Recursion

Recursion is the name given to processes that call themselves. It computational complement to mathematical induction. Let's start with a classic example: factorial. By definition factorial is

$$n! = n \times n - 1 \times n - 2 \times \cdots 1$$

for all non-negative values of n where $0! = 1$. One can write this inductively as a recurrence relation.

$$n! = \begin{cases} n \times (n-1)! & (n > 0) \\ 0 & (n = 0) \end{cases}$$

This inductive definition can be translated, almost verbatim, into code in most programming languages. In Python, we'd write this as

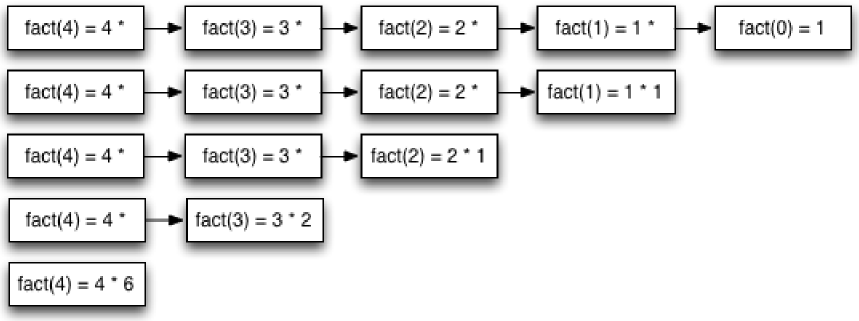
```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

and running the code yields the correct answers:

```
>>> fact(0)
1
>>> fact(1)
1
>>> fact(2)
2
>>> fact(3)
6
>>> fact(4)
24
>>> fact(30)
265252859812191058636308480000000
```

How Recursion Works

Recursion may seem magical, but there is nothing special about a function calling itself—it's just like calling any other function. The execution of the current function moves to the called function and continues when that function returns. The execution of the current function is stored on the *stack*, which acts as a memory of the current state so that it can be restored when execution returns. Here is what the recursive process of the Python `fact(4)` looks like



Thinking Recursively

The key to writing good recursive functions is taking the inductive leap. Put yourself in the mindset where, to solve the current problem, you already have the solution to a slightly simpler problem. Can you use this solution along with your current information to solve the problem at hand? Let's do an example. Suppose you wanted to write a recursive version of multiply called `mult(m, n)` that works for non-negative integers. One way to think about multiplication is as repeated addition. In faction, suppose you knew the result of a slightly simpler problem—`mult(m,n-1)`—adding `m` to the solution of `must(m,n-1)` yields the right answer. What's left is to determine the base case(s). Think about this as instances of your problem that are so simple, you know the answer right away. For multiplication, the base cases are when `n==0` and `n==1`.

Here's how we'd write this in code.

```
def mult(m, n):
  if n == 0:
    return 0
  elif n == 1:
    return m
  else:
    return m + mult(m, n−1)
```

Question 1. *Write a recursive version of exponentiation called* `exp(n, k)` *that computes n^k. Note that exponentiation is repeated multiplication.*

```
>>> exp(2,0)
1
>>> exp(2,1)
2
>>> exp(2,2)
4
>>> exp(2,3)
8
>>> exp(2,10)
1024
```

Structural Recursion

Let's imagine that Python did not support the `len` function on lists. How would we write this ourselves? Let's start with the base case. Let `L` be an empty list. In other words, `L=[]`. Python will evaluate an empty list as `False`, which means, if `L` is a list, you can always write

```
if not L:
  print("empty")
else:
  print("not empty")
```

If `L` is empty, then its length is 0. Base case done. What if `L` is non-empty? Let's think of the list as the first element and the rest of the list. Suppose we know the solution to the length problem on the rest of the list. Then we need only add one more to its length to arrive at a solution to the current problem. Here's the python code.

```
def length(L):
  if not L:
    return 0
    else:
        return 1 + length(L[1:])
```

Here's a couple of executions.

```
>>> length(list(range(10)))
10
>>> length(list(range(20)))
20
```

Let's do another example. Suppose you wanted to write your own version of `sum` that sums a list of numbers in a list. Thinking inductively, we know how to sum an empty list—that's 0—and a non-empty list is just the sum of the first number and the sum of the remaining numbers in the list. Here's the code.

```
def mysum(L):
    if not L:
        return 0
    else:
        return L[0] + mysum(L[1:])
```

Question 2. *Write a recursive version of production called `prod(L)` that computes the product of the numbers in the list `L`.*

```
>>> myprod(list(range(1,5)))
24
>>> myprod(list(range(1,6)))
120
>>> myprod(list(range(1,7)))
720
```

Accumulators

```
def reduce(fn, L):

    def helper(L2, acc):
        if not L2:
            return acc
        else:
            return helper(L2[1:], fn(acc, L2[0]))

    return helper(L[1:], L[0])
```