The Recursive Paradigm

Eric Roberts
CSCI 121
October 14, 2019

Fractals

• Over the last several decades, there has been extensive public interest in fractals, which are mathematical structures with the property of self-similarity in the sense that they consist of similar figures that repeat at different scales.

• Although fractals have a long history dating back to the 17th century, the modern enthusiasm for fractals was sparked by the book *The Fractal Geometry of Nature* by Benoit Mandelbrot, which appeared in 1982.

How Long is the Coast of England?

• The first widely circulated paper about fractals was a 1967 article in *Science* by Mandelbrot that asked the seemingly 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.

The Towers of Hanoi

In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

—Henri de Parmeille, *La Nature*, 1883

Solving the Towers of Hanoi

```python
def moveTower(n, start, finish, tmp):
    if (n == 1):
        moveSingleDisk(start, finish)
    else:
        moveTower(n-1, start, tmp, finish)
        moveSingleDisk(start, finish)
        moveTower(n-1, tmp, finish, start)
```

The Recursive “Leap of Faith”

• The purpose of going through the complete decomposition of the Towers of Hanoi problem is to convince you that the process works and that recursive calls are in fact no different from other method calls, at least in their internal operation.

• The danger with going through these details is that it might encourage you to do the same when you write your own recursive programs. As it happens, tracing through the details of a recursive program almost always makes such programs harder to write. Writing recursive programs becomes natural only after you have confidence in the process.

• As you write a recursive program, it is important to believe that any recursive call will return the correct answer as long as the arguments define a simpler subproblem. Believing that to be true—even before you have completed the code—is called the recursive leap of faith.
The Recursive Paradigm

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

1. Test for a simple case:
   - Compute and return the simple solution without using recursion.

2. Else:
   - Divide the problem into one or more subproblems that have the same form. Solve each of the problems by calling the 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.

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

• 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.

Drawing a Snowflake Polygon

• Although you can draw the snowflake fractal using a series of GLine objects, it makes more sense to use a GPolygon, which makes the entire fractal a single object.
• The reference point for the snowflake polygon is presumably the center of the order-0 triangle, which means that the vertex at the left edge has the following coordinates:

\[
\begin{align*}
x &= -\text{edge} / 2 \\
y &= -\text{edge} / (2 \cdot \text{math.sqrt}(3))
\end{align*}
\]

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

The Fractal Snowflake of Order-0

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

```
def createSnowflake(edge):
    snowflake = GPolygon()
    x = -edge / 2
    y = -edge / (2 \cdot \text{math.sqrt}(3))
    snowflake.addVertex(x, y)
    snowflake.addPolarEdge(edge, 0)
    snowflake.addPolarEdge(edge, 120)
    snowflake.addPolarEdge(edge, -120)
```

• The next step is to add a new parameter to createSnowflake that specifies the order of the fractal.
• In the new version, you must also replace each of the calls to addPolarEdge with a call to a function 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 segment like

```
with a jagged edge like
```

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

Exercise: Code `addFractalEdge`

```python
def addFractalEdge(poly, r, theta, order):
    ""
    Adds a fractal edge to the polygon whose length is r and whose initial direction is theta. If order is 0, the fractal edge is a straight line. Otherwise, the edge is composed of four fractal edges of the next lower order.
    ""
```

Generating Mondrian-Style Paintings

![Mondrian Style Paintings](image)

Fig. 11: Three real Mondrian paintings, and three samples from our targeting function. Can you tell which is which?


Mondrian Decomposition

```
# File: Mondrian.java

""
This program draws a recursive Mondrian style picture by recursively subdividing the plane.
""

```java

from pgl import GWindow, GLine
import random

# Constants

GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 500
MIN_AREA = 10000
MIN_EDGE = 20

def Mondrian():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    subdivide(gw, 0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT)

# Code for the Mondrian Program

```

Code for the Mondrian Program

```
# At each level, subdivide first checks for the simple case, which is when the size of the rectangular canvas is too small to subdivide (i.e., when the area is less than MIN_AREA). In the recursive case, the method splits the canvas along its longest dimension by choosing a random dividing line that leaves at least MIN_EDGE pixels on each side. The program then uses a divide-and-conquer strategy to subdivide the two rectangles.

def subdivide(gw, x, y, width, height):
    if width * height >= MIN_AREA:
        if width > height:
            dx = random.uniform(MIN_EDGE, width - MIN_EDGE)
            gw.add(GLine(x + dx, y, x + dx, y + height))
            subdivide(gw, x, y, dx, height)
            subdivide(gw, x + dx, y, width - dx, height)
        else:
            dy = random.uniform(MIN_EDGE, height - MIN_EDGE)
            gw.add(GLine(x, y + dy, x + width, y + dy))
            subdivide(gw, x, y, width, dy)
            subdivide(gw, x, y + dy, width, height - dy)
```

Code for the Mondrian Program
ColorMondrian Decomposition