Previous Next

234. Palindrome Linked List

Nov. 19, 2019 | 32.2K views

Average Rating: 4.91 (75 votes)

Given a singly linked list, determine if it is a palindrome.

Example 1:

```
Input: 1->2
Output: false
```

Example 2:

```
Input: 1->2->2->1
Output: true
```

Follow up:
Could you do it in O(n) time and O(1) space?

Solution

Approach 1: Copy into Array List and then Use Two Pointer Technique

Intuition

If you're not too familiar with **Linked Lists** yet, here's a quick recap on **Lists**.

There are two commonly used **List** implementations, the **Array List** and the **Linked List**. If we have some values we want to store in a list, how would each List implementation hold them?

- An **Array List** uses an underlying **Array** to store the values. We can access the value at any position in the list in $O(1)$ time, as long as we know the index. This is based on the underlying memory addressing.
- A **Linked List** uses **Objects** commonly called **Nodes**. Each **Node** holds a value and a *next* pointer to the next node. Accessing a node at a particular index would take $O(n)$ time because we have to go down the list using the *next* pointers.

Determining whether or not an *Array List* is a palindrome is straightforward. We can simply use the **two-pointer technique** to compare indexes at either end, moving in towards the middle. One pointer starts at the start and goes up, and the other starts at the end and goes down. This would take $O(n)$ because each index access is $O(1)$ and there are n index accesses in total.

However, it's not so straightforward for a Linked List. Accessing the values in any order other than forward, sequentially, is ***not*** $O(1)$. Unfortunately, this includes (iteratively) accessing the values in *reverse*. We will need a completely different approach.

On the plus side, making a copy of the Linked List values into an *Array List* is $O(n)$. Therefore, the simplest solution is to copy the values of the Linked List into an Array List (or Vector, or plain Array). Then, we can solve the problem using the **two-pointer technique**.

Algorithm

We can split this approach into 2 steps:

1. Copying the Linked List into an Array.
2. Checking whether or not the *Array* is a palindrome.

To do the first step, we need to iterate through the Linked List, adding each value to an Array. We do this by using a variable `currentNode` to point at the current Node we're looking at, and at each iteration adding `currentNode.val` to the array and updating `currentNode` to point to `currentNode.next`. We should stop looping once `currentNode` points to a `null` node.

The best way of doing the second step depends on the programming language you're using. In Python, it's straightforward to make a reversed copy of a list and also straightforward to compare two lists. In other languages, this is not so straightforward and so it's probably best to use the **two-pointer technique** to check for a palindrome. In the two-pointer technique, we put a pointer at the start and a pointer at the end, and at each step check the values are equal and then move the pointers inwards until they meet at the center.

When writing your own solution, remember that we need to compare values in the nodes, not the nodes themselves. `node_1.val == node_2.val` is the *correct* way of comparing the nodes. `node_1 == node_2` will *not work* the way you expect.

Java | Python | Copy

```python
def isPalindrome(self, head: ListNode) -> bool:
    vals = []
    current_node = head
    while current_node is not None:
        vals.append(current_node.val)
        current_node = current_node.next
    return vals == vals[::-1]
```

Complexity Analysis

- Time complexity : $O(n)$, where n is the number of nodes in the Linked List.

  In the first part, we're copying a Linked List into an Array List. Iterating down a Linked List in sequential order is $O(n)$, and each of the n writes to the ArrayList is $O(1)$. Therefore, the overall cost is $O(n)$.

  In the second part, we're using the two pointer technique to check whether or not the Array List is a palindrome. Each of the n values in the Array list is accessed once, and a total of $O(n/2)$ comparisons are done. Therefore, the overall cost is $O(n)$. The Python trick (reverse and list comparison as a one liner) is also $O(n)$.

  This gives $O(2n) = O(n)$ because we always drop the constants.

- Space complexity : $O(n)$, where n is the number of nodes in the Linked List.

  We are making an Array List and adding n values to it.

Approach 2: Recursive (Advanced)

Intuition

In an attempt to come up with a way of using $O(1)$ space, you might have thought of using recursion. However, this is still $O(n)$ space. Let's have a look at it and understand why it is **not** $O(1)$ space.

Recursion gives us an elegant way to iterate through the nodes in reverse. For example, this algorithm will print out the values of the nodes *in reverse*. Given a node, the algorithm checks if it is `null`. If it is `null`, nothing happens. Otherwise, all nodes *after* it are processed, and *then* the value for the current node is printed.

```
function print_values_in_reverse(ListNode head)
    if head is NOT null
        print_values_in_reverse(head.next)
        print head.val
```

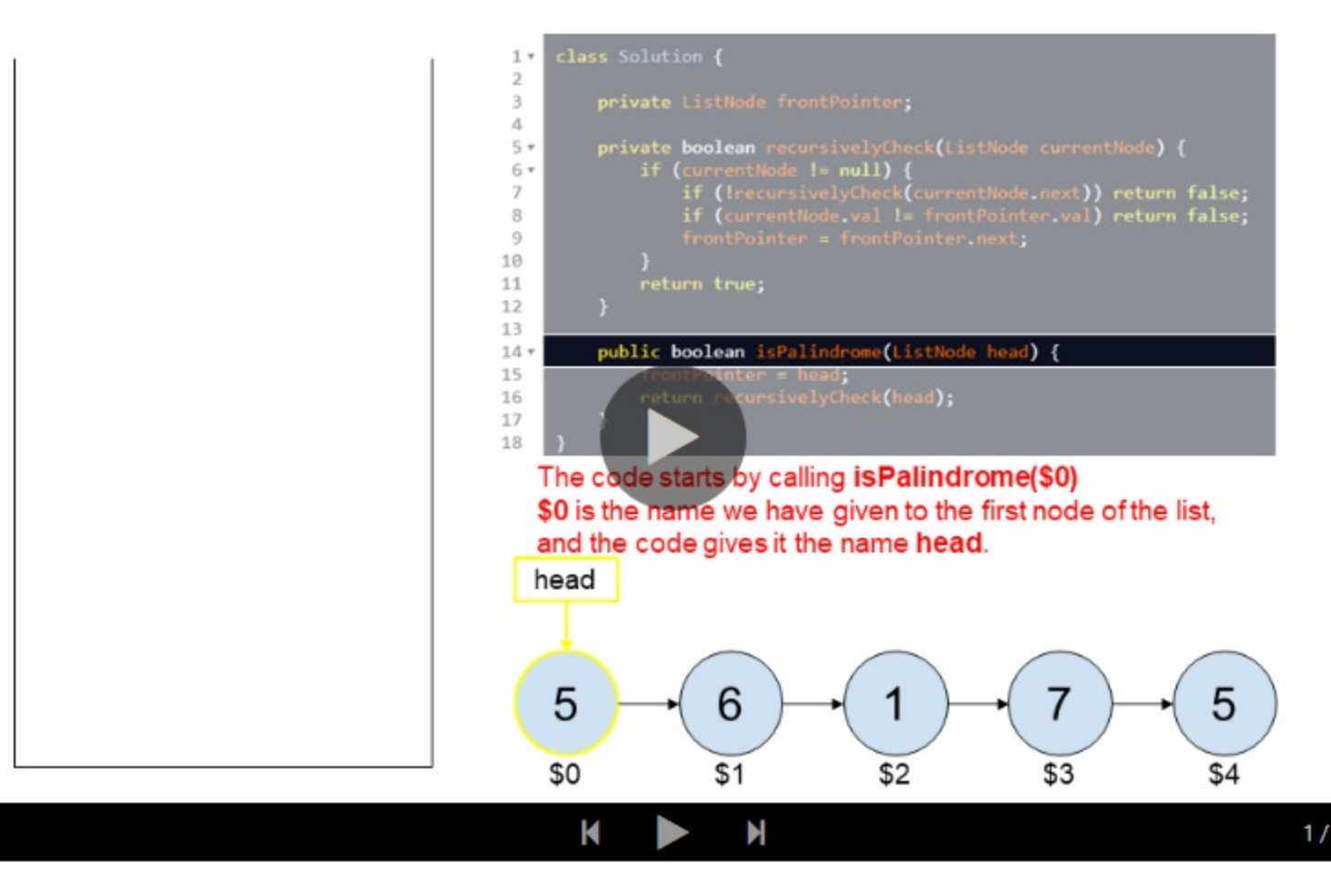
If we iterate the nodes in reverse using recursion, and iterate forward at the same time using a variable *outside* the recursive function, then we can check whether or not we have a palindrome.

Algorithm

When given the head node (or any other node), referred to as `currentNode`, the algorithm firstly checks the *rest* of the Linked List. If it discovers that further down that the Linked List is *not* a palindrome, then it returns `false`. Otherwise, it checks that `currentNode.val == frontPointer.val`. If not, then it returns `false`. Otherwise, it moves `frontPointer` forward by 1 node and returns `true` to say that from this point forward, the Linked List is a valid palindrome.

It might initially seem surprisingly that `frontPointer` is always pointing where we want it. The reason it works is because the order in which nodes are processed by the recursion is in reverse (remember our "printing" algorithm above). Each node compares itself against `frontPointer` and then moves `frontPointer` down by 1, ready for the next node in the recursion. In essence, we are iterating both backwards and forwards at the same time.

Here is an animation that shows how the algorithm works. The nodes have each been given a unique identifier (e.g. `$1` and `$4`) so that they can more easily be referred to in the explanations. The computer needs to use its runtime stack for recursive functions.



Java | Python | Copy

```python
def isPalindrome(self, head: ListNode) -> bool:

    self.front_pointer = head

    def recursively_check(current_node=head):
        if current_node is not None:
            if not recursively_check(current_node.next):
                return False
            if self.front_pointer.val != current_node.val:
                return False
            self.front_pointer = self.front_pointer.next
        return True

    return recursively_check()
```

Complexity Analysis

- Time complexity : $O(n)$, where n is the number of nodes in the Linked List.

  The recursive function is run once for each of the n nodes, and the body of the recursive function is $O(1)$. Therefore, this gives a total of $O(n)$.

- Space complexity : $O(n)$, where n is the number of nodes in the Linked List.

  I hinted at the start that this is not using $O(1)$ space. This might seem strange, after all we aren't creating any new data structures. So, where is the $O(n)$ extra memory we're using? Understanding what is happening here requires understanding how the computer runs a recursive function.

  Each time a function is called within a function, the computer needs to keep track of where it is up to (and the values of any local variables) in the current function before it goes into the called function. It does this by putting an entry on something called the **runtime stack**, called a **stack frame**. Once it has created a stack frame for the current function, it can then go into the called function. Then once it is finished with the called function, it pops the top stack frame to resume the function it had been in before the function call was made.

  Before doing any palindrome checking, the above recursive function creates n of these stack frames because the first step of processing a node is to process the nodes after it, which is done with a recursive call. Then once it has the n stack frames, it pops them off one-by-one to process them.

  So, the space usage is on the *runtime stack* because we are creating n stack frames. Always make sure to consider what's going on the *runtime stack* when analyzing a recursive function. It's a common mistake to forget to.

Not only is this approach still using $O(n)$ space, it is worse than the first approach because in many languages (such as Python), stack frames are large, and there's a maximum runtime stack depth of 1000 (you can increase it, but you risk causing memory errors with the underlying interpreter). With *every* node creating a stack frame, this will greatly limit the maximum Linked List size the algorithm can handle.

Approach 3: Reverse Second Half In-place

Intuition

The ***only*** way we can avoid using $O(n)$ extra space is by modifying the input in-place.

The strategy we can use is to reverse the second half of the Linked List in-place (modifying the Linked List structure), and then comparing it with the first half. Afterwards, we should re-reverse the second half and put the list back together. While you don't need to restore the list to pass the test cases, it is still good programming practice because the function could be a part of a bigger program that doesn't want the Linked List broken.

Algorithm

Specifically, the steps we need to do are:

1. Find the end of the first half.
2. Reverse the second half.
3. Determine whether or not there is a palindrome.
4. Restore the list.
5. Return the result.

To do *step 1*, we could count the number of nodes, calculate how many nodes are in the first half, and then iterate back down the list to find the end of the first half. Or, we could do it in a single parse using the **two runners pointer technique**. Either is acceptable, however we'll have a look at the two runners pointer technique here.

Imagine we have 2 runners one fast and one slow, running down the nodes of the Linked List. In each second, the fast runner moves down 2 nodes, and the slow runner just 1 node. By the time the fast runner gets to the end of the list, the slow runner will be half way. By representing the runners as pointers, and moving them down the list at the corresponding speeds, we can use this trick to find the middle of the list, and then split the list into two halves.

If there is an odd-number of nodes, then the "middle" node should remain attached to the first half.

Step 2 uses the algorithm that can be found in the solution article for the Reverse Linked List problem to reverse the second half of the list.

Step 3 is fairly straightforward. Remember that we have the first half, which might also contain a "middle" node at the end, and the second half, which is reversed. We can step down the lists simultaneously ensuring the node values are equal. When the node we're up to in the second list is `null`, we know we're done. If there was a middle value attached to the end of the first list, it is correctly ignored by the algorithm. The result should be saved, but not returned, as we still need to restore the list.

Step 4 requires using the same function you used for step 2, and then for *step 5* the saved result should be returned.

Java | Python | Copy

```python
class Solution:

    def isPalindrome(self, head: ListNode) -> bool:
        if head is None:
            return True

        # Find the end of first half and reverse second half.
        first_half_end = self.end_of_first_half(head)
        second_half_start = self.reverse_list(first_half_end.next)

        # Check whether or not there's a palindrome.
        result = True
        first_position = head
        second_position = second_half_start
        while result and second_position is not None:
            if first_position.val != second_position.val:
                result = False
            first_position = first_position.next
            second_position = second_position.next

        # Restore the list and return the result.
        first_half_end.next = self.reverse_list(second_half_start)
        return result

    def end_of_first_half(self, head):
        fast = head
        slow = head
```

Complexity Analysis

- Time complexity : $O(n)$, where n is the number of nodes in the Linked List.

  Similar to the above approaches. Finding the middle is $O(n)$, reversing a list in place is $O(n)$, and then comparing the 2 resulting Linked Lists is also $O(n)$.

- Space complexity : $O(1)$.

  We are changing the next pointers for half of the nodes. This was all memory that had already been allocated, so we are not using any extra memory and therefore it is $O(1)$.

  I have seen some people on the discussion forum saying it has to be $O(n)$ because we're creating a new list. This is incorrect, because we are changing each of the pointers one-by-one, in-place. We are not needing to allocate more than $O(1)$ extra memory to do this work, and there is $O(1)$ stack frames going on the stack. It is the same as reversing the values in an Array in place (using the two-pointer technique).

The downside of this approach is that in a concurrent environment (multiple threads and processes accessing the same data), access to the Linked List by other threads or processes would have to be locked while this function is running, because the Linked List is temporarily broken. This is a limitation of many in-place algorithms though.

Rate this article: ★★★★★

Previous Next

Comments: 24

Sort By

Type comment here... (Markdown is supported)

Preview Post

webdev2709 ★31 November 30, 2019 1:45 AM

Thank you @Hai_dee . Your articles on algorithm analysis are very detailed and have been a huge help in understanding things. I really appreciate you for taking time to write such detailed analysis.

Did you post an analysis on Coin Change problem here ? I couldn't find an option to find all articles from a person. Could you share a link if you ever did that ?

Read More

30 Share Reply

maxwellNorah ★99 February 22, 2020 9:28 AM

Perhaps this should be labeled as a medium difficulty problem, was hesitating about reversing the linked list seeing it was easy :p

17 Share Reply

SHOW 1 REPLY

its_dreese ★10 February 5, 2020 5:19 AM

To do this is O(1) space this is definitely not easy. But thanks for the post @Hai_dee

10 Share Reply

SHOW 1 REPLY

petercdcn ★7 January 22, 2020 7:57 PM

This is the Best solution post I have seen so far. Great job @Hai_dee

5 Share Reply

mangoslicer ★40 January 1, 2020 5:50 PM

This is a top notch post. Thanks

2 Share Reply

guanyuWANG ★4 December 28, 2019 4:45 AM

I learned a lot, great post!

2 Share Reply

mrkrispy ★1 June 17, 2020 7:17 AM

I loved the solutions here.
This problem was a lot of fun. Very clever

1 Share Reply

xiayizju ★13 April 19, 2020 1:38 AM

for the approach 1, why we can't use == and != to compare the element in the ArrayList, the type of the element is Integer, but it should be able to be automatically changed to int when we compare using == and !=. I tried and found out almost all the test cases pass except [-129,-129], can someone explain why? Thx

1 Share Reply

SHOW 2 REPLIES

jakexiao ★1 March 7, 2020 9:30 PM

Time complexity: O(n)
Space complexity: O(1)

public class Solution {
public bool IsPalindrome(ListNode head) {

Read More

1 Share Reply

Air-man ★7 January 8, 2020 5:50 AM

Great post! I wish any solution should be written like this, for rookies like me.

1 Share Reply

‹ 1 2 3 ›