234. Palindrome Linked List Stack Recursion **Linked List** Two Pointers Easy

# **Problem Description**

palindrome, but [1 -> 2 -> 3] is not. Intuition

The problem presents a scenario where you're given a singly linked list and asks you to determine if it is a palindrome. A palindrome

is a sequence that reads the same forward and backward. For example, in the case of a linked list, [1 -> 2 -> 2 -> 1] is a

To solve the problem of determining whether a <u>linked list</u> is a palindrome, we need an approach that allows us to compare elements

from the start and end of the list efficiently. Since we can't access the elements of a linked list in a reverse order as easily as we can

with an array, we have to be creative.

The solution involves several steps:

- 1. Find the middle of the linked list: We can find the middle of the linked list using the fast and slow pointer technique. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. When the fast pointer reaches the end of the list, the slow pointer will be in the middle.
- 3. Compare the first and the second half: After reversal, we then compare the node values from the start of the list and the start of the reversed second half. If all corresponding nodes are equal, then the list is a palindrome.

2. Reverse the second half of the linked list: Starting from the middle of the list, reverse the order of the nodes. This will allow us

to directly compare the nodes from the start and the end of the list without needing to store additional data or indices.

4. Restore the list (optional): If the problem required you to maintain the original structure of the list after checking for a

- palindrome, you would follow up by reversing the second half of the list again and reattaching it to the first half. However, this step is not implemented in the provided code, as it is not part of the problem's requirements.
- makes this method quite optimal. Solution Approach

The crux of this approach lies in the efficient O(n) traversal and no additional space complexity apart from a few pointers, which

1. Two-pointer technique: To find the middle of the list, we use two pointers (slow and fast). The slow pointer is incremented by one node, while the fast pointer is incremented by two nodes on each iteration. When the fast pointer reaches the end, slow will be pointing at the middle node.

## while fast and fast.next: slow, fast = slow.next, fast.next.next

1 while pre:

5 return True

Example Walkthrough

Step 1: Finding the Middle

for comparison purposes.

4 List: 1 -> 2 -> 3 -> 2 -> 1

1 Initial state:

2 slow -> 1

3 fast -> 2

7 slow -> 3

1 slow, fast = head, head.next

The solution approach follows the intuition which is broken down into the following algorithms and patterns:

2. Reversing the second half of the list: Once we have the middle node, we reverse the second half of the list starting from slow.next. To do this, we initialize two pointers pre (to keep track of the previous node) and cur (the current node). We then iterate until cur is not None, each time setting cur. next to pre, effectively reversing the links between the nodes. 1 pre, cur = None, slow.next 2 while cur:

the values of the nodes starting from head and pre. If at any point the values differ, we return False indicating that the list is not

a palindrome. Otherwise, we keep advancing both pointers until pre is None. If we successfully reach the end of both halves

t = cur.next cur.next = pre pre, cur = cur, t 3. Comparison of two halves: After reversing the second half, pre will point to the head of the reversed second half. We compare

if pre.val != head.val:

pre, head = pre.next, head.next

size; we're just manipulating the existing nodes in the linked list.

8 fast -> 1 (fast reaches the end of the list so we stop here)

We now iterate and reverse the link between the nodes until cur is None:

Reversed part: None <- 3 <- 2 List: 1 -> 2 -> 3 -> 2 -> 1

At this stage, slow is pointing to the middle of the list.

return False

without mismatches, the list is a palindrome, so we return True.

Let's illustrate the solution approach using a small example. Consider the linked list [1 -> 2 -> 3 -> 2 -> 1]. The goal is to determine if this list represents a palindrome.

We use the two-pointer technique. Initially, both slow and fast point to the first element, with fast moving to the next immediately

The code uses the two-pointer technique and the reversal of a linked list to solve the problem very effectively. The total time

complexity of the algorithm is O(n), and the space complexity is O(1), because no additional space is used proportional to the input

### 1 Iteration 1: 2 slow -> 2

Now we begin traversal:

3 fast -> 3 List: 1 -> 2 -> 3 -> 2 -> 1 6 Iteration 2:

```
Step 2: Reverse the Second Half
Starting from the middle node (where slow is currently pointing), we proceed to reverse the second half of the list. We'll use two
pointers pre and cur to achieve this:
```

2 cur points to 3 (slow.next) 3 List: 1 -> 2 -> 3 -> 2 -> 1

1 pre points to None

1 Iteration 1:

11 Iteration 3:

13 cur -> None

12 pre -> 1

9 List: 1 -> 2 -> 3 -> 2 -> 1

2 pre -> 3 3 cur -> 2 Reversed part: None <- 3 List: 1 -> 2 -> 3 -> 2 -> 1 6 Iteration 2: 7 pre -> 2 8 cur -> 1

After reversing, we have pre pointing to the new head of the reversed second half, which is the node with the value 1.

We now have two pointers, head pointing to the first node of the list and pre pointing to the head of the reversed second half. We

need to compare the values of both halves:

1 pre points to 1, head points to 1

We move both pointers and compare their values:

1 pre -> 2, head -> 2 (values match, move forward)

2 pre -> 3, head -> 3 (values match, move forward)

def \_\_init\_\_(self, val=0, next\_node=None):

self.val = val

slow = head

prev = None

while prev:

return True

fast = head.next

class Solution:

11

12

13

14

15

16

18

19

20

27

28

29

30

31

32

33

34

self.next = next\_node

while fast and fast.next:

slow = slow.next

current = slow.next

fast = fast.next.next

if prev.val != head.val:

prev, head = prev.next, head.next

# If all nodes matched, it's a palindrome

return False

\* Definition for singly-linked list.

ListNode(int val) { this.val = val; }

slow = slow.next;

ListNode prev = null;

ListNode current = slow;

while (current != null) {

fast = fast.next.next;

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

// Reverse the second half of the list

ListNode firstHalfIterator = head;

currentNode = nextTemp;

return false;

head = head->next;

prevNode = prevNode->next;

if (prevNode->val != head->val) {

// Move to the next nodes in both halves

// All values matched, so it is a palindrome

function isPalindrome(head: ListNode | null): boolean {

let fastPointer: ListNode | null = head?.next;

let slowPointer: ListNode | null = head;

// Traverse the list to find the middle

slowPointer = slowPointer.next;

fastPointer = fastPointer.next.next;

let temp: ListNode = current.next;

while (previous !== null && head !== null) {

// If all values matched, then the list is a palindrome

current.next = previous;

// Compare the two halves of the list

previous = previous.next;

head = head.next;

O(n) which is O(3n) or simply O(n).

if (previous.val !== head.val) {

previous = current;

current = temp;

while (prevNode) {

return true;

Typescript Solution

// Compare the reversed second half with the first half

\* Function to determine if a given singly linked list is a palindrome.

\* @param {ListNode | null} head - The head of the singly linked list.

while (fastPointer !== null && fastPointer.next !== null) {

\* @returns {boolean} - True if the list is a palindrome, false otherwise.

// Two pointers: slow moves one step at a time, fast moves two steps.

return false; // Values do not match, not a palindrome

// If the values are different, then it's not a palindrome

ListNode secondHalfIterator = prev;

while (secondHalfIterator != null) {

# Reverse the second half of the list

confirms the linked list  $[1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 1]$  is indeed a palindrome.

def isPalindrome(self, head: Optional[ListNode]) -> bool:

**Step 3: Compare Two Halves** 

14 Reversed part: None <- 3 <- 2 <- 1

Python Solution 1 # Definition for singly-linked list. 2 class ListNode:

When pre becomes None, we've successfully compared all nodes of the reversed half with the corresponding nodes of the first half

When implementing these steps in a programming language like Python, the overall result of this example would be that the function

and found that all the values match, which implies that the list represents a palindrome. Hence, we return True.

# Initialize two pointers, slow moves one step at a time, fast moves two steps

# Move fast pointer to the end of the list, and slow to the middle

ListNode(int val, ListNode next) { this.val = val; this.next = next; }

// Move fast pointer to the end, and slow to the middle of the list

ListNode temp = current.next; // Stores the next node

current = temp; // Move to the next node in the original list

current.next = prev; // Reverses the link

prev = current; // Moves prev to current node

// Compare the reversed second half with the first half

while current: temp = current.next 23 current.next = prev 24 prev, current = current, temp 25 26 # Compare the first half and the reversed second half

```
class Solution {
       public boolean isPalindrome(ListNode head) {
13
           // Use two pointers: slow moves one step at a time and fast moves two steps at a time.
14
           ListNode slow = head;
15
           ListNode fast = head;
16
```

**Java Solution** 

class ListNode {

int val;

ListNode next;

ListNode() {}

/\*\*

\*/

9

11

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

32

33

34

35

36

37

38

39

40

43

44

45

46

47

48

49

50

};

1 /\*\*

\*/

10

11

12

13

14

15

22

23

24

25

26

28

29

30

34

35

36

38

39

40

10 }

```
38
               // If values are different, then it's not a palindrome
39
               if (secondHalfIterator.val != firstHalfIterator.val) {
40
                    return false;
42
43
               // Move to the next nodes in both halves
44
               secondHalfIterator = secondHalfIterator.next;
45
               firstHalfIterator = firstHalfIterator.next;
46
47
           // All values matched, so it's a palindrome
48
49
           return true;
50
51 }
52
C++ Solution
   /**
    * Definition for singly-linked list.
    * struct ListNode {
          int val;
    *
          ListNode *next;
          ListNode() : val(0), next(nullptr) {}
          ListNode(int x) : val(x), next(nullptr) {}
          ListNode(int x, ListNode *next) : val(x), next(next) {}
    * };
    */
11 class Solution {
   public:
13
       bool isPalindrome(ListNode* head) {
           // Use two pointers to find the middle of the list
14
15
           ListNode* slowPtr = head;
           ListNode* fastPtr = head->next;
16
           // Move fastPtr by two and slowPtr by one step
18
           // When fastPtr reaches the end, slowPtr will be at the middle
19
           while (fastPtr && fastPtr->next) {
20
21
               slowPtr = slowPtr->next;
22
               fastPtr = fastPtr->next->next;
23
24
25
           // Reverse the second half of the list
26
           ListNode* prevNode = nullptr;
           ListNode* currentNode = slowPtr->next;
27
28
           while (currentNode) {
29
               ListNode* nextTemp = currentNode->next;
               currentNode->next = prevNode;
30
               prevNode = currentNode;
31
```

### 16 // Reverse the second half of the list 17 let current: ListNode = slowPointer.next; 18 slowPointer.next = null; 19 let previous: ListNode = null; 20 while (current !== null) { 21

return true;

```
Time and Space Complexity
The code above checks if a given singly-linked list is a palindrome. Here is the analysis of its time and space complexity:
Time Complexity
The algorithm uses two pointers (slow and fast) to find the middle of the linked list. The fast pointer moves two steps for every step
the slow pointer takes. This loop will run in O(n/2) time, which is O(n) where n is the number of nodes in the list.
After finding the middle of the list, the code reverses the second half of the linked list. This is another loop that runs from the middle
to the end of the list, which is also 0(n/2) or simplifies to 0(n).
```

Finally, the code compares the values of nodes from the start of the list and the start of the reversed second half. This comparison

The total time complexity is the sum of these steps, which are all linear with respect to the length of the linked list: 0(n) + 0(n) +

**Space Complexity** There are no additional data structures used that grow with the input size. The pointers and temporary variables use a constant amount of space regardless of the size of the linked list. Therefore, the space complexity is 0(1), which means it is constant.

So, the overall time complexity of the algorithm is O(n), and the space complexity is O(1).

stops when the end of the reversed half is reached, which is at most n/2 steps, so 0(n/2) or 0(n).