Computer Science I
Fruitful Functions

CSCI-141
Lecture (2/2)

08/26/2018

1 Recursive Fruitful Functions

We know that *fruitful* functions compute some kind of answer that is returned. Recursive functions can do this, too. It is sometimes helpful for values that have to be computed in a sequence of steps or stages.

Let's do a simple example: division, or how many times a divisor "fits" in a dividend. We will do it using successive subtraction. (Of course, we are pretending that division is not a built-in operation.)

```
def divide( num, denom) :
    """Return quotient of two numbers, rounded down to an integer.
       pre-condition: num, the dividend, is non-negative.
       pre-condition: denom, the divisor, is positive.
    """
    if num < denom:
        return 0
    else:
        # If (num - denom) / denom is q, (num / denom) must be q + 1
        return divide( num - denom, denom ) + 1
```

Note that the same recursion principle applies: There must be at least one base case and some recursive cases.

1.1 Substitution Trace

While constructing execution diagrams is important and useful for understanding and debugging recursive functions, it involves a lot of details that sometimes could be omitted.

Another form of tracing called *substitution tracing* can eliminate a lot of that detail when tracing fruitful functions. Substitution tracing is a generalization of arithmetic expression evaluation. It involves writing a function call with its arguments, an equal sign, and a substitution that is the expression to which it evaluates. This process repeat until the result is computed, and there is nothing to substitute.

Below is an example of tracing the execution of `divide`.

$$\begin{aligned}
\texttt{divide(8,3)} &= \texttt{divide(5,3)} + 1 \\
&= (\texttt{divide(2,3)}) + 1) + 1 \\
&= ((0) + 1) + 1 \\
&= 2
\end{aligned}$$

2 Two Classic Fruitful Functions

There are two mathematical functions that computer scientists like to use as examples: the factorial function and the Fibonacci function. The factorial of a natural number n is written as $n!$, the nth Fibonacci number is written F_n. The mathematical definitions and some sample values are shown below.

$0! = 1$ $n! = n \times (n-1)!$	$Fib_0 = 0$ $Fib_1 = 1$ $Fib_n = Fib_{n-1} + Fib_{n-2}$

n	$n!$	F_n
0	1	0
1	1	1
2	2	1
3	6	2
4	24	3
5	120	5

Here are translations of these functions into Python.

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

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

2.1 Substitution Trace: Factorial

$$\begin{aligned}
\texttt{fact(3)} &= 3 * \texttt{fact(2)} \\
&= 3 * (2 * \texttt{fact(1)}) \\
&= 3 * (2 * (1 * \texttt{fact(0)})) \\
&= 3 * (2 * (1 * 1)) \\
&= 3 * (2 * 1) \\
&= 3 * 2 \\
&= 6
\end{aligned}$$

2.2 Substitution Trace: Fibonacci

$$\begin{aligned}
\texttt{fib(3)} &= \texttt{fib(2)} + \texttt{fib(1)} \\
&= \big(\texttt{fib(1)} + \texttt{fib(0)}\big) + \texttt{fib(1)} \\
&= \big(1 + \texttt{fib(0)}\big) + \texttt{fib(1)} \\
&= (1 + 0) + \texttt{fib(1)} \\
&= 1 + \texttt{fib(1)} \\
&= 1 + 1 \\
&= 2
\end{aligned}$$

Notice that we picked some very small numbers as arguments. What happens with large numbers?

Given a value n as an argument, the factorial function `fact` makes n recursive calls. We can say that the execution time is *linearly proportional to* the value of n, or more succinctly, that the `fact` function's running time is linear in the value of n.

Notice that we are not measuring *real time*, because this is just a piece of code, and we have no idea of the kind of computer or operating system on which the code is running. All we know is that if we provide an argument value that is twice as large as the previous argument value, the function will take about twice as long to execute. Therefore, this is a way of seeing how well an algorithm deals with larger and larger input, and a way of comparing the efficiency of algorithms rather than timing a specific instance of algorithm on a specific computing system.

But what about the Fibonacci algorithm, as coded above? For this example, where the original n is 3, we get 5 calls to the `fib` function. Look what happens with the value 7. Below is an abbreviated substitution trace that shows only the calls to `fib` and the values returned; this was generated automatically.

```
fib( 7 )
  fib( 6 )
    fib( 5 )
      fib( 4 )
        fib( 3 )
          fib( 2 )
            fib( 1 )
            --> 1
            fib( 0 )
            --> 0
          --> 1
          fib( 1 )
          --> 1
        --> 2
        fib( 2 )
          fib( 1 )
          --> 1
          fib( 0 )
          --> 0
        --> 1
      --> 3
      fib( 3 )
        fib( 2 )
          fib( 1 )
          --> 1
          fib( 0 )
          --> 0
        --> 1
        fib( 1 )
        --> 1
      --> 2
    --> 5
    fib( 4 )
      fib( 3 )
        fib( 2 )
          fib( 1 )
          --> 1
          fib( 0 )
          --> 0
        --> 1
        fib( 1 )
        --> 1
      --> 2
      fib( 2 )
        fib( 1 )
        --> 1
        fib( 0 )
        --> 0
      --> 1
    --> 3
  --> 8
  fib( 5 )
    fib( 4 )
      fib( 3 )
        fib( 2 )
          fib( 1 )
          --> 1
          fib( 0 )
          --> 0
        --> 1
        fib( 1 )
        --> 1
      --> 2
      fib( 2 )
        fib( 1 )
        --> 1
        fib( 0 )
        --> 0
      --> 1
    --> 3
    fib( 3 )
      fib( 2 )
        fib( 1 )
        --> 1
        fib( 0 )
        --> 0
      --> 1
      fib( 1 )
      --> 1
    --> 2
  --> 5
--> 13
```

That is 41 calls to the function! Let's look at the trend for different values of n.

What is the pattern?

The number of calls to `fib` when executing `fib(`n`)` is:

$$\begin{gathered}1\ +\\ \text{the number of calls to } \texttt{fib} \text{ when executing } \texttt{fib(}n-1\texttt{)}\ +\\ \text{the number of calls to } \texttt{fib} \text{ when executing } \texttt{fib(}n-2\texttt{)}.\end{gathered}$$

The number of calls almost doubles each time n increases by only one! We call this *exponential growth*, and it is something to avoid whenever possible.

This table shows the growth of the call count.

n	number of calls to `fib(`n`)`
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67

3 Implementation

See the accompanying file, `fruitful.py`, for these functions. Try executing them and tracing how they work. See how long it takes to execute `fib(35)`. Then, edit the code to try putting in slightly larger numbers.

Study of the trace will reveal that the code unnecessarily recomputes values all the time.

There are more efficient algorithms that we will study soon. It is important to understand that we may develop an initial, inefficient algorithm in order to better understand a problem, and then use that to find a better, more efficient solution.