Programming for Science
Workshop 9

This workshop gives you some practice with recursion.

1. Implement the recursive function to calculate $n!$ from lectures. Include the printing that shows how the function works and run the function in a few test cases to make sure that you thoroughly understand how it operates.

If you have implemented a test to guard against negative arguments, disable it and find out what happens if you try to calculate the factorial of -1.

2. Write and test a function `rsum` that *recursively* calculates the sum of the integers $0, \ldots, n$ (recall that this was talked about in lectures).

3. Write and test a function `lsum` that recursively finds the sum of the elements of a list by adding the first and last elements of the list and then calling itself to sum the remainder of the list. That is, if the list L has N terms, the sum could be written as:

$$S = L_0 + \sum_{n=1}^{N-2} L_n + L_{N-1}$$

(using zero-based indices). The recursive function, should sum L_0 and L_{N-1} and call itself to find $\sum_{n=1}^{N-2} L_n$.

4. Yet another look at the Fibonacci series. Recall that the nth Fibonacci number is defined as

$$F_n = F_{n-1} + F_{n-2} \qquad \text{with } F_0 = 0 \text{ and } F_1 = 1$$

This could be written in a recursive style as

$$\text{Fib}(n) = \begin{cases} \text{Fib}(n-1) + \text{Fib}(n-2) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

where $\text{Fib}(n)$ is a function that returns the n Fibonacci number.

Write and test a recursive function to calculate Fibonacci numbers. Modify your function to print out each call so that you can see what it is doing.

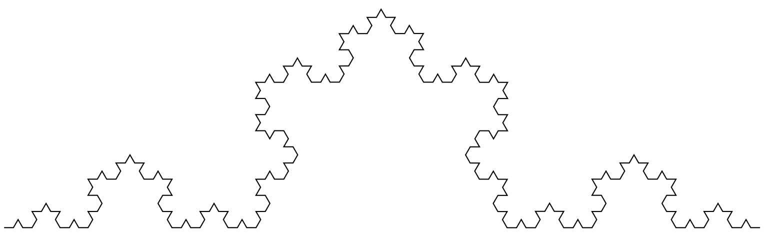
Notice that your function calculates many of the intermediate results several times. This can be avoided by "remembering" or storing the intermediate results. Then each time a new result is needed, check whether it's been calculated previously and only calculate it if it hasn't. This is called *memoisation.* A dictionary is a useful data structure for achieving this. For example, you might initialise the memory for the results as:

```
memory = {0:0, 1:1 }   #  Initialise memory with F(0) and F(1)
```

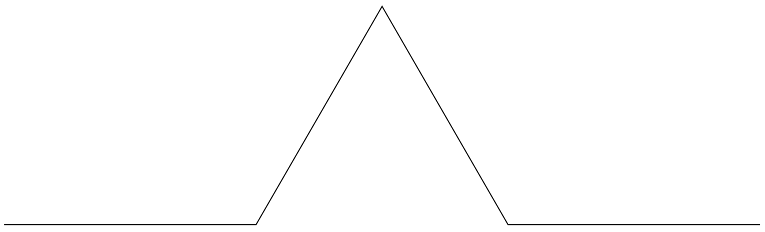
Modify your function so that it keeps a dictionary of the intermediate results and uses them whenever possible. Verify by printing out the calls that it really is using the dictionary.

NB: Make sure that you pass the dictionary as an argument to the function: don't use global variables here!

5. The Koch curve, pictured on the right, is a recursively defined curve with some interesting properties. In this exercise you will write a turtle program to draw a Koch curve.



As you can see the curve has a recursive construction. In the base case a straight line (of length L) is drawn. In the general case, the line is divided into four segments, each of length $L/3$ and a copy of the curve at $1/3$ scale is drawn in each of these. They are oriented to each other as shown.



To start and get a feeling for the curve, write a function, using the `exswampy` TurtleGraphics, to draw the first level of a recursive construction as shown on the right.

The algorithm for drawing a Koch curve with initial segment length `L` is:

(a) Draw a Koch curve with length `L/3`
(b) Turn left 60°
(c) Draw a Koch curve with length `L/3`
(d) Turn right 120°
(e) Draw a Koch curve with length `L/3`
(f) Turn left 60°
(g) Draw a Koch curve with length `L/3`

Write and test a recursive function `koch(t, n, L)` that uses turtle `t` to draw a Koch curve with initial segment length `L` and `n` levels. While you are testing it, I suggest you only use 3 recursion levels otherwise it will take a very long time to draw! Try the intricate detailed version when you're sure the code works.