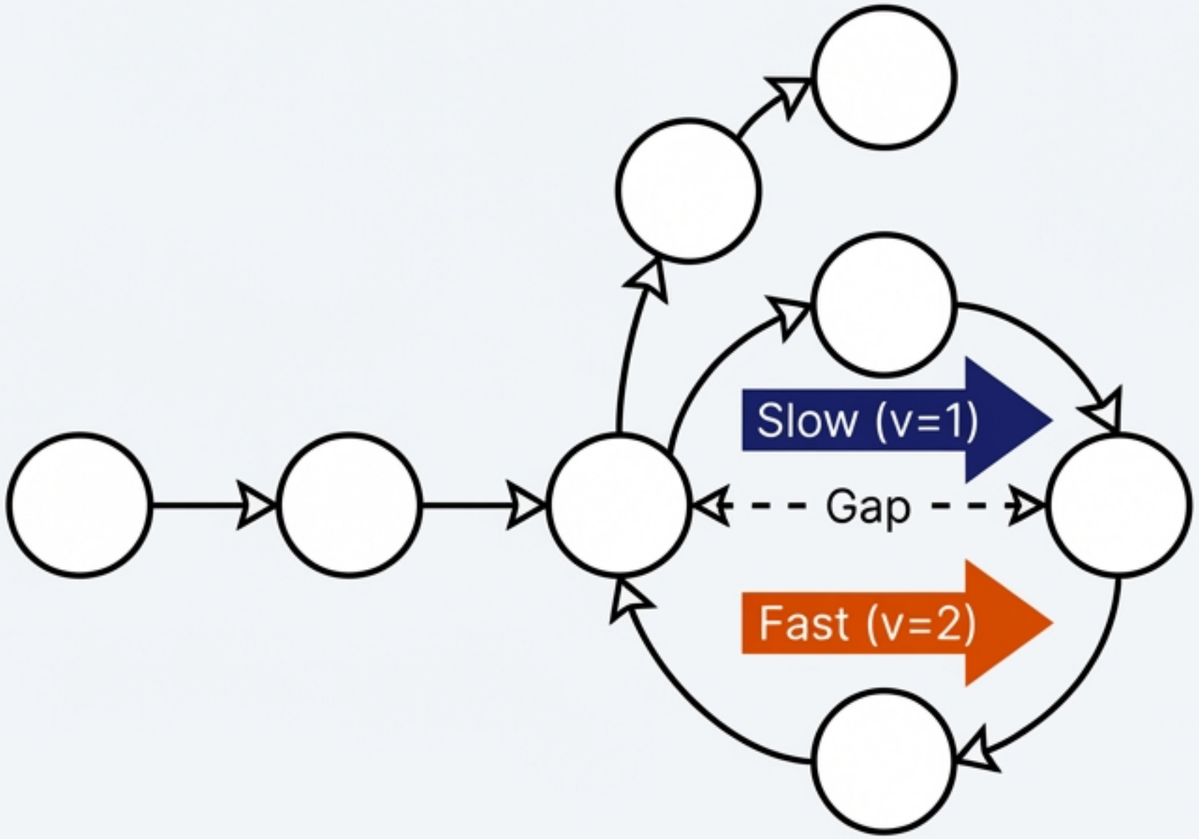
When Fast travels distance N (End), Slow travels N/2 (Middle).

Complexity:

Time: O(N) - Single Pass
Space: O(1) - No extra storage required

Edge Case: Even number of nodes (e.g., 6) → Algorithm lands on the second middle node.

Application 2: Linked List Cycle Detection (LeetCode 141)

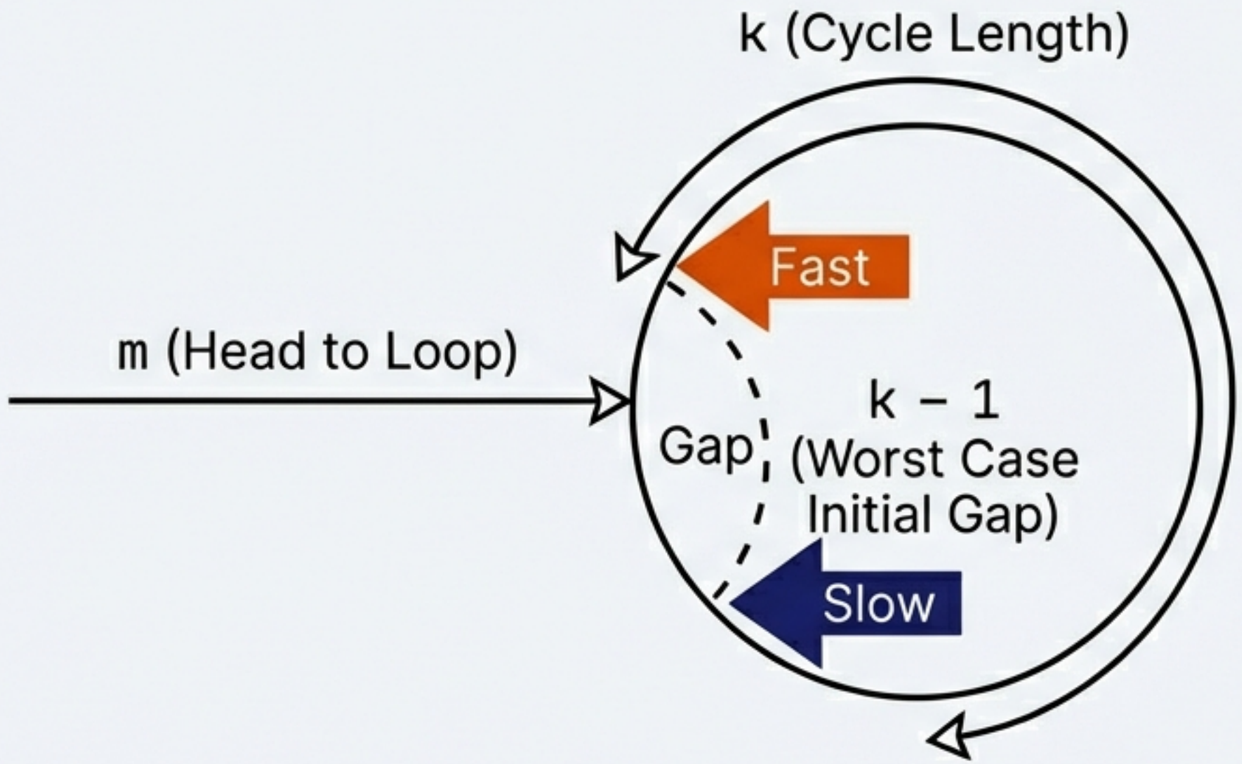


The Mechanics:

1. Initialize both at Head.
2. Loop: Slow moves 1 step, Fast moves 2 steps.
3. Condition: If `Fast == Slow` -> Return `True`.

Why it works (Relative Velocity):
Relative Speed = $2 - 1 = 1$ node per step. From the perspective of the Slow pointer, the Fast pointer is approaching it one step at a time. They cannot jump over each other; they must collide.

The Interviewer's Question: Prove Time Complexity is O(N)



Phase 1: Reaching the Loop
Takes "m" steps. Both pointers are now inside the cycle.

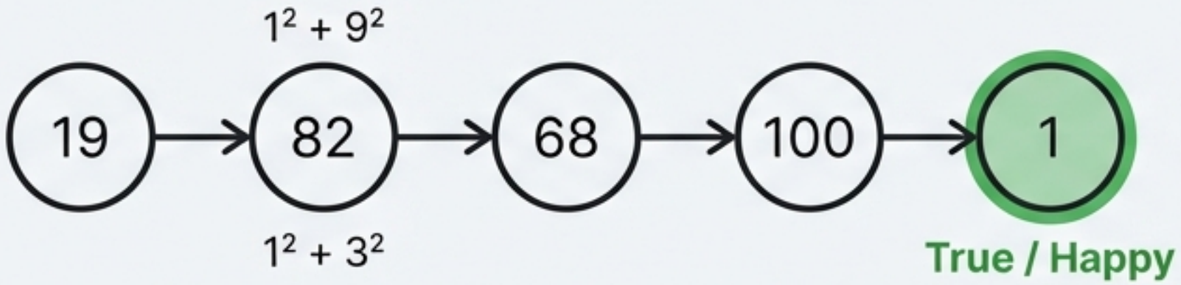
Phase 2: The Chase
Worst case initial gap: $k - 1$ steps.
Closing speed: 1 step per iteration.
Steps to meet: ~k steps.

Total Complexity:
Total Steps = m (entry) + k (chase)
Since $m + k = N$ (Total Nodes), Time Complexity is $O(N)$.
Space Complexity: $O(1)$ (No HashSets).

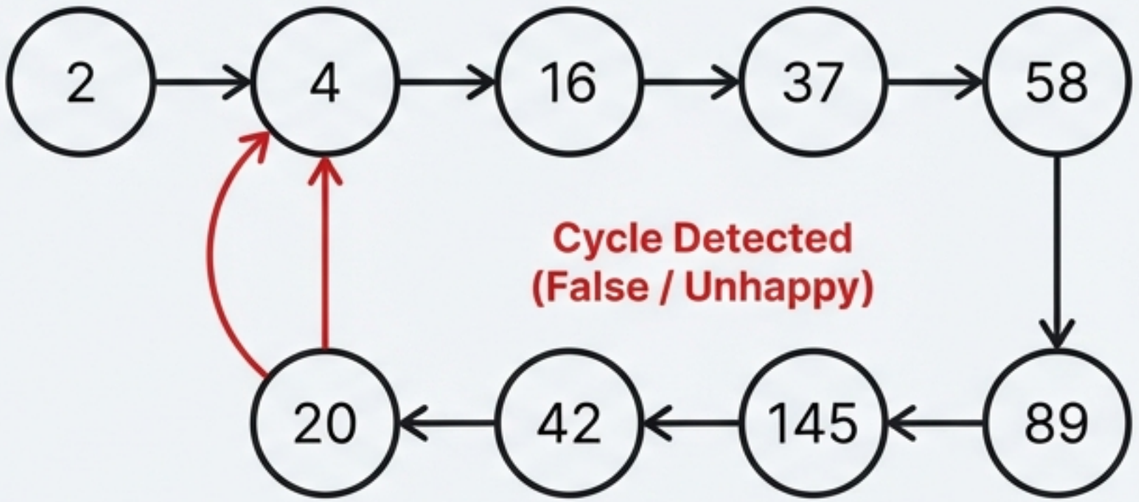
Application 3: Happy Number (LeetCode 202)

Detecting cycles in Number Theory

Happy Case



Unhappy Case



The Hidden Linked List:

- Treat the sequence of sum-of-squares as nodes.
- Use **Fast** & **Slow** pointers: If **Fast** reaches '1', return True. If **Fast** meets **Slow**, return False.

Deep Dive: Why doesn’t the number grow forever?

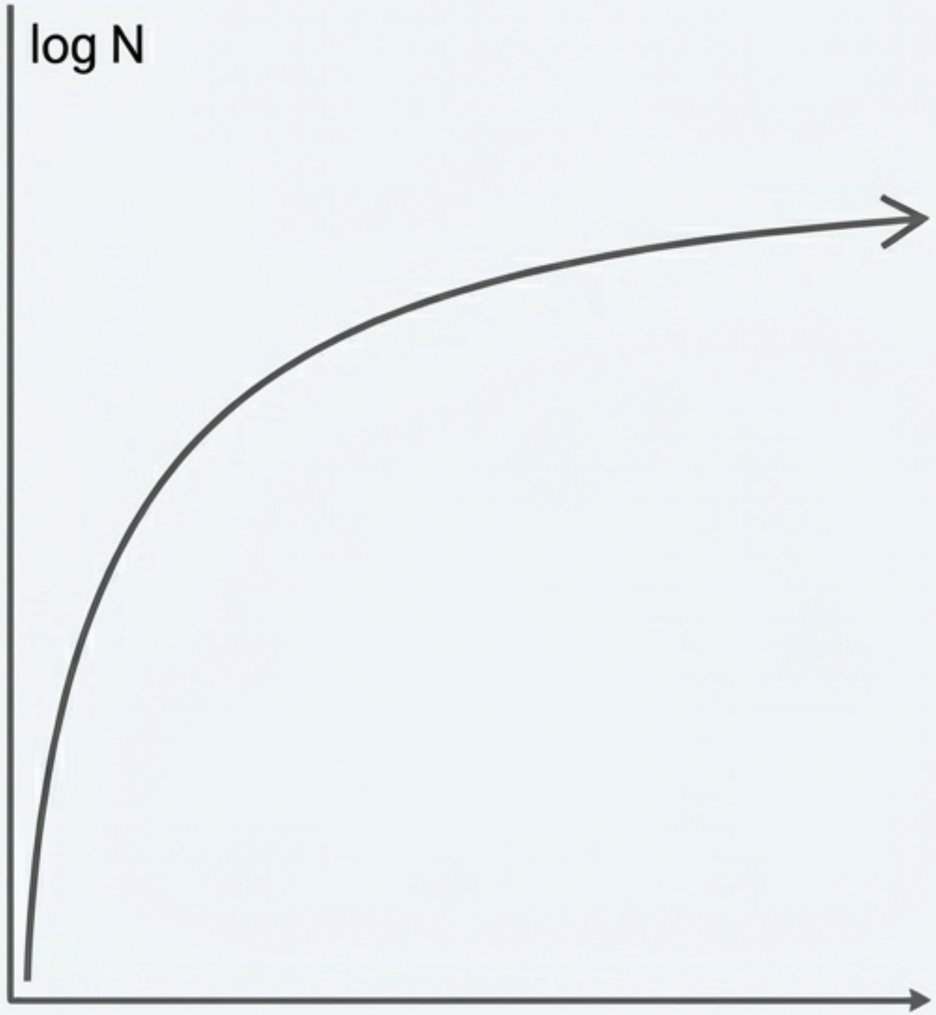
Proof Sheet

The Upper Bound Proof

1. Max Integer (Java): $\sim 2,147,483,647$ (10 digits)
2. Largest possible Next Step for 10 digits:
   Consider $9,999,999,999$
   Sum of Squares $= 9^2 \times 10 = 81 \times 10 = 810$
   Insight: Any number up to 2 Billion collapses to <= 810 immediately.
3. The Next Level (3 Digits):
   Max 3-digit number is 999.
   Sum of Squares $= 9^2 + 9^2 + 9^2 = 243$.

The Guarantee: Any sequence will quickly fall below 243. Once in this range, it must either reach 1 or repeat a number (Cycle) due to the Pigeonhole Principle. It cannot grow to infinity.

Happy Number: Complexity Analysis



Time Complexity

O(log N)

1. **Processing Digits:** Calculating sum of squares for numberN takes logarithmic time (digits = $\log_{10}N$).
2. **Sequence Length:** Once below 243, the sequence length is constant (max 243 steps).
3. **Result:** Finding the next number costs O(log N). The chain length is trivial.

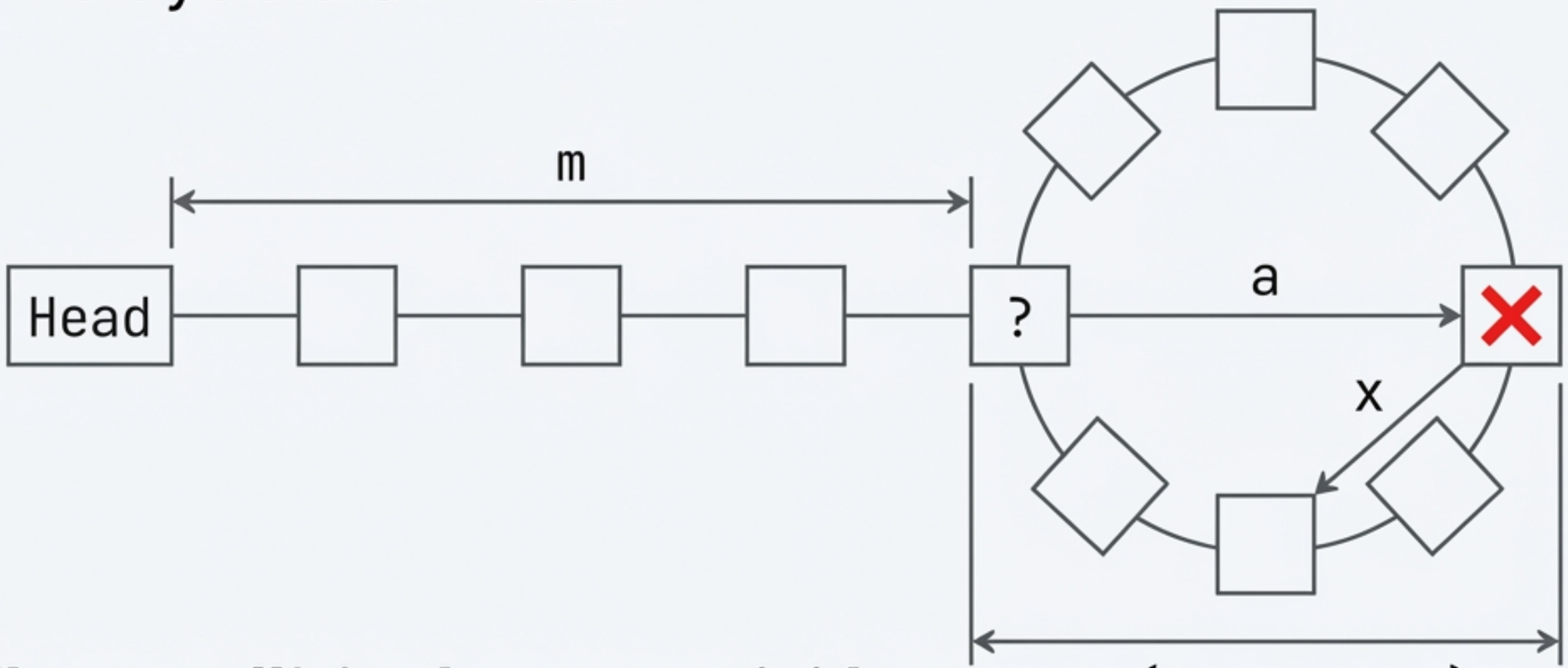
Space Complexity

O(1)

No HashSets or History Arrays required to detect the cycle.

Application 4: Linked List Cycle II (LeetCode 142)

Finding the Cycle Start Node



The Challenge: Collision happens at 'X', but we need to return the node at '?'.

The Mathematical Proof: m = xC - a

Inter in

```
m = Distance to Start
a = Distance inside cycle to
meeting point

C = Cycle Length
x = Number of full laps Fast
pointer made
```

Inter in

Distance Traveled:
Slow: `m + a`
Fast: `m + a + xC`

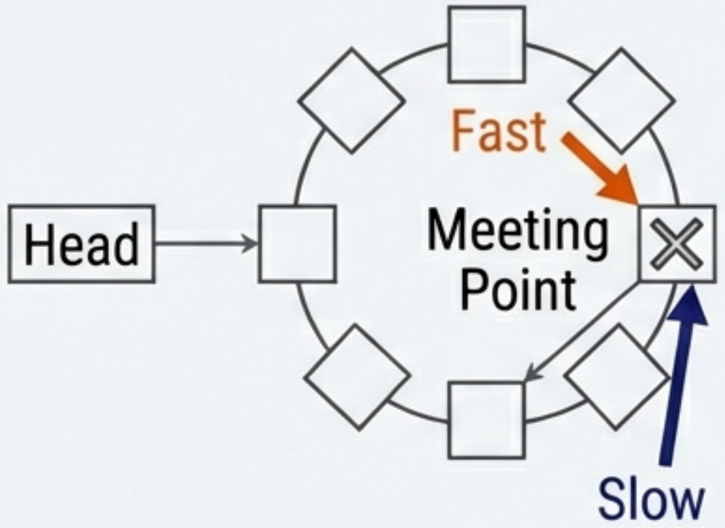
Speed Constraint (`Fast = 2 * Slow`):

```
2(m + a) = m + a + xC
 2m + 2a = m + a + xC
   m + a = xC
       m = xC - a
```

Interpretation: The distance from Head to Start (`m`) is equal to the distance from Meeting Point to Start (`C - a`) considering full laps.

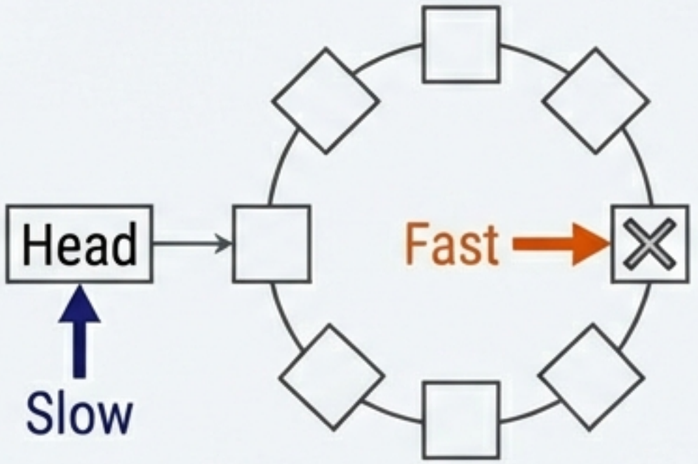
The Algorithm: Reset and March

Phase 1: Detect



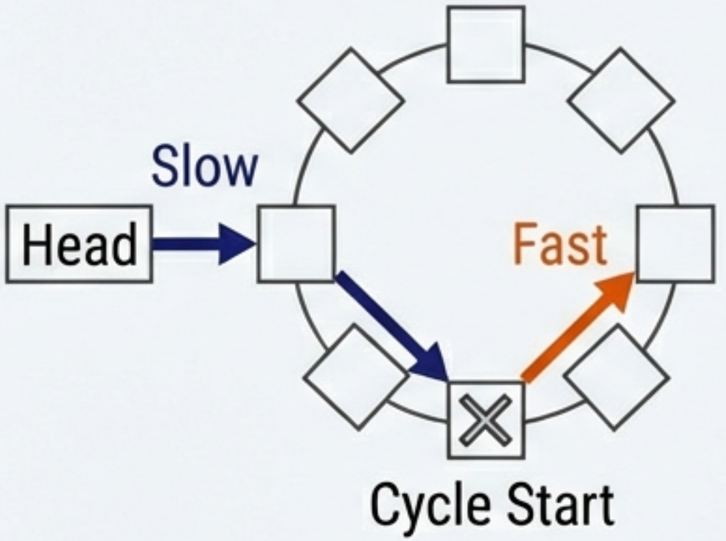
Detect Cycle (Standard Tortoise/Hare).

Phase 2: Reset



Keep **Fast** at collision point. Reset **Slow** to Head.

Phase 3: March



Move both 1 step at a time. The node where they meet is the Cycle Start.

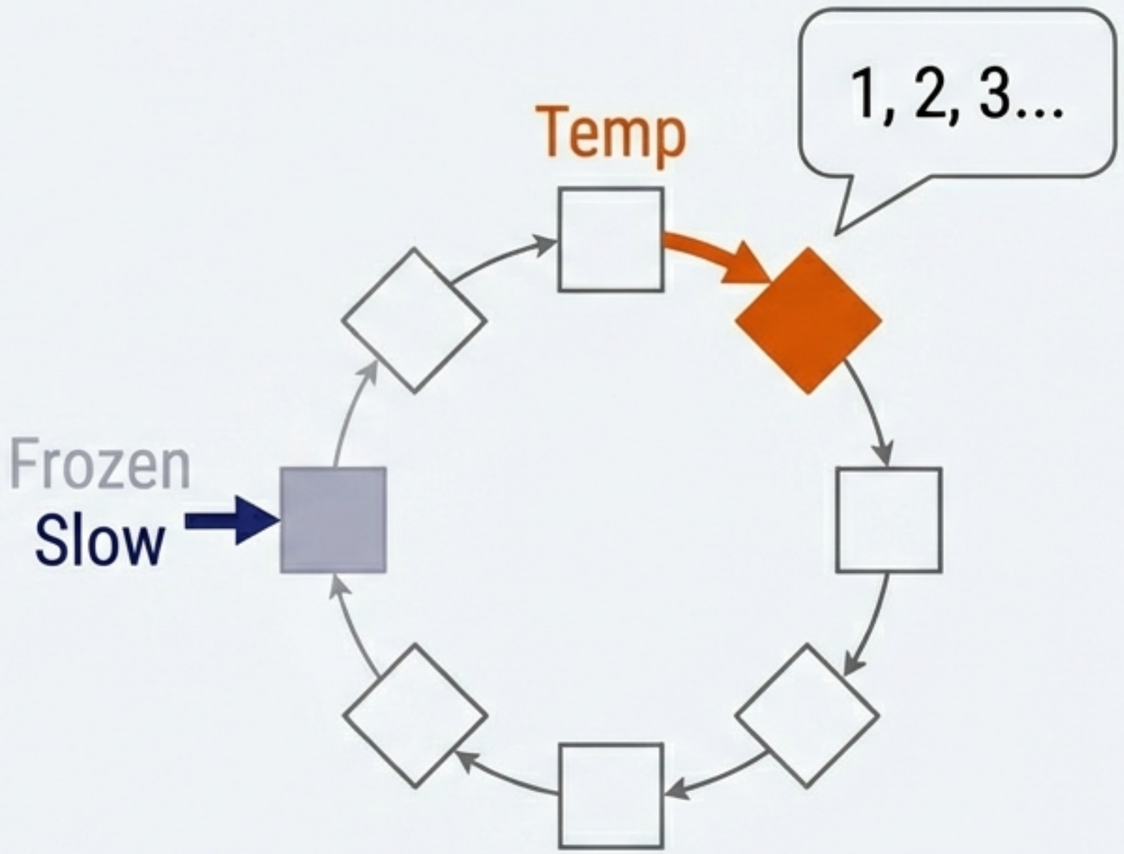
Application 5: Calculate Loop Length

Problem: Find C (number of nodes in the cycle).

Logic:

1. Detect cycle collision.
2. Freeze the Slow pointer at the meeting point.
3. Initialize a counter.
4. Move a temporary pointer step-by-step until it returns to the frozen Slow pointer.

Complexity: Time O(N), Space O(1).

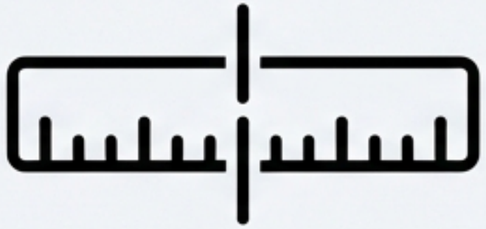


Pattern Summary: When to use Fast & Slow Pointers?

Cyclic Data

Linked Lists with loops, Finite State Machines, or Number Theory sequences.

Position Finding



Finding Middle, Quartiles, or 'K-th from end' in a linear structure.

Space Constraints

When O(1) space is mandatory (HashSets are forbidden).

The Golden Rule: Always check `fast != null && fast.next != null`

Don't Just Watch—Code It.

```
public boolean hasCycle(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;          // 1 step
        fast = fast.next.next;     // 2 steps
        if (slow == fast) {
            return true; // Cycle detected
        }
    }
    return false;
}
```

Final Checklist:

1. Handle edge cases (empty lists, single nodes).
2. Practice the "Happy Number" math proof.
3. Mark Day 8 as DONE.