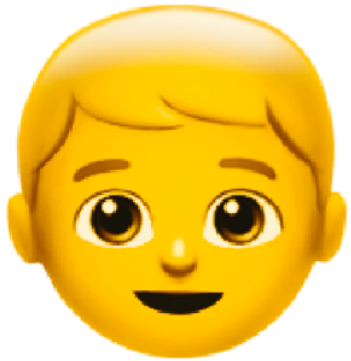
Recursion
Recursion
Recursion
Recursion

ITERATION & RECURSION

FIRST, STORY TIME

Once Upon A Time...



&

Martin

The Dragon

Excuse me Mr. Dragon,
Are any of these numbers odd?

(3142 5798 6550 5914)

ANGRY DRAGON!

Sorry boy, I'll only tell you if the **first number** in that list is odd.

BUT I NEED TO KNOW IF ANY OF THE NUMBERS IN THE LIST ARE ODD!

SORRY BOY, I'LL ONLY TELL YOU IF THE **FIRST** NUMBER IN A LIST IS ODD

Hmmm.....

Ok fine, what about the first number in this list?

(3142 5798 6550 5914)

NOT ODD!

And what about the first number in this other list?

(5798 6550 5914)

NOT ODD!

Ok fine, what about the first number in this list?

(6550 5914)

NOT ODD!

Ok fine, what about the first number in this list?

(5914)

NOT ODD!

Ok fine, what about the first number in this list?

( )

That's an empty list you moron! There isn't a number in there!

AHA! So all the numbers are even in that list!

I NEVER SAID THAT!

No Odds (3142 5798 6550 5914)

No Odds (5798 6550 5914)

No Odds (6550 5914)

No Odds (5914)

No Odds ()

NO ODDS!

Congratulations, you discovered recursion.

Wait, you knew about this the whole time?

What is recursion?

A **process** (a function in our case) that **calls itself**

Why do I need to know this?

It's EVERYWHERE!

- JSON.parse / JSON.stringify
- document.getElementById and DOM traversal algorithms
- Object traversal
- Very common with more complex algorithms
- It's sometimes a cleaner alternative to iteration

How recursive functions work

Invoke the **same** function with a different input until you reach your base case!

Base Case

The condition when the recursion ends.

This is the most important concept to understand

Two essential parts of a recursive function!

- Base Case
- Different Input

Our first recursive function

```
function countDown(num){
    if(num <= 0) {
        console.log("All done!");
        return;
    }
    console.log(num);
    num--;
    countDown(num);
}
```

Our second recursive function

```
function sumRange(num){
   if(num === 1) return 1;
   return num + sumRange(num-1);
}
```

Can you spot the base case?

Do you notice the different input?

What would happen if we didn't return?

The ALL important `return` keyword

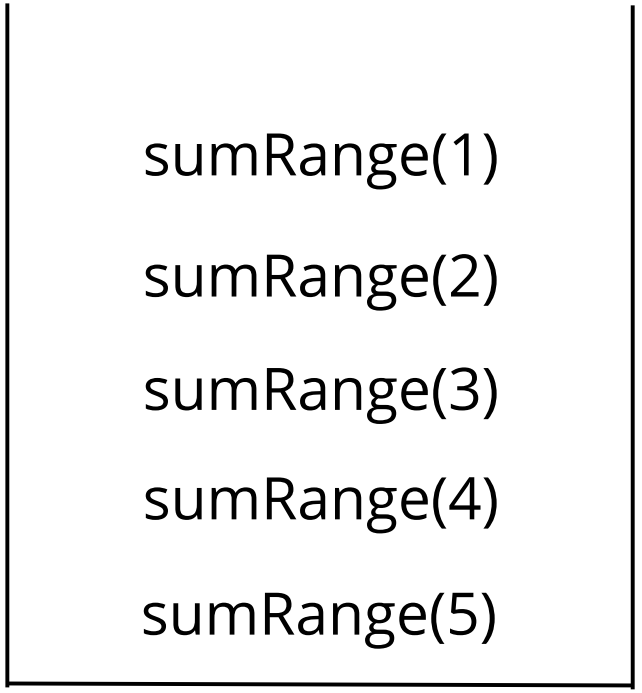
```
function sumRange(num){
   if(num === 1) return 1;
   return num + sumRange(num-1);
}
```

Let's break this down step by step!

sumRange with the call stack

```
function sumRange(num){
    if(num === 1) return 1;
    return num + sumRange(num-1);
}

sumRange(5)
```



Another example

```
function factorial(num){
    if(num === 1) return 1;
    return num * factorial(num-1);
}
```

Let's visualize the call stack!

Visualizing a recursive function with the Chrome Dev Tools

Where things go wrong

- No base case
- Forgetting to return or returning the wrong thing!
- Stack overflow!

```
function factorial(num){
    if(num === 1) return 1;
    return num * factorial(num);
}
```

```
function factorial(num){
    if(num === 1) console.log(1) ;
    return num * factorial(num-1);
}
```

HELPER METHOD RECURSION

```
function outer(input){

    var outerScopedVariable = []

    function helper(helperInput){
        // modify the outerScopedVariable
        helper(helperInput--)
    }

    helper(input)

    return outerScopedVariable;

}
```

ANOTHER EXAMPLE

Let's try to collect all of the odd values in an array!

```
function collectOddValues(arr){

    let result = []

    function helper(helperInput){
        if(helperInput.length === 0) {
            return;
        }

        if(helperInput[0] % 2 !== 0){
            result.push(helperInput[0])
        }

        helper(helperInput.slice(1))
    }

    helper(arr)

    return result;

}
```

PURE RECURSION

```
function collectOddValues(arr){
    let newArr = [];

    if(arr.length === 0) {
        return newArr;
    }

    if(arr[0] % 2 !== 0){
        newArr.push(arr[0]);
    }

    newArr = newArr.concat(collectOddValues(arr.slice(1)));
    return newArr;
}
```

Pure Recursion Tips

- For arrays, use methods like **slice, the spread operator,** and **concat** that make copies of arrays so you do not mutate them
- Remember that strings are immutable so you will need to use methods like **slice, substr, or substring** to make copies of strings
- To make copies of objects use **Object.assign, or the spread operator**

What about big O?

- Measuring **time complexity** is relatively simple. You can measure the time complexity of a recursive function as then number of recursive calls you need to make relative to the input
- Measuring **space complexity** is a bit more challenging. You can measure the space complexity of a recursive function as the **maximum** number of functions on the call stack at a given time, since the call stack requires memory.

RECURSION

PROBLEM SET

POWER

Write a function which accepts a base an an exponent. It should return the result of raising the base to that exponent.

```
power(2,4) //16
power(3,2) //9
power(3,3) //27
```

HINT!

$$2^3 = 2 * 2^2$$

$$2^2 = 2 * 2^1$$

$$2^1 = 2 * 2^0$$

$$2^0 = 1$$

productOfArray

Write a function called productOfArray which takes in an array of numbers and returns the product of them all.

```
productOfArray([1,2,3])     // 6
productOfArray([1,2,3,10]) // 60
```

productOfArray

Write a function called productOfArray which takes in an array of numbers and returns the product of them all.

```
productOfArray([1,2,3])     // 6
productOfArray([1,2,3,10]) // 60
```

Tail Call Optimization

- ES2015 allows for *tail call optimization*, where you can make some function calls without growing the call stack.
- By using the **return** keyword in a specific fashion we can extract output from a function without keeping it on the call stack.
- Unfortunately this has not been implemented across multiple browsers so it is not reasonable to implement in production code.

Recap

- A recursive function is a function that invokes itself
- Your recursive functions should **always** have a base case and be invoked with different input each time
- When using recursion, it's often essential to return values from one function to another to extract data from each function call
- Helper method recursion is an alternative that allows us to use an external scope in our recursive functions
- Pure recursion eliminates the need for helper method recursion, but can be trickier to understand at first