Write a function `fib()` that a takes an integer n and returns the nth fibonacci↴

The **Fibonacci series** is a numerical series where each item is the sum of the two previous items. It starts off like this:

$$0, 1, 1, 2, 3, 5, 8, 13, 21...$$

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
...
```

Gotchas

Our solution runs in $O(n)$ time.

Type code!

There's a clever, more mathey 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)
```

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)`.

```
def fib_recursive(n):
    if n in [1,0]:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)
```

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

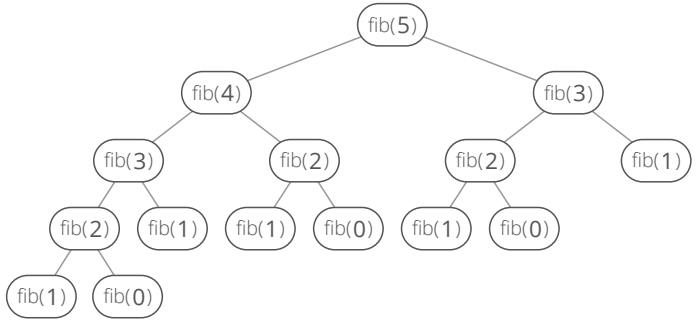
It's not super obvious. We might guess $O(n)$, but that's not quite right. Can you see why?

Type code!

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 $O(n)$, 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

```
Type code!
```

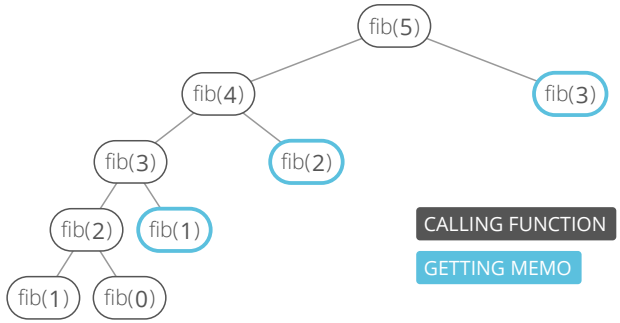
compute:

```
class Fibber:
    def __init__(self):
        self.memo = {}

    def fib(self, n):
        # edge case: negative index
        if n < 0:
            raise Exception("Index was negative. No such thing as a negative index in a ser

        # base case: 0 or 1
        elif n in [0,1]:
            return n

        # see if we've already calculated this
        if self.memo.has_key(n):
            return self.memo[n]

        result = self.fib(n-1) + self.fib(n-2)

        # memoize
        self.memo[n] = result

        return result
```

What's our time cost now?

Our recurrence tree will look like this:

Type code!



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. $O(n)$ time in total.

What about space? `self.memo` takes up $O(n)$ space. Plus we're still building up a call stack that'll occupy $O(n)$ 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.

```
Type code!
```

```
def fib_iterative(n):

    # edge cases:
    if n < 0:
        raise Exception("Index was negative. No such thing as a negative index in a series.

    elif n in [0,1]:
        return n

    prev = 0
    prev_prev = 1

    for index in range(n):
        current = prev + prev_prev
        prev_prev = 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?

We'll review this one again later

Type code!

Next question

Like this problem? Pass it on!

Share (https://www.facebook.com/sharer/sharer.php?u=https://www.interviewcake.com/question/nth-fibonacci) Tweet (http://twitter.com/share?text=This%20coding%20interview%20question%20on%20@interviewcake%20is%20pretty%20cool!&url=https://www.interviewcake.com/question/nth-fibonacci)

(https://www.facebook.com/interviewcake) (https://twitter.com/intervicake)

Type code!