Interview Cake

Write a function `fib()` that takes an integer n and returns the nth Fibonacci number.

Let's say our Fibonacci series is 0-indexed and starts with 0. So:

```
fib(0);  // => 0
fib(1);  // => 1
fib(2);  // => 1
fib(3);  // => 2
fib(4);  // => 3
...
```

JavaScript

Gotchas

Our solution runs in n time.

There's a clever, more mathy solution that runs in $O(\lg n)$ time, but we'll leave that one as a bonus.

If you wrote a recursive function, think carefully about what it does. It might do repeat work, like computing `fib(2)` multiple times!

We can do this in $O(1)$ space. If you wrote a recursive function, there might be a hidden space cost in the call stack!

Breakdown

The nth Fibonacci number is defined in terms of the two *previous* Fibonacci numbers, so this seems to lend itself to recursion.

```
fib(n) = fib(n - 1) + fib(n - 2);
```

JavaScript

Can you write up a recursive solution?

As with any recursive function, we just need a base case and a recursive case:

1. **Base case:** n is 0 or 1. Return n.
2. **Recursive case:** Return `fib(n - 1) + fib(n - 2)`.

JavaScript

```
function fib(n) {
  if (n === 0 || n === 1) {
    return n;
  }
  return fib(n - 1) + fib(n - 2);
}
```

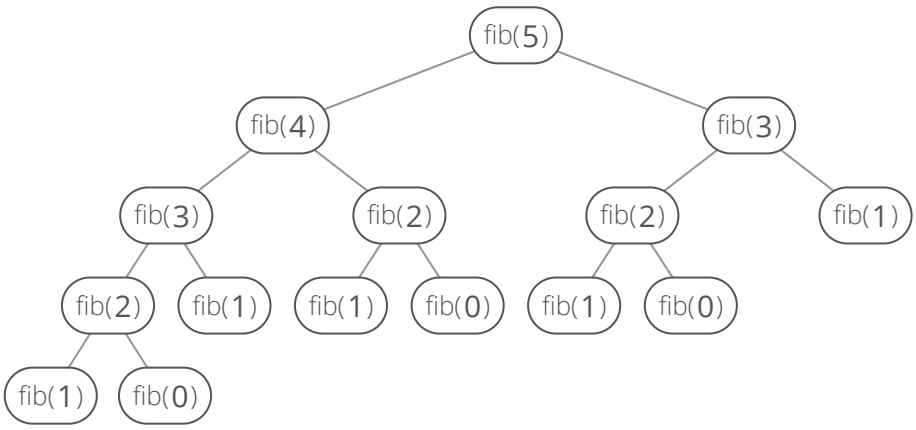
Okay, this'll work! What's our time complexity?

It's not super obvious. We might guess n, but that's not quite right. Can you see why?

Each call to `fib()` makes *two more calls*. Let's look at a specific example. Let's say $n = 5$. **If we call `fib(5)`, how many calls do we make in total?**

Try drawing it out as a tree where each call has two child calls, unless it's a base case.

Here's what the tree looks like:



We can notice this is a binary tree whose height is n, which means the total number of nodes is $O(2^n)$.

So our total runtime is $O(2^n)$. That's an "exponential time cost," since the n is *in an exponent*. Exponential costs are *terrible*. This is way worse than $O(n^2)$ or even $O(n^{100})$.

Our recurrence tree above essentially gets twice as big each time we add 1 to n. So as n gets really big, our runtime quickly spirals out of control.

The craziness of our time cost comes from the fact that we're doing so much repeat work. How can we avoid doing this repeat work?

We can memoize!↴

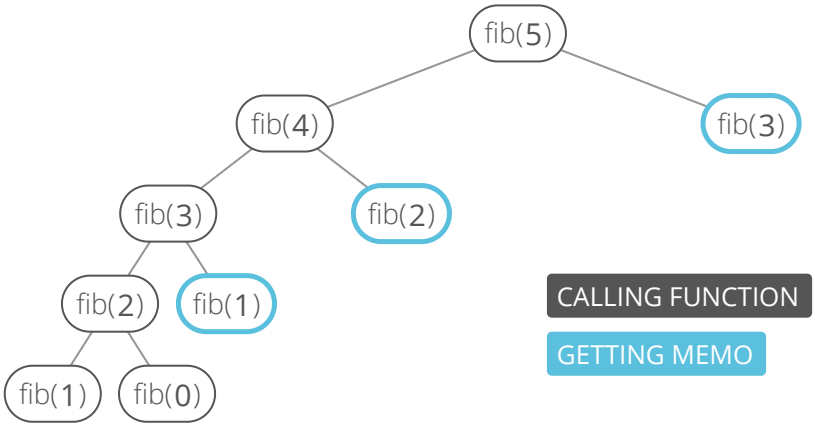
Let's wrap `fib()` in a class with an instance variable where we store the answer for any n that we compute:

JavaScript

```
class Fibber {
  constructor() {
    this.memo = {};
  }

  fib(n) {

    // Edge case: negative index
    if (n < 0) {
      throw new Error('Index was negative. No such thing as a negative index in a series.');
    }

    // Base case: 0 or 1
    else if (n === 0 || n === 1) {
      return n;
    }

    // See if we've already calculated this
    if (this.memo.hasOwnProperty(n)) {
      return this.memo[n];
    }

    const result = this.fib(n - 1) + this.fib(n - 2);

    // Memoize
    this.memo[n] = result;

    return result;
  }
}
```

What's our time cost now?

Our recurrence tree will look like this:



The computer will build up a call stack with `fib(5)`, `fib(4)`, `fib(3)`, `fib(2)`, `fib(1)`. Then we'll start returning, and on the way back up our tree we'll be able to compute each node's 2nd call to `fib()` in constant time by just looking in the memo. n time in total.

What about space? `memo` takes up n space. Plus we're still building up a call stack that'll occupy n space. Can we avoid one or both of these space expenses?

Look again at that tree. Notice that to calculate `fib(5)` we worked "down" to `fib(4)`, `fib(3)`, `fib(2)`, etc.

What if instead we *started* with `fib(0)` and `fib(1)` and worked "up" to n?

Solution

We use a bottom-up↴ approach, starting with the 0th Fibonacci number and iteratively computing subsequent numbers until we get to n.

JavaScript

```
function fib(n) {

  // Edge cases:
  if (n < 0) {
    throw new Error('Index was negative. No such thing as a negative index in a series.');
  } else if (n === 0 || n === 1) {
    return n;
  }

  // We'll be building the fibonacci series from the bottom up
  // So we'll need to track the previous 2 numbers at each step
  let prevPrev = 0;  // 0th fibonacci
  let prev = 1;      // 1st fibonacci
  let current;       // Declare current

  for (let i = 1; i < n; i++) {

    // Iteration 1: current = 2nd fibonacci
    // Iteration 2: current = 3rd fibonacci
    // Iteration 3: current = 4th fibonacci
    // To get nth fibonacci ... do n-1 iterations.
    current = prev + prevPrev;
    prevPrev = prev;
    prev = current;
  }

  return current;
}
```

Complexity

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

Bonus

- If you're good with matrix multiplication you can bring the time cost down even further, to $O(lg(n))$. Can you figure out how?

What We Learned

This one's a good illustration of the tradeoff we sometimes have between code cleanliness and efficiency.

We could use a cute, recursive function to solve the problem. But that would cost $O(2^n)$ time as opposed to n time in our final bottom-up solution. Massive difference!

In general, whenever you have a recursive solution to a problem, think about what's *actually happening on the call stack*. An iterative solution might be more efficient.