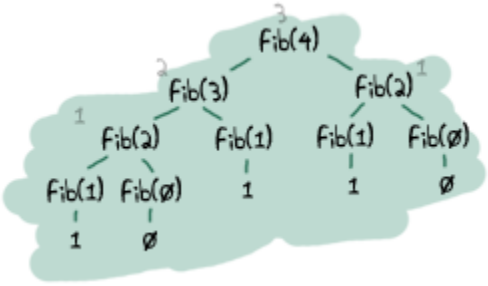
THINKING RECURSIVELY

If you've understood everything in this chapter, thinking recursively is probably becoming more intuitive. A different aspect of recursive definitions when compared to their imperative counterparts (usually in while or for loops) is that instead of taking a step-by-step approach ("do this, then that, then this, then you're done"), our approach is more declarative ("if you get this input, do that, this otherwise"). This property is made more obvious with the help of pattern matching in function heads.

If you still haven't grasped how recursion works, maybe reading this will help you.
Joking aside, recursion coupled with pattern matching is sometimes an optimal solution to the problem of writing concise algorithms that are easy to understand. By subdividing each part of a problem into separate functions until they can no longer be simplified, the algorithm becomes nothing but assembling a bunch of correct answers coming from short routines (that's a bit similar to what we did with quicksort). This kind of mental abstraction is also possible with your everyday loops, but I believe the practice is easier with recursion. Your mileage may vary.

And now ladies and gentlemen, a discussion: *the author vs. himself*

- — Okay, I think I understand recursion. I get the declarative aspect of it. I get it has mathematical roots, like with invariable variables. I get that you find it easier in some cases. What else?
- — It respects a regular pattern. Find the base cases, write them down, then every other cases should try to converge to these base cases to get your answer. It makes writing functions pretty easy.
- — Yeah, I got that, you repeated it a bunch of times already. My loops can do the same.
- — Yes they can. Can't deny that!
- — Right. A thing I don't get is why you bothered writing all these non-tail recursive versions if they're not as good as tail recursive ones.

- — Oh it's simply to make things easier to grasp. Moving from regular recursion, which is prettier and easier to understand, to tail recursion, which is theoretically more efficient, sounded like a good way to show all options.
- — Right, so they're useless except for educational purposes, I get it.
- — Not exactly. In practice you'll see little difference in the performance between tail recursive and normal recursive calls. The areas to take care of are in functions that are supposed to loop infinitely, like main loops. There's also a type of functions that will always generate very large stacks, be slow and possibly crash early if you don't make them tail recursive. The best example of this is the Fibonacci function, which grows exponentially if it's not iterative or tail recursive.
- 

- You should profile your code (I'll show how to do that at a later point, I promise), see what slows it down, and fix it.
- — But loops are always iterative and make this a non-issue.
- — Yes, but... but... my beautiful Erlang...
- — Well isn't that great? All that learning because there is no 'while' or 'for' in Erlang. Thank you very much I'm going back to programming my toaster in C!
- — Not so fast there! Functional programming languages have other assets! If we've found some base case patterns to make our life easier when writing recursive functions, a bunch of smart people have found many more to the point where you will need to write very few recursive functions yourself. If you stay around, I'll show you how such abstractions can be built. But for this we will need more power. Let me tell you about higher order functions...

Source :http://learnyousomeerlang.com/recursion