Recursion

Surya Duggirala

December 2016

1 Introduction

Recursion is a tricky topic to wrap your head around if you're not really used to computer programming or some more theoretical math. It is really quite a useful trick though. The official definition according to wikipedia is: "The repeated application of some method". That definition doesn't really capture the true elegance of the entire process though.

2 Thinking

I like visualizing recursion like one of those collapsible telescopes that pirates use. Imagine opening a telescope all the way. Once you hit the very last segment you can't do anything else with it. Think of the base case as similar to the last segment of the telescope. Once you hit the base case then you can't do anything anymore, in essence you can perform the action that the telescope was meant to perform. Then you collapse the entire problem in on itself so that if there are any other numbers or objects that you're manipulating then their values are also taken into consideration. There's not a chance that that made any sense just yet. So let's break it down slowly. Take this program for example:

```
def recurse(x, y):
...     if x == 0:
...         return y
...     print(y)
...     return recurse(x - 1, y + x)
...
>>> recurse(4, 1)
1
5
8
10
11
>>>
>>>
```

```
>>> recurse(10, 4)
4
14
23
31
38
44
49
53
56
58
59
```

This is a pretty basic recursive function. We have one base case and one action that happens if the entire thing works out. This is a tail recursive implementation of a recursive algorithm. Tail recursive means that there is no memory that's stored in a recursive call and no computation is done after the recursive call. The following function is not tail recursive:

```
>>> def no_tail(x):
...     if x == 1:
...         return 1
...     return x * no_tail(x - 1)
...
>>> no_tail(10)
3628800
```

Congrats! We've just implemented a basic factorial calculator. We don't want to use this kind of implementation for large numbers though. The runtime on this will run in linear time so it's super convenient for ballpark smaller size inputs but once you start increasing the input value then you'll notice the runtime gets longer and longer and eventually you'll get a recursion error if you pass in a value large enough. We'll get into runtimes later on, but its important to know that we will always have some minimum runtime that our function can achieve. Now we'll move forward and try to come up with some types of problems you can implement recursively and what an intuitive way to come up with the solution is.

3 Approaches

So when do we use a recursive function over an iterative function? Well for one thing, it's important to note that **every** recursive function can be written iteratively. But sometimes it's definitely easier to implement recursively than it is to do the entire thing iteratively. For example, think about a situation where we are given a particular number and want to follow a set of rules for when it is even vs when it is odd then print the results. Let's set the rules as this. If the number is even, divide by two. If the number is odd, divide by 3. If the number is 1, return.

```
## I want to write a function that will print all
## even numbers less than or equal
## to x and then print all odd numbers less
## than or equal to x.
def even_odd_print(num):
    ## So to get started on a recursive function, we need
    ## a base case.
    ## Ask yourself when you want the function to stop
    ## working.
    ## Break the problem down and look for a pattern.
    ## When does this problem completely stop?
    if num == 1 or num == 0:
        print(num)
        return
    if num % 2 == 0:
        print(num)
        even_odd_print(num // 2)
    else:
        even_odd_print(num // 3)
        print(num)
```

This should always print out a series of even numbers then 1 or 0 then a series of odd numbers. This is an interesting problem to jump into because you're making a recursive call before the print statement if the number is odd. This means you'll run through the **entire** program all the way to the base case before you start printing any odd numbers. This kind of recursive trick is a little difficult to truly grasp so let's run through another problem called **cascades**.

```
>>> def cascades(num):
...     if num == 1:
...         return 1
...     print(num)
...     cascades(num - 1)
...     print(num)
...
>>> cascades(5)
5
4
3
2
1
2
3
4
```

```
5
>>> cascades(10)
10
9
8
7
6
5
4
3
2
1
2
3
4
5
6
7
8
9
10
>>>
```

Here's an interesting experiment. What would happen if you said something like this?

```
>>> val = cascades(5)
```

Would val have a value or not? If you're having trouble with this, run it in the interpreter and refer to my earlier notes on Python and return values to get an idea.

4 Examples

So where do we use recursion in the real world? Think of a large list of numbers. How can we efficiently sort them in order from least to greatest? There are a few nifty sorting algorithms that you'll learn in CS61B that use recursion. The Merge Sort is a very popular one that requires recursion. Look it up in your free time, it's definitely something to have in your toolkit. Apart from that, recursion is useful when you have complicated problems like the Towers of Hanoi, trying to create an algorithm for making change, or when you want to manipulate trees/lists. There'll definitely be more on this topic as we move further into the course. As a parting note, remember that recursion requires one thing above all: base case(s). If your base cases are good, then 99.9999% of the time your answer will work if the rest of your intuition is sound. The best way to master your intuition is to practice. It's important to really understand the inner working of recursive functions. Functional languages like Scheme and Haskell thrive off of recursive calls. This is the backbone of much of the foundational material in computer science.