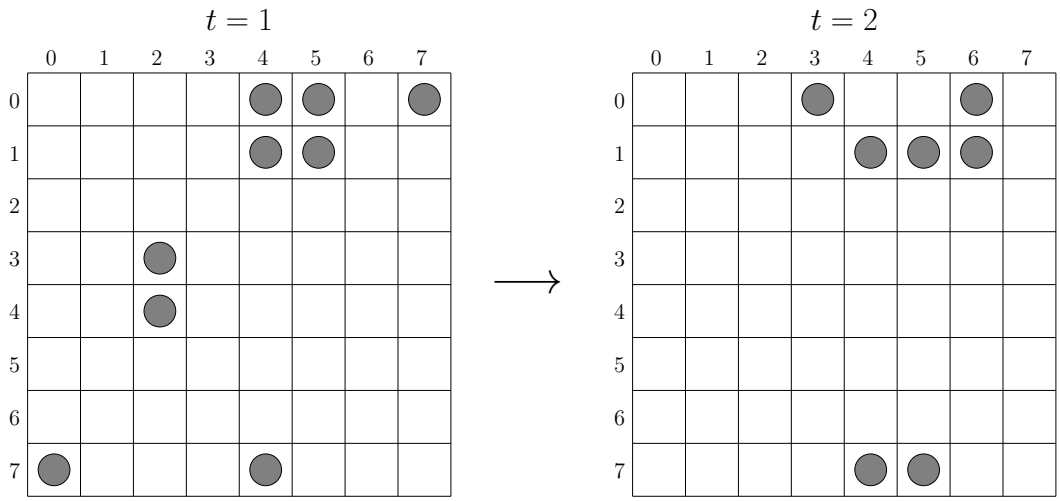
Homework 11: Conway's Game of Life

Conway's Game of Life is an example of how complex behavior can sometimes arise from systems governed by a simple set of rules. The 'game' takes place on a grid of squares called *cells*. Each cell is initially designated as either alive or dead. In the figure below, the empty cells are the dead ones and the others are alive.



The game begins at time (or generation) $t = 1$, and is updated according to the following rules:

A cell which is alive at time t will be:

- dead at time $t + 1$ if it has 0 or 1 live neighbors (it dies of lonliness),
- alive at time $t + 1$ if it has 2 or 3 live neighbors,
- dead at time $t + 1$ if it has 4 or more live neighbors (it dies of overpopulation).

A cell which is dead at time t will be:

- alive at time $t + 1$ if it has exactly 3 live neighbors (by reproduction),
- dead, otherwise.

The 'neighbors' of a cell are the 8 cells that surround it. In the figure above, the cell at row 3, column 2 is alive. Among its 8 neighbors, only one is alive (the cell at row 4, column 2). Since the (3,2) cell has only one live neighbor, according to the rules above it will be dead in the next generation.

The cell at row 1 column 6 is currently dead. It has exactly 3 live neighbors, though, so it will become alive in the next generation.

The borders of the grid may be handled in any number of ways for this game. But for this assignment, we'll play the *torodial* version of the game. It is so named because we're going to connect the top of the grid to the bottom and the left side to right in such a way as

to form a torus. For example, the cell at row 0, column 5 still has eight neighbors in total. They are cells $(0,4),(0,6),(1,4),(1,5),(1,6)$ as well as the three cells we find by wrapping around to the bottom, $(7,4),(7,5)$, and $(7,6)$. So this particular cell at $(0,5)$ has 4 live neighbors and will die in the next generation.

If you would like to use this example to help test your own code, the `input.txt` file would be:

```
00001101
00001100
00000000
00100000
00100000
00000000
00000000
10001000
```

---

1. (10 points) Complete the method `def neighbors(self, i, j):` which counts the number of cells neighboring (i,j) that are alive and returns this value. You should 'wrap around' the edges (as described in the presentation), so that every cell has exactly 8 neighbors. For example, in a 10×10 grid, cell $(0,3)$ would have as its neighbors the cells

$$(9,2),(0,2),(1,2),(9,3),(1,3),(9,4),(0,4),(1,4).$$

Hint: The modulo operator `%` can be very helpful for this purpose.

2. (20 points) Complete the method `def nextGeneration(self):` which updates `board` to the next generation. Note: you will have to create a temporary variable, say `newBoard`, which will represent the cells in the next generation. For each cell, figure out if it will be alive or dead in the next generation and update `newBoard` accordingly. After that is completely done, then just set `self.board = newBoard`. The reason for this is that every cell must be updated at exactly the same time; but if you start changing values in `self.board`, it will affect neighbor counts for other cells before you've had the chance to update them!
Submit a screenshot of the Generation 800 that results from using the accompanying input file `input.txt`.