CS 162
Intro to Programming II
Recursion Review
Definition

• Recursion is a function that calls itself.
• The factorial function is easily defined recursively.
  – Fact(0) = 1
  – Fact(n) = Fact(n-1) * n
Definition- Details

• A technique where a function calls itself with a simpler version of the input
  – More than just a programming technique. We use it in Computer Science as:
    • a. A way to define things
    • b. A way to solve problems using divide and conquer
  – Can prove correctness of recursive functions – use induction!
Definition- Details

• Every recursive solution has:
  – A Base Case (when to stop!)
    • What happens if you forget the base case?
    • Infinite loop. Out of memory because too many activation records.
  – A Recursive Step (calling itself)
    • Deferring responsibility: assumes the function you call does the right thing
    • The recursive call operates on “smaller” inputs (smaller in some sense)
How to Write a Recursive Function

• Identify the base case.
  – This tells your program when to stop and is the simplest input to your program

• Identify the recursive step.
  – Look for places where the same computation occurs repeatedly as the problem is solved.
  – Each time the computation is repeated, the method works on a simpler version of the problem

• What happens if the next call is NOT on a smaller or more simple version of the problem?
How to Think Recursively
Using the factorial function

• The factorial function simply returns the number multiplied by all previous non-negative integers.

• Find the base case.

• 0 is the starting point so Fact(0) = 1.

• Find the recursive case.
  – For any n, smaller would be n-1.
  – So we have Fact(n) = Fact(n-1) * n.
Recursion vs Iteration

• Both are forms of repetition
• Each time through a step of recursion you have a function call
  – This means you have the overhead of the call stack and other OS processes
• Each time through a step of iteration you just execute that many lines of code
• So why use recursion?
Why Recursion?

• Some operations are easier to describe than to program.
• Consider the towers of Hanoi.
  – To move the bottom disk you need to move n-1 disks above it.
  – To move the (n-1)th disk you need to move the n-2 disks above that.
  – ...
  – To move the last disk you just move it to an empty peg.
• As already mentioned, we can use induction to prove the correctness of recursive algorithms
You Need to be Smart

• A common example is the Fibonacci Numbers
• This function is doubly recursive so has 2 function calls for each step!!
• Think about what this means?
  – Each recursive call is 2 more function calls
  – Except for the base case
  – Essentially 3 functions for each step of iteration!
You Need to be Smart II

• Mutual recursion, consider this:
  
  ```c
  int f() { ... g()...};
  int g() { ... f()...};
  ```

• Maybe be hidden
  
  ```c
  int f() { ... g()...};
  int g() { ... h()...};
  int h() { ... j()...};
  int j() { ... f()...};
  ```

• May be a problem if either functions base case can be changed by the other function!
The Nightmare?

```c
int f() { ... g(); h() ... };
int g() { ... h(); J() ... };
int h() { ... j(); f() ... };
int j() { ... f(); g() ... };
```
Tail Recursion

- Tail recursive: if there are no pending operations to be performed on return from a recursive call.
- Tail recursive functions return the value of the last recursive call as the value of the function.
- Some compilers will actually convert tail recursive functions to iteration to avoid the stack call overhead. g++ will with optimization set.
Not Tail Recursive

public long rfactorial (int n) {
    if (n == 1)
        return 1;
    else
        return n * rfactorial(n-1); // * is a pending operation after recursive call
}

Tail Recursive

```java
public long factorialHelper (int n, int result) {
    if (n == 1)
        return result;
    else
        return factorialHelper(n-1, n*result);
}

public long factorial(int n) {
    return factorialHelper(n, 1);
}
```