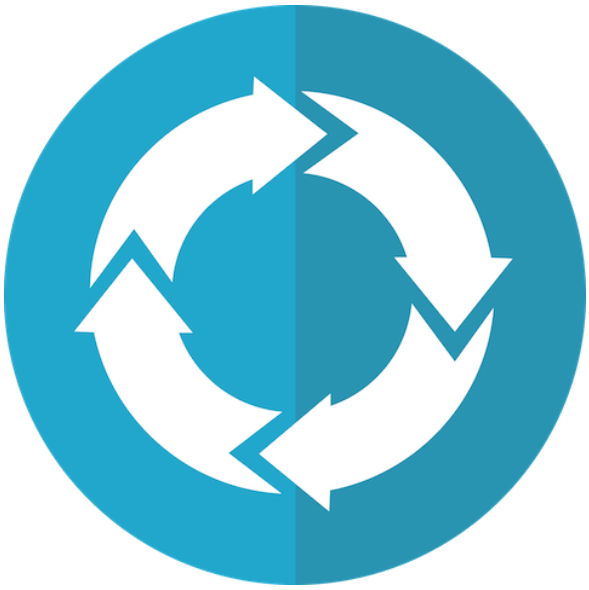


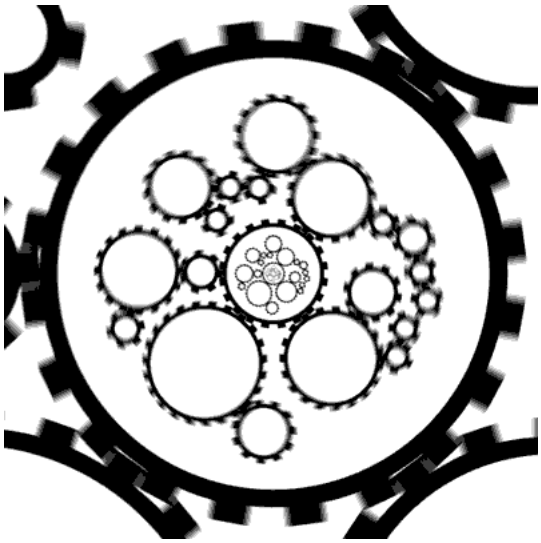
Recursive Programming

How to solve a problem by pretending you already have

Tom Grigg

Jan 17 · 10 min read

Despite often being introduced early-on in most ventures into programming, the concept of recursion can seem strange and potentially off-putting upon first encountering it. It seems almost paradoxical: how can we find a solution to a problem using the solution to the same problem?



Recursion can be a bit of a headache

For those trying to get to grips with the concept of recursion, I often feel it can be beneficial to first realise that recursion is more than just a programmatic practise—it is a philosophy of problem solving that is suitable for problems that can be worked on and partially solved, leaving the remainder of the problem in the same form, but easier or smaller in some way. This does not just apply to functions in

programming; we can frame simple everyday problems using recursion. For example, take me, writing this post: let's say I want to make it around 1000 words long, if I aim to write 100 words every time I open it up, then the first time I write 100 words, and leave myself 900 words left to write. Next time, I write 100 words and only have 800 to go. I can continue this until I have 0 words left to write. Each time, I partially solve the problem and the remaining problem is being reduced.

The pseudo-code for writing my post would look like this:

```
write_words(words_left):
  if words left > 0:
    write_100_words()
    words_left = words_left - 100
    write_words(words_left)
```

I could also implement this algorithm iteratively:

```
write_words(words_left):
  while words_left > 0:
    write_100_words()
    words_left = words_left - 100
```

If you walk the function call `write_words(1000)` through with either implementation, you will find that they have exactly the same behaviour. In fact, every problem we can solve using recursion, we can also solve using iteration (`for` and `while` loops). So why would we ever choose to use recursion?

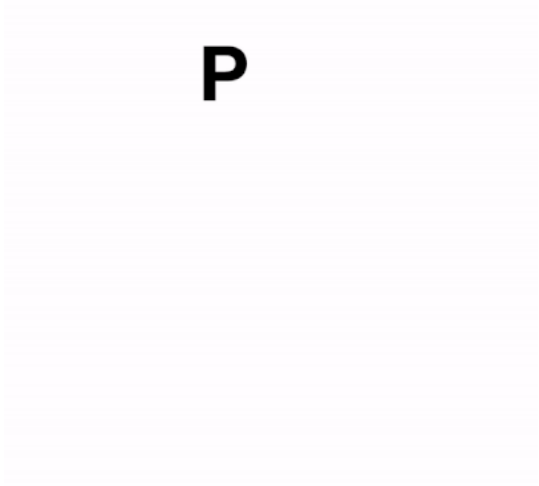
Me writing this post recursively

Why recursion?

Believe it or not, once we get to grips with it, some problems are easier to solve using recursion than they are to solve using iteration. Sometimes recursion is more efficient, and sometimes it is more readable; sometimes recursion is neither faster nor more readable, but quicker to implement. There are data-structures, such as trees, that are well-suited to recursive algorithms. There are even some programming languages with no concept of a loop—purely functional languages such as Haskell depend entirely on recursion for iterative problem solving. The point is simple: You don't have to understand recursion to be a programmer, but you do have to understand recursion to start to become a *good* programmer. In fact, I'd go as far as to say that understanding recursion is part of being a good problem solver, all programming aside!

The Essence of Recursion

In general, with recursion we try to break down a more complex problem into a simple step towards the solution and a remainder that is an easier version of the same problem. We can then repeat this process, taking the same step towards the solution each time, until we reach a version of our problem with a very simple solution (referred to as a base case). The simple solution to our base case aggregated with the steps we took to get there then form a solution to our original problem.



We can solve P by breaking the problem into a step towards the solution and a remaining smaller problem of the same form as the original, until we reach a simple solution to a small problem (a base case).

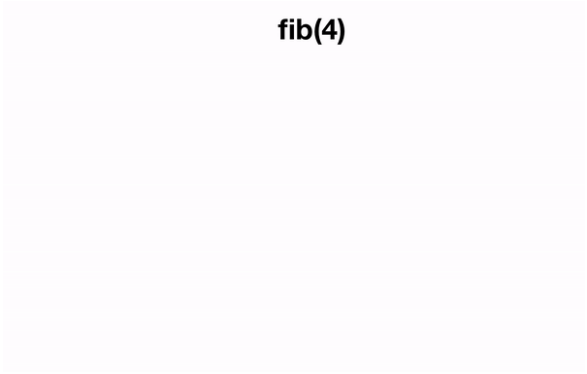
Suppose we are given some actual data of some data-type, call it d_o. The idea with recursion is to pretend that we have already solved the problem or computed the desired function *f* for all forms of this data-type that are simpler than d_o according to some *degree* of difficulty that we need to define. Then, if we can find a way of expressing $f(d_o)$ in terms of one or more *f(d)*s, where all of these *d* s are less difficult (have smaller *degree*) than d_o, then we have found a simple way to reduce and solve $f(d_o)$. We repeat this process, and hopefully, at some point, the remaining *f(d)*s will get so simple that we can easily implement a fixed, closed solution to them. Then, our solution to the original problem will reveal itself as our solutions to progressively simpler problems aggregate and cascade back up to the top.

In the above example of writing this post, the data is the text contained in this document waiting to be written, and the *degree* of difficulty is the length of the document. It's a bit of a contrived example, but assuming I've already solved the problem *f(900)* of how to write 900 words, then all I need to do to solve *f(1000)* is to write 100 words and then execute my solution for 900 words, *f(900)*.

A better example is found in considering the Fibonacci numbers, where the 1st Fibonacci number is 0, the 2nd is 1 and the n^{th} Fibonacci number is equal to the sum of the previous two. Let's say we have a Fibonacci function that tells us the n^{th} Fibonacci number:

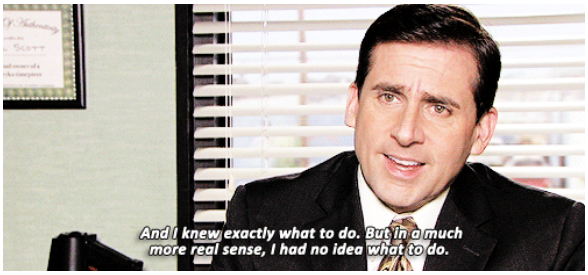
```
fib(n):
  if n == 0:
    return 0
  if n == 1:
    return 1
  else:
    return fib(n-1) + fib(n-2)
```

What does the execution of this function look like? Let's try `fib(4)` :



Visualisation of a recursion tree showing the recursive computation that leads to fib(4) = 3, notice that computation is performed depth-first.

A useful mantra to adopt when solving problems recursively is '*fake it 'til you make it'*, that is, pretend you've already solved the problem for a simpler case, and then try to reduce the larger problem to use the solution for this simpler case. If a problem is suited to recursion, there should actually only be a small number of simple cases which you need to explicitly solve, i.e. this method of reducing to a simpler problem can be used to solve every other case. This is exemplified in the Fibonacci example `fib` where to define `fib(n)` we just act like we've already computed `fib(n-1)` and `fib(n-2)` and, as we hoped, this cascades and reduces the problem to progressively simpler cases, until we reach `fib(0)` and `fib(1)` which have fixed and easy solutions.



Fake it 'til you make it!

Recursive Strategy

Recursion is somewhat nuanced and really depends on what problem you're trying to solve. However, there are some general steps we can come up with that can more or less lead us in the right direction. This strategy is contained in three steps:

1. Order Your Data
2. Solve the Little Cases
3. Solve the Big Cases

As I've said before, I think it can be useful to carry an example through as we learn, but remember that recursion is problem dependent and so try to focus on the general principles here. We'll use the simple example of reversing a string, i.e. we want to write the function `reverse` such that `reverse('Hello world') = 'dlrow olleH'` . I'd recommend going

back and seeing how these steps apply to the Fibonacci function, and then take them and try them on some other examples (there are plenty of exercises online).

Order Your Data

This step is absolutely the key to getting started in solving a problem recursively, and yet it is often overlooked or performed implicitly. Whatever data we are operating on, whether it is numbers, strings, lists, binary trees or people, it is necessary to explicitly find an appropriate ordering that gives us a direction to move in to make the problem smaller. This ordering depends entirely on the problem, but a good start is to think of the obvious orderings: numbers come with their own ordering, strings and lists can be ordered by their length, binary trees can be ordered by depth, and people can be ordered in an infinite number of sensible ways, e.g. height, weight or rank in an organisation. As previously mentioned, this ordering should correspond to the *degree* of difficulty for the problem we are trying to solve.

Order that data, yee-haw!

Once we've ordered our data, we can think of it as something that we can reduce. In fact, we can write out our ordering as a sequence:

0, *1*, *2*, …, *n* for integers (i.e. for integer data *d*, *degree(d)* = *d*)

[], [■], [■, ■], …, [■, … , ■] for lists
(notice len = 0, len = 1, …, len = n i.e. for list data *d*, *degree(d)* = *len(d)*)

Moving from right to left we move through the general ('big') cases, to the base ('little') cases. For our `reverse` example, we are operating on a string, and we can choose the length of the string as an ordering or *degree* of our problem.

Solve the Little Cases

This is normally the easy part. Once we have the correct ordering, we need to look at the smallest elements in our ordering, and decide how we are going to handle them. Usually there is an obvious solution: in the case of `reverse(s)`, once we get to `len(s) == 0` and we have `reverse('')` then we know our answer, because reversing the empty string would do nothing, i.e. we'd just return the empty string since we have no characters to move around. Once we have solved our base cases, and we know our ordering, then solving the general case is as simple as reducing the problem in such a way that the *degree* of the data we're operating on moves towards the base cases. We need to be careful that we don't miss any of the little cases out: the reason they're called base cases is because they cover the base of the ordering—in more complicated recursion problems it is common to miss a base case

so that the reduction step shoots past the sensible end of our ordering and starts operating on nonsense data, or resulting in an error.

Solve the Big Cases

Here, we handle the data rightwards in our ordering, that is, data of high degree. Usually, we consider data of arbitrary *degree* and aim to find a way to solve the problem by reducing it to an expression containing the same problem of lesser *degree*, e.g. in our Fibonacci example we started with arbitrary *n* and reduced `fib(n)` to `fib(n-1) + fib(n-2)` which is an expression containing two instances of the problem we started with, of lesser *degree (n-1* and *n-2*, respectively).

Big case?

When it comes to `reverse` we can consider an arbitrary string of length *n*, and we can pretend our `reverse` function works on all strings of length less than *n*. How can we use this to solve the problem for a string of length *n*? Well, we could just reverse the string containing everything except the last character, and then stick that last character on the front. In code:

```
reverse(string) = reverse(string[-1]) + reverse(string[:-1])
```

where `string[-1]` corresponds to the last character, and `string[:-1]` corresponds to the string without the last character (these are pythonisms). That last `reverse(string[:-1])` term is our original problem, but operating on a string of length *n-1*, i.e. we've expressed our original problem in terms of a step towards the solution combined with the same problem of reduced degree.

Putting the solution to our `reverse` function together, we get the following:

```
reverse(string):
  if len(string) == 0:
    return ''
  else:
    return string[-1] + reverse(string[:-1])
```

Visualisation of recursive function reverse operating on some sample data.

There is often more than one recursive case that needs to be considered as data of a given data-type can take slightly different forms, but this is entirely problem dependent. For an example, consider if we wanted to flatten a list of items, some of which could themselves be lists, we would need to distinguish between the cases where the item we are pulling out of the list is an individual item or a sublist, leading to at least two recursive cases.

. . .

Final Tips

The only real way to get better at recursion is practise. Have a look online for some of the thousands of recursion problems, or challenge yourself to come up with problems that you think might be suited to recursion. Once you inevitably get the hang of recursion, remember that if you find yourself having difficulty solving a problem recursively, then try iteration instead. Outside of learning to be a better programmer, recursion is a method of problem solving to make your life *easier*. If a problem isn't suited to recursion, it just isn't suited to recursion; you'll develop a feel for this as you spend more time approaching problems that lend themselves to either recursive or iterative approaches.

Sometimes in more difficult recursion problems, steps 2 and 3 in the strategy we saw above take the form of a more cyclic feedback-loop process. If you can't find an overall solution to the problem quickly, the best process is to solve the recursive/'big' cases that you can think of, and solve the base/'little' cases that you can think of, and then see how your method breaks on different pieces of data. This should unveil any missing base and recursive cases, or any that are interacting with each other poorly and need to be rethought.

Finally, recall that knowing your ordering is the most important step to solving a recursive problem, and your aim is always to cover both the rightward (recursive) and leftmost (base) cases of this ordering to solve the problem for all data of the given type.

That's a wrap—thanks for reading!

If you enjoyed this introduction to recursion, feel free to get in touch with me (Tom Grigg) regarding any thoughts, queries or suggestions for future

blog posts!

Now, back to working on my data science posts, stay tuned!