CprE 310 / Spring 2019

Lecture 21: Recursion in Algorithms

Recursion is a super-important concept in programming. It is really vital to understand not only how to *implement* recursions, but what it *costs* (in terms of computational resources) in order to perform a recursion.

Analyzing algorithms

Much of what we have learned in 310 so far can be very useful in analyzing *algorithms*.

Let's imagine the steps involved in solve any computational problem. Very crudely, these can divided into the following:

1. You first form an *idea* for solving the problem,
2. which you then convert into an *algorithm*,
3. which you then realize via a *software program*.

Step 1 is (more or less) art. Step 3 is the realm of software design, analysis, and testing. But Step 2 is really the heart of computer *science*. CPRE 310 (and 311 later down the road) is all about understanding the principles of such a scientific discipline.

Whenever you design an algorithm for solving a problem, as an engineer you must be prepared to answer two questions:

- Is your algorithm *correct*?
- How *efficient* is your algorithm?

Correctness is established via a *proof*. Efficiency is established by quantitatively estimating the resources (measured in terms of runtime, or memory consumption, or power, or similar quantity) required by your algorithm to run properly.

(Note that this is a separate from establishing correctness of a software program, which can *also* be done via tools from formal logic. Remember that we are still talking about Step 2.)

Example: String reversal

Let's illustrate this using an example.

> Given an input string of n characters, design an algorithm to print its reverse. Analyze its correctness and efficiency.

In text processing problems of this kind, it is customary to define the notion of an empty string (or a string with no characters), which we will denote as λ. There are many ways of solving this problem, and here is one potential algorithm based on recursion:

Function $reverse(s)$

if $s = \lambda$:

return λ

else //if s is non-empty, we can write $s = s'a$

return $a \cdot reverse(s')$

This seems to work! But we need to analyze it.

Let us first confirm that the algorithm indeed does the job by establishing a proof of correctness. But (a) what are we trying to prove, and (b) how do we prove it?

This is where *mathematical induction* comes into the picture. Induction is (by far) the most common proof technique while analyzing any kind of iterative algorithm, or any algorithm that involves recursion. We will use induction on n, the length of the input string. Formally, we will prove the:

Claim: For any string $s = c_1c_2 \dots c_n$, $reverse(s) = c_n \dots c_2c_1$.

Here is the full proof.

(Base case) $n = 0$. In this case, the only possible input is the empty string λ. The output of $reverse()$ applied to λ is λ (due to Line 2 of the algorithm), which is indeed the correct answer.

(Induction hypothesis) Now assume that for any string of length $k \geq 0$, $s = c_1c_2 \dots c_k$, $reverse(s) = c_k \dots c_2c_1$.

(Induction step) Consider an input string of length $k+1$ (say $s = c_1c_2 \dots c_kc_{k+1}$). What happens when s is provided to $reverse()$ as input? Observe that s is non-empty, so the function returns

$c_{k+1}reverse(c_1 \dots c_k)$

But by the induction hypothesis, $reverse(c_1 \dots c_k) = c_k \dots c_1$. Therefore, the output of $reverse()$ applied to s is $c_{k+1}c_k \dots c_1$. Therefore, our algorithm is correct. QED.

As an **exercise**, prove that the algorithm makes $n+1$ calls to $reverse$ (including all recursive calls). Therefore, the running time of the algorithm is $O(n)$.

Example: Tower of Hanoi

The *Tower of Hanoi* problem is a famous problem in combinatorics. Typically one introduces the problem using a picture of 3 pegs with a stack of disks of increasing sizes – something like this:

Figure 1:

Suppose we start with all the disks on Peg 1 (the leftmost one), arranged in increasing disk-radius from top to bottom.

The goal is to move the disks one by one until they are all on Peg 2, while satisfying the Smaller-on-Top Rule. This Rule states that at no point in time can you place a bigger disk on top of a smaller disk. Now suppose we we want to answer the question:

> How many moves are needed to shift a stack of n disks over to a different peg?

In order to count the number of minimum moves, let us recall a very clever *recursive* algorithm to solve this problem. The idea is that if there was some method/routine (say ROUTINE) to solve the problem for n disks, then one can solve the problem for $n+1$ disks by the following algorithm:

- applying ROUTINE to move the top n disks from Peg 1 to Peg 3.
- moving the largest disk from Peg 1 to Peg 2.
- applying ROUTINE to move the n disks from Peg 3 to Peg 2.

If Routine doesn't violate the Smaller-on-Top Rule, then the above recursive algorithm doesn't either. Moreover, the base case of the recursion is for $n = 1$, where one can (trivially) move the lone disk over from one peg to another. Therefore, the above algorithm is correct.

Given the recursive structure of the algorithm, we can analogously *count* the number of moves as follows: let H_n be the number of moves to solve the Tower of Hanoi problem with n disks. Then, the number of moves to solve it for $n+1$

disks (adding up the number of moves for the 3 steps above) is given by:

$$H_{n+1} = H_n + 1 + H_n = 2H_n + 1.$$

The base case is $H_1 = 1$. Plugging in a few numbers, we obtain $H_2 = 3, H_3 = 7, H_4 = 15, \ldots$.

As an easy **exercise**, prove (using induction) that a closed form expression for H_n is given by:

$$H_n = 2^n - 1.$$

Dynamic programming

In ComS 311, you will be exposed to a powerful paradigm known as *dynamic programming* which heavily relies on understanding the structure of recursively defined problems.

We won't go too much into detail, but here is a taste. Consider the Fibonacci numbers, defined as follows:

$$F_1 = 1, \quad F_2 = 1,$$
$$F_n = F_{n-1} + F_{n-2}.$$

Now, if one were to ask you to compute F_n for some arbitrary n (that is potentially very large), how would you do it?

One way (as we discussed before) is to develop a closed-form expression for F_n. Turns out that there *does* exist a closed form expression, called Binet's formula, for the n^{th} Fibonacci number:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right).$$

Now, this is a weird looking closed-form expression which is hard to remember and even harder to prove. So let's just pretend that we were unaware of this formula, and instead compute F_n by writing a program to do the recursion. Here is the straightforward program:

Fib(n):

if $n \leq 2$: $val = 1$

else: val = Fib($n-1$) + Fib($n-2$)

return val

Notice that each function call (for $n > 2$) can potentially spin off two more function calls. So Fib(1000) calls Fib(999) and Fib(998), Fib(999) itself calls Fib(998) and Fib(997), Fib(998) calls Fib(997) and Fib(996) etc. So one can imagine a binary tree of function calls, with roughly n levels (since we are

counting down all the way to 1). However, since there are roughly 2^n nodes in a binary tree with n levels, this means that the number of function calls *is exponential in* n. Not good!

But a closer look reveals that most of these function calls are wasteful! In the above tree, Fib(998) was called twice, Fib(997) was called twice, etc. Can we do it more efficiently so that we don't repeat computations more than once?

Here is a second program to compute the recursion:

list = []

Fib(n):

if n in list: return list[n]

if $n \leq 1$: $val = 1$

else: val = Fib($n-1$) + Fib($n-2$)

list[n] = val

return val

Notice what we did: for every k, Fib(k) only recurses the first time it is called for each $k = 1, 2, \ldots, n$. In particular, there are only n calls (as we count down from n to 1). All other function calls do not spawn any further functions! (i.e., we don't actually don't do the function calls, we just look up list[n]). So in all, the number of function calls is roughly the same order as n.

The key idea here is the fact that we remembered Fib(k) and re-used these values, rather than compute it over and over again. This is a technique known as *memo-ization* (note: not the similar sounding word "memorization", although the meaning is similar), and really is the essence of dynamic programming. But more about this in later courses.