Homework 3: Serial and Parallel Sandpiles
Programming for Scientists

1. Set up

1. Inside of your `src` directory, create a directory called `sandpile`. This is where you should place your Go files for this assignment.

2. Assignment

2.1 Sandpiles

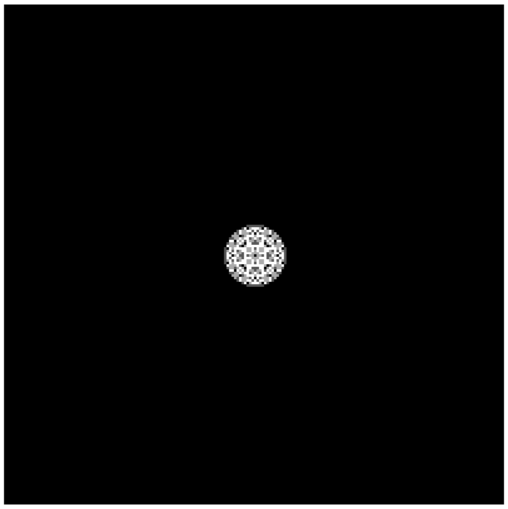
Imagine a huge 2-D checkerboard on which you place piles of coins of various heights on some of the squares (at most 1 pile per square). Consider the following toppling operation:

> topple(r, c): if square (r, c) has ≥ 4 coins on it, move 1 coin from (r, c) to each of the 4 neighbors of (r, c) (diagonal neighbors don't count, only north, south, east, and west).
> If square (r, c) has < 4 coins, do nothing.

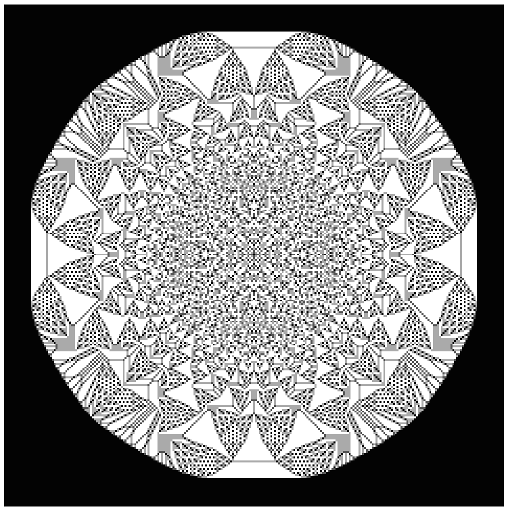
Important Note: if (r, c) lies on the boundary of the board, we will model the topple operation as if the coins fall off the edge of the board. In other words, we do not have to keep track of the coins that fall off, but we should remove four coins from (r, c) even if (r, c) lies on the boundary of the board. This is important, because if we don't account for this assumption, there will be some inputs that cause your program to run forever.

A configuration of coins is said to be **stable** if all squares have < 4 coins on them (i.e., there are no more topple operations to perform). If we repeatedly topple until we can't topple any more, then we will eventually end up at a stable configuration. There is a (surprising!) theorem stating that for a given initial configuration, the order of the topples that you perform won't affect the stable configuration that you reach.

For example, say that you start with a pile of 1000 coins on the center square of a 200 $\times$ 200 board. Then you will obtain the following stable board (zoom in to see the individual pixels):



If you instead start with 100,000 coins on the center square, and no coins elsewhere, then you will end up with the following configuration for a sufficiently large board:



where the color indicates the number of coins (0=black, 3=white, and 1 and 2 are intermediate shades of gray).

These sandpiles have a CMU connection; you can read more about them here:

`http://www.cmu.edu/homepage/computing/2014/fall/lifes-a-beach.shtml`

2.2 What you should do

Write a program that can be run with the following command line (Mac):

```
./sandpile SIZE PILE PLACEMENT
```

where `SIZE` and `PILE` are both positive integers. `SIZE` gives the size of checkerboard which will be `SIZE` $\times$ `SIZE`. `PILE` gives the number of coins that are to be placed on the board.

As for the parameter `PLACEMENT`, it indicates how the coins will be distributed and should take one of two possible string values: `central` and `random`. If the value given is `central`, then the initial board should contain all the coins on the middle square at position ($\lfloor \texttt{SIZE}/2 \rfloor, \lfloor \texttt{SIZE}/2 \rfloor$). If the value given is `random`, then the initial board should select 100 random squares (it is okay if you choose the same one twice) and then randomly distribute the coins over these squares.

You should make two copies of this initial board and then find the stable configuration associated with each board twice. The first time, you should do this serially; the second time, you should do it in parallel. You should time your work as well and print out the time that it takes your board to reach a stable configuration for both the serial and parallel programs.

Your program should print (at least) two lines of the following format:

```
Serial took XXXX seconds.
Parallel took XXXX seconds.
```

where `XXXX` is replaced by the time (in seconds) it takes to find the steady state using the serial and parallel approaches.

Finally, you should draw the final boards into two PNG files, one called `serial.png`, and the other called `parallel.png`. You shouldn't try to do these drawing tasks in parallel, and so you shouldn't include these in your timing. Drawing is just to demonstrate (and test) that the two approaches, which topple coins in completely different ways, are giving you the same result.

The colors corresponding to each number of coins should be, given in (Red, Green, Blue) values:

0	(0,0,0)
1	(85,85,85)
2	(170, 170, 170)
3	(255, 255, 255)

Each board square should be drawn as a single pixel (i.e., a 1×1 square).

Speed. Your program should be fast enough to run `./sandpile 200 10000 central` in at most a few seconds (and `random` in well under a second). It should be able to run `./sandpile 2000 100000 random` in at most 15 minutes (depending on your computer). However, please keep the following quotation in mind:

"*About 97% of the time: premature optimization is the root of all evil.*" — D. Knuth

In other words, it is not a bad idea to start with a working program that works on smaller inputs and then work on ways to speed it up on larger inputs. You can time your functions to find the slowest parts to help identify speedups using the code we have seen for timing functions.

Optional challenge: Draw an animated GIF of your board over a number of "generations". Because there are many different ways to reach the final board, there is no one correct animated GIF, but you may find it fun to view. The GIF may look a little weird just because you will need to color any cell with at least 3 coins on it white.

2.3 Notes on best parallel programming practices

You should get a serial program running before you do any parallel programming. Then use your serial code to design your parallel program.

The best parallel programs for this assignment will look "almost" serial in that they use channels to simply subdivide the work, with each smaller task done serially.

What makes this assignment tricky is that it is not simply a matter of waiting for multiple boards to converge, and then combining them. Regardless of how you divide the board into pieces, you will have coins toppling across the boundaries of the boards. So you need to carefully coordinate how and when your boards communicate across boundaries, being very careful that you are neither slowing yourself down by communicating too much, nor messing up the communication and losing information. The latter is very easy to do if you don't carefully coordinate your channels.

As for subdividing the board, it's important to do so in as pain-free a way as possible. The most natural thing to do is to divide it into `numProcs` pieces of approximate size `numRows/numProcs`, as we have done in class. To avoid race conditions, you should not pass the entire board to a subroutine; instead, split the board into subslices, and pass each subslice to a different subroutine.

Note that if you want to send a "message" corresponding to a row of a matrix, you can use a buffered channel; that is, you might think of a buffered channel of capacity `n` as an array of length `n`.

This assignment is left purposefully open-ended, but perhaps the most natural way to solve this assignment in parallel is to have a highest-level function called

```
func SandpileMultiprocs(board GameBoard, numProcs int)
```

that has `numProcs` calls to a goroutine called `SandpileSingleproc`. While the board has not converged, each of these subroutines topples coins, keeping track of the coins that fall off the top of the board and the bottom of the board using (asynchronous?) channels. These coins will then need to be added to the appropriate cells so that the coins are not forgotten about. (When testing your code, you should make sure that you aren't losing track of coins.)

Something else to reflect on: should your speedup be greater when distributing coins centrally or randomly? Is this what you find in practice?

Finally, don't rig your code so that the parallel portion is more optimized than the serial portion (this will result in a major deduction). We do not expect you to obtain anything close to an ideal speedup, and you may find that you only start observing a parallel speedups for larger datasets. This is OK; that having been said, we should be able to observe a speedup in the parallel version when we run your code on a board that is 1000 by 1000 with 50,000 initial coins.

3. Autograder

The autograder will check that your code complies, runs, and produces the `serial.png` and `parallel.png` files, and writes the timing to the screen. A human grader will check your images. The autograder will have the `canvas` and `gifhelper` packages, and their dependencies installed.

As usual, you should create a gzipped tar file containing the Go files of your program (do not include any folders or subdirectories) to submit to the autograder. On Mac/linux:

```
cd (YOUR HW3 DIRECTORY)
tar czf hw3-answer.tgz *.go
```

4. Learning outcomes

After completing this assignment, you should have:

- learned about sandpiles
- worked on optimizing a program
- worked on timing and testing your own code
- gotten practice with parallel programming
- explored how the speedup obtained by parallel programming can be sensitive to initial conditions