

21

Game of life

15 points

Introduction

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

The game is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

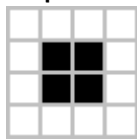
We want to simulate such experiment, by creating a simulator of this Game of Life.

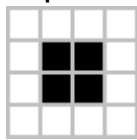
The rules of the game are simple.

We have a board of $N \times N$ size, that will be filled with life or dead cells (# or .). After each turn, every state of the cell will be determined by the previous state according to the following rules:

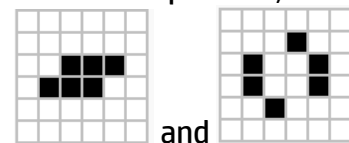
1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors' lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

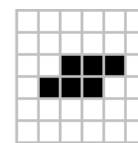
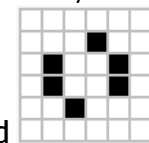
The neighbors of a cell A are determined by all the surroundings cells of the cell A (vertically, horizontally and diagonally). Some patterns will survive forever, other may oscillate among different status and other may even move or create other patterns.



This will be a stable pattern, . As you can see all the dead cells will remain dead as they only have 1 or 2 neighbors, and the already alive cells will always live because they have 3 neighbors.

An oscillator is a pattern that returns to its original state, in the same orientation and position, after a finite



number of turns. The toad is a repeating pattern that oscillates between  and . Those cells that appear, do so because they have exactly 3 neighbors, and those cells that die do so because they have 4 or more neighbors or they have 1 or less neighbors.

Following these rules, we can create complex and bigger patterns.

Your goal will be create an application that can load a board of size $N \times N$ from the input, and will produce as output the final state of the board after m turns.

You can see some examples of input and output in the following sections.



HINT: Don't forget that the total size of the board can be up to 100 x 100, so be careful with sizes, and don't forget to be careful when updating your board (wink, wink). By the way, doing it by hand could take LOTS of times (especially if we will test it with high numbers of iterations...).

Input

< m that represents the number of turns >
 < N that represents the board size >
 < board data matrix displayed as N rows x N columns >

Output

Print out the resulting board data.

Example 1

Input

```
2
5
. . . . .
. . # # .
. . # # .
. . . . .
. . . . .
```

Output

```
. . . . .
. . # # .
. . # # .
. . . . .
. . . . .
```

Example 2

Input

```
2
5
# . # # .
# . . # .
. . # # .
. . . . .
. . . . .
```

Output

```
. . # # .
. # . . #
. . . # .
. . . . .
. . . . .
```

