// 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 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  
l 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  
for a, b of some reference type, use == to determine  
whether a and b point to the same object.  
a.equals(b) compares the two objects using method equals  
The value of a.equals(b) depends on the specification of  
equals in the class!
== vs equals: Reference types

For reference types, `p1 == p2` determines whether `p1` and `p2` contain the same reference (i.e., point to the same object or are both null).

`p1.equals(p2)` tells whether the objects contain the same information (as defined by whoever implemented equals).

```java
Pt a0 = new Pt(3, 4);
Pt a1 = new Pt(3, 4);
p1 = a0
p2 = a0
p3 = a1
p4 = null

p2 == p1 true
p3 == p1 false
p4 == p1 false
p2.equals(p1) true
p3.equals(p1) true
p4.equals(p1) NullPointerException!
```

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.

For function call: When control given back to call, pop return value, use it as the value of the function call.

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

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

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 \( n \geq 0 \), \( b^n = b \cdot b^{n-1} \)
- If \( n \geq 0 \) and even, \( b^n = (b^2)^{n/2} \)

Judicious use of the third property gives far better algorithm

Example: \( 3^8 = (3^3) \cdot (3^3) \cdot (3^3) = (3^3)^4 \)
Computing $b^n$ for $n \geq 0$

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

```java
static int power(double b, int n) {
    if (n == 0) return 1;
    if (n % 2 == 0) return power(b * b, n / 2);
    return b * power(b, n - 1);
}
```

** = $b^n$. Precondition: $n \geq 0$

Table of log to the base 2

<table>
<thead>
<tr>
<th>$k$</th>
<th>$n = 2^k$</th>
<th>$\log n$ (= $k$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>2148</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>15</td>
</tr>
</tbody>
</table>

Difference between linear and log solutions?

```java
/** = $b^n$. Precondition: $n \geq 0$ */
static int power(double b, int n) {
    if (n == 0) return 1;
    return b * power(b, n - 1);
}
```

Number of recursive calls is $n$

```java
/** = $b^n$. Precondition: $n \geq 0$ */
static int power(double b, int n) {
    if (n == 0) return 1;
    if (n % 2 == 0) return power(b * b, n / 2);
    return b * power(b, n - 1);
}
```

Number of recursive calls is ~ $\log n$.

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

Permutations of a String

```java
public static void perms(String str) {
    if (str.length() == 0) return;
    for (int i = 0; i < str.length(); i++)
        perms(str.substring(0, i) + str.substring(i + 1));
}
```

```java
perms(abc);
```

```
abc, acb, bac, bca, cab, cba
```

Recursive definition:

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

Tiling Elaine’s kitchen

**Per side:**

```
8
```

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

We generalize to a $2^m$ by $2^m$ kitchen.
Tiling Elaine’s kitchen

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

Tiling Elaine’s kitchen

```java
/** tile a 2^n by 2^n kitchen with 1 square filled. */
public static void tile(int n) {
    if (n == 0) return;
    Place one tile so that each kitchen has one square filled;
    Tile upper left kitchen recursively;
    Tile upper right kitchen recursively;
    Tile lower left kitchen recursively;
    Tile lower right kitchen recursively;
}
```

Sierpinski triangles

**S triangle of depth 0:**

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

**S triangle of depth d at points p1, p2, p3:**

```
3 S triangles of depth d-1 drawn at p1, p2, p3
```

Sierpinski triangles of depth d-1

```
p1
p2
p3
```

Sierpinski triangles

```
x
y
```

**S triangle of depth d:**

```
S/4
S/2
S√3/2
```

Sierpinski triangles of depth d-1

```
p1
p2
p3
```
## 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](http://codingbat.com/java/Recursion-1)