Assignment 3 Warm-ups: 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 18.

Solutions to Warm-up Problems Due: Friday, October 21st at 5:00 p.m.
Solution to Boggle Due: Monday, October 24th at 5:00 p.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 nine calendar days to complete the warm-ups and just three for Boggle. In practice, you'll want to press through these problems fairly soon and move on to Boggle ASAP. I'm giving you nine 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, and I won't be covering everything you need until this Friday.

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 Friday. As opposed to Assignment 1's checkpoint, these problems are **required** and solutions to them need to be submitted.

Problem 1: Twiddles

Two English words are considers to be *twiddles* whenever the letters at each position are either the same, neighboring letters (meaning next to each other in the alphabet), or next-to-neighboring letters. For instance, **`sparks`** and **`snarls`** are twiddles. Their second and second-to-last characters are different, but **`p`** is just two past **`n`** in the alphabet, and **`k`** comes just before **`l`**. A more dramatic example: **`craggy`** and **`eschew`**. They have no letters in common, but **`craggy`**'s **`c`**, **`r`**, **`a`**, **`g`**, **`g`**, and **`y`** are **–2**, **–1**, **–2**, **–1**, **2**, and **2** away from the **`e`**, **`s`**, **`c`**, **`h`**, **`e`**, and **`w`** in **`eschew`**. And just to be clear, **`a`** and **`z`** are **not** next to each other in the alphabet—there's no wrapping around at all.

Write a recursive procedure called **`FindTwiddles`**, which accepts a string **`str`** and a reference to an English language **`Lexicon`**, and collates all of its twiddles into a sorted **`Vector<string>`** and returns it. You'll probably want to write a wrapper function.

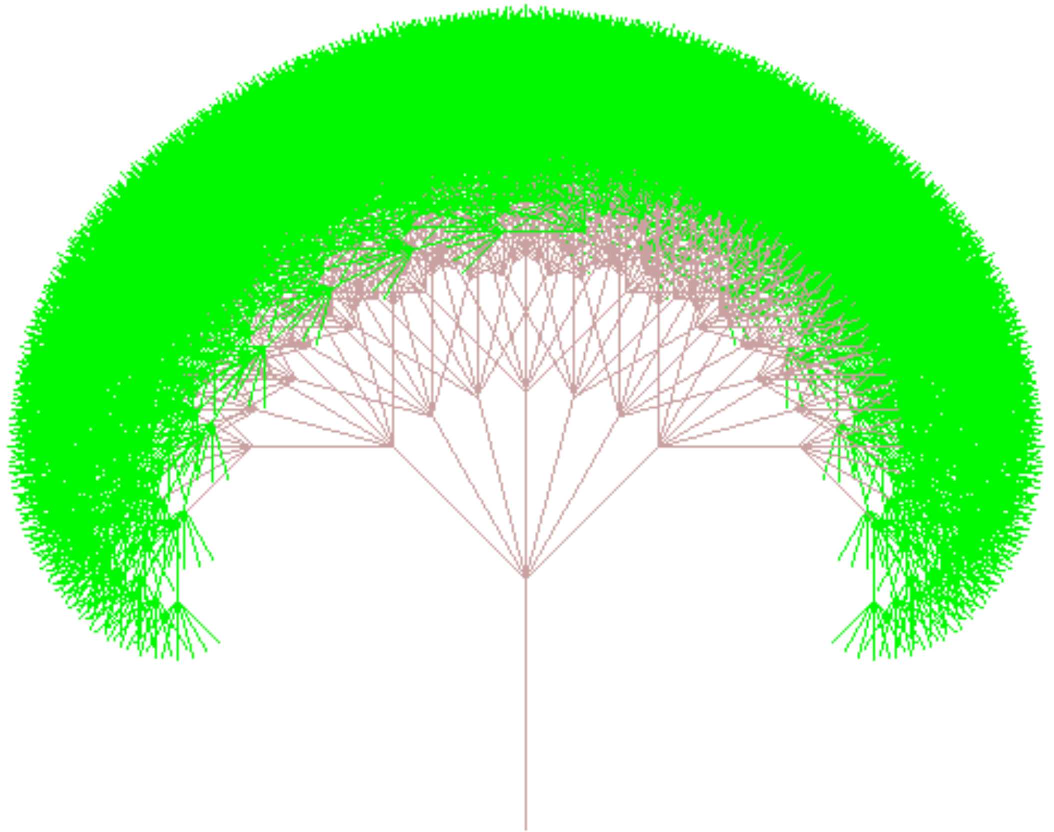
```
Vector<string> FindTwiddles(string str, Lexicon& lex);
```

The starter code for this is an interactive program that repeatedly prompts the user for a string, and—once fully implemented—lists all of the word's twiddles. (Your implementation should exclude the original word from the `Vector<string>`.)

Here's a sample run of my solution:

```
Welcome to the wonderful world of Twiddles! Enter a word, and if that word
has any twiddles, I'll print them out.

Enter a word [or enter to break]: craggy
"craggy" has 4 twiddles, and here they are:

1.) braggy
2.) draffy
3.) draggy
4.) eschew

Enter a word [or enter to break]: draft
"draft" has 7 twiddles, and here they are:

1.) brads
2.) braes
3.) brags
4.) craft
5.) crags
6.) drags
7.) frags

Enter a word [or enter to break]: clueless
"clueless" has just one twiddle, and it's "blueness".

Enter a word [or enter to break]: partner
"partner" has 6 twiddles, and here they are:

1.) obtunds
2.) papules
3.) partlet
4.) rassles
5.) rattler
6.) rattles

Enter a word [or enter to break]: groggy
"groggy" has just one twiddle, and it's "froggy".

Enter a word [or enter to break]: market
"market" has 6 twiddles, and here they are:

1.) larker
2.) laties
3.) maples
4.) marker
5.) masker
6.) naties

Enter a word [or enter to break]: health
"health" doesn't have any twiddles.

Enter a word [or enter to break]:
```

Problem 2: Trees

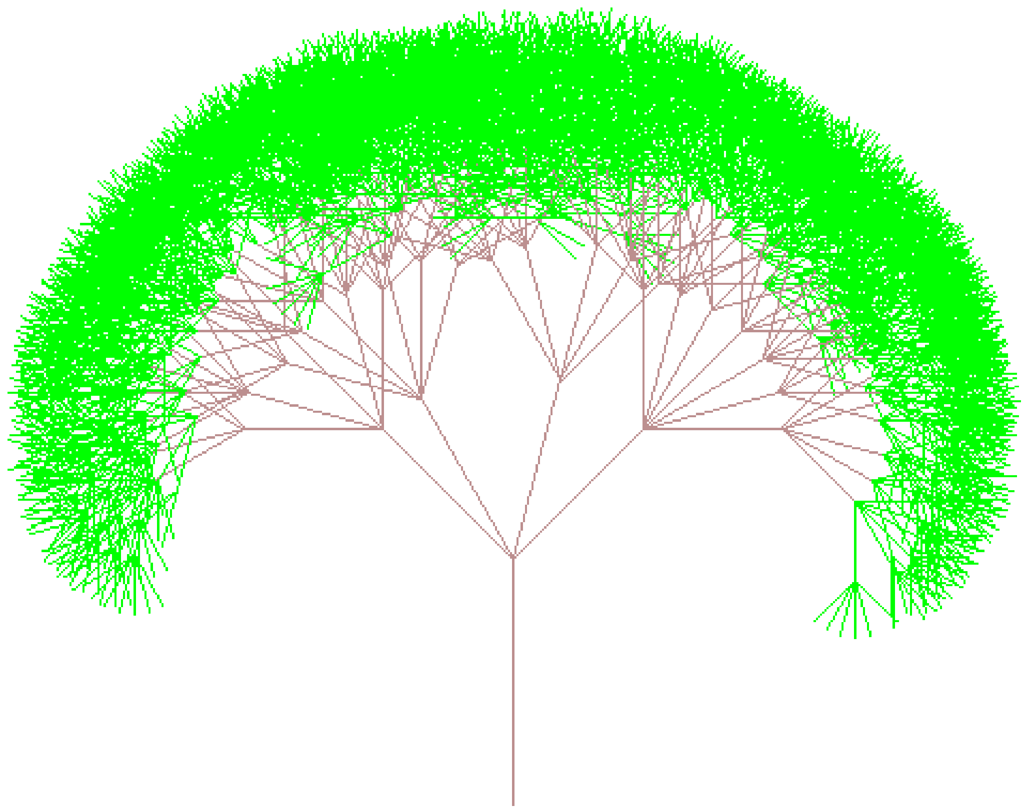


The drawing on the right is a tree of order 7, where the trunk of the tree is drawn from the bottom center of the graphics window straight up through a distance of **`kTrunkLength`** inches. Sitting on top of that trunk are **seven** trees of order 6, each with a base trunk length that's 75% of the original. Each of the order-6 trees is comprised of seven order-5 trees, which are themselves comprised of order-4 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 of the tree are drawn in brown, and the leafy fringe of the tree is drawn in green.

I've set up a **`trees.cpp`** file that draws the trunk of an order-7 tree. You're to complete the implementation so that the full sweep of the tree gets drawn. You should rely on the supplied **`DrawPolarLine`** function to draw lines at various angles, and you can use the library functions **`GetCurrentX()`** and **`GetCurrentY()`** to be reminded of the current position of the imaginary pen poised above the graphics window.

Once you get this working, adapt your implementation so that it potentially draws something like the tree drawn on the next page:

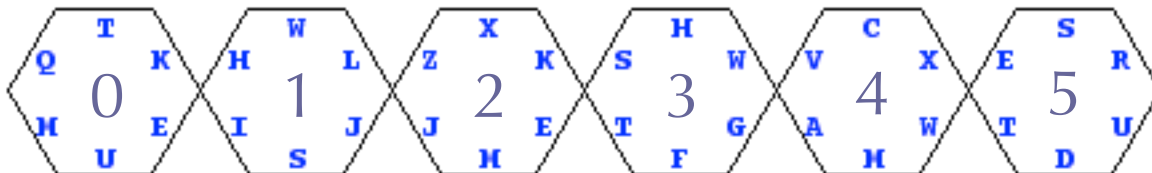


The same code used to generate the first drawing on the previous page was used to generate this one, except that in the first, each recursive call was made with probability 1.0, whereas in the second, each recursive call was made with probability **less than** 1.0. In fact, I drew the most askew trees (those at relative angles of + and – 45 degrees) with probability 0.9, and I drew the fourth of the seven recursive trees with probability 0.6. You can invent your own probability distribution, though your trees should still look fairly full. Anemic trees will lose points. ☺

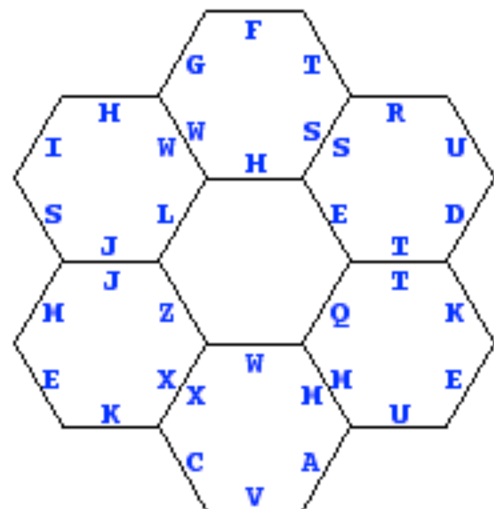
Problem 3: Beehive Puzzle

You're given six hexagonal games pieces, where each piece has a series of six letters—one per edge—around its perimeter. You're interested in finding an arrangement of the six pieces so that all six wrap around a hollow center, and all letters on adjacent edges match.

So, if you're given the following **`Vector`** of six pieces:



you should be able to permute and rotate as necessary in order to find something like this:



Implement a function called **`ArrangementExists`**, which takes a **`Vector`** of six **`string`**s, where each **`string`** consists of the six characters along the perimeter of one hexagonal piece. The order of the letters dictates the clockwise ordering along the perimeter of the piece, but the piece may be rotated by any multiple of 60 degrees if it'll help to produce a solution. **`ArrangementExists`** reports back a **`true`** or a **`false`**—**`true`** if and only if some solution could be

found. You don’t need to return what the solution is; you only need to return a yes or a no.

```
bool ArrangementExists(Vector<string>& pieces);
```

We’ve included some data files to provide multiple beehive puzzle examples so that you can effectively exercise your solution. Everything has already been implemented for you, save for the meat of the **`ArrangementExists`** function, which returns **`true`** if and only if the provided supplied hexagonal pieces can be chained in a loop, and **`false`** otherwise.

Thoughts on recursion

Recursion is a tricky topic, so don't be dismayed if you can't immediately sit down and code these perfectly the first time. Take time to figure out how each problem is recursive in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the same problem. You will need to depend on a recursive "leap of faith" to write the solution in terms of a problem you haven't solved yet. Be sure to take care of your base case(s) lest you end up in infinite recursion.

The great thing about recursion is that once you learn to think recursively, recursive solutions to problems seem very intuitive. Spend some time on these problems and you'll be in much better shape to tackle Boggle.