

# CSE 250

## Data Structures

Dr. Eric Mikida  
epmikida@buffalo.edu  
208 Capen Hall

**Lec 11: Recursion**

# Announcements

- PA1 Implementation due Sunday, 9/24 @ 11:59PM
  - Continue with the same repo you've been using
  - Special weekend office hours (see Piazza)
- WA2 will be released after the PA1 deadline, due 9/31 @ 11:59PM
- Midterm #1 will be Monday 10/2 in class
  - **Covers:** Summations, Asymptotics, Sequences/Lists, Arrays, Linked Lists, Recursion, Bounds (Tight Upper/Lower, **Unqualified** vs Expected vs **Amortized**)

# Seq Summary So Far

|                    | <b>ArrayList</b>               | <b>Linked List<br/>(by index)</b> | <b>Linked List<br/>(by reference)</b> |
|--------------------|--------------------------------|-----------------------------------|---------------------------------------|
| <b>get(...)</b>    | $\Theta(1)$                    | $\Theta(\text{idx})$ or $O(n)$    | $\Theta(1)$                           |
| <b>set(...)</b>    | $\Theta(1)$                    | $\Theta(\text{idx})$ or $O(n)$    | $\Theta(1)$                           |
| <b>size()</b>      | $\Theta(1)$                    | $\Theta(1)$                       | $\Theta(1)$                           |
| <b>add(...)</b>    | $O(n)$ , Amortized $\Theta(1)$ | $\Theta(\text{idx})$ or $O(n)$    | $\Theta(1)$                           |
| <b>remove(...)</b> | $O(n)$                         | $\Theta(\text{idx})$ or $O(n)$    | $\Theta(1)$                           |

# What Data Structure is Best?

**Scenario #1:** You need to read in the lines of a CSV file, store them in a List, and later be able to access individual records based on index.

# What Data Structure is Best?

**Scenario #1:** You need to read in the lines of a CSV file, store them in a List, and later be able to access individual records based on index.

## ArrayList

Since the amortized runtime of add for **ArrayList** and **LinkedList**, adding the  $n$  lines of the CSV file will take  $O(n)$  time for both...

But **ArrayLists** will then have an advantage because looking up records by index will be  $O(1)$  whereas **LinkedLists** will be  $O(n)$

# What Data Structure is Best?

**Scenario #2:** Users logging onto an online game need to be efficiently added to a List in the order they log on. From time to time you must be able to iterate through the list from beginning to end.

# What Data Structure is Best?

**Scenario #2:** Users logging onto an online game need to be efficiently added to a List in the order they log on. From time to time you must be able to iterate through the list from beginning to end.

## **LinkedList**

The enumeration will cost a total of  $O(n)$  for both types of List

But some users will experience longer waits being added to the List if implemented as an **ArrayList** due to the need for it to occasionally resize

# Recursion



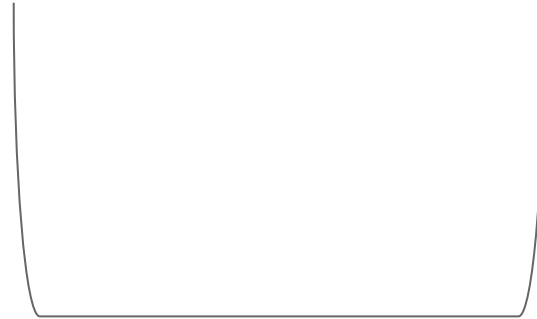


# Factorial

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

# Factorial

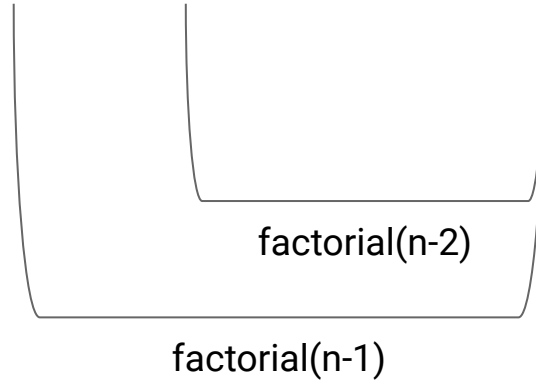
$$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 2 * 1$$



factorial(n-1)

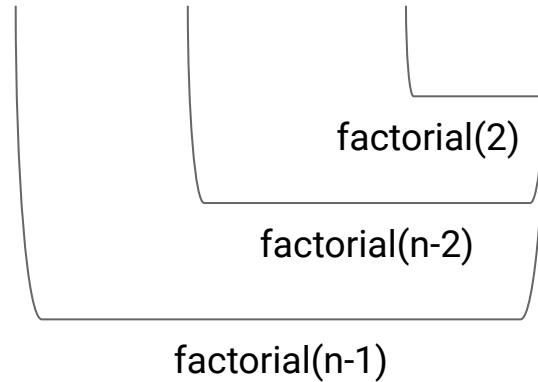
# Factorial

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

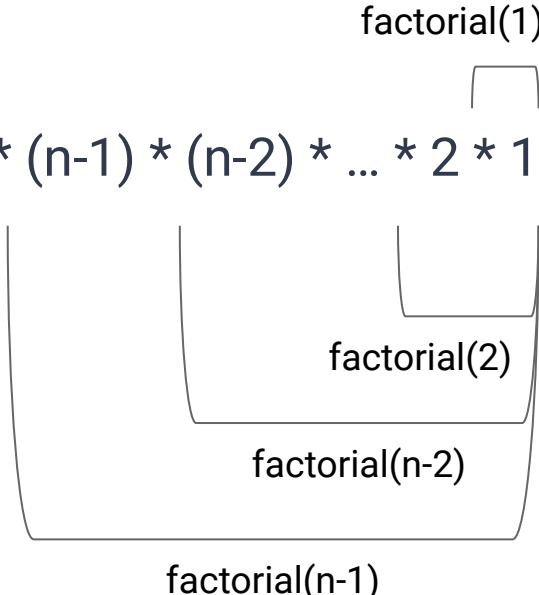


# Factorial

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



# Factorial

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


The diagram illustrates the recursive nature of the factorial function. It shows the expression  $n * (n-1) * (n-2) * \dots * 2 * 1$  with several brackets underneath. A small bracket under the '1' is labeled  $\text{factorial}(1)$ . A larger bracket under the '2 \* 1' is labeled  $\text{factorial}(2)$ . A bracket under the entire sequence from  $(n-2)$  to  $1$  is labeled  $\text{factorial}(n-2)$ . The largest bracket, under the entire sequence from  $(n-1)$  to  $1$ , is labeled  $\text{factorial}(n-1)$ .

# Factorial

```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; }  
3     else { return n * factorial(n - 1); }  
4 }
```

# Factorial

```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; }           ← Base Case  
3     else { return n * factorial(n - 1); }  
4 }
```

# Factorial

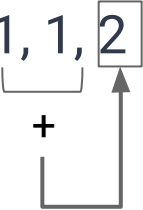
```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; }           ← Base Case  
3     else { return n * factorial(n - 1); } ← Recursive Case  
4 }
```



# Fibonacci

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

# Fibonacci

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


The diagram illustrates the calculation of the third Fibonacci number, fib(3) = 2. It shows the sequence 1, 1, 2. A horizontal bracket is drawn under the first two '1's, with a '+' sign centered below it. A vertical line extends from the right end of this bracket, and a horizontal line extends from the left end of this vertical line, meeting at a point directly below the '2'. From this meeting point, an arrow points vertically upwards to the '2', indicating that the third term is the sum of the two preceding terms.

# Fibonacci

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

The diagram illustrates the calculation of the 4th Fibonacci number. The sequence is shown as 1, 1, 2, 3. A bracket under the first two 1s and a plus sign below it indicate their sum is 2. A bracket under the 1 and 2, and an arrow pointing to the boxed 3, indicate their sum is 3.

# Fibonacci

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

The diagram illustrates the calculation of the 5th Fibonacci number. It shows the sequence 1, 1, 2, 3, 5. A bracket is drawn under the numbers 2 and 3, with a plus sign (+) below it. An arrow points from the bracketed sum up to the number 5, which is enclosed in a box to indicate it is the result of the calculation.

# Fibonacci

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

# Fibonacci

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

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

# Fibonacci

```
1 public int fib(int n) {  
2     if(n < 2) { return 1; }  
3     else { return fib(n-1) + fib(n - 2); }  
4 }
```

# Fibonacci

```
1 public int fib(int n) {  
2     if(n < 2) { return 1; }           ← Base Case  
3     else { return fib(n-1) + fib(n - 2); }  
4 }
```



# Fibonacci

```
1 public int fib(int n) {  
2     if(n < 2) { return 1; }           ← Base Case  
3     else { return fib(n-1) + fib(n - 2); } ← Recursive Case  
4 }
```

# Towers of Hanoi

*Live demo!*

# But What is the Complexity?

```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; }  
3     else { return n * factorial(n - 1); }  
4 }
```

# But What is the Complexity?

```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; }           ←  $\Theta(1)$   
3     else { return n * factorial(n - 1); }  
4 }
```

# But What is the Complexity?

```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; }           ←  $\Theta(1)$   
3     else { return n * factorial(n - 1); } ←  $\Theta(1) + \Theta(???)$   
4 }
```

# But What is the Complexity?

```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; } ←  $\Theta(1)$   
3     else { return n * factorial(n - 1); } ←  $\Theta(1) + \Theta(???)$   
4 }
```

*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  $n$

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

# 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  $c \geq c_0$ , 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  $c \geq c_1$

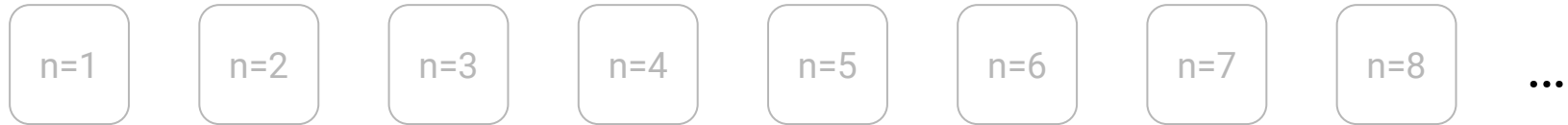


# Proving the Hypothesis Inductively

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

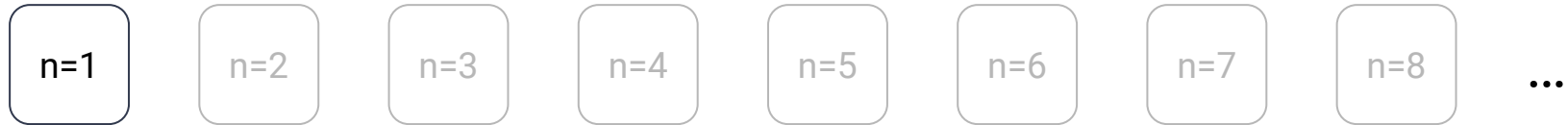
# Proving the Hypothesis Inductively

We can prove our hypothesis for specific values of  $n$ ...



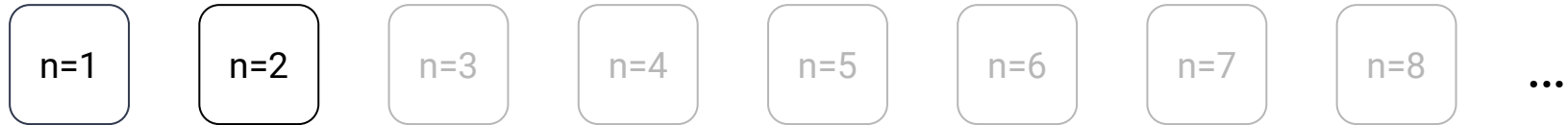
# Proving the Hypothesis Inductively

We can prove our hypothesis for specific values of  $n$ ...



# Proving the Hypothesis Inductively

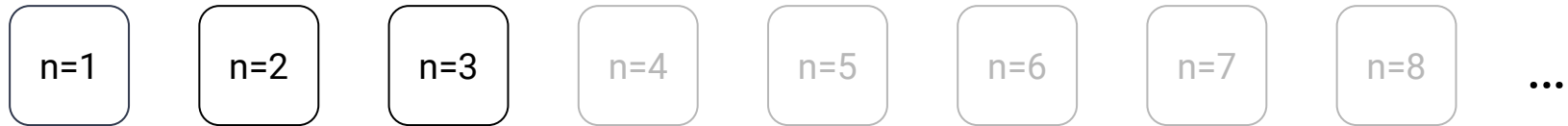
We can prove our hypothesis for specific values of  $n$ ...



# Proving the Hypothesis Inductively

We can prove our hypothesis for specific values of  $n$ ...

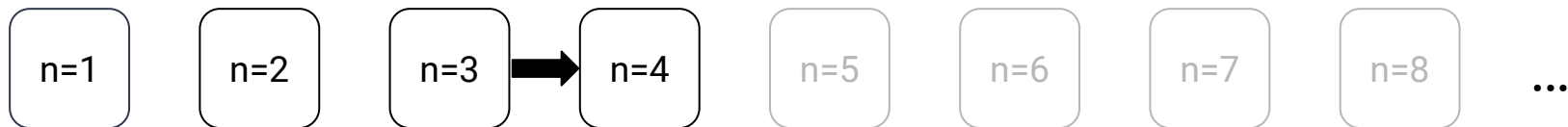
...but there are infinitely many possible values of  $n$



# Proving the Hypothesis Inductively

We can prove our hypothesis for specific values of  $n$ ...

...but there are infinitely many possible values of  $n$



Instead, let's prove that we can derive an unproven case from a proven one!

# 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  $c \geq c_1$

# 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  $c \geq c_1$

**Therefore, we've proven our hypothesis for the Base Case, and inductively for the Recursive Case.  
Therefore, the complexity of factorial is  $\Theta(n)$**

# 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 single recursive call, we shouldn't need to create an entire stack of all the function calls...

```
1 public int factorial(int n) {  
2     if(n <= 1) { return 1; }  
3     else { return n * factorial(n - 1); }  
4 }
```

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

```
1 public int factorial(int n) {  
2     int total = 1;  
3     for (int i = 0; i < n; i++) { total *= i; }  
4     return total;  
5 }
```

# Fibonacci

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

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

```
1 public int fib(int n) {  
2     if(n < 2) { return 1; }  
3     else { return fib(n-1) + fib(n - 2); }  
4 }
```

# Next time...

Divide and Conquer

Recursion Trees