

Some quick remarks on Recursion vs Iteration

Take the function that yields the Fibonacci-Sequence which in Mathematics is usually defined as follows:

$$\text{Fib}(1) = 1 \quad (\text{i})$$

$$\text{Fib}(2) = 1 \quad (\text{ii})$$

$$\text{Fib}(n+2) = \text{Fib}(n) + \text{Fib}(n+1), n \geq 1 \quad (\text{iii})$$

This is what a mathematician calls an *iterated* or *recursive* function – note that at the level of abstraction mathematics operates, no further distinction can be drawn. Arguing whether the Fibonacci-Sequence is generated by a recursive or an iterative process seems to me to be completely senseless, mathematically speaking at least.

In computer science, however, there is a fundamental distinction between recursive and iterative *algorithms*. An algorithm contrasts with a function in that functions are usually taken to have extensional identity conditions – if you asked me to evaluate the function

$$f(x) = 3x + 2x + 9x$$

and to not evaluate the function

$$g(x) = 14x,$$

I would not know what to do. Evaluating $f(x)$ just IS evaluating $g(x)$, for in some sense, $f(x)$ IS $g(x)$. Algorithms, however, do not simply specify a pairing of input values with output values, they also specify the precise way of how to get from the inputs to the outputs. I'm afraid this is rather confusing, so let's try to give an example. One way to go about calculating the n^{th} Fibonacci-Sequence, i.e. evaluating Fib at n , is to just cling to the definition given above. This results in the following recursive program (this is actual python code – if you go to <http://www.datamech.com/devan/trypython/trypython.py>, you can enter the code and try it out, or you could just run Python locally):

```
>>> def fib_rec(n):
...     if n==1 or n==2:
...         return 1
...     else:
...         return fib_rec(n-2) + fib_rec(n-1)
```

Why this works should be pretty obvious. The first if-statement covers the two base-clauses of our inductive definition, and the else-clause covers the inductive step. It does this by *recursively calling* the function itself (`fib_rec`) just as stated by our definition (it adds the two predecessors), and therefore is a recursive algorithm for calculating the n^{th} Fibonacci number. It is interesting to see that while this works fine for small n , it runs into performance problems for larger n – and it is not only that your computer might take some time to give

you the result, he might be unable to give you any result at all. This has to do¹ with the fact that each function call needs some amount of memory – this memory can only be freed once the function has terminated, i.e. returned its target value. If, however, a function has to call other functions (which in turn have to call other functions and so on) to terminate, more and more memory is used up and cannot be freed again – and it might happen that there is no more free memory to execute the function calls that are needed for the already used memory to be freed again ... you literally run out-of memory. Again, this might be rather confusing, so I'll try to illustrate what happens if a computer executes the above algorithm for $n=5$.

```
(1) Fib_rec(4)    → case (iii) → ret Fib_rec(2) + Fib_rec(3)
(2) Fib_rec(2)    → case (ii)  → ret 1
(1) Fib_rec(4)    → 1 + Fib_rec(3)
(3) Fib_rec(3)    → case (iii) → ret    Fib_rec(1)    +    Fib_rec(2)
(4) Fib_rec(1)    → case (i)   → ret 1
(3) Fib_rec(3)    → ret 1 + Fib_rec(2)
(5) Fib_rec(2)    → case (ii)  → ret 1
(3) Fib_rec(3)    → ret 1 + 1 → ret 2
(1) Fib_rec(4)    → 1 + 2 → ret 3
```

This is a somewhat informal and, I'm afraid, messy depiction but it should suffice to illustrate the important point: the first function calls result (1) and is dependant on the results of functions calls (2) to (5), and (3) is dependant on (4) and (5). You can imagine that for a very large value, e.g. $n=1000$, you have a lot more nested function calls and need lots of memory.

Also, it is quite obvious that the above function is very inefficient – `Fib_rec(2)`, for example, is called twice. Generally speaking, the number of function calls needed to calculate the function for some n grows exponentially in n which is just as bad as it sounds. This is, however, a problem of our specific algorithm and independent of the general problems with recursion. Still using a recursive algorithm, we can easily avoid doing the multiple calculations:

¹ I am somewhat oversimplifying, of course, but you should get the general idea.

```

2def Fib_rec2 (n, current=3, predec=1, prepredec=1):
    if n == 1 or n == 2: return 1
    else: if (current = n): return predec + prepredec
    else: return Fib_rec2(n, current+1, predec+prepredec, predec)

```

This leaves unaddressed the more general problem with recursion, however, i.e. that each function call allocates (reserves) a certain amount of memory which is freed again only after it has terminated. As the termination of a recursive function usually depends on a lot of other function calls, recursive functions tend to run out of memory quickly.

A *real* alternative to recursive algorithms is using an iterative algorithm – iteration is simply the idea of repeating a certain sequence of commands. In a way, recursive functions do the same but if you use iteration you do this in a more memory-friendly way. An example illustrates this best:

```

>>> def fib_iter(n):
...     if n==1 or n==2:
...         return 1
...     pre = 1
...     prepre = 1
...     for i in range(3,n):
...         pre, prepre = pre+prepre, pre
...     return pre+prepre

```

At the heart of this algorithm is a so-called Loop³ (marked in bold) which basically does the same that our recursive else-clause did in the other algorithm; it sums up two Fibonacci-numbers to calculate the next one. It contrasts with the first recursive algorithm in that it is bottom-up and not top-down and with the second recursive algorithm in that it only needs constant space, independent of n. This algorithm, therefore, does not run out of memory and is a lot faster than its recursive semi-twin.

It would be, however, false to say that iterative algorithms always only need constant memory. For more complex problems, additional memory (in the form of a so-called stack) is needed. The details, however, are obviously beyond the scope of this short paper and our reading group.⁴ If you want I might squeeze in a short demonstration of this fact involving

2 It does not matter if you do not understand why this works or how it works. The general idea is that you do not work to your goal top-to-bottom, i.e. starting at n and then terminating once you reach the base clauses building up a huge binary tree but that you start at the bottom and remember the already calculated numbers so that you only calculate every Fibonacci number once.

3 Strictly speaking, For-Loops (which need to know the number of iterations once they start) are less-expressive than While-Loops. Every recursive algorithm can be transformed into a 'While-Iterative' algorithm. I think we can safely ignore these details here.

4 Learning a programming language, however, is really worth the effort. There are a lot of good introductions

traversing a binary tree (this actually has some relation to syntax if you think of it as linearization) which might hint at why recursive algorithms are used at all – often, it is non-trivial to come up with a non-recursive alternative and the non-recursive alternatives usually are a lot harder to understand.

I guess this is a good point to stop and to ask what relevance all of this has for our discussion.

- At the level of abstraction mathematics operates (in Marr's terms the computational level specifying input-output correlations, i.e. *what* actually is calculated), 'recursive' and 'iterative' are synonyms.
- At the level of abstraction of computer science (the algorithmic level), there is a principled distinction:
 - Recursive algorithms involve functions re-calling themselves and are therefore very memory-demanding.
 - Iterative algorithms involve a Loop-Construct and are less memory demanding, even though they also might need additional memory.
 - Recursive algorithms are usually a lot easier to understand than their iterative siblings but less efficient.
- (Now I become more speculative) Whether at the level of abstraction neuro-science operates (the implementational level) this distinction can be sustained is not at all obvious – this has to do with the general problem of indeterminacy; if you want a short and really good illustration of this problem, take a look at Harman 1973⁵. If you want a longer and probably confused discussion, take a look at the final chapters of my thesis.
- For this reason I am highly sceptical of the alleged iteration-vs-recursion divide in linguistics and cognitive science for it is not clear to me that it has some substantial basis. Which level of abstraction is assumed? What precisely do the different parties mean by recursion/iteration (note the nominalization)? And most important: What is the empirical content of this alleged distinction which, to reiterate, stems from the formal sciences (mathematics and theoretical computer science)?

for linguists, e.g. Bird et al 2009, Natural Language Processing with Python, freely available at <http://www.nltk.org/book>

5 Harman, Gilbert (1973): Quine's Grammar. In Hahn/Schilpp (eds) (1986), The Philosophy of W.V.Quine. Open Court, pp 165–180.