Announcements

● WA1 posted. Due Wed 3/1/23.
## Seq Summary So Far

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply(i)</code></td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(i)$, $O(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>update(i, val)</code></td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(i)$, $O(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>insert(i, val)</code></td>
<td>$\Theta(n)$</td>
<td>$O(n)$</td>
<td>$\Theta(i)$, $O(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>remove(i, val)</code></td>
<td>$\Theta(n)$</td>
<td>$\Theta(n-i)$, $O(n)$</td>
<td>$\Theta(i)$, $O(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>append(i)</code></td>
<td>$\Theta(n)$</td>
<td>$O(n)$, Amortized $\Theta(1)$</td>
<td>$\Theta(i)$, $O(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Recursion
Factorial

factorial(n) = n * (n-1) * (n-2) * ... * 2 * 1
factorial(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1

factorial(n) = \text{factorial}(n-1)
factorial(n) = n * (n-1) * (n-2) * ... * 2 * 1
Factorial

\[\text{factorial}(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1\]
Factorial

\[ \text{factorial}(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \]
def factorial(n: Int): Long =
  if(n <= 1){ 1 }
else { n * factorial(n - 1) }
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

fibb(n) = 1, 1
Fibonacci

\[ \text{fibb}(n) = 1, 1, 2 \]
Fibonacci

$fibb(n) = 1, 1, 2, 3$
Fibonacci

\[ \text{fibb}(n) = 1, 1, 2, 3, 5, \ldots \]
Fibonacci

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

\[ \text{fibb}(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \]

\[ \text{fibb}(n) = \text{fibb}(n-1) + \text{fibb}(n-2) \]
def fibb(n: Int): Long =
    if(n < 2){ 1 }
    else { fibb(n-1) + fibb(n-2) }
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!
But What is the Complexity?

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

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 } ← Θ(1)
  else { n * factorial(n - 1) } ← Θ(1) + ⋯
But What is the Complexity?

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

How do we figure out complexity of a function, when part of the runtime of the function is calling itself?
def factorial(n: Int): Long =
    if(n <= 1){ 1 } ← Θ(1)
    else { n * factorial(n - 1) } ← Θ(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 $n$

$\Theta(1), 2\Theta(1), 3\Theta(1), 4\Theta(1), 5\Theta(1), 6\Theta(1), 7\Theta(1)$
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, let's 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: \( T(n) \in O(n) \) (i.e.: 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:** $T(n) \in O(n)$ (i.e., 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:** $T(n) \in O(n)$ (i.e. 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:** $n = 2$

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

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

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 + 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: \( T(n) \in O(n) \) (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: $T(n) \in O(n)$ (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: \( T(n) \in O(n) \) (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. $T(n) \leq c \cdot n$

\[ T(n) \leq c \cdot n \]
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...
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...

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

<table>
<thead>
<tr>
<th>factorial(n-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(n)</td>
</tr>
</tbody>
</table>
How much space is used?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(n-2)</td>
<td></td>
</tr>
<tr>
<td>factorial(n-1)</td>
<td></td>
</tr>
<tr>
<td>factorial(n)</td>
<td></td>
</tr>
</tbody>
</table>
How much space is used?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(n-3)</td>
<td>factorial(n-2)</td>
<td>factorial(n-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>factorial(n)</td>
</tr>
</tbody>
</table>
How much space is used?

\[
\begin{array}{|c|}
\hline
\text{factorial}(n) \\
\text{factorial}(n-1) \\
\text{factorial}(n-2) \\
\text{factorial}(n-3) \\
\text{factorial}(n-4) \\
\vdots \\
\hline
\end{array}
\]
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...

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

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

```scala
def factorial(n: Int): Long = {
  var total = 1l
  for(i <- 1 until n){ total *= i }
  return total
}
```
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.
What about a function without tail recursion, or with multiple recursive calls?

What is the complexity of \texttt{fibb}(n)\texttt{?}

\begin{verbatim}
def fibb(n: Int): Long =
    if (n < 2) { 1 }
    else { fibb(n-1) + fibb(n-2) }
\end{verbatim}
Next time...

Divide and Conquer

Recursion Trees