Assignment 3: Meta Academy

Synthesis by Chris Piech. Based on a handout by Marty Stepp. Problems by Eric Roberts, Stuart Reges and Chris Piech.

**OCTOBER 14, 2016**

*In order to learn recursion, you will teach recursion.   
In order to teach recursion, you must first learn recursion.   
Thanks to Khan Academy (no affiliation) for the image.*

In this assignment you will build tools for a learning platform, called Meta Academy, that teaches recursion. Online learning platforms like Coursera, Khan Academy and one day Meta Academy, offer education to anyone for free. If the quality of the education delivered were high enough this could have an enormous social impact. There are many open questions in this domain, and several of the open questions like "how do we generate personalized content?" and "how can we understand students as they learn?" can be approached using computer science techniques. In fact, current state of the art solutions for knowledge tracing, finding optimal student routes and several other problems are recursive algorithms.

This assignment consists of two main types of contributions: (1) demonstrations of how recursion works and (2) tools for computational education. The aim of this assignment is to get you warmed up to recursion: that beautiful but sometimes mind-bending new way of solving problems. In order to build the Meta Academy you will write five different recursive functions. The starter code for this project is available as a ZIP archive:

[Starter Code](http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172//assnFiles/meta/cs106b-hw3-starter.zip)

Turn in only the following file:

1. **recursion.cpp**, the C++ code for all of your recursion functions.

This is a pair assignment. You may work in a pair or alone. Find a partner in section or use the course message board. If you work as a pair, comment both members' names atop every code file. Only one of you should submit the program; do not turn in two copies. **You will not help either partner by "dividing up the assignment (e.g., "you do 1,2,3 and I'll do 4,5")**. Working as partners means that both partners should *always* work at one computer, together. There should never be a time where a partner is working independently on a pair-programming assignment.

Submit using the Paperless system linked on the class web site.

**Due Date:** Meta Academy is due Monday, October 24th at 12:00pm (noon).

**Y.E.A.H hours:**   
 Date Monday, October 17th  
 Time 7pm-8pm  
 Location TBA

**Related Reading:**   
[Euclid's Algorithm](https://en.wikipedia.org/wiki/Euclidean_algorithm)  
[Grammars](https://en.wikipedia.org/wiki/Formal_grammar)  
[Code.org research](https://code.org/research)  
[Open Syllabus Proj.](http://www.nytimes.com/2016/01/24/opinion/sunday/what-a-million-syllabuses-can-teach-us.html)

Overview

When you run Meta Academy you will be presented with a menu that has 5 options (plus the choice to exit). You would have been more than able to write the code to; navigate the menu, output instructions and update the display. However, we provide you with code that already does those basic operations. This allows you to focus on writing the recursive functions.

*The Meta Academy menu. This should run with your starter code.*

When you first download the starter code, run the program and try each menu item to see how the program works. The five menu items are:

1. [Demo recursion by definition](http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172/assn/metaAcademy.html#demoDef). You write a simple recursive function for greatest common divisor.
2. [Demo recursion for fractals](http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172/assn/metaAcademy.html#demoFractals). You write a serpinskii fractal.
3. [Demo recursion for exploration](http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172/assn/metaAcademy.html#demoExploration). You write flood fill.
4. [Personal curriculum](http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172/assn/metaAcademy.html#personalCurriculum). You write a recursive function that helps a student find the best path to their learning goal.
5. [Generate question](http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172/assn/metaAcademy.html#generate). You write a recursive function that generates a random question.

Once all of these are written you will have a tool to start teaching others recursion. And there is nothing like teaching to help you learn.

For all the functions you have to write in this assignment you **must use recursion**. That is for two reasons. First, the point of this assignment is for you to learn recursion. Second, all of the problems are ones for which recursion is the most natural fit.

Milestone 1: Greatest Common Denominator

For this problem, write a recursive function to compute the greatest common divisor that shows its work:

int **gcd**(int a, int b)

Some operations are much easier to define recursively. One amazing example of this is Euclid's Algorithm to calculate the greatest common divisor (GCD). In the algorithm Euclid famously shows that the **gcd(a, b)** is equal to **gcd(b, r)** where r is the remainder when you divide a by b. In the case where b is equal to 0, **gcd(a, 0)** is simply a. Since gcd is defined recursively, it is much easier to program using recursion. Let's calculate gcd.

*Euclid's recursive definition of GCD.*

When the user choses to show the demo of recursion by definition, the meta program will call the function **int gcd(int a, int b)** that you will write (see **recursion.cpp**). Your job is to calculate the gcd of the values that are passed in and to show your work as you go. Each invocation of the GCD function should create an output that shows how that invocation calculated the GCD.

*An example run of the recursive definition demo, that calls your gcd function.*

As an aside, though its straightforward to take Euclid's recursive definition and turn it into C++ code, it's slightly more challenging to see why it holds true. Understanding the proof for why it is true is beyond what we are asking of you, but for those so inclined you can check out the proof on the [wikipedia](https://en.wikipedia.org/wiki/Euclidean_algorithm#Proof_of_validity) page (warning: not simple). It is amazing to imagine that someone came up with this idea around 2,300 years ago.

Milestone 2: Serpinskii Fractal

For this problem, write a recursive function to draw a Serpinskii triangle:

void **drawSierpinskiTriangle**(GWindow& gw, int x, int y, int size, int order)

Recursion can also create incredible graphic designs of images that have self-similar subparts. One example is the Sierpinski Triangle, named after the Polish mathematician Waclaw Sierpinski.

Your function should draw a black outlined Sierpinski triangle when passed a reference to a graphical window, the x/y coordinates of the top/left of the triangle, the length of each side of the triangle, and the order of the figure to draw (such as 1 for Order-1, etc.). The provided files already contain a main function that constructs the window and passes the relevant information to your function. The rest is up to you.

If the order passed is 0, your function should not draw anything. If the order passed is negative, your function should throw a string exception. Otherwise you may assume that the window passed is large enough to draw the figure at the given position and size.

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 Waclaw Sierpinski (1882–1969). The order-1 Sierpinski Triangle is an equilateral triangle, as shown in the diagram below.

To create an order-K Sierpinski Triangle, you draw three Sierpinski Triangles of order K-1, each of which has half the edge length of the original. Those three triangles are placed in the corners of the larger triangle. Take a look at the Order-2 Sierpinski triangle below to get the idea

The upward-pointing triangle in the middle of the Order-2 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 order of the fractal decomposition. Thus, the Order-3 Sierpinski Triangle has the same open area in the middle.

Order 1

Order 2

Order 3

Order 7

*Different order Serpinskii triangles.*

We have not really learned much about the GWindow class, but you do not need to know much about it to solve this problem. The only member function you will need from the GWindow is its drawLine function:

gw.drawLine(x1, y1, x2, y2);

Which draws a line from point (x1, y1) to point (x2, y2)

Note: You may find yourself needing to compute the height of a given triangle so you can pass the right x/y coordinates to your function or to the drawing functions. Keep in mind that the height h of an equilateral triangle is not the same as its side length s. The diagram at right shows the relationship between the triangle and its height. You may want to look at information about [equilateral triangles](https://en.wikipedia.org/wiki/Equilateral_triangle) on Wikipedia and/or refresh your trigonometry. You may also find the worked examples from the [Fractals](http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172//lectures/8-Fractals) class to be useful.

Another thing you should avoid is re-drawing the same line multiple times. If your code is structured well, you should never end up drawing a line again (or part of a line again) that was already drawn.

Milestone 3: Flood Fill

For this problem, write a recursive function to perform a "flood fill" on a graphical window as described below.

int **floodFill**(GBufferedImage& image, int x, int y, int color)

A third great use of recursion is when you want to explore a series of decisions. Often decision making is recursive. Once you make one decision you find yourself in a situation which is of the same format. Here we will show how recursion can be used to explore a region of pixels. We will recursively try all possible combinations of making the decisions: move up a pixel, move left, move right and move down.

Most computer drawing programs make it possible to fill a region with a solid color. Typically you do this by selecting the "fill" tool and selecting a color, then clicking the mouse somewhere in your drawing. When you click, the paint color spreads outward from the point you clicked to every contiguous part of the image that is the same color as the pixel where you clicked. For example, if you select the Fill tool and choose Red as your color, then click on a purple part of the image, the Fill operation will cause that part of the image, along with any purple part that is touching it, to become red. The screenshots below provide an illustration:

*Flood fill demonstration.*

Think of the image as being composed of a 2-D grid of tiny square dots called pixels (picture elements). When the user clicks, whatever color that pixel was, it should be replaced by the fill color outward in all four directions: up, down, left, and right. For example, if the initial pixel color is purple, change it to red, then explore one pixel up, down, left, and right from there, looking for more purple pixels and changing them to red. If your exploration leads you to a pixel that is a different color (such as yellow rather than purple in our example), stop exploring in that direction.

Drawing pixels: The Sierpinski Triangle problem asks you to interact with a GWindow object, but that class is not built for accessing individual pixels. So we will use a different object called GBufferedImage to color pixels here. You can use the following member functions of the buffered image object to get and set the color of individual pixels.

| **Command** | **Description** |
| --- | --- |
|  |  |
| **image.getRGB(x, y);** | returns the color of the given pixel as an integer. |
| **image.setRGB(x, y, color);** | sets the color of the given pixel to the given color as an integer. |
| **image.getWidth(); image.getHeight();** | returns the width or height of the image in pixels. |
| **image.inBounds(x, y);** | returns true if pixel (x, y) is within the bounds of the image |

Your code should take care not to access any pixel outside of the bounds of the graphical image, (0, 0) through (width-1, height-1). If you try to get/set a pixel color that is out of bounds, your program will crash.

int color: The colors of pixels are returned as int values. The exact values that map to various colors don't especially matter to your code, though you can see what color maps to what integer value by looking at the **meta.cpp** file. If you want to print a color out on the console for debugging, it is easier to view it in hexadecimal (base-16) mode. To do that, you'd issue a special 'hex' command to cout to make it print an integer in hexadecimal rather than decimal (base-10), such as the following:

cout << hex << color << endl;

Optimization: For efficiency, your function is required to implement a particular optimization. If the caller passes the same fill color as the current pixel's existing color, do not color any pixels and immediately return 0. For example, if the user has the Fill color set to Blue and clicks a blue area, your function should immediately return 0. You can test whether your function is implementing this correctly by watching the console's output while clicking

Return value: Your function must return the total number of pixels that changed color. For example, if your code ends up filling a 50x30 rectangular region, your function would return 1500. If no pixels change color, return 0.

On some machines (such as my machine), filling a very large area of pixels can crash your program because of too many nested function calls ("stack overflow"). This is out of your control. You may ignore this issue, so long as your algorithm is correct.

Milestone 4: Personalized Curriculum

For this problem, write a recursive function to calculate a personalized curriculum for a student as described below.

void **personalCurriculum**(Map<string, Vector<string> > & prereqMap, string goal)

When students learn, they come with a variety of different goals. If a learner wants to master a particular goal-concept they may not need to learn every other concept in the course. Instead they just need to learn that goal-concept and all the course concepts that it builds upon. The point of the Personalized Curriculum tool is to output, for a particular course and a particular goal, all of the required concepts that the student must learn to be ready to learn their "goal-concept." The concepts should be printed in an order such that if the student were to learn the concepts in that order they never would be working on a concept for which they don't have the necessary pre-requisites. Here are two examples:

*Two example runs of Personalized Curriculum*

Prerequisite Concepts

In order to compute a personalized curriculum we are going to be given a prereqMap that associate each concept with the immediate prerequisites needed for that concept. You are probably familiar with the idea of prerequisites on the course level. Our class cs106b lists cs106a as a prerequisite. The course cs107 lists cs106b as its only prerequisite. However, in order to take cs107 you must first take cs106a then cs106b as prerequisites apply recursively. In general the you can calculate all prerequisites of a concept using the following recursion:

*allPrereqsOfConcept(prereqMap, concept){ it's direct prerequisites and for (childConcept : direct prerequisites){ allPrereqsOfConcept(prereqMap, childConcept) } }*

The first example run (the one on the left) invokes personalized curriculum on the "recursion" course, which is a tiny course you can use to test your program (it does not include all concepts in cs106b). Here is the chain of prerequisites for that course:

*The recursion prerequisite relationship is the smallest one you can test your algorithm on. An arrow means that in order to learn the concept on the right, you must first learn the concept on the left.*

If your goal is to learn "exploration recursion" you must first learn "recursion." Before you learn "recursion" you must learn its prerequisites, "collections" and "functions." The pattern continues until you get to "simpleC++" which has no prerequisites. Thus if your goal was to learn "exploration recursion" then a good personalized curriculum for you would be:

simpleC++ function calls collections recursion exploration recursion

One important note: if you recursively expand the prerequisites of "exploration recursion" some concepts will be generated many times. A common mistake will be to output the concept more than once.

simpleC++ functionCalls simpleC++ collections recursion explorationRecursion

Use a collection to keep track of the concepts you have already printed so that if you have already told the student they should learn, for example, "simpleC++" you don't list "simpleC++" again.

Prerequisite Map

When you run Personalized Curriculum you will be given the prerequisite map that corresponds to the course the user selected. You do **not** have to load the prerequisite map from file. As an example, if personalized curriculum was run on the **recursion** course you would be passed the following **prereqMap**:

"fractals" → ["recursion"] "explorationRecursion" → ["recursion"] "definitionRecursion" → ["recursion"] "recursion" → ["collections", "functionCalls"]

Courses

There are three different courses that we have created:

| **Command** | **Description** |
| --- | --- |
|  |  |
| **cs106b** | A general prerequisite map for the entire cs106b course. |
| **recursion** | A tiny prerequisite map only focused on the concept of recursion. |
| **stanfordcs** | The prerequisite map for many of the courses in the stanford cs department. (There is a corresponding file stanfordcs-names.txt that shows how the course numbers map to course names). |

Milestone 5: Generate Question

For this problem, write a recursive function to generate a random expansion of a grammar as described below. You can use this function to create random question for students to answer -- specifically we are going to make random program traces (like the sort you will have to do on your midterm).

string **generate**(Map<string, Vector<string> > & grammar, string symbol)

Grammars

**Edit (Oct 17th):** when working on this milestone we stongly suggest that you **first** test your code on **sentence.txt** and then make sure that the same code works with **mystery.txt**. Both are grammars. We added a new example to show a grammer expansion for mystery.

A formal language is a set of words and/or symbols along with a set of rules, collectively called the syntax of the language, defining how those symbols may be used together. A grammar is a way of describing the syntax and symbols of a formal language. Many language grammars are described in a format called Backus-Naur Form (BNF). Since most languages are recursive, BNF is too.

Some symbols in a grammar are called terminals because they represent fundamental words of the language. A terminal in English might be "boy" or "run" or "Jessica". Other symbols of the grammar are called non-terminals and represent high-level parts of the language syntax, such as a noun phrase or a sentence. Every non-terminal consists of one or more terminals; for example, the verb phrase "throw a ball" consists of three terminal words.

The BNF description of a language consists of a set of derivation rules, where each rule names a symbol and the legal transformations that can be performed between that symbol and other constructs in the language. For example, a BNF grammar for the English language might state that a sentence consists of a noun phrase and a verb phrase, and that a noun phrase can consist of an adjective followed by a noun or just a noun. Rules can be described recursively (in terms of themselves). For example, a noun phrase might consist of an adjective followed by another noun phrase, such as the phrase "big green tree" which consists of the adjective "big" followed by the noun phrase "green tree".

A BNF grammar is specified as an input file containing one or more rules, of the form:

non-terminal: rule|rule|rule|...|rule

In the following example, non-terminal names <s>, <np> and <tv> are short for linguistic elements sentences, noun phrases, and transitive verbs respectively.

<s>:<np> <vp> ----- <np>:<dp> <adjp> <n>|<pn> ----- <dp>:the|a ----- <adjp>:<adj>|<adj> <adjp> ----- <adj>:big|fat|green|wonderful|faulty|subliminal|pretentious ----- <n>:dog|cat|man|university|father|mother|child|television ----- <pn>:John|Jane|Sally|Spot|Fred|Elmo ----- <vp>:<tv> <np>|<iv> ----- <tv>:hit|honored|kissed|helped ----- <iv>:died|collapsed|laughed|wept -----

This grammar's language can represent sentences such as "The fat university laughed" and "Elmo kissed a green pretentious television". This grammar cannot describe the sentence "Stuart kissed the teacher" because the words "Stuart" and "teacher" are not part of the grammar. It also cannot describe "fat John collapsed Spot" because there are no rules that permit an adjective before the proper noun "John", nor an object after intransitive verb "collapsed".

In this milestone you are going to use grammars that describe different question types. Your task will be to generate random questions using those grammars. Your function will be passed in a grammar as a **Map<string, Vector<string> > grammar** which will associate each non-terminal with the Vector of each of its production rules (the strings separated by the | symbols). Using the grammar above as an example the key **<np>** would map to a vector with two elements:

<np> → ["<dp> <adjp> <n>", "<pn>"]

Random Expansions

Your function involves recursively walking the grammar data structure to generate elements by successively expanding them. To generate a random occurrence of a symbol S:

* If S is a terminal symbol, there is nothing to do; the result is the symbol itself.
* If S is a non-terminal symbol, choose a random expansion rule R for S. For each of the symbols in the rule R, generate a random occurrence of that symbol.

For example, the grammar on the previous page could be used to randomly generate a <s> non-terminal for the sentence, "Fred honored the green wonderful child", as shown in the following diagram.

*Exansion example.*

If the string passed to your function is a non-terminal in your grammar, use the grammar's rules to recursively expand that symbol fully into a sequence of terminals. For example, using the grammar on the previous pages, a call of grammarGenerate("<np>") might potentially return the string, "the green wonderful child"

Generating a non-terminal involves picking one of its rules at random and then generating each part of that rule, which might involve more non-terminals to recursively generate. For each of these you pick rules at random and generate each part, etc.

When you encounter a terminal, simply include it in your string. This becomes a base case. If the string passed to your function is not a non-terminal in your grammar, you should assume that it is a terminal symbol and return it.

Generate

Your job is to write the recursive function generate that takes in a grammar and a symbol and expands that symbol using the grammar.

When you want to loop over a string exansion you will need to process the string one "token" at a time where a token could be a non-terminal or a terminal. CS106B has a handy library **TokenScanner** that makes this very easy. Here is an example:

TokenScanner scanner(production); while (scanner.hasMoreTokens()) { string token = scanner.nextToken(); // do something with token }

Random Question Expansion

Start by testing your code on **sentence.txt**. Once it is working, try using the grammar **mystery.txt** to expand the symbol QUESTION. If your code is working correctly this should produce a random recursion code-trace question.

There are three different question grammars that come with the starter code (and the sentence grammar described above):

| **Command** | **Description** |
| --- | --- |
| **sentence.txt** | This is not a recursion question, but we provided this grammar for testing. |
|  |  |
| **mystery.txt** | A template for a random mystery trace problem. |
| **mysteryHard.txt** | A template for a random mystery trace problem for people who love a challenge. |
| **collectionsRecursion.txt** | A template for slightly ambiguous recursion problems using collections. |

Here is an example question that we generated when expanding the QUESTION symbol for the **mystery.txt** grammar:

int mystery(int level, int x) { if(level == 0) { return 5; } else { int r = mystery(level - 1, r); return r % x + 1; } } What is the result of mystery(3, 5);

Possible Extra Features:

Thats all! You are done. Consider adding extra features.