Week 13, Lecture 1

Today we are going to look at recursive functions.

Interestingly, recent work has shown that crows are capable of understanding the idea of recursion. So if they can, I am very sure you can. :)

https://www.scientificamerican.com/article/crows-perform-yet-another-skill-once-thought-distinctively-human/

First, let's just look at how function calls use program stacks:

Your Python or C program is converted from the high-level in which you write it into an executable form. You can think of this as binary instructions. In this form it is just called an executable or text. When this code actually runs, it needs to access memory where program variables are stored.

When the main program "main()" begins to execute, it is given an "infinite" amount of memory where it can store its variables, objects, lists etc. This area is called the program stack. When main() calls a function, say func1(x,y), the text of the main program will have a "jump to address xxxx" in the place in the text where the call is made. Here xxxx is the address of the first executable

instruction in the function func1(x,y).

Since func1(x,y) is a function just like main(), it must have a stack on which it can store and manipulate its own local variables. Since main() is making a call to func1(x,y), just before the jump to func1() is done, main() sets up a new stack frame for func1() on its own program stack (main() says, I have used memory locations from 0 to 1024, say, and since I am calling func1(), this function can start to use the memory from 1025 onwards, and as much as it needs). One final thing main needs to do before jumping into the entry point of func1() is that it must place the parameter values for x and y at the start of func1()'s stack frame, so func1() can immediately access and use its parameters when it starts to run.

When func1() returns, its stack is removed, and its return tuple (if any) is assigned to some variable(s) on main()'s stack, at the place where the call is made. In this way the entire program stack grows and shrinks during program execution, and disappears altogether when main() terminates.

Now for recursion. Because it will be easier for you to understand recursion by looking at actual code we'll keep this really brief.

Recursion occurs when some function, say func(n) calls itself. Loot at the factorial function, for example.

```
def fact(n):

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

It's okay that we don't have a main() program here. If we make the call to fact(4), Python sets up a main program stack and immediately calls fact(4). Of course, the parameter n is given the value 4 before fact() starts to run.

Function fact(4) executes a return(4*fact(3)), and it does it on the current stack which is the stack of fact(4). It sets up the computation 4 times R where in place of R it makes a function call to fact(3). And in making this call it must compute the n-1 value which is 3, and set it up before jumping into fact() again on a new program stack. That is, the stack of fact(3) is built on top of the stack for fact(4). Now fact(3) will run again and set up the stack of fact(2), and the whole process "bottoms" when fact(1) is called.

When fact(1) executes a return(1), this tuple is returned on the stack of fact(2), where the result 2*1 is made, and now fact(2) returns this result 2 to fact(3)'s stack where the result 3*2 is made, and this result is returned on fact(4)'s stack where the final result 4*6 is made and the

recursion terminates. Notice how the sequence of calls travels to the “bottom” where the “base case result” is made and then the reverse happens as the returns bubble up the chain.

Notice that the recursive code is **simple and pretty**. It takes little time to write, if you understand the recursive structure. However, because of the function calls and the setup of program stacks, it can be (relatively) expensive in terms of time and memory, when compared to non-recursive (iterative — meaning loop-based) versions of the code. But it many cases, it’s not a significant difference.

In this lecture we’ll simply look at a number of examples of recursion, starting with the simplest and working up to a recursive binary search. You’ll notice how elegant a recursive binary search is, compared to the iterative version. It will look mathematical and precise, and often one can usually develop mathematical recurrences (for complexity analysis) simply by looking at the code and writing down symbols for the recursive pattern. We’ll look at an example or two of this later.

In the next lecture we’ll look at other recursions, and revisit the Merge sort and a new sort called the Quicksort.