

# CSE 250

## Data Structures

Dr. Eric Mikida  
epmikida@buffalo.edu

Dr. Oliver Kennedy  
okennedy@buffalo.edu

212 Capen Hall

**Day 11**  
**Recursion**  
**Textbook Ch 15**

# Recursion

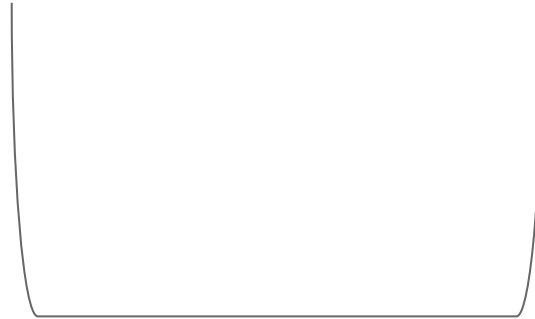


# Factorial

$$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 2 * 1$$

# Factorial

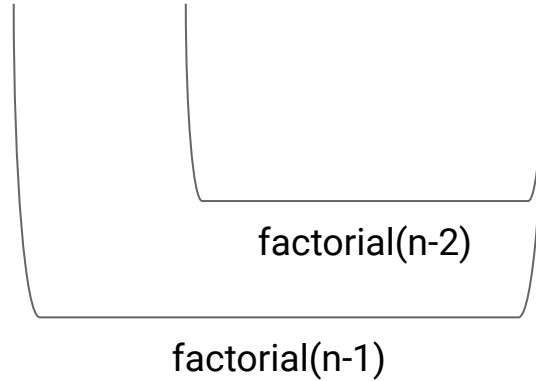
$$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 2 * 1$$



$\text{factorial}(n-1)$

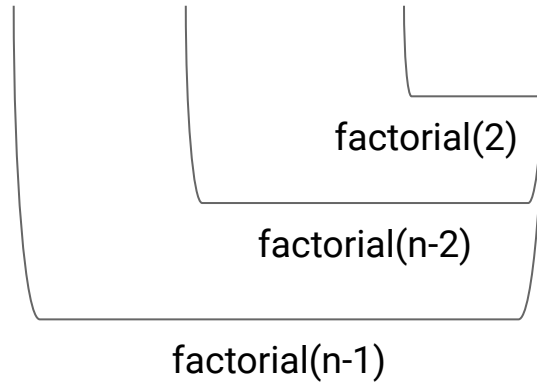
# Factorial

$$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 2 * 1$$

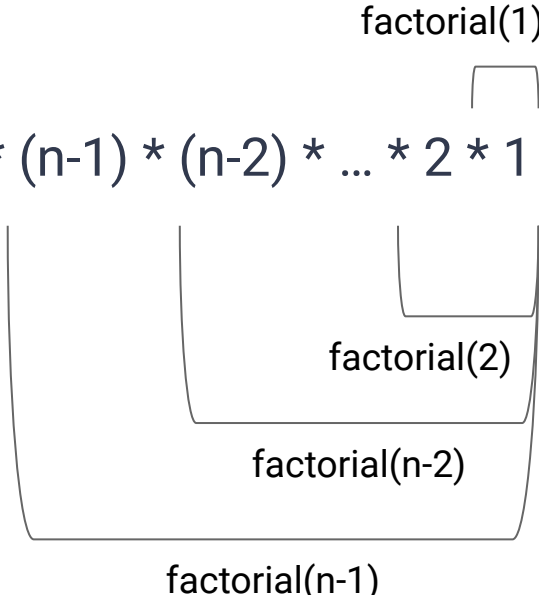


# Factorial

$$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 2 * 1$$



# Factorial

$$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 2 * 1$$


The diagram illustrates the recursive nature of the factorial function. It features several nested brackets below the equation. The outermost bracket spans from the first term 'n' to the last term '1' and is labeled 'factorial(n-1)' underneath. Inside this, a bracket spans from '(n-2)' to '1' and is labeled 'factorial(n-2)'. Further in, a bracket spans from '(n-1)' to '1' and is labeled 'factorial(2)'. Finally, a small bracket is positioned above the '1' and is labeled 'factorial(1)'.

# Factorial

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 }  
  else { n * factorial(n - 1) }
```



# Factorial

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 }           ← Base Case  
  else { n * factorial(n - 1) }
```

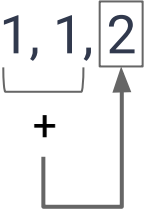
# Factorial

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 } ← Base Case  
  else { n * factorial(n - 1) } ← Recursive Case
```

# Fibonacci

$$\text{fibb}(n) = 1, 1$$

# Fibonacci

$$\text{fibb}(n) = 1, 1, \boxed{2}$$


# Fibonacci

$$\text{fibb}(n) = 1, 1, 2, \boxed{3}$$

The diagram illustrates the calculation of the 4th Fibonacci number. A horizontal line is drawn under the first two numbers, '1' and '1'. From the right end of this line, a vertical line goes down to a plus sign '+'. From the plus sign, a horizontal line goes right, and then a vertical line goes up to an arrow that points to the number '3', which is enclosed in a box. This shows that the 4th number is the sum of the two preceding numbers.

# Fibonacci

$$\text{fibb}(n) = 1, 1, 2, 3, \boxed{5}$$

The diagram illustrates the calculation of the 5th Fibonacci number. A bracket is drawn under the numbers 2 and 3, with a plus sign (+) centered below it. An arrow starts from the right side of the bracket and points upwards to the number 5, which is enclosed in a box. This indicates that 5 is the sum of 2 and 3.

# Fibonacci

$\text{fibb}(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$

# Fibonacci

$\text{fibb}(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$

$$\text{fibb}(n) = \text{fibb}(n-1) + \text{fibb}(n-2)$$



# Fibonacci

```
def fibb(n: Int): Long =  
  if(n < 2){ 1 }  
  else { fibb(n-1) + fibb(n-2) }
```

# Fibonacci

```
def fibb(n: Int): Long =  
  if(n < 2){ 1 } ← Base Case  
  else { fibb(n-1) + fibb(n-2) }
```

# Fibonacci

```
def fibb(n: Int): Long =  
  if(n < 2){ 1 }           ← Base Case  
  else { fibb(n-1) + fibb(n-2) } ← Recursive Case
```

# Towers of Hanoi

*Live demo!*

# Towers of Hanoi

```
var towers = Array(new Stack(), new Stack(), new Stack())

def moveFrom(fromTower: Int, toTower: Int, numDisks: Int): Unit
= {
  val otherTower = (Set(0, 1, 2) - fromTower - toTower).head

  if(numDisks < 0){
    return
  } else if(numDisks == 1) {
    moveTopDisk(from = fromTower, to = toTower)
  } else {
    moveFrom(fromTower, otherTower, numDisks-1)
    moveTopDisk(from = fromTower, to = toTower)
    moveFrom(otherTower, toTower, numDisks-1)
  }
}
```

# Towers of Hanoi

```
var towers = Array(new Stack(), new Stack(), new Stack())

def moveFrom(fromTower: Int, toTower: Int, numDisks: Int): Unit
= {
  val otherTower = (Set(0, 1, 2) - fromTower - toTower).head

  if(numDisks < 0){
    return
  } else if(numDisks == 1) {    ← Base Case
    moveTopDisk(from = fromTower, to = toTower)
  } else {
    moveFrom(fromTower, otherTower, numDisks-1)
    moveTopDisk(from = fromTower, to = toTower)
    moveFrom(otherTower, toTower, numDisks-1)
  }
}
```

# Towers of Hanoi

```
var towers = Array(new Stack(), new Stack(), new Stack())

def moveFrom(fromTower: Int, toTower: Int, numDisks: Int): Unit
= {
  val otherTower = (Set(0, 1, 2) - fromTower - toTower).head

  if(numDisks < 0){
    return
  } else if(numDisks == 1) {← Base Case
    moveTopDisk(from = fromTower, to = toTower)
  } else {
    ← Recursive Case
    moveFrom(fromTower, otherTower, numDisks-1)
    moveTopDisk(from = fromTower, to = toTower)
    moveFrom(otherTower, toTower, numDisks-1)
  }
}
```

# But What is the Complexity?

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 }  
  else { n * factorial(n - 1) }
```



# But What is the Complexity?

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 } ←  $\Theta(1)$   
  else { n * factorial(n - 1) }
```

# But What is the Complexity?

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 } ←  $\Theta(1)$   
  else { n * factorial(n - 1) } ←  $\Theta(1) + ???$ 
```

# But What is the Complexity?

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 } ←  $\Theta(1)$   
  else { n * factorial(n - 1) } ←  $\Theta(1) + ???$ 
```

*How do we figure out complexity of a function, when part of the runtime of the function is calling itself?*

# But What is the Complexity?

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 } ←  $\Theta(1)$   
  else { n * factorial(n - 1) } ←  $\Theta(1) + ???$ 
```

*How do we figure out complexity of a function, when part of the runtime of the function is calling itself?*

*To know the complexity of `factorial`, we need to...know the complexity of `factorial`?*

# Complexity of factorial

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(n-1) + \Theta(1) & \text{otherwise} \end{cases}$$

Solve for  $T(n)$

# Complexity of factorial

Solve for  $T(n)$

## **Approach:**

1. Generate a hypothesis
2. Prove your hypothesis for the base case
3. Prove the hypothesis for the recursive case *inductively*

# Step 1 - Generate a Hypothesis

Let's start by looking at the runtime for increasing values of  $n$

$$\Theta(1)$$

# Step 1 - Generate a Hypothesis

Let's start by looking at the runtime for increasing values of  $n$

$$\Theta(1), 2\Theta(1)$$



# Step 1 - Generate a Hypothesis

Let's start by looking at the runtime for increasing values of  $n$

$\Theta(1)$ ,  $2\Theta(1)$ ,  $3\Theta(1)$

# Step 1 - Generate a Hypothesis

Let's start by looking at the runtime for increasing values of  $n$

$\Theta(1)$ ,  $2\Theta(1)$ ,  $3\Theta(1)$ ,  $4\Theta(1)$ ,  $5\Theta(1)$ ,  $6\Theta(1)$ ,  $7\Theta(1)$

# Step 1 - Generate a Hypothesis

Let's start by looking at the runtime for increasing values of  $n$

$\Theta(1)$ ,  $2\Theta(1)$ ,  $3\Theta(1)$ ,  $4\Theta(1)$ ,  $5\Theta(1)$ ,  $6\Theta(1)$ ,  $7\Theta(1)$

What is the pattern?

# Step 1 - Generate a Hypothesis

Let's start by looking at the runtime for increasing values of  $n$

$\Theta(1)$ ,  $2\Theta(1)$ ,  $3\Theta(1)$ ,  $4\Theta(1)$ ,  $5\Theta(1)$ ,  $6\Theta(1)$ ,  $7\Theta(1)$

What is the pattern?

**Hypothesis:**  $T(n) \in O(n)$

(there is some  $c > 0$  such that  $T(n) \leq c \cdot n$ )

# Prove for the Base Case

First, lets make our constants explicit

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 1 \\ T(n - 1) + c_1 & \text{otherwise} \end{cases}$$

# Prove $T(n) \in O(n)$ for the Base Case

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case:**  $n = 1$

$$T(1) \leq c \cdot 1$$

# Prove $T(n) \in O(n)$ for the Base Case

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case:**  $n = 1$

$$T(1) \leq c \cdot 1$$

$$T(1) \leq c$$

# Prove $T(n) \in O(n)$ for the Base Case

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case:**  $n = 1$

$$T(1) \leq c \cdot 1$$

$$T(1) \leq c$$

$$c_0 \leq c$$



# Prove $T(n) \in O(n)$ for the Base Case

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case:**  $n = 1$

$$T(1) \leq c \cdot 1$$

$$T(1) \leq c$$

$$c_0 \leq c$$

True for any  $c \geq c_0$

# Prove $T(n) \in O(n)$ for the Base Case + 1

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 1:  $n = 2$**

$$T(2) \leq c \cdot 2$$

# Prove $T(n) \in O(n)$ for the Base Case + 1

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 1:  $n = 2$**

$$T(2) \leq c \cdot 2$$

$$T(1) + c_1 \leq 2c$$

# Prove $T(n) \in O(n)$ for the Base Case + 1

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 1:  $n = 2$**

$$T(2) \leq c \cdot 2$$

$$T(1) + c_1 \leq 2c$$

$$c_0 + c_1 \leq 2c$$

# Prove $T(n) \in O(n)$ for the Base Case + 1

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 1:  $n = 2$**

$$T(2) \leq c \cdot 2$$

$$T(1) + c_1 \leq 2c$$

$$c_0 + c_1 \leq 2c$$

We already know there's a  $c \geq c_0$ , so...

# Prove $T(n) \in O(n)$ for the Base Case + 1

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 1:  $n = 2$**

$$T(2) \leq c \cdot 2$$

$$T(1) + c_1 \leq 2c$$

$$c_0 + c_1 \leq 2c$$

We already know there's a  $c \geq c_1$ , so...

True for any  $c \geq c_1$

# Prove $T(n) \in O(n)$ for the Base Case + 2

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 2:  $n = 3$**

$$T(3) \leq c \cdot 3$$

# Prove $T(n) \in O(n)$ for the Base Case + 2

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 2:  $n = 3$**

$$T(3) \leq c \cdot 3$$

$$T(2) + c_1 \leq 3c$$



# Prove $T(n) \in O(n)$ for the Base Case + 2

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 2:  $n = 3$**

$$T(3) \leq c \cdot 3$$

$$T(2) + c_1 \leq 3c$$

We know there's a  $c$  s.t.  $T(2) \leq 2c$ ,

# Prove $T(n) \in O(n)$ for the Base Case + 2

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 2:  $n = 3$**

$$T(3) \leq c \cdot 3$$

$$T(2) + c_1 \leq 3c$$

We know there's a  $c$  s.t.  $T(2) \leq 2c$ ,

So if we show that  $2c + c_1 \leq 3c$ , then  $T(2) + c_1 \leq 2c + c_1 \leq 3c$

# Prove $T(n) \in O(n)$ for the Base Case + 2

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 2:  $n = 3$**

$$T(3) \leq c \cdot 3$$

$$T(2) + c_1 \leq 3c$$

We know there's a  $c$  s.t.  $T(2) \leq 2c$ ,

So if we show that  $2c + c_1 \leq 3c$ , then  $T(2) + c_1 \leq 2c + c_1 \leq 3c$

True for any  $c \geq c_1$

# Prove $T(n) \in O(n)$ for the Base Case + 3

Prove:  $T(n) \in O(n)$  (ie: there exists a constant,  $c$ , such that  $T(n) \leq c \cdot n$ )

**Base Case + 2:**  $n = 4$

$$T(4) \leq c \cdot 4$$

$$T(3) + c_1 \leq 4c$$

We know there's a  $c$  s.t.  $T(3) \leq 3c$ ,

So if we show that  $3c + c_1 \leq 4c$ , then  $T(3) + c_1 \leq 3c + c_1 \leq 4c$

True for any  $c \geq c_1$

# Proving the Hypothesis Inductively

*We're starting to see a pattern...*

# Proving the Hypothesis Inductively

**Approach:** Assume our hypothesis is true for any  $n' < n$ ;  
Now prove it must also hold true for  $n$ .

# Proving the Hypothesis Inductively

**Assume:** There is a  $c > 0$  s.t.  $T(n - 1) \leq c \cdot (n - 1)$

**Prove:** There is a  $c > 0$  s.t.  $T(n) \leq c \cdot n$

$$T(n) \leq c \cdot n$$

# Proving the Hypothesis Inductively

**Assume:** There is a  $c > 0$  s.t.  $T(n - 1) \leq c \cdot (n - 1)$

**Prove:** There is a  $c > 0$  s.t.  $T(n) \leq c \cdot n$

$$T(n) \leq c \cdot n$$

$$T(n - 1) + c_1 \leq c \cdot n$$



# Proving the Hypothesis Inductively

**Assume:** There is a  $c > 0$  s.t.  $T(n - 1) \leq c \cdot (n - 1)$

**Prove:** There is a  $c > 0$  s.t.  $T(n) \leq c \cdot n$

$$T(n) \leq c \cdot n$$

$$T(n - 1) + c_1 \leq c \cdot n$$

By the inductive assumption, there is a  $c$  s.t.  $T(n - 1) \leq (n - 1)c$

# Proving the Hypothesis Inductively

**Assume:** There is a  $c > 0$  s.t.  $T(n - 1) \leq c \cdot (n - 1)$

**Prove:** There is a  $c > 0$  s.t.  $T(n) \leq c \cdot n$

$$T(n) \leq c \cdot n$$

$$T(n - 1) + c_1 \leq c \cdot n$$

By the inductive assumption, there is a  $c$  s.t.  $T(n - 1) \leq (n - 1)c$

So if we show that  $(n - 1)c + c_1 \leq nc$ , then...

# Proving the Hypothesis Inductively

**Assume:** There is a  $c > 0$  s.t.  $T(n - 1) \leq c \cdot (n - 1)$

**Prove:** There is a  $c > 0$  s.t.  $T(n) \leq c \cdot n$

$$T(n) \leq c \cdot n$$

$$T(n - 1) + c_1 \leq c \cdot n$$

By the inductive assumption, there is a  $c$  s.t.  $T(n - 1) \leq (n - 1)c$

So if we show that  $(n - 1)c + c_1 \leq nc$ , then...

$$T(n - 1) + c_1 \leq (n - 1)c + c_1 \leq nc$$

# Proving the Hypothesis Inductively

**Assume:** There is a  $c > 0$  s.t.  $T(n - 1) \leq c \cdot (n - 1)$

**Prove:** There is a  $c > 0$  s.t.  $T(n) \leq c \cdot n$

$$T(n) \leq c \cdot n$$

$$T(n - 1) + c_1 \leq c \cdot n$$

By the inductive assumption, there is a  $c$  s.t.  $T(n - 1) \leq (n - 1)c$

So if we show that  $(n - 1)c + c_1 \leq nc$ , then...

$$T(n - 1) + c_1 \leq (n - 1)c + c_1 \leq nc$$

True for any  $c \geq c_1$

# How much space is used?

`factorial (n)`

# How much space is used?

<code>factorial (n-1)</code>
<code>factorial (n)</code>

# How much space is used?

<code>factorial (n-2)</code>
<code>factorial (n-1)</code>
<code>factorial (n)</code>

# How much space is used?

<code>factorial (n-3)</code>
<code>factorial (n-2)</code>
<code>factorial (n-1)</code>
<code>factorial (n)</code>



# How much space is used?

•  
•  
•

<code>factorial (n-4)</code>
<code>factorial (n-3)</code>
<code>factorial (n-2)</code>
<code>factorial (n-1)</code>
<code>factorial (n)</code>

# Tail Recursion

If the last thing we do in the function is a recursive call, we shouldn't need to create an entire stack of all the function calls...

```
def factorial(n: Int): Long =  
  if(n <= 1){ 1 }  
  else { n * factorial(n - 1) }
```

*...smart compilers can often automatically convert to a loop...*

```
def factorial(n: Int): Long = {  
  var total = 1  
  for(i <- 1 until n){ total *= i }  
  return total  
}
```

# Tail Recursion

The Scala compiler will attempt to turn tail recursion into a loop

If you add `@tailrec` before your function definition, the compiler will yell at you if it cannot do the conversion

# Fibonacci

*What about a function without tail recursion, or with multiple recursive calls?*

What is the complexity of `fibb(n)`?

```
def fibb(n: Int): Long =  
  if(n < 2){ 1 }  
  else { fibb(n-1) + fibb(n-2) }
```

# Next time...

Divide and Conquer

Recursion Trees