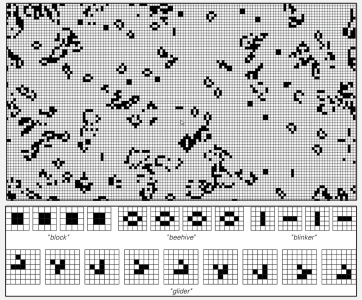
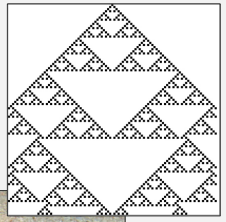
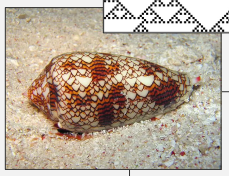
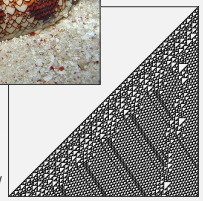
Cellular Automata

- another example of systems where simple short-range rules give rise to complex emergent behavior









natureofcode.com/cellular-automata/
https://en.wikipedia.org/wiki/Rule_110
https://plato.stanford.edu/entries/cellular-automata/
https://www.forbes.com/sites/startswithabang/2017/09/26/it-from-bit-is-the-universe-a-cellular-automaton/
https://www.wolframscience.com/nks/notes-6-8--structures-in-the-game-of-life/

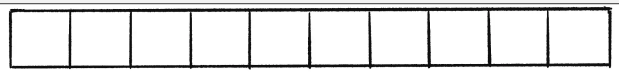
Cellular Automata

A *cellular automaton* has four key elements:

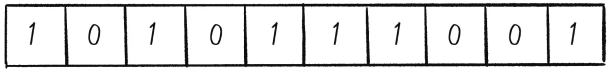
- a collection of cells arranged in a grid
  - there can be any (finite) number of dimensions
- each cell has a state
  - there are a finite number of possible states
- each cell has a neighborhood
  - typically all cells adjacent to that cell
- a ruleset defining how a cell's state changes from one generation to the next

Elementary CAs

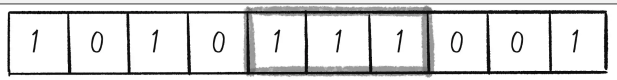
- the simplest grid is a 1D line of cells



- the simplest set of states is two states – 1/0, on/off, true/false, alive/dead

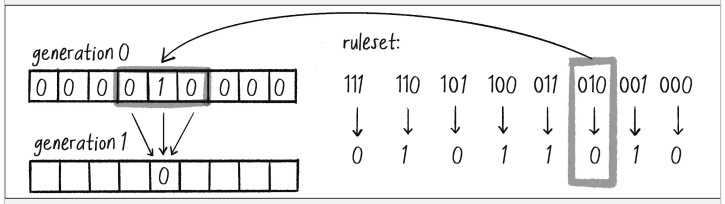


- the simplest neighborhood is the two immediate neighbors (left and right)

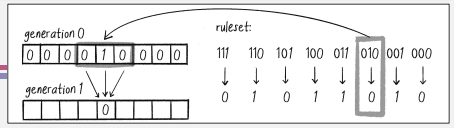


Elementary CAs

- the ruleset defines how a cell's state changes from one generation to the next



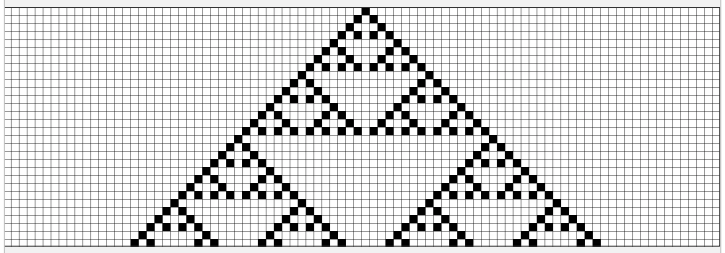
Elementary CAs



- the evolution over time can be shown by drawing each generation on successive rows, using black for 1 and white for 0

gen 0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
gen 1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
gen 2	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
gen 3	0	0	0	0	1	0	1	0	1	0	1	0	0	0	0	0
gen 4	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
gen 5	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0
gen 6	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
gen 7	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
gen 8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

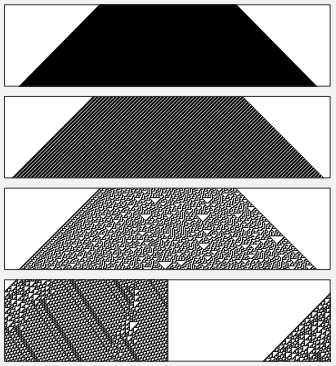
Elementary CAs



Elementary CAs

Different rulesets lead to different kinds of behaviors.

- class 1 – uniformity
  - after enough time, every cell ends up in the same state
- class 2 – repetition
  - after enough time, cell states cycle through a repeating pattern
- class 3 – random
  - no discernable pattern
- class 4 – complexity
  - repetitive patterns appear randomly



2D Cellular Automata

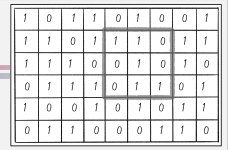
- the line of cells becomes a grid of cells
- the neighborhood contains all adjacent cells

1	0	1	1	0	1	0	0	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	1	0	1	0
0	1	1	1	0	1	1	0	1
1	0	0	1	0	1	0	1	1
0	1	1	0	0	0	1	1	0

Conway's Game of Life

- John Conway, 1937-2020
  - British mathematician
  - known for
    - the Game of Life
    - combinatorial games (sprouts, phutball, Conway's soldiers, angels and devils)
    - the Doomsday algorithm for determining the day of the week for a given date
    - results in a variety of other areas (geometry, geometric topology, group theory, number theory, algebra, analysis)

Conway's Game of Life

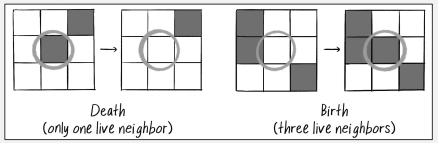


Rules –

- a living cell (state 1) dies (state 0) if –
  - it has four or more living neighbors (overpopulation)
  - it has one or fewer living neighbors (loneliness)

  otherwise it remains alive
- a dead cell (state 0) comes to life (state 1) if –
  - it has exactly three living neighbors

  otherwise it remains dead

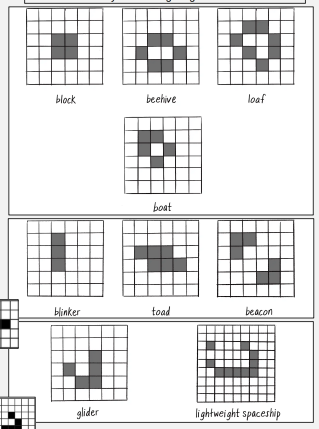


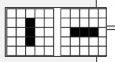
Conway's Game of Life

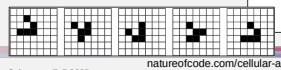
- a living cell (state 1) dies (state 0) if –
  - it has four or more living neighbors (overpopulation)
  - it has one or fewer living neighbors (loneliness)
- a dead cell (state 0) comes to life (state 1) if –
  - it has exactly three living neighbors

Common patterns –

- still lifes
  - fixed patterns which do not change from generation to generation
- oscillators
  - patterns which return to their initial state after a finite number of generations
- spaceships
  - patterns which move across the grid

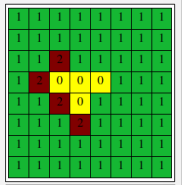


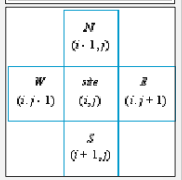




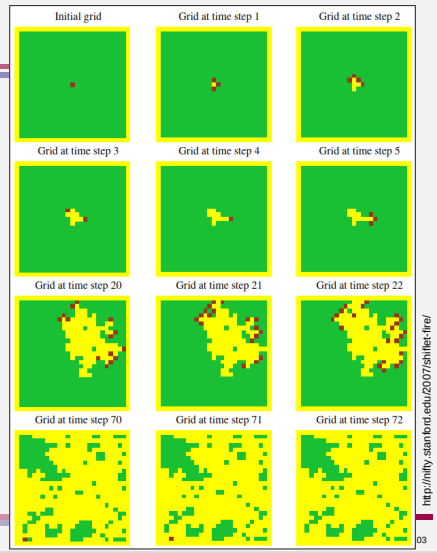
Fire Simulation

- 2D cellular automaton
- three cell states
  - 0 (empty) – the cell is empty ground or contains a burnt tree
  - 1 (tree) – the cell contains a tree that is not burning
  - 2 (burning) – the cell contains a tree that is burning
- the neighborhood is only the cells directly north, south, east, west
- rules
  - if a cell is empty, it remains empty
  - if a cell is burning, it becomes empty
    - (it only takes one time step to fully burn a tree)
  - if a cell contains a tree and has at least one burning neighbor, it catches fire with a probability p_{catch}
  - otherwise the cell remains in the same state

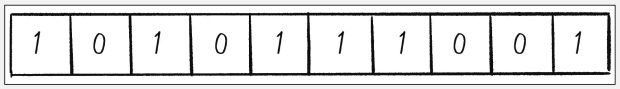




Fire Simulation



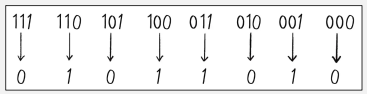
Implementing Elementary CAs



- a line of cells becomes an array
- two possible states means the base type is `boolean` (`true`=1, `false`=0)
- initialization options
  - set the middle cell to `true` and the rest to `false`
  - initialize each cell randomly (`true` or `false`)

```
if ( random(1.0) < 0.5 ) { … }
else { … }
```


Implementing Elementary CAs



- a ruleset can be implemented with an array
  - assume the neighborhoods are always listed in the order shown (111, 110, 101, …)

```
boolean[] ruleset = { false, true, false, true, true,
                      false, true, false };
```

  - this *initializer list* syntax declares an array variable `ruleset`, creates 8 slots, and initializes the slots all in one step

Implementing Elementary CAs

- computing the next generation

For every cell in the array:

1. Take a look at the neighborhood states: left, middle, right.
2. Look up the new value for the cell state according to a ruleset.
3. Set the cell's state to that new value.

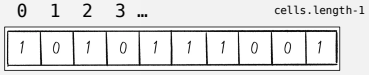
this code has bugs!
do not use without modifications!

```
for ( int i = 0 ; i < cells.length ; i = i+1 ) {
  boolean left = cells[i-1];
  boolean middle = cells[i];
  boolean right = cells[i+1];

  boolean newstate;
  if ( left && middle && right ) { newstate = ruleset[0]; }
  else if ( left && middle && !right ) { newstate = ruleset[1]; }
  else if ( left && !middle && right ) { newstate = ruleset[2]; }
  else if ( left && !middle && !right ) { newstate = ruleset[3]; }
  else if ( !left && middle && right ) { newstate = ruleset[4]; }
  else if ( !left && middle && !right ) { newstate = ruleset[5]; }
  else if ( !left && !middle && right ) { newstate = ruleset[6]; }
  else { newstate = ruleset[7]; }    // !left && !middle && !right

  cells[i] = newstate;
}
```

Implementing Elementary CAs

- wrinkle #1 – edge cases

```
for ( int i = 0 ; i < cells.length ; i = i+1 ) {
  boolean left = cells[i-1];
  boolean middle = cells[i];
  boolean right = cells[i+1];
  …
}
```



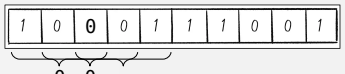
  - what happens when `i` is 0? or `cells.length-1`?

- options for handling edge cases
  - edges remain constant
    - don't update `cells[0]` and `cells[cells.length-1]`
  - edges wrap around
    - `left` for `cells[0]` is `cells[cells.length-1]`
    - `right` for `cells[cells.length-1]` is `cells[0]`
  - define special neighborhoods and rules

Implementing Elementary CAs

- wrinkle #2 – inconsistent update

```
for ( int i = 0 ; i < cells.length ; i = i+1 ) {
  boolean left = cells[i-1];
  boolean middle = cells[i];
  boolean right = cells[i+1];

  boolean newstate;
  …

  cells[i] = newstate;
}
```



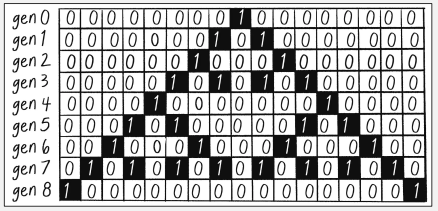
neighborhood is now a mixture of the current generation and the new generation

- fix: use a separate array for the next generation

```
{
  boolean[] newcells = new boolean[cells.length];   // same size as cells

  for ( int i = 0 ; i < cells.length ; i = i+1 ) {
    boolean left = cells[i-1];
    boolean middle = cells[i];
    boolean right = cells[i+1];
    …

    newcells[i] = newstate;  // save the new state in the next generation
  }

  cells = newcells;         // replace the current generation with the new
}
```


Drawing Elementary CAs

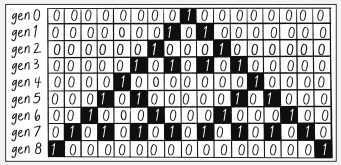
- each generation is drawn on a separate row



- is this making choices, repetition, or just a series of steps?
  - repetition

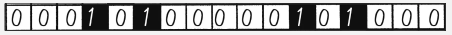
Drawing Elementary CAs

- what is repeated?
  - drawing a row
- what differs from one row to the next?
  - the current generation
  - y coordinate
- how do we start?
  - initialize gen 0 with random values
  - y = 0 (top of row)
- how do things change?
  - compute the next generation
  - y = y+height of row
- when do we keep going?
  - as long as the current row fits in the window (repeat-as-long-as pattern)



Drawing Elementary CAs

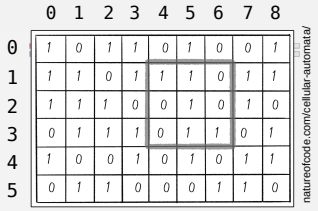
- drawing one row



- what is repeated?
  - drawing one cell
- what differs from one cell to the next?
  - the slot in the array
  - the x coordinate
- how do we start?
  - slot 0
  - x = 0 (left side of rect)
- how do things change?
  - increment slot
  - x = x + width of cell
- when do we keep going?
  - for each slot of the array (going-through-an-array loop)

Implementing 2D CAs

- a grid corresponds to a 2D array
  - rows and columns



- declaring a 2D array variable

```
boolean[][] cells;
```

- creating a 2D array

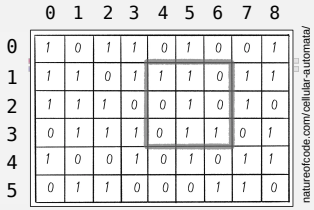
```
cells = new boolean[rows][columns];
```

- going through every slot of a 2D array
  - nested loops

```
for ( int i = 0 ; i < cells.length ; i = i+1 ) {    // rows
  for ( int j = 0 ; j < cells[i].length ; j=j+1 ) { // columns
    cells[i][j] = …;
  }
}
```

Implementing 2D CAs

- drawing a 2D CA
  - for each slot of the array, draw a black- or white-filled rect

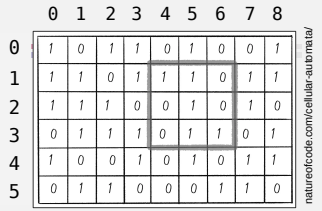


- for each slot of the array –

```
for ( int i = 0 ; i < cells.length ; i = i+1 ) {
  for ( int j = 0 ; j < cells[i].length ; j=j+1 ) {
    …
  }
}
```

- what changes from one slot to the next?
  - rows (i) – y coordinate
  - columns (j) – x coordinate

Implementing 2D CAs



- accessing neighbors

i-1,j-1	i-1,j	i-1,j+1
i,j-1	i,j	i,j+1
i+1,j-1	i+1,j	i+1,j+1

Implementing the Game of Life

- counting the number of living neighbors

i-1,j-1	i-1,j	i-1,j+1
i,j-1	i,j	i,j+1
i+1,j-1	i+1,j	i+1,j+1

```
int numliving = 0;

if ( cells[i-1][j-1] ) { numliving = numliving+1; }
if ( cells[i-1][j] ) { numliving = numliving+1; }
if ( cells[i-1][j+1] ) { numliving = numliving+1; }

if ( cells[i][j-1] ) { numliving = numliving+1; }
if ( cells[i][j+1] ) { numliving = numliving+1; }

if ( cells[i+1][j-1] ) { numliving = numliving+1; }
if ( cells[i+1][j] ) { numliving = numliving+1; }
if ( cells[i+1][j+1] ) { numliving = numliving+1; }
```