Principles of Writing Recursive Code

coursera.org/learn/pointers-arrays-recursion/supplement/n2ORu/principles-of-writing-recursive-code

Previously, we learned that when translating your steps to code, a complex step should be translated into a call to a function. If you do not already have a function which performs the task you require, you go and write that function once you finish the current one. Writing a recursive function is actually just a special case of this principle: the function we need to call just happens to be the one we are writing. If we are willing to accept that this function will work correctly when we finish it, then it is exactly what we need to use.

At this point, many novice programmers are confused by the seeming “magic” of recursion: if you can just call the function you are writing, why not just write:

```
int factorial_broken(int n) {
  return factorial_broken(n);
}
```

Surely, if the computer can figure out the “magic” of a function calling itself in our first example, it can do the same magic here.

Of course, recursion is not actually magic. In our first example of factorial, we could execute factorial(3) by hand and come up with the correct answer. If we try to execute *factorial_broken(3)*, we will actually never come up with an answer. Instead, we will recurse infinitely—that is *factorial_broken(3)* will just call *factorial_broken(3)*, which will just call *factorial_broken(3)*, and so on forever. Executing this code by hand, we would quickly realize the problem, and stop. The computer, on the other hand, does not reason, and would continue following our instructions, until stopped by something—when it either runs out of memory to make frames, or when the user kills the program.

Instead, let us better understand recursion by examining some key differences between the two functions. One key difference is that the first function was obtained by following The Seven Steps to write a program, and the recursive call corresponded to something we actually did in trying to solve the problem. By contrast, the *factorial_broken* function corresponds to saying “Look, the way you compute factorial is to just compute the factorial." Such a statement is pretty useless in telling you how to perform the required task—although it does lend itself to the joke, "All you have to do to write a correct recursive function is write a correct recursive function."

The second difference is that the factorial function has a *base case* — a condition in which it can give an answer without calling itself—in addition to its *recursive case*—when the function calls itself. In this particular function, the base case is when $n <= 0$ —the function just returns 1 without further work.

Another important aspect of the correct factorial function is that the recursive calls always make progress towards the base case. In this particular case, whenever factorial calls itself, it does so with a smaller parameter value than its own: *factorial* (*3*) calls *factorial* (*2*) which calls *factorial* (*1*), which calls *factorial* (*0*).

In the next videos, we will walk you through Steps 1--5 of writing two recursive functions. The first example we will work through is the factorial function, which we have already been discussing. After that, we will walk you through the Fibonacci function, which is defined as:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{if } n > 1 \\ \mathrm{fib}(-n) & \text{if } n < 0 \text{ and } n \text{ is odd.} \\ -\mathrm{fib}(-n) & \text{if } n < 0 \text{ and } n \text{ is even.} \end{cases}$$