Tree Recursion in Python

Another common pattern of computation is called tree recursion. As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two.

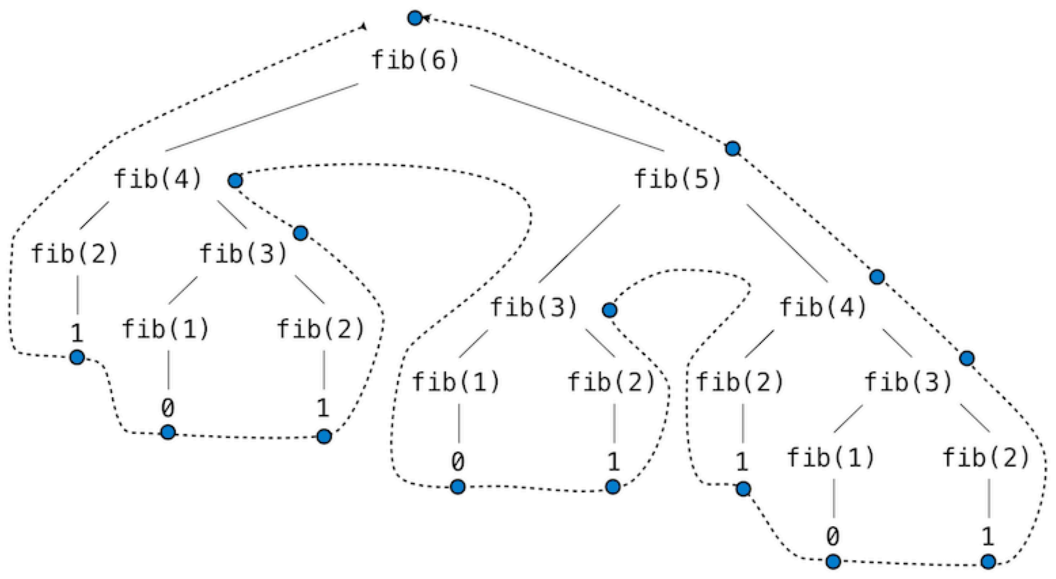
```
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)

result = fib(6)
```

Edit code

< Back Step 1 of 59 Forward >

This recursive definition is tremendously appealing relative to our previous attempts: it exactly mirrors the familiar definition of Fibonacci numbers. Consider the pattern of computation that results from evaluating `fib(6)`, shown below. To compute `fib(6)`, we compute `fib(5)` and `fib(4)`. To compute `fib(5)`, we compute `fib(4)` and `fib(3)`. In general, the evolved process looks like a tree (the diagram below is not a full environment diagram, but instead a simplified depiction of the process). Each blue dot indicates a completed computation of a Fibonacci number in the traversal of this tree.



Functions that call themselves multiple times in this way are said to be *tree recursive*. This function is instructive as a prototypical tree recursion, but it is a terribly inefficient way to compute Fibonacci numbers because it does so much redundant computation. Notice that the entire computation of `fib(4)` -- almost half the work -- is duplicated. In fact, it is not hard to show that the number of times the function will compute `fib(1)` or `fib(2)` (the number of leaves in the tree, in general) is precisely `fib(n+1)`. To get an idea of how bad this is, one can show that the value of `fib(n)` grows exponentially with `n`. `fib(40)` is 63,245,986! The function above uses a number of steps that grows exponentially with the input.

We have already seen an iterative implementation of Fibonacci numbers, repeated here for convenience.

```
>>> def fib_iter(n):
        prev, curr = 1, 0  # curr is the first Fibonacci
number.
        for _ in range(n-1):
             prev, curr = curr, prev + curr
        return curr
```

The state that we must maintain in this case consists of the current and previous Fibonacci numbers. Implicitly, the `for` statement also keeps track of the iteration count. This definition does not reflect the standard mathematical definition of Fibonacci numbers as clearly as the recursive approach. However, the amount of computation required in the iterative implementation is only linear in `n`, rather than exponential. Even for small values of `n`, this difference can be enormous.

One should not conclude from this difference that tree-recursive processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerful tool. Furthermore, tree-recursive processes can often be made more efficient, as we will see in Chapter 3.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#tree-recursion