More Complicated Recursion
CMPSC 122

Now that we've gotten a taste of recursion, we'll look at several more examples of recursion that are special in their own way.

I. Example with More Involved Arithmetic

Consider this function, with precondition $n > 0$:

```c
int FOO(n) {
    if (n == 1)
        return 0;
    else
        return 1 + FOO(n/4);
}
```

This is an example to show that we don't always subtract something from $n$ in recursive calls. But, remember the motto that we want to get closer to base cases, so division will accomplish that.

Let's trace the execution of this function for $n = 4$, $n = 16$, and $n = 256$. What does it compute?

II. Example with Multiple Base Cases

Recursion allows us to have multiple ways of defining both the base and recursive cases. More often than multiple recursive cases, we have multiple base cases. Consider the following sequence definition:

$a_1 = 5$
$a_2 = 20$
$a_3 = 21$
$a_n = 10a_{n-1}$ for $n > 3$

**Problem:** Write recursive code to compute the $n$th term of this sequence. Trace its execution for inputs of 5 and 2.
III. Examples where Recursive Case Spawns Multiple Calls

A classic example of a recursively-defined sequence is the Fibonacci sequence:

\[
f_0 = 0 \\
f_1 = 1 \\
f_k = f_{k-1} + f_{k-2} \quad \text{for } k > 2
\]

Let's begin by computing the \(k = 5\) term from the recursive definition:

Now, let's see how this translates to a code solution:

It turns out recursive algorithms that spawn multiple calls are very important in computer science. Often, we wish to solve a problem by breaking it down into smaller instances of the same problem, solving those, and combining the results together for an overall result. This strategy is known as **divide-and-conquer** algorithm design.

**Example:** Suppose we have an array whose size is a power of 2. To sum that array, let's divide it in half, and sum each half independently. In summing the halves, we'll employ the same strategy.

First, let's visualize an array \(A[p..q]\) that we will break apart:
Now let's write a recursive definition for the sum of an array $A[p..q]$.

where $p$ and $q$ are nonnegative integers such that ________________________________:

Base:

Recursion:

Now, let's translate that to code (as an in-class problem, you start this and I'll help):

**Homework Exercise 1**: Trace the execution of this algorithm for the array (1, 2, 3, 4, 5, 6, 7, 8).

**Note**: I'm not advocating you sum an array this way. This solution turns out to be equally efficient as the solution you'd naturally think to do, so it only overcomplicates. But it serves as a nice example of recursion that leads to something deeper we'll do later too.
IV. Example with Multiple Parameters

Just like any ordinary function can have multiple parameters, so can a recursive function.

Sometimes, recursive functions bring multiple parameters "along for the ride" and only one parameter drives the recursion, such as in our first example:

**Example:** Consider this function (whose precondition is that \( n \) is nonnegative and \( a \) is initialized and which doesn't do anything of practical significance):

```c
RECDemo(n, a)
{
    if n = 0         // base case
        return a
    if n > 0 and n mod 2 = 0     // recursive case for even n
        return RECDemo(n-2, a) + 3
    if n > 0 and n mod 2 = 1     // recursive case for odd n
        return RECDemo(n-1, a) + 1
}
```

Trace the action of the call `RECDemo(5, 10)`.

**Homework Exercise 2:** Trace the action of the call `RECDemo(6, 1)`.

Sometimes, multiple parameters mean recursion in multiple dimensions. Applications include problems with two-way tables that arise in optimization problems, problems involving matrix processing, and problems in processing images (after all, images are really matrices that store mathematical representations of color or grayscale).

**Example:** Consider this recursive sequence definition:

\[
A[0, 0] = 2 \\
A[row, 0] = A[row-1, 0] + 2^{\text{row}} \quad \text{for}\ row \geq 1 \\
A[0, col] = A[0, col-1] + 4 \quad \text{for}\ col \geq 1 \\
A[row, col] = \frac{A[row-1, col-1] + A[row, col-1]}{2} \quad \text{for}\ row \geq 1 \text{ and } col \geq 1
\]

Using this definition, fill in a table \( A[0..3, 0..2] \)
Problem: Suppose we wanted to compute \( A[50, 75] \). What terms would we need directly? What all recursive calls would be made?

Homework Exercise 3: Write pseudocode for a recursive function that does the same thing as the table-filling example we just saw. Assume \( A \) is passed by reference and inputs are a row and column index.

Homework Exercise 4: What recursive call would you need to make to your function to fill a table?

V. Lab and Homework

First, on paper, solve the homework exercises that have been scattered through the notes so far. You should find 4 of them.

Next, solve the following exercise on paper:

Homework Exercise 5: Suppose you are building a program which draws several stars on the console. On the first three lines, there should be 1, 2, and 3 stars, respectively. On all subsequent lines, the number of stars should either be twice the number of stars 2/3 of the way up the screen, for lines numbers that are exact multiples of three, or otherwise 2 more than the preceding line. Write a recursive definition for the number of stars on each line.

Then, as a lab exercise, implement the recursive function from Homework Exercise 5, add a method to draw any number of stars, and call it enough times print stars on 54 lines. Prepare a report with the code and a sample run and call this Lab 11.

The first few lines should look like this:

*  
**  
***  
****  
*****  
******
VI. Recursion vs. Iteration and Further Directions

First, recall the classic factorial example. You could certainly write code that computes factorials via a loop:

```java
FACT(n)
{
    result = 1
    for i = 1 to n
    {
        result = result * i
    }
    return result
}
```

And, we could visualize its execution in a different way than call stacks:

Recall the Fibonacci numbers problem we looked at earlier. We wrote a nice recursive function for Fibonacci numbers, but it turns out we could also have written a function that uses a loop. As the process of using loops is sometimes called iteration, a solution based on loops is called an iterative solution.

Consider this pseudocode for an iterative solution, which works for any nonnegative integer n:

```java
FIB(n)
{
    allocate array F[0..n]
    F[0] = 0
    F[1] = 1

    for i = 2 to n
    {
        F[i] = F[i-1] + F[i-2]
    }
}
```
Let's now try to visualize the execution of the recursive function:

What do you observe?

We'll study the running times of iterative algorithms in more detail in this course, as well as learn more about representing trees (which are inherently recursive) as well. We'll also use recursion in deriving a few sorting algorithms that turn out to be more efficient than the ones you know already.

More on recursion comes in the courses to follow. In 360, we'll look at recursion in much greater detail, in particular, deriving closed forms of recursively-defined ideas and proving things about them via recursion's cousin induction. We'll continue that in 465, also looking at many more algorithms that are naturally recursive, as well as studying recursion vs. iteration in more depth via an algorithm design technique called dynamic programming. This is just the theory side of things where we study recursion itself; you'll see applications of recursion in most every remaining course in the curriculum.

**VII. Another Lab Problem**

Recall the binary search algorithm from your CS1 course or look it up in the review materials. There, you looked at iterative solution to binary search, but it can also very naturally be implemented recursively. Your task is to do so. Begin by building pseudocode on paper for a recursive version of binary search. Also, illustrate on paper the action of the recursive binary search for two significantly different kinds of executions. Then, turn your pseudocode into code and test it for the inputs you illustrated. Turn in both a lab report and your work on paper. Staple your lab report on top of your handwritten work.