Recursion

To read for Monday, October 14.

This handout offers an introduction to the idea of recursion, which is a fundamental idea in computer science. This material goes beyond what is covered in the book, but I think it is important to include some of this additional material in CSCI 121.

1. The Towers of Hanoi

The classic example of recursion comes from a simple puzzle that has come to be known as the Towers of Hanoi. Invented by French mathematician Édouard Lucas in the 1880s, the Towers of Hanoi puzzle quickly became popular in Europe. Its success was due in part to the legend that grew up around the puzzle, which was described as follows in La Nature by the French mathematician Henri de Parville (as translated by the mathematical historian W. W. R. Ball):

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 Brahmans alike will crumble into dust, and with a thunderclap the world will vanish.

Over the years, the setting has shifted from India to Vietnam, but the puzzle and its legend remain the same.

As far as I know, the Towers of Hanoi puzzle has no practical use except one: teaching recursion to computer science students. In that domain, it has tremendous value because the solution involves nothing other than recursion. In contrast to most recursive algorithms that arise in response to real-world problems, the Towers of Hanoi problem has no extraneous complications that might interfere with your understanding and keep you from seeing how the recursive solution works. Because it works so well as an example, the Towers of Hanoi is included in most textbooks that treat recursion and has become—much like the “hello, world” program—part of the cultural heritage that computer scientists share.

In commercial versions of the puzzle, the 64 golden disks of legend are replaced with eight wooden or plastic ones, which make the puzzle considerably easier to solve (not to mention less expensive). The initial state of the puzzle looks like this:
At the beginning, all eight disks are on spire A. Your goal is to move the eight disks from spire A to spire B, while adhering to the following rules:

- You can only move one disk at a time.
- You are not allowed to move a larger disk on top of a smaller one.

**Framing the problem**

In order to apply recursion to the Towers of Hanoi problem, you must first frame the problem in more general terms. Although the ultimate goal is to move eight disks from A to B, the recursive decomposition of the problem will involve moving smaller subtowers from spire to spire in various configurations. In the more general case, the problem you need to solve is moving a tower of a given height from one spire to another, using the third spire as a temporary repository. To ensure that all subproblems fit the original form, your recursive procedure must therefore take the following parameters:

1. The number of disks to move
2. The name of the spire where the disks start out
3. The name of the spire where the disks should finish
4. The name of the spire used for temporary storage

The function header for `moveTower` therefore looks like this:

```python
def moveTower(n, start, finish, tmp):
```

To move the eight disks in the example, the initial call is

```python
moveTower(8, "A", "B", "C")
```

This method call corresponds to the English command “Move a tower of size 8 from spire A to spire B using spire C as a temporary repository.” As the recursive decomposition proceeds, `moveTower` will be called with different arguments that move smaller towers in various configurations.

**Finding a recursive strategy**

Now that you have a formulation in Python for solving the puzzle, you can return to the problem of finding a strategy for moving a large tower. To apply recursion, you must first make sure that the problem meets the following conditions:

1. *There must be a simple case.* In this problem, the simple case occurs when *n* is equal to 1, which means that there is only a single disk to move. As long as you don’t
violate the rules by placing a larger disk on top of a smaller one, you can move a single disk in a single operation.

2. **There must be a recursive decomposition.** In order to implement a recursive solution, it must be possible to break the problem down into simpler problems in the same form as the original. This part of the problem is harder and will require closer examination.

To see how solving a simpler subproblem helps solve a larger problem, it helps to go back and consider the original example with eight disks.

![Diagram of three spires with disks](image)

The goal here is to move eight disks from spire A to spire B. You need to ask yourself how it would help if you could solve the same problem for a smaller number of disks. In particular, you should think about how being able to move a stack of seven disks would help you to solve the eight-disk case.

If you think about the problem for a few moments, it becomes clear that you can solve the problem by dividing it into these three steps:

1. Move the entire stack consisting of the top seven disks from spire A to spire C.
2. Move the bottom disk from spire A to spire B.
3. Move the stack of seven disks from spire C to spire B.

Executing the first step takes you to the following position:

Once you have gotten rid of the seven disks on top of the largest disk, the second step is simply to move that disk from spire A to spire B, which results in the following configuration:

Once you have gotten rid of the seven disks on top of the largest disk, the second step is simply to move that disk from spire A to spire B, which results in the following configuration:
All that remains is to move the tower of seven disks back from spire C to spire B, which is again a smaller problem of the same form. This operation is the third step in the recursive strategy and leaves the puzzle in the desired final configuration:

That’s it! You’re finished. You’ve reduced the problem of moving a tower of size eight to one of moving a tower of size seven. More importantly, this recursive strategy generalizes to towers of size $N$, as follows:

1. Move the top $N - 1$ disks from the start spire to the temporary spire.
2. Move a single disk from the start spire to the finish spire.
3. Move the stack of $N - 1$ disks from the temporary spire back to the finish spire.

At this point, it is hard to avoid saying to yourself, “Okay, I can reduce the problem to moving a tower of size $N - 1$, but how do I accomplish that?” The answer, of course, is that you move a tower of size $N - 1$ in precisely the same way. You break that problem down into one that requires moving a tower of size $N - 2$, which further breaks down into moving a tower of size $N - 3$, and so forth, until there is just one disk to move. Psychologically, however, the important thing is to avoid asking that question altogether. The recursive leap of faith should be sufficient. You’ve reduced the scale of the problem without changing its form. That’s the hard work. All the rest is bookkeeping, and it’s best to let the computer take care of that.

Once you have identified the simple cases and the recursive decomposition, all you need to do is plug them into the standard recursive paradigm, which results in the following pseudocode procedure:

```python
def moveTower(n, start, finish, tmp):
    if n == 1:
        Move a single disk from start to finish.
    else:
        Move a tower of size $n - 1$ from start to tmp.
        Move a single disk from start to finish.
        Move a tower of size $n - 1$ from tmp to finish.
```
The `moveTower` code itself looks like this:

```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 `moveSingleDisk` function takes whatever actions are needed to model the action of moving a single disk from one tower to another. If your goal is simply to list the moves on the console, `moveSingleDisk` would look like this:

```python
def moveSingleDisk(start, finish):
    print("Move a disk from " + start + " to " + finish + ".")
```  

2. Graphical recursion

Some of the most exciting applications of recursion use graphics to create intricate pictures in which a particular motif is repeated at many different scales. The remainder of this chapter offers a few examples of graphical recursion that make use of the `GWindow` and `GObject` classes introduced in Chapter 3 and expanded in Chapter 5.

An example from computer art

In the early part of the twentieth century, a controversial artistic movement arose in Paris, largely under the influence of Pablo Picasso and Georges Braque. The Cubists—as they were called by their critics—rejected classical artistic notions of perspective and representationalism and instead produced highly fragmented works based on simple geometrical forms. Strongly influenced by Cubism, the Dutch painter Piet Mondrian (1872–1944) produced a series of compositions based on horizontal and vertical lines. Those paintings have a recursive structure that makes them ideal candidates for computer simulation.

Suppose, for example, that you wanted to generate a Mondrian-like composition such as the following:

How would you go about designing a general strategy to create such a figure?
To understand how a program might produce such a figure, it helps to think about the process as one of successive decomposition. At the beginning, the canvas was simply an empty rectangle that looked like this:

If you want to subdivide the canvas using a series of horizontal and vertical lines, the easiest way to start is by drawing a randomly chosen line that divides the rectangle in two:

If you’re thinking recursively, the thing to notice at this point is that you now have two empty rectangular canvases, each of which is smaller in size. The task of subdividing these rectangles is the same as before, so you can perform it by using a recursive implementation of the same procedure. The overall approach is therefore to divide the entire rectangle in two, subdivide each rectangle in turn, and then put the two pieces together, as follows:

The only thing you need to complete the recursive strategy is a simple case. The process of dividing up rectangles should not go on indefinitely. As the rectangles get
smaller and smaller, you have to stop the process at some point. One approach is to look at the area of each rectangle before you start. Once the area of a rectangle falls below some threshold, you needn’t bother to subdivide it any further.

The Mondrian program in Figure 1 implements this recursive strategy, starting with the entire graphics canvas. In this program, the `subdivideCanvas` method does all the work. The arguments give the position and dimensions of the current rectangle on the canvas. At each step in the decomposition, the method checks whether that rectangle is large enough to split. If so, the method checks to see which dimension—width or height—is larger and then divides the rectangle with a vertical or horizontal line. In each

**Figure 1. Program to draw a random decomposition of the plane in a Mondrian-like style**

```java
# File: Mondrian.java

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

```from` pgl import` GWindow, GLine
import random

# Constants
GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300
MIN_AREA = 10000
MIN_EDGE = 20

def Mondrian():
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    subdivide(gw, 0, 0, GWINDOW_WIDTH, GWINDOW_HEIGHT)
    # 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 simple case, # the method does nothing. 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 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)

# Startup code
if __name__ == "__main__":
    Mondrian()
```
case, the method draws only a single line; all remaining lines in the figure are drawn by subsequent recursive calls.

**Fractals**

In the late 1970s, a researcher at IBM named Benoit Mandelbrot (1924–2010) generated a great deal of excitement by publishing a book on **fractals**, which are geometrical structures in which the same pattern is repeated at many different scales. Although mathematicians have known about fractals for a long time, there was a resurgence of interest in the subject during the 1980s, partly because the development of computers made it possible to do so much more with fractals than had ever been possible before.

One of the earliest examples of fractal figures is called the **Koch snowflake** after its inventor, Helge von Koch (1870–1924). The Koch snowflake begins with an equilateral triangle like this:

![Koch Snowflake Order 0](image)

This triangle, in which the sides are straight lines, is called the Koch snowflake of order 0. The figure is then revised in stages to generate fractals of successively higher orders. At each stage, every straight-line segment in the figure is replaced by one in which the middle third consists of an equilateral triangular bump protruding outward from the figure. Thus, the first step is to replace each line segment in the triangle with a line that looks like this:

![Koch Snowflake Order 1](image)

Applying this transformation to each of the three sides of the original triangle generates the Koch snowflake of order 1, as follows:

![Koch Snowflake Order 2](image)

If you then replace each line segment in this figure with a new line that again includes a triangular wedge, you create the order-2 Koch snowflake:
Replacing each of these line segments gives the order-3 fractal shown in the following diagram, which now looks even more like a snowflake:

because figures like the Koch snowflake are much easier to draw by computer than by hand, it makes sense to write a program that uses the graphical facilities introduced in Chapters 3 and 5 to generate this design. The Koch snowflake is always a closed polygon, so it makes sense to use the GPolygon class to represent it. Also, since the edges of the polygon are easiest to describe in terms of their length and direction, the simplest strategy is to use addVertex to set the initial vertex and then use addPolarEdge for all the others.

For example, the following function creates an order-0 snowflake fractal whose edge length is given by the parameter size and for which the reference point is the center of the triangle:

```python
def createSnowflakeOfOrder0(size):
snowflake = GPolygon()
snowflake.addVertex(-size / 2, -size / (2 * math.sqrt(3)))
snowflake.addPolarEdge(size, 0)
snowflake.addPolarEdge(size, -120)
snowflake.addPolarEdge(size, 120)
return snowflake
```

This only tricky part of this code is computing the coordinates of the first vertex as a function of the parameter size. The x coordinate of that vertex is shifted left by half the fractal size. The y coordinate is slightly more complicated but follows directly from the geometry of a 30-60-90 triangle. The first polar edge runs horizontally from left to right, the second is rotated clockwise from the horizontal axis by 120 degrees, and the third is rotated 120 degrees from the horizontal in the opposite direction.

Extending this function to draw a fractal of a higher order requires replacing the calls to addPolarEdge with calls that add a sequence of edges that form a fractal edge of the desired order. The createSnowflake function itself looks like this:
def createSnowflake(size, order):
    snowflake = GPolygon()
    snowflake.addVertex(-size / 2, -size / (2 * math.sqrt(3)))
    addFractalEdge(snowflake, size, 0, order)
    addFractalEdge(snowflake, size, -120, order)
    addFractalEdge(snowflake, size, +120, order)
    return snowflake

The new createSnowflake function takes an additional argument that specifies the order of the fractal. It then replaces the calls to addPolarEdge with calls to an as-yet-unwritten function called addFractalEdge, which adds an edge of the specified order to the GPolygon stored in snowflake.

The only remaining task is to implement addFractalEdge, which is easy if you think about it recursively. The simple case for addFractalEdge occurs when order is 0, in which case the method adds a polar edge with the length and direction. If order is greater than 0, the fractal line is broken down into four components, each of which is a fractal line of the next lower order. The code for addFractalEdge therefore looks like this:

def addFractalEdge(poly, r, theta, order):
    if order == 0:
        poly.addPolarEdge(r, theta)
    else:
        addFractalEdge(poly, r / 3, theta, order - 1)
        addFractalEdge(poly, r / 3, theta + 60, order - 1)
        addFractalEdge(poly, r / 3, theta - 60, order - 1)
        addFractalEdge(poly, r / 3, theta, order - 1)