Interview Cake

Find a duplicate, Space Edition™ BEAST MODE

In Find a duplicate, Space Edition™ (find-duplicate-optimize-for-space), we were given an array of integers where:

1. the integers are in the range $1..n$
2. the array has a length of $n + 1$

These properties mean the array *must have at least 1 duplicate*. Our challenge was to find a duplicate number, while optimizing for *space*. We used a divide and conquer approach, iteratively cutting the array in half to find a duplicate integer in $O(n \lg n)$ time and $O(1)$ space (sort of a modified binary search).

But we can actually do *better*. **We can find a duplicate integer in $O(n)$ time while keeping our space cost at $O(1)$.**

This is a tricky one to derive (unless you have a strong background in graph theory), so we'll get you started:

Imagine each item in the array as a node in a linked list. In any linked list,

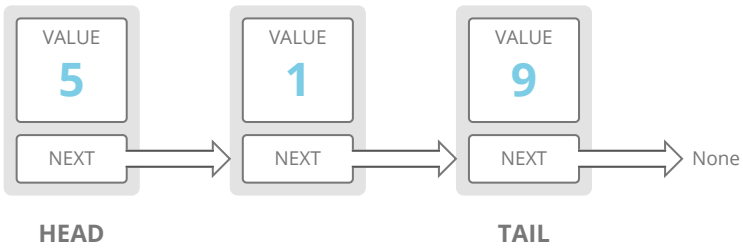
A **linked list** organizes items sequentially, with each item storing a pointer to the next one.

Picture a linked list like a chain of paperclips linked together. It's quick to add another paperclip to the top or bottom. It's even quick to insert one in the middle—just disconnect the chain at the middle link, add the new paperclip, then reconnect the other half.

An item in a linked list is called a **node**. The first node is called the **head**. The last node is called the **tail**.

	Worst Case
space	$O(n)$
prepend	$O(1)$
append	$O(1)$
lookup	$O(n)$
insert	$O(n)$
delete	$O(n)$

> Confusingly, *somtimes* people use the word **tail** to refer to "the whole rest of the list after the head."



> Unlike an array, consecutive items in a linked list are not necessarily next to each other in memory.

Strengths:

- **Fast operations on the ends**. Adding elements at either end of a linked list is $O(1)$. Removing the first element is also $O(1)$.
- **Flexible size**. There's no need to specify how many elements you're going to store ahead of time. You can keep adding elements as long as there's enough space on the machine.

Weaknesses:

- **Costly lookups**. To access or edit an item in a linked list, you have to take $O(i)$ time to walk from the head of the list to the ith item.

Uses:

- **Stacks (/concept/stack) and queues (/concept/queue)** only need fast operations on the ends, so linked lists are ideal.

In JavaScript

Most languages (including JavaScript) don't provide a linked list implementation. Assuming we've already implemented our own, here's how we'd construct the linked list above:

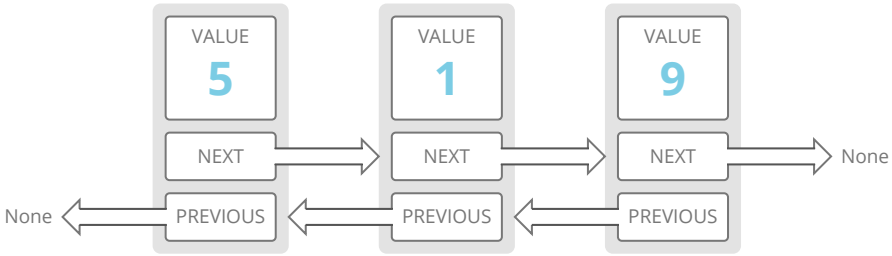
```javascript
const a = new LinkedListNode(5);
const b = new LinkedListNode(1);
const c = new LinkedListNode(9);

a.next = b;
b.next = c;
```

Doubly Linked Lists

In a basic linked list, each item stores a single pointer to the next element.

In a **doubly linked list**, items have pointers to the next *and the previous* nodes.



Doubly linked lists allow us to traverse our list *backwards*. In a *singly* linked list, if you just had a pointer to a node in the *middle* of a list, there would be *no way* to know what nodes came before it. Not a problem in a doubly linked list.

Not cache-friendly

Most computers have caching systems that make reading from sequential addresses in memory faster than reading from scattered addresses (/article/data-structures-coding-interview#ram).

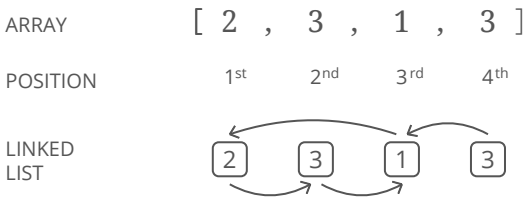
Array (/concept/array) items are always located right next to each other in computer memory, but linked list nodes can be scattered all over.

So iterating through a linked list is usually quite a bit slower than iterating through the items in an array, even though they're both theoretically $O(n)$ time.

each node has a **value** and a **"next"** pointer. In *this* case:

- The **value** is the *integer* from the array.
- The **"next"** pointer points to the **value-eth** node in the list (numbered starting from 1). For example, if our value was 3, the "next" node would be the *third* node.

Here's a full example:



> **Notice we're using "positions" and not "indices."** For this problem, we'll use the word "position" to mean something *like* "index," but different: indices start at 0, while positions start at 1. More rigorously: `position` = `index` + 1.

Using this, **find a duplicate integer in $O(n)$ time while keeping our space cost at $O(1)$.**

> **Drawing pictures will help a lot with this one.** Grab some paper and pencil (or a whiteboard, if you have one).

Gotchas

We don't need any new data structures. Your final space cost *must* be $O(1)$.

We can do this without destroying the input.

Breakdown

Here are a few sample arrays. Try drawing them out as linked lists:

```
[3, 4, 2, 3, 1, 5]
[3, 1, 2, 2]
[4, 3, 1, 1, 4]
```

JavaScript

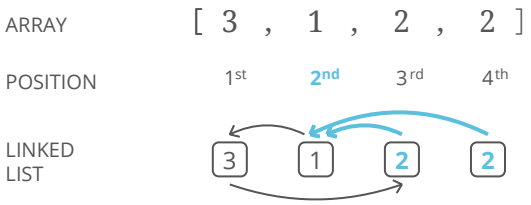
Look for patterns. Then think about how we might use those patterns to find a duplicate number in our array.

When a *value* is repeated, how will that affect the structure of our linked list?

If two nodes have the same *value*, their *next* pointers will point to the same node!

So if we can find a node with *two incoming* next pointers, we know the *position* of that node is a duplicate integer in our array.

For example, if there are two 2s in our array, the node in the 2nd position will have two incoming pointers.



Alright, we're on to something. But hold on—creating a linked list would take $O(n)$ space, and we don't want to change our space cost from $O(1)$.

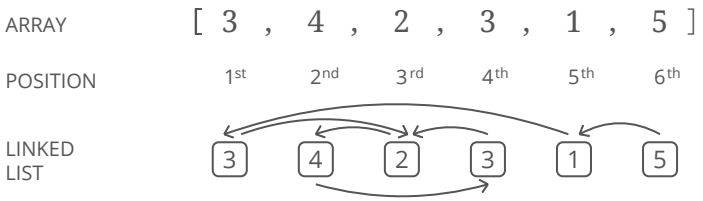
No problem—turns out we can just *think* of the array as a linked list, and traverse it without *actually* creating a new data structure.

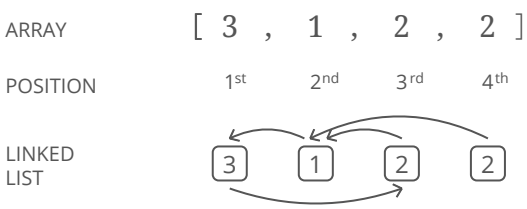
> If you're stuck on figuring out how to traverse the array like a linked list, don't sweat it too much. Just use a real linked list for now, while we finish deriving the algorithm.

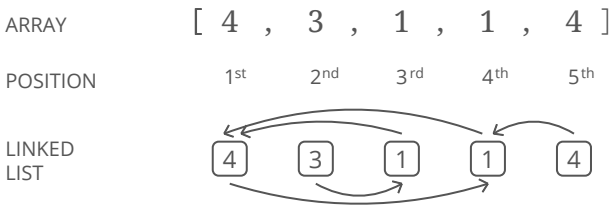
Ok, so we figured out that the ***position* of a node with multiple incoming pointers must be a duplicate**. If we can find a node with multiple incoming pointers in a *constant* number of walks through our array, we can find a duplicate value in $O(n)$ time.

How can we find a node with multiple incoming pointers?

Let's look back at those sample arrays and their corresponding linked lists, which we drew out to look for patterns:







Are there any patterns that might help us find a node with two incoming pointers?

Here's a pattern: **the last node never has any incoming pointers**. This makes sense—since the array has a length $n + 1$ and all the values are n or less, there can never be a pointer to the last position. If n is 5, the length of the array is 6 but there can't be a value 6 so no pointer will ever point to the 6th node. Since it has no incoming pointers, **we should treat the last position in our array as the "head" (starting point) of our linked list.**

Here's another pattern: **there's *never an end* to our list**. No pointer ever points to `null`. Every node has a value in the range $1..n$, so every node points to another node (or to itself). **Since the list goes on forever, it must have a cycle (a loop)**. Here are the cycles in our example lists:

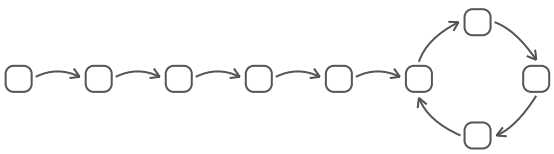
Can we use these cycles to find a duplicate value?

If we walk through our linked list, starting from the head, at some point we will *enter* our cycle. Try tracing that path on the example lists above. Notice anything special about the *first* node we hit when we *enter* the cycle?

The first node in the cycle always has *at least two incoming pointers*!

We're getting close to an algorithm for finding a duplicate value. How can we find the *beginning* of a cycle?

Again, drawing an example is helpful here:

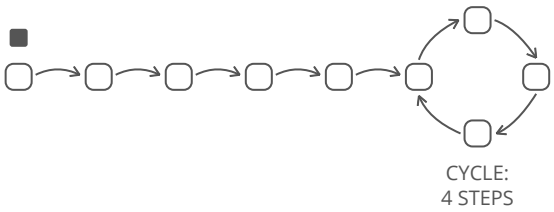


If we were traversing this list and wanted to know *if* we were inside a cycle, that would be pretty easy—we could just remember our current position and keep stepping ahead to see if we get to that position again.

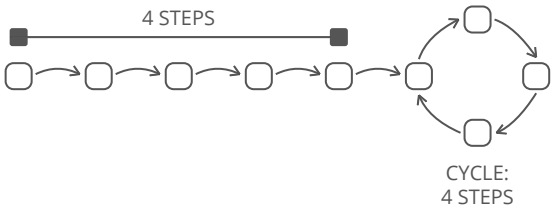
But our problem is a little trickier—we need to know the *first* node in the cycle.

What if we knew the **length of the cycle**?

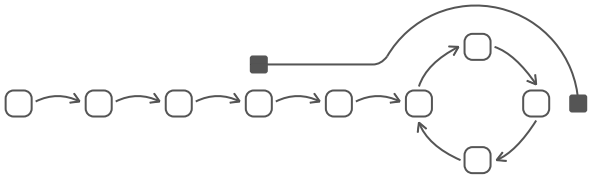
If we knew the length of the cycle, we could use the "stick approach" to start at the head of the list and find the first node. We use two pointers. One pointer starts at the head of the list:



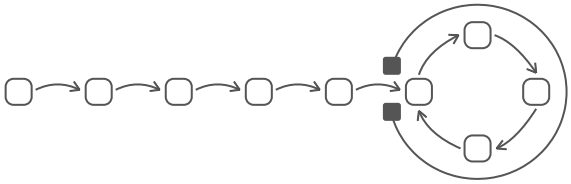
Then we lay down a "stick" with the same length as the cycle, by starting the second pointer at the end. So here, for example, the second pointer is starting 4 steps ahead because the cycle has 4 steps:



Then we move the stick along the list by advancing the two pointers at the same speed (one node at a time).



When the first pointer reaches the first node in the cycle, the second pointer will have circled around exactly once. The stick wraps around the cycle, and the two ends are in the same place: *the start of the cycle*.



We already know where the head of our list is (the last position in the list) so we just need the length of the cycle. **How can we find the length of a cycle?**

If we can get *inside* the cycle, we can just remember a position and count how many steps it takes to get back to that position.

How can we make sure we've gotten inside a cycle?

Well, there *has* to be a cycle in our list, and at the *latest*, the cycle is *just the last node* we hit as we traverse the list from the head:

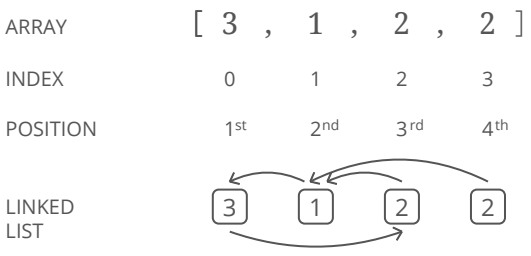
So we can just start at the head and walk n steps. By then we'll *have* to be inside a cycle.

Alright, we've pieced together an **entire strategy to find a duplicate integer!** Working backward:

A. We know the *position* of a node with multiple incoming pointers is a **duplicate in our array** because the nodes that pointed to it must have the same value.
B. We find a **node with multiple incoming pointers** by finding the first node in a cycle.
C. We find the **first node in a cycle** by finding the length of the cycle and advancing two pointers: one starting at the head of the linked list, and the other starting ahead as many steps as there are steps in the cycle. The pointers will meet at the first node in the cycle.
D. We find the **length of a cycle** by remembering a position inside the cycle and counting the number of steps it takes to get back to that position.
E. We **get inside a cycle** by starting at the head and walking n steps. We know the **head of the list is at position $n + 1$**.

Can you implement this? And remember—we won't want to *actually* create a linked list. Here's how we can traverse our array *as if* it *were* a linked list.↴

Let's take an example array and try walking through it as if it were a linked list:



Remember that our input array is defined as having a length $n + 1$. So we know n is 3 because the array has a length of 4.

The *head* (starting point) is the *4th node*, since it has no incoming pointers. We'll want to go from the 4th position to the 2nd position to the 1st position to the 3rd position. Or, in terms of *indices in our array*, we'll want to go from index 3 to index 1 to index 0 to index 2.

Let's get set up:

JavaScript

```
var n = 3;
var intArray = [3, 1, 2, 2];

// start at the head
var currentPosition = 4;
```

Now we need to take n steps:

JavaScript

```
for (var i = 0; i < n; i++) {
    // step ahead
}
```

On our first step, `currentPosition` is 4 and the value at the 4th position is 2, so we want to update `currentPosition` to 2. The only trick is that we need to convert our *position* to an *index*. That's easy—we just subtract 1 (the 1st position of an array is index 0).

JavaScript

```
for (var i = 0; i < n; i++) {

    // subtract 1 from the current position to get
    // the current index
    var currentIndex = currentPosition - 1;

    // take a step, updating the current position
    // to the *value* at its previous position
    currentPosition = intArray[currentIndex];
}
```

So if we're at a `currentPosition`, the next position we want to go to is the value at the *index* `currentPosition - 1`. We can refactor this to 1 line and have this general way to take n steps forward in our array as if it were a linked list:

JavaScript

```
for (var i = 0; i < n; i++) {
    currentPosition = intArray[currentPosition - 1];
}
```

To get inside a cycle (step E above), we identify n, start at the head (the node in position $n + 1$), and walk n steps.

JavaScript

```
function findDuplicate(intArray) {

    const n = intArray.length - 1;

    // STEP 1: GET INSIDE A CYCLE
    // start at position n+1 and walk n steps to
    // find a position guaranteed to be in a cycle
    var positionInCycle = n + 1;
    for (var i = 0; i < n; i++) {
        positionInCycle = intArray[positionInCycle - 1];
    }
}
```

Now we're guaranteed to be inside a cycle. To find the cycle's length (D), we remember the current position and step ahead until we come back to that same position, counting the number of steps.

JavaScript

```
function findDuplicate(intArray) {

    const n = intArray.length - 1;

    // STEP 1: GET INSIDE A CYCLE
    // start at position n+1 and walk n steps to
    // find a position guaranteed to be in a cycle
    var positionInCycle = n + 1;
    for (var i = 0; i < n; i++) {
        positionInCycle = intArray[positionInCycle - 1];
    }

    // STEP 2: FIND THE LENGTH OF THE CYCLE
    // find the length of the cycle by remembering a position in the cycle
    // and counting the steps it takes to get back to that position
    const rememberedPositionInCycle = positionInCycle;
    var currentPositionInCycle = intArray[positionInCycle - 1];  // 1 step ahead
    var cycleStepCount = 1;

    while (currentPositionInCycle !== rememberedPositionInCycle) {
        currentPositionInCycle = intArray[currentPositionInCycle - 1];
        cycleStepCount += 1;
    }
}
```

Now we have the *head* and the *length* of the cycle. We need to find the *first node* in the cycle (C). We set up 2 pointers: 1 at the head, and 1 ahead as many steps as there are nodes in the cycle. These two pointers form our "stick."

JavaScript

```
// STEP 3: FIND THE FIRST NODE OF THE CYCLE
// start two pointers
//   (1) at position n+1
//   (2) ahead of position n+1 as many steps as the cycle's length
var pointerStart = n + 1;
var pointerAhead = n + 1;
for (var i = 0; i< cycleStepCount; i++) {
    pointerAhead = intArray[pointerAhead - 1];
}
```

Alright, we just need to find to the first node in the cycle (B), and return a duplicate value (A)!

Solution

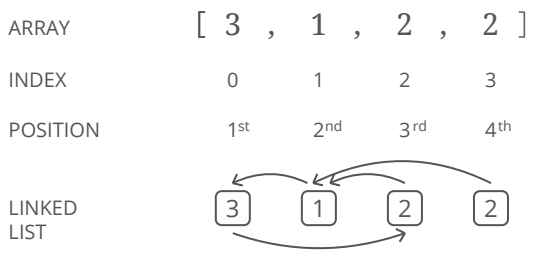
We treat the input array as a linked list like we described at the top in the problem.

To find a duplicate integer:

A. We know the *position* of a node with multiple incoming pointers is a **duplicate in our array** because the nodes that pointed to it must have the same value.
B. We find a **node with multiple incoming pointers** by finding the first node in a cycle.
C. We find the **first node in a cycle** by finding the length of the cycle and advancing two pointers: one starting at the head of the linked list, and the other starting ahead as many steps as there are nodes in the cycle. The pointers will meet at the first node in the cycle.
D. We find the **length of a cycle** by remembering a position inside the cycle and counting the number of steps it takes to get back to that position.
E. We **get inside a cycle** by starting at the head and walking n steps. We know the **head of the list is at position $n + 1$**.

We want to *think* of our array as a linked list but we don't want to *actually* use up all that space, so we traverse our array *as if* it were a linked list.

Let's take an example array and try walking through it as if it were a linked list:



Remember that our input array is defined as having a length $n + 1$. So we know n is 3 because the array has a length of 4.

The *head* (starting point) is the *4th node*, since it has no incoming pointers. We'll want to go from the 4th position to the 2nd position to the 1st position to the 3rd position. Or, in terms of *indices in our array*, we'll want to go from index 3 to index 1 to index 0 to index 2.

Let's get set up:

```
var n = 3;
var intArray = [3, 1, 2, 2];

// start at the head
var currentPosition = 4;
```

Now we need to take n steps:

```
for (var i = 0; i < n; i++) {
    // step ahead
}
```

On our first step, `currentPosition` is 4 and the value at the 4th position is 2, so we want to update `currentPosition` to 2. The only trick is that we need to convert our *position* to an *index*. That's easy—we just subtract 1 (the 1st position of an array is index 0).

```
for (var i = 0; i < n; i++) {

    // subtract 1 from the current position to get
    // the current index
    var currentIndex = currentPosition - 1;

    // take a step, updating the current position
    // to the *value* at its previous position
    currentPosition = intArray[currentIndex];
}
```

So if we're at a `currentPosition`, the next position we want to go to is the value at the *index* `currentPosition - 1`. We can refactor this to 1 line and have this general way to take n steps forward in our array as if it were a linked list:

```
for (var i = 0; i < n; i++) {
    currentPosition = intArray[currentPosition - 1];
}
```

by converting positions to indices.

JavaScript

```
function findDuplicate(intArray) {

    const n = intArray.length - 1;

    // STEP 1: GET INSIDE A CYCLE
    // start at position n+1 and walk n steps to
    // find a position guaranteed to be in a cycle
    var positionInCycle = n + 1;
    for (var i = 0; i < n; i++) {

        // we subtract 1 from the current position to step ahead:
        // the 2nd *position* in an array is *index* 1
        positionInCycle = intArray[positionInCycle - 1];
    }

    // STEP 2: FIND THE LENGTH OF THE CYCLE
    // find the length of the cycle by remembering a position in the cycle
    // and counting the steps it takes to get back to that position
    const rememberedPositionInCycle = positionInCycle;
    var currentPositionInCycle = intArray[positionInCycle - 1];  // 1 step ahead
    var cycleStepCount = 1;

    while (currentPositionInCycle !== rememberedPositionInCycle) {
        currentPositionInCycle = intArray[currentPositionInCycle - 1];
        cycleStepCount += 1;
    }

    // STEP 3: FIND THE FIRST NODE OF THE CYCLE
    // start two pointers
    //   (1) at position n+1
    //   (2) ahead of position n+1 as many steps as the cycle's length
    var pointerStart = n + 1;
    var pointerAhead = n + 1;
    for (var i = 0; i< cycleStepCount; i++) {
        pointerAhead = intArray[pointerAhead - 1];
    }

    // advance until the pointers are in the same position
    // which is the first node in the cycle
    while (pointerStart !== pointerAhead) {
        pointerStart = intArray[pointerStart - 1];
        pointerAhead = intArray[pointerAhead - 1];
    }
```

```
    // since there are multiple values pointing to the first node
    // in the cycle, its position is a duplicate in our array
    return pointerStart;
}
```

Complexity

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

Our space cost is $O(1)$ because all of our additional variables are integers, which each take constant space.

For our runtime, we iterate over the array a constant number of times, and each iteration takes $O(n)$ time in its worst case. So we traverse the linked list more than once, but it's still a *constant* number of times—about 3.

Bonus

There another approach using randomized algorithms that is $O(n)$ time and $O(1)$ space. Can you come up with that one? (Hint: You'll want to focus on the median.)

What We Learned

This one's pretty crazy. It's hard to imagine an interviewer expecting you to get all the way through this question without help.

But just because it takes a few hints to get to the answer doesn't mean a question is "too hard." Some interviewers *expect* they'll have to offer a few hints.

So if you get a hint in an interview, just relax and listen. The most impressive thing you can do is drop what you're doing, fully understand the hint, and then run with it.

Ready for more?

Check out our full course ➡

Want more coding interview help?

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