RECURSION 3

COMPUTER SCIENCE 61A

June 30, 2016

1 Recursion

A *recursive* function is a function that calls itself. Here's a recursive function:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Although we haven't finished defining `factorial`, we are still able to call it since the function body is not evaluated until the function is called. We do have one *base case*: when `n` is `0` or `1`. Now we can compute `factorial(2)` in terms of `factorial(1)`, and `factorial(3)` in terms of `factorial(2)`, and `factorial(4)` – well, you get the idea.

There are *three* common steps in a recursive definition:

1. *Figure out your base case*: What is the simplest argument we could possibly get? For example, `factorial(0)` is 1 by definition.
2. *Make a recursive call with a simpler argument*: Simplify your problem, and assume that a recursive call for this new problem will simply work. This is called the "leap of faith". For `factorial`, we reduce the problem by calling `factorial(n-1)`.
3. *Use your recursive call to solve the full problem*: Remember that we are assuming the recursive call works. With the result of the recursive call, how can you solve the original problem you were asked? For `factorial`, we just multiply $(n-1)!$ by n.

1.1 Cool recursion questions!

1. Create a recursive countdown function that takes in an integer `n` and prints out a countdown from `n` to 1. The function is defined on the next page.

   First, think about a base case for the `countdown` function. What is the simplest input the problem could be given?

   After you've thought of a base case, think about a recursive call with a smaller argument that approches the base case. What happens if you call `countdown(n - 1)`?

   Then, put the base case and the recursive call together, and think about where a print statement would be needed.

```
def countdown(n):
    """
    >>> countdown(3)
    3
    2
    1
    """
```

2. Is there an easy way to change `countdown` to count up instead?

2 Iteration vs. Recursion

We've written `factorial` recursively. Let's compare the iterative and recursive versions:

```
def factorial_recursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n-1)

def factorial_iterative(n):
    total = 1
    while n > 1:
        total = total * n
        n = n - 1
    return total
```

Let's also compare `fibonacci`.

```
def fib_recursive(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)

def fib_iterative(n):
    prev, curr = 0, 1
    while n > 0:
        prev, curr = curr, prev + curr
        n = n - 1
    return prev
```

For the recursive version, we copied the definition of the Fibonacci sequence straight into code! The nth Fibonacci number is simply the sum of the two before it. In iteration, you need to keep track of more numbers and have a better understanding of the code.

Some code is easier to write iteratively and some recursively. Have fun experimenting with both!

1. Our Python interpreter is broken and `pow` no longer works! However, we can write our own replacement. Let's try writing it both iteratively and recursively. Both `expt_iter(base, power)` and `expt_rec(base, power)` implement the exponent function, so we should get the same answer regardless of which one we use. Assume that `power` is a non-negative integer.

```
def expt_iter(base, power):
    """ Implements the exponent function iteratively
    >>> expt_iter(2, 3)
    8
    >>> expt_iter(4, 0)
    1
    """
```

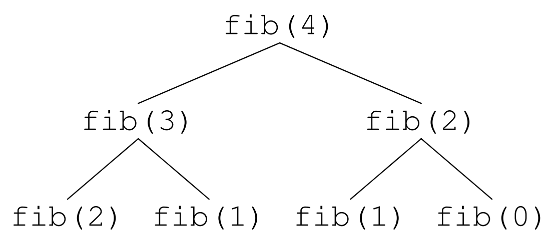
```
def expt_rec(base, power):
    """ Implements the exponent function recursively
    >>> expt_rec(3, 2)
    9
    """
```

3 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is the previous function:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called *tree recursion*, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

1. I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me. Assume `n` is positive.

   Before we start, what's the base case for this question? What is the simplest input?

   What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

   Use those two recursive calls to write the recursive case:

```
def count_stair_ways(n):
```

2. Consider an insect in an `M` by `N` grid. The insect starts at the bottom left corner, `(0, 0)`, and wants to end up at the top right corner `(M-1, N-1)`. The insect is only capable of moving right or up. Write a function `paths` that takes a grid length and width and returns the number of different paths the insect can take from the start to the goal. (There is a closed-form solution to this problem, but try to answer it procedurally using recursion.)

```
def paths(m, n):
    """
    >>> paths(2, 2)
    2
    >>> paths(117, 1)
    1
    """
```

3. The TAs want to print handouts for their students. However, for some unfathomable reason, both the printers are broken; the first printer only prints multiples of `n1`, and the second printer only prints multiples of `n2`. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

   First try to solve without a helper function. Also try to solve using a helper function and adding up to the sum.

```
def has_sum(total, n1, n2):
    """
    >>> has_sum(1, 3, 5)
    False
    >>> has_sum(5, 3, 5) # 0(3) + 1(5) = 5
    True
    >>> has_sum(11, 3, 5) # 2(3) + 1(5) = 11
    True
    """
```

4. The next day, the printers break down even more! Each time they are used, Printer A prints a random x copies $50 \leq x \leq 60$, and Printer B prints a random y copies $130 \leq y \leq 140$. The TAs also relax their expectations: they are satisfied as long as they get at least `lower`, but no more than `upper`, copies printed. (More than `upper` copies is unacceptable because it wastes too much paper.)

   Hint: Try using a helper function.

```
def sum_range(lower, upper):
    """
    >>> sum_range(45, 60) # Printer A prints within this range
    True
    >>> sum_range(40, 55) # Printer A can print a number 56-60
    False
    >>> sum_range(170, 201) # Printer A + Printer B will print
    ...           # somewhere between 180 and 200 copies total
    True
    """
```