Contents

Series and Recursion

The problem we were dealing with provided this series, which we solved via a `for` loop:

$$g\left(k\right)=\frac{\sqrt{2}}{2}\cdot\frac{\sqrt{2+\sqrt{3}}}{3}\frac{\sqrt{2+\sqrt{3+\sqrt{4}}}}{4}\cdot\ldots\frac{\sqrt{2+\sqrt{3+\ldots+\sqrt{k}}}}{k}$$

let's modify this for the sake of discussion:

$$h\left(k\right)=\frac{\sqrt{2}}{2}\cdot\frac{\sqrt{3+\sqrt{2}}}{3}\cdot\frac{\sqrt{4+\sqrt{3+\sqrt{2}}}}{4}+\ldots+\frac{\sqrt{k+\sqrt{k-1+\ldots\sqrt{3+\sqrt{2}}}}}{k}$$

Essentially this difference obviates the need to use the `reversed` function in the previous `for` loop.

The function h can be expressed by the series:

$$h\left(k\right)=\prod_{i=2}^{k}\left(\frac{f_k}{k}\right)\quad:\quad f_i=\sqrt{i+f_{i-1}}$$

Within *Python* this isn't actually to difficult to express, something to the effect of the following would be sufficient:

```
from sympy import *
def h(k):
    if k > 2:
    return f(k) * f(k-1)
    else:
        return 1

def f(i):
    expr = 0
    if i > 2:
        return sqrt(i + f(i -1))
    else:
        return 1
```

This is a very natural way to define series and sequencies and can be contrasted with a for loop.

```
from sympy import *
def h(k):
    k = k + 1 # OBOB
    l = [f(i) for i in range(1,k)]
    return prod(l)

def f(k):
```

```
    expr = 0
    for i in range(2, k+2):
        expr = sqrt(i + expr, evaluate=False)
    return expr/(k+1)
```

If a function can be defined using a loop it can always be defined via recursion as well, [1] the question is which will be more appropriate, generally the process for determining which is more appropriate is to the effect of:

1. Write the problem in a way that is easier to write or is more appropriate for demonstration
2. If performance is necessary then consider restructuring recursion for loops
    - In languages such ***R*** and *Python* :snake:, loops are usually faster, although there may be exceptions to this.

. ### Some Functions are more difficult to express with Recursion in *Python*

Consider the function $g\left(k\right)$:

$$\begin{aligned} g\left(k\right) &= \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{3}}}{3} \frac{\sqrt{2+\sqrt{3+\sqrt{4}}}}{4} \cdot \ldots \frac{\sqrt{2+\sqrt{3+\ldots+\sqrt{k}}}}{k} \\ &= \prod_{i=2}^{k} \left(\frac{f_k}{k}\right) \quad : \quad f_i = \sqrt{i+f_{i+1}} \end{aligned}$$

Observe that the sequence now *looks* forward, not back, to solve using a `for` loop the `reversed` function dealt with that issue quite nicely and the technique corresponding to $h\left(k\right)$ was able to be implemented, however mathematically this can be a little clumsy to illustrate, for example the following rewrite would allow this to occur:

$$g\left(k\right) = \prod_{i=2}^{k} \left(\frac{f_k}{k}\right) \quad : \quad f_i = \sqrt{(k-i)+f_{k-i-1}}$$

Now the function could be performed recursively in *Python* in a similar way, in particular, by performing something like this:

```
from sympy import *
def h(k):
    if k > 2:
    return f(k) * f(k-1)
    else:
        return 1

def f(i, k):
    if k > i:
        return 1

    if i > 2:
    return sqrt((k-i) + f(k - i -1))
    else:
    return 1
```

[1] This Stack Answer puts it well: > There's a simple ad hoc proof for this. Since you can build a Turing complete language using strictly iterative structures and a Turning complete language using only recursive structures, then the two are therefore equivalent.

The thing that's worth noting though is that it is much tricker to implement the recursive approach for $g(k)$ than $h(k)$, whereac the approach with the loop doesn't really change but for the inclusion of the `reversed()` function.

```
from sympy import *
def h(k):
    k = k + 1 # OBOB
    l = [f(i) for i in range(1,k)]
    return prod(l)

def f(k):
    expr = 0
    for i in reversed(range(2, k+2)):
        expr = sqrt(i + expr, evaluate=False)
    return expr/(k+1)
```