Is Recursion Slow? | Coursera

Is Recursion Slow?

Is Recursion Slow?

We will note that this implementation of the Fibonacci function is quite slow, even for moderately sized *n*. On my computer, computing *fib*(*46*) takes about 8 seconds—quite a long time for the computer. One incorrect conclusion that perpetuates as a bit of an "urban myth" is that recursion is slow. What is actually happening here is that the way that we have written our *fib* algorithm duplicates work—our implementation will compute *fib*(*44*) twice, *fib*(*43*) three times, *fib*(*42*) five times, *fib*(*41*) eight times, …*fib*(*1*) 1,836,311,903 times, and *fib*(*0*) 1,134,903,170 times! In total, evaluating *fib*(*46*) will require 5,942,430,145 total calls to the Fibonacci function. To help you see how this duplication of work occurs, the next video will illustrate all the calls that are made to evaluate *fib(5)*.

The problem here is not that recursion is slow. It is that algorithms which recompute the same thing millions of times are slow. For the moment, we are not terribly concerned with performance—we want to focus on learning to write correctly working code. However, there are times when performance matters. In some cases, a horribly inefficient/slow algorithm such as this would be unacceptable, but anything with reasonable performance would be just fine. In other cases, every last bit of performance counts, and highly skilled (and well paid!) programmers spend hours refining their code for speed.

What could we do if we had an algorithm such as this, but the slow performance were unacceptable? We need to find some way to express our algorithm such that it does not repeat work that it has already done. In the case of Fibonacci, we could rethink our algorithm to work up from 0 and 1 (computing *fib*(*2*), then *fib*(*3*), then *fib*(*4*), and so on)—at any step, we just add together the previous two values (we compute *fib*(*5*) by adding together whatever we just computed for *fib*(*3*) and *fib(4)*). However, rethinking the algorithm may be tricky—to come up with a different approach, you need to convince yourself to think about the problem differently.

Another way to fix the performance problem without changing the underlying algorithm is *memoization*—keeping a table of values that we have already computed, and checking that table to see if we already know the answer before we recompute something. We will learn later how to efficiently store and lookup data, which would make a memoized version of the function quite fast.

✓

Completed