Droste Effect:
The picture is in the picture which is in the picture …

CMPT 120
Lecture 20 – Graphics and Animation
Python – Implementing and Visualizing Recursion
Last Lectures

• Solved the *chocolate chip cookie problem* using *Turtle* + *Loops* + *Functions* + *Tuples*

• Summarized various topics related to *functions* using
  • *OperationsOnList.py* posted on our course website
  • And the Python Visualizer

• Introduced (very briefly) a new kind of algorithm: *Recursion*

• We had our *Practice Exam 5*: Feedback
Today’s Menu

• Investigate **Recursion**
• Solve problems using **recursion**
• **Visualize** the execution of our **recursive** solutions
Review: Recursion - Definition

From our Readings (16.1)

- **Recursion** is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially.

- **Recursion** occurs when an **object** or a **process** is defined in terms of itself (or a version of itself).
Review: Recursion in the real world

- Russian dolls
- Searching for a word in a dictionary

Source: http://www.eslstation.net/ESL310L/310L_dict.htm
Recursion in the mathematical world

- Factorials

The factorial rule:
- $n! = n \times (n-1)!$ if $n > 1$
- $n! = 1$ if $n = 1$ or $0$

Source: https://www.quora.com/How-is-recursion-used-to-compute-the-factorial-function

- Fractals

The Sierpinski triangle is a confined recursion of triangles that form a fractal

See section 16.6 in our online e-textbook

Source: https://en.wikipedia.org/wiki/Recursion
Recursion in the software world

• So far, when solving problems using software (algorithms), we have achieved iteration by using **iterative** statements -> **loops**
  • By putting statements we wanted to execute more than once in a **loop**
Recursion in the software world

- **Recursion** is an **elegant** way of solving problems where we achieve iteration by putting statements we want to execute more than once in a **function** and having this **function** calling itself.

```
Factorial(5)
return 5 * Factorial(4) = 120
return 4 * Factorial(3) = 24
return 3 * Factorial(2) = 6
return 2 * Factorial(1) = 2
1
```
Let’s give Recursion a go!

Step 1 - Problem Statement
• Implement a **factorial** function using **recursion**

The factorial rule:
• $n! = n \times (n-1)!$ if $n > 1$
• $n! = 1$ if $n = 1$ or $0$

Step 2 – Design?

Step 3 – Implementation

Step 4 - Testing

Source: https://en.wikipedia.org/wiki/File:Man-scratching-head.gif
Step 2 – Design

- Design **recursive function** in two parts called **case**:  
  
  1. **Base case**: Version of the problem that is small enough to be solved trivially  
     - **Function** stops calling itself and we start “recursing up”.  
     
     The factorial rule:  
     - $n! = n \times (n-1)!$ if $n > 1$  
     - $n! = 1$ if $n = 1$ or $0$  
  
  2. **Recursive case**: Break a problem down into smaller version of itself to eventually become the base case  
     - **Function** calling itself with diminishing argument(s)  
     
     The factorial rule:  
     - $n! = n \times (n-1)!$ if $n > 1$  
     - $n! = 1$ if $n = 1$ or $0$
Using an example, we observe how we solve the factorial problem by hand looking for the pattern. Then we implement the pattern!
Step 3 – Implementation

And let’s **visualize** the execution of our **factorial function** as we are performing **Step 4 – Testing**
Let’s try again!

Step 1 - Problem Statement

- Remember the **palindrome function** of Practice Exam 4?
- Solve the **palindrome problem recursively**
Step 2 – Design

Step 2 - Design
• Using an example: kayak

Step 3 – Implementation

Step 4 - Testing
Next Lecture

• Use Computer Graphics, turtle and recursion to draw trees