SE106, Fall 2012
Lab 2: Recursion
Assigned: October 8
Due: November 4, 24:00

Introduction

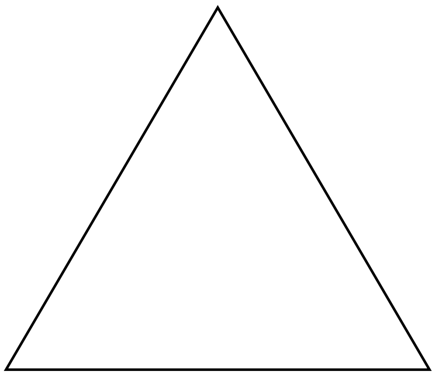
Learning to solve problems recursively can be challenging, especially at first. We think it's best to practice in isolation before adding the complexity of integrating recursion into a larger program.

The recursive solutions to most of these problems are quite short - typically less than a dozen lines each. That doesn't mean you should put this assignment off until the last minute though - recursive solutions can often be formulated in a few concise, elegant lines but the density and complexity that can be packed into such a small amount of code may surprise you.

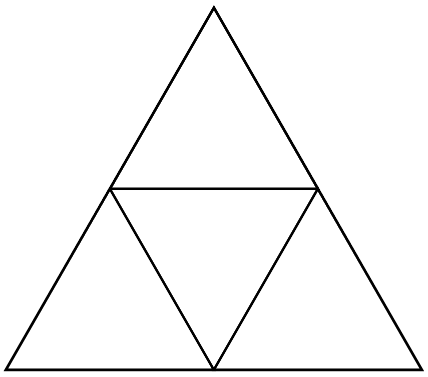
Problem I: Sierpinski Triangle

If you search the web for fractal designs, you will find many intricate wonders beyond the Koch snowflake illustrated in Chapter 8. One of these is the *Sierpinski triangle,* named after its inventor, the Polish mathematician Wacław Sierpiński (1882 – 1969).

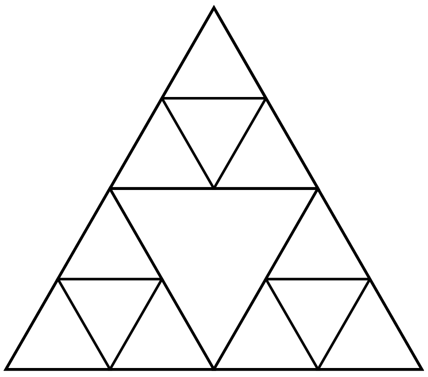
The order-0 Sierpinski Triangle is an equilateral triangle:



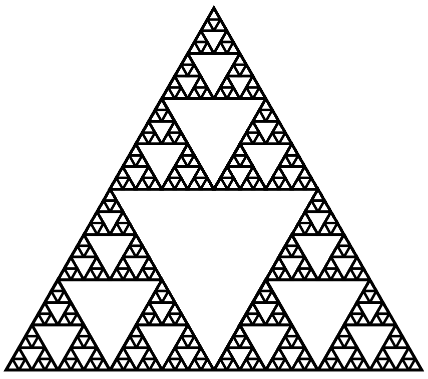
To create an order-*N* Sierpinski Triangle, you draw three Sierpinski Triangles of order-(*N*-1), each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle, which means that the order-1 Sierpinski Triangle looks like this:



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:

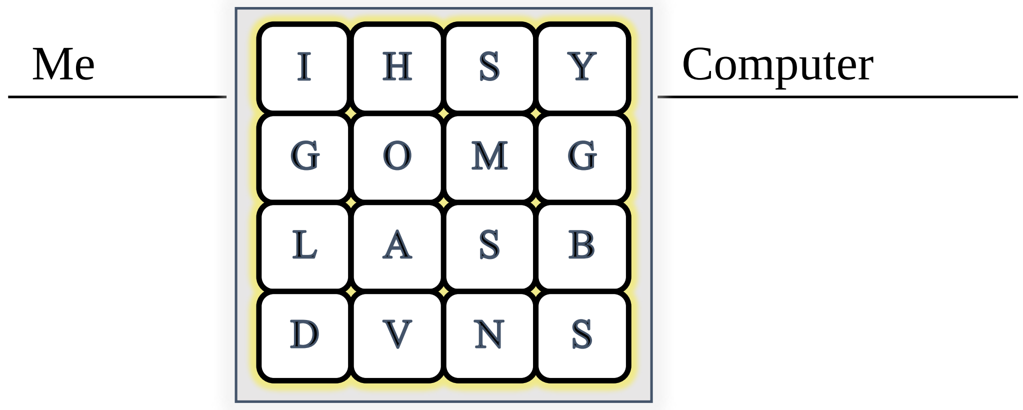


If you continue this process through three more recursive levels, you get the order-5 Sierpinski Triangle, which looks like this:



Write a program that asks the user for an edge length and a fractal order and draws the resulting Sierpinski Triangle in the center of the graphics window.

Problem II: The Boggle Game



This assignment was originally developed by Todd Feldman and then enhanced by Julie Zelenski. Your mission is to write a program to play the game of Boggle™, which should help you get over any lingering doubts about the power of recursive techniques.

The Boggle Game

You set up the letter cubes, shake them up, and lay them out on the board. The human player gets to go first (nothing like trying to give yourself the advantage). The player enters words one by one. After verifying that a word is legitimate, you highlight its letters on the board, add it to the player's word list, and award the player points according to the word's length.

Once the player has found as many words as he or she can, the computer takes a turn. The computer searches through the board to find all the remaining words and awards itself points for those words. The computer typically beats the player mercilessly, but the player is free to try again and again, until finally ready to acknowledge the superiority of silicon.

The Letter Cubes

The letters in Boggle are not simply chosen at random. Instead, the letter cubes are designed in such a way that common letters come up more often and it is easier to get a good mix of vowels and consonants. To recreate this, our starter code declares an array of the cubes from the or iginal Boggle. Each cube is described using a string of 6 letters, as shown below:

```
const string STANDARD_CUBES[16]  = {
   "AAEEGN", "ABBJOO", "ACHOPS", "AFFKPS",
   "AOOTTW", "CIMOTU", "DEILRX", "DELRVY",
   "DISTTY", "EEGHNW", "EEINSU", "EHRTVW",
   "EIOSST", "ELRTTY", "HIMNQU", "HLNNRZ"
};
```

These strings are used to initialize the cubes on the board. At the beginning of each game, "shake" the board cubes. There are two different random aspects to consider. First, the cubes themselves need to be shuffled so that the same cube is not always in the same location on the board. Second, a random side from each cube needs to be chosen to be the face-up letter.

Alternatively, the user can choose to enter a custom board configuration. In this case, you still use your same board data structure. The only difference is where the letters come from. The user enters a string of characters, representing the cubes from left to right, top to bottom. Verify that this string is long enough to fill the board and re-prompt if it is too short. If it's too long, just ignore the ones you don't need. You do not need to verify that the entered character s are legal letters.

Big Boggle variant uses a 5x5 board. User may choose the size of Boggle before playing. Our starting code declares two different cube arrays, one with the 16 cubes for the standard game and another with the 25 cubes for the bigger version.

The Human Player's Turn

The human player enters each word she finds on the board. For each word, check that:

- It is at least four letters long.
- It is contained in the English lexicon.
- It has not already been included in the player's word list (even if there is an alternate path on the board to form the same word, the word is counted at most once).
- It can be formed on the board (i.e., it is composed of adjoining letters and each cube is used at most once).

If any of these conditions fail, the word is rejected. If all is good, you add the word to the player's word list and score. In addition, you graphically show the word's path by temporarily highlighting its cubes on the board. The length of the word determines the score

The player enters a blank line when done finding words, which signals the end of the human's turn.

Notice: words should be considered case-insensitively: PEACE is the same as peace.

The Computer's Turn

The computer then searches the entire board to find the remaining words missed by the human player. The computer earns points for each word found that meets the requirements (minimum length, contained in English lexicon, not already found, and can be formed on board).

As with any exponential search algorithm, it is important to look for ways to limit the search to ensure that the process can be completed in a reasonable time. One of the most important Boggle strategies is to prune dead end searches. For example, if you have a path starting `zx`, the lexicon's `containsPrefix` member function will inform you that there are no English words down that path. Thus, you should stop right here and move on to more promising combinations. If you miss this optimization, you'll find yourself taking long coffee breaks while the computer is futilely looking for words like `zxgub`, `zxaep`, etc.

Our Provided Code

We have written all the fancy graphics functions for you. The functions exported by the `gboggle.h` interface are used to manage the appearance of the game window. It includes functions for initializing the display, labeling the cubes with letters, highlighting cubes, and displaying the word lists. Read the interface file for more details.

Hand in

You only need to turn in your `Sierpinski.cpp` and `Boggle.cpp` via:

```
$ turnin lab2@cplusplus Sierpinski.cpp Boggle.cpp
```