Learning to think with recursion

Daniel King Sep 12, 2016 · 5 min read
From https://medium.com/@daniel.oliver.king/getting-started-with-recursion-f89f57c5b60e

I have frequently heard people new to programming express that they have difficulty understanding how to write recursive algorithms. Although they understand *what* recursion is, they find difficult to understand *how* to go about creating a recursive solution. In this article, my goal is *not* to discuss in detail what recursion is — instead, I would like to discuss how to *do* recursion — that is, how to think through the process of writing a recursive algorithm to solve a problem.

When talking about writing recursive functions, most people focus on the fact that any recursive function needs to have two parts:

1. A base case, in which the function can return the result immediately
2. A recursive case, in which the function must *call itself* to break the current problem down to a simpler level

This pattern can be seen very clearly in one of the most commonly cited examples of a recursive function, the factorial function, shown here in JavaScript:

```
function factorial(n) {
  // Base case
  if (n === 0 || n === 1) return 1;  // Recursive case
  return n * factorial(n – 1);
}
```

Most of the time, I have heard people teaching recursion focus far too much on the base case, and not enough on the recursive case. People usually don't

have much trouble realizing what the base case should be — the harder part is deciding how the recursive call should be structured in order to accomplish the computation you want. Unfortunately, it is difficult to find useful advice about how we should go about doing this. This is what I would like to talk about here — how we should *think* when writing a recursive algorithm.

When I sit down to write a recursive algorithm to solve a problem, I have found it to be helpful to go through the following thought process in order to decide how the recursive call should be structured:

1. Break the problem I am trying to solve down into a problem that is *one step simpler*
2. *Assume* that my function will work to solve the simpler problem — really *believe* it beyond any doubt
3. Ask myself: Since I *know* I can solve the simpler problem, how would I solve the more complex problem?

To illustrate this thought process, let's look at an example. Suppose that we want to write a recursive function that will return a reversed copy of a given string:

```
function reverse(string) {
  // Recursive stuff here
}
```

I won't be surprised if you're wondering, "Why do we need a recursive function to reverse a string? Wouldn't it be easier to write a loop?" I completely agree! However, it's important for us to look at a straightforward example first, before we tackle any algorithms that really require recursion.

First, let's get the base case out of the way: If the string is only one character (or, for that matter, if it is empty), then we don't need to do anything to reverse the string, and we can simply return it:

```
function reverse(string) {
  // Base case
```

```
  if (string.length < 2) return string;
}
```

Now, for the more challenging part — figuring out how to write the recursive call to accomplish the rest of the task. Let's go through the thought process that we outlined above:

1. I am trying to reverse a string. A problem one step simpler would be to reverse a string that is *one letter shorter*.
2. I will assume, and *believe* with every fiber of my being, that my function can correctly reverse a string that is one letter shorter than the one I am
   currently trying to reverse.
3. I ask myself: Since I *know and believe* that my function can correctly reverse a string that is one letter shorter than the one I am currently trying to reverse, how can I reverse the whole string? Well, I can take all of the characters except the first one, reverse those (which I *know and believe* that my function can do), and then tack the first character on to the end! In code, it would look like this:

```
function reverse(string) {
  // Base case
  if (string.length < 2) return string;  // Recursive case
  return reverse(string.slice(1, string.length)) + string[0];
}
```

Think carefully and make sure that you understand steps 1–3 of the thought process above, and how those logical steps resulted in the code shown. If there is anything unclear about it, read it again and think about it some more. Or, leave a question below!

One common mistake that I see people make when trying to develop a recursive algorithm to solve a problem is that they try to think about how to break the problem down *all the way to the base case*. I would like to emphasize that in order to develop the function above, I did *not* think about how I could break the problem down all the way to the base case. That is the *function's* job, not yours. Instead, I *only* thought about the problem that

is *one step simpler* than the problem I am really trying to solve, and then I wrote my recursive algorithm to build up from there to solve the real problem.

When learning to work with recursion myself, I found it extremely helpful to try doing simple tasks with recursion that I would normally do with a loop. If you're interested in practicing the skill of writing recursive algorithms, try applying the thought process above to a few of the following problems — no loops allowed!

1. Go through an array and print out all of the elements
2. Determine whether or not a string is a palindrome
3. Calculate *a* raised to the power of *b*
4. Extra credit: Try implementing the *map* function (the one that transforms arrays) without using loops

Although the concept of recursion is fairly straightforward to describe (a function that calls itself, how complicated can it be?), it is notoriously challenging to think about how we can use this technique in practice to solve problems. I have enjoyed the process of learning to use recursion in my own programming, and I hope that this article has helped to demystify it a little bit.

Stay tuned for part 2, where we'll explore some more interesting examples of recursion in practice!

Daniel King is a professional software engineer and educator in the Los Angeles area who is available for software development and coaching on a contract basis.

daniel.oliver.king@gmail.com