Interview Cake

← course home (/table-of-contents#section_array-and-string-manipulation_question_cafe-order-checker)

My cake shop is so popular, I'm adding some tables and hiring wait staff so folks can have a cute sit-down cake-eating experience.

I have two registers: one for take-out orders, and the other for the other folks eating inside the cafe. All the customer orders get combined into one list for the kitchen, where they should be handled first-come, first-served.

Recently, some customers have been complaining that people who placed orders after them are getting their food first. Yikes—that's not good for business!

To investigate their claims, one afternoon I sat behind the registers with my laptop and recorded:

- The take-out orders as they were entered into the system and given to the kitchen. (`takeOutOrders`)
- The dine-in orders as they were entered into the system and given to the kitchen. (`dineInOrders`)
- Each customer order (from either register) as it was finished by the kitchen. (`servedOrders`)

Given all three arrays, write a function to check that my service is first-come, first-served. All food should come out in the same order customers requested it.

We'll represent each customer order as a unique integer.

As an example,

```
Take Out Orders: [1, 3, 5]
 Dine In Orders: [2, 4, 6]
  Served Orders: [1, 2, 4, 6, 5, 3]
```

would *not* be first-come, first-served, since order 3 was requested before order 5 but order 5 was served first.

But,

```
Take Out Orders: [17, 8, 24]
 Dine In Orders: [12, 19, 2]
  Served Orders: [17, 8, 12, 19, 24, 2]
```

would be first-come, first-served.

> Note: Order numbers are arbitrary. They do **not** have to be in increasing order.

Gotchas

Watch out for bugs because your index is outside an array! Will your function ever try to grab the 0th item from an empty array, or the n^{th} item from an array with n elements (where the last index would be $n-1$)?

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

Did you come up with a recursive solution? Keep in mind that you may be incurring a hidden space cost (probably $O(n)$) in the call stack! You can avoid this using an iterative approach.

Breakdown

How can we re-phrase this problem in terms of smaller subproblems?

Breaking the problem into smaller subproblems will clearly involve reducing the size of at least one of our lists of customer order numbers. So to start, let's try taking the first customer order out of `servedOrders`.

What should be true of this customer order number if my service is first-come, first-served?

If my cake cafe is first-come, first-served, then the first customer order finished (first item in `servedOrders`) has to either be the first take-out order entered into the system (`takeOutOrders[0]`) or the first dine-in order entered into the system (`dineInOrders[0]`).

Once we can check the *first* customer order, how can we verify the remaining ones?

Let's "throw out" the first customer order from `servedOrders` as well as the customer order it matched with from the beginning of `takeOutOrders` or `dineInOrders`. That customer order is now "accounted for."

Now we're left with a smaller version of the original problem, which we can solve using the same approach! So we keep doing this over and over until we exhaust `servedOrders`. If we get to the end and every customer order "checks out," we return `true`.

How do we implement this in code?

Now that we have a problem that's the same as the original problem except smaller, our first thought might be to use recursion. All we need is a base case. What's our base case?

We stop when we run out of customer orders in our `servedOrders`. So that's our base case: when we've checked all customer orders in `servedOrders`, we return `true` because we know all of the customer orders have been "accounted for."

JavaScript

```
function isFirstComeFirstServed(takeOutOrders, dineInOrders, servedOrders) {

    // base case
    if (servedOrders.length === 0) {
        return true;
    }

    // if the first order in servedOrders is the same as the
    // first order in takeOutOrders
    // (making sure first that we have an order in takeOutOrders)
    if (takeOutOrders.length && takeOutOrders[0] === servedOrders[0]) {

        // take the first order off takeOutOrders and servedOrders and recurse
        return isFirstComeFirstServed(takeOutOrders.slice(1), dineInOrders, servedOrde

    // if the first order in servedOrders is the same as the first
    // in dineInOrders
    } else if (dineInOrders.length && dineInOrders[0] === servedOrders[0]) {

        // take the first order off dineInOrders and servedOrders and recurse
        return isFirstComeFirstServed(takeOutOrders, dineInOrders.slice(1), servedOrde

    // first order in servedOrders doesn't match the first in
    // takeOutOrders or dineInOrders, so we know it's not first-come, first-served
    } else {
        return false;
    }
}
```

This solution will work. But we can do better.

Before we talk about optimization, note that our inputs are probably pretty small. This function will take hardly any time or space, even if it *could be* more efficient. In industry, especially at small startups that want to move quickly, optimizing this might be considered a "premature optimization." Great engineers have both the *skill* to see how to optimize their code and the *wisdom* to know when those optimizations aren't worth it. At this point in the interview I recommend saying, "I think we can optimize this a bit further, although given the nature of the input this probably won't be very resource-intensive anyway...should we talk about optimizations?"

Suppose we *do* want to optimize further. What are the time and space costs to beat? This function will take $O(n^2)$ time and $O(n^2)$ additional space.

Whaaaaat? Yeah. Take a look at this snippet:

```
JavaScript
return isFirstComeFirstServed(takeOutOrders.slice(1), dineInOrders, servedOrders.slic
```

In particular this expression:

```
JavaScript
takeOutOrders.slice(1);
```

That's a slice, and it costs $O(m)$ time and space, where m is the size of the resulting array. That's going to determine our overall time and space cost here—the rest of the work we're doing is constant space and time.

In our recursing we'll build up n frames on the call stack. Each of those frames will hold a *different slice* of our original `servedOrders` (and `takeOutOrders` and `dineInOrders`, though we only slice one of them in each recursive call).

So, what's the total time and space cost of all our slices?

If `servedOrders` has n items to start, taking our first slice takes $n - 1$ time and space (plus half that, since we're also slicing one of our halves—but that's just a constant multiplier so we can ignore it). In our second recursive call, slicing takes $n - 2$ time and space. Etc.

So our total time and space cost for slicing comes to:

$$(n-1) + (n-2) + \ldots + 2 + 1$$

This is a common series that turns out to be $O(n^2)$.

We can do better than this $O(n^2)$ time and space cost. One way we could to that is to avoid slicing and instead keep track of indices in the array:

JavaScript

```
function isFirstComeFirstServed(takeOutOrders, dineInOrders, servedOrders, servedOrder
    servedOrdersIndex = (typeof servedOrdersIndex !== 'undefined') ? servedOrdersInde
    takeOutOrdersIndex = (typeof takeOutOrdersIndex !== 'undefined') ? takeOutOrdersIr
    dineInOrdersIndex = (typeof dineInOrdersIndex !== 'undefined') ? dineInOrdersInde

    // base case we've hit the end of servedOrders
    if (servedOrdersIndex === servedOrders.length) {
        return true;
    }

    // if we still have orders in takeOutOrders
    // and the current order in takeOutOrders is the same
    // as the current order in servedOrders
    if ((takeOutOrdersIndex < takeOutOrders.length) &&
            (takeOutOrders[takeOutOrdersIndex] === servedOrders[servedOrdersIndex])) 
        takeOutOrdersIndex++;


    // if we still have orders in dineInOrders
    // and the current order in dineInOrders is the same
    // as the current order in servedOrders
    } else if ((dineInOrdersIndex < dineInOrders.length) &&
            (dineInOrders[dineInOrdersIndex] === servedOrders[servedOrdersIndex])) {
        dineInOrdersIndex++;

    // if the current order in servedOrders doesn't match
    // the current order in takeOutOrders or dineInOrders, then we're not
    // serving in first-come, first-served order.
    } else {
        return false;
    }

    // the current order in servedOrders has now been "accounted for"
    // so move on to the next one
    servedOrdersIndex++;
    return isFirstComeFirstServed(takeOutOrders, dineInOrders, servedOrders, servedOr
}
```

So now we're down to $O(n)$ time, but we're still taking $O(n)$ space in the call stack because of our recursion. We can rewrite this as an iterative function to get that memory cost down to $O(1)$.

What's happening in each iteration of our recursive function? Sometimes we're taking a customer order out of `takeOutOrders` and sometimes we're taking a customer order out of `dineInOrders`, but we're *always* taking a customer order out of `servedOrders`.

So what if instead of taking customer orders out of `servedOrders` 1-by-1, we *iterated over them*?

That should work. But are we missing any edge cases?

What if there are *extra* orders in `takeOutOrders` or `dineInOrders` that don't appear in `servedOrders`? Would our kitchen be first-come, first-served then?

Maybe.

It's possible that our data doesn't include everything from the afternoon service. Maybe we stopped recording data before every order that went into the kitchen was served. It would be reasonable to say that our kitchen is still first-come, first-served, since we don't have any evidence otherwise.

On the other hand, if our input is supposed to cover the entire service, then any orders that went into the kitchen but weren't served should be investigated. We don't want to accept people's money but not serve them!

> When in doubt, ask! This is a *great* question to talk through with your interviewer and shows that you're able to think through edge cases.

Both options are reasonable. In this writeup, we'll enforce that *every* order that goes into the kitchen (appearing in `takeOutOrders` or `dineInOrders`) should come out of the kitchen (appearing in `servedOrders`). How can we check that?

To check that we've served every order that was placed, we'll validate that when we finish iterating through `servedOrders`, we've exhausted `takeOutOrders` and `dineInOrders`.

Solution

We walk through `servedOrders`, seeing if each customer order *so far* matches a customer order from one of the two registers. To check this, we:

1. Keep pointers to the current index in `takeOutOrders`, `dineInOrders`, and `servedOrders`.
2. Walk through `servedOrders` from beginning to end.
3. If the current `order` in `servedOrders` is the same as the current customer order in `takeOutOrders`, increment `takeOutOrdersIndex` and keep going. This can be thought of as "checking off" the current customer order in `takeOutOrders` and `servedOrders`, reducing the problem to the remaining customer orders in the arrays.
4. Same as above with `dineInOrders`.
5. If the current `order` isn't the same as the customer order at the front of `takeOutOrders` or `dineInOrders`, we know something's gone wrong and we're not serving food first-come, first-served.
6. If we make it all the way to the end of `servedOrders`, we'll check that we've reached the end of `takeOutOrders` and `dineInOrders`. If every customer order checks out, that means we're serving food first-come, first-served.

JavaScript ▾

```
function isFirstComeFirstServed(takeOutOrders, dineInOrders, servedOrders) {
    var takeOutOrdersIndex = 0;
    var dineInOrdersIndex = 0;
    var takeOutOrdersMaxIndex = takeOutOrders.length - 1;
    var dineInOrdersMaxIndex = dineInOrders.length - 1;

    for (var i = 0; i < servedOrders.length; i++) {
        var order = servedOrders[i];

        // if we still have orders in takeOutOrders
        // and the current order in takeOutOrders is the same
        // as the current order in servedOrders
        if (takeOutOrdersIndex <= takeOutOrdersMaxIndex &&
                order === takeOutOrders[takeOutOrdersIndex]) {
            takeOutOrdersIndex++;

        // if we still have orders in dineInOrders
        // and the current order in dineInOrders is the same
        // as the current order in servedOrders
        } else if (dineInOrdersIndex <= dineInOrdersMaxIndex &&
                order === dineInOrders[dineInOrdersIndex]) {
            dineInOrdersIndex++;

        // if the current order in servedOrders doesn't match the current
        // order in takeOutOrders or dineInOrders, then we're not serving first-come,
        // first-served
        } else {
            return false;
        }
    }

    // check for any extra orders at the end of takeOutOrders or dineInOrders
    if (dineInOrdersIndex != dineInOrders.length ||
          takeOutOrdersIndex != takeOutOrders.length) {
        return false;
    }

    // all orders in servedOrders have been "accounted for"
    // so we're serving first-come, first-served!
    return true;
}
```

Complexity

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

Bonus

1. This assumes each customer order in `servedOrders` is unique. How can we adapt this to handle arrays of customer orders with *potential repeats*?
2. Our implementation returns `true` when all the items in `dineInOrders` and `takeOutOrders` are first-come first-served in `servedOrders` and `false` otherwise. That said, it'd be reasonable to throw an exception if some orders that went into the kitchen were never served, or orders were served but not paid for at either register. How could we check for those cases?
3. Our solution iterates through the customer orders from front to back. Would our algorithm work if we iterated from the back towards the front? Which approach is cleaner?

What We Learned

If you read the whole breakdown section, you might have noticed that our recursive function cost us extra space. If you missed that part, go back and take a look.

Be careful of the hidden space costs from a recursive function's call stack! If you have a solution that's recursive, see if you can save space by using an iterative algorithm instead.

⬅ course home (/table-of-contents)

Next up: Hashing and Hash Functions ➡ (/concept/hashing?course=fc1&section=hashing-and-hash-tables)

Want more coding interview help?