Overview references to sections in text

- What is recursion? 7.1-7.39 slide 1.7
- Base case 7.1-7.10 slide 13
- How Java stack frames work 7.8-7.10 slide 28-32

Gries slides on A2 function evaluate and evaluate itself are on pinned Piazza note Supplementary Material

// invariant: p = product of c[0..k-1]
what’s the product when k == 0?
Why is the product of an empty bag of values 1?
Suppose bag b contains 2, 2, 5 and p is its product: 20.
Suppose we want to add 4 to the bag and keep p the product.
We do:
insert 4 in the bag;
p = 4 * p;
Suppose bag b is empty and p is its product: what value?.
Suppose we want to add 4 to the bag and keep p the product.
We want to do the same thing:
insert 4 in the bag;
p = 4 * p;
For this to work, the product of the empty bag has to be 1, since 4 * 1 = 4

0 is the identity of + because 0 + x = x
1 is the identity of * because 1 * x = x
false is the identity of || because false || b = b
true is the identity of && because true & b = b
1 is the identity of gcd because gcd((1, x)) = x
For any such operator o, that has an identity, o of the empty bag is the identity of o.
Sum of the empty bag = 0
Product of the empty bag = 1
OR (||) of the empty bag = false.
gcd of the empty bag = 1

gcd: greatest common divisor of the elements of the bag

== vs equals

Once you understand primitive vs reference types, there are only two things to know:
a == b compares a and b’s values
a.equals(b) compares the two objects using the equals method

<table>
<thead>
<tr>
<th>Primitive Types:</th>
<th>Reference Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Object</td>
</tr>
<tr>
<td>boolean</td>
<td>JFrame</td>
</tr>
<tr>
<td>int</td>
<td>String</td>
</tr>
<tr>
<td>float</td>
<td>PHD</td>
</tr>
<tr>
<td>double</td>
<td>int[]</td>
</tr>
<tr>
<td>byte</td>
<td>Animal</td>
</tr>
<tr>
<td>short</td>
<td>Animal[]</td>
</tr>
<tr>
<td>long</td>
<td>... (everything else!)</td>
</tr>
</tbody>
</table>

A variable of the type contains:

A value of that type | A reference to an object of that type
Recap: Executing Recursive Methods

1. Push frame for call onto call stack.
2. Assign arg values to pars.
3. Execute method body.
4. Pop frame from stack and (for a function) push return value on the stack.

```
public int m(int p) {
    int k = p + 1;
    m(5 + 2)
    return p;
}
```

call stack

Recap: Understanding Recursive Methods

1. Have a precise specification
2. Check that the method works in the base case(s).
3. Look at the recursive case(s). In your mind, replace each recursive call by what it does according to the spec and verify correctness.
4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method

Let’s write some recursive methods!

Code will be available on the course webpage.

1. `len` – length of a string without `.length()`
2. `dup` – repeat each character in a string twice
3. `isPal` – check whether a string is a palindrome
4. `rev` – reverse a string

Check palindrome-hood

A String palindrome is a String that reads the same backward and forward:

- `isPal("racecar")` → true
- `isPal("pumpkin")` → false

A String with at least two characters is a palindrome if

- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:

```
AMANAPLANACANALPANAMA
```

A recursive definition!
Problems with recursive structure

Code will be available on the course webpage.
1. exp - exponentiation, the slow way and the fast way
2. perms – list all permutations of a string
3. tile-a-kitchen – place L-shaped tiles on a kitchen floor
4. drawSierpinski – drawing the Sierpinski Triangle

Computing \( b^n \) for \( n \geq 0 \)

Power computation:
- \( b^0 = 1 \)
- If \( n \equiv 0 \), \( b^n = b \cdot b^{n-1} \)
- If \( n \equiv 0 \) and even, \( b^n = (b \cdot b)^{n/2} \)

Judicious use of the third property gives far better algorithm

Example: \( 3^8 = (3 \cdot 3) \cdot (3 \cdot 3) \cdot (3 \cdot 3) = (3 \cdot 3)^4 \)

Computing \( b^n \) for \( n \geq 0 \)

If \( n = 2^{**k} \)
- \( k \) is called the logarithm (to base 2) of \( n \): \( k = \log n \) or \( k = \log(n) \)

}\( ** = b\^{*n}. \) Precondition: \( n \geq 0 */

\textbf{Example:} \( n = 16 \)
Next recursive call: 8
Next recursive call: 4
Next recursive call: 2
Next recursive call: 1

To show difference, we run linear version with bigger \( n \) until out of stack space. Then run log one on that \( n \). See demo.

Table of log to the base 2

\[
\begin{array}{ccc}
\text{k} & \text{n = 2**k} & \text{log n (=} k) \\
0 & 1 & 0 \\
1 & 2 & 1 \\
2 & 4 & 2 \\
3 & 8 & 3 \\
4 & 16 & 4 \\
5 & 32 & 5 \\
6 & 64 & 6 \\
7 & 128 & 7 \\
8 & 256 & 8 \\
9 & 512 & 9 \\
10 & 1024 & 10 \\
11 & 2148 & 11 \\
12 & 32768 & 15 \\
\end{array}
\]
Permutations of a String

\[ \text{perms}(abc): \quad abc, \quad acb, \quad bac, \quad bca, \quad cab, \quad cba } 

Recursive definition:
Each possible first letter, followed by all permutations of the remaining characters.

Tiling Elaine’s kitchen

Kitchen in Gries's house: 8 x 8. Fridge sits on one of 1x1 squares. His wife, Elaine, wants kitchen tiled with el-shaped tiles — every square except where the refrigerator sits should be tiled.

/** tile a 2^n by 2^n kitchen with 1 square filled. */
public static void tile(int n) {
if (n == 0) return;
}

We abstract away keeping track of where the filled square is, etc.

/** tile a 2^n by 2^n kitchen with 1 square filled. */
public static void tile(int n) {
if (n == 0) return;
}

Base case?

We generalize to a 2^n by 2^n kitchen

n > 0. What can we do to get kitchens of size 2^{n-1} by 2^{n-1}

We can tile the upper-right 2^{n-1} by 2^{n-1} kitchen recursively. But we can’t tile the other three because they don’t have a filled square.
What can we do? Remember, the idea is to tile the kitchen!
Sierpinski triangles

- **S** triangle of depth 0:
- **S** triangle of depth 1: 3 **S** triangles of depth 0 drawn at the 3 vertices of the triangle
- **S** triangle of depth 2: 3 **S** triangles of depth 1 drawn at the 3 vertices of the triangle

Conclusion

Recursion is a convenient and powerful way to define functions. Problems that seem insurmountable can often be solved in a “divide-and-conquer” fashion:
- Reduce a big problem to smaller problems of the same kind, solve the smaller problems
- Recombine the solutions to smaller problems to form solution for big problem

http://codingbat.com/java/Recursion-1