Announcements

- A2 grades are available
- For prelim conflicts, please follow instructions at http://www.cs.cornell.edu/courses/CS2110/2015fa/exams.html
Two views of recursive methods

- How are calls on recursive methods executed? We saw that in the lecture. Useful for understanding how recursion works.

- How do we understand a recursive method — know that it satisfies its specification? How do we write a recursive method? This requires a totally different approach. Thinking about how the method gets executed will confuse you completely! We now introduce this approach.
Understanding a recursive method

Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s): Cases where the parameter is small enough that the result can be computed simply and without recursive calls.

If $n < 10$ then $n$ consists of a single digit. Looking at the spec we see that that digit is the required sum.

```java
/** = sum of digits of n. 
 * Precondition: n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;
    // n has at least two digits
    return sum(n/10) + n%10;
}
```
Understanding a recursive method

Step 1. Have a precise spec!
Step 2. Check that the method works in the base case(s).
Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the method spec and verify that the correct result is then obtained.

```
/** = sum of digits of n.
 * Precondition: n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;
    // n has at least two digits
    return sum(n/10) + n%10;
}
```

```
return sum(n/10) + n%10;
```

```
return (sum of digits of n/10) + n%10;  // e.g. n = 843
```
Understanding a recursive method

Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s).

Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does acc. to the spec and verify correctness.

Step 4. (No infinite recursion) Make sure that arguments to recursive calls are in some sense smaller than the parameters of the method.

```java
/** = sum of digits of n. * Precondition: n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;
    // n has at least two digits
    return sum(n/10) + n%10;
}
```

\[ n/10 < n \]
Understanding a recursive method

Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s).

Step 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.

Step 4. (No infinite recursion) Make sure that arguments to recursive calls are in some sense smaller than the parameters of the method.

Important! Can’t do step 3 without it

Once you get the hang of it this is what makes recursion easy! This way of thinking is based on math induction which we will see later in the course.
Writing a recursive method

Step 1. Have a precise spec!

Step 2. Write the base case(s): Cases in which no recursive calls are needed. Generally for “small” values of the parameters.

Step 3. Look at all other cases. See how to define these cases in terms of smaller problems of the same kind. Then implement those definitions using recursive calls for those smaller problems of the same kind. Done suitably point 4 is automatically satisfied.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method.
Example: Palindromes

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

\[
\begin{align*}
\text{e.g.} & \quad \text{AMANAPLANACANALPANAMA} \\
\text{have to be the same} & \\
\text{have to be a palindrome}
\end{align*}
\]

A recursive definition!
Example: Counting characters

```java
/**
 * countEm(c, s) = number of times c occurs in s */
 * public static int countEm(char c, String s) {
 *     if (s.length() == 0) return 0;
 *     // { s has at least 1 character }
 *     if (s.charAt(0) != c)
 *         return countEm(c, s.substring(1));
 *     // { first character of s is c}
 *     return 1 + countEm(c, s.substring(1));
 * }
 * 
 * countEm('e' "it is easy to see that this has many e’s") = 4
 * countEm('e' "Mississippi") = 0
```
Example: Exponentiation

Power computation:
- $a^0 = 1$
- If $n \neq 0$ then $a^n = a \times a^{n-1}$

```c
/**
 * power(a,n) returns $a^n$
 * Precondition: n >= 0 */
static int power(int a, int n) {
    if (n == 0) return 1;
    return a * power(a, n-1);
}
```
Example: Fast Exponentiation

Power computation:
- $a^0 = 1$
- If $n \neq 0$ then $a^n = a \cdot a^{n-1}$
- If $n \neq 0$ and even then $a^n = (a \cdot a)^{n/2}$

```cpp
/** power(a,n) computes $a^n$
 * Precondition: n >= 0 */
static int power(int a, int n) {
    if (n == 0) return 1;
    if (n%2 == 0) return power(a*a, n/2);
    return a * power(a, n-1);
}
```
Example: Searching in a list

```c
/**
 * Returns an index \(i\) between \(fst\) and \(lst\), inclusive,
 * where \(a[i] = k\).
 * Precondition: such an index exists,
 * and \(a\) is sorted in ascending order
 */
static int search(int[] a, int k, int fst, int lst) {
    ...
}
```

```
0  1  2  3  ...
|   |   |   |   |
| n0 | n1 | n2 | n3 |   |

|   |   |   |   |
| fst |   | lst |
```
Example: Sierpinski’s Triangle

/** draws Sierpinski’s triangle bounded
* by p1, p2, and p3 on g up to order n
* Precondition: pi not null, n >= 0 */

private static void displayTriangles(Graphics g, int order, Point p1, Point p2, Point p3) {
    ...
}

- drawLine(g, p1,p2)
- midpoint(p1,p2)