Introduction to Recursion

Today we'll start working through one of computer science's neatest ideas: recursion. Recursion often does the trick whenever the problem to be solved can be broken down into virtually identical (though smaller) sub-problems. The classic introductory example employing recursion is an implementation of the **factorial** function:

```
static int factorial(int n) {
   if (n == 0) return 1;
   return n * factorial(n - 1);
}
```

> Go ahead and start reading through Chapters 7 and 8. Your third assignment, going out on Wednesday, will be all about recursion, and will include one of our most popular assignments: The Game of Boggle

Every recursive function lists a sequence of base cases, and then one or more recursive cases. Occasionally, the problem to be solved is so simple that we can return or terminate execution without any further computation. The first of the two lines in **factorial** is an example of such a base case—**factorial(0)** is always **1** and is easily understood. However, whenever the specified integer **n** is larger than **0**, it helps to calculate **factorial(n - 1)** and multiply the result of that computation by **n** itself. That's precisely what the recursive call is doing.

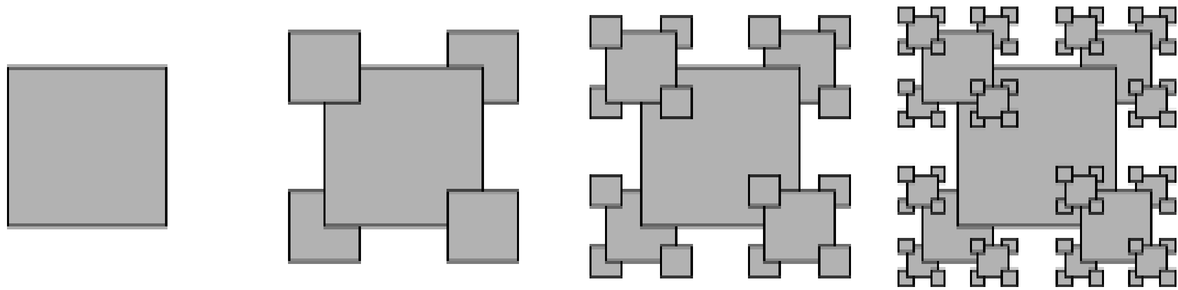
We'll be spending the rest of this week and all of next week learning recursion. Recursion is difficult to understand the first time you see it, so be patient if it doesn't click right away. I'll be covering many of the examples covered in the reader, but I don't reproduce those here. I do, however, have a good number of examples that aren't in the reader, and that's what this handout is all about.

Fractals I: Boxy Snowflakes

Assume you're given the following function, which draws a shaded square of the specified dimension with a solid border, centered at (cx, cy):

```
static void drawFilledBox(GWindow& window, double cx, double cy, double side
                          const string& fillcolor, const string& bordercolor);
```

Presented below is the recursive implementation of **drawBoxyFractal**, which is capable of drawing the following order-0, order-1, order-2, and order-3 fractals:



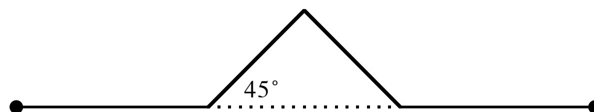
Note our implementation is sensitive to the way the centered squares are layered—clearly the sub-fractals drawn in the southwest and northeast corners are drawn before the large center square, which is drawn before the sub-fractals at 4:30 and 10:30. The same layering scheme is respected at all recursive levels:

```
static void drawBoxyFractal(GWindow& window, double cx, double cy,
                            double dimension, int order) {
    if (order >= 0) {
        drawBoxyFractal(window, cx - dimension/2, cy + dimension/2,
                        kScale * dimension, order - 1);
        drawBoxyFractal(window, cx + dimension/2, cy - dimension/2,
                        kScale * dimension, order - 1);
        drawFilledBox(window, cx, cy, dimension, "Gray", "Black");
        drawBoxyFractal(window, cx - dimension/2, cy - dimension/2,
                        kScale * dimension, order - 1);
        drawBoxyFractal(window, cx + dimension/2, cy + dimension/2,
                        kScale * dimension, order - 1);
    }
}
```

Fractals II [Problem and prose courtesy of Eric Roberts]

Although fractals have been a mathematical curiosity for more than a century, modern interest in fractals as a practical tool can be traced largely to Benoit Mandelbrot, a researcher at IBM who made an extensive study of the field. As a way of getting people to understand the concept, Mandelbrot posed the following question: How long is the coastline of England? You can look up an answer in an encyclopedia, but that answer turns out to be meaningless unless you know the level of granularity at which it is measured. As you move through successively finer scales, the coast grows progressively longer as you count the length of each little inlet or peninsula.

The coastline problem in the text provides (Chapter 8, Exercise 15) the background for one of today's example. Because the fractals come out a little cleaner if you do so, we've changed the dimensions of the triangle from the one in the text so that the angle of the bump in the fractal line is 45 rather than 60 degrees, like this:



The code for the finished implementation of **`drawCoastline`** looks like this:

```
static GPoint drawCoastline(GWindow& window, GPoint pt,
                            double length, double theta, int order) {

    if (order == 0) return window.drawPolarLine(pt, length, theta);
    double angle = randomChance(0.5) ? 45 : -45;
    pt = drawCoastline(window, pt, length/3, theta, order - 1);
    pt = drawCoastline(window, pt, length * sqrt(2)/6, theta + angle, order - 1);
    pt = drawCoastline(window, pt, length * sqrt(2)/6, theta - angle, order - 1);
    return drawCoastline(window, pt, length/3, theta, order - 1);
}

int main() {
    GWindow window(getScreenWidth()/2, getScreenHeight()/2);
    GPoint middleLeft(0, window.getHeight()/2);
    drawCoastline(window, middleLeft, window.getWidth(), 0, 8);
    return 0;
}
```

Listing All Subsets

The reader invests a good amount of energy talking about how to generate all of the **permutations** of a string—that is, how given the string **`PDQ`** you can programmatically generate:

```
PDQ
PQD
DPQ
DQP
QDP
QPD
```

The recursive formulation is easily explained in English, but coding it up is a different matter, so we'll definitely speak of it during lecture (though I don't repeat the code here.)

A different but related problem asks not for the permutations of a string, but rather the list of ordered **subsets**. Given the string **`"ABCD"`**, for instance, figure out how to print all 16 ordered subsets/subsequences—**`"ABCD"`**, **`"ACD"`**, **`"BD"`**, **`"C"`**, and **`""`** are five of the 16 strings that would need to be generated.

```
static void listSubsets(const string& prefix, const string& remaining) {
   if (remaining.empty()) {
      cout << prefix << endl;
      return;
   }

   listSubsets(prefix + remaining[0], remaining.substr(1));
   listSubsets(prefix, remaining.substr(1));
}

static void listSubsets(string str) {
   listSubsets("", str);
}
```

The Periodic Table Alphabet [problem courtesy of Keith Schwarz]

Have you ever wondered what English words can be spelled using just the symbols from the periodic table? You're about to find out.

The periodic table lists abbreviations used for all of the known elements—hydrogen, lithium, oxygen, carbon, molybdenum, uranium, and so forth. Each element has its own one, two, or three-letter symbol: H for hydrogen, Li for lithium, Mo for molybdenum, Cf for Californium etc. All in all, there are 118 elements, so there are 118 abbreviations.

For this problem, we're pretending that these symbols are the 'letters' of a new alphabet. Your task here is, given a **`Vector<string>`** storing all element symbols and a **`Lexicon`** of all English words, to print out all those English words that can be constructed using just the symbols from the periodic table. You should only print out words of length 11 or more, and you should retain the capitalization scheme of the atomic symbols when printing out the words. Here's a small window of the words that should be printed out:

```
...
IrReSOLuTiON
IrReSOLuTeNEsS
IrReSOLuTeNeSS
IrReSPONSiBILiTiEs
IrReSPONSiBiLiTiEs
IrReSPONSIBILiTiEs
IrReSPONSIBiLiTiEs
...
```

Ir is iridium, Re is rhenium, S is sulfur, O is oxygen, Lu is lutetium, etc.: Trust that every capitalized substring identifies some element from the periodic table. O solution builds on a working prefix up from the empty string by recursively considering every single symbol in the periodic table as a possible extension. If you construct a string that isn't even a prefix of a word, then you prune that search, recognize your dead end, and back off. But if you notice that the working string is actually a word, then as a side effect you print the string out and dig even further for a longer word beyond what you already have.

```
static const int kMinWordLength = 11;
void printAllWords(const Lexicon& english, const Vector<string>& elements,
                   const string& prefix) {
   if (!english.containsPrefix(prefix)) return;
   if (english.contains(prefix) && prefix.length() >= kMinWordLength) {
      cout << prefix << endl;
   }

   foreach (string element in elements) {
      printAllWords(english, elements, prefix + element);
   }
}

void printAllWords(const Lexicon& english, const Vector<string>& elements) {
   printAllWords(english, elements, "");
}
```