

Grokking the Coding Interview: Patterns for Coding Questions

16% completed

- 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

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

Solution Review: Problem Challenge 1

We'll cover the following

- Palindrome LinkedList (medium)
- Solution
 - Code
 - Time complexity
 - Space complexity

Palindrome LinkedList (medium)

Given the head of a **Singly LinkedList**, write a method to check if the **LinkedList** is a **palindrome** or not.

Your algorithm should use **constant space** and the input **LinkedList** should be in the original form once the algorithm is finished. The algorithm should have $O(N)$ time complexity where 'N' is the number of nodes in the **LinkedList**.

Example 1:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> null
Output: true
```

Example 2:

```
Input: 2 -> 4 -> 6 -> 4 -> 2 -> 2 -> null
Output: false
```

Solution

As we know, a palindrome **LinkedList** will have nodes values that read the same backward or forward. This means that if we divide the **LinkedList** into two halves, the node values of the first half in the forward direction should be similar to the node values of the second half in the backward direction. As we have been given a **Singly LinkedList**, we can't move in the backward direction. To handle this, we will perform the following steps:

1. We can use the **Fast & Slow pointers** method similar to [Middle of the LinkedList](#) to find the middle node of the **LinkedList**.
2. Once we have the middle of the **LinkedList**, we will reverse the second half.
3. Then, we will compare the first half with the reversed second half to see if the **LinkedList** represents a palindrome.
4. Finally, we will reverse the second half of the **LinkedList** again to revert and bring the **LinkedList** back to its original form.

Code

Here is what our algorithm will look like:

```
Java Python3 C++ JS
1
2 class ListNode {
3     int value = 0;
4     ListNode next;
5
6     ListNode(int value) {
7         this.value = value;
8     }
9 }
10
11 class PalindromicLinkedList {
12
13     public static boolean isPalindrome(ListNode head) {
14         if (head == null || head.next == null) {
15             return true;
16
17             // find middle of the LinkedList
18             ListNode slow = head;
19             ListNode fast = head;
20             while (fast != null && fast.next != null) {
21                 slow = slow.next;
22                 fast = fast.next.next;
23             }
24
25             ListNode headSecondHalf = reverse(slow); // reverse the second half
26             ListNode copyHeadSecondHalf = headSecondHalf; // store the head of reversed part to revert back later
27
28             // compare the first and the second half
```

RUN

SAVE

RESET



Close

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

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

Output

4.241s

Is palindrome: true
Is palindrome: false

Time complexity

The above algorithm will have a time complexity of $O(N)$ where 'N' is the number of nodes in the LinkedList.

Space complexity

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

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#)

← Back

Problem Challenge 1

MARK AS COMPLETED

Next →

Problem Challenge 2

Report an Issue