Interview Cake

Write a method `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:

Java

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

If you wrote a recursive method, 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 method, there might be a hidden space cost in the call stack!

Overview

The **call stack** is what a program uses to keep track of method calls. The call stack is made up of **stack frames**—one for each method call.

For instance, say we called a method that rolled two dice and printed the sum.

Java ▾

```
def roll_die():
    return random.randint(1, 6)

def roll_two_and_sum():
    total = 0
    total += roll_die()
    total += roll_die()
    print total

roll_two_and_sum()
```

First, our program calls `rollTwoAndSum()`. It goes on the call stack:

```
rollTwoAndSum()
```

That function calls `rollDie()`, which gets pushed on to the top of the call stack:

```
rollDie()
```

```
rollTwoAndSum()
```

Inside of `rollDie()`, we call `random.randint()`. Here's what our call stack looks like then:

```
random.randint()
```

```
rollDie()
```

```
rollTwoAndSum()
```

When `random.randint()` finishes, we return back to `rollDie()` by removing ("popping") `random.randint()`'s stack frame.

```
rollDie()
```

```
rollTwoAndSum()
```

Same thing when `rollDie()` returns:

```
rollTwoAndSum()
```

We're not done yet! `rollTwoAndSum()` calls `rollDie()` *again*:

```
rollDie()
```

```
rollTwoAndSum()
```

Which calls `random.randint()` again:

```
random.randint()
```

```
rollDie()
```

```
rollTwoAndSum()
```

`random.randint()` returns, then `rollDie()` returns, putting us back in `rollTwoAndSum()`:

```
rollTwoAndSum()
```

Which calls `print()`:

```
print()
```

```
rollTwoAndSum()
```

What's stored in a stack frame?

What *actually* goes in a method's stack frame?

A stack frame usually stores:

- Local variables
- Arguments passed into the method
- Information about the caller's stack frame
- The *return address*—what the program should do after the function returns (i.e.: where it should "return to"). This is usually somewhere in the middle of the caller's code.

> Some of the specifics vary between processor architectures. For instance, AMD64 (64-bit x86) processors pass some arguments in registers and some on the call stack. And, ARM processors (common in phones) store the return address in a special register instead of putting it on the call stack.

The Space Cost of Stack Frames

Each method call creates its own stack frame, taking up space on the call stack. That's important because it can impact the *space complexity* of an algorithm. *Especially* when we use **recursion**.

For example, if we wanted to multiply all the numbers between 1 and n, we could use this recursive approach:

Java

```java
public static int product1ToN(int n) {
    // we assume n >= 1
    return (n > 1) ? (n * product1ToN(n-1)) : 1;
}
```

What would the call stack look like when `n = 10`?

First, `product1ToN()` gets called with `n = 10`:

```
product1ToN()    n = 10
```

This calls `product1ToN()` with `n = 9`.

```
product1ToN()    n = 9
product1ToN()    n = 10
```

Which calls `product1ToN()` with `n = 8`.

```
product1ToN()    n = 8
product1ToN()    n = 9
product1ToN()    n = 10
```

And so on until we get to `n = 1`.

```
product1ToN()    n = 1
product1ToN()    n = 2
```

```
product1ToN()    n = 3
```

```
product1ToN()    n = 4
```

```
product1ToN()    n = 5
```

```
product1ToN()    n = 6
```

```
product1ToN()    n = 7
```

```
product1ToN()    n = 8
```

```
product1ToN()    n = 9
```

```
product1ToN()    n = 10
```

Look at the size of all those stack frames! The entire call stack takes up $O(n)$ space. That's right—we have an $O(n)$ space cost even though our method itself doesn't create any data structures!

What if we'd used an iterative approach instead of a recursive one?

Java ▾

```java
public static int product1ToN(int n) {
    // we assume n >= 1

    int result = 1;
    for (int num = 1; num <= n; num++) {
        result *= num;
    }

    return result;
}
```

This version takes a constant amount of space. At the beginning of the loop, the call stack looks like this:

```
product1ToN()    n = 10, result = 1, num = 1
```

As we iterate through the loop, the local variables change, but we stay in the same stack frame because we don't call any other functions.

```
product1ToN()    n = 10, result = 2, num = 2
```

```
product1ToN()    n = 10, result = 6, num = 3
```

```
product1ToN()    n = 10, result = 24, num = 4
```

In general, even though the compiler or interpreter will take care of managing the call stack for you, it's important to consider the depth of the call stack when analyzing the space complexity of an algorithm.

Be especially careful with recursive functions! They can end up building huge call stacks.

> What happens if we run out of space? It's a **stack overflow**! In Java, you'll get a `StackOverflowError`.

> If the *very last* thing a method does is call another method, then its stack frame might not be needed any more. The method *could* free up its stack frame before doing its final call, saving space.
>
> This is called **tail call optimization** (TCO). If a recursive function is optimized with TCO, then it may not end up with a big call stack.
>
> In general, most languages *don't* provide TCO. Scheme is one of the few languages that guarantee tail call optimization. Some Ruby, C, and Javascript implementations *may* do it. Python and Java decidedly don't.

Breakdown

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

Java

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

Can you write up a recursive solution?

As with any recursive method, 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)`.

Java ▾

```java
public static int fib(int n) {
    if (n == 0 || n == 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

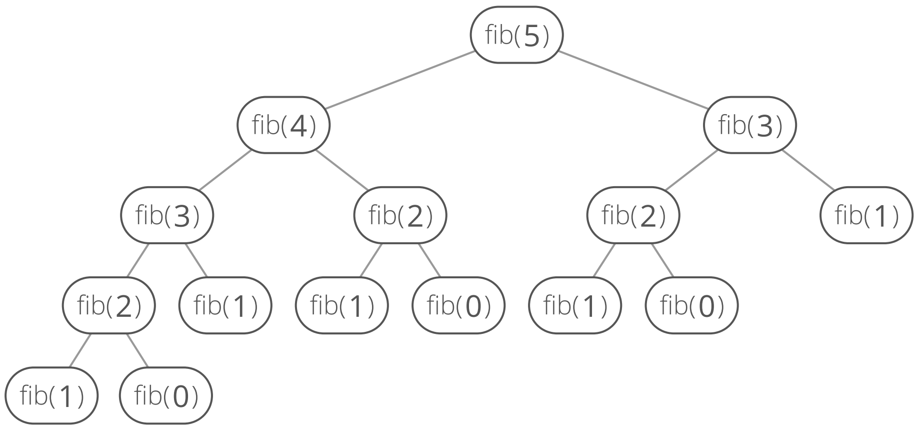
Okay, this'll work! What's our time complexity?

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:



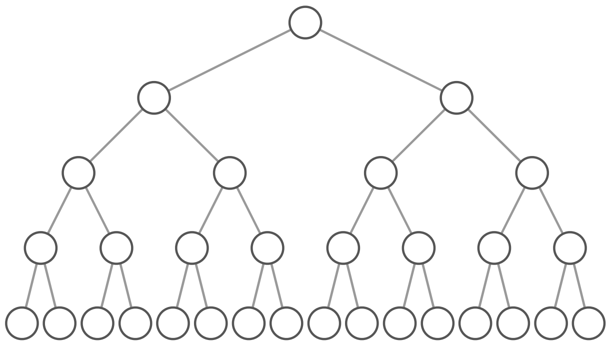
We can notice this is a binary tree↴

A **binary tree** is a **tree** where every node has two or fewer children. The children are usually called `left` and `right`.

Java

```java
public class BinaryTreeNode {

    public int value;
    public BinaryTreeNode left;
    public BinaryTreeNode right;

    public BinaryTreeNode(int value) {
        this.value = value;
    }
}
```

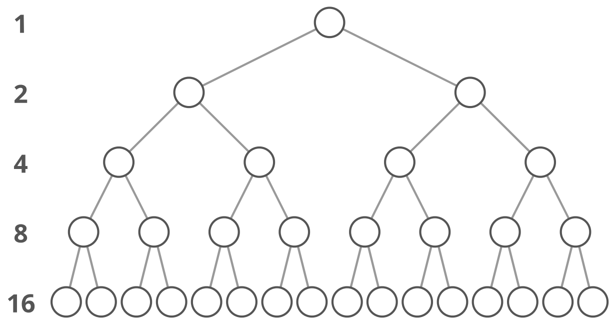
This lets us build a structure like this:



That particular example is special because every level of the tree is completely full. There are no "gaps." We call this kind of tree "**perfect**."

Binary trees have a few interesting properties when they're perfect:

Property 1: the number of total nodes on each "level" doubles as we move down the tree.



Property 2: the number of nodes on the last level is equal to the sum of the number of nodes on all other levels (plus 1). In other words, about *half* of our nodes are on the last level.

Let's call the number of nodes n, and the height of the tree h. h can also be thought of as the "number of levels."

If we had h, how could we calculate n?

Let's just add up the number of nodes on each level! How many nodes are on each level?

If we zero-index the levels, the number of nodes on the xth level is exactly 2^x.

1. Level 0: 2^0 nodes,
2. Level 1: 2^1 nodes,
3. Level 2: 2^2 nodes,
4. Level 3: 2^3 nodes,
5. *etc*

So our total number of nodes is:

$$n = 2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{h-1}$$

> Why only up to 2^{h-1}? Notice that we started counting our levels at 0. So if we have h levels in total, the last level is actually the "$h-1$"-th level. That means the number of nodes on the last level is 2^{h-1}
> .

But we can simplify. Property 2 tells us that the number of nodes on the last level is (1 more than) half of the total number of nodes, so we can just take the number of nodes on the last level, multiply it by 2, and subtract 1 to get the number of nodes overall. We know the number of nodes on the last level is 2^{h-1}, So:

$$n = 2^{h-1} * 2 - 1$$

$$n = 2^{h-1} * 2^1 - 1$$

$$n = 2^{h-1+1} - 1$$

$$n = 2^h - 1$$

So that's how we can go from h to n. What about the other direction?

We need to bring the h down from the exponent. That's what logs are for!

First, some quick review. $\log_{10}(100)$ simply means, **"What power must you raise 10 to in order to get 100?"**. Which is 2, because $10^2 = 100$.

We can use logs in algebra to bring variables down from exponents by exploiting the fact that we can simplify $\log_{10}(10^2)$. What power must we raise 10 to in order to get 10^2? That's easy—it's 2.

So in this case we can take the $\log_2$ of both sides:

$$n = 2^h - 1$$

$$n + 1 = 2^h$$

$$\log_2\left((n+1)\right) = \log_2\left(2^h\right)$$

$$\log_2\left(n+1\right) = h$$

So that's the relationship between height and total nodes in a perfect 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 method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs (usually in a hash map).

For example, a simple recursive method for computing the nth Fibonacci number:

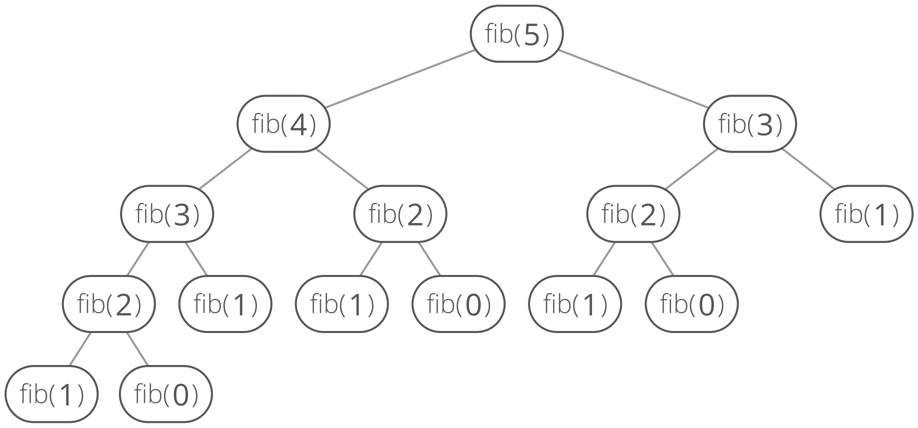
Java ▼

```
public static int fib(int n) {

    if (n < 0) {
        throw new IllegalArgumentException(
            "Index was negative. No such thing as a negative index in a series.");
    }

    // base cases
    if (n == 0 || n == 1) {
        return n;
    }

    System.out.printf("computing fib(%d)\n", n);
    return fib(n - 1) + fib(n - 2);
}
```

Will run on the same inputs multiple times:

```
// output for fib(5)
computing fib(5)
computing fib(4)
computing fib(3)
computing fib(2)
computing fib(2)
computing fib(3)
computing fib(2)
5
```

We can imagine the recursive calls of this method 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 method 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`.

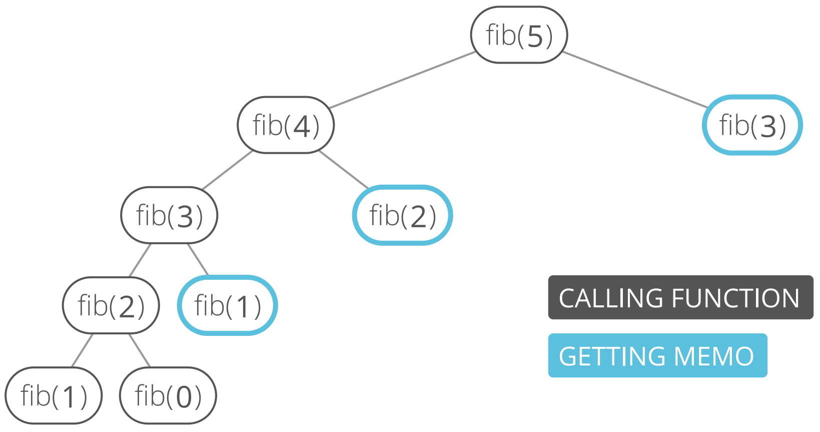
Java ▼

```java
import java.util.Map;
import java.util.HashMap;

class Fibber {

    private Map<Integer, Integer> memo = new HashMap<>();

    public int fib(int n) {

        if (n < 0) {
            throw new IllegalArgumentException(
                "Index was negative. No such thing as a negative index in a series."

        // base cases
        } else if (n == 0 || n == 1) {
            return n;
        }

        // see if we've already calculated this
        if (memo.containsKey(n)) {
            System.out.printf("grabbing memo[%d]\n", n);
            return memo.get(n);
        }

        System.out.printf("computing fib(%d)\n", n);
        int result = fib(n - 1) + fib(n - 2);

        // memoize
        memo.put(n, result);

        return result;
    }
}
```

We save a bunch of calls by checking the memo:

```
// output of new Fibber().fib(5)
computing fib(5)
computing fib(4)
computing fib(3)
computing fib(2)
grabbing memo[2]
grabbing memo[3]
5
```

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.

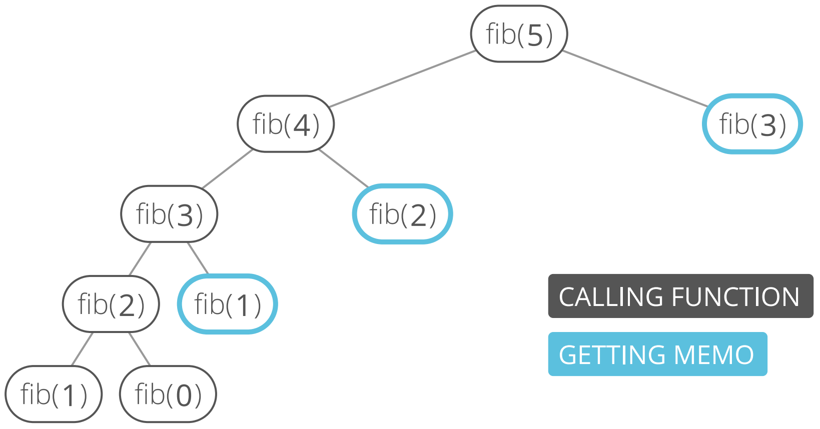
Let's wrap `fib()` in a class with an instance variable where we store the answer for any n that we compute:

Java

```
import java.util.Map;
import java.util.HashMap;

public class Fibber {

    private Map<Integer, Integer> memo = new HashMap<>();

    public int fib(int n) {

        // edge case: negative index
        if (n < 0) {
            throw new IllegalArgumentException("Index was negative. No such thing as a nega
        }

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

        // see if we've already calculated this
        if (memo.containsKey(n)) {
            return memo.get(n);
        }

        int result = fib(n - 1) + fib(n - 2);

        // memoize
        memo.put(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↓

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:

Java ▾

```java
public static int product1ToN(int n) {
    // we assume n >= 1
    return (n > 1) ? (n * product1ToN(n-1)) : 1;
}
```

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**:

Java ▾

```java
public static int product1ToN(int n) {
    // we assume n >= 1

    int result = 1;
    for (int num = 1; num <= n; num++) {
        result *= num;
    }

    return result;
}
```

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 methods 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 multiplying the numbers $1..n$, 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.

Java

```java
public static int fib(int n) {

    // edge cases:
    if (n < 0) {
        throw new IllegalArgumentException("Index was negative. No such thing as a negative
    } 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
    int prevPrev = 0;  // 0th fibonacci
    int prev = 1;      // 1st fibonacci
    int current = 0;   // Declare and initialize current

    for (int 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 method 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.

Ready for more?

Check out our full course ➡

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.