

## Grokking the Coding Interview: Patterns for Coding Questions

13% completed

Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

### Pattern: Fast & Slow pointers

Introduction

LinkedList Cycle (easy)

Start of LinkedList Cycle (medium)

Happy Number (medium)

Middle of the LinkedList (easy)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

### Pattern: Merge Intervals

Introduction

Merge Intervals (medium)

Insert Interval (medium)

Intervals Intersection (medium)

Conflicting Appointments (medium)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

### Pattern: Cyclic Sort

Introduction

Cyclic Sort (easy)

Find the Missing Number (easy)

Find all Missing Numbers (easy)

Find the Duplicate Number (easy)

Find all Duplicate Numbers (easy)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

## LinkedList Cycle (easy)

We'll cover the following

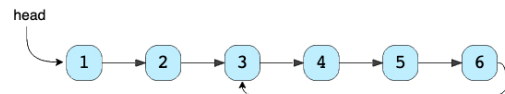
- Problem Statement
- Try it yourself
- Solution
  - Code
  - Time Complexity
  - Space Complexity
- Similar Problems

### Problem Statement

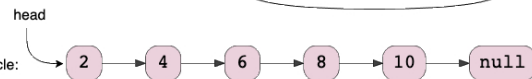
Given the head of a **Singly LinkedList**, write a function to determine if the LinkedList has a **cycle** in it or not.

**Example:**

Following LinkedList has a cycle:



Following LinkedList doesn't have a cycle:



### Try it yourself

Try solving this question here:

Java Python3 JS JS C++

```
1 class Node {
2   constructor(value, next=null){
3     this.value = value;
4     this.next = next;
5   }
6 }
7
8 const has_cycle = function(head) {
9   // TODO: Write your code here
10  return false
11 }
12
13
14 head = new Node(1)
15 head.next = new Node(2)
16 head.next.next = new Node(3)
17 head.next.next.next = new Node(4)
18 head.next.next.next.next = new Node(5)
19 head.next.next.next.next.next = new Node(6)
20 console.log('LinkedList has cycle: ${has_cycle(head)}')
21
22 head.next.next.next.next.next.next = head.next.next
23 console.log('LinkedList has cycle: ${has_cycle(head)}')
24
25 head.next.next.next.next.next.next = head.next.next.next
26 console.log('LinkedList has cycle: ${has_cycle(head)}')
27
```

RUN

SAVE

RESET



Close

Output

4.630s

```
LinkedList has cycle: false
LinkedList has cycle: false
LinkedList has cycle: false
```

### Solution

Imagine two racers running in a circular racing track. If one racer is faster than the other, the faster racer is bound to catch up and cross the slower racer from behind. We can use this fact to devise an algorithm to determine if a LinkedList has a cycle in it or not.

Imagine we have a slow and a fast pointer to traverse the LinkedList. In each iteration, the slow pointer moves one step and the fast pointer moves two steps. This gives us two conclusions:

## Pattern: In-place

### Reversal of a LinkedList

- Introduction
- Reverse a LinkedList (easy)
- Reverse a Sub-list (medium)
- Reverse every K-element Sub-list (medium)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2

## Pattern: Tree Breadth First Search

- Introduction
- Binary Tree Level Order Traversal (easy)
- Reverse Level Order Traversal (easy)
- Zigzag Traversal (medium)
- Level Averages in a Binary Tree (easy)
- Minimum Depth of a Binary Tree (easy)
- Level Order Successor (easy)
- Connect Level Order Siblings (medium)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2

## Pattern: Tree Depth First Search

- Introduction
- Binary Tree Path Sum (easy)
- All Paths for a Sum (medium)
- Sum of Path Numbers (medium)
- Path With Given Sequence (medium)
- Count Paths for a Sum (medium)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2

## Pattern: Two Heaps

- Introduction
- Find the Median of a Number Stream (medium)
- Sliding Window Median (hard)
- Maximize Capital (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1

## Pattern: Subsets

- Introduction
- Subsets (easy)
- Subsets With Duplicates (easy)
- Permutations (medium)
- String Permutations by changing case (medium)
- Balanced Parentheses (hard)
- Unique Generalized

1. If the LinkedList doesn't have a cycle in it, the fast pointer will reach the end of the LinkedList before the slow pointer to reveal that there is no cycle in the LinkedList.
2. The slow pointer will never be able to catch up to the fast pointer if there is no cycle in the LinkedList.

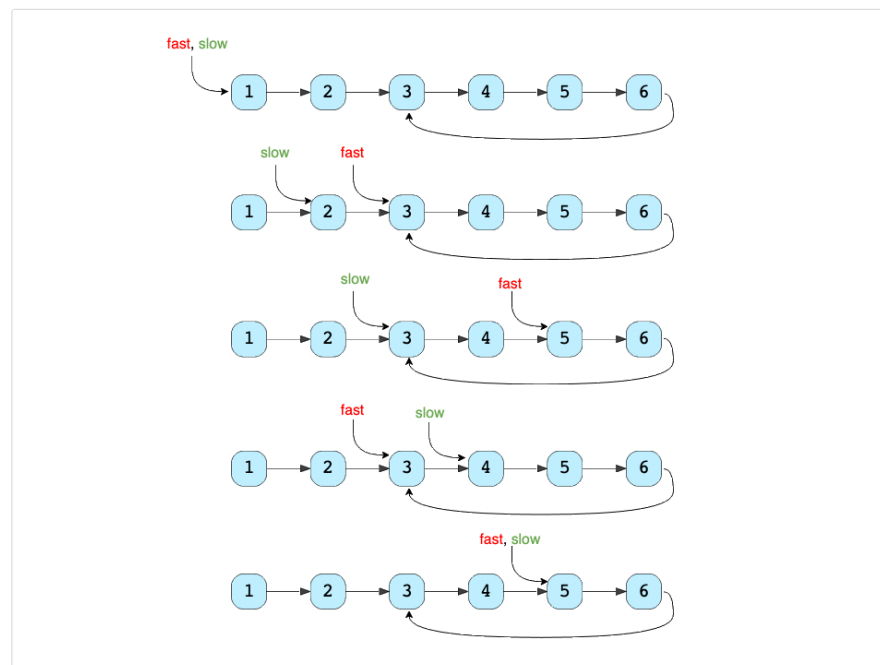
If the LinkedList has a cycle, the fast pointer enters the cycle first, followed by the slow pointer. After this, both pointers will keep moving in the cycle infinitely. If at any stage both of these pointers meet, we can conclude that the LinkedList has a cycle in it. Let's analyze if it is possible for the two pointers to meet. When the fast pointer is approaching the slow pointer from behind we have two possibilities:

1. The fast pointer is one step behind the slow pointer.
2. The fast pointer is two steps behind the slow pointer.

All other distances between the fast and slow pointers will reduce to one of these two possibilities. Let's analyze these scenarios, considering the fast pointer always moves first:

1. **If the fast pointer is one step behind the slow pointer:** The fast pointer moves two steps and the slow pointer moves one step, and they both meet.
2. **If the fast pointer is two steps behind the slow pointer:** The fast pointer moves two steps and the slow pointer moves one step. After the moves, the fast pointer will be one step behind the slow pointer, which reduces this scenario to the first scenario. This means that the two pointers will meet in the next iteration.

This concludes that the two pointers will definitely meet if the LinkedList has a cycle. A similar analysis can be done where the slow pointer moves first. Here is a visual representation of the above discussion:



### Code

Here is what our algorithm will look like:

```
Java Python3 C++ JS

1 class Node {
2   constructor(value, next = null) {
3     this.value = value;
4     this.next = next;
5   }
6 }
7
8 function has_cycle(head) {
9   let slow = head,
10     fast = head;
11   while (fast !== null && fast.next !== null) {
12     fast = fast.next.next;
13     slow = slow.next;
14     if (slow === fast) {
15       return true; // found the cycle
16     }
17   }
18   return false;
19 }
20
21
22 const head = new Node(1);
23 head.next = new Node(2);
24 head.next.next = new Node(3);
25 head.next.next.next = new Node(4);
26 head.next.next.next.next = new Node(5);
27 head.next.next.next.next.next = new Node(6);
28 console.log('LinkedList has cycle: ', has_cycle(head));
```

Abbreviations (hard)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

Pattern: Modified Binary Search

Introduction

Order-agnostic Binary Search (easy)

Ceiling of a Number (medium)

Next Letter (medium)

Number Range (medium)

Search in a Sorted Infinite Array (medium)

Minimum Difference Element (medium)

Bitonic Array Maximum (easy)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

Pattern: Bitwise XOR

Introduction

Single Number (easy)

Two Single Numbers (medium)

Complement of Base 10 Number (medium)

Problem Challenge 1

Solution Review: Problem Challenge 1

Pattern: Top 'K' Elements

Introduction

Top 'K' Numbers (easy)

Kth Smallest Number (easy)

'K' Closest Points to the Origin (easy)

Connect Ropes (easy)

Top 'K' Frequent Numbers (medium)

Frequency Sort (medium)

Kth Largest Number in a Stream (medium)

'K' Closest Numbers (medium)

Maximum Distinct Elements (medium)

Sum of Elements (medium)

Rearrange String (hard)

Problem Challenge 1

Solution Review: Problem Challenge 1

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

RUN

SAVE

RESET

Close

Output

2.546s

LinkedList has cycle: false  
LinkedList has cycle: true  
LinkedList has cycle: true

Time Complexity #

As we have concluded above, once the slow pointer enters the cycle, the fast pointer will meet the slow pointer in the same loop. Therefore, the time complexity of our algorithm will be  $O(N)$  where 'N' is the total number of nodes in the LinkedList.

Space Complexity #

The algorithm runs in constant space  $O(1)$ .

Similar Problems #

**Problem 1:** Given the head of a LinkedList with a cycle, find the length of the cycle.

**Solution:** We can use the above solution to find the cycle in the LinkedList. Once the fast and slow pointers meet, we can save the slow pointer and iterate the whole cycle with another pointer until we see the slow pointer again to find the length of the cycle.

Here is what our algorithm will look like:

Java

Python3

C++

JS

```
21 }
22
23
24 function calculate_cycle_length(slow) {
25     let current = slow,
26     cycle_length = 0;
27     while (true) {
28         current = current.next;
29         cycle_length += 1;
30         if (current === slow) {
31             break;
32         }
33     }
34     return cycle_length;
35 }
36
37
38 const head = new Node(1);
39 head.next = new Node(2);
40 head.next.next = new Node(3);
41 head.next.next.next = new Node(4);
42 head.next.next.next.next = new Node(5);
43 head.next.next.next.next.next = new Node(6);
44 head.next.next.next.next.next.next = head.next.next;
45 console.log('LinkedList cycle length: ' + find_cycle_length(head));
46
47 head.next.next.next.next.next.next = head.next.next.next;
48 console.log('LinkedList cycle length: ' + find_cycle_length(head));
```

RUN

SAVE

RESET

**Time and Space Complexity:** The above algorithm runs in  $O(N)$  time complexity and  $O(1)$  space complexity.

← Back

Introduction

MARK AS COMPLETED

Next →

Start of LinkedList Cycle (medium)

Report an Issue