Interview Cake

You have a singly-linked list and want to check if it contains a cycle.

A singly-linked list is built with nodes, where each `node` has:

- `node.next`—the next node in the list.
- `node.value`—the data held in the node. For example, if our linked list stores people in line at the movies, `node.value` might be the person's name.

For example:

Python

```
class LinkedListNode(object):

    def __init__(self, value):
        self.value = value
        self.next  = None
```

A **cycle** occurs when a node's `next` points *back to a previous node in the list*. The linked list is no longer linear with a beginning and end—instead, it cycles through a loop of nodes.

Write a function `contains_cycle()` that takes the first node in a singly-linked list and returns a boolean indicating whether the list contains a cycle.

Gotchas

Careful—a cycle can occur in the *middle* of a list, or it can simply mean the last node links back to the first node. Does your function work for both?

We can do this in $O(n)$ time and $O(1)$ space!

Breakdown

Because a cycle could result from the last node linking to the first node, we might need to look at every node before we even see the start of our cycle again. So it seems like we can't do better than $O(n)$ runtime.

How can we track the nodes we've already seen?

Using a set, we could store all the nodes we've seen so far. The algorithm is simple:

1. If the current node is already in our set, we have a cycle. Return `True`.
2. If the current node is `None` we've hit the end of the list. Return `False`.
3. Else throw the current node in our set and keep going.

What are the time and space costs of this approach? Can we do better?

Our runtime is $O(n)$, the best we can do. But our space cost is also $O(n)$. Can we get our space cost down to $O(1)$ by storing a *constant* number of nodes?

Think about a *looping* list and a *linear* list. What happens when you traverse one versus the other?

A linear list has an *end*—a node that doesn't have a `next` node. But a looped list will run forever. We know we don't have a loop if we ever reach an end node, but how can we tell if we've run into a loop?

We can't just run our function for a really long time, because we'd never really know with certainty if we were in a loop or just a really long list.

Imagine that you're running on a long, mountainous running trail that happens to be a loop. What are some ways you can tell you're running in a loop?

One way is to **look for landmarks**. You could remember one specific point, and if you pass that point again, you know you're running in a loop. Can we use that principle here?

Well, our cycle can occur *after* a non-cyclical "head" section in the beginning of our linked list. So we'd need to make sure we chose a "landmark" node that is in the cyclical "tail" and not in the non-cyclical "head." That seems impossible unless we *already know* whether or not there's a cycle...

Think back to the running trail. Besides landmarks, what are some other ways you could tell you're running in a loop? What if you had **another runner**? (Remember, it's a *singly*-linked list, so no running backwards!)

A tempting approach could be to have the other runner stop and act as a "landmark," and see if you pass her again. But we still have the problem of making sure our "landmark" is in the loop and not in the non-looping beginning of the trail.

What if our "landmark" runner moves continuously but *slowly*?

If we sprint *quickly* down the trail and the landmark runner jogs *slowly*, we will eventually "lap" (catch up to) the landmark runner!

But what if there isn't a loop?

Then we (the faster runner) will simply hit the end of the trail (or linked list).

So let's make two variables, `slow_runner` and `fast_runner`. We'll start both on the first node, and every time `slow_runner` advances one node, we'll have `fast_runner` advance *two* nodes.

If `fast_runner` catches up with `slow_runner`, we know we have a loop. If not, eventually `fast_runner` will hit the end of the linked list and we'll know we *don't* have a loop.

This is a complete solution! Can you code it up?

Make sure the function eventually terminates in all cases!

Solution

We keep two pointers to nodes (we'll call these "runners"), both starting at the first node. Every time `slow_runner` moves one node ahead, `fast_runner` moves *two* nodes ahead.

If the linked list has a cycle, `fast_runner` will "lap" (catch up with) `slow_runner`, and they will momentarily equal each other.

If the list does not have a cycle, `fast_runner` will reach the end.

Python ▾

```
def contains_cycle(first_node):
    # Start both runners at the beginning
    slow_runner = first_node
    fast_runner = first_node

    # Until we hit the end of the list
    while fast_runner is not None and fast_runner.next is not None:
        slow_runner = slow_runner.next
        fast_runner = fast_runner.next.next

        # Case: fast_runner is about to "lap" slow_runner
        if fast_runner is slow_runner:
            return True

    # Case: fast_runner hit the end of the list
    return False
```

Complexity

$O(n)$ time and $O(1)$ space.

The runtime analysis is a little tricky. The worst case is when we *do* have a cycle, so we don't return until `fast_runner` equals `slow_runner`. But how long will that take?

First, we notice that when both runners are circling around the cycle **`fast_runner` can never skip over `slow_runner`**. Why is this true?

Suppose `fast_runner` *had just* skipped over `slow_runner`. `fast_runner` would only be 1 node ahead of `slow_runner`, since their speeds differ by only 1. So we would have something like this:

```
[ ] -> [s] -> [f]
```

What would the step right *before* this "skipping step" look like? `fast_runner` would be 2 nodes back, and `slow_runner` would be 1 node back. But wait, that means they would be at *the same node*! So `fast_runner` *didn't* skip over `slow_runner`! (This is a proof by contradiction.)

Since `fast_runner` can't skip over `slow_runner`, *at most* `slow_runner` will run around the cycle once and `fast_runner` will run around twice. This gives us a runtime of $O(n)$.

For space, we store two variables no matter how long the linked list is, which gives us a space cost of $O(1)$.

Bonus

1. How would you detect the *first node* in the cycle? Define the first node of the cycle as the one closest to the head of the list.
2. Would the program always work if the fast runner moves *three* steps every time the slow runner moves one step?
3. What if instead of a simple linked list, you had a structure where each node could have several "`next`" nodes? This data structure is called a "directed graph." How would you test if your directed graph had a cycle?

What We Learned

Some people have trouble coming up with the "two runners" approach. That's expected—it's somewhat of a blind insight. Even great candidates might need a few hints to get all the way there. And that's fine.

Remember that the coding interview is a *dialogue*, and sometimes your interviewer *expects* she'll have to offer some hints along the way.

One of the most impressive things you can do as a candidate is listen to a hint, fully understand it, and take it to its next logical step. Interview Cake gives you lots of opportunities to practice this. Don't be shy about showing *lots* of hints on our exercises—that's what they're there for!

Ready for more?

Check out our full course ➞

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.