RECURSION 7

COMPUTER SCIENCE 61A

October 15, 2012

1 Recursion

We say a procedure is *recursive* if it calls itself in its body. Below is an example of a recursive procedure to find the factorial of a positive integer n:

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

Upon first glance, it seem like this might not work, since we haven't finished defining factorial, but we're already calling it. However, note that we do have one case that is simple: when `n` is either `0` or `1`. This case, called our *base case*, gives us a starting point for our definition. Now we can compute factorial of 1 in terms of factorial of 0, and the factorial of 2 in terms of the factorial of 1, and the factorial of 3, ... well, you get the idea.

Three common steps in a recursive definition:

1. *Figure out your base case*: Ask yourself, "what is the simplest argument someone could possibly give me?" The answer should be extremely simple, and is often given simply by definition. For example, the factorial of 0 is 1, by definition, or the first two Fibonacci numbers are 0 and 1.

2. *Make a recursive call with a slightly simpler argument*: Simplify your problem somehow, and assume that a recursive call for this new problem will simply just work. This is sometimes referred to as a "leap of faith" – as you do more recursion problems, you will get more used to this idea. For the defintion of `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. Write 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. Use recursion.

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

4. Remember `map`? Given a list of elements and a function, we want to return a list with the function applied to each element. Let's write it recursively!

```
def map(fn, seq):
```

(Extra Challenge: Try to write a mutation version! i.e. Don't return a list and apply `map` by altering the original list.)

5. Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) lists and returns a single list with all of the elements of the two lists, in ascending order. Use recursion.

   Hint: If you can figure out which list has the smallest element out of both, then we know that the resulting merged list will have that smallest element, followed by the merge of the two lists with the smallest item removed. Don't forget to handle the case where one list is empty!

   ```
   def merge(s1, s2):
   ```

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

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

7. 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):
   ```

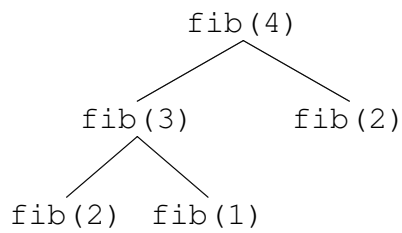
2 Tree Recursion

Say we had a procedure that requires more than one possibility to be computed in order to get an answer. A simple example is a function that computes Fibonacci numbers:

```
def fib(n):
    if n == 1:
        return 0
    elif n == 2:
        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.

We often call this type of recursion, where we use more than one recursive call to find an answer *tree recursion*. We call it 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. Let's say that 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!

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

4. Consider the subset sum problem: you are given a list of integers and a number k. Is there a subset of the list that adds up to k? For example:

```
>>> subset_sum([2, 4, 7, 3], 5)          # 2 + 3 = 5
True
>>> subset_sum([1, 9, 5, 7, 3], 2)
False
>>> subset_sum([1, 1, 5, -1], 3)
False
```

```
def subset_sum(seq, k):
```

5. We will now write one of the faster sorting algorithms commonly used, known as *merge sort*. Merge sort works like this:

   1. If there is only one (or zero) item(s) in the sequence, it is already sorted!
   2. If there are more than one item, then we can split the sequence in half, sort each half recursively, then merge the results, using the `merge` procedure from earlier in the notes). The result will be a sorted sequence.

   Using the algorithm described, write a function `mergesort(seq)` that takes an unsorted sequence and sorts it.

```
def mergesort(seq):
```

3 Iteration vs. Recursion

We have written `factorial` recursively. Let us 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
```

Notice how, recursively, 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 what the code is doing.

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