# CSE 250
## Data Structures

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

# Lec 11: Recursion

# Announcements

- PA1 Implementation due Sunday, 9/22 @ 11:59PM
  - Continue with the same repo you've been using
- WA2 will be released after the PA1 deadline, due 9/29 @ 11:59PM

# `List` Summary So Far

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

# Follow-Up Questions

What is the amortized runtime of **`add`** for a **`LinkedList`**?

What is the runtime of **`add(int idx, E elem)`** for an **`ArrayList`**?

# Follow-Up Questions

What is the amortized runtime of `add` for a `LinkedList`?

Each `add` is $O(1)$. Total for $n$ calls is $O(n)$. Amortized is $O(n/n) = O(1)$

What is the runtime of `add(int idx, E elem)` for an `ArrayList`?

To `add` between two elements requires the rest of the elements to be shifted to the right (opposite of `remove`), so runtime is always $O(n)$.

# 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

factorial(n) = n * (n-1) * (n-2) * … * 2 * 1

# Factorial

factorial(n) = n * (n-1) * (n-2) * … * 2 * 1

factorial(n-1)

# Factorial

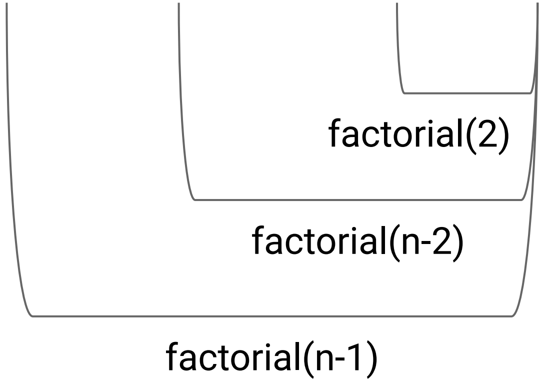
factorial(n) = n * (n-1) * (n-2) * … * 2 * 1

factorial(n-2)

factorial(n-1)

# Factorial

factorial(n) = n * (n-1) * (n-2) * … * 2 * 1



# Factorial

factorial(1)

$$factorial(n) = n * (n-1) * (n-2) * \ldots * 2 * 1$$

factorial(2)

factorial(n-2)

factorial(n-1)

# Factorial

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

# Factorial

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

# Factorial

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

# Fibonacci

fib(n) = 1, 1

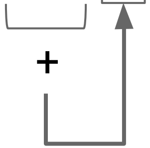
# Fibonacci

fib(n) = 1, 1, 2

+

# Fibonacci

fib(n) = 1, 1, 2, 3



# Fibonacci

fib(n) = 1, 1, 2, 3, 5

+

# Fibonacci

fib(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, …

# Fibonacci

fibb(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, …

fib(n) = fib(n-1) + fib(n-2)

# Fibonacci

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

# Fibonacci

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

# Fibonacci

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

# Towers of Hanoi

*Live demo!*

# But What is the Complexity?

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

# But What is the Complexity?

```
public int factorial(int n) {
    if(n <= 1) { return 1; }                    ← Θ(1)
    else { return n * factorial(n - 1); }
}
```

# But What is the Complexity?

```
public int factorial(int n) {
    if(n <= 1) { return 1; }                   ← Θ(1)
    else { return n * factorial(n - 1); }  ← Θ(1) + Θ(???)
}
```

# But What is the Complexity?

```
public int factorial(int n) {
    if(n <= 1) { return 1; }                    ← Θ(1)
    else { return n * factorial(n - 1); }  ← Θ(1) + Θ(???)
}
```

*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`

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

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

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

# 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

Expand $T(2)$ based on the definition of $T$

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

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

# 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 $T(n) \le c \cdot n$)

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

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

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

$$c_0 + c_1 \le 2c$$

We already know there's a $c \ge c_0$, 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

Expand $T(3)$ based on the definition of $T$

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

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

# 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. $T(2) \leq 2c$…therefore $T(2) + c_1 \leq 2c + c_1$,

# 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 $T(n) \le c \cdot n$)

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

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

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

We know there's a $c$ s.t. $T(2) \le 2c$...therefore $T(2) + c_1 \le 2c + c_1$,

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

# 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. $T(2) \leq 2c$…therefore $T(2) + c_1 \leq 2c + c_1$,

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. $T(3) \leq 3c$...therefore $T(3) + c_1 \leq 3c + c_1$,

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…

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| n=1 | n=2 | n=3 | n=4 | n=5 | n=6 | n=7 | n=8 | … |

# Proving the Hypothesis Inductively

We can prove our hypothesis for specific values of n…

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| n=1 | n=2 | n=3 | n=4 | n=5 | n=6 | n=7 | n=8 | … |

# Proving the Hypothesis Inductively

We can prove our hypothesis for specific values of n…

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| n=1 | n=2 | n=3 | n=4 | n=5 | n=6 | n=7 | n=8 | … |

# Proving the Hypothesis Inductively

We can prove our hypothesis for specific values of n…

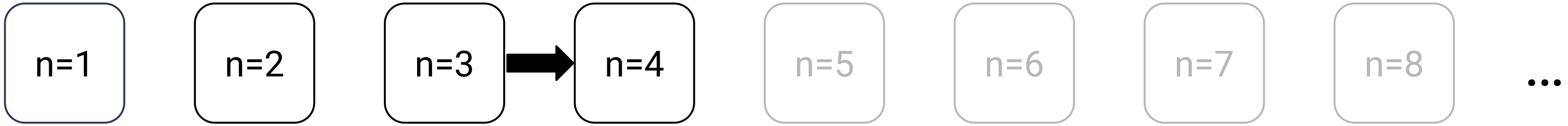
…but there are infinitely many possible values of n

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| n=1 | n=2 | n=3 | n=4 | n=5 | n=6 | n=7 | n=8 | … |

# 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. $T(n - 1) \le c \cdot (n - 1)$

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

Expand $T(n)$ based on the definition of $T$

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

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

# 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?

| |
|---|
| `factorial(n-1)` |
| `factorial(n)` |

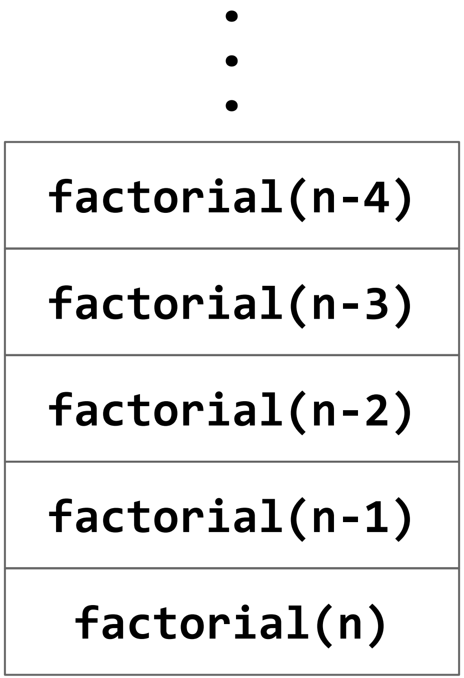
# How much space is used?

| |
|---|
| `factorial(n-2)` |
| `factorial(n-1)` |
| `factorial(n)` |

# How much space is used?

| |
|---|
| `factorial(n-3)` |
| `factorial(n-2)` |
| `factorial(n-1)` |
| `factorial(n)` |

# How much space is used?



# 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…

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

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

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

# Fibonacci

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

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

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

# Next time...

Divide and Conquer

Recursion Trees