**Write a function fib() that takes an integer n*n* and returns the n*n*th 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

...



**Gotchas**

Our solution runs in n*n* time.

There's a clever, more mathy solution that runs in O(\lg{n})*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)*O*(1) space. If you wrote a recursive function, there might be a hidden space cost in the call stack! ↴

**Breakdown**

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

def fib(n):

if n in [1, 0]:

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*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*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*n*, which means the total number of nodes is O(2^n)*O*(2*n*).

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

Our recurrence tree above essentially gets twice as big each time we add 1 to n*n*. So as n*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*n* that we compute:

class Fibber(object):

def \_\_init\_\_(self):

self.memo = {}

def fib(self, n):

if n < 0:

# Edge case: negative index

raise ValueError('Index was negative. No such thing as a '

'negative index in a series.')

elif n in [0, 1]:

# Base case: 0 or 1

return n

# See if we've already calculated this

if n in self.memo:

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:

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*n* time in total.

What about space? memo takes up n*n* space. Plus we're still building up a call stack that'll occupy n*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*n*?

**Solution**

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

def fib(n):

# Edge cases:

if n < 0:

raise ValueError('Index was negative. No such thing as a '

'negative index in a series.')

elif n in [0, 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

prev\_prev = 0 # 0th fibonacci

prev = 1 # 1st fibonacci

for \_ in range(n - 1):

# 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 + prev\_prev

prev\_prev = prev

prev = current

return current



**Complexity**

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

**Bonus**

* If you're good with matrix multiplication you can bring the time cost down even further, to O(lg(n))*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)*O*(2*n*) time as opposed to n*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.