Recursion Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Given a recursive function, identify what is the base case and the recursive case.
2. Explain when a recursive solution is appropriate to solving a problem over an iterative solution.
3. Write a recursive function that takes in a number, n, argument and calculates the n-th number of the Fibonacci sequence.
4. Write a function that calculates a factorial recursively.
5. Write a function that calculates an exponent (positive and negative) recursively.
6. Write a function that sums all elements of an array recursively.
7. Write a function that flattens an arbitrarily nested array into one dimension.
8. Given a buggy recursive function that causes a RangeError: Maximum call stack and examples of correct behavior, debug the function.

Re-learning Functions With Recursion

Imagine it's your first day at a new job and your boss has asked you to unpack some fruit crates. Not too bad, right? Now imagine each crate has smaller crates inside. Still easy? Consider even smaller crates, nested within one another, each needing to be unpacked. How long before you throw your hands up in frustration and walk away?

Sometimes simple tasks get complicated and we need new tools to help us solve them. Working with digital data is a little like working with those crates: they may be simple to unpack, or they may be incredibly dense! Let's explore a new way of approaching problems: *recursion*.

We'll cover:

- what recursion is and how to identify it,
- implementing recursive functions,
- and breaking complex problems down into simpler tasks.

Re-what now?

So far, we've solved complex problems with *iteration*: the process of counting through each item in a collection. Like most things in programming, though, there's another way! Let's check out *recursion*: the process of calling a function within itself.

To wrap your mind around this new concept, think back to that example of crates within crates. If we have to gently unpack each crate but we don't know the contents, we'll have to go one-by-one through each crate, pulling items out individually. Let's think about a better way! What if we open each crate and look inside. If it's more crates, we dump them out. If it's fruit, we gently remove the fruit and set it aside.

What might this process look like in code? Here's some pseudocode to help us think through it:

```
function gentlyUnpackFruit(contents) {
  console.log("Your " + contents + " have been unpacked!");
}

function dump(crate) {
    if (crate.content_type === "crate") {
        dump(crate.contents);
    } else if (crate.content_type === "fruit") {
        gentlyUnpackFruit(crate.contents);
    }
}
```

Notice how we call the `dump`function from **within**the `dump`function. That's recursion in action! The `dump`function may *recurse*if we have crates nested within each other.

A note on language

You'll notice we've used the term *recurse*here, which you may not have heard before. Technically, the root word in "recursion" is "recur", but this is ambiguous. Consider these two examples:

```
console.log("Hello"); console.log("Hello");

// versus...

console.log(console.log("Hello"));
```

Both of these functions **recur**(as in, call the `console.log`function more than once), but only one of these functions is **recursive**(as in, calling`console.log`from within another `console.log`). To reduce confusion, researchers began using the term "recurse"to refer specifically to functions that are being called from within themselves. Creating a new word by removing a suffix in this way is known as *back-formation*.

We'll prefer "recurse" when discussing this topic, but you may see "recur" in other places! Carefully read the context and make sure you understand how these words might differ. Interviewers may use the terms interchangeably to trip you up, but we know you'll be ready for the challenge!

Two cases

Understanding recursion means understanding the two *cases*, or expected output for a particular input, in a recursive function. These are known as the *base case* and *recursive case*.

- The *base case*describes the situation where data passed into our function is processed without any additional recursion. When the base case is executed, the function runs once and ends. Since this results in the function stopping, we may also refer to this as the *terminating case*.
- The *recursive case*, as the name suggests, is the situation where the function recurses. This represents the data state that causes a function to call itself. Without a recursive case, a function's just another function!

In our fruit crate example, the base case is "when the crate contains fruit" and the recursive case is "when the crate contains other crates". When we encounter fruit, we remove the fruit and the action is complete. However, when we encounter more crates, we go back to the start and repeat the whole process again.

Identifying these cases for a process won't always be that simple, but it's critical to figure out each one before writing any code. Without a recursive case, we don't need recursion at all, and we should consider an alternative approach. Without a base case, we might be creating an *infinite loop* - yikes! We need to know when to stop the process before we start it.

A recursive example

Let's look at a more practical problem you might encounter in the wild. We're going to use the "Movie Theater Problem" to demonstrate how recursion can help us with a real world issue.

Imagine you're meeting a friend in a movie theater. The lights have gone down, it's totally dark, and your friend just sent you a message asking which row you're seated in. Without being able to see the rows or your ticket, how might you figure out the row number?

Let's assume a few things:

- The theater is mostly occupied, so you can rely on people being in front of & behind you.
- You don't want to knock over anyone's drinks & snacks, so you must remain in your seat.
- Your phone is almost dead, so you can't use the flashlight or screen to illuminate the seats - no cheating!

What if we tap the person in front of us on the shoulder? If there's someone in front of us, we know that we're at least one row back from the front of the theater. If someone is in front of **them**, we're at least **two**rows back! This pattern continues until we reach the screen.

If each person performs this action and they all report back, we can count how many rows back we are! We've missed a key part in this analysis, though -

when do we stop tapping each other? If someone reaches forward and there's no one else in front of them, we can assume we've reached the front of the theater. That person becomes "Row #1", and our test stops.

In this example, our base case is "No one in front of me = Row #1" and our recursive case is "Someone in front of me = Row #(1 + person in front of me's row #)". Now the we know both cases, we can build a recursive function out of them! Here's what this might look like in JavaScript:

```
determineRow = function(moviegoer) {
  if (moviegoer.personInFront) {
      return 1 + determineRow(moviegoer.personInFront);
  } else {
      return 1;
  }
}
```

Now it doesn't matter if our movie theater has 5 rows or 5,000 rows - we have a tool to figure out where we are at any time. We've also gone through an important exercise in understanding our space to get here! By working to our sides instead of in front of us, we could use the same process to figure out exactly which seat we're in on the row.

What we've learned

Whew! If your head is spinning, don't worry - it's totally natural. Recursion can get a lot mre complex than what we've covered here, but it comes down to working smarter, not harder. We'll dig a little deeper into advanced recursion and how to know when to build a recursive function in our next lesson.

Check out Computerphile's What on Earth is Recursion?to learn more about recursion and the stack.

After completing the reading and video, you should be able to:

- define *recursion*,
- explain its use,
- and identify a simple base & recursive case in a problem.

When To Hold & When To Fold(Fold(Fold())): Recursion vs. Iteration

We know what *recursion*is, but to truly understand what's happening, we need to go deeper! Let's investigate the process of recursion and build a better understanding of the risks involved.

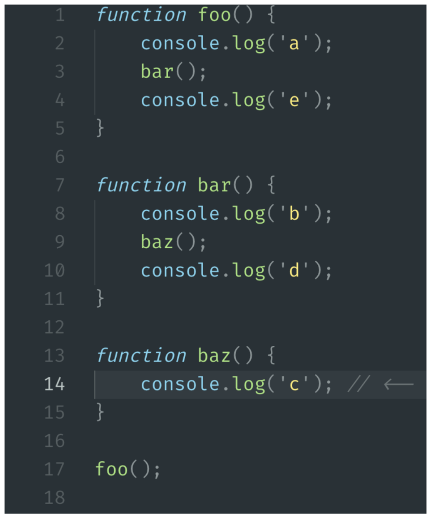
We'll cover:

- recursion and the call stack,
- differentiating recursive and iterative functions,
- and know when to use each tool for maximum effect.

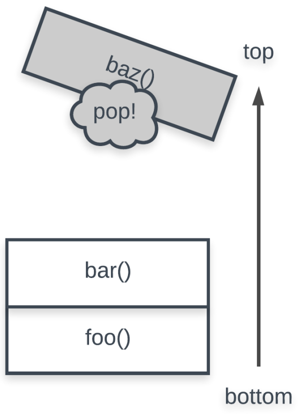
A deeper dive into recursion

Learning about recursion requires that we review the *call stack*. Remember that each function call in JavaScript *pushes* a new *stack frame*onto the top of the call stack, and the last pushed frame gets *popped*off as it gets executed. We sometimes refer to this as a *Last In, First Out*, or *LIFO*, stack.

Here's an example to jog your memory:



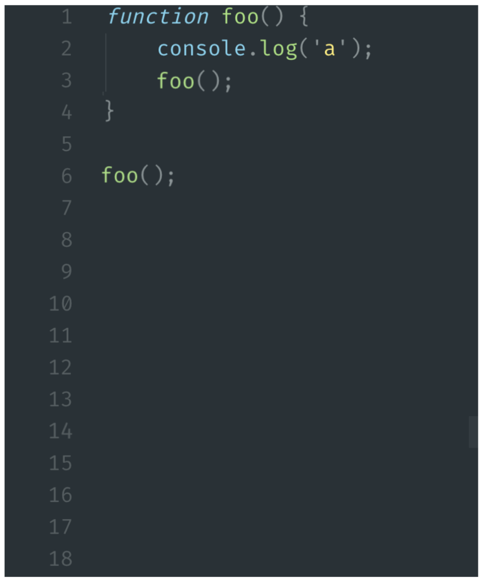
```
function foo() {
    console.log('a');
    bar();
    console.log('e');
}

function bar() {
    console.log('b');
    baz();
    console.log('d');
}

function baz() {
    console.log('c'); // ⟵
}

foo();

```



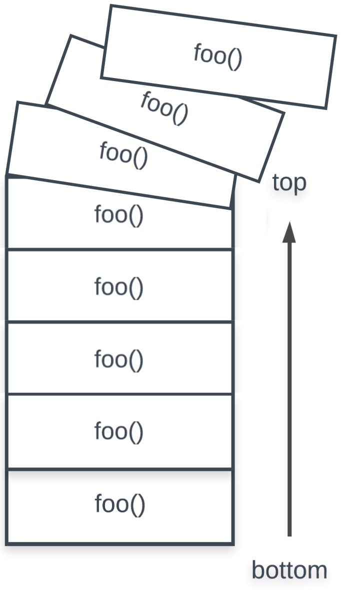
Recursive functions risk placing extra load on the call stack. Each recursive function call depends on the call before it, meaning we can't start executing a recursive function's stack frames until we reach the *base case*. So what happens if we never do? Look out!

London Stack is falling down!

The JavaScript call stack has a size limit which varies between different browsers and even different systems! Once the stack reaches this limit, we get what's called a *stack overflow*. The program halts, the stack gets wiped out entirely, and we're left with no results wondering what we did wrong.



```
function foo() {
    console.log('a');
    foo();
}

foo();
```



Let's look at an example of an obvious stack overflow issue:

```
function pythagoreanCup() {
    pythagoreanCup();
};

pythagoreanCup();
```

Output:

```
Uncaught RangeError: Maximum call stack size exceeded
    at pythagoreanCup (<anonymous>)
    at pythagoreanCup (<anonymous>)
    at pythagoreanCup (<anonymous>)
    at pythagoreanCup (<anonymous>)
    at pythagoreanCup (<anonymous>)
    ...
```

The function `pythagoreanCup`is clearly recursive, since it calls itself, but we're missing a base case to work towards! This means the function will recurse until the call stack overflows, resulting in a `RangeError`. Whoops! Notice that in our *stack trace*(the output below the error name & message), we can see that `pythagoreanCup`is the only function currently in the call stack.

Fixing the overflow issue in this case is straightforward: determine a base case and implement it in your function. Here's a fixed version of the example above with some extra comments:

```
let justEnoughWine = false;

function pythagoreanCup() {
    // Base case:
    // - Is `justEnoughWine` true? Return & exit.
    if (justEnoughWine === true) {
        console.log("That's plenty, thanks!");
        return true;
    }

    // Recursive case:
    // - justEnoughWine must not have been true,
    //   so let's set it and check again.
    justEnoughWine = true;
    pythagoreanCup();
};

pythagoreanCup();
```

Output:

```
"That's plenty, thanks!"
```

The stack size limit varies due to different implementations: some JavaScript environments might have a fixed limit, while others will rely on available *memory on your computer. Regardless of your environment, if you're receiving`RangeError`s, you should refactor your function! It's bad practice to build software that only runs in one particular browser or using a specific runtime.*

Step by step

Notice that our change to `pythagoreanCup`did two things:

- Provided a base case that lets us end the recursion,
- and added an action that moves us **towards**the base case.

Without changing the value of `justEnoughWine`, we would never enter the base case and our stack would still be at risk of growing out of control.

We refer to the action that gets us closer to the base case as the *recursive step*. Don't forget to build this step into your function! Your base case doesn't mean anything if you're not moving towards it with each recurrence.

Types of recursion

Our examples of recursion so far have involved a single function calling itself. We refer to this situation as *direct recursion*: functions directly calling themselves. There's a trickier type of recursion to debug, though. Take a look at the following example:

```
function didYouDoTheThing() {
    ofCourseIDidTheThing();
}

function ofCourseIDidTheThing() {
    didYouDoTheThing();
}

didYouDoTheThing();
```

Uh oh! Neither of these functions appears to be recursive by itself, but calling either of them will put us into a recursive loop with no base case in sight. We refer to recursive loops across multiple functions as *indirect recursion*. There's nothing wrong with using this technique, but be careful! Because the call stack will have multiple function names in it, debugging problems with indirectly recursive functions can be a headache.

When to iterate, when to recur

Alright, slow down, programmers. Before you rewrite all your recent projects to use recursive functions, let's investigate why you might choose iteration instead.

Remember that *iteration*is when we call a function for each member of a collection, instead of letting the function call itself. So far, the code you've written using `for`loops and iterator functions like `.forEach`has been*iterative*. Iterative code tends to be less resource-intensive than recursive code, and it requires less planning to get working. It's also usually easier to read & understand - an important thing to consider when writing software!

Iterative approaches tend to break down when our data becomes *very complex*or *very large*. Consider the task of sorting paper folders by the number of files in each. If you only had a few folders, this wouldn't be very daunting, and you could take an iterative approach: open each folder individually, count the files, and place the folders in the correct order.

However, if you had thousands or even millions of folders, iteration would take days instead of minutes. You'd want time to implement a system for getting through those folders efficiently, and that system would likely involve a procedure for ordering in batches. This is exactly what recursive functions do best: repetitive processes on subsets of given data.

Consider recursion when your inputs are unpredictable, large, or highly complex. Otherwise, iteration will almost always be the best approach. We'll look at lots of examples to help you get a feel for this before someone asks you to identify the best approach to a given problem in an interview

Compare these approaches

Before we move on, let's compare an iterative & recursive approach with each other. We'll create a `countdown`function that takes a number and counts down from that number to zero.

Here's an iterative approach:

```
countdown(startingNumber) {
    for(let i = startingNumber; i > 0; i--) {
        console.log(i);
    }

    console.log("Time's up!");
}
```

For comparison, a recursive approach:

```
  countdown(startingNumber) {

      if (startingNumber === 0) {
          console.log("Time's up!");
          return;
      }

      console.log(startingNumber);
      countdown(startingNumber - 1);
  }
```

Can you identify the base case, recursive case, and recursive step?

What we've learned

We're now equipped to solve problems of varying complexity with two different approaches! Don't forget to be considerate of which approach might be best for new challenges you encounter.

After completing this lesson, you should be able to:

- define "stack overflow" and debug functions causing one,
- differentiate between iterative and recursive functions,
- and identify good candidates for each type of approach.