Assignment 3: Recursion

THANKS TO JULIE ZELENSKI, JERRY CAIN, MARTY STEPP, KEITH SCHWARZ AND CHRIS PIECH

DUE: FRIDAY, OCTOBER 19, 11AM

MUST BE DONE INDIVIDUALLY

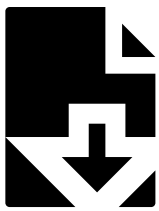
The purpose of this assignment is to gain familiarity with recursive problem solving in both graphics and console programs. Note that this assignment must be done individually; you may not work in pairs for this assignment.

This assignment consists of several parts: **Fractals**, where you implement several recursive algorithms to draw recursive graphics; **Grammar Generator**, where you recursively generate text based on certain rules, or "grammars"; and **Human Pyramid**, where you recursively calculate the weight on different members' shoulders in a human pyramid. Each part can be programmed separately, but they should be submitted together. Your output should match exactly as much as possible (e.g. randomization for grammar generator is acceptable, as are slight shifts in fractal output, as long as the images are identical to the expected output by the naked eye). The starter code for this project is available as a ZIP archive (this contains separate QT projects, one for each part of this assignment).

Due Date: October 19 at 11:00am.

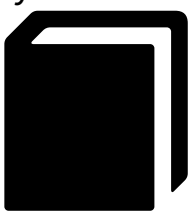
Submit: You can submit multiple times. We will grade your latest submission. Submit via Paperless here.

Important Files and Links

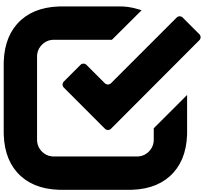


Starter Code

Style Guide



Stanford Library Docs



Text Diff Tool

Image Diff Tool

Turn in only the following files:

1. **`humanpyramid.cpp`**, the C++ code for the human pyramid function (excluding a main function)
2. **`fractals.cpp`**, the C++ code for the fractals functions (excluding a main function)
3. **`grammargenerator.cpp`**, the C++ code for the Grammar Generator program (excluding a main function)
4. **`debugging.txt`**, a file detailing a bug you encountered in this assignment and how you approached debugging it

---

Development Strategy and Hints

Here's some clarifications, notifications, expectations, and recommendations for this assignment.

- ***Your functions must actually use recursion***; it defeats the point of the assignment to solve these problems iteratively!
- ***Do not change any of the function prototypes.*** We have given you function prototypes for each of the problems in this assignment, and the functions you write must match those prototypes exactly – the same names, the same arguments, the same return types. You are welcome to add your own helper functions if you would like.
- ***Test your code thoroughly!*** We'll be running a battery of automated tests on your code, and it would be a shame if you turned in something that almost worked but failed due to a lack of proper testing.
- ***Recursion is a tricky topic***, so don't be dismayed if you can't immediately sit down and solve these problems. Please feel free ask for advice and guidance if you need it. Once everything clicks, you'll have a much deeper understanding of just how cool a technique this is. We're here to help you get there!
- You can compare your output by saving your images to file using the provided GUI, and using the online image comparison tool with the provided output.

Style

As in other assignments, you should follow our Style Guide for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1 and 2 specs, such as the ones about good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style constraints specific to this problem:

Recursion: Part of your grade will come from appropriately utilizing recursion to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Avoid "arm's length" recursion, which is where the true base case is not found and unnecessary code/logic is stuck into the recursive case. Redundancy in recursive code is another major grading focus; avoid repeated logic as much as possible. As mentioned previously, it is fine (sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Variables: While not new to this assignment, we want to stress that you should not make any global or static variables (unless they are constants declared with the const keyword). Do not use globals as a way of getting around proper recursion and parameter-passing on this assignment.

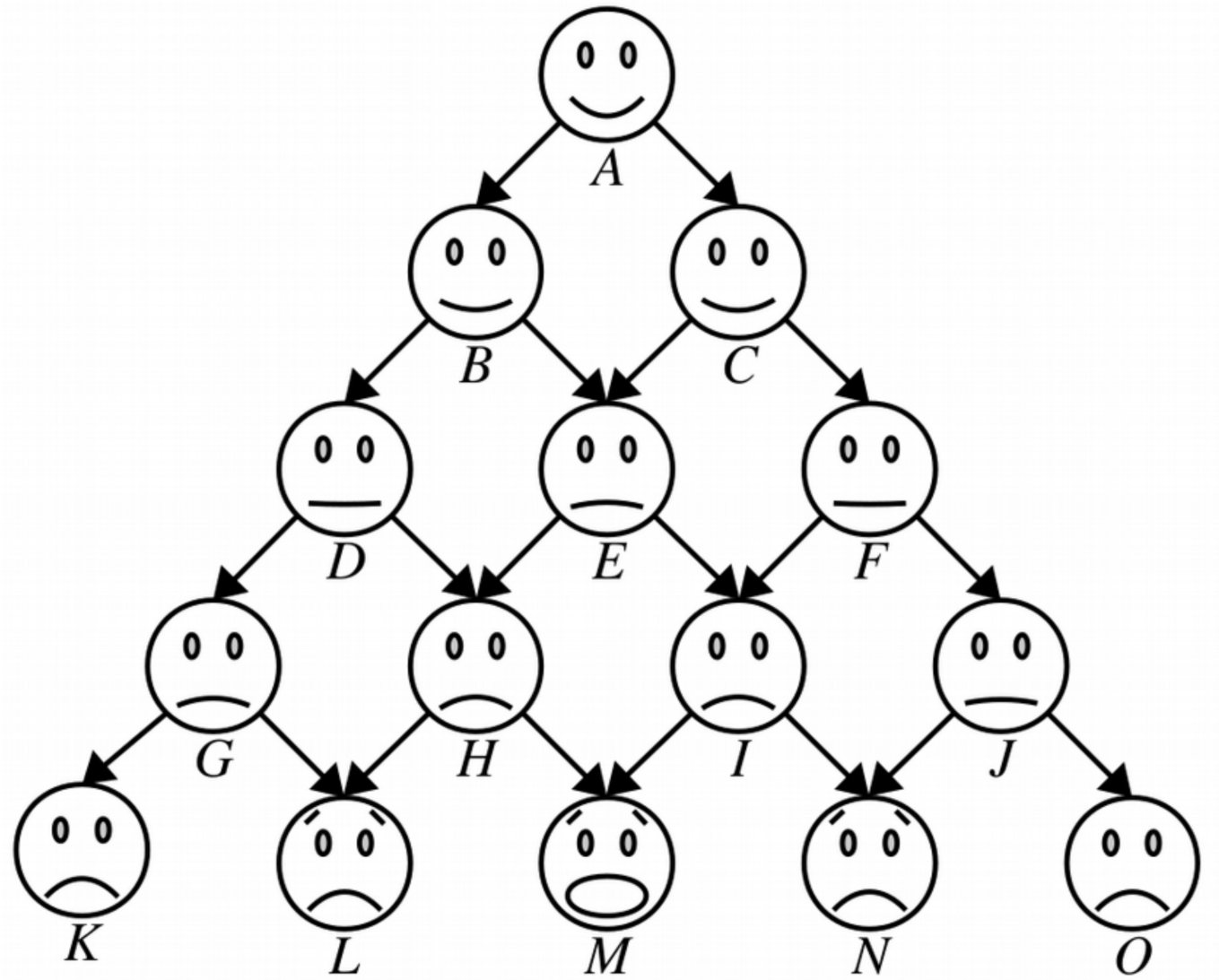
Human Pyramid | Fractals | Grammar Generator | Extra Features | FAQ

Human Pyramid

A human pyramid is a way of stacking people vertically in a triangle. With the exception of the people in the bottom row, each person splits their weight evenly on the two people below them in the pyramid. For example, in the pyramid to the right, person A splits her weight across people B and C, and person H splits his weight – plus the accumulated weight of the people he's supporting – onto people L and M. It can be mighty uncomfortable to be in the bottom row, since you'll have a lot of weight on your back! In this question, you'll explore just how much weight that is. Just so we have nice round numbers here, let's assume that everyone in the pyramid weighs exactly 200 pounds.

Person A at the top of the pyramid has no weight on her back. People B and C are each carrying half of person A's weight. That means that each of them are shouldering 100 pounds. Slightly uncomfortable, but not too bad.

Now, let's look at the people in the third row. Let's begin by focusing on person E. How much weight is she supporting? Well, she's directly supporting half the weight of person B (100 pounds) and half the weight of person E (100 pounds), so she's supporting at least 200 pounds. On top of this, she's feeling some of the weight that people B and C are carrying. Half of the weight that person B is shouldering (50 pounds) gets transmitted down onto person E and half the weight that person C is shouldering (50 pounds) similarly gets sent down to person E, so person E ends up feeling an extra 100 pounds. That means she's supporting a net total of 300 pounds. That's going to be noticeable!



Not everyone in that third row is feeling the same amount, though. Look at person D for example. The only weight on person D comes from person B. Person D therefore ends up supporting

- half of person B's body weight (100 pounds), plus
- half of the weight person B is holding up (50 pounds),

so person D ends up supporting 150 pounds, only half of what E is feeling! Going deeper in the pyramid, how much weight is person H feeling? Well, person H is supporting

- half of person D's body weight (100 pounds),
- half of person E's body weight (100 pounds), plus
- half of the weight person D is holding up (75 pounds), plus
- half of the weight person E is holding up (150) pounds.

The net effect is that person H is carrying 425 pounds – ouch! A similar calculation shows that person I is also carrying 425 pounds – can you see why? Compare this to person G. Person G is supporting

- half of person D's body weight (100 pounds), plus
- half of the weight person D is holding up (75 pounds)

for a net total of 175 pounds. That's a lot, but it's not nearly as bad as what person H is feeling! Finally, let's look at poor person M in the middle of the bottom row. How is she doing? Well, she's supporting

- half of person H's body weight (100 pounds),
- half of person I's body weight (100 pounds),
- half of the weight person H is holding up (212.5 pounds), and
- half of the weight person I is holding up (215.5 pounds),

for a net total of 625 pounds. Yikes! No wonder she looks so unhappy.

There's a nice, general pattern here that lets us compute how much weight is on each person's back:

- Each person weighs exactly 200 pounds.
- Each person supports half the body weight of each of the people immediately above them, plus half of the weight that each of those people are supporting.

Using this general pattern, write a recursive function

```
double weightOnBackOf(int row, int col);
```

that takes as input the row and column number of a person in a human pyramid, then returns the total weight on that person's back. The row and column are each zero-indexed, so the person at row 0, column 0 is on top of the pyramid, and person M in the above picture is at row 4, column 2. For example, **`weightOnBackOf(1, 1)`** would return 100 pounds (since person C is shouldering 100 pounds on her back), and **`weightOnBackOf(4, 2)`** should return 625 (since person M is shouldering a whopping 625 pounds on her back).

Your implementation must be implemented recursively and must not use any loops (**`for, while, or do ... while`**) or ADTs (**`Grid, Vector, Stack`**). We hope that you'll be pleasantly surprised how little code is required!

Speeding Things Up

When you first code up this function, you'll likely find that it's pretty quick to tell you how much weight is on the back of the person in row 5, column 3, but that it takes a long time to tell you how much weight is on the back of the person in row 30, column 15. Why is this?

Think about what happens if we make a call to **`weightOnBackOf(30, 15)`**. This will make two new recursive calls: one to **`weightOnBackOf(29, 14)`**, and one to **`weightOnBackOf(29, 15)`**. This first recursive call will in turn fire off two of its own calls: one to **`weightOnBackOf(28, 13)`**, and another to **`weightOnBackOf(28, 14)`**. The second recursive call fires of two calls: a first recursive call to **`weightOnBackOf(28, 14)`**, and second one **`weightOnBackOf(28, 15)`**.

Notice that there are two calls to **`weightOnBackOf(28, 14)`** here. This means that there's a redundant call being made to **`weightOnBackOf(28, 14)`**, so all the work done to compute that intermediate answer is done twice. That call will in turn fire off its own redundant recursive calls, which in turn fire off their own redundant calls, etc. This might not seem like much, but the number of recursive calls can be huge. For example, calling **`weightOnBackOf(30, 15)`** makes a whopping 601,080,389 recursive calls!

As a final step in this part of the assignment, once you've gotten everything working, modify your function so that it uses memoization to avoid recomputing values unnecessarily. However, the **`weightOnBackOf`** function must still take the same arguments as before, since our starter code expects to be able to call it with just two arguments.

Once you've done that, try comparing how long it takes to evaluate **`weightOnBackOf(40, 20)`** both with and without memoization. Notice a difference? For fun, try computing **`weightOnBackOf(200, 100)`**. This will take a staggeringly long time to complete without memoization – so long, in fact, that the sun will probably burn out before you get an answer – but with memoization you should get back an answer extremely quickly!
