Problem Set #5—Thinking Recursively

Due: Wednesday, October 30

This problem set is intended to give you practice with recursion, which is probably the topic from CSCI 121 that gives students the most trouble on the junior qual. None of these programs requires more than 15 lines of code. What makes them challenging is figuring out how to think about them in a way that makes the recursive solution jump out at you. Come to office hours if you get stuck.

**Problem 1**

Write a recursive function

```python
def binarySearch(value, array, p1, p2)
```

that uses the binary-search algorithm from Chapter 8 to return some index in `array` at which `value` appears, looking only at the array elements that fall in the range from `p1` to `p2` (where `p1` and `p2` are interpreted in the standard Python way to include the index `p1` up to but not including `p2`). If the value does not appear in the specified range of elements, your implementation of `binarySearch` should return the value `-1`.

**Problem 2**

Let’s define an English word as **collapsible** if it is possible to cross out one of its letter and still have an English word and, moreover, it is possible to repeat the process all the way to a single letter. As a simple example, the word *cats* is collapsible because you can cross out first the *s*, then the *c*, and then the *t*, leaving the words *cat*, *at*, and *a*, in order.

As a more extensive example, the longest collapsible word in English is *complecting* (the process of joining by weaving or binding together), which survives the following chain of deletions (some of which are admittedly unusual English words):

```
complecting
completing
competing
compting
comping
coping
oping
ping
pig
pi
i
```

Write a recursive function

```python
def isCollapsible(word)
```
that takes an English word and returns \texttt{True} if the word is collapsible. Assume that you have access to the \texttt{english} module, which exports the function \texttt{isEnglishWord}.

As is often the case with recursive problems, the hard part of this problem is recognizing how to break the problem down into simpler subproblems of the same form. The following pseudocode will get you started:

```python
def isCollapsible(word):
    if the word is a one-letter English word:
        return True.
    for each letter position in the word:
        check to see whether removing the letter leaves a collapsible word.
        if so, return true.
    if you get all the way through the word without finding a collapsible substring, return false.
```

Problem 3

Besides the Koch snowflake I showed in Monday’s class, another interesting fractal is the \textbf{Sierpinski Triangle}, named after its inventor, the Polish mathematician Waclaw Sierpiński (1882–1969). The order-0 Sierpinski Triangle is an equilateral triangle:

![Equilateral Triangle](image)

To create an order-$N$ Sierpinski Triangle, you draw three Sierpinski Triangles of order $N-1$, each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle, which means that the order-1 Sierpinski Triangle looks like this:

![Order-1 Sierpinski Triangle](image)

The downward-pointing triangle in the middle of this figure is not drawn explicitly, but is instead formed by the sides of the other three triangles. That area, moreover, is not recursively subdivided and will remain unchanged at every level of the fractal
decomposition. Thus, the order-2 Sierpinski Triangle has the same open area in the middle:

If you continue this process through three more recursive levels, you get the order-5 Sierpinski Triangle, which looks like this:

Write a function `createSierpinskiTriangle(size, order)` that takes the length of the triangle edge and the order of the fractal and returns a `GCompound` containing the shapes necessary to display the entire figure. The reference point of the `GCompound` should be the center of the triangle.