CS106X Autumn 2015

Cynthia Lee

Assignment 3: Recursion Warm-Ups

Assignment handout authors and problem concept contributors include: Cynthia Lee, Marty Stepp, Jess Fisher, Keith Schwarz, and Eric Roberts.

Due: Friday, October 16th at 5:00pm

Pair programming is permitted on this assignment. See course information sheet and honor code.

There are two Recursion assignments: Assignment 3 (Warm-Ups) and Assignment 4 (Boggle). You could really think of them as one assignment, but we separate them out into two due dates to alleviate risk of being overwhelmed at the last minute by such a large and (for many) tricky coding project. After the Assignment 3 due date, there are only 5 days until the Boggle due date, so you should plan to complete work on these warm-ups as soon as possible and, ideally, move on to Boggle before the warm-up deadline.

The assignment has three main purposes:

1. To give you experience applying recursion as a solution to a variety of interesting problem areas.
2. To give you experience applying a variety of recursion patterns.
3. To serve as a "warm up" for the Boggle recursion assignment, a more substantial software engineering endeavor.

General guidelines:

- For the these warm-up exercises, we specify the function prototype. **Your function must exactly match that prototype** (same name, same arguments, same return type). You are welcome to use a wrapper function as a way of structuring your code, as discussed in class.
- **You must use recursion**; even if you can come up with an iterative alternative, we insist on a recursive formulation!

PROJECT 1: HUMAN PYRAMID

(Cynthia Lee and Keith Schwarz)

A human pyramid is a formation of people where participants make a horizontal row along the ground, and then additional rows of people climb on top of their backs. Each row contains 1 person fewer than the row below it. The top row has a single person in it. The image at right depicts a human pyramid with four rows. Before you decide to participate in one of these, you may want to know how much weight you will be supporting.

Your task is to write a function to compute the weight being supported by the knees of a given person in the human pyramid. We will define the weight on a given person *P*'s knees recursively

to be *P*'s own weight, plus half of the weight on the knees of each of the two people directly above *P*. For example, in the pyramid figure at right, the weight on the knees of person I below is I's own weight, plus half of the weight on the knees of persons E and F.

The weight on the knees of persons E and F can be computed recursively using the same process; for example, the weight on the knees of person E is E's own weight, plus half of the weight on the knees of person B, plus half of the weight on the knees of person C. If a given person does not have two people directly above them, any "blank" or "missing" persons should be ignored. For example, the weight on the knees of person F in the figure at right is F's own weight, plus half of the weight on the knees of person C. No truncation/rounding of the result should be done during any of these recursive calculations.

To represent the pyramid we will use a 2-dimensional vector of vectors (we don't use Grid because much of it would be wasted), where each person's own weight in kilograms is stored as a *real number*. So for example, `weights[0][0]` refers to the weight of the person at the top of the pyramid, and `weights[weights.size() - 1][0]` refers to the weight of the bottom-left person in the pyramid. The table below illustrates which person's weight from the above-right figure would be stored in which index of the vector. You can think of it as a left-aligned version of the human pyramid figure.

```
  col     0   1   2   3
row 0   {{A},
    1    {B, C},
    2    {D,  E,  F},
    3    {G,  H,  I,  J}}
```

Our provided code will create the nested vector of weights and pass it to your function. **You may assume** that the vector passed to your function is valid and matches the structure described above, such that the pyramid will always be fully filled in with values in the proper indexes. That is, if there are n people on the bottom row, then there are n-1 people on the next row up, and n-2 people on the next row above that, and so on. Use exactly this function signature:

```
double weightOnKnees(int row, int col,
                     const Vector<Vector<double>>& weights)
```

If the row/column passed is outside the bounds of the vector, return 0. Our starter code provides the overall program and a loop to prompt for the pyramid's size. Here is a sample output:

```
How many people are on the bottom row? 4
Each person's own weight:
51.18
55.90 131.25
69.05 133.66 132.82
53.43 139.61 134.06 121.63

Weight on each person's knees:
51.18
81.49 156.84
109.80 252.82 211.24
108.32 320.92 366.09 227.25
```

Next step (special for CS106X):

Note that the naïve recursive implementation of weightOnKnees will end up calculating the weightOnKnees value for some people in the pyramid many times over. Write a second implementation (same input/output, etc.) that reduces the runtime of your recursive algorithm using memoization. Make a separate function, `weightOnKneesFaster`, with the same signature:

```
double weightOnKneesFaster(int row, int col,
                           const Vector<Vector<double>>& weights)
```

and use it as a wrapper for a recursive function that passes an additional data structure for storing the memos:

```
double weightOnKneesFaster(int row, int col,
                           const Vector<Vector<double>>& weights,
                           Vector<Vector<double>>& memos)
```

You should initialze the memos to be the same size (and 2nd-dimension sizes) as the `weights` vector (and its 2nd dimensions vectors), all "empty" (all value 0.0). As each person's weight on knees is calculated, store it in the corresponding indices location in `memos` and use it to speed subsequent calculations. You may need to test on a very large pyramid before there is a noticeable improvement in speed.

PROJECT 2: SIERPINSKI TRIANGLE *(Marty Stepp)*

For this problem, write a recursive function that draws the Sierpinski triangle fractal image:

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

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

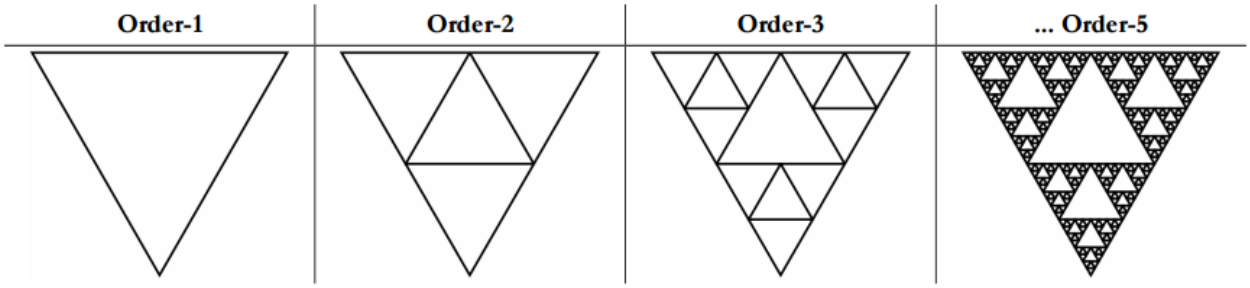
already contain a main function that constructs the window and prompts the user to type an order, then passes the relevant information to your function. The rest is up to you.

If the order passed is 0, your function should not draw anything. If the x, y, order, or size 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.

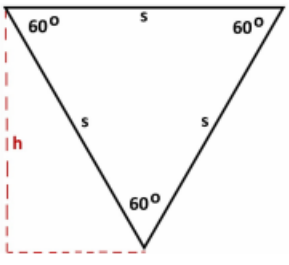
The upward-pointing triangle in the middle of the Order-2 figure is not drawn explicitly, but is instead formed by the sides of the other three triangles. That area, moreover, is not recursively subdivided and will remain unchanged at every order of the fractal decomposition. Thus, the Order-3 Sierpinski Triangle has the same open area in the middle.



The only `GWindow` member function you will need for this assignment is its `drawLine` function:

```
// draws a line from point (x1, y1) to point (x2, y2)
gw.drawLine(x1, y1, 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 on Wikipedia.



Your solution should not use any loops or data structures; you must use recursion. **A particular style of solution we want you to avoid is the "pair of functions" solution**, where you write one function to draw "downward-pointing" triangles and another to draw "upward-pointing" triangles, and each calls the other in an alternating fashion. That is a poor solution that does not capture the self-similarity inherent in this fractal figure. **Another thing you should avoid is re-drawing the same line multiple times.** If your code is structured poorly, you end up drawing a line again (or part of a line again) that was already drawn, which is unnecessary and inefficient.

PROJECT 3: MARBLE SOLITAIRE *(Cynthia Lee)*

The object of the Marble Solitaire game is to clear the board of marbles and leave the last remaining marble in the center position. Each move consists of a marble jumping over one of its orthogonal neighbors into an empty space. The jumped-over marble is cleared from the board. Some of you may have seen a similar game at the Cracker Barrel restaurant chain, where it is presented as a wooden peg board game, rather than with marbles (the board layout is also different).

The provided code will allow you to play the game. Do take a moment to play the game to learn the rules for valid moves (briefly: no diagonal moves, jumps are only one marble wide, no jumping over an already empty space). You will soon discover that it is challenging to solve. After a few failed attempts, you might say to yourself (as I did over winter break in 2013), "I should just write a program to solve this!" Fortunately, you will be doing exactly that for this assignment. You will write code that does a **randomized depth-first (recursive backtracking) search** of all possible moves, starting from the board configuration at the point where the human player gave up and asked the computer to take over.

Your only job is to implement the recursive part of the solver by filling in the body of the `solvePuzzle` function in `marbles.cpp`. Although you should not edit anything else about the code, you may wish to add helper functions to help with decomposition of the `solvePuzzle` function. The `solvePuzzle` function signature is as follows:

```
bool solvePuzzle(const Grid<MarbleType>& board, int marblesLeft,
                 Set<uint32_t>& exploredBoards, Vector<Move>& moveHistory)
```

- `board` is the current game board configuration. More on what `MarbleType` is below, but the idea is that it keeps track of which spaces are currently occupied by a marble, which are free, and which are not playable (*i.e.* the four corners of the board where there are no marbles).
- `marblesLeft` is a count of the marbles remaining on the board (we keep this for convenience so we don't have to count from the `board`).
- `exploredBoards` is a set containing all the board configurations we have already explored. Because it is possible to reach a given board configuration via different sequences of moves, your recursive function should test if we have seen this `board` configuration before. If the current `board` is found in `exploredBoards`., return `false` to avoid repeating work. (Also be sure to add new boards to `exploredBoards`.) Note that the type is `Set<uint32_t>&`, not `Set<Grid<MarbleType>>&`, as you might expect! More on this below.
- `moveHistory` is the sequence of moves that led to the current board (not including human-played moves, if any). These are saved so that if/when a winning sequence is found and the function returns, the original calling function can reproduce the sequence of moves in the graphics display. More on what the `Move` type is below.

What we are providing for you:

- Use the `makeMove` and `undoMove` functions that are provided for you.
- You do not necessarily need to use the `isValidMove` function, which is designed to check human moves for validity (humans can click anywhere and specify all sorts of invalid moves!). As the computer player, you could just write your code to never attempt an invalid move in the first place. That said, if you want to use `isValidMove` to, for example, catch out of bounds moves or moves that attempt to jump over an already empty square, you are welcome to do so.
- Use our system for keeping track of previously explored boards, as described above (see "**`exploredBoards`**" explanation). Note again that we do not store "Grid" type in `exploredBoards.` The reason is that this would consume too much memory. To save memory, we have provided you a way to pack your boards into a single 32-bit unsigned integer (`uint32_t`). You don't need to worry about this, just do the conversion from your Grid-based `board` to the compressed `uint32_t` version using this provided function (found in "`compression.h`"):
  - `uint32_t compressMarbleBoard(const Grid<MarbleType>& board)`
- Take a moment to look in marbletypes.h to learn about these important things you need to use:
  - You should use the **`Move`** `struct` provided in marbletypes.h. A `struct` is a feature of C++ that Java programmers could think of as a super-lightweight class. It is a way of gathering data into one place, but (usually) without the overhead of a constructor, get/set methods, etc. All you need to know is that the `Move struct` contains the data needed to represent one move as four fields: `startRow`, `startCol`, `endRow`, `endCol`. Sample use:

```
Move mymove;
move.startCol = 3;
```

  - You will also need to use the following `enum`, representing the three possible states of a position in the grid that represents the board. An **`enum`** is a C++ feature that is similar to making global constant values (e.g., `static const int N_ROWS`) that allow you to associate a name with some sentinel value. Compared to named constants, `enum` has the added benefit that it collects in one place several possible values a variable could take on:
  -

```
enum MarbleType {
   MARBLE_OCCUPIED, /* has a marble in it */
   MARBLE_EMPTY,    /* could hold a marble but is empty */
   MARBLE_INVALID   /* can't hold a marble (four corners) */
};
```

- Print out a status update message while the computer is playing, to let you know it's still "thinking." It can sometimes take a long time to find a solution, so it is comforting for the user to know that progress is being made. Insert this code at/near the beginning of your `solvePuzzle` function (or you could put it in a helper function and call it from there):

```
if (exploredBoards.size() % 10000 == 0) {
   cout << "Boards evaluated: " << exploredBoards.size()
        << "\tDepth: " << moveHistory.size() << endl;
```

```
    cout.flush();
}
```

Important requirements:

- Your search should be depth-first, which means something roughly like this pseudocode:
  solvePuzzle():
    for each move in the list of possible next moves:
      If solvePuzzle() with the new board after that move finds a solution, then this move was a good move, return true!
      Otherwise, undo the move and continue for loop
- Per the pseudocode above, you will want to have a function that finds all possible next moves given the current board, so you can determine where the recursion should go next. Suggested signature for this helper function (`Move` is defined in the starter code):
  `Vector<Move> findPossibleMoves(const Grid<MarbleType>& board);`
- **Randomization:** Note that there are many possible sequences of moves to win the game, but code with no randomization will always give the same solution given the same starting configuration. Boring! You must randomize the order in which it explores next moves by permuting the ordering of the `Vector` inside the `findPossibleMoves` function. One easy way to do this is to have each insert operation into the vector insert into a random index from `0` to `size()`, instead of always adding to the end.

Debugging using the full board is infeasible because of the number of steps of calculation involved. To give yourself a more manageable test case, experiment with pre-set board states with only a few marbles left (just a few moves away from winning). We have included one like this in the starter code.

EXTRA CREDIT PROJECT: RANDOM SENTENCE GENERATOR

For extra credit, write (or rewrite) the Assignment 2, Project 2, Part 2 (whew that's a mouthful—it's the random sentence generator that used the grammar files) using recursion. Use all the same grammar files, starter code, console interaction, and output behavior—except your solution should use recursion instead of looping. You may choose appropriate data structures to support your solution. Just be sure that your solution is fundamentally recursive, reflecting the inherent recursive nature of the grammar rules.

GRADING AND GENERAL REMARKS (this applies to all parts)

- Note the instructions in the previous assignment about style, pass by reference, passing by `const` reference, and so on.
- Don't forget to cite any sources you used in your comments. Please read the entire CS106 Honor Code on the course website.
- Refer to the course Style Guide for a more thorough discussion of good coding style.
- Do not use pointers, arrays, or STL containers on this program. You should also avoid

expensive operations that would cause you to reconstruct bulky collections multiple times unnecessarily. For example, in N-grams, generate the map of prefixes exactly once; do not regenerate it each time the user asks to generate random text.