RECURSION, RECURSION, (TREE) RECURSION! 3

COMPUTER SCIENCE 61A

September 18, 2013

1 Recursion

A function is *recursive* if it calls itself. Below is recursive `factorial` function.

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

It seems like this won't work – we haven't finished defining `factorial`, yet we're already calling it. However, we do have one *base case*: when `n` is `0` or `1`. Now we can compute `factorial(1)` in terms of `factorial(0)`, and `factorial(2)` in terms of `factorial(1)`, and `factorial(3)` – well, you get the idea.

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

1. *Figure out your base case*: Ask yourself, "what is the simplest argument I could possibly get?" The answer should be simple, and is often given by definition. For example, `factorial(0)` is 1, by definition, or the first two Fibonacci numbers are 0 and 1.
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" – as you use more recursion, you will get more used to this idea. For `factorial`, we make the recursive call `factorial(n-1)` – this is the recursive breakdown.
3. *Use your recursive call to solve the full problem*: Remember that we are assuming your recursive call just 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 Questions!

1. Print out a countdown using recursion.

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

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

3. Write a procedure `expt(base, power)`, which implements the exponent function. For example, `expt(3, 2)` returns 9, and `expt(2, 3)` returns 8. Assume `power` is always an integer. Use recursion, not `pow`!

```
def expt(base, power):
```

4. Write `sum_primes_up_to(n)`, which sums up every prime up to and including `n`. Assume you have an `is_prime()` predicate.

```
def sum_primes_up_to(n):
```

5. Now write `sum_filter_up_to(n, pred)`, which is a general version that adds all integers `1` through `n` that satisfy the argument `pred`.

```
def sum_filter_up_to(n, pred):
```

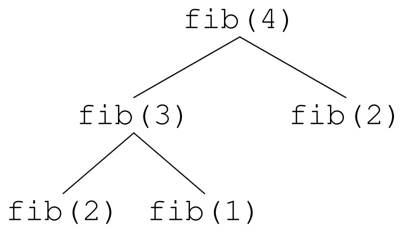
2 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is a function that computes Fibonacci numbers:

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

This is where recursion really begins to shine: it allows us to explore two different calculations at the same time. In this case, we are exploring two different possibilities (or paths): the $n - 1$ case and the $n - 2$ case. With the power of recursion, exploring all possibilities like this is very straightforward. You simply try everything using recursive calls for each case, then combine the answers you get back.

This type of recursion is called *tree recursion*, because the different branches of computation that form from this recursion end up looking like 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, and require the use of additional data structures to hold information. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

2.1 Exercises

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.

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

2. Pascal's triangle is a useful recursive definition that tells us the coefficients in the expansion of the polynomial $(x + a)^n$. Each element in the triangle has a coordinate, given by the row it is on and its position in the row (which you could call its column). Every number in Pascal's triangle is defined as the sum of the item above it and the item that is directly to the upper left of it. If there is a position that does not have an entry, we treat it as if we had a 0 there. Below are the first few rows of the triangle:

```
Item:     0    1    2    3    4    5
Row 0:    1
Row 1:    1    1
Row 2:    1    2    1
Row 3:    1    3    3    1
Row 4:    1    4    6    4    1
Row 5:    1    5    10   10   5    1
...
```

Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle. Don't use the closed-form solution, if you know it.

```
def pascal(row, column):
```

3. Using any combination of numbers 1 through `num`, we want to figure out whether or not we can sum to a given `total`. You can only use each number once. Challenge: Try to sum to `total` using only the factors of `num`.

```
def sum_less_than(total, num):
    """
    >>> sum_less_than(8, 5) # 5 + 3 = 8
    True
    >>> sum_less_than(23, 5) # no way to make 23 by summing 1-5
    False
    """
```

4. 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!

   Also try to solve using a helper function and adding up to the sum.

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

5. 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;
    ...     # the TAs would be satisfied with any number it prints
    ...
    True
    >>> sum_range(40, 55) # Printer A can print some number 56-60
    ...     # copies, which is not within the TA acceptable range
    ...
    False
    >>> sum_range(170, 201) # Printer A + Printer B will print
    ...     # somewhere between 180 and 200 copies total
    ...
    True
    """
```

3 Iteration vs. Recursion

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

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

def factorial_iterative(n):
    total = 1
    while n > 0:
        total = total * n
        n = n - 1
    return total
```

Notice that the recursive test corresponds to the iterative test. While the recursive function "works" until n is less than or equal to 0, the iterative "works" while n is greater than 0. They are essentially the same.

Let's also compare `fibonacci`.

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

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

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

Sometimes code is easier to write iteratively, sometimes code is easier to write recursively. Have fun experimenting with both!