Assignment 3: Short Recursion Problems

Assignment 3 is going out in two parts: this one, which has you implement a few short recursion problems and submit them for feedback, and a larger one, which has you implement the game of Boggle. Both parts are required, but you're to complete and submit solutions for the problems described in this handout first, and then move on to the larger assignment—one that has you implement the game of Boggle—afterwards, which is discussed in Handout 17.

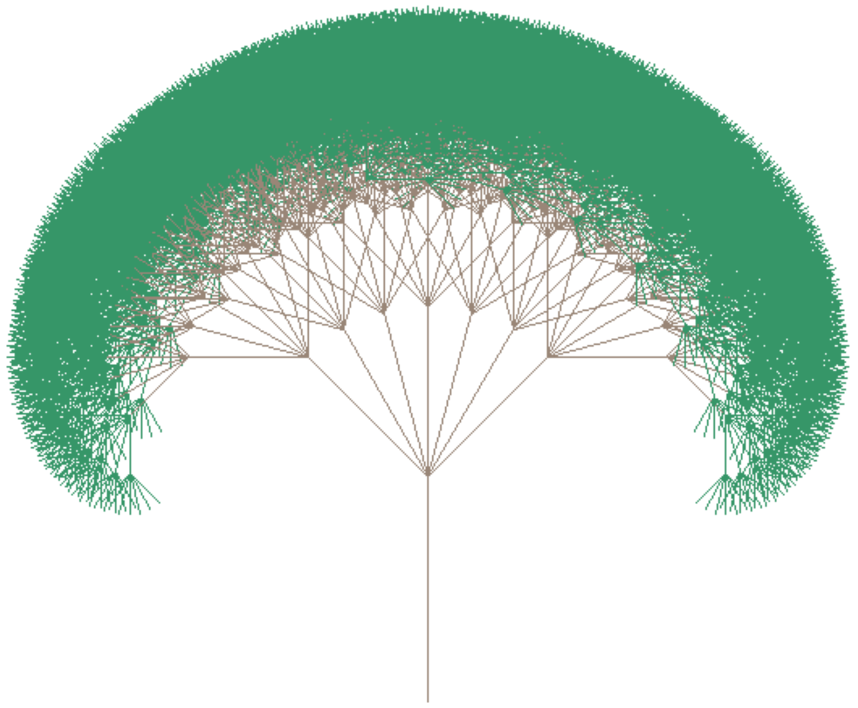
Solutions to Warm-up Problems Due: Wednesday, October 17th at 8:30 a.m.
Solution to Boggle Due: Friday, October 19th at 8:30 a.m.

For the three warm-up exercises, we specify the function prototype. **Your function must exactly match that prototype** (same name, same arguments, same return type). Your function must use recursion; even if you can come up with an iterative alternative, we insist on a recursive formulation! Also, note that the Boggle assignment is much more involved than these warm-up problems, so don't be left with the impression that you somehow need a week to complete the warm-ups and just two days for Boggle. In practice, you'll want to press through these problems fairly quickly and move on to Boggle pronto. I'm giving you seven days for this part not because it takes that long to complete them, but because it's obnoxious to give you less than a week for anything.

Think of these three problems and the Boggle portion of the assignment as one big assignment, and consider the completion of these three problems to be a milestone that needs to be completed by next Wednesday. As opposed to Assignment 1's checkpoint, these problems are **required** and solutions to them need to be submitted.

Late day computation is the sum of the late days used between the two, so ideally you'd turn in the solutions to the warm-up exercises on time, take at most one late day for Boggle, and then feel ready to go on the first CS106X midterm, to be held on Tuesday evening, October 23rd.

Problem 1: Trees

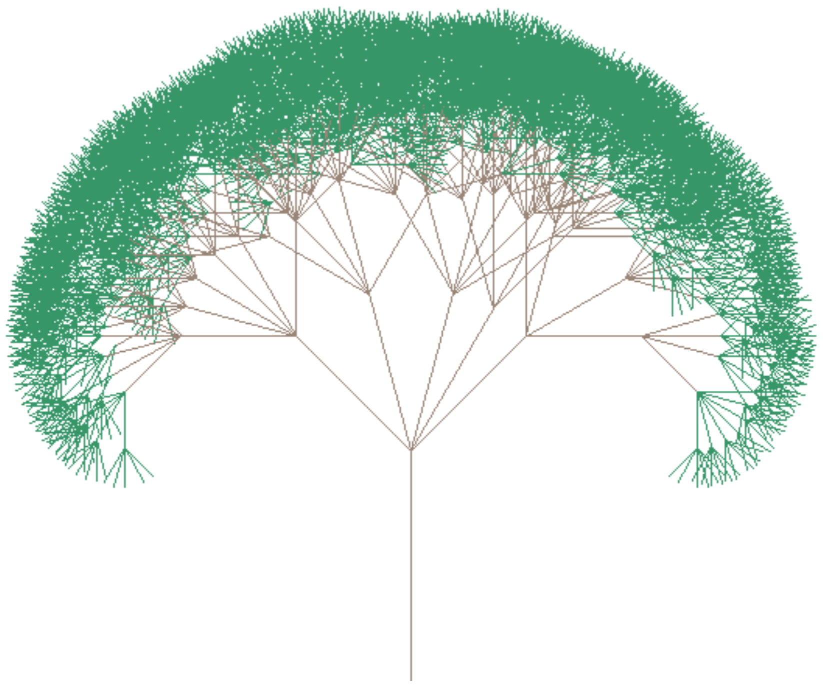


The drawing on the right is a tree of order 5, where the trunk of the tree is drawn from the bottom center of the graphics window straight up through a distance of **`kTrunkLength`** pixels. Sitting on top of that trunk are **seven** trees of order 4, each with a base trunk length that's 70% of the original. Each of the order-4 trees is topped off with seven order-3 trees, which are themselves comprised of order-2 trees, and so on.

The seven trees extend from the top of the tree trunk at relative angles of ±45, ±30, ±15, and 0 degrees. And even though you can't see it in the printout, if you run the sample application, you'll notice that the inner branches—or more specifically, all contributions at order 2 and higher—are drawn in **`kTrunkColor`**, and the leafy fringe of the tree is drawn in **`kLeafColor`**.

I've set up a **`trees.cpp`** file that draws an order-0 tree, and then layers an order-1 on top of it, and then an order-2 tree on top of that, and so forth. You're to complete the implementation so that the full sweep of the tree gets drawn. You should rely on the library method **`GWindow::drawPolarLine`** to draw lines at various angles, just as the **`draw-coastline`** example in Handout 14 does.

Once you get this working, adapt your implementation so that it potentially draws something like the tree drawn on the right. The same code used to generate the first drawing above was used to generate this one, except that in the first, each recursive call was made with probability 1.0, whereas in the second, each recursive call was made with probability 0.8.



Problem 2: Vampire Numbers

When two positive numbers **m** and **n** have the same number of digits and their product **p** is a permutation of the digits of **m** and **n** combined, we call **p** a **Vampire number** (and **m** and **n** are its **fangs**). For instance, 125460 is a vampire number, because it can be written as 204 times 615. 16758243290880 is also a Vampire number, because it just happens to equal 1982736 times 8452080. Note that the digits present in 1982736 and 8452080 are also present—with the same frequencies—in 16758243290880.

Using recursive backtracking, implement the **`isVampireNumber`** predicate function, which has the following prototype:

```
bool isVampireNumber(int number, int& first, int& second);
```

When the provided number is a vampire number, **`isVampireNumber`** should return **`true`** and place the two factors in the spaces referenced by **`first`** and **`second`**. When the provided number is not a vampire number, **`isVampireNumber`** should return **`false`** without concern for what **`first`** and **`second`** end up referencing.

The starter files provide a test framework to help exercise your implementation. And while your implementation should generalize to arbitrarily large integers, it only needs to be fast for numbers with 8 digits or less. Google **`"vampire numbers"`** for a list of 4, 6, and 8-digit vampire numbers so you know what numbers to test. In the rare case where a Vampire number can be split into fangs multiple ways (e.g. 12054060 is 2004 * 6015 and it's also 2406 * 6010), your **`isVampireNumber`** procedure can surface any single pairing.

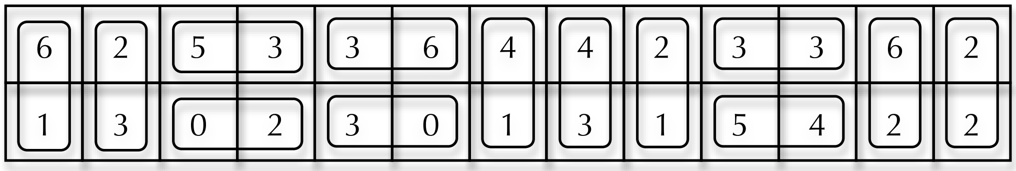
Problem 3: Finding Dominosa Solutions

The game of Dominosa presents a grid of small nonnegative integers, perhaps as follows:

6	2	5	3	3	6	4	4	2	3	3	6	2
1	3	0	2	3	0	1	3	1	5	4	2	2

There are always two rows of numbers, but the number of columns can, in principle, be any positive integer.

The goal is to pair horizontally and vertically adjacent numbers so that every number takes part in some pair, and no two pairs include the same two numbers. As such, one solution to the above problem would pair everything as follows:



Sadly, not all boards can be solved. One small, obvious example is:

3	1
1	3

Run the **`dominosa-sample`** sample application we've included in the collection of starter files. You'll see that the program generates random 2 x n boards (where you choose the value of n to be between 9 and 25 inclusive). For each randomly generated board, the application will animate the recursive backtracking search that determines whether some such pairing exists. The starter code provides the core of the interactive program, and it also provides a fully operational **`DominosaDisplay`** class that can be used to manage all aspects of the visualization. Your job is to implement the **`canSolveBoard`** function, which has the following prototype:

```
   bool canSolveBoard(DominosaDisplay& display, Grid<int>& board);
```

You'll need to read over the **`dominosa-graphics.h`** file to see how the **`DominosaDisplay`** can be used to script the animation, which if properly implemented will do a superb job of visually confirming that your recursive backtracking algorithm is working properly.