6.0 Introduction

• The material we study here involves no new syntax
• Instead, we will be analyzing and writing methods, such as we studied in Module 5
• We have seen methods calling other methods
  – Compute the distance between two points
    • We need the sqrt method
  – Multiplication (pedagogical example)
    • \texttt{mpy(c,d)} calls \texttt{add(a,b)}
• Now we examine methods that call themselves
  – Seems impossible to solve problem P using P
  – But it works under the right circumstances
  – And is a very powerful technique
6.0 Introduction

• Why study recursion? Could we live without knowing recursion?
  – Yes: with arrays, iteration, and choice, we know enough to solve any problem that can be solved on a computer.
    • Mechanisms that allow us to write "any program" are called Turing-complete
  – However, recursion is a powerful technique for specifying computation
    • Mathematically satisfying
    • And some computations are more naturally specified using recursion
6.0 Introduction

• Nature offers many examples of recursion
  – A tree grows its branches by applying the same pattern "in the small" as it does "in the large"
  – Shapes manipulated recursively yield interesting results

From Sedgewick's book

http://forums.artistserver.com/messages.cfm?threadid=7D6AF4F8-1143-DBB3-C62D3B1756BD9457
6.0 Introduction

• Nature offers many examples of recursion
• But recursion sometimes seems to do the impossible
6.1 Explicit recursion

- Some formula are *explicitly recursive*
  - $\text{factorial}(n) = n \times \text{factorial}(n-1)$, if $n > 0$
  - $\text{factorial}(n) = 1$, otherwise
6.1 Explicit recursion

• Some formula are *explicitly recursive*
  – $\text{factorial}(n) = n \times \text{factorial}(n-1)$, if $n > 0$
  – $\text{factorial}(n) = 1$, otherwise

• This works because a larger problem
  – $\text{factorial}(n)$
6.1 Explicit recursion

• Some formula are *explicitly recursive*
  – $\text{factorial}(n) = n \times \text{factorial}(n-1)$, if $n > 0$
  – $\text{factorial}(n) = 1$, otherwise

• This works because a larger problem
  – $\text{factorial}(n)$

• is phrased in terms of a smaller problem
  – $\text{factorial}(n-1)$
6.1 Explicit recursion

• Some formula are *explicitly recursive*
  – \( \text{factorial}(n) = n \times \text{factorial}(n-1) \), if \( n > 0 \)
  – \( \text{factorial}(n) = 1 \), otherwise

• This works because a larger problem
  – \( \text{factorial}(n) \)

• is phrased in terms of a smaller problem
  – \( \text{factorial}(n-1) \)

• and because the computation bottoms out with a base case
  – \( \text{factorial}(0) = 1 \)
6.1 Explicit recursion

• Some formula are explicitly recursive
  – factorial(n) = n * factorial(n-1), if n > 0
  – factorial(n) = 1, otherwise

• This is the standard way such formulae are defined in math
6.1 Explicit recursion

• Some formula are *explicitly recursive*
  – factorial(n) = 1, if n <= 0
  – factorial(n) = n * factorial(n-1), otherwise

• Rewrote the definition to put the base case first
  – *Common in computer science*
6.1 Explicit recursion

• Some formula are *explicitly recursive*
  – factorial(n) = 1, if n <= 0
  – factorial(n) = n * factorial(n-1), otherwise

• These are simply transcribed into recursive functions

```java
public static int factorial(int n) {
    if (n <= 0)
        return 1;
    else
        return n * factorial(n-1);
}
```
6.1 Explicit recursion

• This looks like ordinary code using methods
• The only difference is that the method calls itself
• Let's look at the structure of a recursive method in more detail

```java
public static int factorial(int n) {
    if (n <= 0)
        return 1;
    else
        return n * factorial(n-1);
}
```
6.1 Explicit recursion

```java
public static int factorial(int n) {
    if (n <= 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

recursive call
public static int factorial(int n) {
    if (n <= 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
6.2a Exercise

• Video intro
  – Implement the factorial function
  – Write unit tests
  – Debugger and tracing through the code

• Point students to their repo for the upcoming exercise
6.2a Exercise

• **Question card**
  – You do the same for:
    • \( \text{sum}(n) = \text{sum}(n-1) + n \), if \( n > 0 \)
    • \( \text{sum}(n) = 0 \), otherwise
  – In your code, identify (make comments on)
    • Recursive call
    • Base case

• **Implement and test**
6.2 Exercise

• Video response
  – Show solution
6.2b Exercise

- Same setup as previous BLT so this can be a short video intro

  - Insist students look at each others' work before continuing (is this possible?)
6.2b Exercise

• Question card
  – Now do the same for:
    • add(x,y) = x, if y == 0
    • add(x,y) = add(x+1, y-1), otherwise
  – In your code, identify (make comments on)
    • Recursive call
    • Base case

• Implement and test
• Under what circumstances does your method work?
• How could you generalize it to work for any x and y?
6.2b Exercise

• Video response
  – Show solution
6.3 Finding recursive substructure

- Sometimes, we have to find the recursion in a problem
  - It's not explicitly stated
  - This is harder, but more like solving a puzzle
- Requires a bit of faith and vision
  - Trust that a method does what we want it to do
  - See the problem in terms of its simplest subproblem
6.3 Finding recursive substructure

- Example
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

- Think of the \( \text{sum}(n) \) function as a generator of the expression you see on the right
6.3 Finding recursive substructure

- Example
  - $\text{sum}(n) = 0 + 1 + \ldots + n$

- Think of the $\text{sum}(n)$ function as a generator of the expression you see on the right

- Believe that $\text{sum}(n)$ really generates such an expression
  - $\text{sum}(4) = 0 + 1 + 2 + 3 + 4$
6.3 Finding recursive substructure

- Example
  - $\text{sum}(n) = 0 + 1 + \ldots + n$

- Think of the $\text{sum}(n)$ function as a generator of the expression you see on the right

- Believe that $\text{sum}(n)$ really generates such an expression
  - $\text{sum}(4) = 0 + 1 + 2 + 3 + 4$
  - $\text{sum}(x) = 0 + 1 + \ldots + (x-1) + x$
6.3 Finding recursive substructure

• Example
  – \( \text{sum}(n) = 0 + 1 + \ldots + n \)

• Think of the \( \text{sum}(n) \) function as a generator of the expression you see on the right

• Believe that \( \text{sum}(n) \) really generates such an expression
  – \( \text{sum}(4) = 0 + 1 + 2 + 3 + 4 \)
  – \( \text{sum}(x) = 0 + 1 + \ldots + (x-1) + x \)
  – \( \text{sum}(x+9) = 0 + 1 + \ldots + x + (x+1) + \ldots + (x+8) + (x+9) \)
6.3 Finding recursive substructure

- Example
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

- Think of the \( \text{sum}(n) \) function as a generator of the expression you see on the right

- Believe that \( \text{sum}(n) \) really generates such an expression
  - \( \text{sum}(4) = 0 + 1 + 2 + 3 + 4 \)

Where in this expression do we see a smaller example of the expression itself?
6.3 Finding recursive substructure

• Example
  – \( \text{sum}(n) = 0 + 1 + \ldots + n \)

• Think of the \( \text{sum}(n) \) function as a generator of the expression you see on the right

• Believe that \( \text{sum}(n) \) really generates such an expression
  – \( \text{sum}(4) = 0 + 1 + 2 + 3 + 4 \)

Where in this expression do we see a smaller example of the expression itself?
6.3 Finding recursive substructure

• Example
  – \( \text{sum}(n) = 0 + 1 + \ldots + n \)

• Think of the \( \text{sum}(n) \) function as a generator of the expression you see on the right

• Believe that \( \text{sum}(n) \) really generates such an expression
  – \( \text{sum}(4) = 0 + 1 + 2 + 3 + 4 \)
  – \( \text{sum}(3) = 0 + 1 + 2 + 3 \)

Where in this expression do we see a smaller example of the expression itself?
6.3 Finding recursive substructure

- Example
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

- Think of the \( \text{sum}(n) \) function as a generator of the expression you see on the right

- Believe that \( \text{sum}(n) \) really generates such an expression
  - \( \text{sum}(4) = 0 + 1 + 2 + 3 + 4 \)
  - \( \text{sum}(3) = 0 + 1 + 2 + 3 \)
  - \( \text{sum}(4) = \text{sum}(3) + 4 \)

Where in this expression do we see a smaller example of the expression itself?
6.3 Finding recursive substructure

- Where is the recursive substructure?
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)
6.3 Finding recursive substructure

• Where is the recursive substructure?
  – sum(n) = 0 + 1 + … + n

  – sum(n) = 0 + 1 + 2 + …. + (n-1) + n
6.3 Finding recursive substructure

- Where is the recursive substructure?
  - $\text{sum}(n) = 0 + 1 + \ldots + n$

- $\text{sum}(n) = 0 + 1 + 2 + \ldots + (n-1) + n$

Aha, the boxed portion also looks like $\text{sum}(\text{something})$, but sum of what?
6.3 Finding recursive substructure

- Where is the recursive substructure?
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

\[
\text{sum}(n) = \begin{array}{cccc}
0 & + & 1 & + \\
& & 2 & + \\
& & \ldots & + \\
& & (n-1) & + \\
\end{array} + n
\]

if we believe \( \text{sum}(x) \) really generates

\[
0 + 1 + \ldots + x
\]
6.3 Finding recursive substructure

- Where is the recursive substructure?
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

\[
\text{sum}(n) = \text{sum}(n-1) + n
\]

if we believe \( \text{sum}(x) \) really generates

\[
0 + 1 + \ldots + x
\]
6.3 Finding recursive substructure

• Where is the recursive substructure?
  – $\text{sum}(n) = 0 + 1 + \ldots + n$

  – $\text{sum}(n) = \text{sum}(n-1) + n$

Now we have an explicitly recursive formula.
6.3 Finding recursive substructure

- Where is the recursive substructure?
  - sum(n) = 0 + 1 + … + n
  
  - sum(n) = sum(n-1) + n
  - sum(?) = ?

Now we have an explicitly recursive formula.

But it is missing a base case.

What is the smallest example of sum(n)?
6.3 Finding recursive substructure

- Where is the recursive substructure?
  - sum(n) = 0 + 1 + ... + n
    
    - sum(n) = sum(n-1) + n
    - sum(0) = 0

Now we have an explicitly recursive formula.

But it is missing a base case.

What is the smallest example of sum(n)?
6.3 Finding recursive substructure

• Where is the recursive substructure?
  – \( \text{sum}(n) = 0 + 1 + \ldots + n \)

• Leads to
  – \( \text{sum}(n) = 0 \), if \( n == 0 \)
  – \( \text{sum}(n) = \text{sum}(n-1) + n \), otherwise
6.3 Finding recursive substructure

• Where is the recursive substructure?
  – sum(n) = 0 + 1 + … + n

• Leads to
  – sum(n) = 0, if n == 0
  – sum(n) = sum(n-1) + n, otherwise

```java
private static int sum(int n) {
    if (n == 0)
        return 0;
    else
        return sum(n-1) + n;
}
```
6.3 Finding recursive substructure

• Why does the following substructure not work?
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

- \( \text{sum}(n) = 0 + 1 + 2 + \ldots + (n-1) + n \)
Why does the following substructure not work?

- \( \text{sum}(n) = 0 + 1 + \ldots + n \)

\[
\text{sum}(n) = 0 + 1 + 2 + \ldots + (n-1) + n
\]

Boxed expression is

\( \text{sum}(n) - 0 \)
6.3 Finding recursive substructure

• Why does the following substructure not work?
  – \( \text{sum}(n) = 0 + 1 + \ldots + n \)

\[
- \text{sum}(n) = 0 + 1 + 2 + \ldots + (n-1) + n
\]

Leads to:

\[
\text{sum}(n) = 0 + \text{sum}(n) - 0
\]
6.3 Finding recursive substructure

- Why does the following substructure not work?
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

\[
\begin{align*}
\text{sum}(n) & = 0 + 1 + 2 + \ldots + (n-1) + n \\
\end{align*}
\]

Leads to:

\[
\text{sum}(n) = 0 + \text{sum}(n) - 0
\]

This is true, mathematically
6.3 Finding recursive substructure

• Why does the following substructure not work?
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

  - \( \text{sum}(n) = 0 + 1 + 2 + \ldots + (n-1) + n \)

  Leads to:

  \[
  \text{sum}(n) = 0 + \text{sum}(n) - 0
  \]

  This is true, mathematically
  But it does not lead to a simpler use of the sum function
6.3 Finding recursive substructure

- Why does the following substructure not work?
  - \( \text{sum}(n) = 0 + 1 + \ldots + n \)

\[
\text{sum}(n) = 0 + 1 + 2 + \ldots + (n-1) + n
\]

Leads to:

\[
\text{sum}(n) = 0 + \text{sum}(n) - 0
\]

This is true, mathematically

But it does not lead to a simpler use of the sum function
6.3 Finding recursive substructure

To work, we must find a smaller instance of the function

- One that seems easier to compute
- One that points us toward the base case

\[ \text{sum}(n) = 0 + 1 + \ldots + n \]

\[ \text{sum}(n) = 0 + 1 + 2 + \ldots + (n-1) + n \]
Each box is smaller than its containing box. Where does this recursion stop?
Before we understood human reproduction, it was thought that a woman was born with all of the complete humans inside of her that would ever be born.
6.3a Visual recursion and base cases

Before we understood human reproduction, it was thought that a woman was born with all of the complete humans inside of her that would ever be born
6.3a Visual recursion and base cases

Before we understood human reproduction, it was thought that a woman was born with all of the complete humans inside of her that would ever be born. **What could possibly go wrong?**
6.3a Visual recursion and base cases

• Question card
  – What is wrong with the procreation story?

• Write and submit your answer
6.3a Visual recursion and base cases

• Question card
  – Try factorial without its base case

```java
public static int factorial(int n) {
    return n * factorial(n-1);
}
```

  – What do you see when you run this on factorial(2)?
6.3a Visual recursion and base cases

What could possibly go wrong?
Response: presumes an infinite amount of matter
6.3a Visual recursion and base cases

• For factorial
  • We see stack overflow
  • We could make the stack bigger
  • But it would still overflow
• Recursion without base cases
  • leads to stack overflow
6.3b Roundtable

• For formulae, find substructure
  – \( \text{sum}(n) = n + (n-1) + (n-2) + \ldots + 0 \)
    • This is backwards from our earlier example
  – Multiplication
    • \( \text{mpy}(x,y) = x + x + \ldots + x \) (y times)

• Do some pictures, finding substructure
  – Circles
  – Drawing a line
  – Sierpinski
  – square snowflake from intro
  – graph paper
    • Why not \( \frac{1}{2} \) horizontally or vertically?
6.4 Example

- Let's try to generate graph paper recursively
6.4 Example

This is our goal
but how do we
see this
recursively?
This is our goal but how do we see this recursively?
6.4 Example

(size)

(llx,lly)
6.4 Example

(size)

(llx, lly+size/2)

(llx, lly)
6.4 Example

![Diagram of a rectangle with coordinates labeled]

- Top right corner: \((llx + \text{size}, lly + \text{size}/2)\)
- Bottom left corner: \((llx, lly + \text{size}/2)\)
- Bottom right corner: \((llx + \text{size}, lly)\)
- Top left corner: \((llx, lly)\)

Size of the rectangle: \(\text{size}\)
6.4 Example
6.4 Example

(size

(llx+size/2, lly+size)

(llx+size, lly+size/2)

(llx, lly)

(llx, lly+size/2)

(llx+size/2, lly)

(llx+size/2, lly+size)
6.4 Example

(size)

(ulx, uly)

(llx, lly)
6.4 Example

size

(ullx, lly)

(urx, ury)

ur
Example

size

(lrx, lry)

(lr, r)

(lx, ly)
6.4 Example

(size/2) size

(llx, lly+size/2)

(llx, lly)
6.4 Example

```
(size)

(llx+size/2, lly+size/2)

(llx, lly)
```

```
ul

ur

ll

lr

size/2

size
```
Example

(size/2, (llx, lly))
Example

(size)

\[(llx, lly)\]

\[(llx + \text{size}/2, lly)\]

Ir

\[\text{size}/2\]
6.5 Exercise

• Video intro
  – Review slides and coordinates
6.5 Exercise

• Question card
  – Implement graph paper problem

• Think about
  – Simplest substructure
  – Base case
6.5 Exercise

• Video response
  – Show solution
6.6 There is no 6.6
6.7 Substitution model

- How do we reason about the recursive behavior of methods?
- If they are functional – meaning, they have no side effects
- Then we can substitute recursive calls mathematically
  – wherever we see
    - foo(x)
  – we substitute the text of foo using x as its input
- Let's look at factorial again….
public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) =
Substitution model for recursive evaluation

```java
public static int fact(n) {
    if (n==0) return 1;  // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 *
Substitution model for recursive evaluation

```java
public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 *
```
public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1)
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
Substitution model for recursive evaluation

public static int fact(n) {
    if (n==0) return 1;       // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
    = 3 * 2 * fact(1)
Substitution model for recursive evaluation

```java
public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
    = 3 * 2 * fact(1)
    = 3 * 2 * 1 * fact(1-1)
```
Substitution model for recursive evaluation

public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
    = 3 * 2 * fact(1)
    = 3 * 2 * 1 * fact(1-1)
    = 3 * 2 * 1 * fact(0)
public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
    = 3 * 2 * fact(1)
    = 3 * 2 * 1 * fact(1-1)
    = 3 * 2 * 1 * fact(0)
public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
    = 3 * 2 * fact(1)
    = 3 * 2 * 1 * fact(1-1)
    = 3 * 2 * 1 * fact(0)
    = 3 * 2 * 1 * 1
Substitution model for recursive evaluation

```java
public static int fact(int n) {
    if (n==0) return 1;        // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
      = 3 * fact(2)
      = 3 * 2 * fact(2-1)
      = 3 * 2 * fact(1)
      = 3 * 2 * 1 * fact(1-1)
      = 3 * 2 * 1 * fact(0)
      = 3 * 2 * 1 * 1
      = 3 * 2 * 1
      = 3 * 2 * 1
```
public static int fact(n) {
    if (n==0) return 1; // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
    = 3 * 2 * fact(1)
    = 3 * 2 * 1 * fact(1-1)
    = 3 * 2 * 1 * fact(0)
    = 3 * 2 * 1 * 1
    = 3 * 2 * 1
    = 3 * 2
public static int fact(n) {
    if (n==0) return 1;                      // I tend to put base case first
    else return n * fact(n-1);
}

fact(3) = 3 * fact(3-1)
    = 3 * fact(2)
    = 3 * 2 * fact(2-1)
    = 3 * 2 * fact(1)
    = 3 * 2 * 1 * fact(1-1)
    = 3 * 2 * 1 * fact(0)
    = 3 * 2 * 1 * 1
    = 3 * 2
    = 6
Substitution model for recursive evaluation

```java
public static int mys(int n) {
    if (n == 0) return 0;  // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
    = 3 - 2 - mys(1)
    = 3 - 2 - 1 - mys(0)
    = 3 - 2 - 1 - 0
    ....
    = 0 ?
```

No, must keep computation in the boxes
Substitution model for recursive evaluation

public static int mys(n) {
    if (n==0) return 0;                     // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
    = 3 - 2 - mys(1)
    = 3 - 2 - 1 - mys(0)
    = 3 - 2 - 1 - 0
    ....

No, must keep computation in the boxes
Each box deserves a pair of parentheses
Substitution model for recursive evaluation

public static int mys(n) {
    if (n==0) return 0;                      // I tend to put base case first
    else return n - mys(n-1);
}

mys(3)  = 3 - mys(2)
  = 3 - 2 - mys(1)
public static int mys(n) {
    if (n==0) return 0; // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
     = 3 - 2 - mys(1)
     = 3 - 2 - 1 - mys(0)
public static int mys(n) {
    if (n==0) return 0; // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
= 3 - 2 - mys(1)
= 3 - 2 - 1 - mys(0)
= 3 - 2 - 1 - 0

mys(0) = 0
public static int mys(n) {
    if (n==0) return 0; // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - myst(2)
= 3 - 2 - myst(1)
= 3 - 2 - 1 - myst(0)
= 3 - 2 - 1 - 0

mys(0) = 0

mys(1) = 1-0 = 1
public static int mys(n) {
    if (n==0) return 0;  // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
    = 3 - 2 - mys(1)
    = 3 - 2 - 1 - mys(0)
    = 3 - 2 - 1 - 0
    = 3 - 2 - 1

mys(0) = 0
mys(1) = 1-0 = 1
public static int mys(n) {
    if (n==0) return 0;                    // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
    = 3 - 2 - mys(1)
    = 3 - 2 - 1 - mys(0)
    = 3 - 2 - 1 - 0
    = 3 - 2 - 1
    = 3 - 2 - 1

    mys(0) = 0
    mys(1) = 1-0 = 1
    mys(2) = 2-1 = 1
public static int mys(n) {
    if (n==0) return 0; // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
  = 3 - 2 - mys(1)
  = 3 - 2 - 1 - mys(0) 
  = 3 - 2 - 1 - 0 
  = 3 - 2 - 1
  = 3 - 1

mys(0) = 0
mys(1) = 1-0 = 1
mys(2) = 2-1 = 1
Substitution model for recursive evaluation

public static int mys(n) {
    if (n==0) return 0; // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)  
= 3 - 2 - mys(1)  
= 3 - 2 - 1 - mys(0)  
= 3 - 2 - 1 - 0 
= 3 - 2 - 1 
= 3 - 1  
= 3 - 1 

mys(0) = 0

mys(1) = 1 - 0 = 1

mys(2) = 2 - 1 = 1
public static int mys(n) {
    if (n==0) return 0;     // I tend to put base case first
    else return n - mys(n-1);
}

mys(3) = 3 - mys(2)
    = 3 - 2 - mys(1)
        = 3 - 2 - 1 - mys(0)
            = 3 - 2 - 1 - 0
                = 3 - 2 - 1
                    = 3 - 1
                        = 2

mys(0) = 0
mys(1) = 1-0 = 1
mys(2) = 2-1 = 1
6.7b Practice doing substitution

- Let's practice substitution
- Always write = down the page
- \( \text{fact}(3) = 3 \times \text{fact}(2) \)
  
  =

etc
6.7b Practice doing substitution

- Question card
- For sum(n)
  - sum(4)
- For add(x,y) now show substitution for
  - add(10,3)
6.7b Practice doing substitution

• Video response
• For sum(n)
  – sum(4)
• For add(x, y) now show substitution for
  – add(10, 3)
6.7c Tracing recursion

- Use the debugger to trace recursive calls
- Find the code in your repository
6.7c Tracing recursion

- Go over the following with each of the 3 students in roundtable:
  - For `sum(n)`
    - `sum(4)`
  - For `add(x,y)` now show substitution for
    - `add(10,3)`
  - For `mys(n)`
    - `mys(3)`

End of Roundtable
6.8 Conclusion

• Recursion is a powerful technique
  – Nice results with relatively little coding
  – Code often directly reflects the specified computation

• Ingredients of recursion
  – Base case
  – Recursive call

• Sometimes we have to find the substructure
  – Best to find the next smallest subcase of the larger problem

• Substitution model helps us reason about recursion