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:

Ruby

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

Gotchas

Our solution runs in N time.

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!

> The **call stack** is what a program uses to keep track of what function it's currently running and what to do with that function's return value.
>
> Whenever you call a function, a new **frame** gets pushed onto the call stack, which is popped off when the function returns. As functions call other functions, the stack gets taller. In recursive functions, the stack can get as tall as the number of times the function calls itself. This can cause a problem: the stack has a limited amount of space, and if it gets too big you can get a **stack overflow** error.

Breakdown

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

```ruby
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)`.

```ruby
def fib_recursive(n)
    if n == 0 || n == 1
        return n
    end
    return fib_recursive(n-1) + fib_recursive(n-2)
end
```

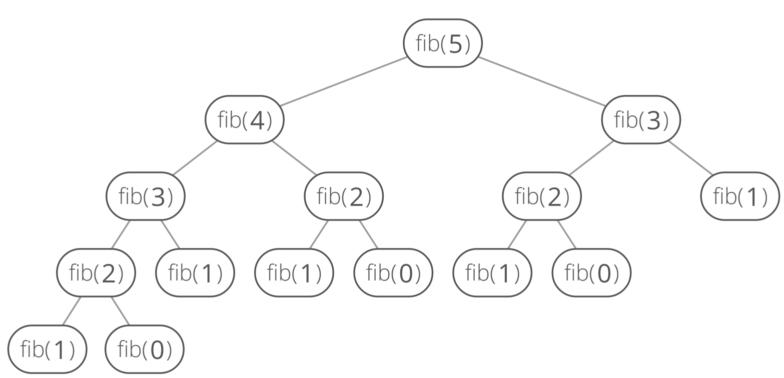
Okay, this'll work! What's our time complexity?

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!

Memoization ensures that a function doesn't run for the same inputs more than once by keeping a record of the results for given inputs (usually in a hash).

For example, a simple recursive function for computing the nth fibonacci number:

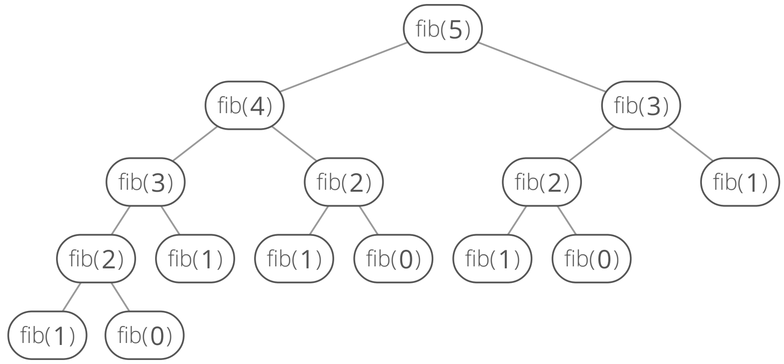
Ruby

```
def fib_recursive(n)

    if n < 0
        raise IndexError, 'Index was negative. No such thing as a negative index in
    end

    # base cases
    if n == 0 || n == 1
        return n
    end

    puts "computing fib_recursive(#{n})"
    return fib_recursive(n - 1) + fib_recursive(n - 2)
end
```

Will run on the same inputs multiple times:

```
001:0> fib_recursive(8)
computing fib_recursive(8)
computing fib_recursive(7)
computing fib_recursive(6)
computing fib_recursive(5)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(5)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(6)
computing fib_recursive(5)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
=> 21
```

We can imagine the recursive calls of this function as a tree, where the two children of a node are the two recursive calls it makes. We can see that the tree quickly branches out of control:



To avoid the duplicate work caused by the branching, we can wrap the function in a class that stores an instance variable, `@memo`, that maps inputs to outputs. Then we simply:

1. Check `@memo` to see if we can avoid computing the answer for any given input, and
2. Save the results of any calculations to `@memo`.

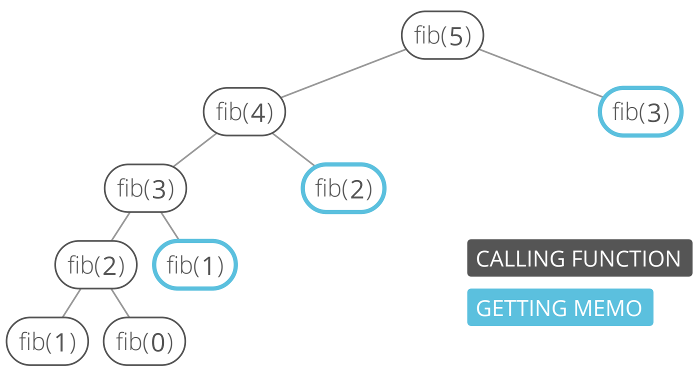
Ruby

```ruby
class Fibber

    def initialize
        @memo = {}
    end

    def fib(n)

        if n < 0
            raise Exception, "Index was negative. No such thing as a negative index

        # base cases
        elsif n == 0 || n == 1
            return n
        end

        # see if we've already calculated this
        if @memo.include? n
            puts "grabbing memo[#{n}]"
            return @memo[n]
        end

        print "computing fib(#{n})"
        result = self.fib(n - 1) + self.fib(n - 2)

        # memoize
        @memo[n] = result

        return result
    end
end
```

We save a bunch of calls by checking the memo:

```
001:0> Fibber().fib(8)
computing fib(8)
computing fib(7)
computing fib(6)
computing fib(5)
computing fib(4)
computing fib(3)
computing fib(2)
grabbing memo[2]
grabbing memo[3]
grabbing memo[4]
grabbing memo[5]
grabbing memo[6]
=> 21
```

Now in our recurrence tree, no node appears more than twice:



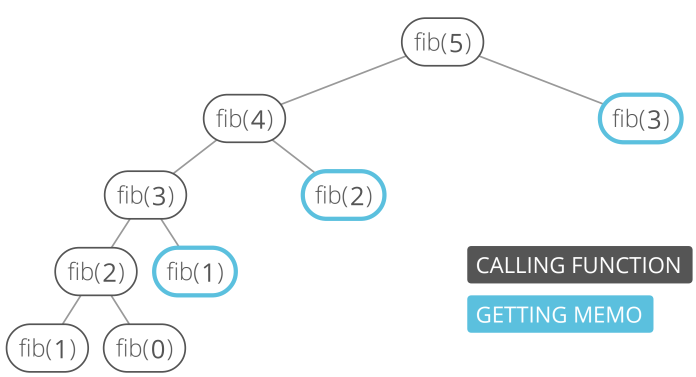
Memoization is a common strategy for **dynamic programming** problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the fibonacci problem, above). The other common strategy for dynamic programming problems is **going bottom-up (/concept/bottom-up)**, which is usually cleaner and often more efficient.

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

```ruby
class Fibber
    def initialize
        @memo = {}
    end

    def fib(n)

        # edge case: negative index
        if n < 0
            raise Exception, "Index was negative. No such thing as a negative index in a se

        # base case: 0 or 1
        elsif n == 0 || n == 1
            return n
        end

        # see if we've already calculated this
        if @memo.include? n
            return @memo[n]
        end

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

        # memoize
        @memo[n] = result

        return result
    end
end
```

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⤵

Going **bottom-up** is a way to avoid recursion, saving the **memory cost** that recursion incurs when it builds up the **call stack**.

Put simply, a bottom-up algorithm "starts from the beginning," while a recursive algorithm often "starts from the end and works backwards."

For example, if we wanted to multiply all the numbers in the range $1..n$, we could use this cute, **top-down**, recursive one-liner:

Ruby ▾

```
def product_1_to_n(n)
    # we assume n >= 1
    return n > 1 ? n * product_1_to_n(n-1) : 1
end
```

This approach has a problem: it builds up a **call stack** of size $O(n)$, which makes our total memory cost $O(n)$. This makes it vulnerable to a **stack overflow error**, where the call stack gets too big and runs out of space.

To avoid this, we can instead go **bottom-up**:

Ruby ▾

```
def product_1_to_n(n)
    # we assume n >= 1

    result = 1
    (1..n).each do |num|
        result *= num
    end

    return result
end
```

This approach uses $O(1)$ space ($O(n)$ time).

> *Some* compilers and interpreters will do what's called **tail call optimization** (TCO), where it can optimize *some* recursive functions to avoid building up a tall call stack. Python and Java decidedly do not use TCO. Some Ruby implementations do, but most don't. Some C implementations do, and the JavaScript spec recently *allowed* TCO. Scheme is one of the few languages that *guarantee* TCO in all implementations. In general, best not to assume your compiler/interpreter will do this work for you.

Going bottom-up is a common strategy for **dynamic programming** problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the fibonacci problem, above). The other common strategy for dynamic programming problems is **memoization (/concept/memoization)**.

approach, starting with the 0th fibonacci number and iteratively computing subsequent numbers until we get to n.

Ruby

```
def fib(n)

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

    elsif n == 0 || n == 1
        return n
    end

    # 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
    current = 0     # Declare and initialize current

    (n - 1).times do

        # 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
    end

    return current
end
```

Complexity

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.