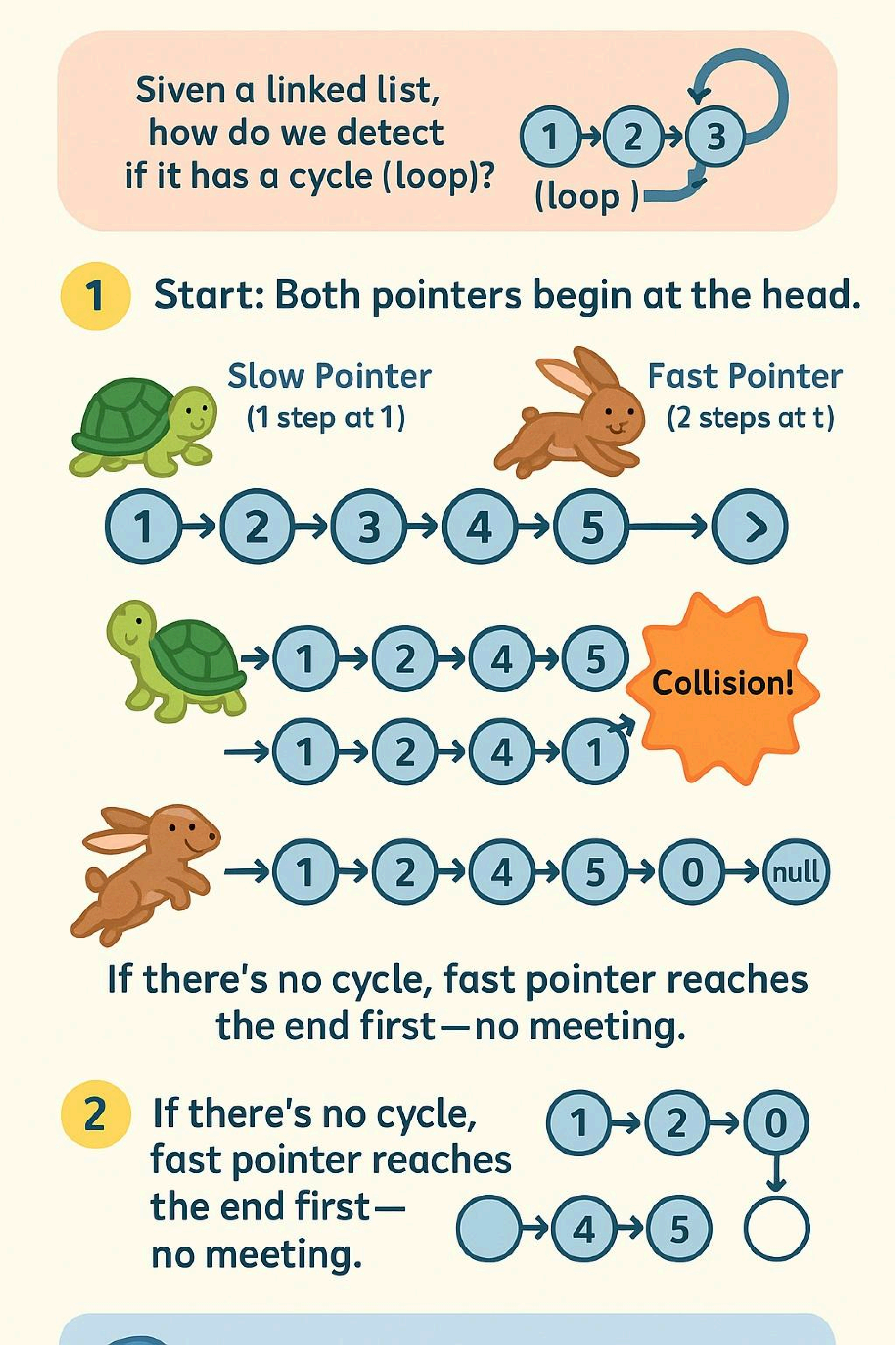
Fast & Slow Pointers Pattern (Hare & Tortoise Algorithm)

This pattern uses **two pointers** that move at **different speeds** (one fast, one slow) to traverse an array or linked list.
It’s mainly used to detect **cycles** or find the **middle** of a data structure without extra space.



Siven a linked list, how do we detect if it has a cycle (loop)?

Start: Both pointers begin at the head.

Slow Pointer (1 step at 1)

Fast Pointer (2 steps at t)

If there's no cycle, fast pointer reaches the end first—no meeting.

If there's no cycle, fast pointer reaches the end first— no meeting.

Classic Use Cases

- **Detecting cycles** in linked lists (Is there a loop? Where does it start?)
- **Finding the middle node** in a linked list

- **Happy number** problems (in number theory)

Example 1: Find the Middle of a Linked List

Problem:
Given a singly linked list, find the middle node.
If there are two middles, return the second one.

Explanation

1. **Slow Pointer:** Moves one step at a time (`slow = slow->next`)
2. **Fast Pointer:** Moves two steps at a time (`fast = fast->next->next`)
3. **When fast reaches the end**, slow will be at the middle!

C Code Example: Find Middle Node

```
#include <stdio.h>
#include <stdlib.h>

// Linked list node definition
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->next = NULL;
    return node;
}

// Function to find the middle of the linked list
struct Node* findMiddle(struct Node* head) {
    struct Node *slow = head, *fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;         // Move slow by 1
        fast = fast->next->next;   // Move fast by 2
    }
    return slow;
}

int main() {
    // Log Session linked list: 1->2->3->4->5
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    struct Node* mid = findMiddle(head);
    printf("Middle node value: %d\n", mid->data);

    return 0;
}
```

How does it work?

- At every loop, `fast` moves twice as fast as `slow`.
- When `fast` is at the end, `slow` is exactly in the middle.

Example 2: Detect Cycle in a Linked List

Problem:
Given a linked list, determine if it has a cycle (loop).

Steps

1. Start both pointers at the head.
2. Slow moves one step; fast moves two steps.
3. If there’s a cycle, **fast and slow will meet** inside the loop.
4. If there’s no cycle, fast will reach the end (`NULL`).

C Code Example: Detect Cycle

```
int hasCycle(struct Node* head) {
    struct Node *slow = head, *fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return 1; // Cycle detected
    }
    return 0; // No cycle
}
```

Example 3: Happy Number (Number Theory)

A “happy number” is a number which eventually reaches 1 when replaced by the sum of the square of each digit repeatedly. If not, it falls into a cycle.

You can use **fast and slow pointers** to detect the cycle!

Why Does This Work?

- **If there is a cycle:** The fast pointer will “lap” the slow pointer and eventually they will meet.
- **If there’s no cycle:** The fast pointer will reach the end (null or 1).

When To Use

- When you need to **detect cycles** (linked lists, sequences, graphs).
- When you need to **find the middle** of a structure in a single pass.
- When two “agents” move at different speeds and you want to detect intersection.

Key Takeaways

- **Fast & Slow pointers**: Powerful for cycle detection and efficient traversals.
- **No extra space needed**: Doesn’t need hash tables or visited markers.

Practice Challenge

- Try writing code to **find the starting point of a cycle** in a linked list after detecting the cycle (Floyd’s algorithm extension).