Eric Roberts
CSCI 121

Handout #56
November 26, 2018

Thinking Recursively

Thinking Recursively

Eric Roberts
CSCI 121
November 26, 2018

How Long is the Coast of England?

- The first widely circulated paper about fractals was a 1967 article in *Science* by Benoit Mandelbrot that asked the innocuous question, "How long is the coast of England?"
- The point that Mandelbrot made in the article is that the answer depends on the measurement scale, as these images from Wikipedia show.
- This thought-experiment serves to illustrate the fact that coastlines are ***fractal*** in that they exhibit the same structure at every level of detail.

Review: The Recursive Paradigm

- Most recursive functions you encounter in an introductory course have bodies that fit the following general pattern:

```
if test for a simple case:
    Compute and return the simple solution without using recursion.
else:
    Divide the problem into one or more subproblems that have the same form.
    Solve each of the problems by calling this method recursively.
    Return the solution from the results of the various subproblems.
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
  1. Identify ***simple cases*** that can be solved without recursion.
  2. Find a ***recursive decomposition*** that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.

Review: Recursive Checklist

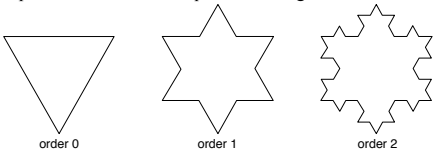
- Does your recursive implementation begin by checking for simple cases?
- Have you solved the simple cases correctly?
- Does your recursive decomposition make the problem simpler?
- Does the simplification process eventually reach the simple cases, or have you left out some of the possibilities?
- Do the recursive calls in your method represent subproblems that are truly identical in form to the original?
- Do the solutions to the recursive subproblems provide a complete solution to the original problem?

Graphical Recursion

- Recursion comes up in certain graphical applications, most notably in the creation of ***fractals***, which are mathematical structures that consist of similar figures repeated at various different scales. Fractals were popularized in a 1982 book by Benoit Mandelbrot entitled *The Fractal Geometry of Nature*.
- One of the simplest fractal patterns to draw is the ***Koch fractal***, named after its inventor, the Swedish mathematician Helge von Koch (1870-1924). The Koch fractal is sometimes called a ***snowflake fractal*** because of the beautiful, six-sided symmetries it displays as the figure becomes more detailed, as illustrated in the following diagram:

Drawing Koch Fractals

- The process of drawing a Koch fractal begins with an equilateral triangle, as shown in the diagram on the lower left.
- From the initial position (which is called a fractal of ***order 0***), each higher fractal order is created by replacing each line segment in the figure by four segments that connect the same endpoints but include an equilateral wedge in the middle.



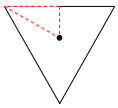
The figure on the previous slide is the Koch fractal of order 4.

The `GSnowflake` Class

- The snowflake fractal is a polygon, which suggests that the best strategy for implementing it would be to create a class called `GSnowflake` that extends `GPolygon`.
- The reference point for the `GSnowflake` class is presumably the center of the order-0 triangle, which means that the vertex at the left edge has the following coordinates:

```
x = -edge / 2
y = -edge / 2 / math.sqrt(3)
```

The `y` value reflects the geometry of the 30-60-90 triangle:



The `GSnowflake` of Order-0

- As a starting point, the following class creates an order-0 fractal snowflake:

```
class GSnowflake(GPolygon):

    def __init__(self, edge):
        GPolygon.__init__(self)
        x = -edge / 2
        y = -edge / 2 / math.sqrt(3)
        self.addVertex(x, y)
        self.addPolarEdge(edge, 0)
        self.addPolarEdge(edge, -120)
        self.addPolarEdge(edge, +120)
```

- The next step is to add a new argument to the constructor that specifies the order of the fractal.

Adding in the Recursion

- The first step in completing the definition is to replace each of the calls to a new method called `addFractalEdge` that also takes the order:

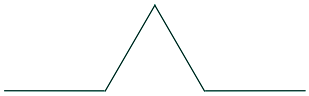
```
class GSnowflake(GPolygon):

    def __init__(self, edge, order):
        GPolygon.__init__(self)
        x = -edge / 2
        y = -edge / 2 / math.sqrt(3)
        self.addVertex(x, y)
        self.addFractalEdge(edge, 0, order)
        self.addFractalEdge(edge, -120, order)
        self.addFractalEdge(edge, +120, order)
```

- The next step is to write `addFractalEdge`.

Adding a Fractal Edge

- The simple case for `addFractalEdge` occurs when the order is 0, in which case the edge is just a straight line in the direction given by the second argument to `addPolarEdge`.
- The recursive case is to replace each straight line like

with a jagged line like



where each segment in the jagged line is a fractal line of the next lower order.

Exercise: Code `addFractalEdge`

```
class GSnowflake(GPolygon):

    def __init__(self, edge, order):
        GPolygon.__init__(self)
        x = -edge / 2
        y = -edge / 2 / math.sqrt(3)
        self.addVertex(x, y)
        self.addFractalEdge(edge, 0, order)
        self.addFractalEdge(edge, -120, order)
        self.addFractalEdge(edge, +120, order)

    def addFractalEdge(self, r, theta, order):
```

Searching in a Branching Structure

- Recursive strategies are particularly appropriate when the solution requires exploring a range of possibilities at each of a series of choice points, much like in solving a maze—an application that I'll cover on Wednesday.
- The primary advantage of using recursion in these problems is that doing so dramatically simplifies the bookkeeping. Each level of the recursive algorithm considers one choice point. The historical knowledge of what choices have already been tested and which ones remain for further exploration is maintained automatically in the execution stack.

Exercise: Generating Subsets

- Write a function

```
def subsets(s):
```

  that generates a list showing all subsets of the set `s`.

- For example, calling `subsets({1, 2, 3})` should return a list containing the following eight sets, in some order:

```
set()    {1}    {1, 2}    {1, 2, 3}
         {2}    {1, 3}
         {3}    {2, 3}
```

- The recursive insight is that you can form all subsets by
  - Removing an element
  - Finding all subsets of the remaining elements
  - Creating a new list containing those subsets plus all those that also contain the removed element

The Subset Tree