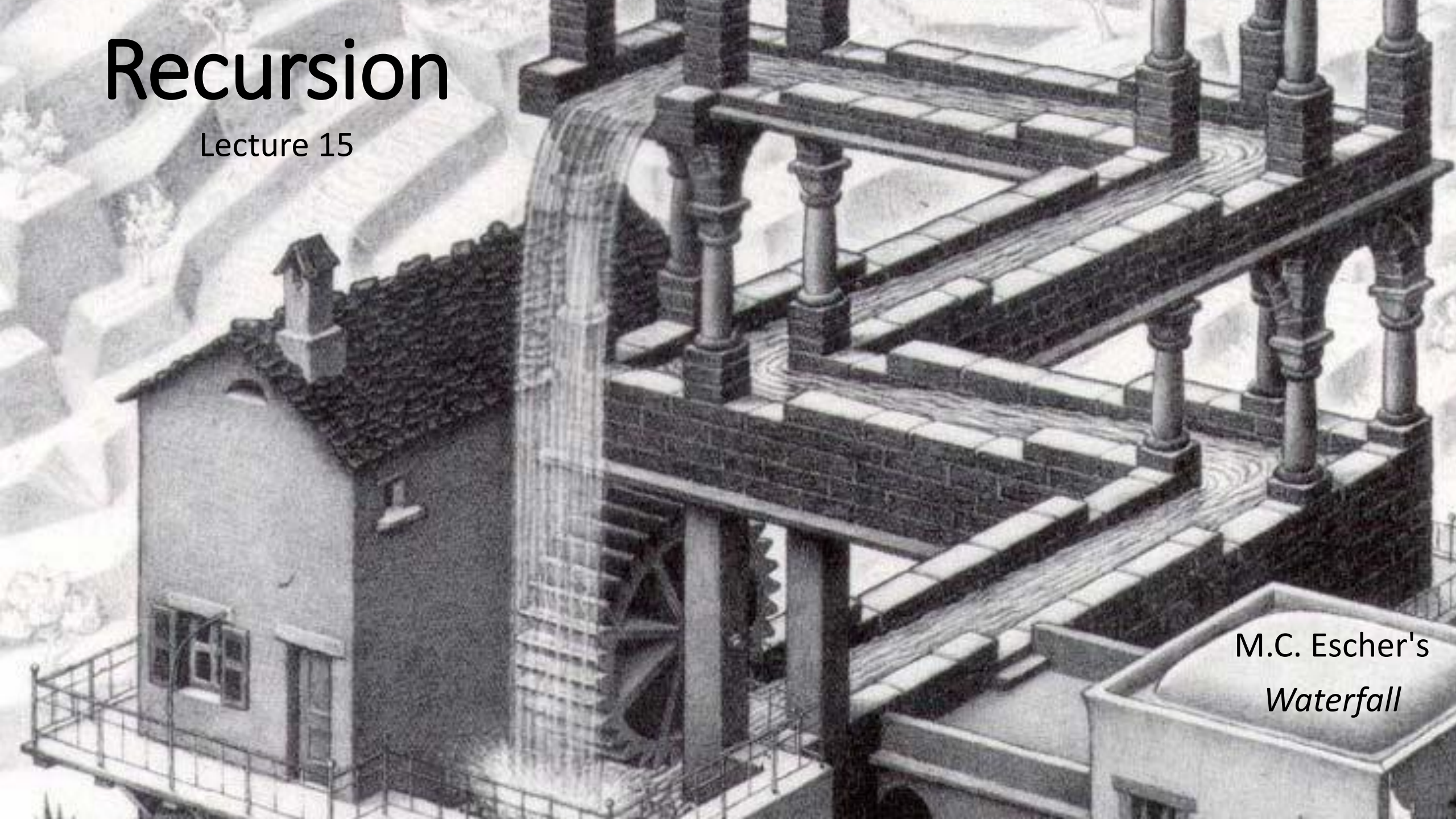
Recursion

Lecture 15



M.C. Escher's
Waterfall

Announcements

- Small Group Review Sessions Today & Tomorrow
- PS03 Due by Tuesday 10/17 at 11:59pm
- Late night with Roy

Warm-up Question #0: When the **main** function runs, what is printed?

```
function main(): void {
    let spooky: string = "sp" + o(-2) + "ky";
    print(spooky);
}

function o(n: number): string {
    if (n === 0) {
        return "~";
    } else {
        return "oO" + o(n + 1) + "Oo";
    }
}
```

Warm-up Question #1:

When the **main** function runs, what is printed?

```
function main(): void {
    f1();
}

function f0(): void {
    print(0);
    f1();
    print(0);
}

function f1(): void {
    print(1);
    f2();
    print(1);
}

function f2(): void {
    print(2);
    f3();
    print(2);
}

function f3(): void {
    print(3);
    print("Base Case!");
    print(3);
}
```

Warm-up Question #2:

When the **main** function runs, what is printed?

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}

function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
    }
    print("Leave f(" + n + ")");
}
```

Hands-on: A Recursive Function

- Open lec15 / 00-recursion-demo-app.ts
- Under the print statement that prints "Recur!", still inside of the else block, try calling:

```
f(n + 1);
```

- Now try changing the call to **f** in **main** to use **2** as an argument instead of 3.
- Check-in on PollEv.com/comp110 once you've tried f(2)
  - Done? Try other numbers, as well...

Code

Memory
Stack

Output

When we enter a function via a function call, a space for that function's local variables, called a **frame**, is reserved in a part of memory called the **stack**.

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Code

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack

Output

Enter main()

The print statement causes Enter main() to print out.

Code

We've reached a **function call**! First we drop our bookmark (dotted triangle).

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack

Output

Enter main()

Code

Then we assign argument(s), in this case the number 3, to corresponding parameters, in this case n. How does this work?

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack

Output

Enter main()

Code

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

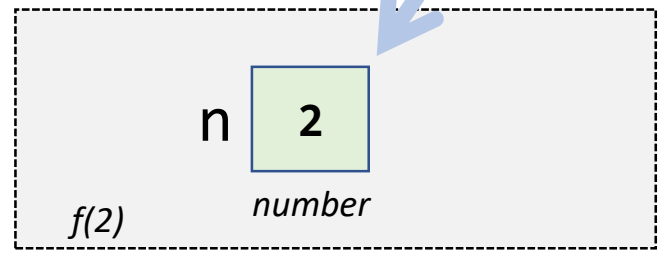
```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack

A new frame on the stack is allocated for the function fall f(2).

This is where the local variables (including parameters) are stored.



Output

Enter main()

Code

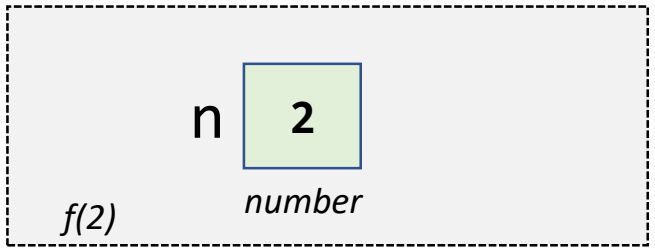
Now that parameters are assigned, we enter the function. We reach the print statement.

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack



Output

Enter main()

Enter f(2)

Code

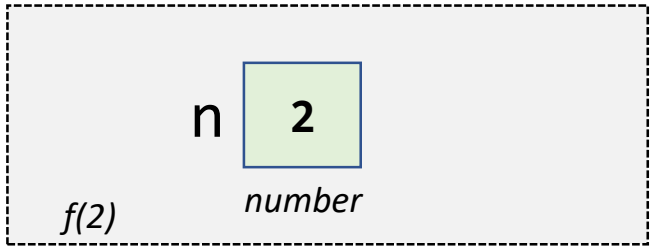
n (2) was *not >= 3,* so we enter the *else* branch and hit another *print* statement

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack



Output

Enter main()

Enter f(2)

Recur!

Code

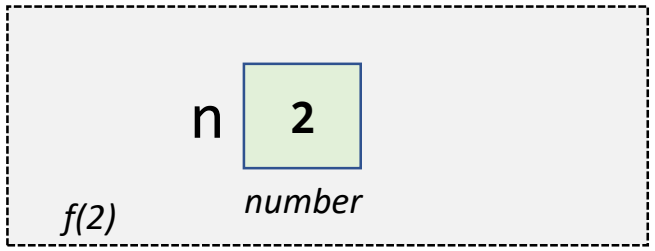
We've reached a **function call**! First we drop our bookmark (dotted triangle).

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack



Output

Enter main()

Enter f(2)

Recur!

Code

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

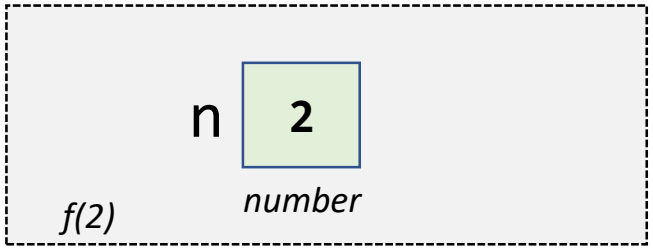
```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print
}
```

```
function
    print
    f(2)
    print("Leave main()");
}
```

Notice, it's ***as if*** we have a complete copy of the function f! All the same rules apply... so what do we do?

Memory

Stack



Output

Enter main()

Enter f(2)

Recur!

Code

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

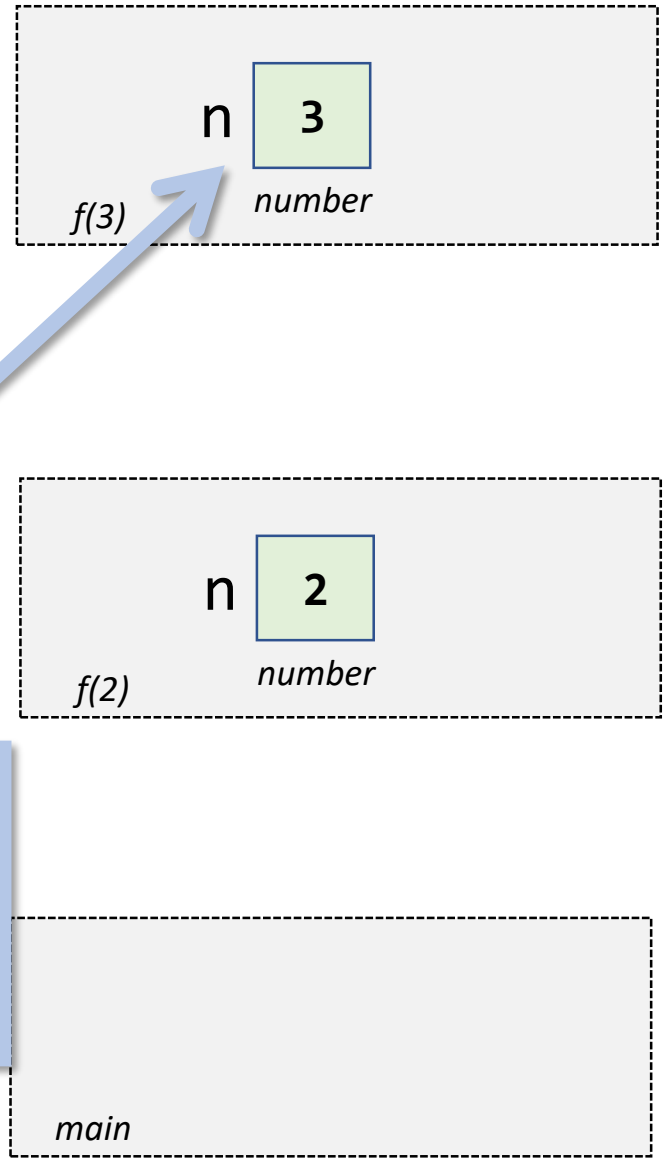
```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    prin
}
```

```
functio
    prin
    f(2)
    print("Leave main()");
}
```

First, we assign our parameter(s)!

Notice a new **frame** on the **stack** has been setup for **f(3)**.

Memory
Stack



Output

Enter main()

Enter f(2)

Recur!

Code

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

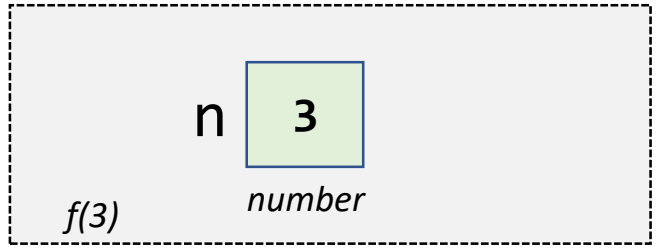
Then, we enter the function.

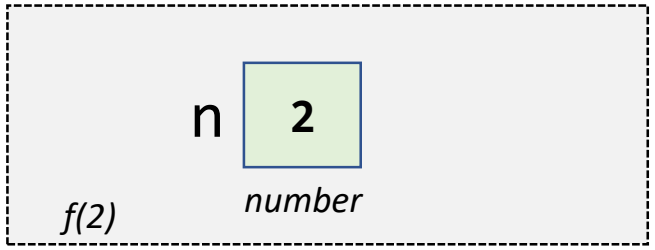
A **print** statement is encountered.

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack





Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Code

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

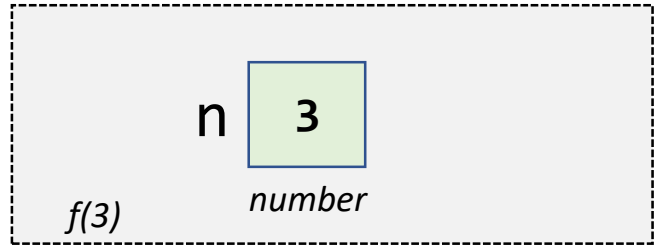
```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

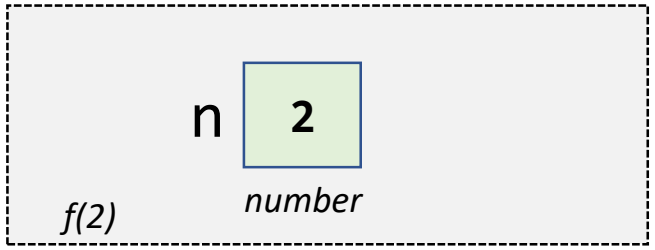
```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

The condition n >= 3 is true when n is 3, so we enter the then-block and print "Base case!"

Memory

Stack





Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Code

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

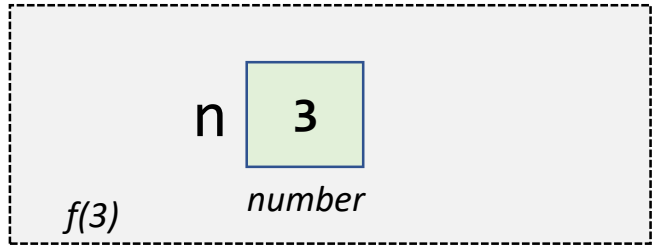
```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if
    }
    }
    print( Leave f(  + n +  ) );
}
```

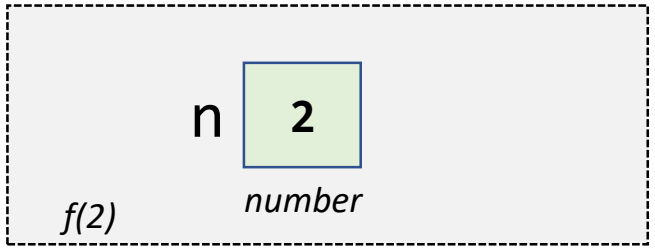
Skipping the else-block, we reach another print statement.

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack





Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

Code

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

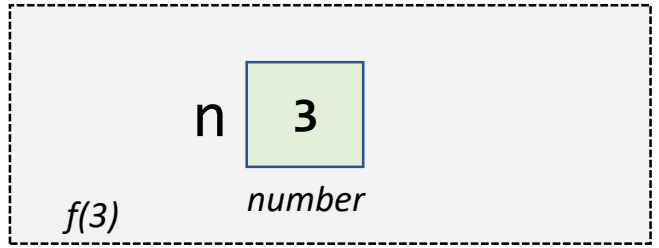
```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if
    }
    }
    print("Leave f(" + n + ")");
}
```

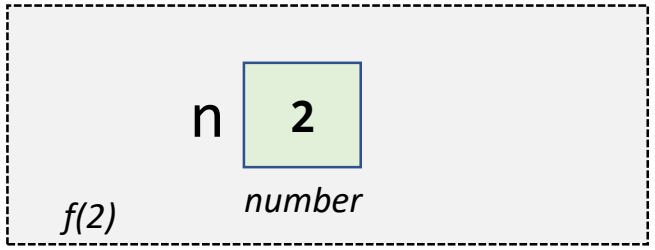
We've reached the end of this function call. What do we do?

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack





Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

Code

We return back to our bookmark!

What happens to the stack frame of f(3), though?

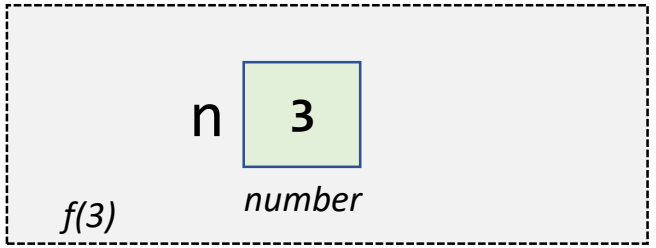
It gets "popped" off of the stack and is deleted.

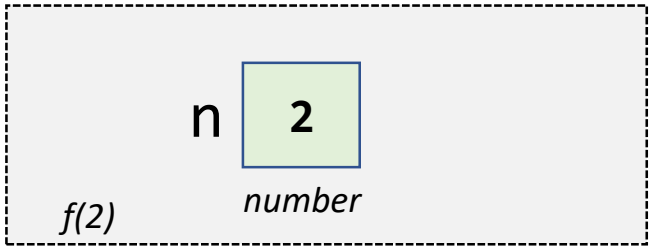
```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory

Stack





Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

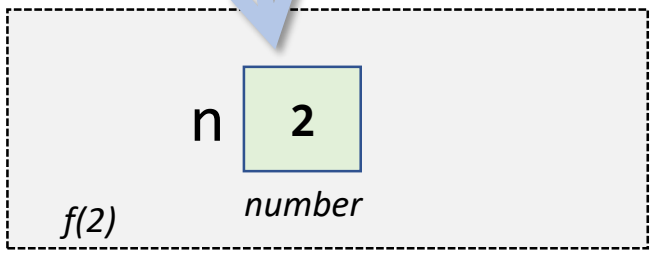
Code

So, the frame we're working within is once again f(2)'s frame.

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack



Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

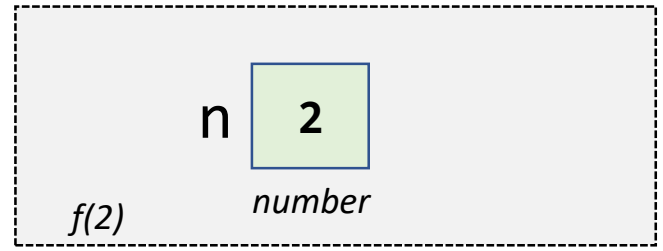
Code

A print statement is reached, in this frame **n** is 2.

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack



Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

Leave f(2)

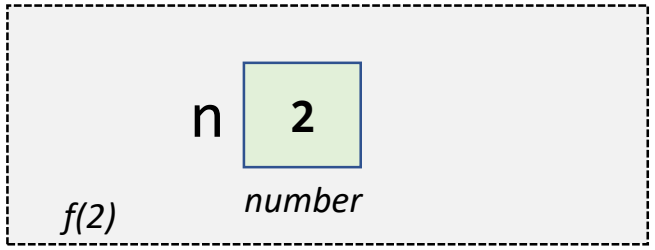
Code

We've hit the end of a function call! What do we do?

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n >= 3) {
        print("Base case!");
    } else {
        print("Recur!");
        f(n + 1);
    }
    print("Leave f(" + n + ")");
}
```

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack



Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

Leave f(2)

Code

We return back to our previous bookmark in **main** and pop f(2)'s frame off of the stack.

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack

Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

Leave f(2)

Code

We print "Leave main()"

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Memory
Stack

Output

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

Leave f(2)

Leave main()

Code

Memory
Stack

Output

Finally, we've reached the end of the main function. What do we do?

```
function main(): void {
    print("Enter main()");
    f(2);
    print("Leave main()");
}
```

Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

Leave f(3)

Leave f(2)

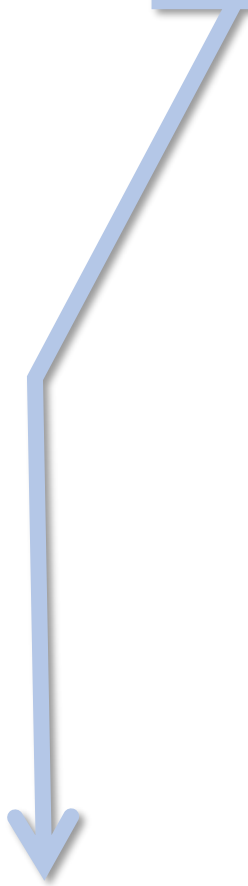
Leave main()

Code

Memory
Stack

Output

We return back to wherever **main** was called from...



Enter main()

Enter f(2)

Recur!

Enter f(3)

Base case!

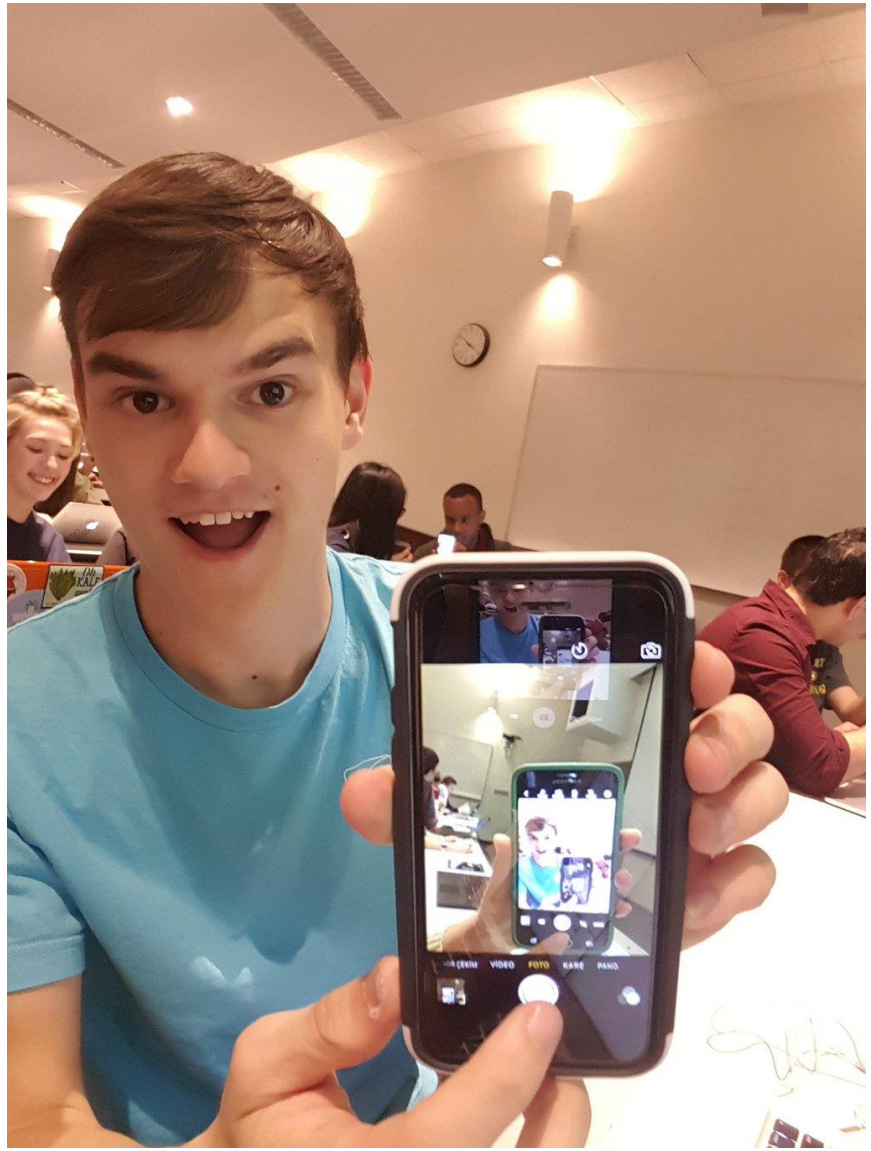
Leave f(3)

Leave f(2)

Leave main()

Recursive Selfie

- Pair up with a neighbor!
- Turn on your front facing cameras
- Hold your cameras between you and point the front of your phones at each other…
- Now try to get in the picture!
- Post to Twitter with #comp110
- Check-in on PollEv.com/comp110



Recursion

- A recursive method is **a method that calls itself**
- It's a *beautiful*, *powerful* concept!
- Anything you can do with a loop, you can do with recursion.

Recursion and the Call Stack

- Recursion works because every time a method call is encountered a new frame is added to the call stack
- A frame's local variable values are independent of other frames'
- We are able to leave a bookmark in the middle of a method, add a new frame, and jump into the same method!
- It's a lot like inception…

What happens when the following code runs?

```
function main(): void {
    print("Enter main()");
    f(10);
    print("Leave main()");
}

function f(n: number): void {
    print("Enter f(" + n + ")");
    f(n - 1);
    print("Leave f(" + n + ")");
}
```

- Open 01-stack-overflow-app
- Run it.
- Scroll wayyy down...

What's a "**Stack Overflow Error**"?

- Remember, every function call gets its own "frame" on the "stack" for storing a reference to this, its parameters, and any local variables
- Here we're continuing to infinitely call the same function using recursion
- Each frame on the stack requires memory, though!
- When the stack grows too tall, we run out of memory and crash.

How do we *prevent* Stack Overflows?

- The same way we prevent *infinite loops*.
- We need a *test condition* at which point we *stop recurring.*
- This is called a **base case**.
- Additionally, we need something to *change* as we recur. Often, this is the *argument* we pass to the recursive call.

Hands-on #2: Adding a Base Case

1. Open **02-base-case-app.ts**

2. When you run this file, you will see another Stack Overflow error. Let's fix it!

3. In the function **f**, add an if statement that tests if n is less than or equal to 0
    1. If so, print out **"Base case!"**
    2. If not, call the **f** function again *recursively* with an argument of n – 1

4. Try running again! Try changing the argument in **main** to test.

5. Check-in when you're complete!

Base Case

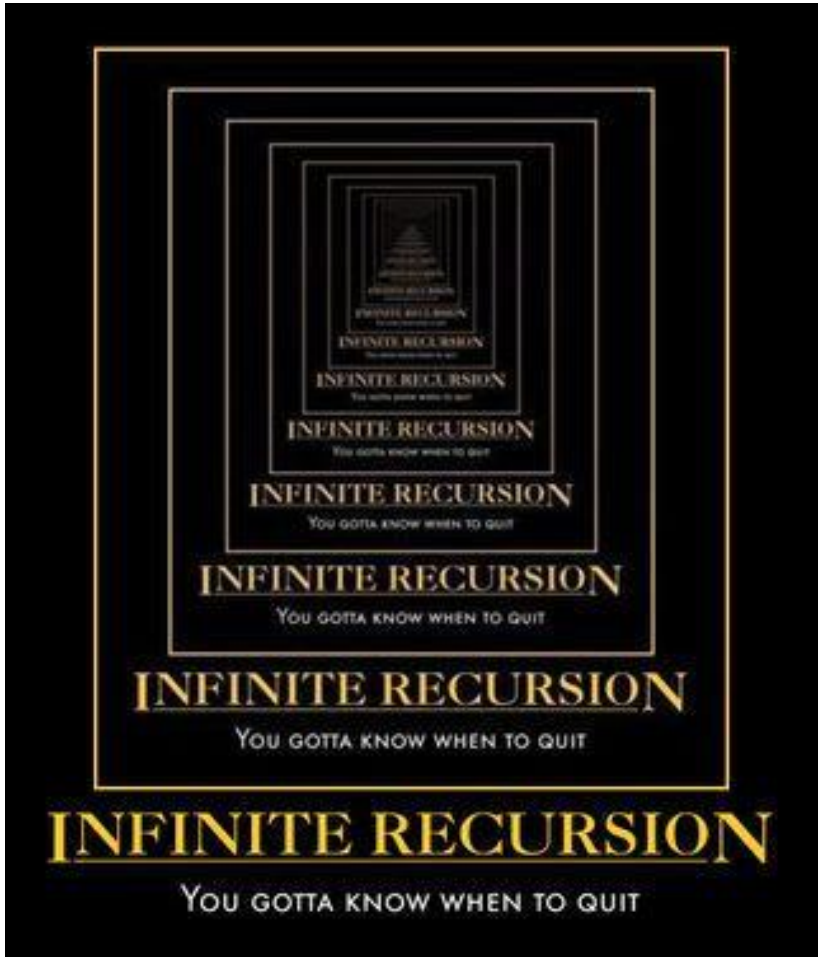
Recursive Case

```
function f(n: number): void {
    print("Enter f(" + n + ")");
    if (n <= 0) {
        print("Base Case!");
    } else {
        f(n - 1);
    }
    print("Leave f(" + n + ")");
}
```

1. Notice when a function call reaches the **base case**, it ***does not*** **recur**

2. The **recursive case** is recurring and changing the argument it recurs with
   - Notice this argument is bringing the value n *towards* to the base case. What would happen if it were $f(n+1)$?

Every Recursive Function *Needs* a Base Case



- The base case is the end of a series of recursive function calls
- Until the computer reaches the base case, each recursive call is adding another frame to the stack
- Once the recursive call is complete, we are popping frames off of the stack as we return to the earlier functions

Applications of Recursion

- Recursive Art
  - Fractals
- Algorithms
  - Some great sorting algorithms are naturally recursive: merge sort, quick sort
  - Binary search
- Programming Language Compilers

Working with a sequence of values recursively

- Many recursive algorithms work on a sequence of values
- These algorithms tend to process the "first" value and then recur on the "rest" of the values
- We can simulate this with helper functions to treat arrays as a list
  - The **first** function will return the 0^{th}-index element of an array
  - The **rest** function will return a new array without the first element

Array's `slice` method

- Every array of type T[] has a method named slice
- Its signature is:

```
slice(start: number, end: number): T[]
```

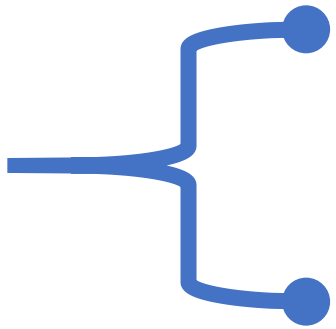
- Usage: `a.slice(1, a.length);`
- Returns a new array containing the elements of the original array starting at index **`start`** and ending at index **`end - 1`**.

03-first-rest-example-app.ts

```
import "introcs";

import { first, rest } from "./helpers";

    function main(): void {
    let a: number[] = [1, 2, 3];
    print(a);

    // TODO: Print the first element of a
    print(first(a));

    // TODO: Print the rest of a using the rest
    function
    print(rest(a));
    print(rest(rest(a)));
    print(rest(rest(rest(a))));
}

main();
```

Defining **sum** recursively

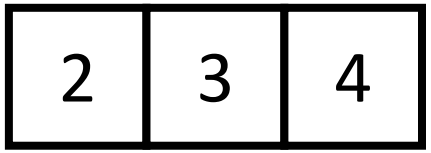
sum(a)



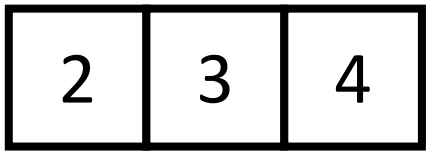
0, when a.length === 0

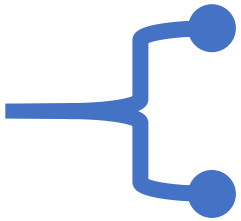
a_{first} + sum(a_{rest}), when a.length > 0

We can think of recursive functions similar to piecewise functions in math.

sum()

Tracing **sum(a)**

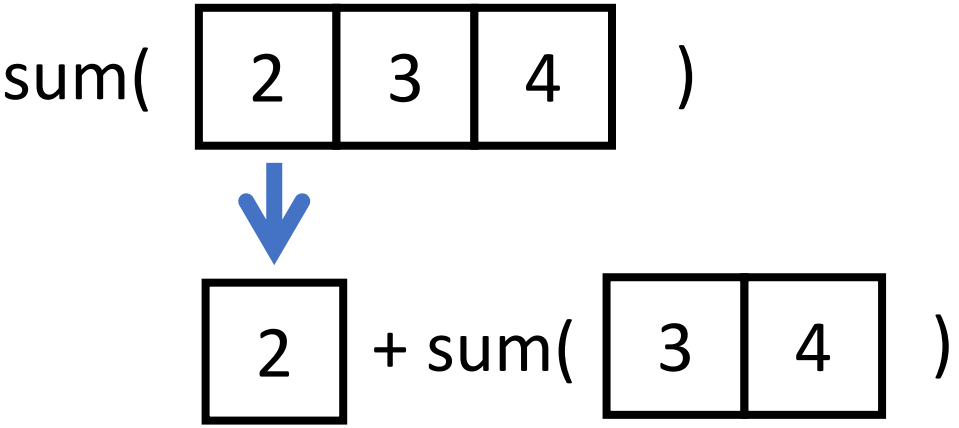
a = 

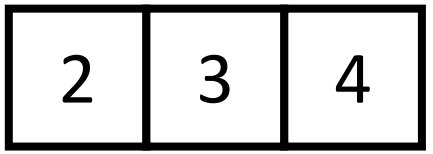
sum(a) 

0, when a.length === 0

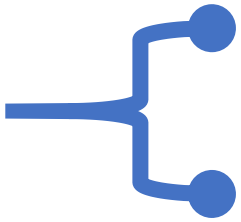
a_{first} + sum(a_{rest}), when a.length > 0

Tracing **sum(a)**



a = 

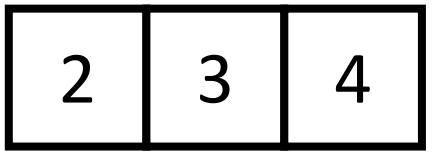
sum(a)

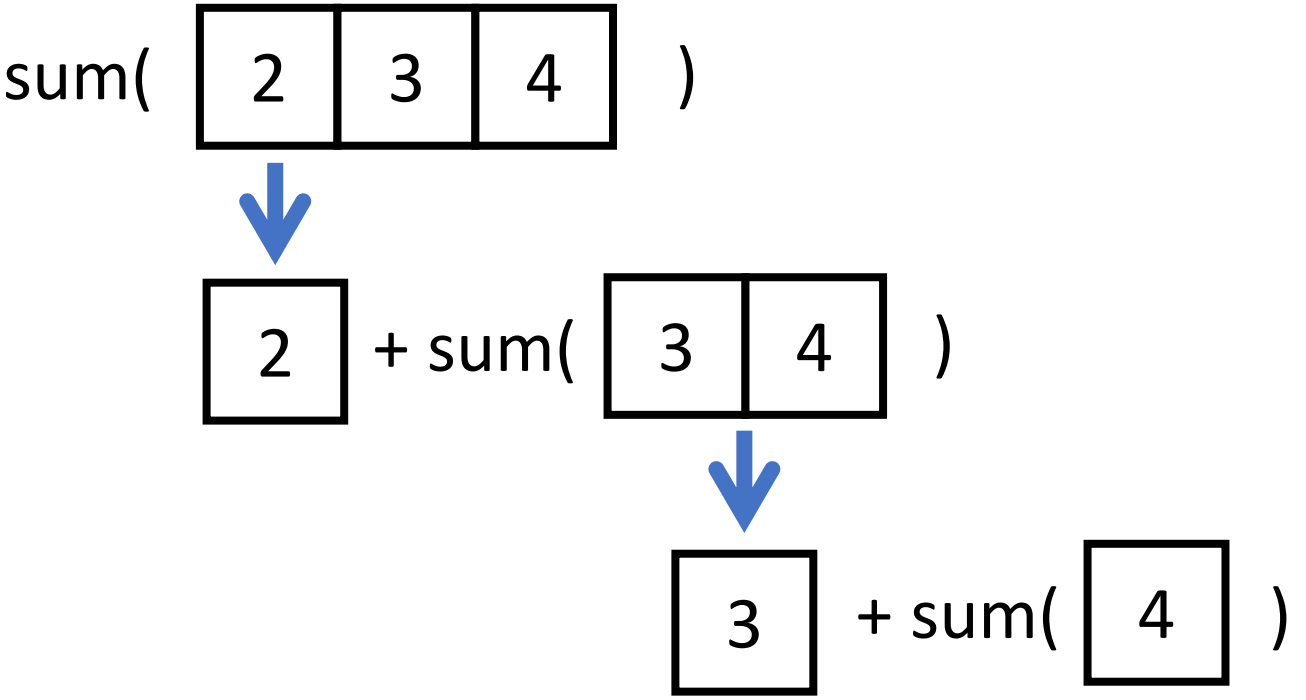


0, when a.length === 0

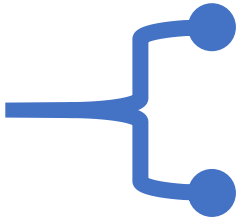
a_{first} + sum(a_{rest}), when a.length > 0

Tracing **sum(a)**

a = 



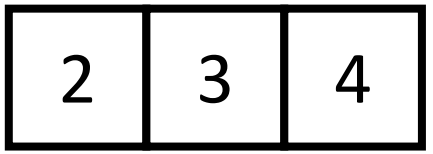
sum(a)

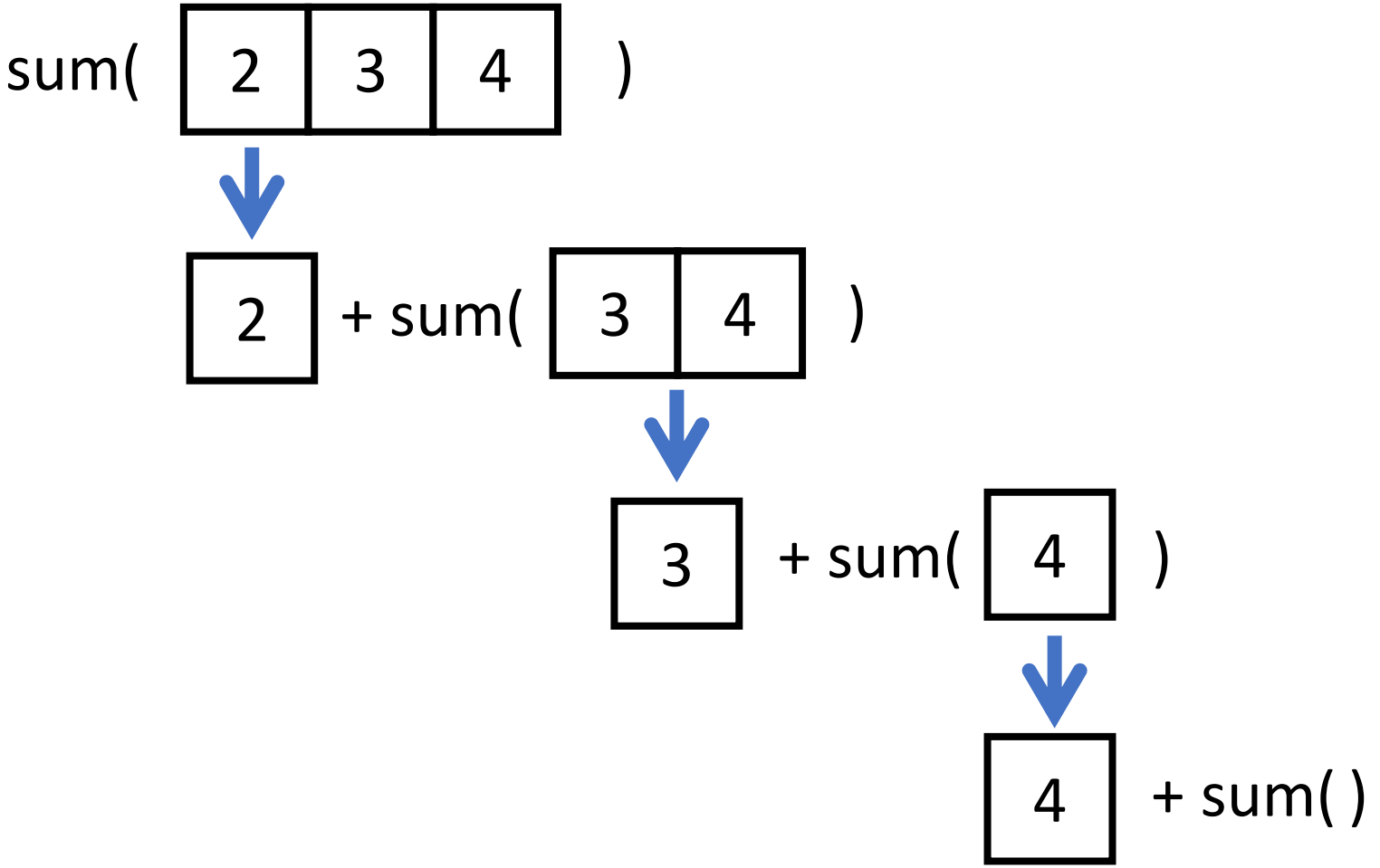


0, when a.length === 0

a_{first} + sum(a_{rest}), when a.length > 0

Tracing **sum(a)**

a = 

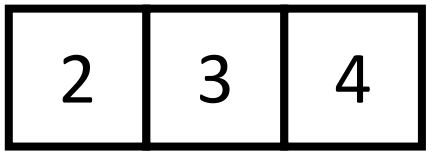


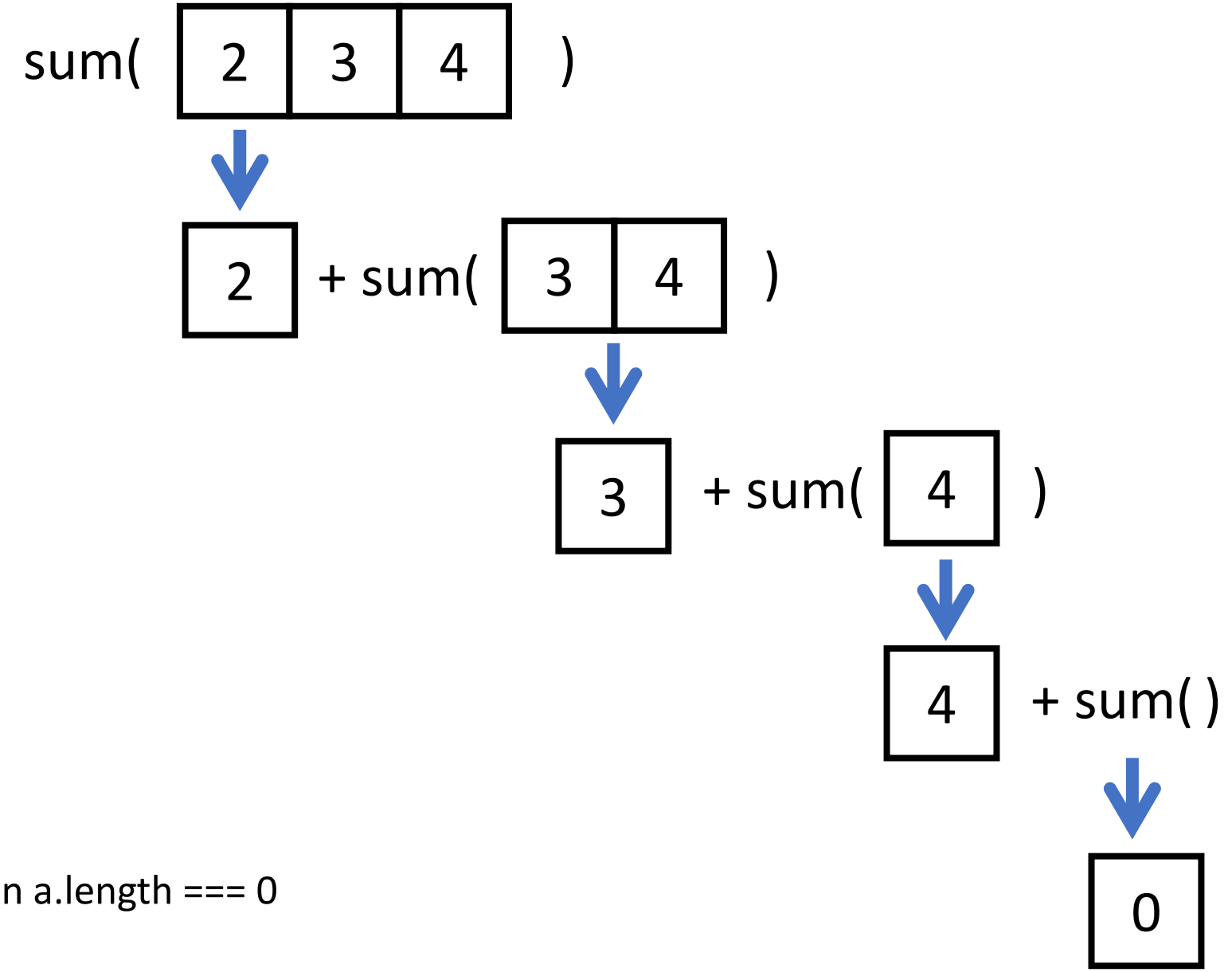
sum(a)

0, when a.length === 0

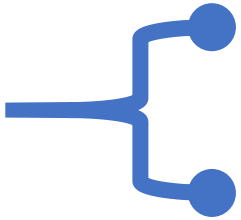
a_{first} + sum(a_{rest}), when a.length > 0

Tracing **sum(a)**

a = 



sum(a)

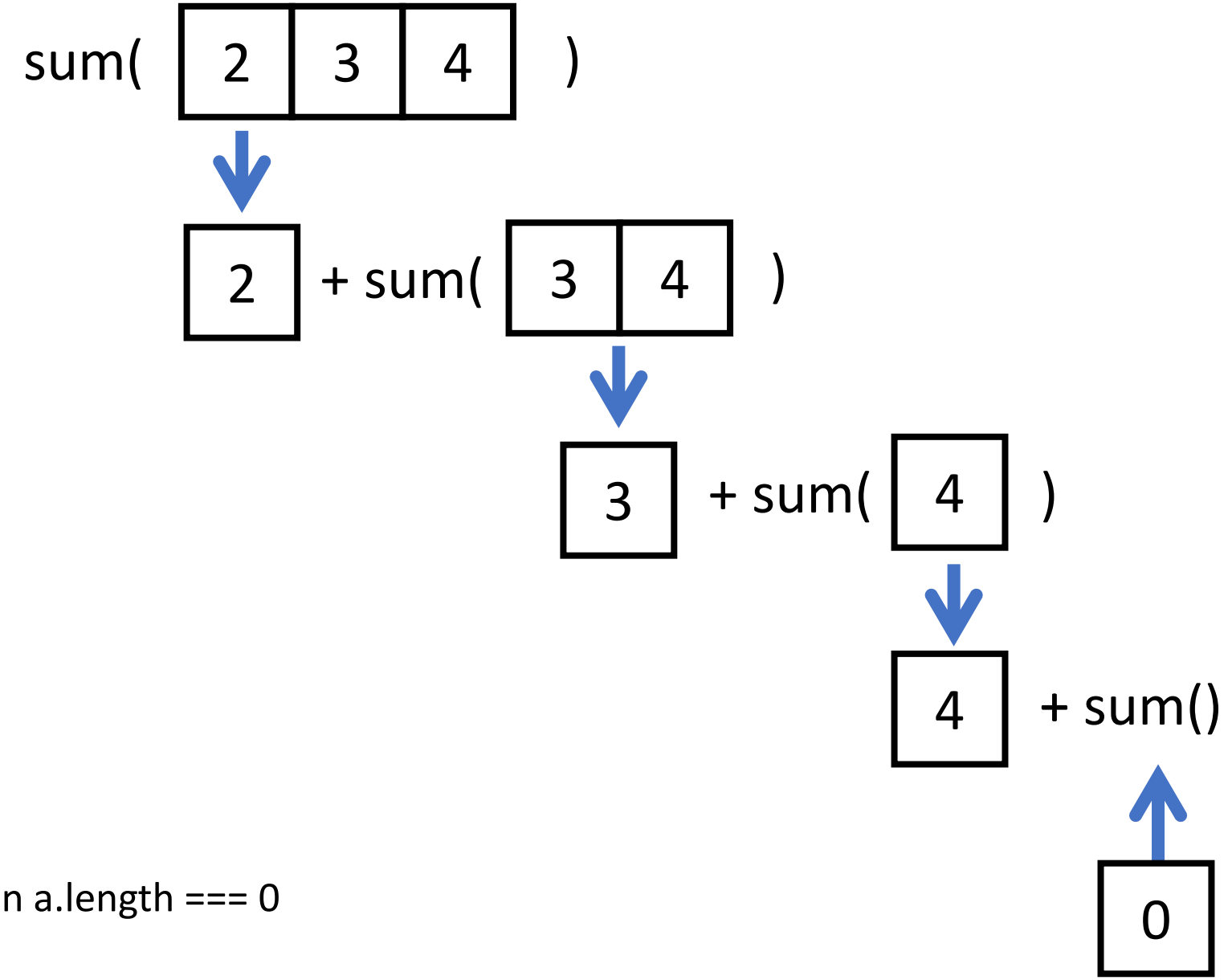


0, when a.length === 0

a_{first} + sum(a_{rest}), when a.length > 0

Tracing **sum(a)**

a = [2 | 3 | 4]



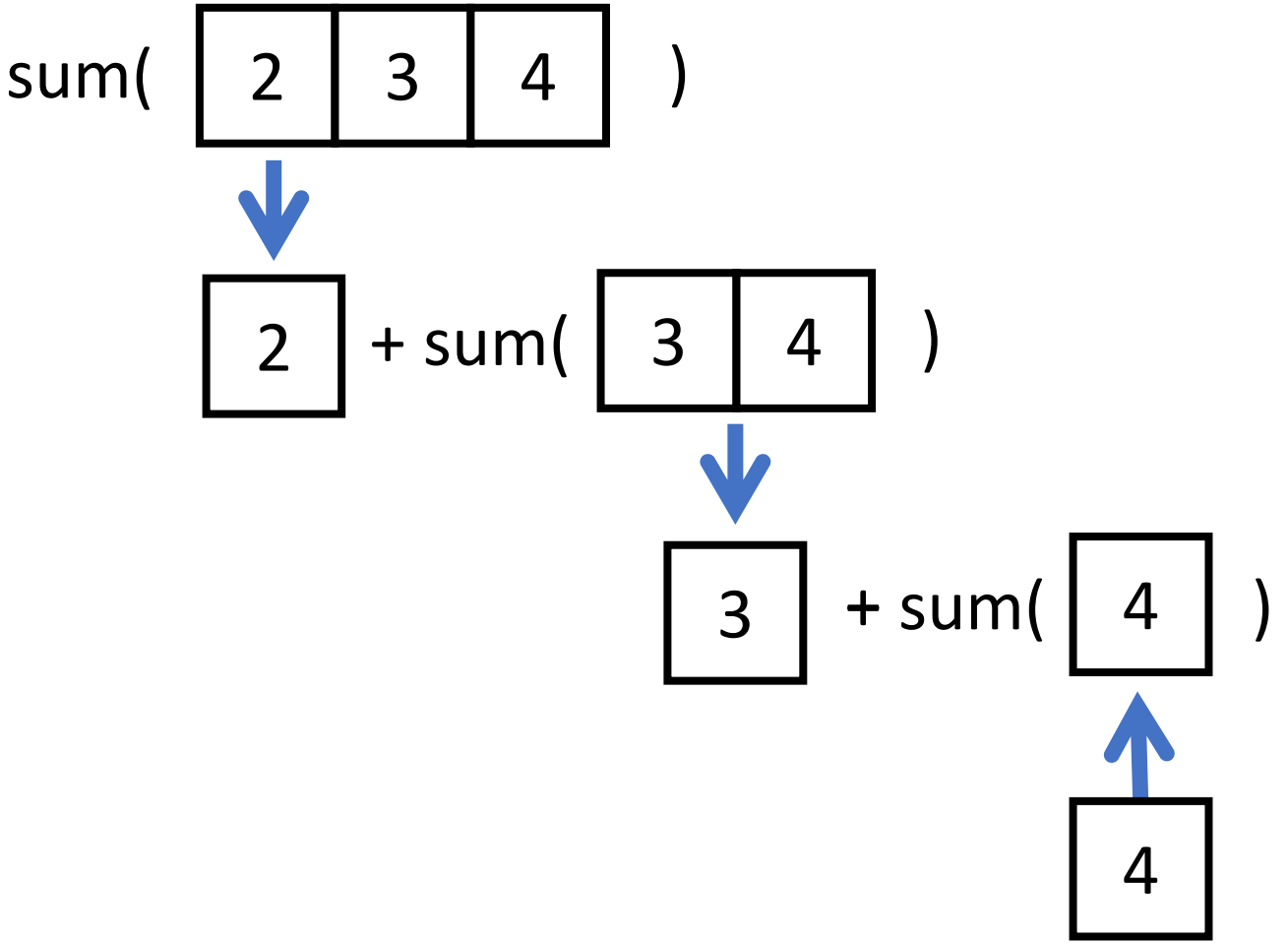
sum(a)

0, when a.length === 0

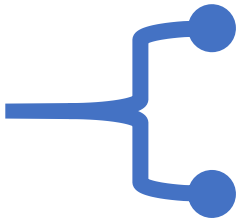
a_{first} + sum(a_{rest}), when a.length > 0

Tracing **sum(a)**

a = [2 | 3 | 4]



sum(a)

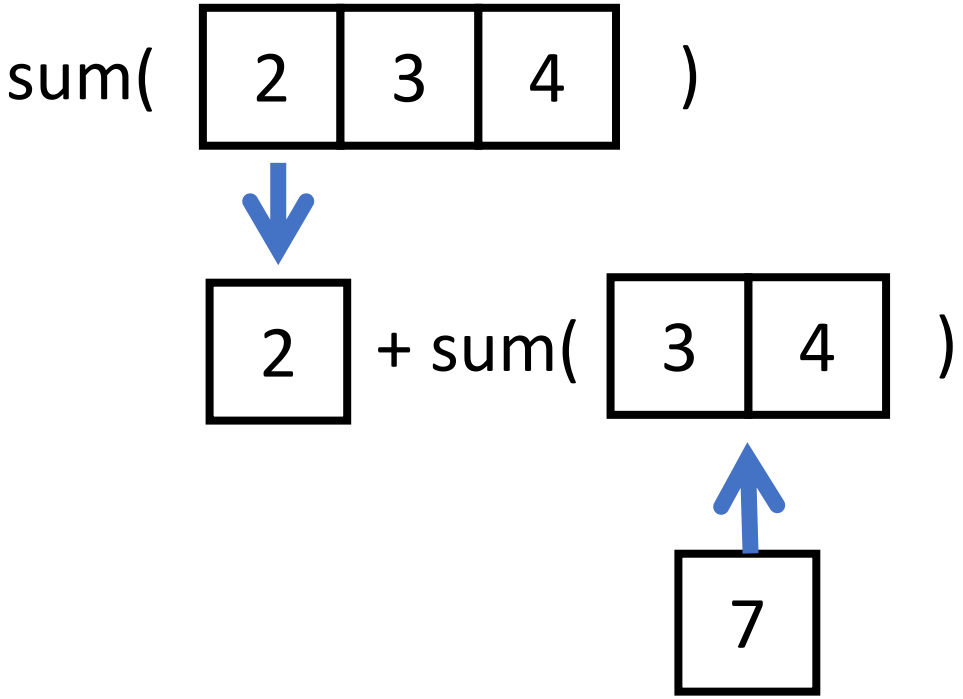


0, when a.length === 0

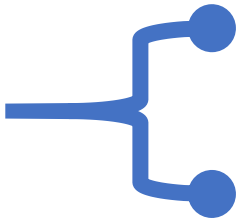
a_{first} + sum(a_{rest}), when a.length > 0

Tracing **sum(a)**

a = | 2 | 3 | 4 |

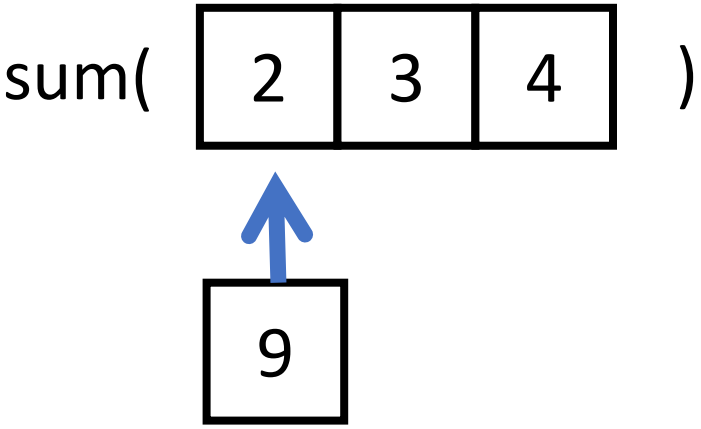


sum(a)



0, when a.length === 0

a_{first} + sum(a_{rest}), when a.length > 0

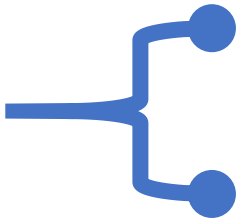


Tracing **sum(a)**

a =

2	3	4

sum(a)



0, when a.length === 0

a_{first} + sum(a_{rest}), when a.length > 0

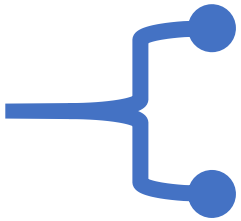
9

Tracing **sum(a)**

a =

2	3	4

sum(a)



0, when a.length === 0

a_{first} + sum(a_{rest}), when a.length > 0

Hands-on: Implement **sum** Recursively

1. Open 04-sum-app.ts

2. Implement the following logic in the sum function:
   - When a's length is 0, return 0
   - Otherwise, return the **first** element of **a** *plus* the **sum** of the **rest** of **a**

3. Reminders:
   - To get the first element use first(a)
   - To get the rest of an array use rest(a)

4. Check-in on PollEv.com/comp110 when the correct sum (9) is printing

```
function sum(a: number[]): number {
    if (a.length === 0) {
        return 0;
    } else {
        return first(a) + sum(rest(a));
    }
}
```

Follow-along:
Defining the **reduce** algorithm recursively

- Just like any looping algorithm, the reduce algorithm can be written recursively as well.

- It's a little more involved than the sum algorithm we just wrote, but not by much.

- Spending some time on your own to reason through how this recursive reduce works is a big step on the way to functional programming enlightenment…

```
function reduce<T, U>(a: T[], f: Reducer<T, U>, memo: U): U {
    if (a.length === 0) {
        return memo;
    } else {
        return reduce(rest(a), f, f(memo, first(a)));
    }
}
```