# COM S 327, Fall 2022
## Programming Project 0
## A Modified Abelian Sandpile

The Abelian Sandpile model is one of a class of mathematical games known as *cellular automata*. Perhaps the best known of these is Conway's *Game of Life*, in which the cells represent living organisms which breed and die according to specific rules, and which has been studied for decades and has had countless scholarly articles written about it (and which has had a computer implemented within it!).

Most cellular automata simulate life in some in arbitrary mathematical space, usually $\mathbb{Z}^2$ (two-dimensional, integer Euclidean space). This space could be unlimited (i.e., limited only by available memory) or limited to some fixed size. It could be flat, or mapped to, e.g., a cylinder or a torus. We will play our game in discreet, integral a $23 \times 23$ Euclidean plane.

The Abelian Sandpile is unusual in that, rather than life forms, it simulates something inanimate: small piles of sand, which grow as new grains are dropped on top of them until they become unstable and topple onto neighboring piles.

The rules for the Abelian Sandpile are not based on physics and are very simple:

1. At each step of the simulation, a grain of sand is dropped onto the center pile of the plane.
2. All piles of height $\leq 8$ are stable.
3. Piles of height $> 8$ are unstable, so they topple. Toppling is achieved by removing 8 grains from the pile and depositing them one each on each of the cardinally and diagonally adjacent piles
4. A simulation step ends when all piles are stable.
5. Sand grains that would be placed on piles which would be outside our $23 \times 23$ "world" are removed from the simulation; i.e., they are not placed anywhere.

Additionally, our model will include *sinks*. A sink is a location in the world where sand cannot accumulate. Sand that is spilled into a sink is removed from the simulation. The edges of the world behave like sinks; however, since they are not actually part of the world, they are not technically sinks.

The center cell—where the sand enters the world—may not be a sink.

See this Wikipedia article for more about the Abelian Sandpile model:

https://en.wikipedia.org/wiki/Abelian_sandpile_model.

Your simulation should be implemented on a $23 \times 23$ grid with the grains that fall off the grid leaving the world. Your program should accept an unlimited number of parameters; these parameters, three-tuples in the form "$y\ x\ h$", giving $y$ and $x$ coordinates of piles of height $h$ that exist before the start of your simulation. We call this kind of configuration data *salt*. You do not need to check for errors in the input; you may assume that all input will be well formed. Sinks will be specified in the salt by giving a "height" of $-1$. A sink at the center is invalid; you my implement this by either ignoring a center-cell sink, or by printing an error message and aborting.

At the end of each timestep of the simulation, print out your sandpile in a square, $23 \times 23$ matrix using the height values of each pile and hashes ('#') to representing sinks. Print a blank line at the end, demarcating the edge of one timestep and the beginning of the next. Your simulation should run in an infinite loop; use Control-c to terminate it (no special code is required for this).

Your program should also take an optional frame rate on the command line. Since the frame rate is an integer, just like salt values, we'll need some way to disambiguate it: If the first command line parameter is the string "`--fps`", then the second parameter is an integer frame rate and salt—if there is any—begins with the third parameter; otherwise; salt—again, if there is any—begins with the first parameter.

The default frame rate may be whatever you like. We'll use `usleep()` to pause execution between frames. `usleep()` takes an integer parameter in microseconds, so your delay will be $1000000/\text{frame rate}$[1].

Here follows an example simulation. For illustration purposes, the example world is $5 \times 5$. We run the program with some salt as follows (assume the binary is named `sandpile`, and note that $x$ grows from left to right and $y$ grows from **top to bottom**):

```
./sandpile 2 2 7 3 2 8 4 2 8 2 3 -1
```

The world is salted with a sandpile of height 7 at $(2, 2)$, piles of height 8 at $(3, 2)$ and $(4, 2)$, and a sink at $(2, 3)$[2], giving:

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 7 | # | 0 |
| 0 | 0 | 8 | 0 | 0 |
| 0 | 0 | 8 | 0 | 0 |

A grain of sand is dropped on the center-most cell, and after one iteration the pile looks like:

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 8 | # | 0 |
| 0 | 0 | 8 | 0 | 0 |
| 0 | 0 | 8 | 0 | 0 |

Another grain is dropped, causing a domino effect of toppled tiles with three grains falling off the bottom edge and two into the sink, yielding:

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 2 | 2 | # | 0 |
| 0 | 3 | 2 | 3 | 0 |
| 0 | 2 | 1 | 2 | 0 |

An efficient solution will require a queue (or a stack), or could be implemented recursively. A solution that probes the whole world for unstable piles repeatedly until there are none will be very slow. You are not required to implement an efficient solution for full credit; slow is fine (but fast is better!).

See the syllabus for information about what to turn in and submission format. In particular, you must write, use, and turn in a *Makefile*, and you must include a *README*!

[1] There is an additional delay each frame to compute, thus our framerate is actually a bit slower than specified. There are other, more complicated way to implement frame timing that avoid this drift. The method specified here is sufficient for our needs

[2] It's pretty common (not universal) in computing APIs for $y$ dimensions to come before $x$ dimensions. I know it seems odd at first, but it is what it is, and there are sensible reasons for it, so get used to it.

**Extra Challenges (nothing below this line is required)**

- Modify your world so that your simulation runs on a cylinder.
- Modify your world so that your simulation runs on a torus. There will be no end of the world, so eventually your simulation will reach a secondary infinite loop where it will never finish toppling unstable piles.
- Modify your world so that sand can fall from arbitrary locations.
- Modify your rules by redefining what it means to be stable and what it means to topple.
- Make the world size a command line parameter.
- Use *curses* or *ANSI escape codes* to color your output.
- Encode your output using images with colored pixels representing the piles instead of numbers (`NetPBM` is an easy target format, here).
- Encode your output into a video.
- Other fun ideas of your own device.

Any *extra challenges* you choose to implement should be *off* by default, able to be turned *on* by way of a command line switch and not interfering with any of the required functionality described above.