Contents

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

```
from __future__ import division
from sympy import *
x, y, z, t = symbols('x y z t')
k, m, n = symbols('k m n', integer=True)
f, g, h = symbols('f g h', cls=Function)
init_printing()
init_printing(use_latex='mathjax', latex_mode='equation')


import pyperclip
def lx(expr):
    pyperclip.copy(latex(expr))
    print(expr)

import numpy as np
import matplotlib as plt
```

First let's see if we can just print the numbers

```
expr = 0
k = 7
for i in reversed(range(k)):
    print(i)
```

```
6
5
4
3
2
1
0
```

Right now let's start building an expression to get each term of the sequence:

```
expr = 0
k = 1
for i in reversed(range(2, k+2)):
```

```
    expr = sqrt(i + expr)
expr/(k+1)
```

$$\frac{\sqrt{2}}{2} \tag{1}$$

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

$$\frac{\sqrt{\sqrt{3}+2}}{3} \tag{2}$$

```
expr = 0
k = 3
for i in reversed(range(2, k+2)):
    expr = sqrt(i + expr)
expr/(k+1)
```

$$\frac{\sqrt{2+\sqrt{5}}}{4} \tag{3}$$

So it seems to work but it's decided to simplify the surd which will get confusing, so avoid that we will pass the `evaluate = False` paraemeter as described in the docs

```
expr = 0
k = 3
for i in reversed(range(2, k+2)):
    expr = sqrt(i + expr, evaluate=False)
expr/(k+1)
```

$$\frac{\sqrt{2+\sqrt{\sqrt{4}+3}}}{4} \tag{4}$$

Now let's wrap the term generator into a function:

```
k = 3
def surd_seq(k):
    expr = 0
    for i in reversed(range(2, k+2)):
        expr = sqrt(i + expr, evaluate=False)
    return expr/(k+1)
surd_seq(3)
```

$$\frac{\sqrt{2+\sqrt{\sqrt{4}+3}}}{4} \tag{5}$$

Ok so that seems right so now let's instead look at a lot of the terms using list comprehension:

```
[surd_seq(i) for i in range(1, 7)]
```

$$\left[\frac{\sqrt{2}}{2}, \frac{\sqrt{\sqrt{3}+2}}{3}, \frac{\sqrt{2+\sqrt{\sqrt{4}+3}}}{4}, \frac{\sqrt{2+\sqrt{\sqrt{\sqrt{5}+4}+3}}}{5}, \frac{\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{6}+5}+4}+3}}}{6}, \frac{\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{7}+6}+5}+4}+3}}}{7}\right] \tag{6}$$

Unfourtunately there is no simple way, that I am aware of, to have it print the terms in the order we would expect, which is unfourtunate.

Now that we have the sequence, we can turn it into a series with the prod function:

```
l = [surd_seq(i) for i in range(1,7)]
prod(l)

def surd_ser(k):
    k = k + 1 # OBOB
    l = [surd_seq(i) for i in range(1,k)]
    return prod(l)

surd_ser(2)/surd_seq(1)
```

$$\frac{\sqrt{\sqrt{3}+2}}{3} \tag{7}$$

So now this seems to check out so let's go onto the actual problem and look at $\mathrm{surd_s er}\,(100)$, unfourtunately we won't be able to use `simplify` here because each term relies on the previous term and hence this will not scale in polynomial time, rather it will scale in exponential time and it's going to be very resource intensive.

To get an idea of what I mean, compare the time to evaluate 7 and 14 terms:

```
simplify(surd_ser(7))
```

$$\frac{\sqrt{2}\sqrt{2+\sqrt{5}}\sqrt{2+\sqrt{\sqrt{\sqrt{5}+4}+3}}\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{6}+5}+4}+3}}\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{7}+6}+5}+4}+3}}\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{2\sqrt{2}+7}+}}}}}}{40320} \tag{8}$$

```
simplify(surd_ser(14))
```

$$\sqrt{2}\sqrt{2+\sqrt{5}}\sqrt{2+\sqrt{\sqrt{\sqrt{5}+4}+3}}\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{6}+5}+4}+3}}\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{7}+6}+5}+4}+3}}\sqrt{2+\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{11}+7}+$$

(9)

Instead we will need to approach this numerically, again though we will have performance limitations, consider for example, evaluating this expression to 40:

```
N(surd_ser(40))
```

$$1.18026611043774 \cdot 10^{-37} \tag{10}$$

```
Now consider evaluating it to 50:
```

```
N(surd_ser(50))
```

$$4.05422002157275 \cdot 10^{-51} \tag{11}$$

It takes a considerably longer time, instead we need to look at this from a functional perspective and rewrite the function:

```
k = 3
def surd_seq(k, numerical = False):
    expr = 0
    if numerical:
        for i in reversed(range(2, k+2)):
            expr = N(sqrt(i + expr, evaluate=False))
    else:
        for i in reversed(range(2, k+2)):
            expr = sqrt(i + expr, evaluate=False)
    return expr/(k+1)
surd_seq(3)

def surd_ser(k, numerical = False):
    if numerical:
        k = k + 1 # OBOB
        l = [surd_seq(i, numerical = True) for i in range(1,k)]
    else:
        k = k + 1 # OBOB
```

```
        l = [surd_seq(i, numerical = False) for i in range(1,k)]
    return prod(l)

surd_ser(2, True)
```

$$0.45534180126148 \tag{12}$$

Now we will be able to evaluate this much more easily:

```
surd_ser(100, numerical = True)
```

$$6.83824783929329 \cdot 10^{-129} \tag{13}$$

```
This clearly demonstrates that the value is less than $10^-100$
```

Numerical Performance

Sympy does have some limitations, let's demonstrate this by pushing the evaluation to it's limits.

```
surd_ser(500, numerical = True)
```

$$1.28458730802734 \cdot 10^{-977} \tag{14}$$

By rewriting this in `numpy` we can get even better performance, so first import `numpy`:

```
import numpy
np.sqrt(3)
```

$$1.7320508075688772 \tag{15}$$

```
Now it's just a matter of swapping `sqrt()` for `np.sqrt`.
```

```
k = 3
def surd_seq(k, numerical = False):
    expr = 0
    if numerical:
        for i in reversed(range(2, k+2)):
            expr = np.sqrt(i + expr, )
    else:
```

```
        for i in reversed(range(2, k+2)):
            expr = sqrt(i + expr, evaluate=False)
    return expr/(k+1)
surd_seq(3)

def surd_ser(k, numerical = False):
    if numerical:
        k = k + 1 # OBOB
        l = [surd_seq(i, numerical = True) for i in range(1,k)]
    else:
        k = k + 1 # OBOB
        l = [surd_seq(i, numerical = False) for i in range(1,k)]
    return prod(l)

surd_ser(2, True)
```

$$0.4553418012614796 \tag{16}$$

```
surd_ser(500, numerical = True)
```

$$0.0 \tag{17}$$

Now `0` doesn't seem helpful, but in an engineering context it is if you bear in mind that `numpy` uses `double` precision (as in double a `float32` which is `float64`), this has an accuracy of 16 digits.

So this answer tells us that this series reaches 0, to a `double` precision, by 500 iterations.

Symbolically this series is limited by 0.

Choosing the Tool

What this example demonstrates is the need to choose the right tool in *Python*, compared to a language like *Mathematica*, *Python* requires the user to be mindful of what they are trying to do beforehand as opposed to after, generally:

Library	Use Case
SymPy	Necessary for Symbolic Algebra
SymPy `N()`	When arbitrary numerical precision is required despite performance
NumPy	When High Performance Numerical Solutions are required up to 15 dp

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 $g(k)$:

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

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 $h(k)$ 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:

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

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
```

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):
```

[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.

```
    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)
```