11 Cellular Automaton

Introduction

Imagine a fire in a forest, like that shown in the satellite image. If we know where the fire started, then

- in what direction is it likely to burn?
- what is the probability of the fire consuming the entire forest?
- how few trees do we need to fell to be sure that a single fire will not consume the entire forest?

Or consider the spread of an infectious disease in a city — will everybody get ill? Or consider the spread of a rumour in the school yard — how soon until everybody hears the rumour?

All three problems, seem very different, but all can be studied using *Cellular Automaton*.

Cellular Automaton

- We take our world (the forest) and split it into separate regions, called *cells*.
- Each cell is treated as a single unit, and is classified in a fixed set of *states*. For example, in a forest fire model each cell is classified as

$$\text{states} = \{\text{unburnable}, \text{burnable}, \text{burning}, \text{burnt}\}$$

- We need a set of rules that describe how the state of the cells change in response of their current state and the state of their neighbouring cells.

  For example, in a forest fire model:

  - a cell can change from state burnable to burning, when fire spreads from a neighbouring cell, but cannot change to states unburnable or burnt.
  - a cell can change from state burning to burnt, when all the fuel (trees) in the cell are consumed, but cannot change to states unburnable or burnable.

- Finally we setup our model with some initial conditions (say, a fire in a random cell), and we get the computer to do the work of simulating what will happen over time.

Some of the material covered here came from the following paper, but google ”cellular automation applications” for more example, in particular search for the Game of Life.

- **A Cellular Automata Model for Fire Spreading Prediction**
  by Quartieri, J. and Mastorakis, Nikos and Iannone, Gerardo and Guarnaccia, Claudio. International Conference on Urban Planning and Transportation, 2010.

CoderDojo, Tramore, Waterford. (kmurphy@wit.ie)

1 Before we start ...

We need to install two python packages before we can start our simulation. The two packages are:

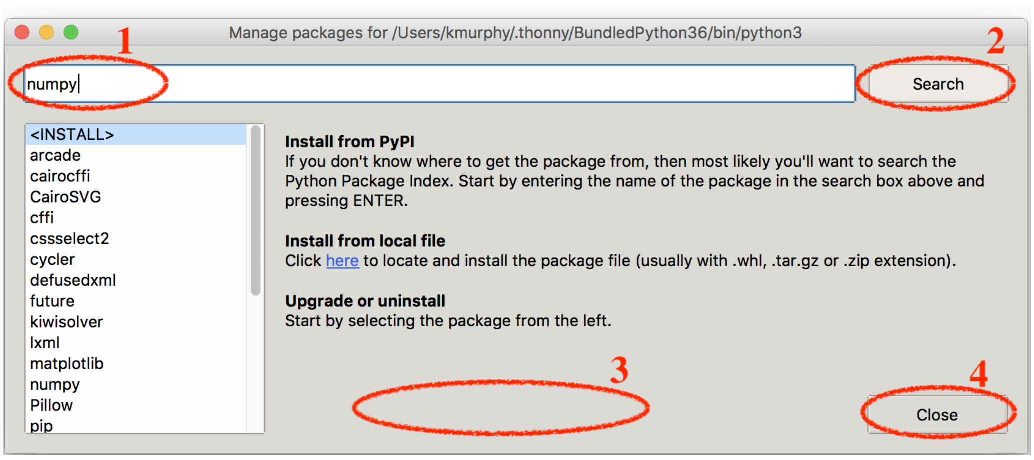
- `numpy`
  This is a replacement for python lists, that can run up to 100s of times faster. This is important in cellular automation where we want lots of cells.

- `matplotlib`
  This is a library for drawing graphs, we could use `turtle` package for this but `matplotlib` will make our life easier.

Installing packages on Thonny

To install a python package in Thonny, select menu option `Tools`→`Manage Packages...` Then:

1. Enter the package name in the search box (`numpy` or `matplotlib`).
2. Click on `Search`.
3. An `Install` button should appear and click on this.
4. When installed both packages, exit by clicking on `close`.



To check that the installation of the packages worked, type in the following program and verify the output.

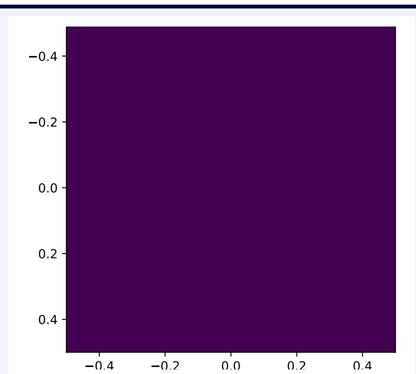
All this code does in create a world consisting on 1×1 cells, and displays it.

test_packages.py

```python
import numpy as np
import matplotlib.pyplot as plt

model  = np.zeros((1,1), int)

plt.imshow(model)
plt.pause(1)
```

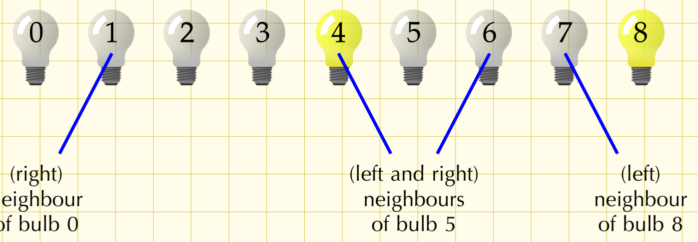


2 One-Dimensional (1D) Cellular Automation

Before we get to the forest fire model we will use the following simpler problem to introduce the main concepts.

Line of Bulbs

Consider a line of bulbs. Each bulb can be either off or on — i.e. in cellular automation terms, each bulb is in one of two **states**. One possible configuration of states is shown below:



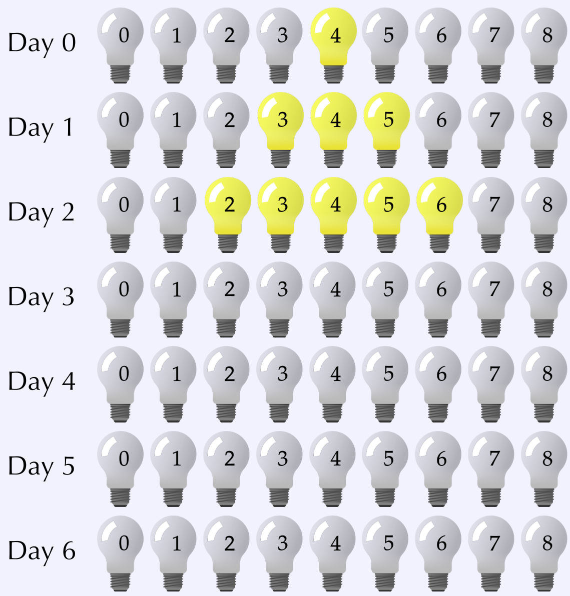
Every day we update each bulb, turning it off or on, or leaving it unchanged based on its current state and the state of its neighbours — this is our **update rule**.
Depending on the update rule, over time which bulbs are on and which bulbs are off?

2.1 Before we try to tell a computer to ...

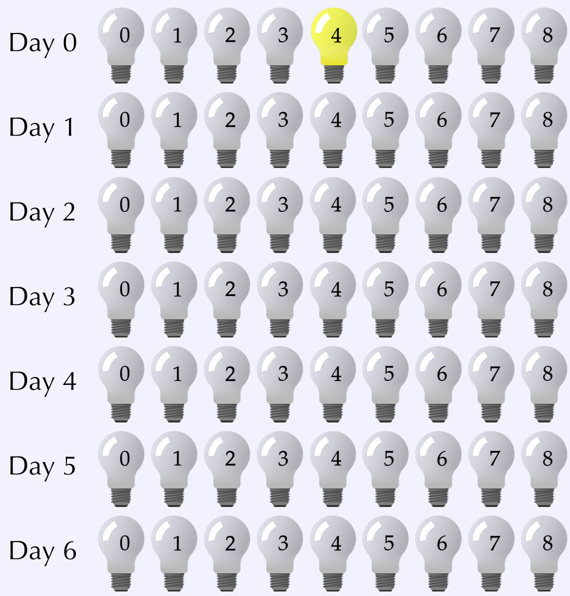
In programming circles you often hear the advice "before we try to tell a computer to do something we should make sure that we understand exactly what we are trying to do". So let us think about this problem by hand first.

So consider both of the following and colour the bulbs that are on each day according to the given rules. (The first example is half-done to get you started.)

Leave on any bulb that is on, and turn on any bulb that has at least one neighbour that is on.

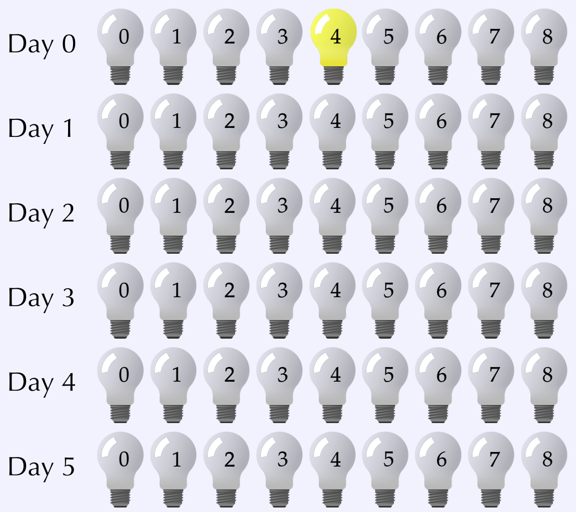


Turn on any bulb that has at least one neighbour that is on. If a bulb is on, then turn it off.

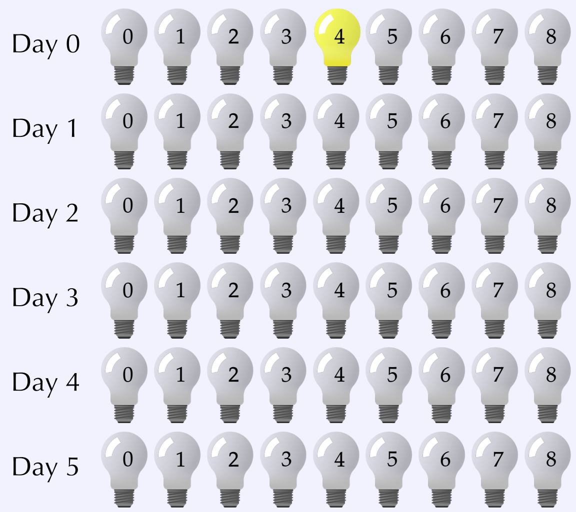


Colour the bulbs that are on each day according to the given rules.

Bulb is on if right neighbour is on, otherwise bulb is off.

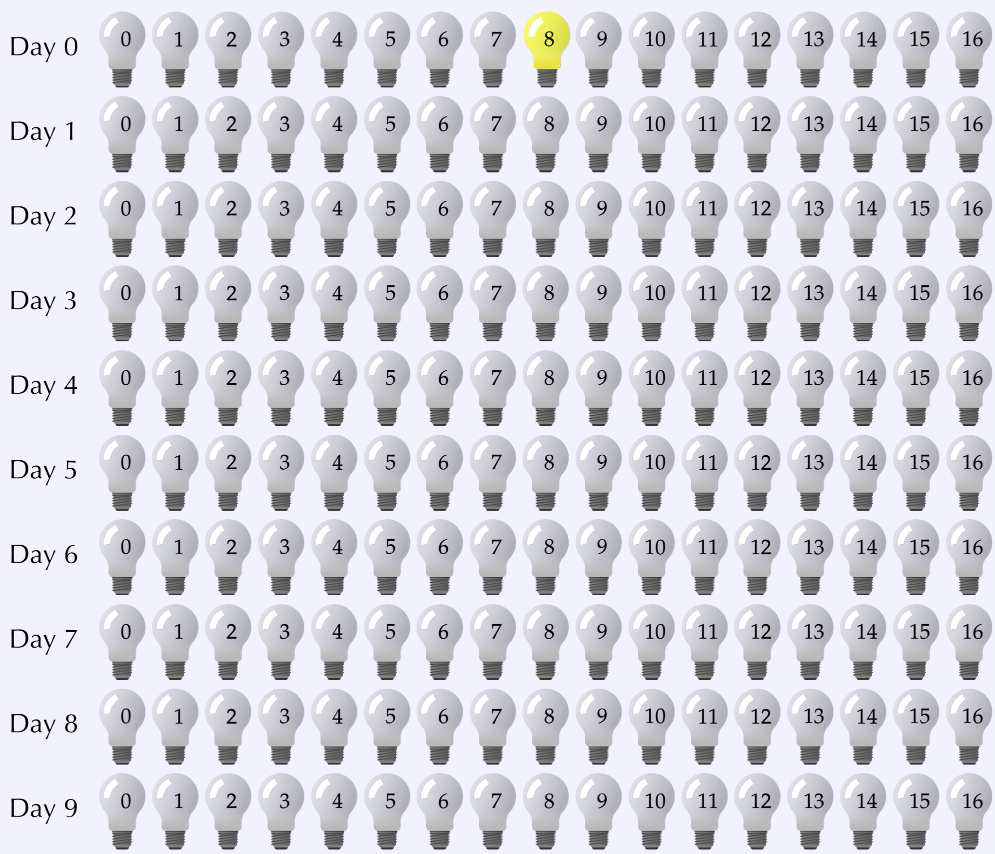


Bulb is on if left neighbour is on, otherwise bulb is off.



Notice that, the bulbs at either end have only one neighbour. This becomes important for some rules. For example, in the following example we only update the bulbs in the middle because the rule does not make sense at either end.

Turn on any bulb that has exactly on one neighbour that is on, otherwise turn bulb off.



2.2 Step 0 — Import the desired packages

Create a new file `bulbs_1d.py` and enter the following code.

bulbs_1d.py
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
```

The first two line load the two packages that we installed earlier, `numpy` and `matplotlib`, and we give then shorter names of `np` and `plt` to make life easier for us later. The third line loads a function used to set the colours to indicate a bulb is off (black) or on (yellow).

2.3 Step 1 — Create world

Let us assume we have 9 bulbs — we can change this later — then append the following code to `bulbs_1d.py`

bulbs_1d.py
```
5  # STEP 1 - setup model
6  n = 17
7  states = 2
8  off, on = 0, 1
9  colors = ListedColormap(['white','yellow'])
10
11 model = np.zeros(n, int)
12
13 print (model)
```

- In line 6, we store the number of bulbs in variable `n`, this will make things easier later when we want to change the number of bulbs later.
- In lines 7, 8 and 9, we define the states that we need. Here `off=0` and `on=1`, and the colours used to represent them.
- In line 11, we allocate space to store the states of the `n` bulbs. The function `np.zeros` creates a list of zero to the size we want.

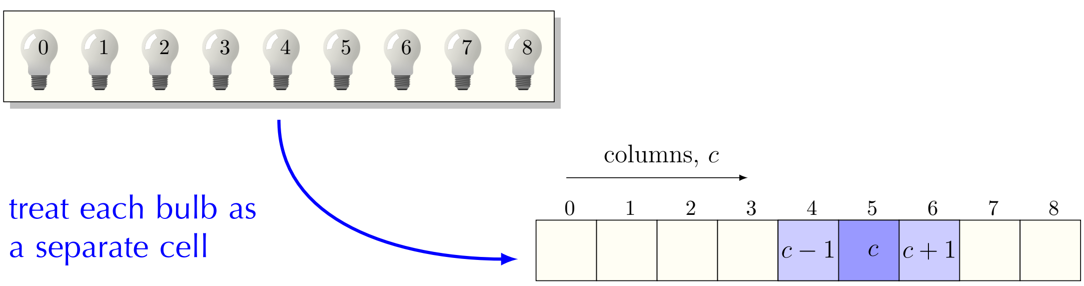


Figure 1 – A 1D cellular automation consists of a row of cells. Each cell is identified by its column number (like houses on a street), but starting from the left with cell zero.
If we ignore the start and end cells, then every cell has two neighbours, one to the left (the west neighbour) and one to the right (the east neighbour). And we can get the address of the neighbours by adding/subtracting 1.

Run the above code and verify that we see a line of zeros.

2.4 Step 2 — Initialise model

At the start the middle bulb is on. This is called our **initial conditions**.

Append the following code to `bulbs_1d.py`, and run the above code and verify that we see a line of zeros, with a single one (bulb on) in the centre.

bulbs_1d.py

```
# STEP 2 - initialise model
model[n//2] = on

print (model)
```

2.5 Step 3 — Display model

Append the following code to `bulbs_1d.py`, and run the program, you should see the graphic shown below.

bulbs_1d.py

```
20 # STEP 3 - display model using 'nice' graphics
21 def show_model(t, duration):
22     plt.imshow(model.reshape((1,n)), vmin=0, vmax=states-1, cmap=colors)
23     plt.title("1D Cellular Automoation (day %s)" % t)
24     plt.pause(duration)
25
26 show_model(0, 1)
```

- In line 21–24 we define a function to draw our model.
  - Line 22, displays the contents of `model` using function `plt.imshow`. The function `plt.imshow` is a general image viewer function and takes lots of parameters. We will cover this in more detail later.
  - Line 23, adds a title to the image.
  - Line 24, displays the graphics and pauses for 1 second.
- Line 26, tests our function (always a good idea) and shows the starting state for one second.

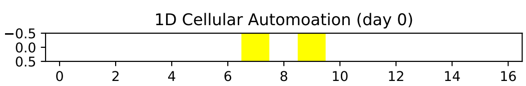


Figure 2 – Start point (initial state) for bulbs — only centre bulb is on.

Modify the code so that the leftmost bulb and the right most bulb is also on. You should get the following graphic:

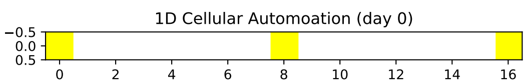


Figure 3 – Centre bulb and both end bulbs are on.

2.6 Step 4 — Run simulation

Finally, we come to the simulation bit — this is where our work to date will pay off and the computer now does all of the heavy work for us.

Append the following code to `bulbs_1d.py`

bulbs_1d.py

```
28 # STEP 4 - simulate the model
29 for t in range(n//2-1):
30 
31     # STEP 4a - update model based on current state of cells
32     current = model.copy()
33     for c in range(1,n-1):
34         model[c] = current[c+1] != current[c-1]
35 
36     # STEP 4b - display model
37     show_model(t+1, 0.1)
```

- Lines 27 to 29 apply the rule. In this case we turn on a bulb when its two neighbours are different (i.e., one neighbour is off and the other neighbour is on).

$$\text{neighbours of cell } c \text{ are at } \underbrace{\text{address } c-1}_{\text{left neighbour (west)}} \text{ and } \underbrace{\text{address } c+1}_{\text{right neighbour (east)}}$$

  Notice two important things of this step:

  - before we started to update each of the cells (stored in `model`), we first had to make a copy of their current state, and saved this to `current`.
  - Our rule applies to cells with two neighbours — so we don't update the leftmost and rightmost cells.

- Lines 37 display the model and pauses for 0.1 seconds, before continuning the simulation.

Some Quick Check Puzzles

Save file as `bulbs_1d_do_nothing.py` and modify the rule so that the all we see is a single light bulb in the centre.

Save file as `bulbs_1d_one_right_only.py` and modify the rule so that the all we see is a single light bulb on moving to the right.

Save file as `bulbs_1d_one_left_only.py` and modify the rule so that the all we see is a single light bulb on moving to the left.

Save file as `bulbs_1d_grow.py` and modify the rule so that we see the number of bulbs on growing outwards from the centre.

Save file as `bulbs_1d_flash.py` and modify the rule so that we see the flashing bulbs on and off every day.

3 Storing History in 1D Cellular Automation

Ok, we now have a working simulation now but it could be improved. One improvement is that we could save the history of the bulbs every day and at the end display this history. Showing the history would make it easier to track what is happening and allow us to test more complicated rules.

Save your `bulbs_1d.py` as `bulbs_1d_history.py` and insert the changes below.

bulbs_1d_history.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# STEP 1 - setup model
n = 17
tmax = n//2

states = 2
off, on = 0, 1
colors = ListedColormap(['white','yellow'])

model = np.zeros(n, int)
history = np.zeros((tmax,n), int)

# STEP 2 - initialise model
model[n//2] = on
history[0] = model

# STEP 3 - display model using 'nice' graphics
def show_model(t, duration):
    plt.imshow(model.reshape((1,n)), vmin=0, vmax=states-1, cmap=colors)
    plt.title("1D Cellular Automoation (day %s)" % t)
    plt.pause(duration)

show_model(0, 1)

# STEP 4 - simulate the model
for t in range(tmax-1):

    # STEP 4a - update model based on current state of cells
    current = model.copy()
    for c in range(1,n-1):
        model[c] = current[c+1] != current[c-1]
    history[t+1] = model

    # STEP 4b - display model
    show_model(t, 0.1)

plt.close()
plt.imshow(history, vmin=0, vmax=states-1, cmap=colors)
plt.title("1D Cellular Automoation (history)")
plt.pause(1)
```

number of days to run simulation. The more days we simulate the more history we need to remember

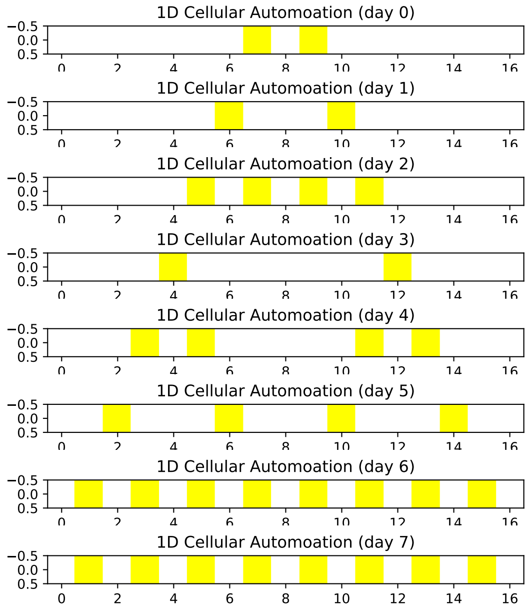
allocate space to store history

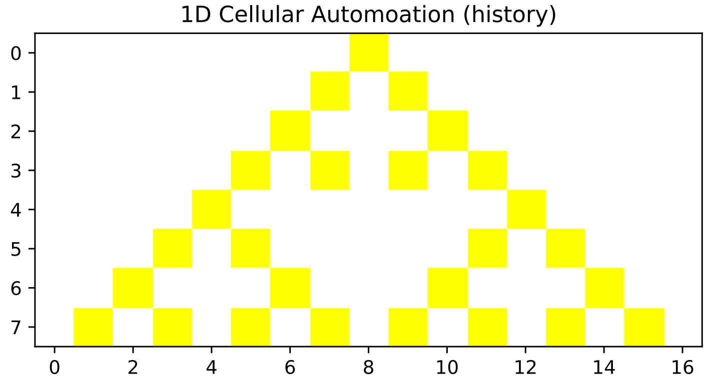
store initial state into history

store model into history

display history

When you run your `bulbs_1d_history.py` program, you should see the sequence of images showing the daily state (bottom left) and finally the complete history (bottom right).

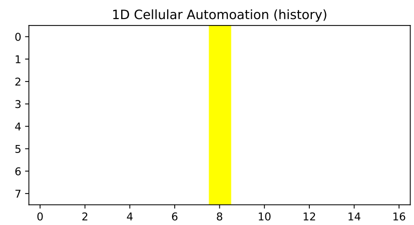




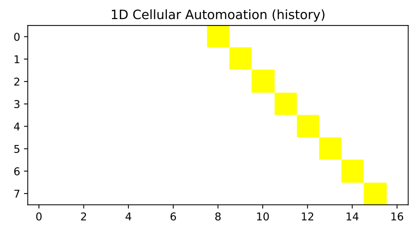
Some Quick Check Puzzles

Modify your `bulbs_1d_history.py` program, changing either the initial condition and/or the update rule to generate each of the following effects.

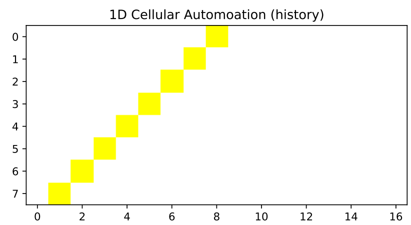
A



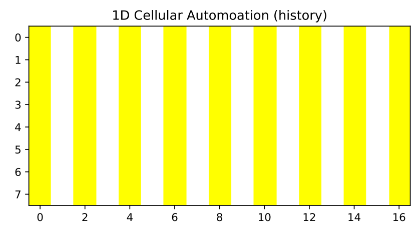
B



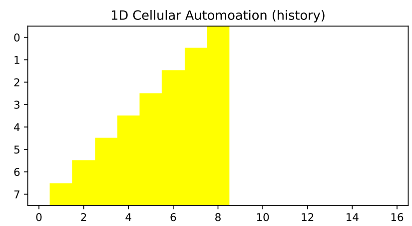
C



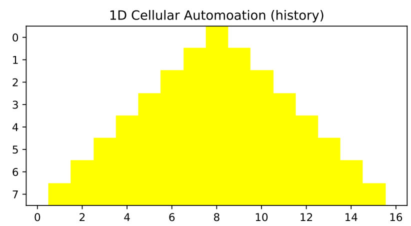
D



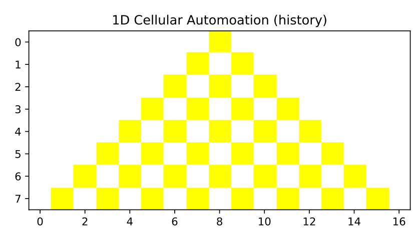
E



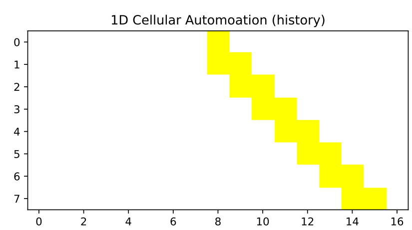
F



G



H



I