

# Recursion

---

**CPSC 217: Introduction to Computer Science for Multidisciplinary  
Studies I  
July 2024**

Jonathan Hudson, Ph.D.  
Instructor  
Department of Computer Science  
University of Calgary

*June 4, 2024*

*Copyright © 2024*



**UNIVERSITY OF  
CALGARY**

# Recursion

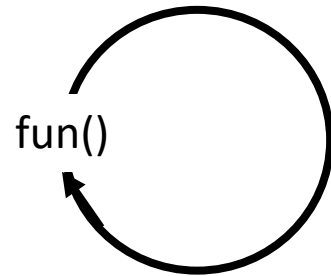
---

- Definition:
  - See Recursion
  - Defining something in terms of itself
    - Generally using a smaller or simpler version
- Recursive Function
  - A function that calls itself

# Recursion

- A programming technique whereby a function calls itself either directly or indirectly

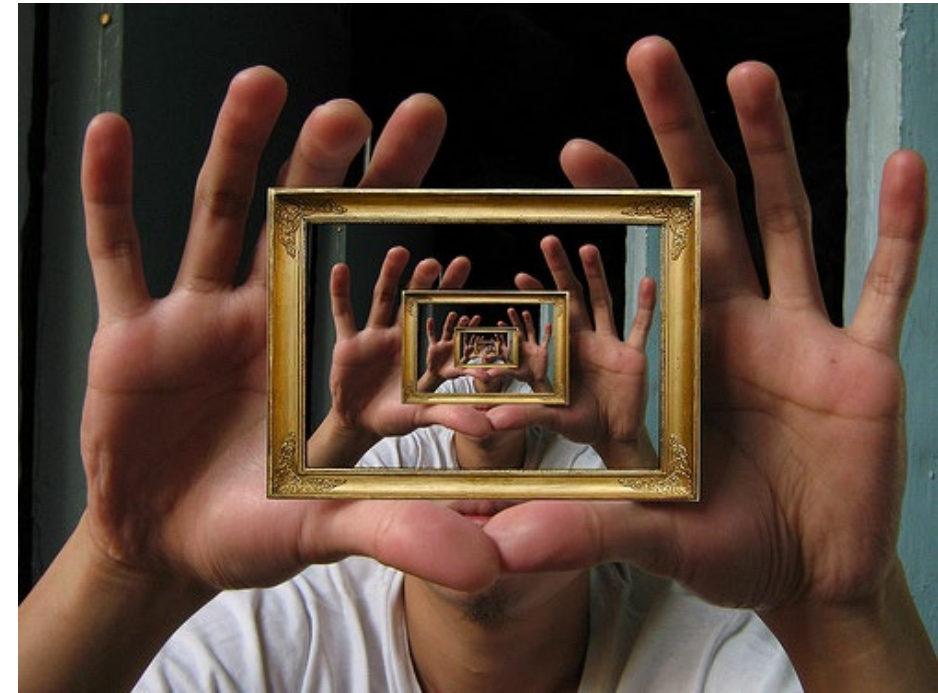
```
def fun ():  
    :  
    fun ()  
    :
```



```
def fun1 ():  
    fun2 ()  
  
def fun2 ():  
    fun1 ()
```

indirectly (mutual recursive)

Directly



# Recursion

---

- A well-formed recursive function normally has two cases
  - Base Case:
    - Does not make a recursive call
    - Permits function to return (allowing us to fall out of the function calls)
    - Without a well-formed base case, you will have infinite recursive calls leading to the popular *StackOverflowException* or equivalently *RecursionError* in Python
  - Recursive Case:
    - Function calls itself
    - Generally must be a call to a smaller or simpler version of the problem

# Tail Recursion

---

# Tail Recursion

---

- a tail call is a subroutine call performed as the final action of a function (Ex. A final return statement function call)
- If a tail call calls the function itself (as the last thing the function does), then this function is called tail recursive

#Not tail-recursive

```
def fact(n):
```

```
    if (n == 0):
```

```
        return 1
```

```
    return n * fact(n-1)
```

#Tail recursive

```
def fact(n, a = 1):
```

```
    if (n == 1):
```

```
        return a
```

```
    return fact(n - 1, n * a)
```

# Tail Recursion

---

- Typically when a function is called a record of memory (stack frame) is stored to track state from the previous function call
- But a tail recursive function doesn't need anything from before the final function call
- Modern compilers can identify this and reduce the process into an iterative loop for you. Although this is not guaranteed.
  - python is a procedural language, there is a class of languages called functional languages which will often guarantee this behaviour
  - Python does not optimize tail recursion (neither does Java)

# Usage of Recursion

---



# Useful Examples of Recursion

---

- Drawing fractals
- Finding a path through a maze
- Flood fill / “paint bucket” tool
- Merge sort, quick sort, binary search
- Finding the total size of all of the files in a directory and its subdirectories
- Parsing / evaluating expressions (how your code compiler knows you have syntax errors!)
- ...

# Factorial

---

# A Small Example - Factorial

---

Computing n factorial (n!):

- Defined as the result of multiplying all numbers from n to 1 for all  $n > 0$ . If  $n = 0$ , then result is 1.

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n - 1)! & n > 0 \end{cases}$$

- $0! \rightarrow 1$
- $1! \rightarrow 1 * 1 = 1$
- $2! \rightarrow 2 * 1 * 1 = 2$
- $3! \rightarrow 3 * 2 * 1 * 1 = 6$

# A Small Example - Factorial

---

Computing n factorial:

- Using a loop:
  - Initialize a variable named *answer* to 1 (the answer to 0!)
  - for i ranging from 2 to n (inclusive)
    - Multiply *answer* by i, storing the result back into *answer*
  - return *answer*

# A Small Example - Factorial

---

Computing  $n$  factorial:

- Using a loop:
  - Initialize a variable named *answer* to 1 (the answer to 0!)
  - for  $i$  ranging from 2 to  $n$  (inclusive)
    - Multiply *answer* by  $i$ , storing the result back into *answer*
  - return *answer*
- Another solution comes from the definition

# A Small Example

---

## Recursive Definition!

- Another solution
  - By definition,
  - View

$$0! == 1$$

$$n! == n * (n-1)!$$

# A Small Example

---

## Recursive Definition!

- Another solution
  - BASE CASE,
  - RECURSIVE CASE,

$$0! == 1$$

$$n! == n * (n-1)!$$

# A Small Example - Factorial

---

```
def factorial(n):  
    if n == 0: #BASE CASE  
        return 1  
    return n * factorial(n-1) #RECURSIVE CASE
```

Note: code is easy, directly follows from recursive definition

How does it look like in memory?



# Factorial - Memory

---

# A Small Example - Factorial

---

```
def factorial(n):  
    if n == 0: #BASE CASE  
        return 1  
    return n * factorial(n-1) #RECURSIVE CASE
```

Note: code is easy, directly follows from recursive definition

How does it look like in memory?

# A Small Example - Factorial

---

- Let's trace this code:

```
def factorial(n):  
    if n == 0: #BASE CASE  
        return 1  
    return n * factorial(n-1) #RECURSIVE CASE  
factorial(3)
```

# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

Trace:

STACK

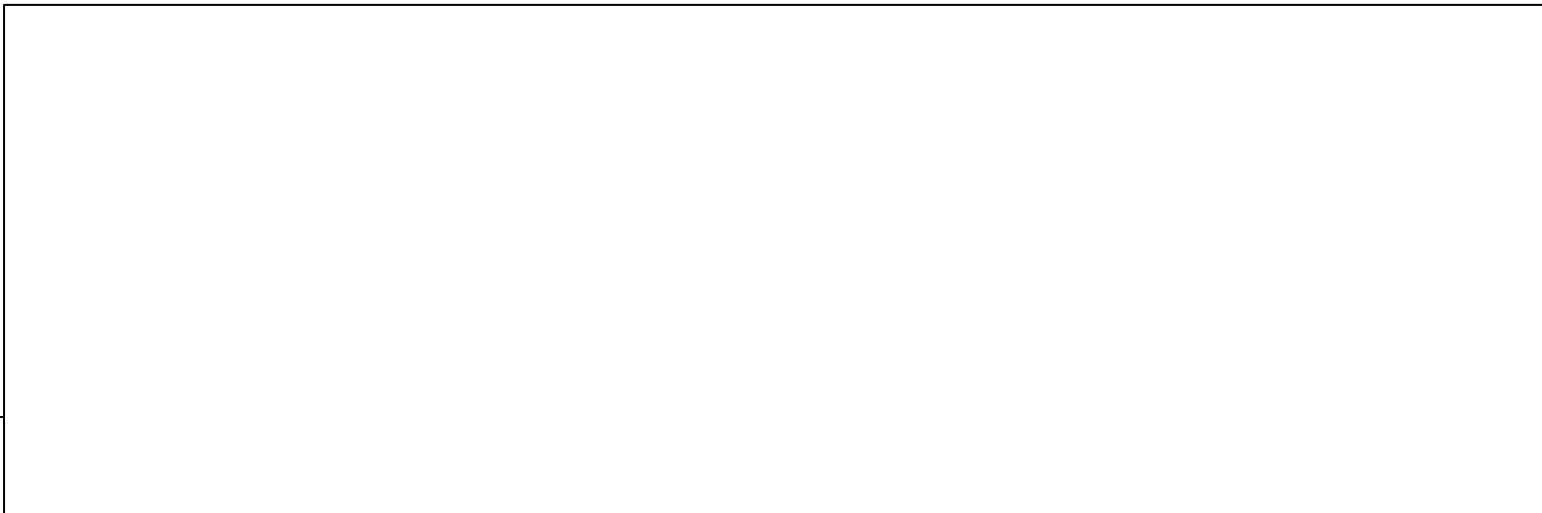
# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)
```

 factorial(3)

Trace:



STACK



# A Small Example - Factorial - Walkthrough

Code:

STACK

11

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

Trace:

n=3

Factorial(3)  
n=3

# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

Trace:

n=3

STACK

Factorial(3)  
n=3

# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

FALSE



`if n == 0:`

`return 1`

`return n * factorial(n-1)`

Trace:

n=3

STACK

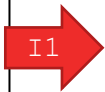
Factorial(3)  
n=3



# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```



Trace:

```
n=3  
return 3*(factorial(2))
```

STACK

Factorial(3)  
n=3

# A Small Example - Factorial - Walkthrough

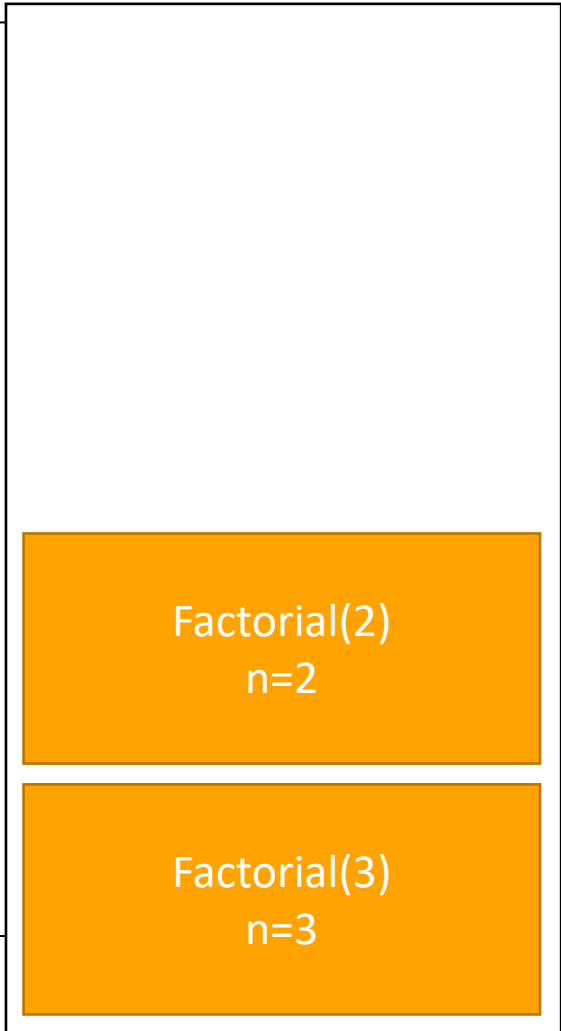
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=2  
return 3*(factorial(2))
```

STACK



# A Small Example - Factorial - Walkthrough

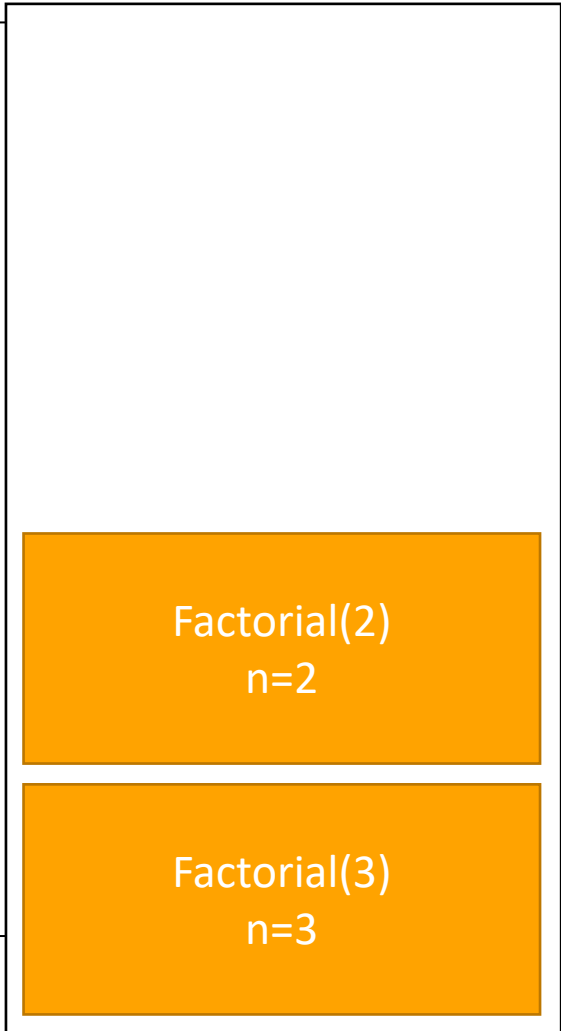
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=2  
return 3*(factorial(2))
```

STACK



# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

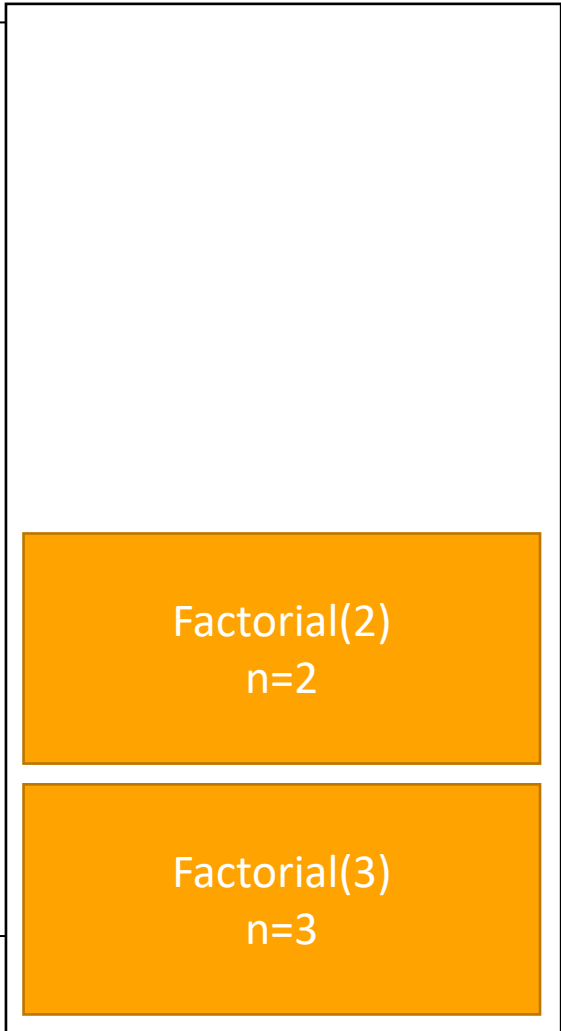
FALSE



Trace:

```
n=2  
return 3*(factorial(2))
```

STACK



# A Small Example - Factorial - Walkthrough

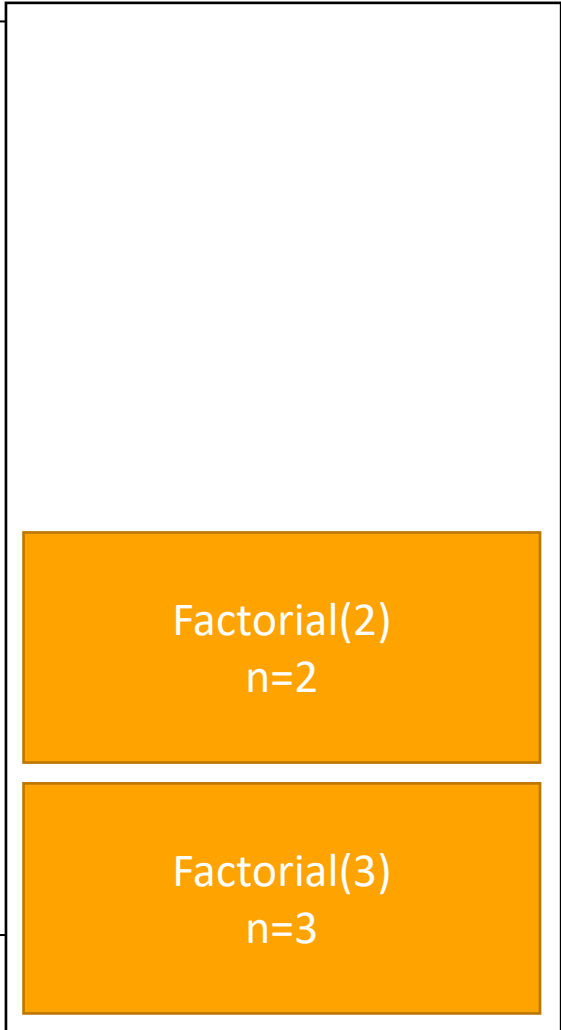
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    12 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=2  
return 3*(2*factorial(1))
```

STACK



# A Small Example - Factorial - Walkthrough

Code:

STACK

I3

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

I2

Trace:

```
n=1  
return 3*(2*factorial(1))
```

Factorial(1)  
n=1

Factorial(2)  
n=2

Factorial(3)  
n=3

# A Small Example - Factorial - Walkthrough

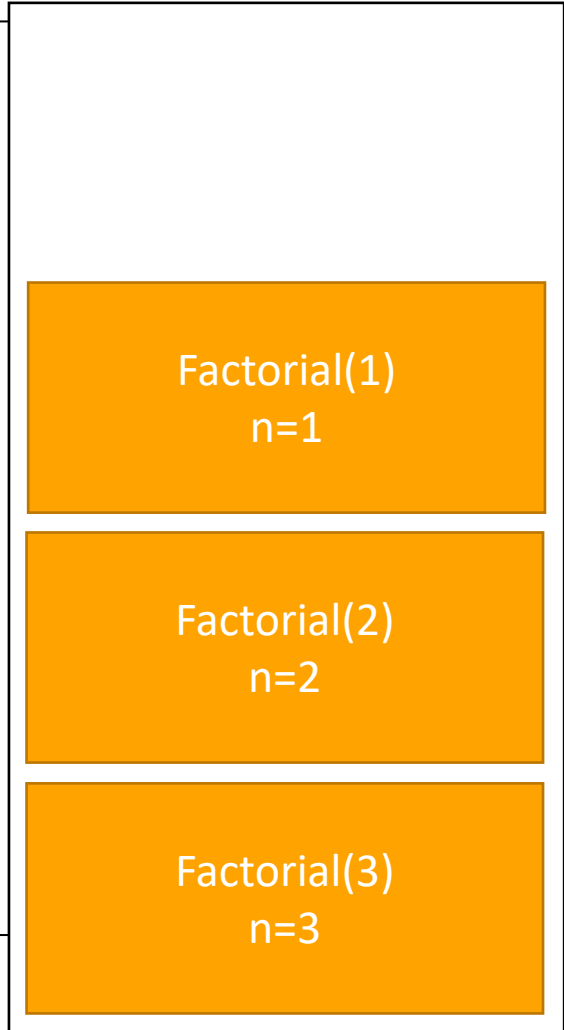
Code:

```
def factorial(n):  
    I3 → if n == 0:  
        return 1  
    I2 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=1  
return 3 * (2 * factorial(1))
```

STACK



# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

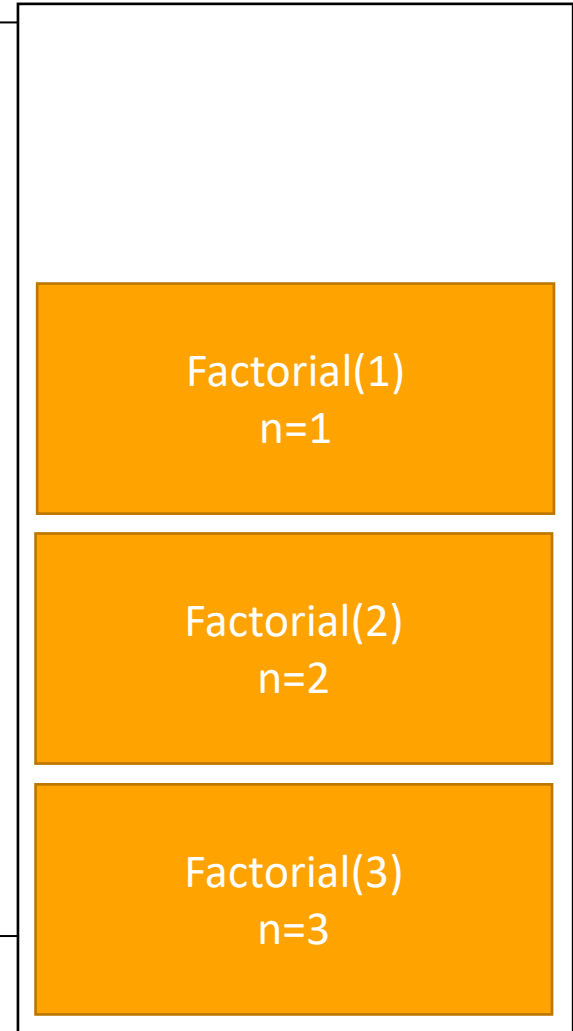
FALSE



Trace:

```
n=1  
return 3*(2*factorial(1))
```

STACK





# A Small Example - Factorial - Walkthrough

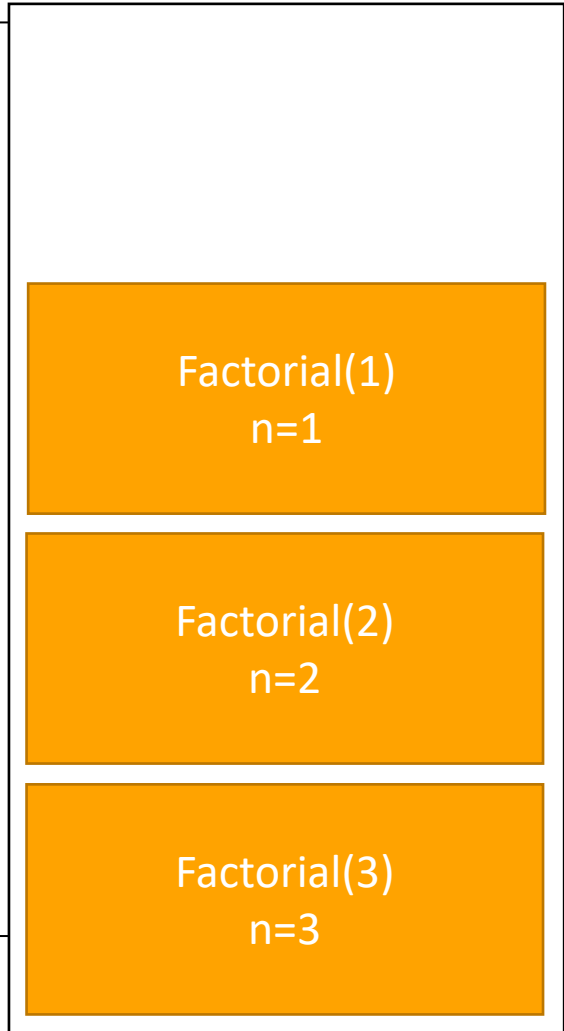
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    I3 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=1  
return 3 * (2 * (1 * factorial(0)))
```

STACK



# A Small Example - Factorial - Walkthrough

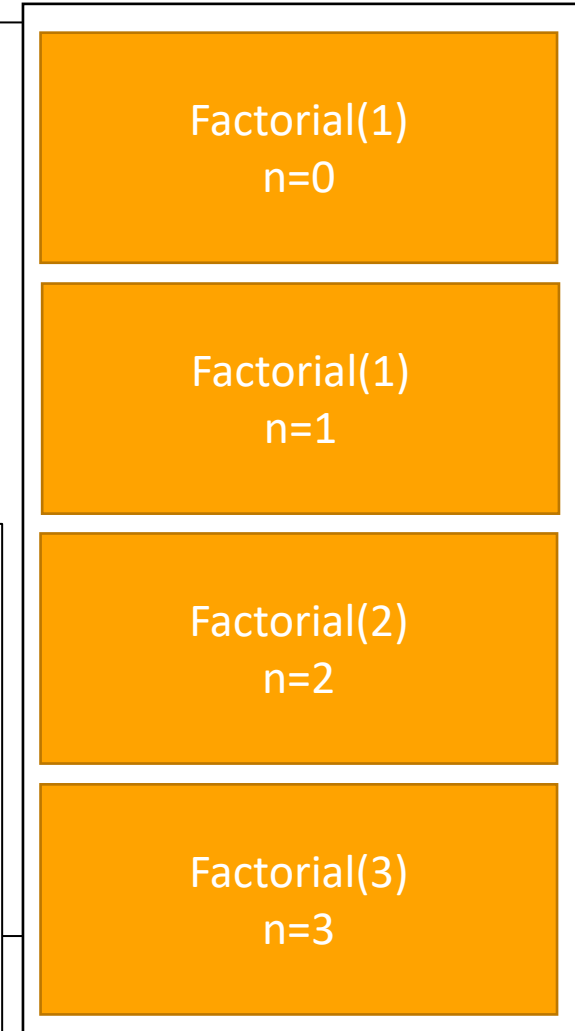
Code:

STACK

```
I4 → def factorial(n):  
    if n == 0:  
        return 1  
I3 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=0  
return 3*(2*(1*factorial(0)))
```



# A Small Example - Factorial - Walkthrough

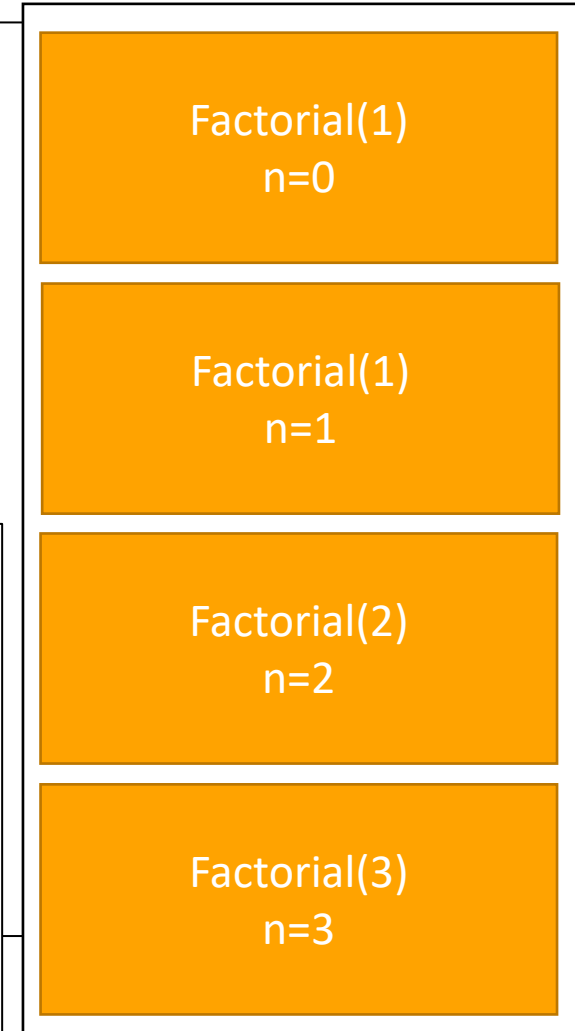
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=0  
return 3*(2*(1*factorial(0)))
```

STACK



# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

TRUE

I4

I3

Trace:

```
n=0  
return 3*(2*(1*factorial(0)))
```

STACK

Factorial(1)  
n=0

Factorial(1)  
n=1

Factorial(2)  
n=2

Factorial(3)  
n=3

# A Small Example - Factorial - Walkthrough

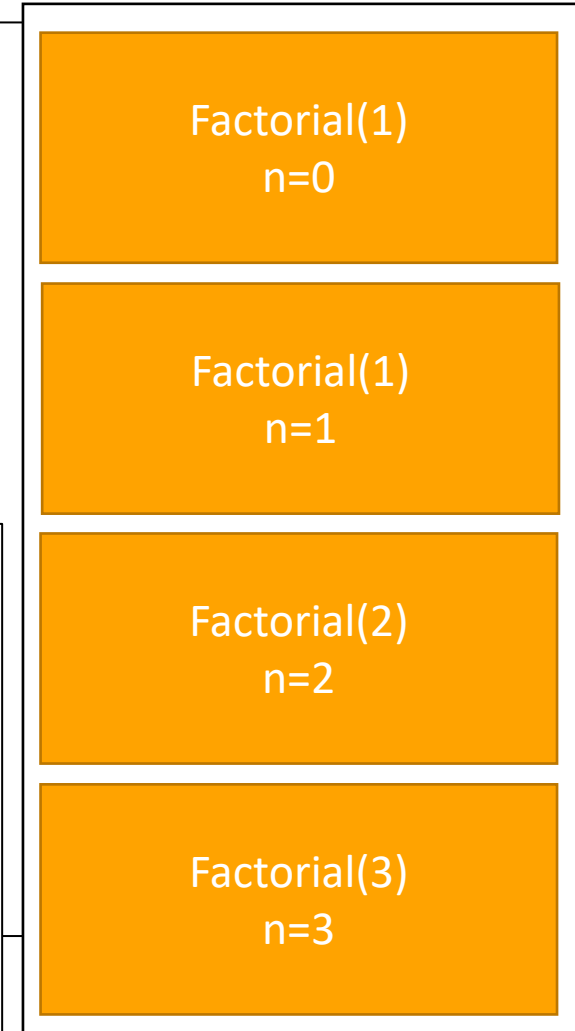
Code:

```
def factorial(n):  
    if n == 0:  
        I4 → return 1  
    I3 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=0  
return 3*(2*(1*factorial(0)))
```

STACK



# A Small Example - Factorial - Walkthrough

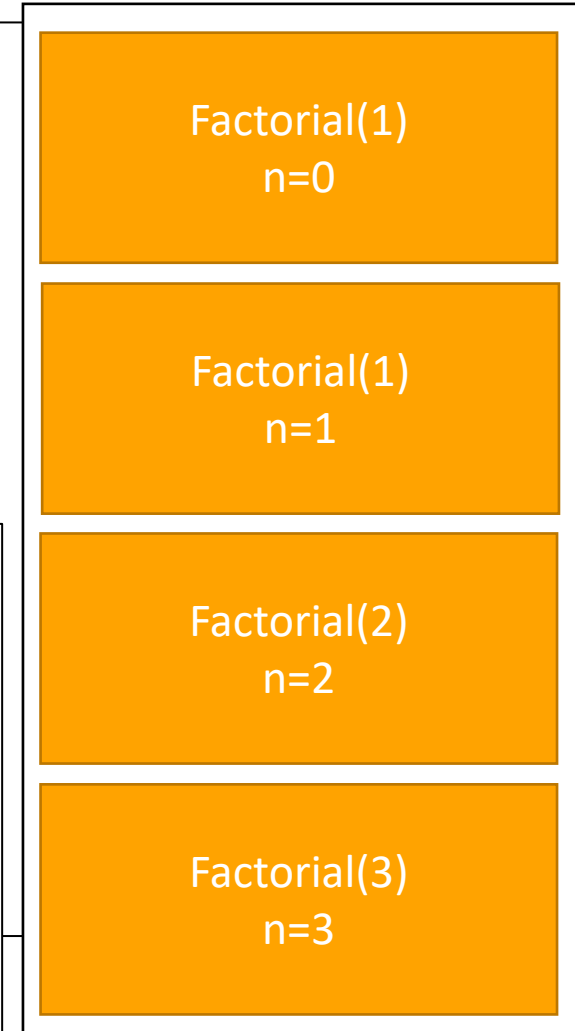
Code:

```
def factorial(n):  
    if n == 0:  
        I4 → return 1  
    I3 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=0  
return 3 * (2 * (1 * factorial(0)))  
                    └──────────┘  
                    factorial(0) return 1
```

STACK



# A Small Example - Factorial - Walkthrough

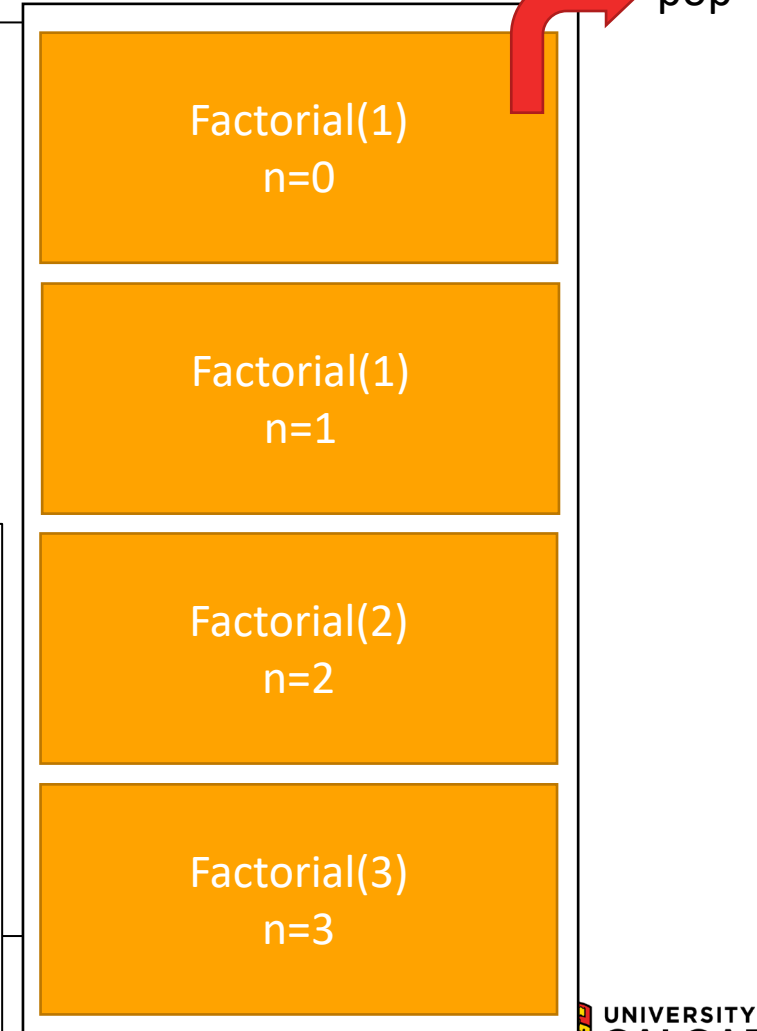
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=1  
return 3 * (2 * (1 * 1))
```

STACK



# A Small Example - Factorial - Walkthrough

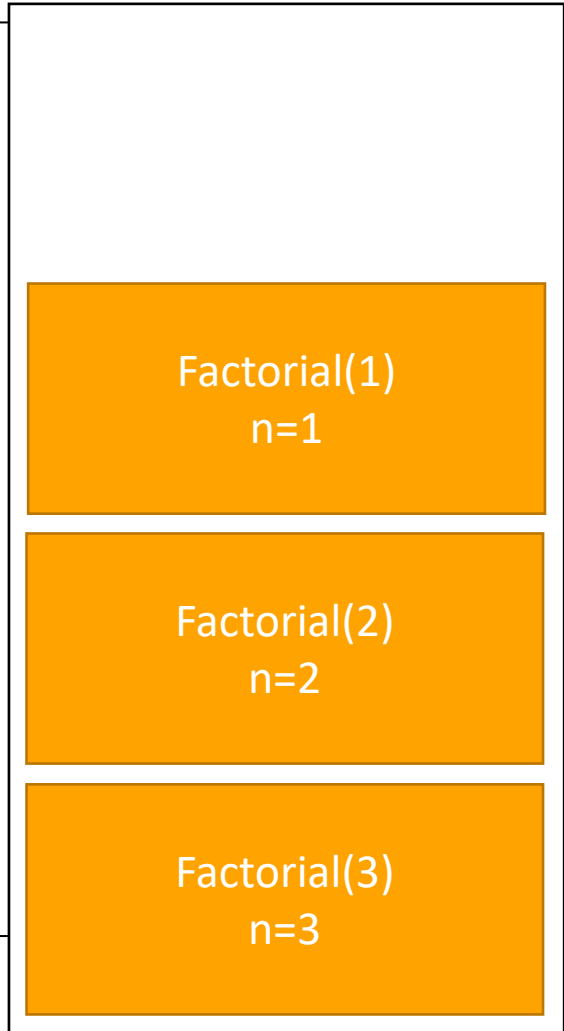
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    I3 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=1  
return 3 * (2 * (1 * 1))
```

STACK





# A Small Example - Factorial - Walkthrough

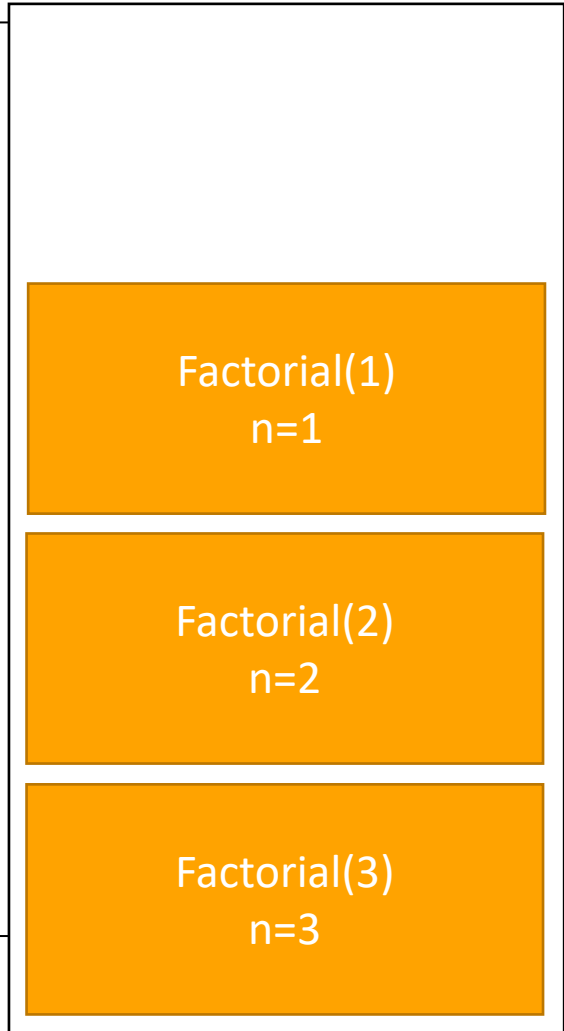
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    ↪ return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=1  
return 3 * (2 * (1 * 1))  
                └──┬──  
                factorial(1) returns  
                1 * 1 = 1
```

STACK



# A Small Example - Factorial - Walkthrough

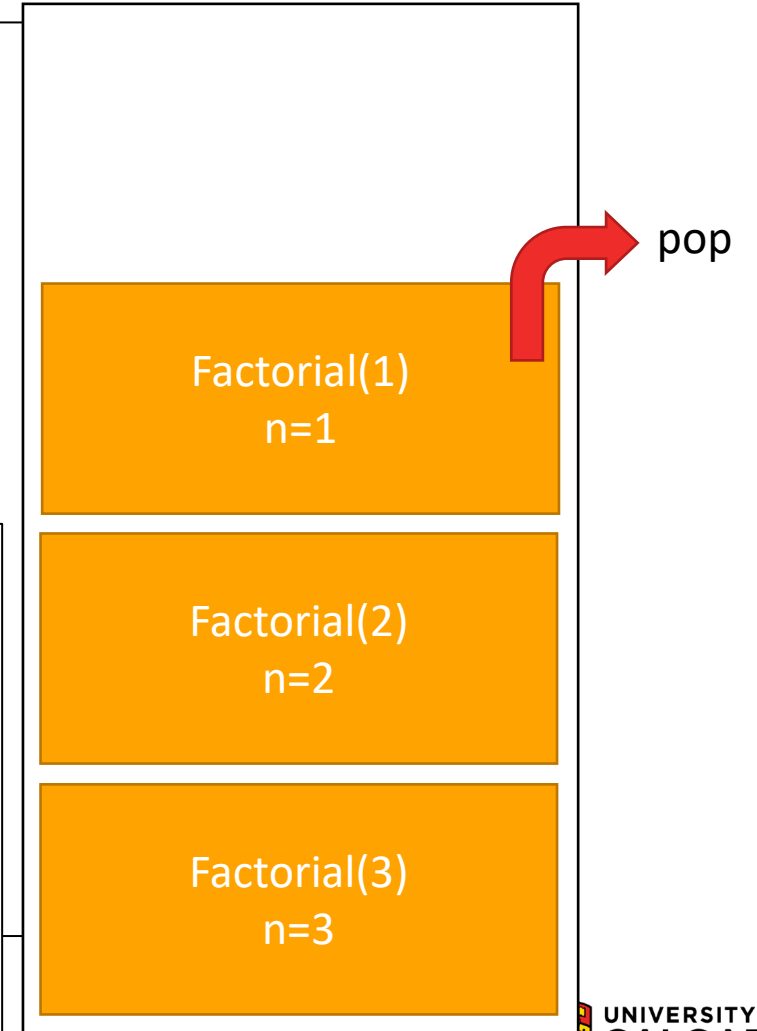
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    I2 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=2  
return 3 * (2 * 1)
```

STACK



# A Small Example - Factorial - Walkthrough

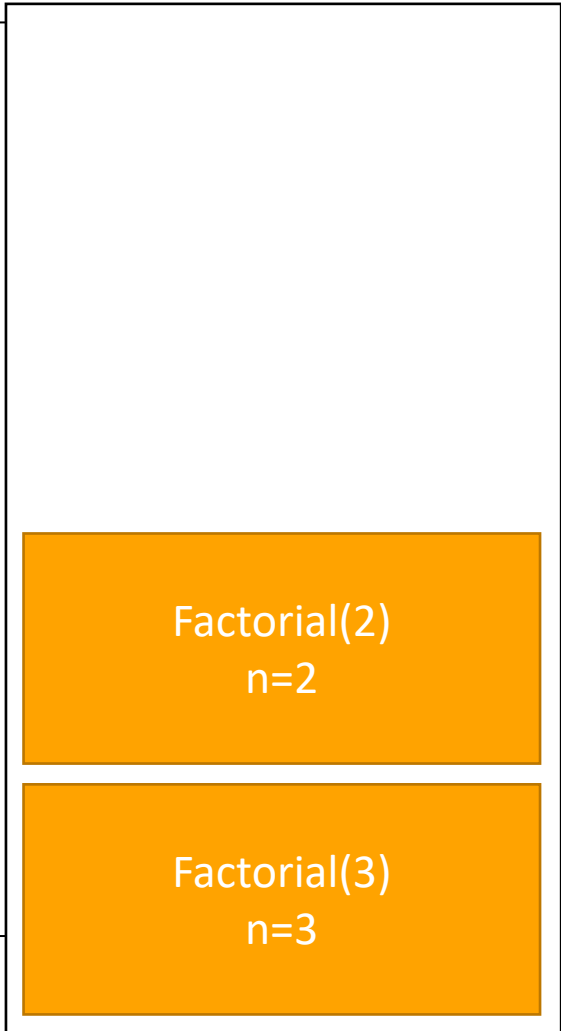
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    I2 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=2  
return 3 * (2 * 1)
```

STACK



# A Small Example - Factorial - Walkthrough

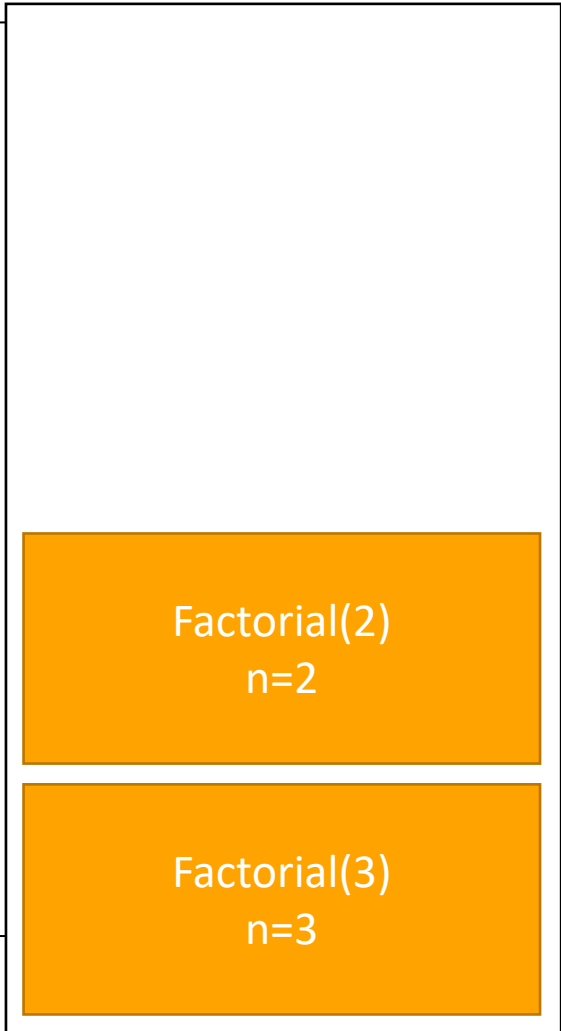
Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    I2 → return n * factorial(n-1)  
factorial(3)
```

Trace:

```
n=2  
return 3 * (2 * 1)  
           └─┬─┘  
           factorial(2) returns  
           2 * 1 = 2
```

STACK



# A Small Example - Factorial - Walkthrough

Code:

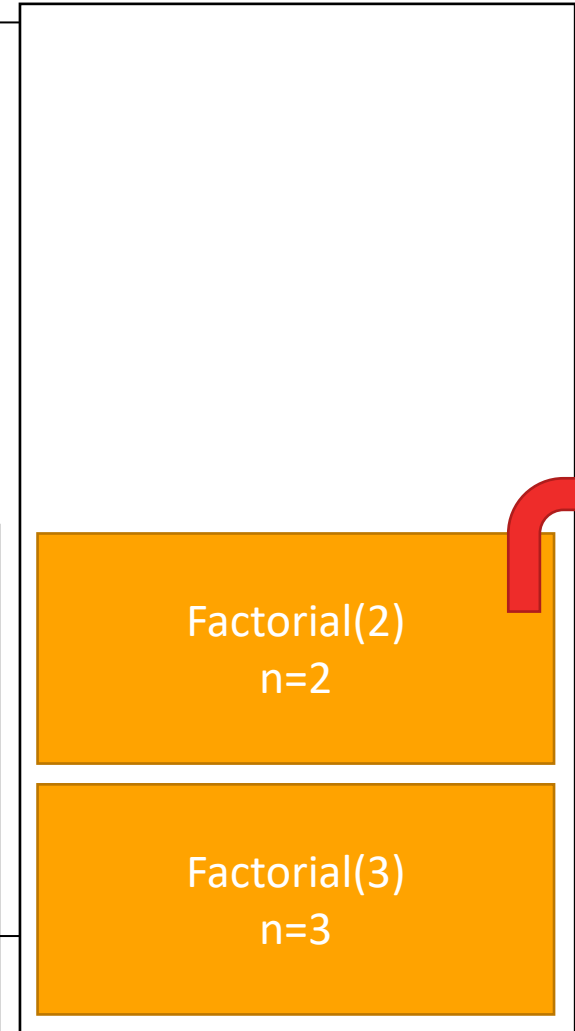
```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```



Trace:

```
n=3  
return 3*2
```

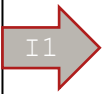
STACK



# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```



Trace:

```
n=3  
return 3*2
```

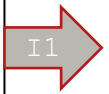
STACK

Factorial(3)  
n=3


# A Small Example - Factorial - Walkthrough

Code:

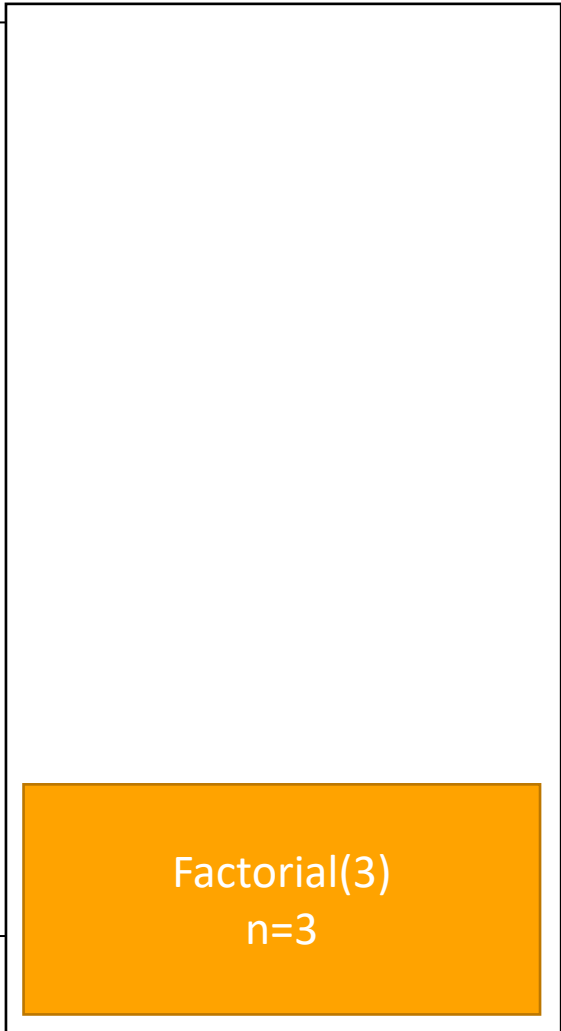
```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```



Trace:

```
n=3  
return 3*2  
        
factorial(3) returns  
3*2 = 6
```

STACK



# A Small Example - Factorial - Walkthrough

Code:

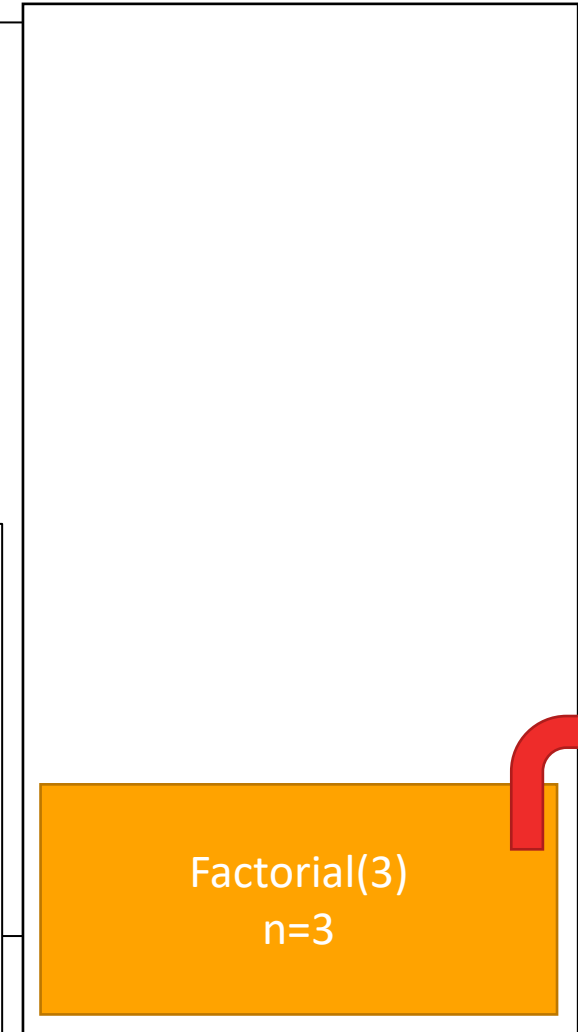
```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```



Trace:

```
n=3  
return 6
```

STACK





# A Small Example - Factorial - Walkthrough

Code:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
factorial(3)
```



Trace:

6

STACK

# Fibonacci

---

# Fibonacci Numbers

---

- A sequence of values:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

# Fibonacci Numbers

---

- A sequence of values:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Defined recursively:
  - By definition:
    - fib(0) is 0
    - fib(1) is 1

# Fibonacci Numbers

---

- A sequence of values:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Defined recursively:
  - By definition:
    - fib(0) is 0
    - fib(1) is 1
  - Remaining values:
    - Formed by computing the sum of the previous two values in the sequence

# Fibonacci Numbers

---

- A sequence of values:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Defined recursively:
  - By definition:
    - fib(0) is 0
    - fib(1) is 1
  - Remaining values:
    - $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

# Fibonacci Numbers

---

Note that there are two base cases here!

Eg. fib(4)

$$\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$$

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$$

$$\text{fib}(2) = \text{fib}(1) + \text{fib}(0)$$

$$\text{fib}(1) = 1$$

$$\text{fib}(0) = 0$$

# Fibonacci Numbers

---

Note that there are two base cases here!

Eg. fib(4)

$$\text{fib}(4) = \text{fib}(3) + \text{fib}(2) = \text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(1) + \text{fib}(0) = 3$$

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1) = \text{fib}(1) + \text{fib}(0) + \text{fib}(1) = 2$$

$$\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(0) = 0$$



# Example - Fibonacci Numbers

---

Note that there are two base cases here!

Eg. fib(4)

$$\text{fib}(4) = \text{fib}(3) + \text{fib}(2) = \underbrace{\text{fib}(1) + \text{fib}(0) + \text{fib}(1)}_{\text{fib}(2)} + \underbrace{\text{fib}(1) + \text{fib}(0)}_{\text{fib}(2)} = 3$$

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1) = \text{fib}(1) + \text{fib}(0) + \text{fib}(1) = 2$$

$$\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(0) = 0$$

# Fibonacci Numbers

---

Note that there are two base cases here!

Eg. fib(5)

$$\text{fib}(5) = \text{fib}(4) + \text{fib}(3) = \text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(0) + \text{fib}(1) = 5$$

$$\text{fib}(4) = \text{fib}(3) + \text{fib}(2) = \text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(1) + \text{fib}(0) = 3$$

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1) = \text{fib}(1) + \text{fib}(0) + \text{fib}(1) = 2$$

$$\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(0) = 0$$

# Example - Fibonacci Numbers

---

Note that there are two base cases here!

Eg. fib(5)

$$\text{fib}(5) = \text{fib}(4) + \text{fib}(3) = \overbrace{\text{fib}(1)+\text{fib}(0)+\text{fib}(1)+\text{fib}(1)+\text{fib}(0)}^{\text{fib}(4)} + \overbrace{\text{fib}(1)+\text{fib}(0)+\text{fib}(1)}^{\text{fib}(3)} = 5$$

$$\text{fib}(4) = \text{fib}(3) + \text{fib}(2) = \text{fib}(1)+\text{fib}(0)+\text{fib}(1)+\text{fib}(1)+\text{fib}(0) = 3$$

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1) = \text{fib}(1)+\text{fib}(0)+\text{fib}(1) = 2$$

$$\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(0) = 0$$

# Fibonacci Numbers

---

```
def fibRec(n):  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    return fibRec(n-1) + fibRec(n-2)
```

# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

fib(4)

fib(4)

# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(4) \rightarrow fib(3) + fib(2)$

fib(4)

# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

fib(4) → **fib(3)** + fib(2)

The expression is evaluated left to right.  
So, fib(3) is calculated first.

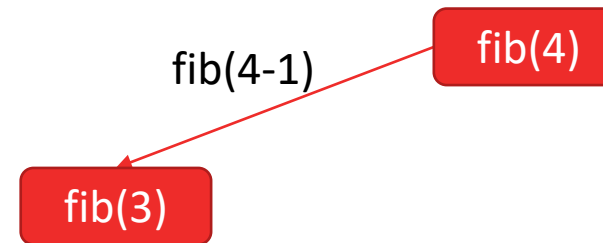
fib(4)

# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

fib(3)



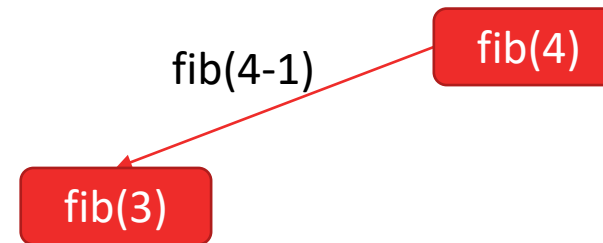


# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(3) \rightarrow fib(2) + fib(1)$



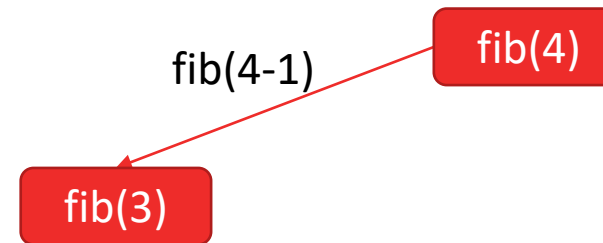
# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(3) \rightarrow \mathbf{fib(2)} + fib(1)$

Similarly,  $fib(2)$  is evaluated first.

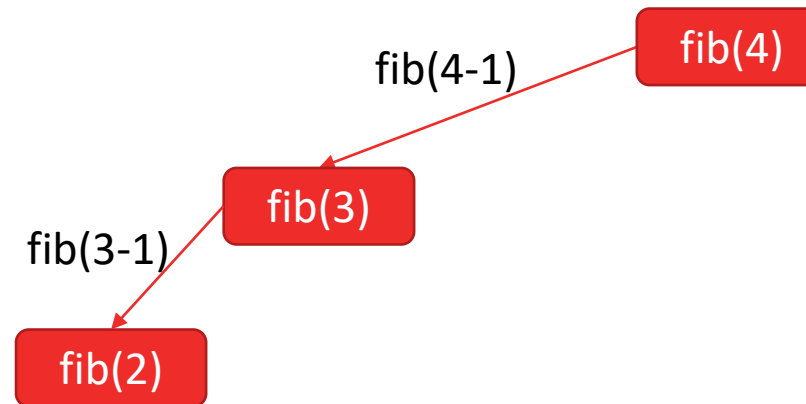


# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

fib(2)

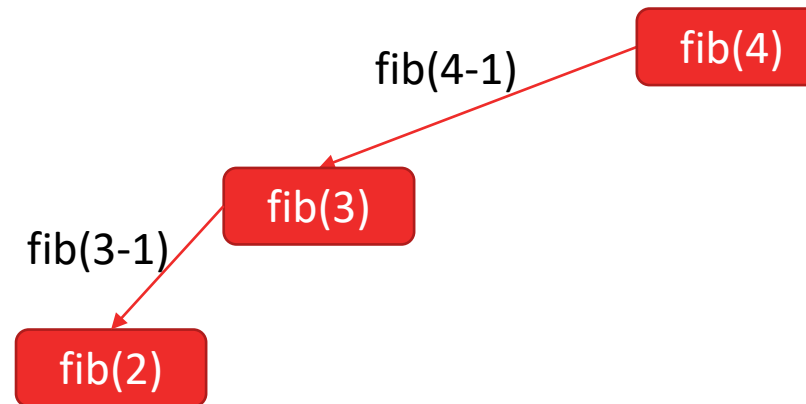


# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(2) \rightarrow fib(1) + fib(0)$



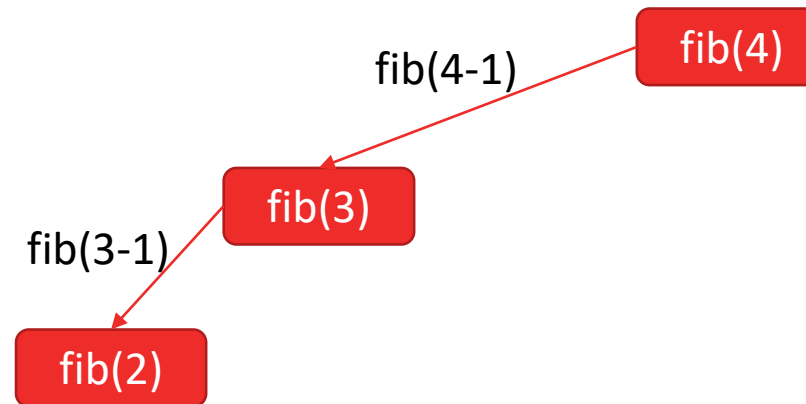
# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(2) \rightarrow fib(1) + fib(0)$

$fib(1)$  is evaluated first

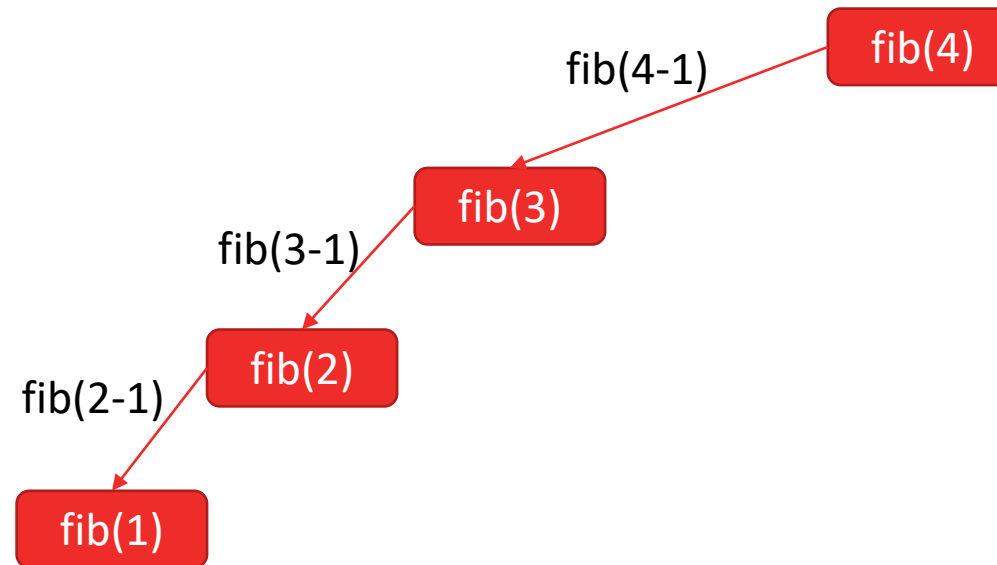


# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

fib(1)



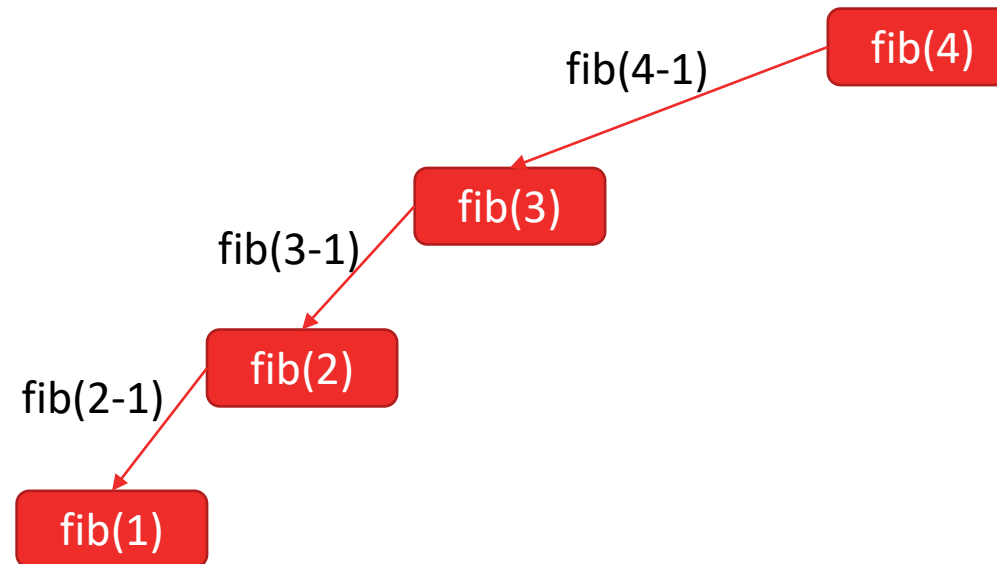
# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

fib(1) → 1

fib(1) returns 1 to caller



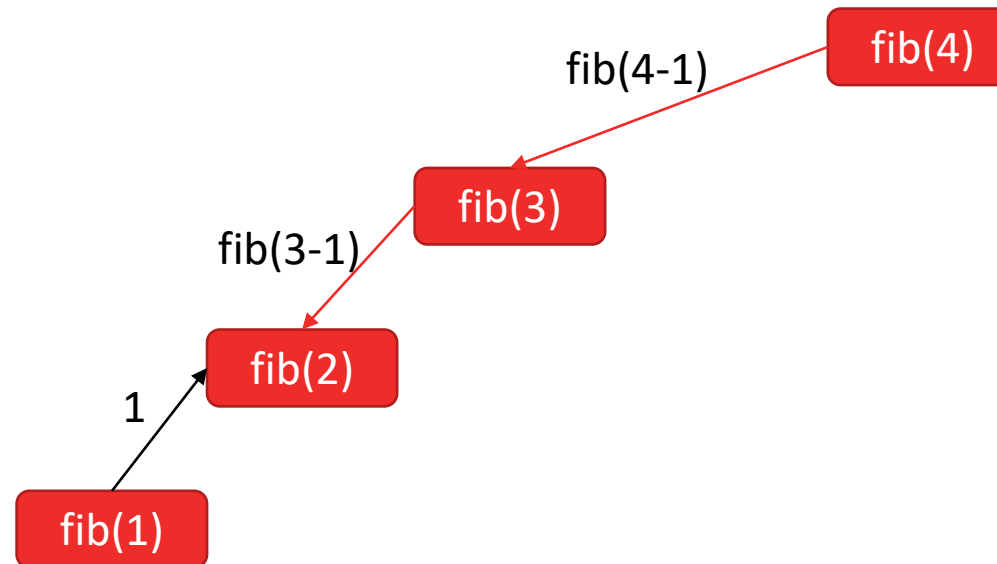
# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

fib(1) → 1

fib(1) returns 1 to caller





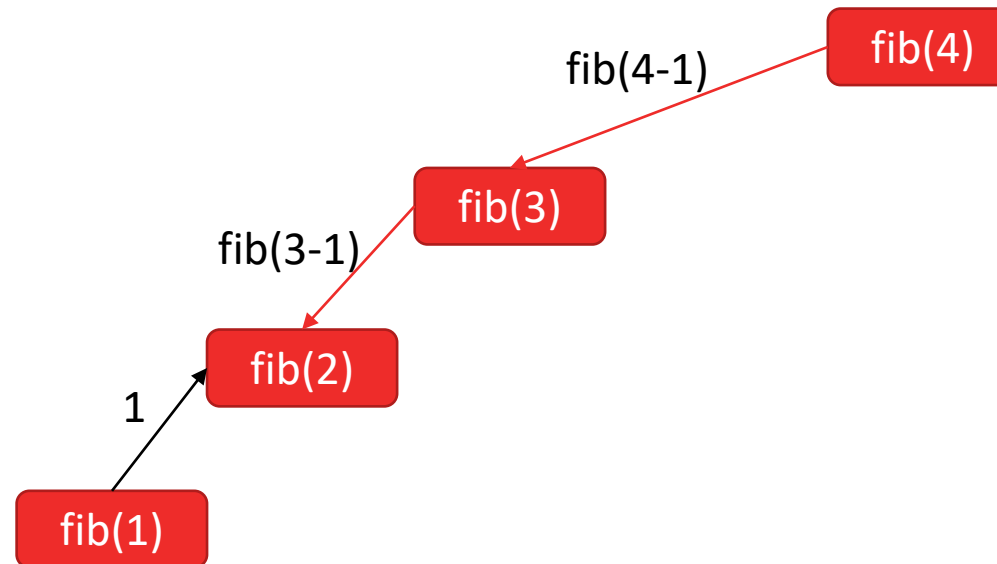
# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(2) \rightarrow 1 + \mathbf{fib(0)}$

Now,  $fib(0)$  is evaluated.

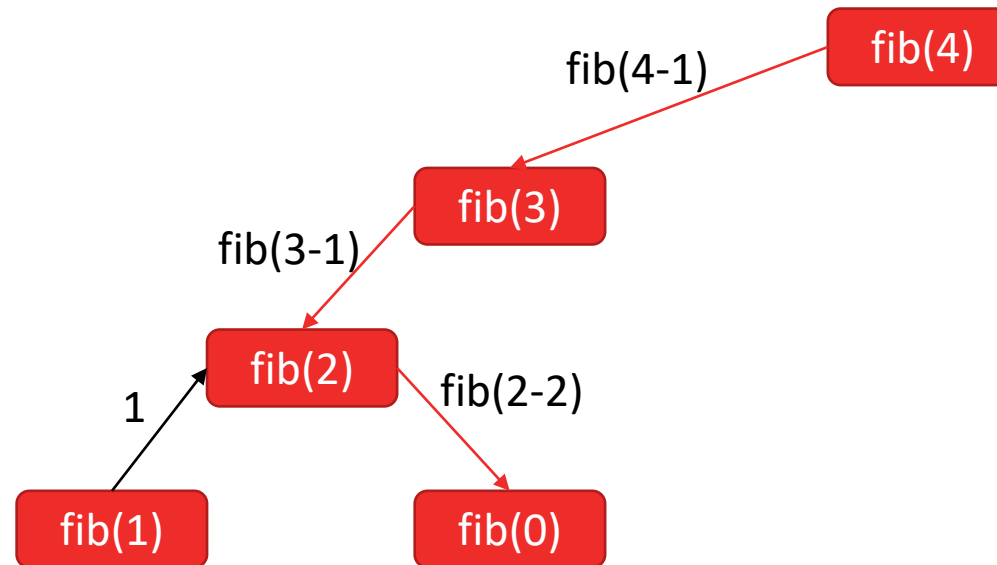


# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

fib(0)



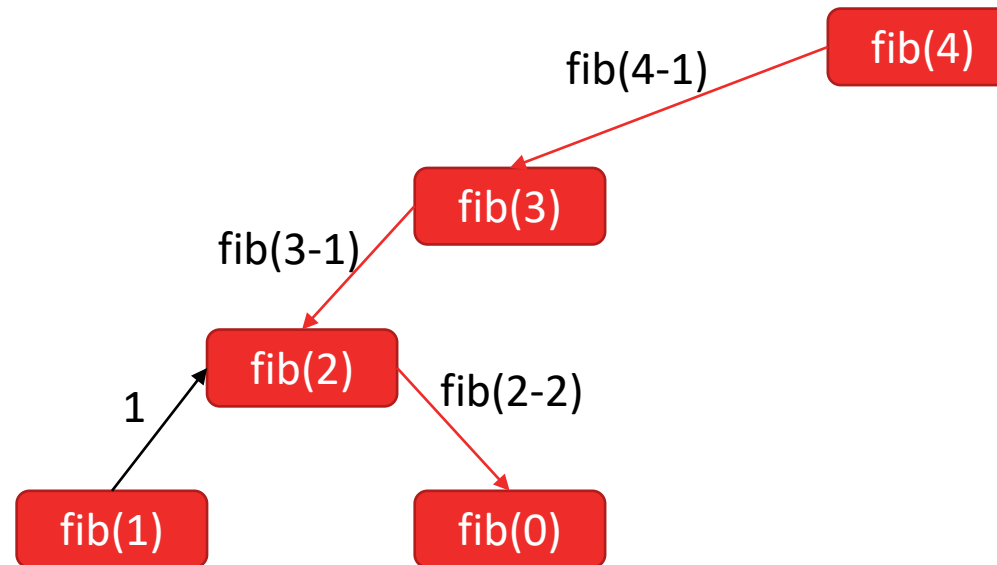
# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(0) \rightarrow 0$

$fib(0)$  returns 0 to caller

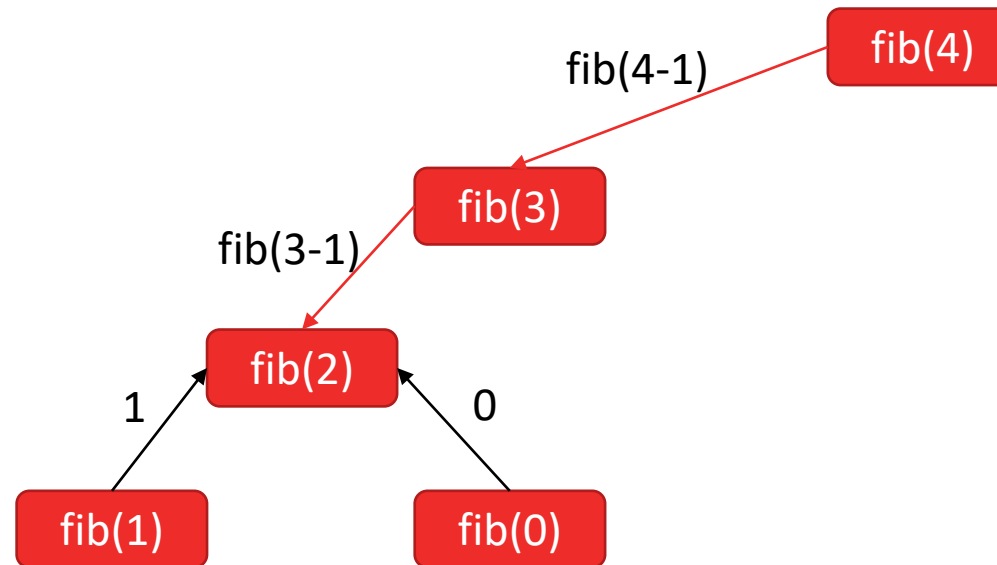


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(0) \rightarrow 0$

$fib(0)$  returns 0 to caller

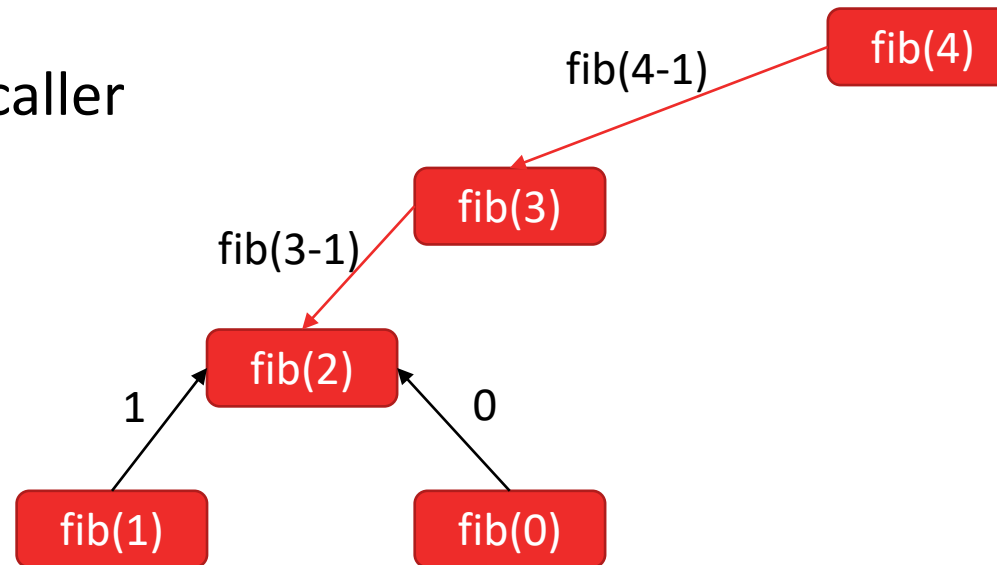


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(2) \rightarrow 1+0 = 1$

Now,  $fib(2)$  returns 1 to the caller



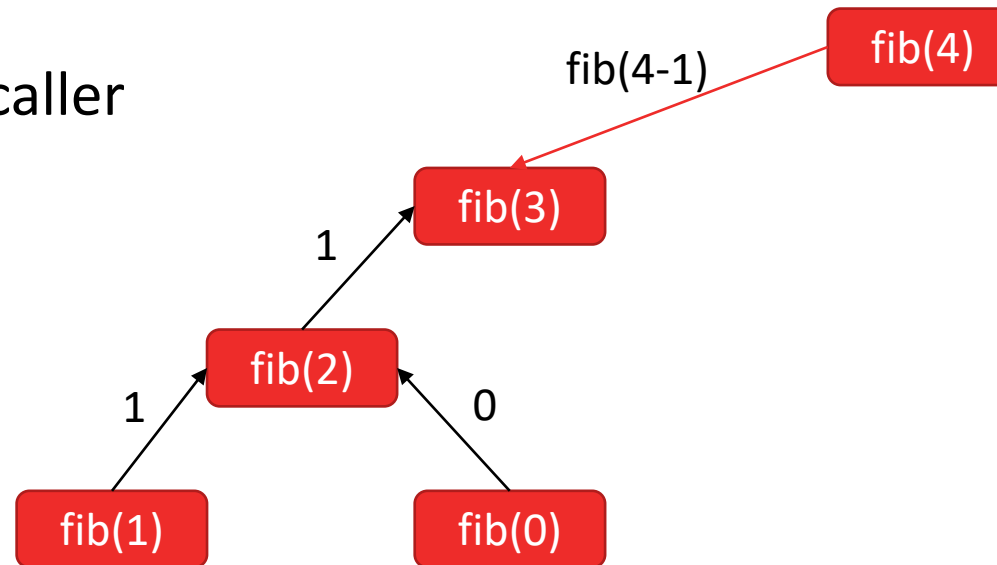
# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(2) \rightarrow 1+0 = 1$

Now,  $fib(2)$  returns 1 to the caller

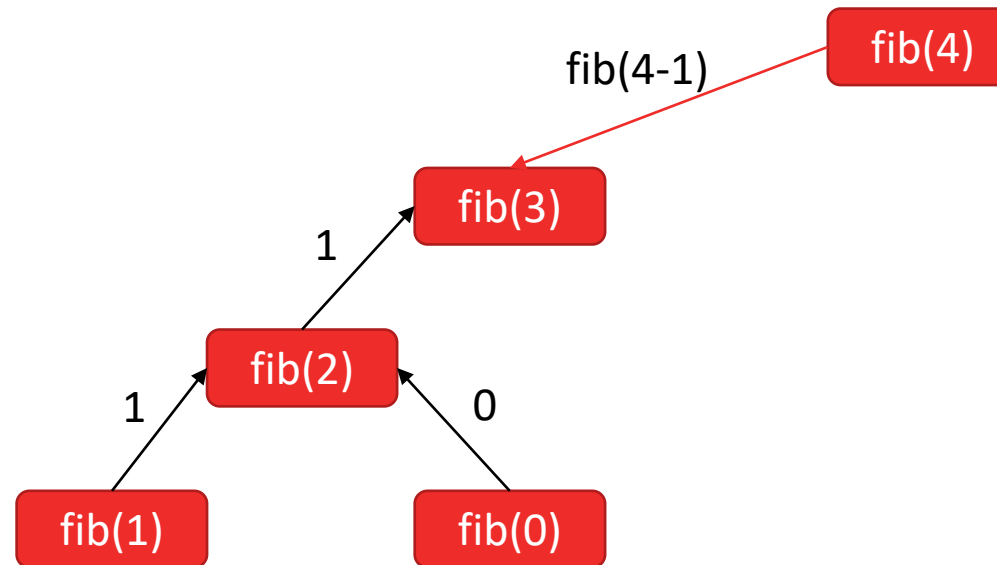


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(3) \rightarrow 1 + \mathbf{fib(1)}$

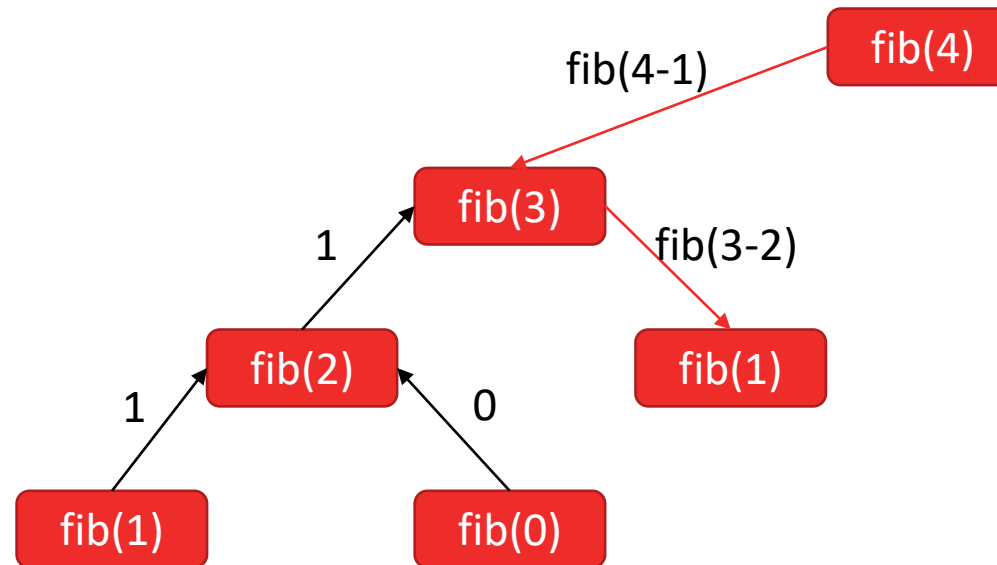
Now,  $fib(1)$  is calculated



# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

fib(1)



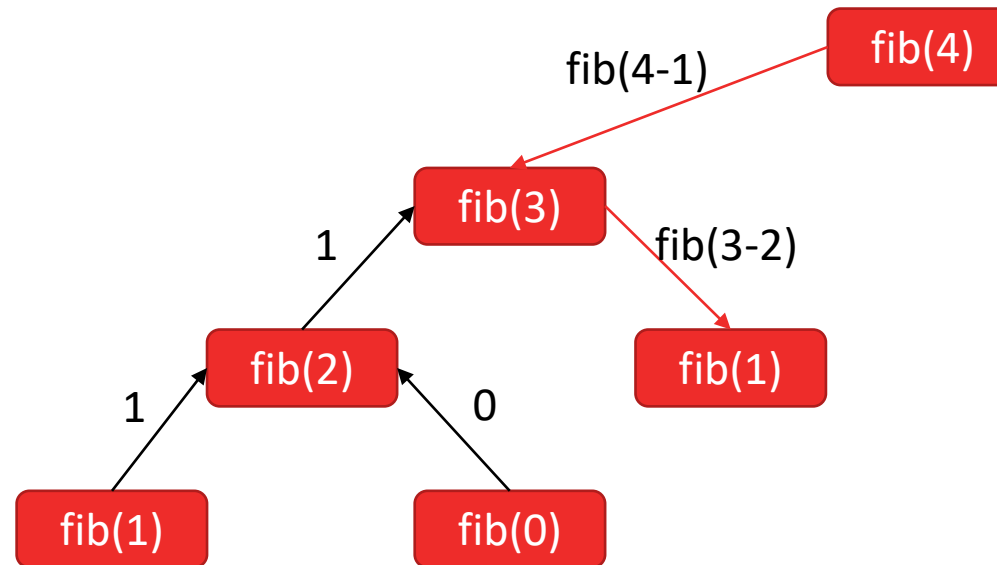


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(1) \rightarrow 1$

$fib(1)$  returns 1 to the caller.

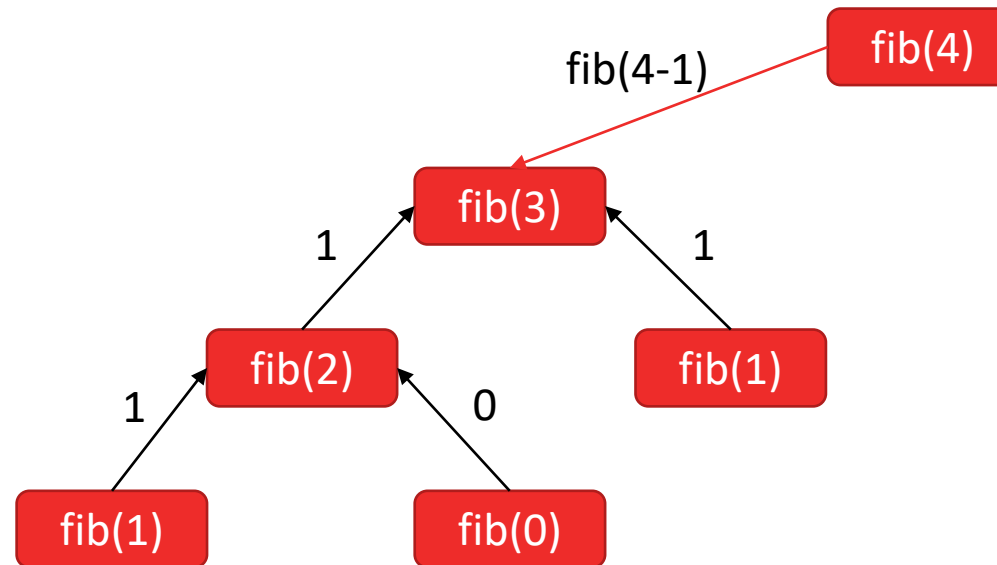


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

fib(1) → 1

fib(1) returns 1 to the caller.

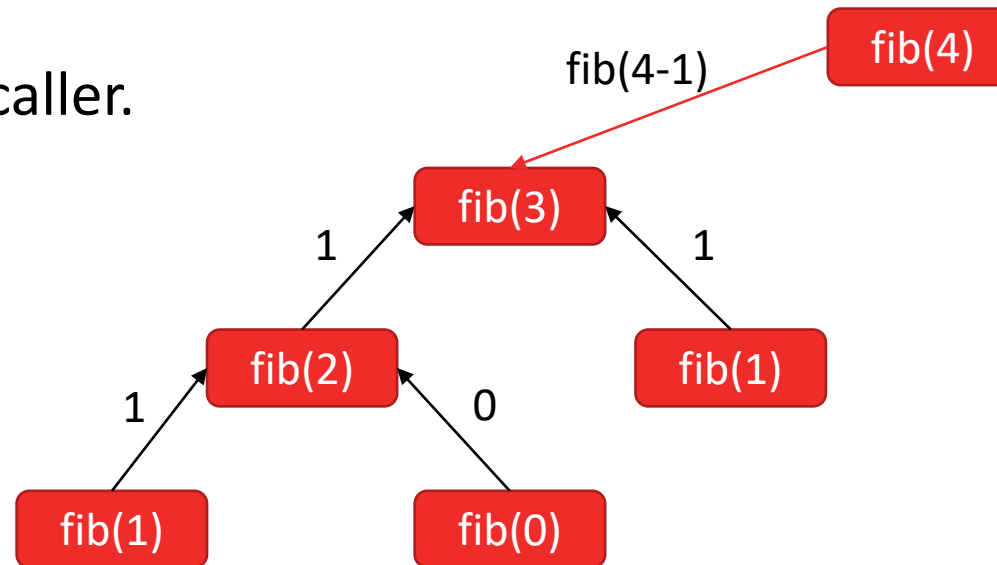


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(3) \rightarrow 1 + 1 = 2$

Now,  $fib(3)$  returns 2 to the caller.

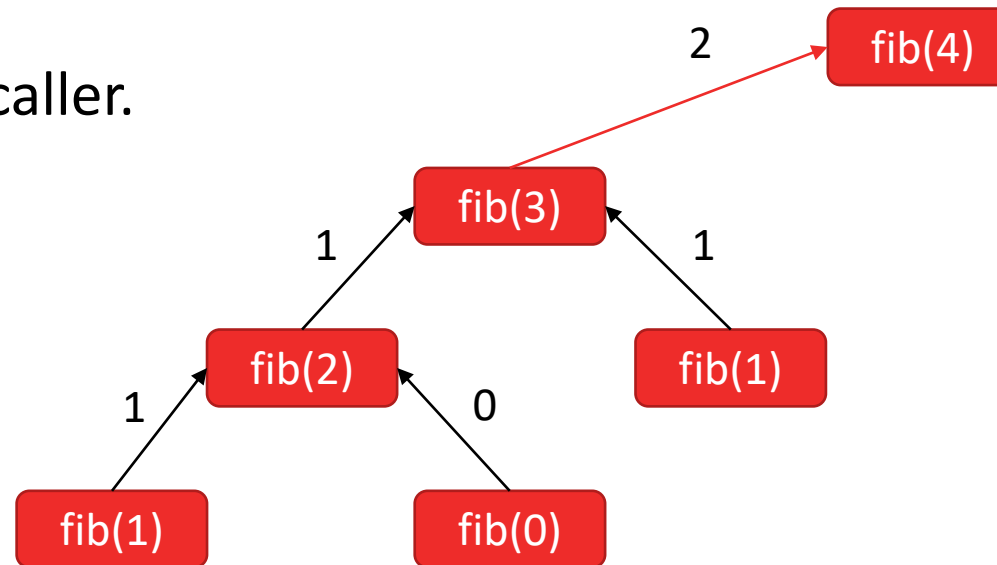


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(3) \rightarrow 1 + 1 = 2$

Now,  $fib(3)$  returns 2 to the caller.

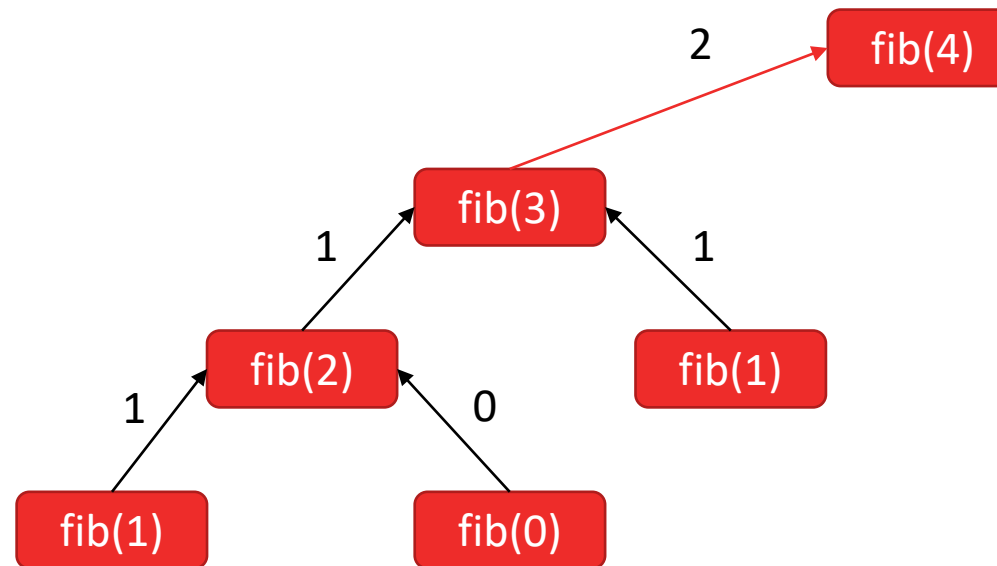


# Example - Fibonacci Numbers - Visualization

---

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(4) \rightarrow 3 + fib(2)$

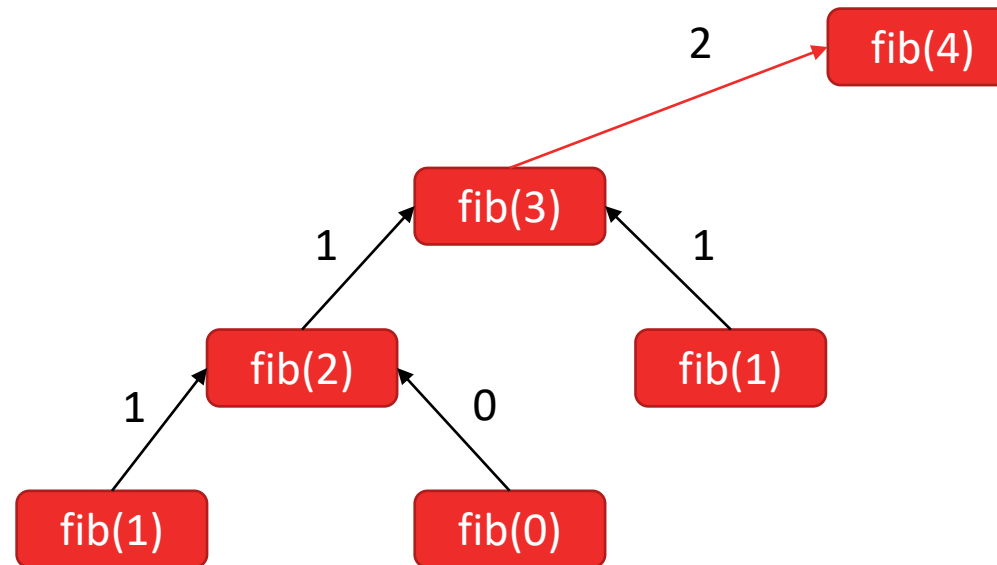


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(4) \rightarrow 3 + \mathbf{fib(2)}$

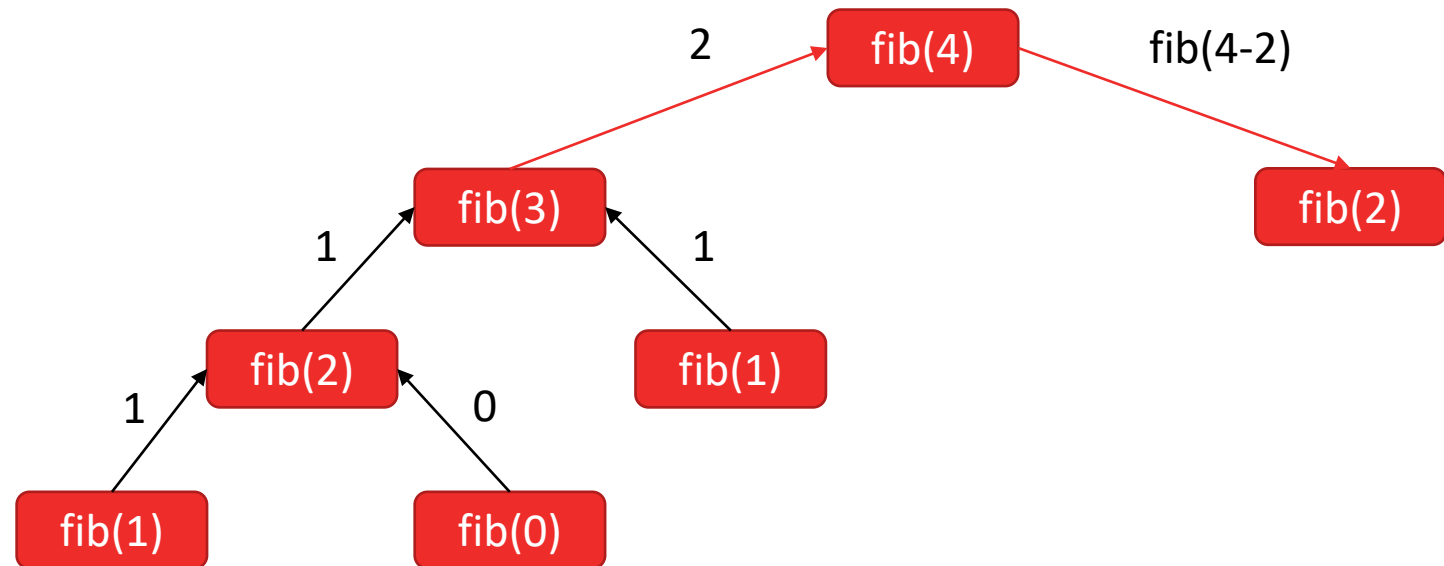
$fib(2)$  is calculated.



# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

fib(2)

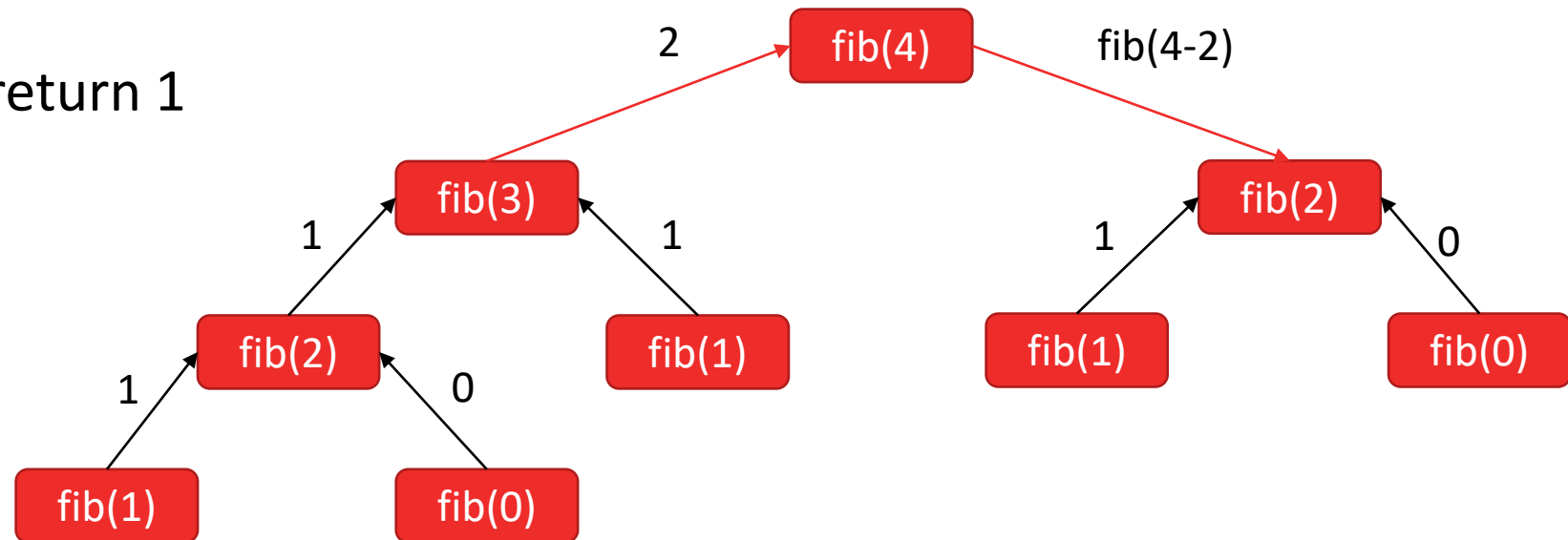


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

$fib(2) \rightarrow fib(1) + fib(0)$

Similar to before,  $fib(2)$  will return 1



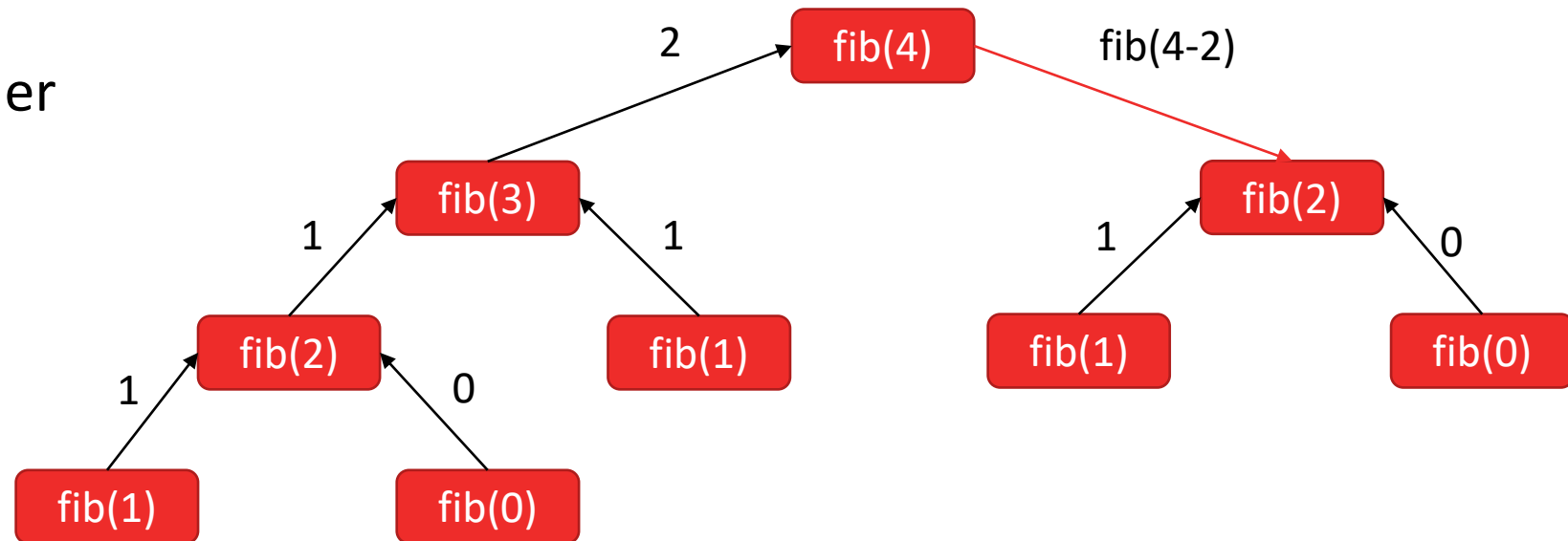


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

fib(2) → 1

fib(2) will return 1 to the caller

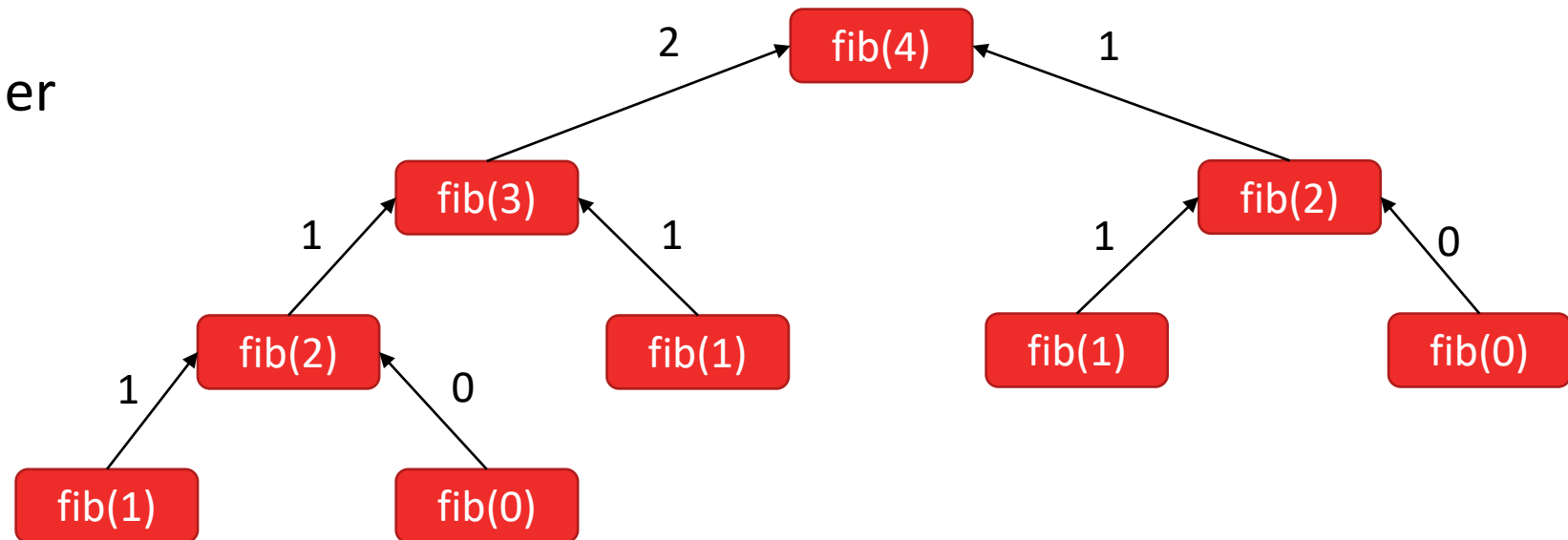


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

fib(2) → 1

fib(2) will return 1 to the caller

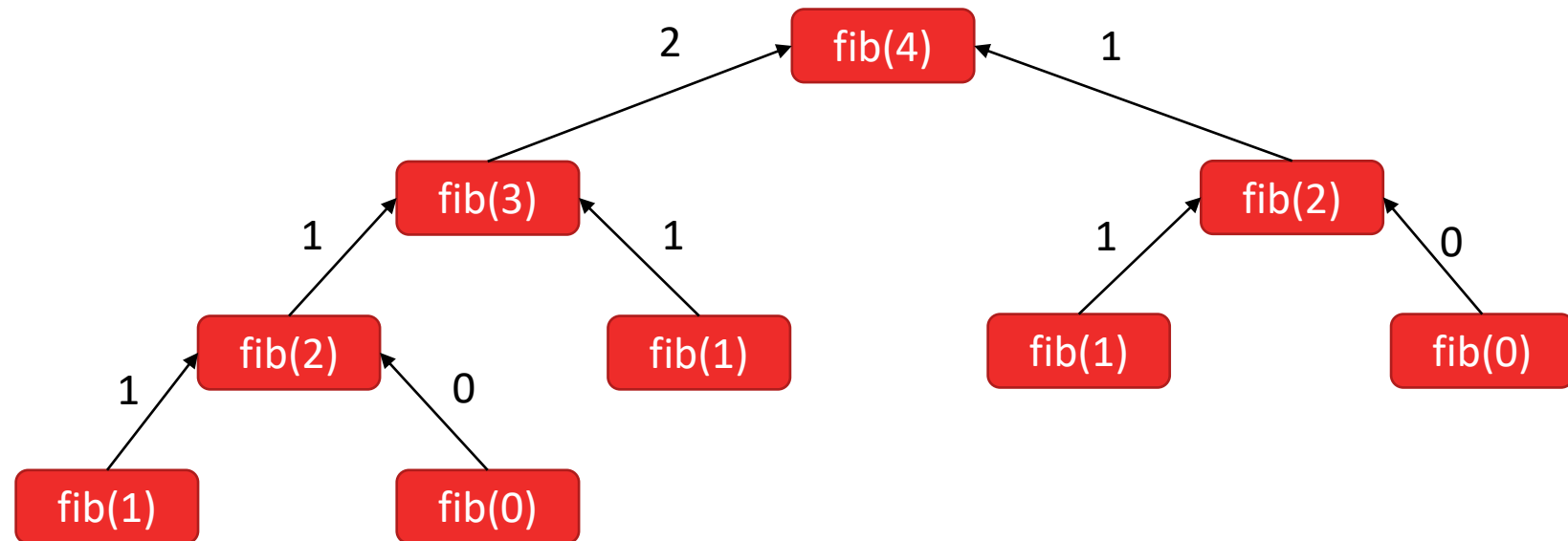


# Example - Fibonacci Numbers - Visualization

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

$fib(4) \rightarrow 2 + 1 = 3$

Answer is 3.



# Advantages

---

# Advantages of Recursion

---

- Very well suited to some problems
  - Tree traversals
  - Flood fill
  - Fractal images
  - Quick sort / merge sort
  - ...
- Often easier to implement, sometimes faster than iterative

# Notice Anything About Fibonacci Numbers

---

- ?

# Notice Anything About Fibonacci Numbers

---

- We called fib(1), fib(0) many many many times
- Why not store this information once

# Dynamic Programming

---



# Dynamic Programming

---

n=	0	1	2	3	4	5	6	7	...
fib(n)	0	1	1	2	3	5	8	13	

```
def fibDyn(n):  
    fib = [0, 1]  
    for i in range(2, n+1):  
        fib.append(fib[i-2]+fib[i-1])  
    return fib[n]
```

# Advantages of Dynamic Programming

---

- Dynamic programming is recursive in concept
  - but we store solutions we already found!
- We start bottom up instead of top down
  - Generally we find ourselves calculating many solutions at the bottom we find again and again
  - Let's calculate it once and store it
- Generally faster than similar recursive solution
  - We don't have to push information on the stack with recursion calls (we store our own state instead)

# Advantages of Dynamic Programming

---

- Tricky to implement (especially for more complex problems see CPSC 413)
- But not always faster, or guaranteed to take less space than similar recursive solution
  - Sometimes we don't need need to track all this state, recursion lets us dispose of information stored on stack as the function call returns

# Iterative Programming

---

# Iterative Programming

---

- Sometimes we can notice in dynamic programming that we didn't need to store all the previous function calls
- Or we can noticed that each function call was base iteratively on the previous function calls

# Iterative Programming

---

```
def fibIter(n):  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    prev1 = 1  
    prev2 = 0  
    result = None  
    for i in range(2, n+1):  
        result = prev1 + prev2  
        prev2 = prev1  
        prev1 = result  
    return result
```

# Advantages of Iteration

---

- Typically
  - Faster (but not always)
  - Requires less memory (most of the time)
- But some problems are messy to express iteratively

# Fractals

---



# Fractals

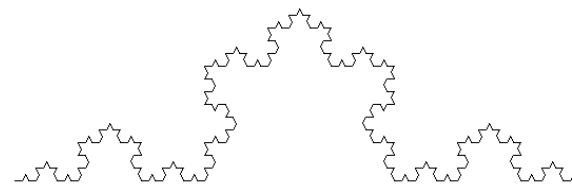
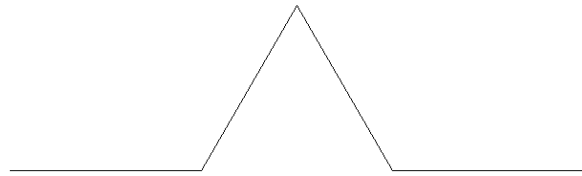
---

- Self similar images
- Often have reasonably simple recursive definitions

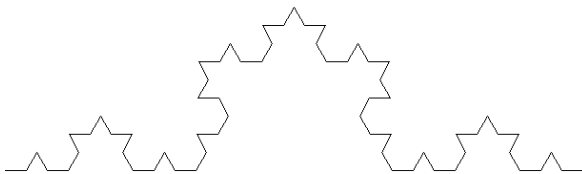
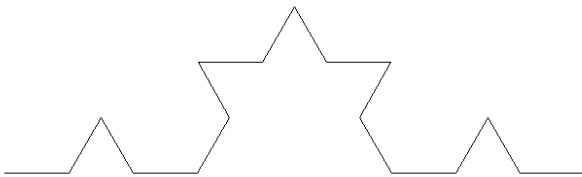
• 1

4

• 2

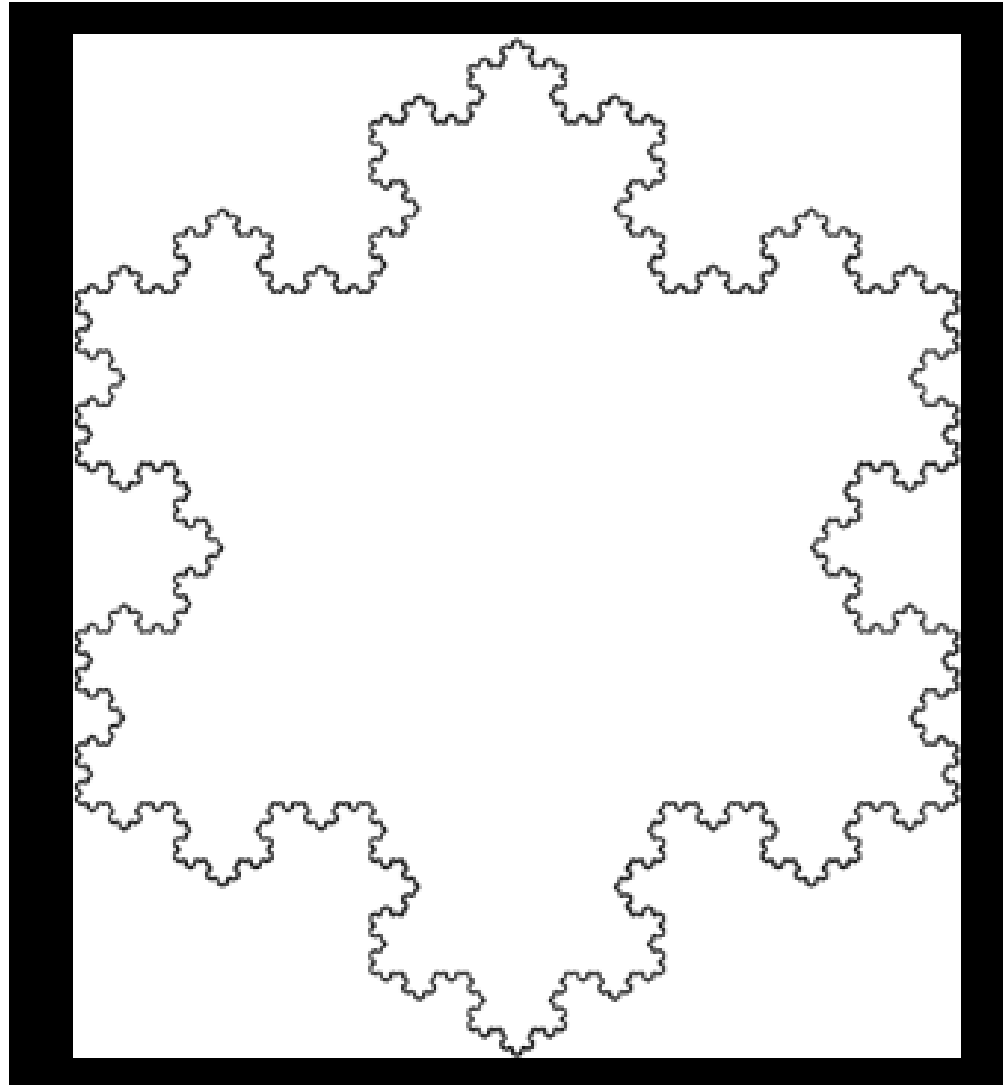


• 3



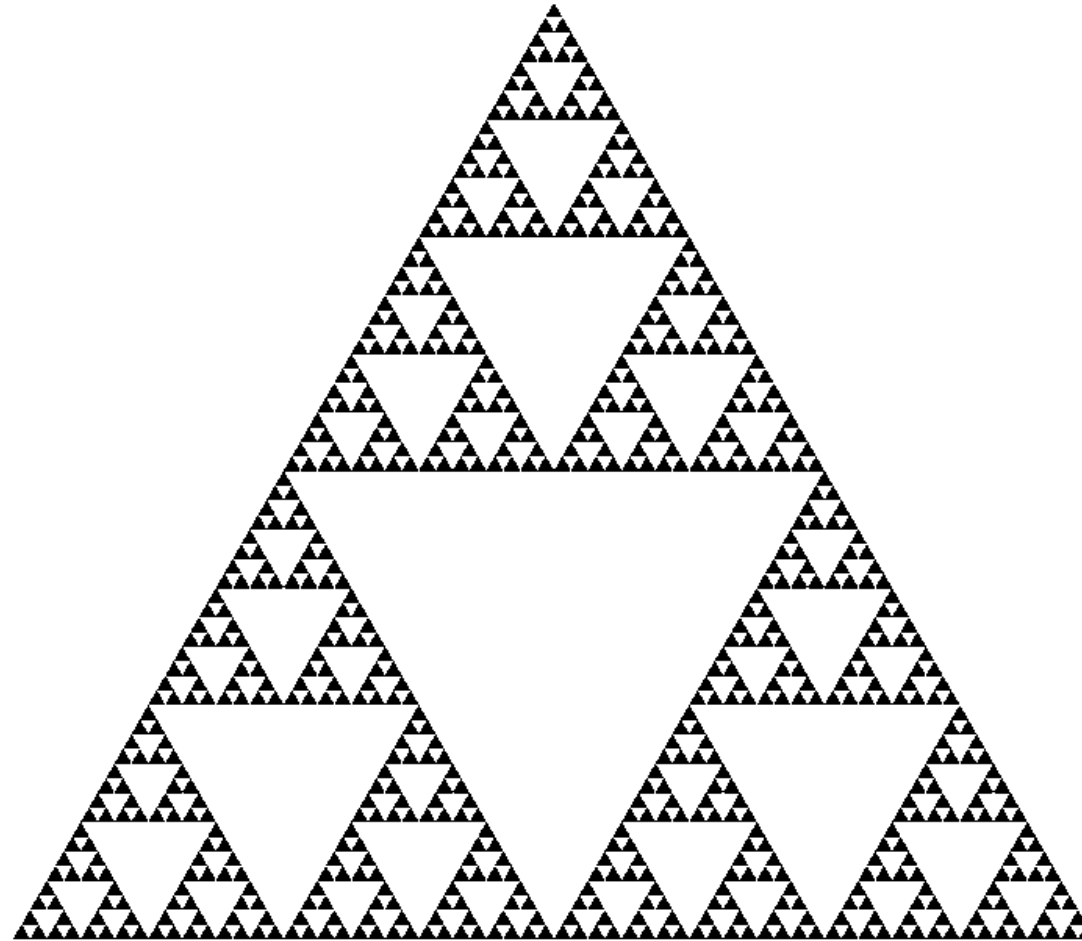
# Koch Snowflake

---



# Sierpinski Triangle

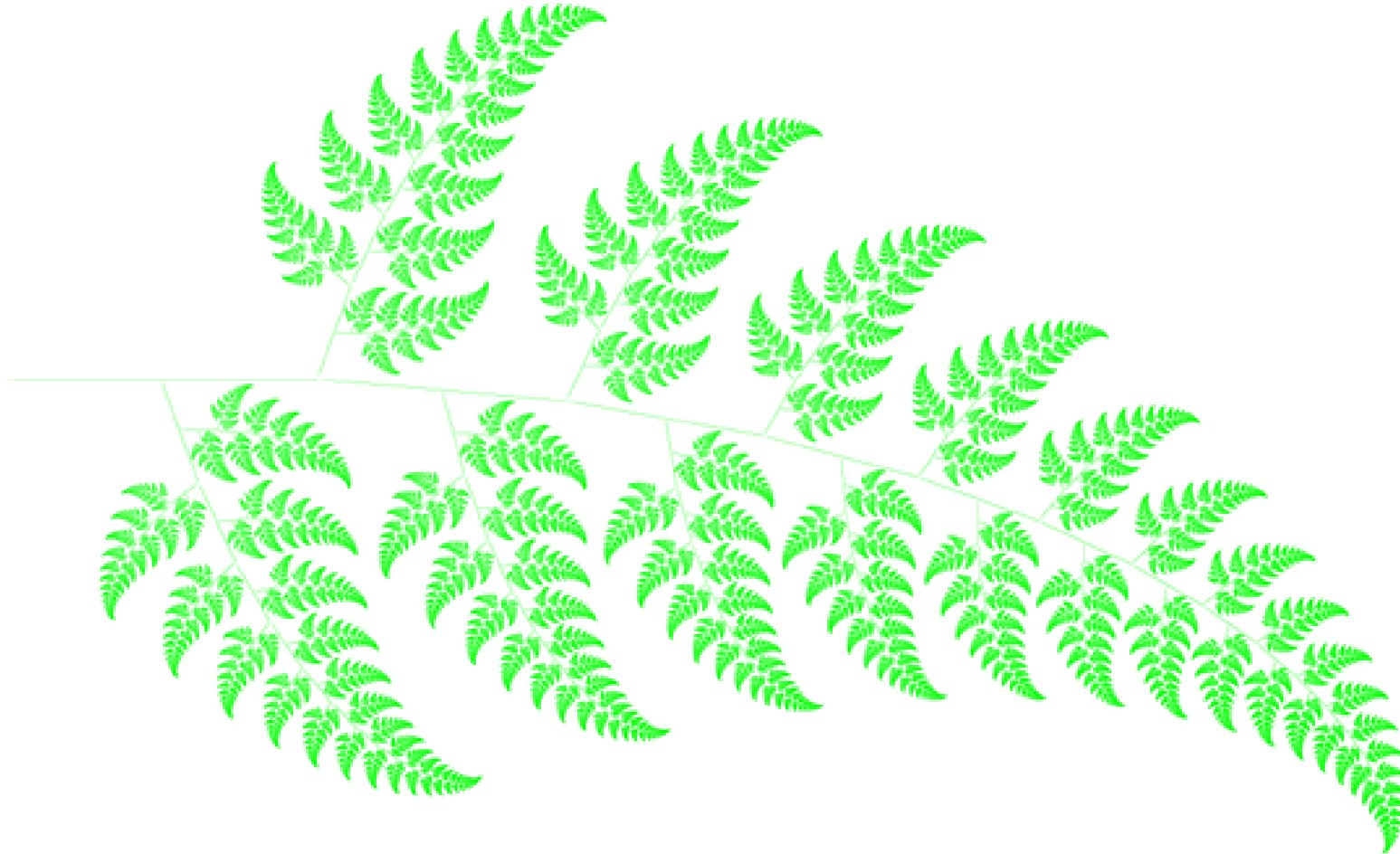
---



Sierpinski Trianglge  
Source: <http://commons.wikimedia.org/wiki/File:Sierpinski-Trigon-7.svg>  
License: Public Domain

# Fractal Fern

---



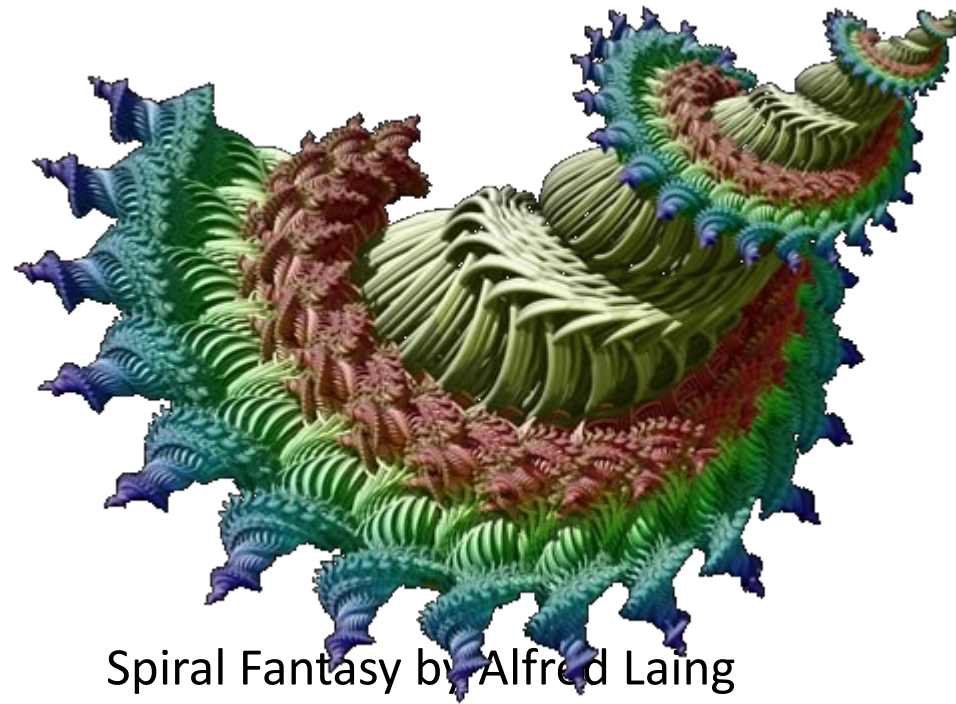
Fractal Fern

Source: <http://schools-wikipedia.org/images/67/6740.png.htm>

License: Creative Commons Attribution-Share Alike 3.0 Unported <http://creativecommons.org/licenses/by-sa/3.0/deed.en>

# Fractal Art

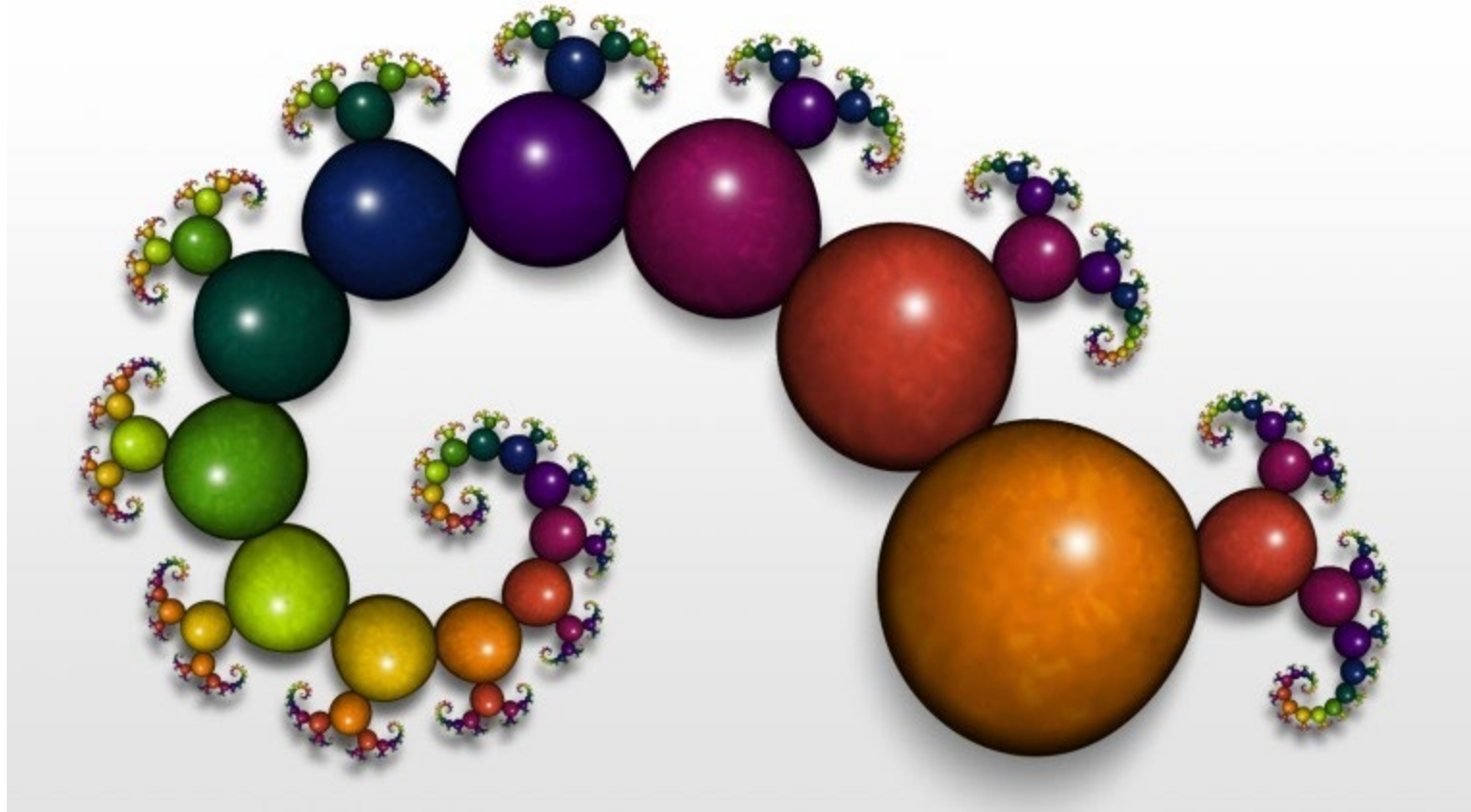
---



Spiral Fantasy by Alfred Laing

# Fractal Art

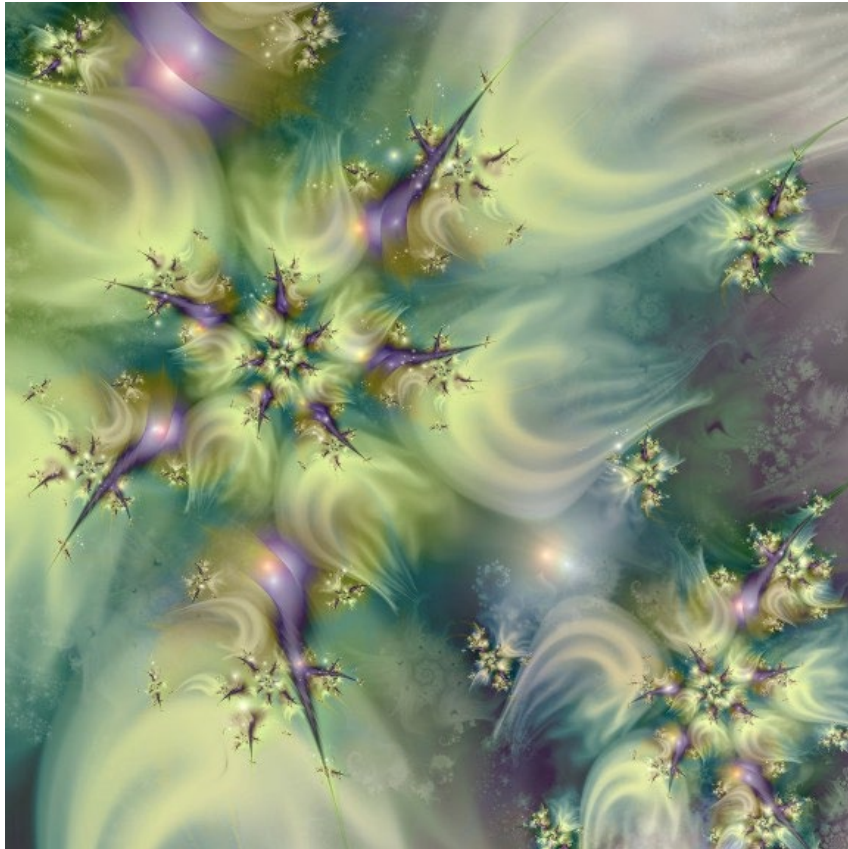
---





# Fractal Art

---



Rose by Keith Mack



# Mazes

---



# Maze Path Finding

---

- Consider a two dimensional list containing 4 different values
  - Entrance for the maze
  - Exit for the maze
  - Open spaces
  - Walls
- Assume that the maze is fully enclosed

# Maze Path Finding

---

- Algorithm solve(map, x, y)
  - If the current square is a wall or a space we have already visited, return failure
  - If the current square is the exit point, mark it as part of the solution and return success
  - Mark the current square as part of the solution
  - If solve(map, x, y+1) is successful, return success
  - If solve(map, x, y-1) is successful, return success
  - If solve(map, x+1, y) is successful, return success
  - If solve(map, x-1, y) is successful, return success
  - Mark the current square as visited but not part of the solution
  - Return failure

# Maze Path Finding

---

```
def solve(map, x, y):  
    if map[x][y] == "wall" or map[x][y] == "visited":  
        return False  
  
    if map[x][y] == "exit":  
        print "Go to (%d,%d)" % (x, y)  
        return True
```

# Maze Path Finding

---

```
print "Go to (%d,%d)" % (x,y)
if solve(map,x,y+1):
    return True
elif solve(map,x,y-1):
    return True
elif solve(map,x+1,y):
    return True
elif solve(map,x-1,y):
    return True
else:
    map[x][y] = "visited"
    return False
```

# Recursion – Why?

---

# Why recursion

---

- Recursive implementation is the perfect fit for naturally recursive problems.
- The result of one step depends on the previous steps.
- There is a base case where the process will stop at.
- e.g., search, games, various math problems
- Computationally elegant
- Easy to program than using loops
- Tradeoffs:
  - Function calls can be costly in terms of memory and time

# Induction

---

- Recursive functions performance and even their results are often shown via proofs using induction
  - Base Case: Where recursion bottoms out and returns
  - Inductive Case: Where recursion calls another functions call
- Induction is generally be taught in CPSC 251, 331, and 413

# Example

---

```
def binary_search(t_list, k):
    if len(t_list) != 0:
        center = len(t_list)//2
        if t_list[center] == k:
            return True
        if t_list[center] > k:
            return binary_search(t_list[:center], k)
        else:
            return binary_search(t_list[center+1:], k)
    return False
```



# Onward to ... Flex Topics.

---

Jonathan Hudson  
[jwhudson@ucalgary.ca](mailto:jwhudson@ucalgary.ca)  
<https://cspages.ucalgary.ca/~jwhudson/>



# Recursion – Beyond Algorithms

---

# Not all about code

---

- We can also define structures that are recursive
- Most common examples are tree structures

```
class Node:
```

```
    def __init__(self):  
        self.children = [] #list of Node
```

```
root = Node()
```

```
root.children.append(Node())
```

```
child = root.children[0]
```

```
child.children.append(Node())
```

# Binary Tree

---

- We can also define structures that are recursive
- Most common example is a binary tree

class Node:

```
    def __init__(self, value):  
        self.value = None  
        self.left = None    #Node  
        self.right = None   #Node
```

# Binary Search Tree

---

- We can also define structures that are recursive
- Most common example is a binary tree

class Node:

```
def __init__(self, value):  
    self.value = None  
    self.left = None    #Node with value less than self value  
    self.right = None   #Node with value greater than self value
```

# Heap

---

- We can also define structures that are recursive

# Max Heap

class Node:

```
    def __init__(self, value):
```

```
        self.value = None
```

```
        self.left = None    #Node with value less than self value
```

```
        self.right = None   #Node with value less than self value
```