Algorithms

(for Game Design)

Session 3: Maze Algorithms

Last time

- Fonts
- Sprites
- Randomness

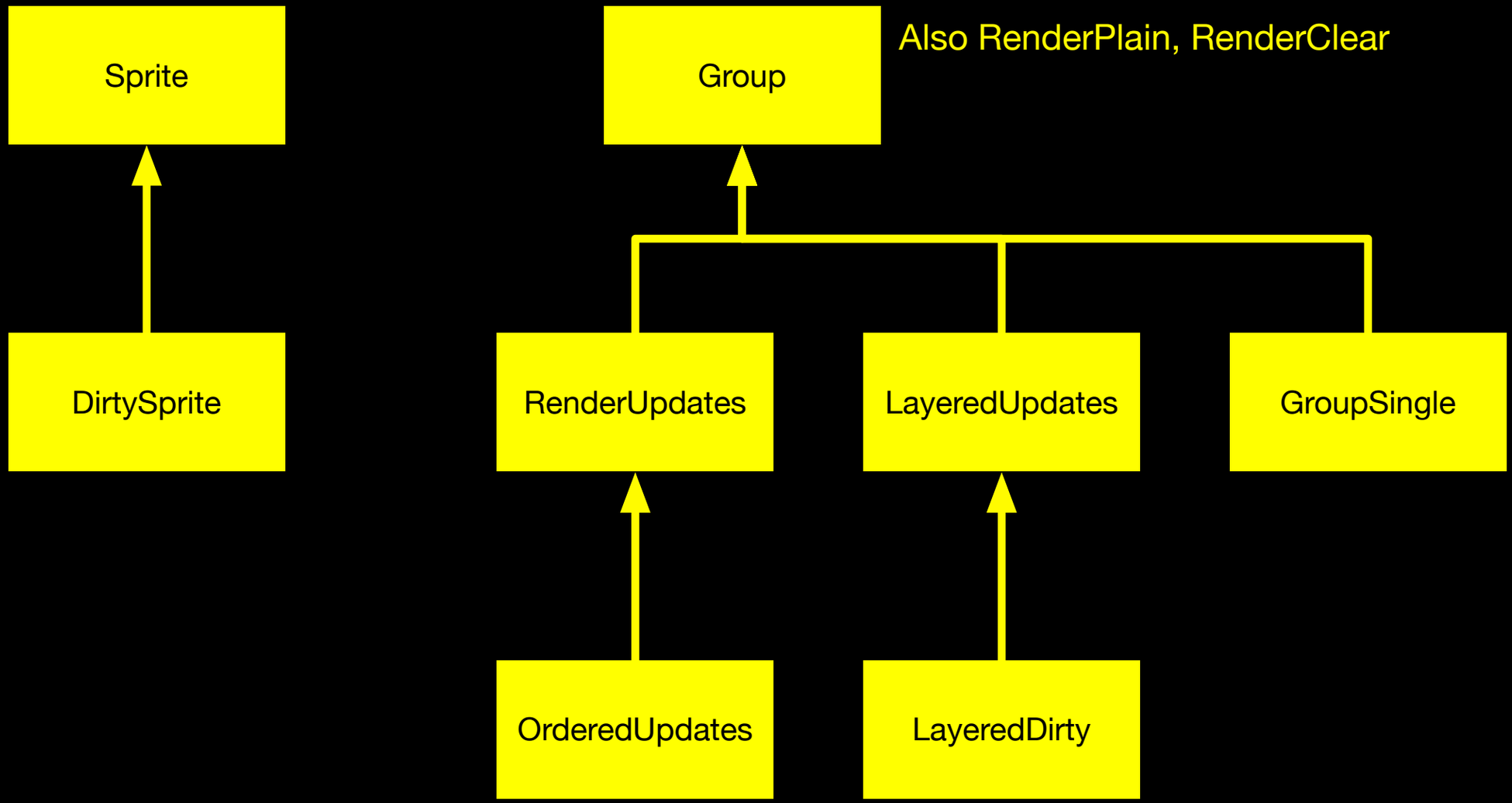
The Font Object

- Great! You've got a font object. What do you do with it?

```
render(text, antialias, color, background=None)
```

- Returns a new `Surface`, with your text drawn on it
- Can only be a single line of text
- If antialias is `True`, the text will be smoothed at the edges
- You can specify `color` of text and the `background`
- Transparent background unless one is specified

pygame.sprite Class Hierarchy



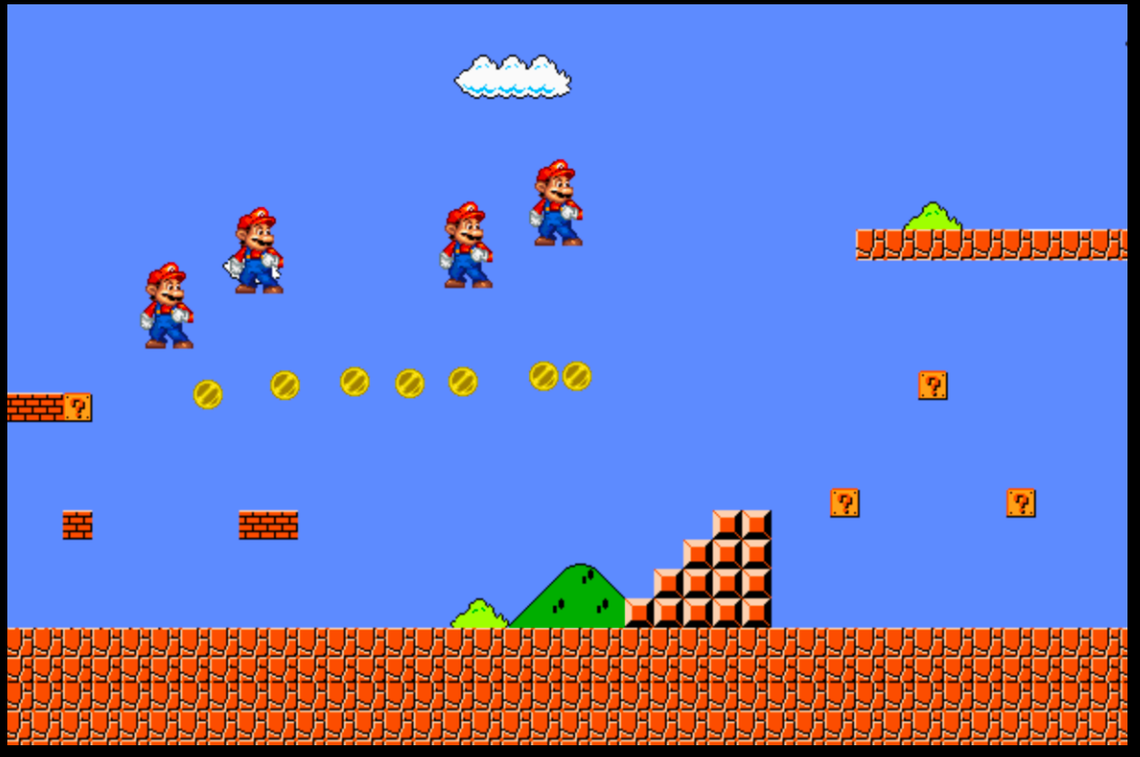
Sprites First

- `pygame.sprite.Sprite` should always be subclassed
  - Your subclass should have `image` and `rect` attributes
- Makes sense: A sprite is an on-screen object with an image and a position
- Override the `update()` method for your chosen way of updating

Multiple Groups

- `sprite4.py` shows a group for `mario` sprites and a separate group for `coin` sprites

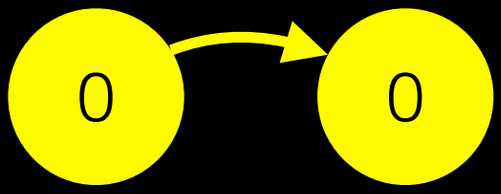
- left mouse click places a coin
- right places a mario (who then starts to drop)

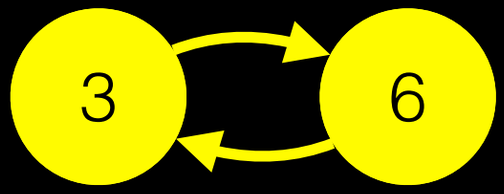


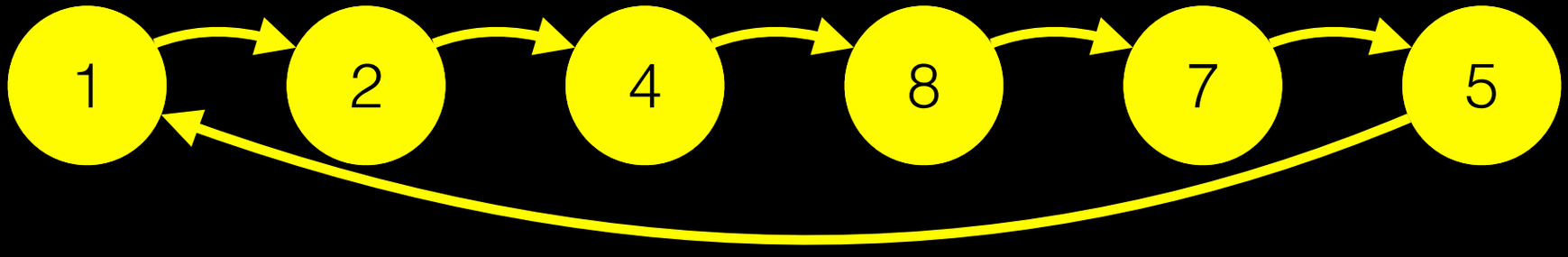
$X_{n+1} = (a * X_n + c) \bmod m$

- For a = 2, c = 0, m = 9 there is a maximum cycle of 9 (you only have 9 possible outputs, 0-8)

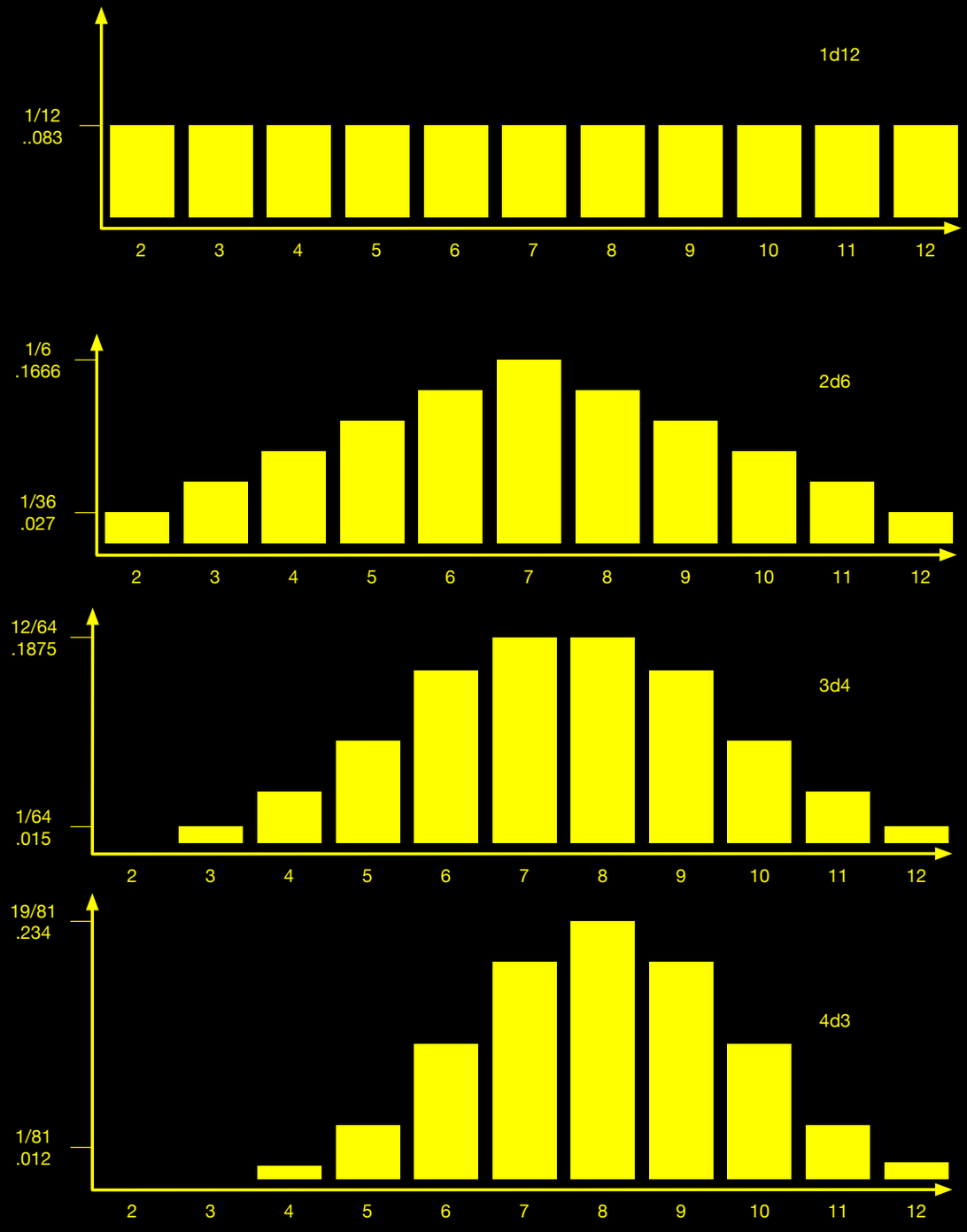
- But, if you check every possible input, you'll realize your cycles are smaller







- Several distributions in the range of 12
- Notice, distribution narrows as more, lower valued, dice are used
- Also, distribution shifts from left to right

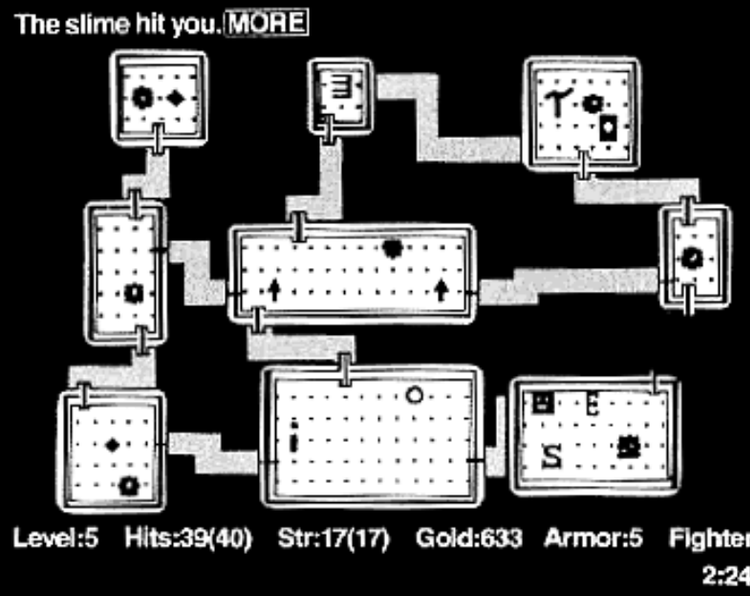


Today

- Mazes
- Maze Creation Algorithms
- Maze Solution Algorithms

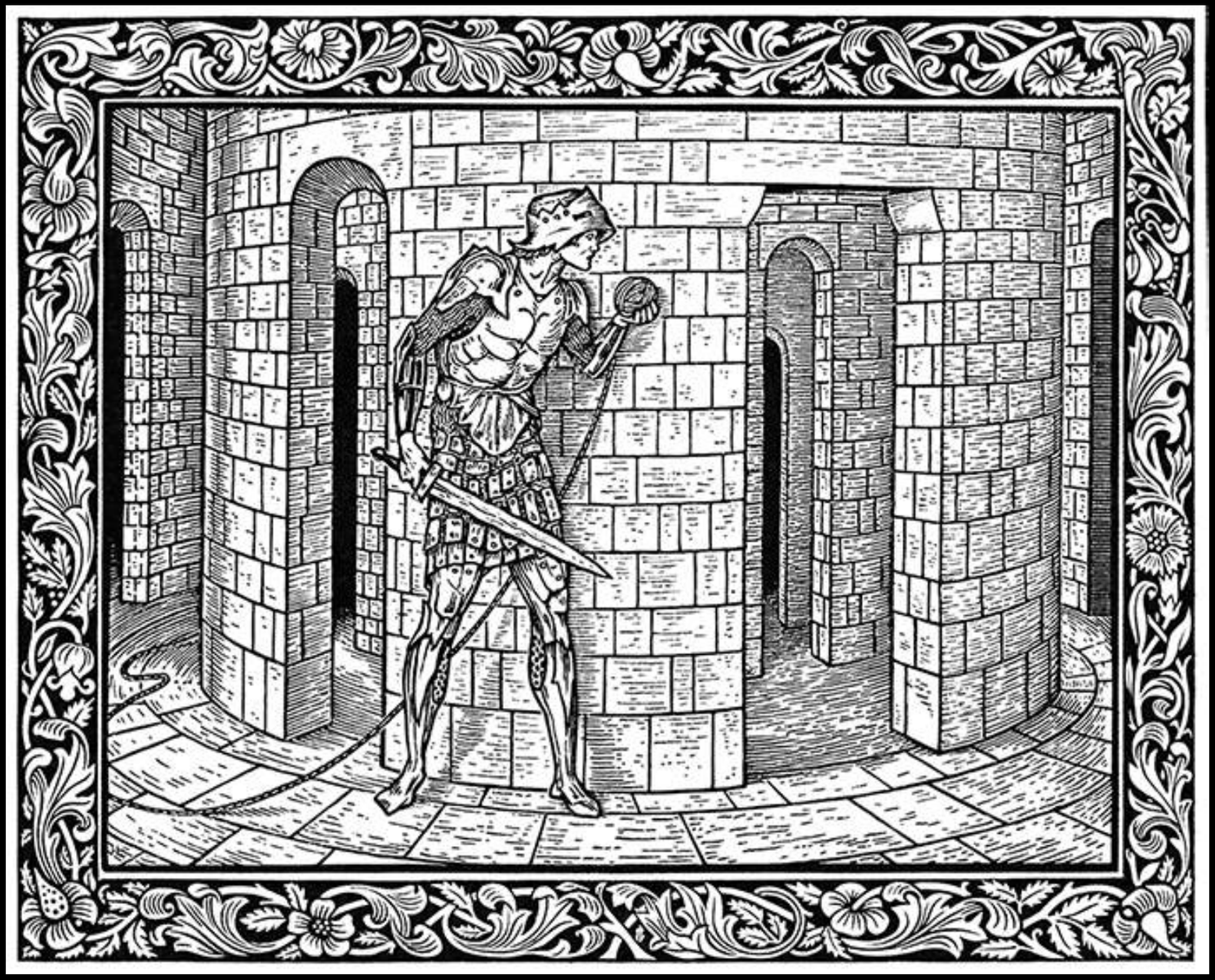
mazes

Mazes



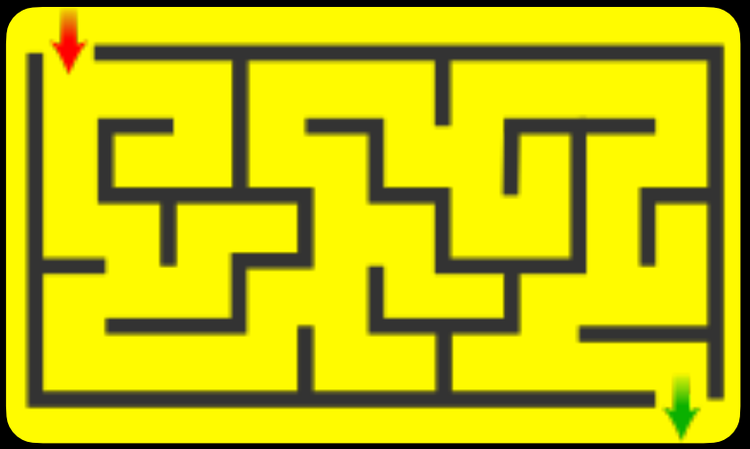
- Mazes are very important for many games
  - Famous games: Pac-Man, Zelda, Doom, Rogue, Robot Rescue, The Last Guy, etc.
- Many games advertise a "procedurally generated" world
  - A random process decides where you are allowed to go, where the monsters are, loot, ...

Mazes in Myth: Labyrinth



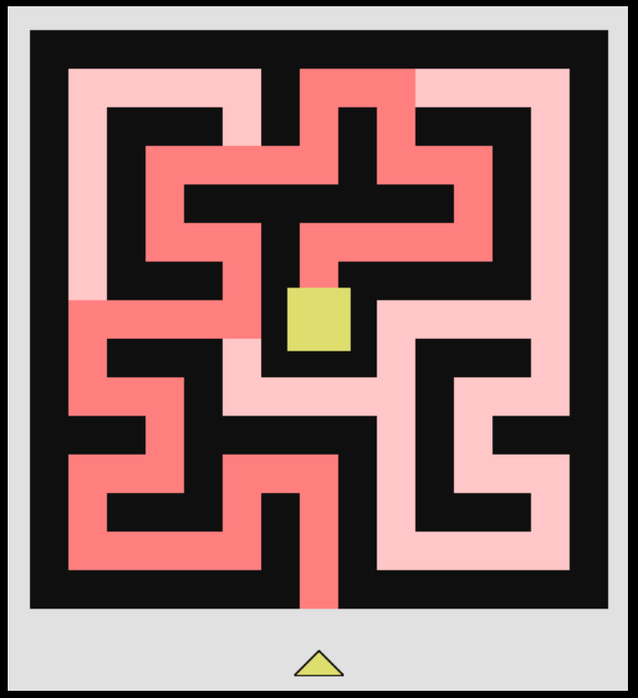
Theseus in the Minotaur's Labyrinth

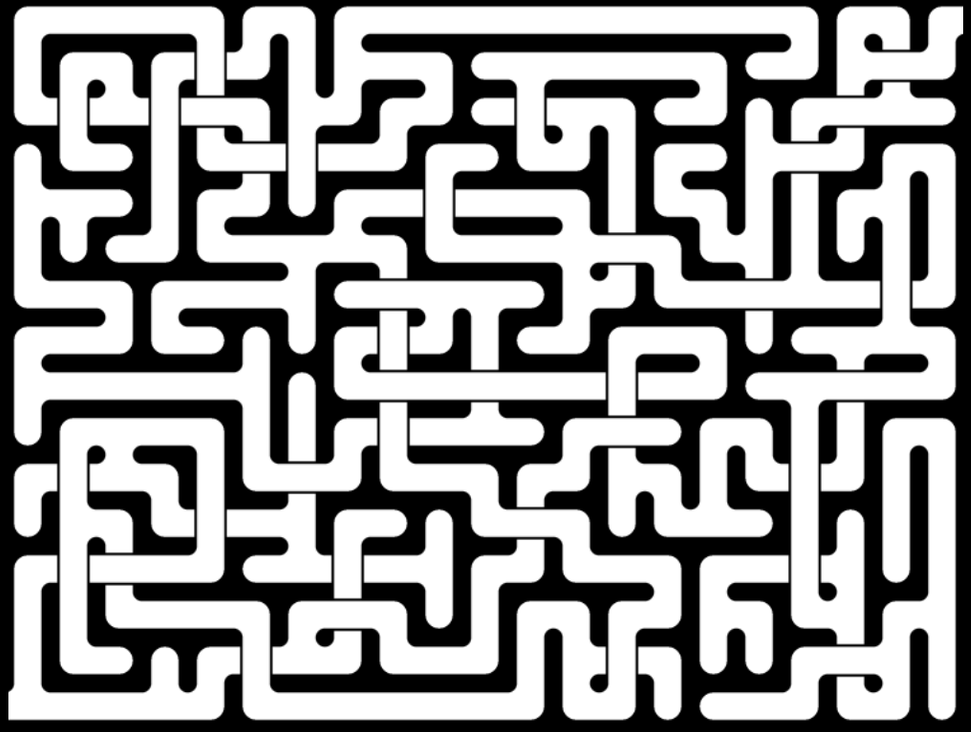
Classifications



- **Perfect** maze: Has one and only one path between any two cells -- thus plenty of branching, but no loops
  - **Cell**: position in the maze, the smallest geographical element of the maze
- **Unicursal** maze: Has no branches, so there is just a single path from beginning to end

- **Braided** maze: has loops, no dead-ends
- **Weave** maze: passages go over and under each other -- not quite 3d





Maze Features

- Dimensionality: We will focus on 2d, but it is easy to do 2.5d and 3d
- Shapes: Mazes can fill non-rectangular spaces, be toroidal
- Tiles: the cells can be non-square to make circular, hexagonal, triangular mazes

Metrics

- **Passage length:** Are there long distances between branching points?
- **Dead ends:** What percentage of cells have only one exit point?
- **Crossroads:** Cells with 3+ exit points
- **Bias:** How uniformly random is the maze?
- **Difficulty:** How hard is it to solve?
- **Others?**

Terminology

- The maze is a grid/collection of **cells**
  - Each is one space along a maze path
- Cells have **neighbors**: other cells they are adjacent to
  - For us, North/South/East/West, not diagonals

- Some cells are **linked**, in which case they form part of a path
    - When solving the maze, the player goes from cell to cell only if they are linked
- Some cells are **unlinked**, in which case there is a wall between the cells
    - Direct travel is prohibited between cells that are unlinked
- All neighboring cells will be either linked or unlinked

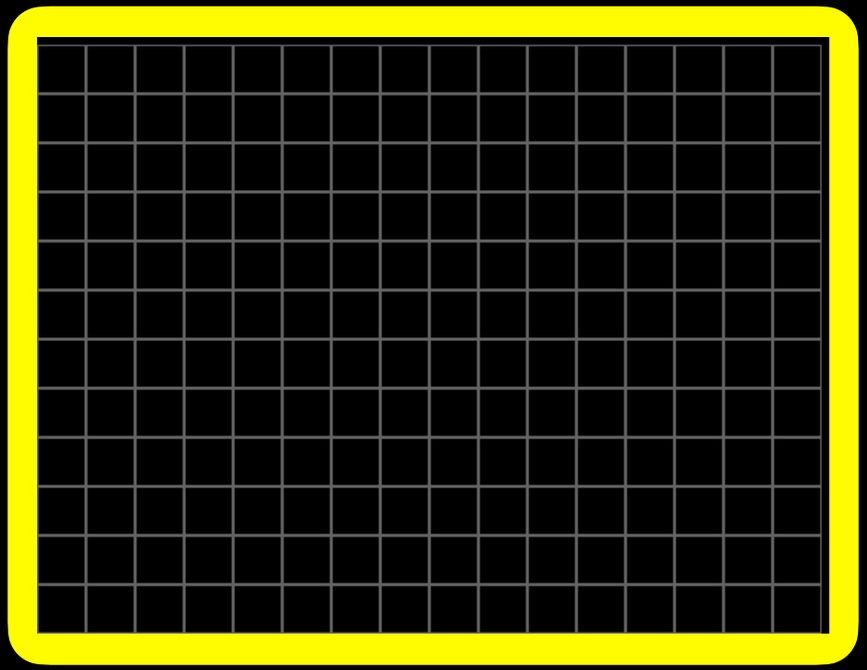
Algorithms

- For **random** mazes, an algorithm generates the maze
- Many algorithms start with a grid of unlinked cells and then choose pairs to be linked
  - This is known as **carving** a path between the cells
- Other algorithms are **wall adders**. They start with empty space and add walls to it

binary tree

Binary Tree Algorithm

- Our first maze generation algorithm
- A wall carving algorithm
  - So, we start with a grid of unlinked cells



- Each cell will, when possible, be linked to either the North neighbor or the East neighbor
  - Cells on the East column are all linked to their North neighbor
  - Cells on the North row are all linked to their East neighbor, except...
  - North East cell is not linked North or East

The Algorithm

simple, simple

- For each cell:
  - Make a list of all North and East neighbors
  - If the list is not empty:
    - Pick a random neighbor
    - Carve a passage (i.e. link to that neighbor)

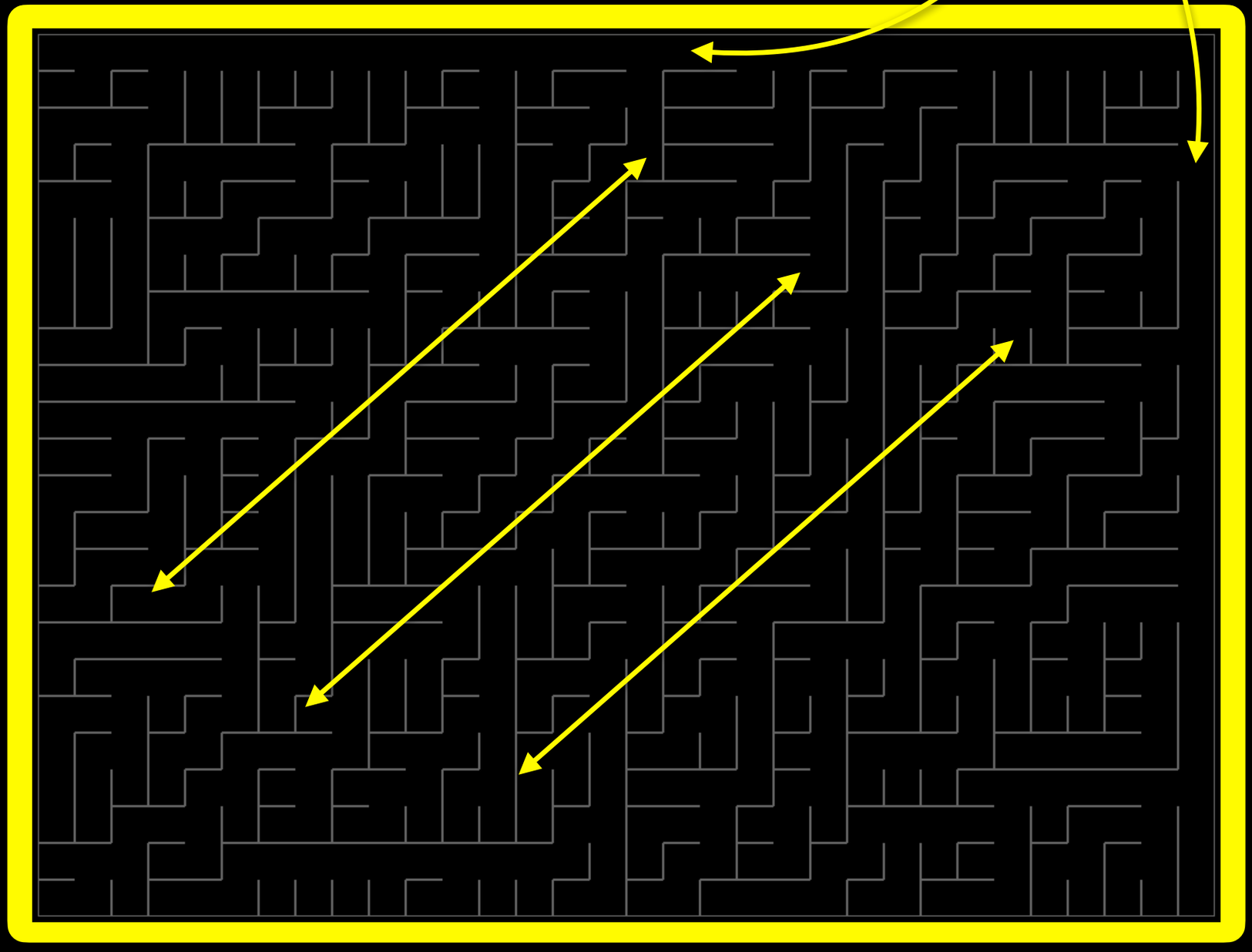
Why?

Easy to parallelize!

Result: Binary Tree

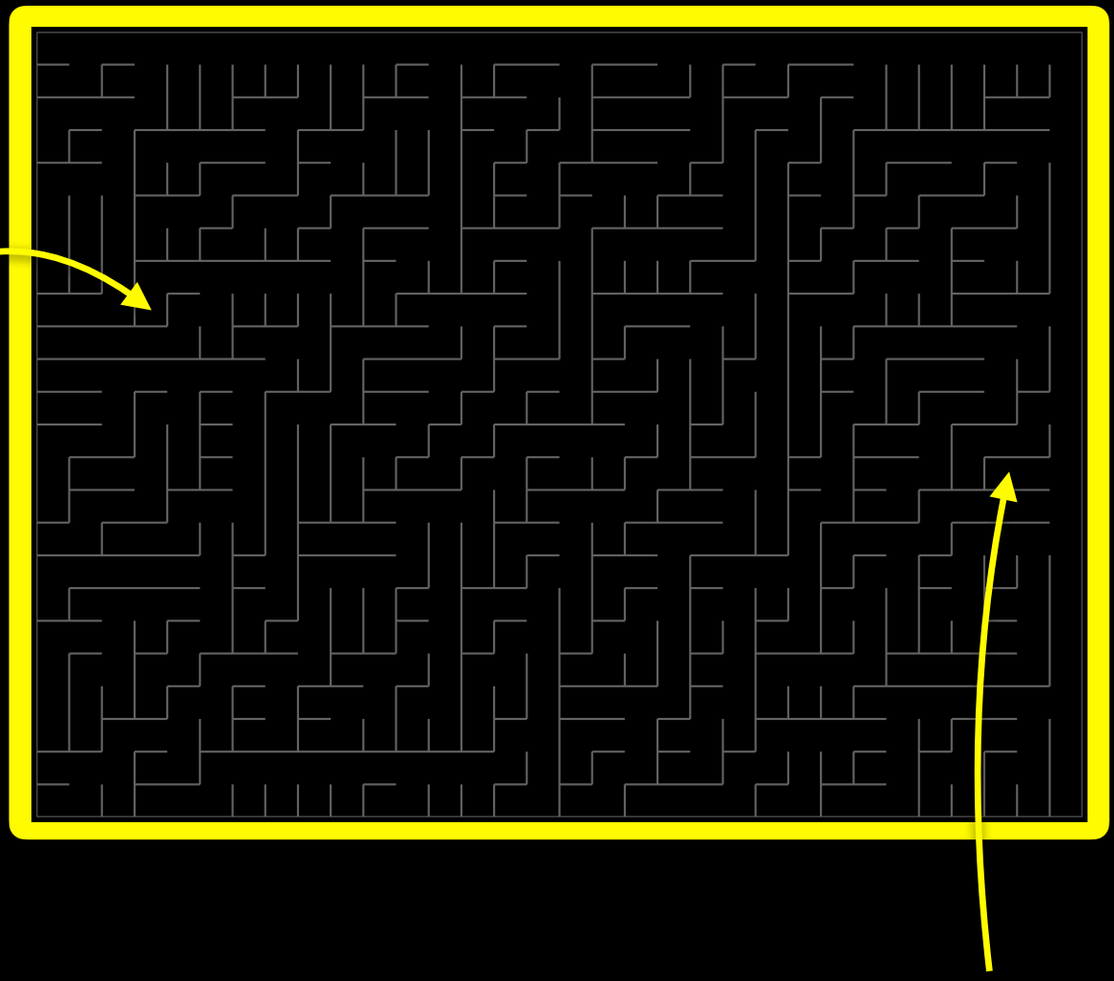
Top-row and Right-column are always clear

Bias



Where is the Exit?

- Our maze has no entry or exit
- But, its a perfect maze
  - Put the entry / exit wherever you want



We will learn techniques to help decide

Data structures

- `Cell:` keep track of neighbors, links
- `Grid:` container for all cells
- `Markup:` keep track of some form of annotations / data for each cell (color, a number, text, etc)
- in `mazes.py`

```
# Some code from mazes.py: The Cell class
class Cell:
    ''' Represents a single cell of a maze...'''
    def __init__(self, row, column):
        assert row >= 0
        assert column >= 0
        self.row = row
        self.column = column
        self.links = {}
        self.north = None
        self.south = None
        self.east  = None
        self.west  = None

    def link(self, cell, bidirectional=True):
        ''' Carve a connection to another cell (i.e. the maze connects them)'''
       self.links[cell] = True
       if bidirectional:
           cell.link(self, bidirectional=False)

    def unlink(self, cell, bidirectional=True):
        ''' Remove a connection to another cell.
            Argument bidirectional is here so that I can call unlink on either
            of the two cells and both will be unlinked.
        '''
        del self.links[cell]
        if bidirectional:
            cell.unlink(self, bidirectional=False)
```

Do you know and use assert?

There isn't much to cell

-- Keep track of neighbors

-- And which are linked

```
# Some code from mazes.py: The Grid class
class Grid:

    def __init__(self, num_rows, num_columns):
        self.num_rows = num_rows
        self.num_columns = num_columns
        self.grid = self.create_cells()
        self.connect_cells()

    def create_cells(self):
        ''' Call the cells into being.  Keep track of them in a list
            for each row and a list of all rows (i.e. a 2d list-of-lists).'''
        pass

    def connect_cells(self):
        ''' Now that all the cells have been created, connect them to
            each other - set the north/south/east/west attributes.'''
        pass

    def cell_at(self, row, column):
        ''' Retrieve the cell at a particular row/column index.'''
        pass

    def deadends(self):
        ''' Return a list of all cells that are deadends.'''
        pass
```

Grid: A container for cells

Cells are kept in a 2-d array (list of lists)

With some management features

```
# Some code from mazes.py: The Grid class
class Grid:
    # continued from previous page

    def each_cell(self):
        ''' A generator.  Each time it is called, it will return one of
            the cells in the grid.
        '''
        for row in range(self.num_rows):
            for col in range(self.num_columns):
                c = self.cell_at(row, col)
                yield c

    def each_row(self):
        ''' A row is a list of cells.'''
        for row in self.grid:
            yield row

    def random_cell(self):
        ''' Chose one of the cells in an independent, uniform distribution. '''
        pass

    def size(self):
        ''' How many cells are in the grid? '''
        pass
```

A generator lets you write code like:

```
for cell in a_grid.each_cell():
    ...
```

What does uniform mean?

```
# Some code from mazes.py: The Markup class
class Markup:
    ''' A Markup is a way to add data to a grid.  It is associated with
        a particular grid.'''

    def __init__(self, grid, default=' '):
        self.grid = grid
        self.marks = {}  # Key: cell, Value = s
        self.default = default

    def reset(self):
        self.marks = {}

    def __setitem__(self, cell, value):
        self.marks[cell] = value

    def __getitem__(self, cell):
        return self.marks.get(cell, self.default)

    def set_item_at(self, row, column, value):
        assert row >= 0 and row < self.grid.num_rows
        assert column >= 0 and column < self.grid.num_columns
        cell = self.grid.cell_at(row, column)
        if cell:
            self.marks[cell]=value
        else:
            raise IndexError
```

A Markup is basically a dictionary, where key=cell and value=the data

Python magic! Now, we can do `a_markup[a_cell]`

Set the value for a cell without having the cell (just the row/column indexes)

```
# Some code from mazes.py: The Markup class
class Markup:
    # continued from previous page

    def get_item_at(self, row, column):
        ...

    def max(self):
        ''' Return the cell with the largest markup value. '''
        return max(self.marks.keys(), key=self.__getitem__)

    def min(self):
        ''' Return the cell with the largest markup value. '''
        return min(self.marks.keys(), key=self.__getitem__)
```

Binary Tree: The Code

- So, what does the code for the Binary Tree algorithm look like?

```
def binary_tree(grid):
    ''' The Binary Tree Algorithm.

        This algorithm works by visiting each cell and randomly choosing
        to link it to the cell to the east or the cell to the north.
        If there is no cell to the east, then always link to the north
        If there is no cell to the north, then always link to the east.
        Except if there are no cells to the north or east (in which case
        don't link it to anything.)
    '''
    pass
```

Complete this for the Exercise

Binary Tree: Coding Hints

```
def binary_tree(grid):
```

- Iterate through all cells

```
for a_cell in grid.each_cell():
```

- If there are no North or East neighbors do nothing
- If there is not an East neighbor, link to the North neighbor

```
if not a_cell.east:
```

```
a_cell.link(a_cell.north)
```

- Else, randomly choose between north and east neighbors

```
r = random.choice([a_cell.north, a_cell.east])
```

- And link to it

```
a_cell.link(r)
```

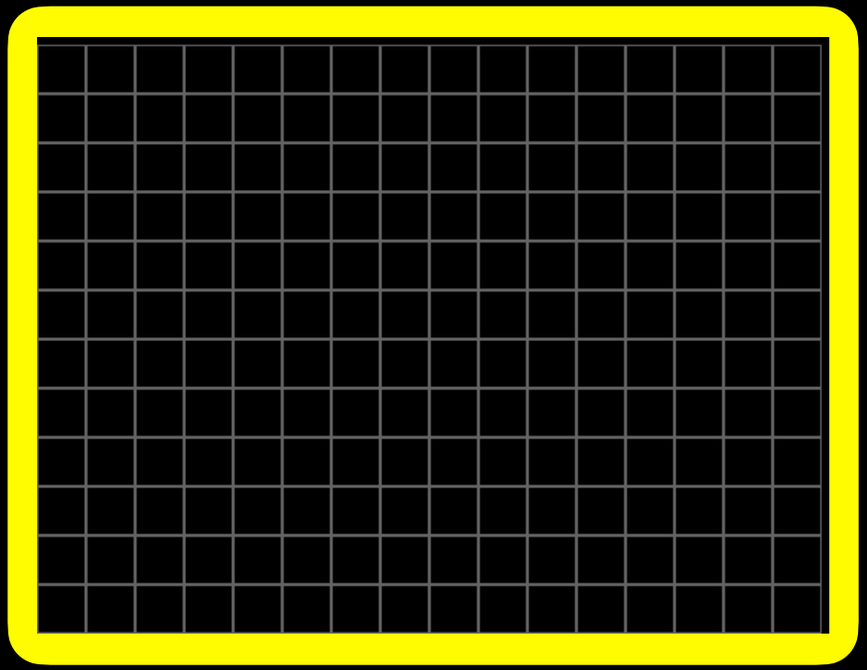
Judgement

- A very easy algorithm
- Each cell can be handled in parallel
- Distinct look
  - Top row and Right column are connected
- Bias for diagonal flow

sidewinder

Sidewinder Algorithm

- Another maze generation algorithm
- Also a wall carving algorithm
  - So, we start with a grid of unlinked cells



Row at a time

- Top row: link all cells together
- Other rows, collect adjacent cells into "runs"
    - Add leftmost cell to a run_list
    - Randomly decide to add the next cell to the run_list
        - Or end the run
            - Always end the run when there is no cell to the East
- When ending the run, link all cells in the run together
    - Randomly choose one cell to link to North neighbor

The top row: all cells are linked

0	1	2	3	4	5	6	7	8	9	10	11

A run	A run	4	5	6	7				

run_list = [4 | 5 | 6 | 7]

Add Cell 8?
Or stop?

If stop, randomly choose 4,5,6 or 7 to connect North

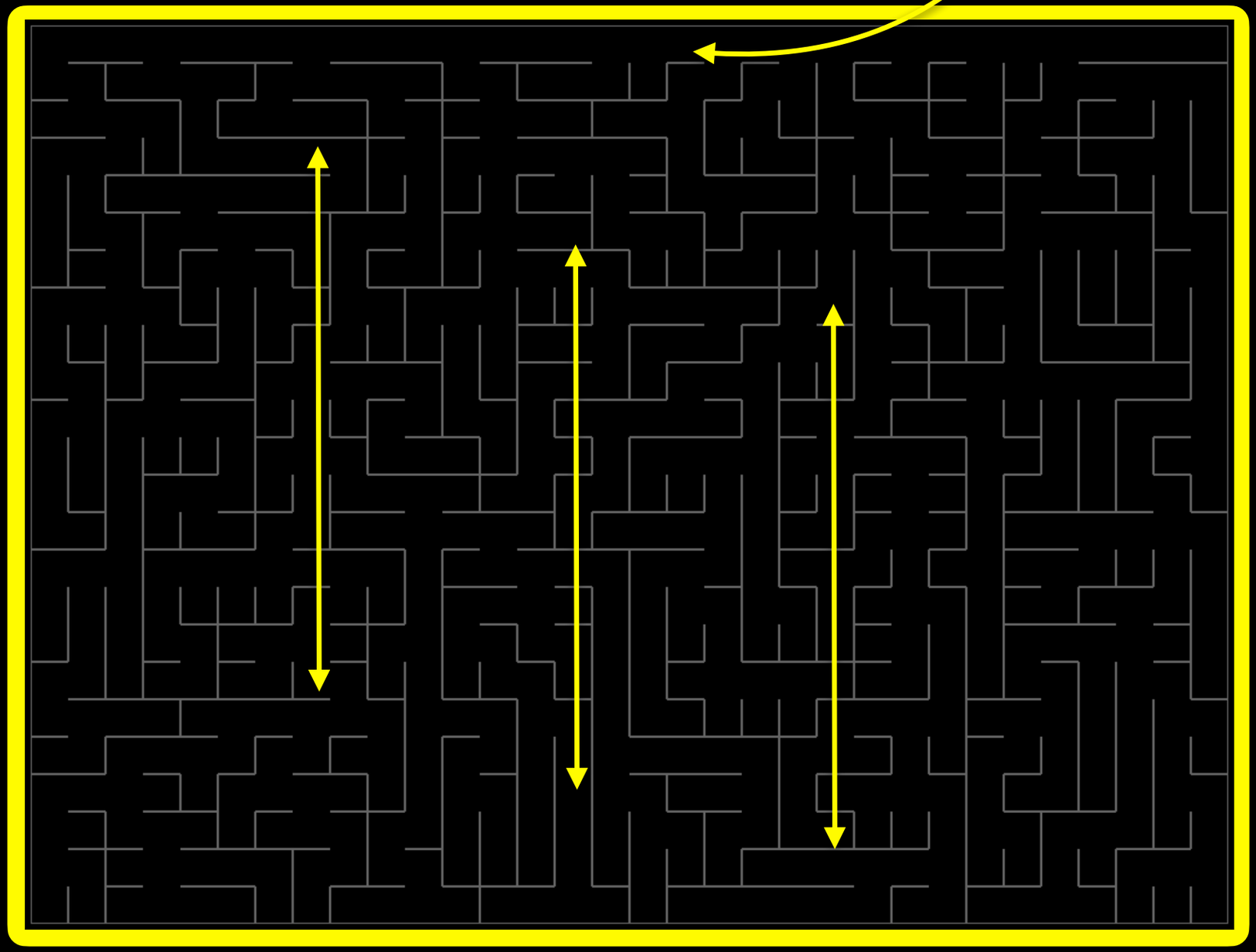
A run	A run	4	5	6	7				

A run	A run				

Result: Sidewinder

Top-row is always clear

Bias



Judgement

- An easy, efficient, simple algorithm
- Each row can be handled in parallel
- Modify the length of the horizontal passages by changing the odds of the add/end coin flip
- Has a bias for north-south passages

Aldous-Broder random walk

Aldous-Broder

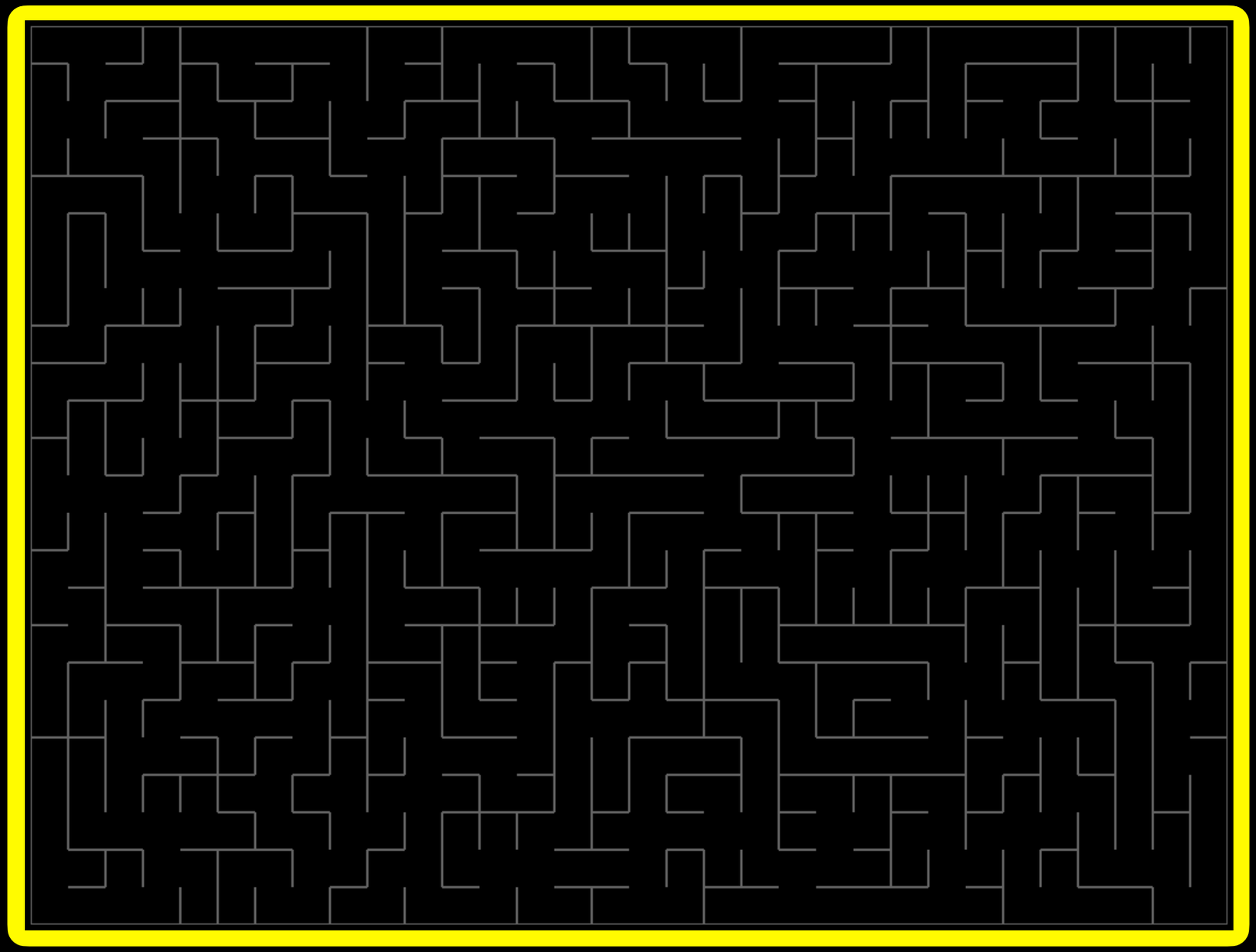
- Discovered simultaneously by two professors with those names
- A **random walk** algorithm: step from cell to random neighbor to create random paths

Random Walk

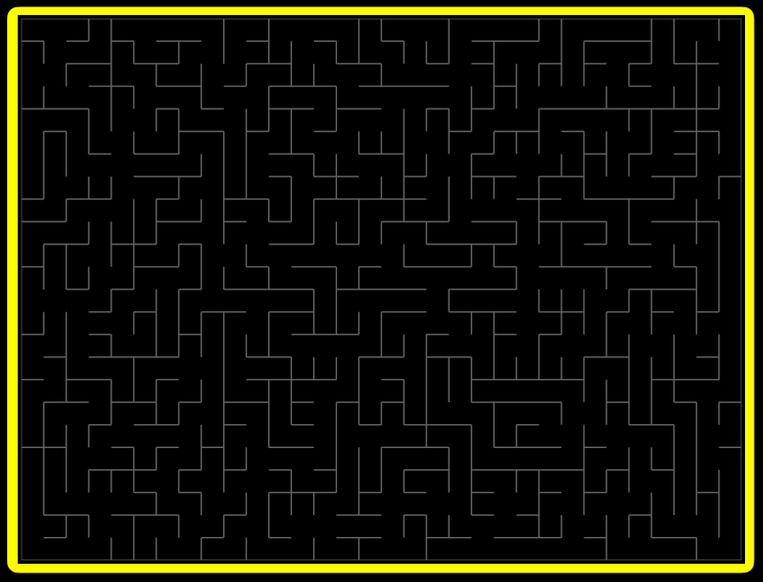
1. Start in a random cell
2. Link to a random neighbor if that neighbor hasn't been visited yet
3. Move to that neighbor
4. If there are unvisited cells anywhere, go back to step 2
5. Finished

Result: Aldous-Broder

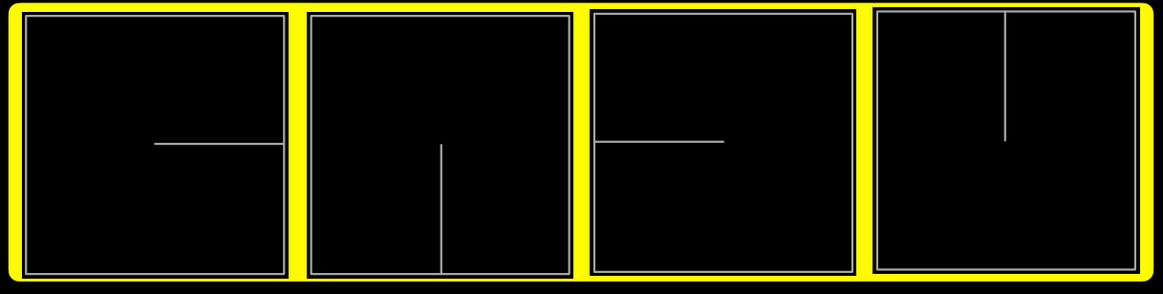
Looks great!



Bias



- A-B has no bias
- What does bias really mean?
- Imagine all possible perfect mazes of a particular size (here, for 2x2)

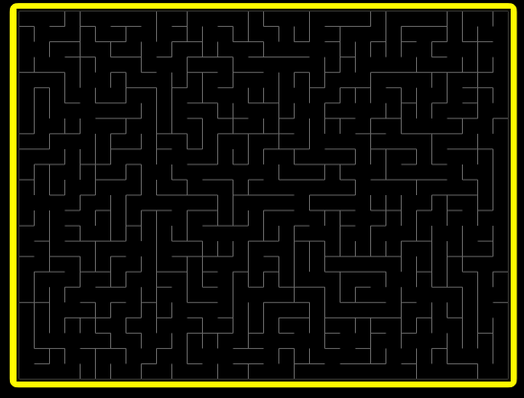


  - An unbiased algorithm will potentially generate any of them -- with a uniform probability distribution over them all

Could BT or Sidewinder have generated all of these?

Judgement

- Aldous-Broder generates mazes with no bias
- They look really good!
- BUT:
  - It can take a long time to randomly walk into the last of the cells
  - This maze has 768 cells, but the A-B algorithm took 15,232 steps to complete it



Wilson's random walk

Wilson's Algorithm

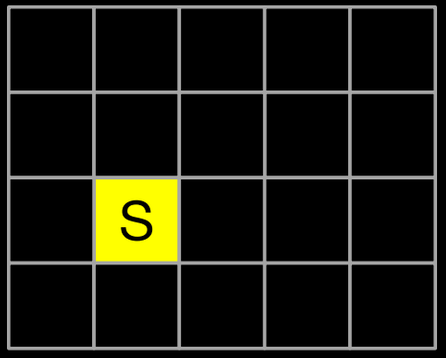
- Also a random walk algorithm
- Also creates mazes with no bias
- Also has inefficient performance characteristics

The Algorithm

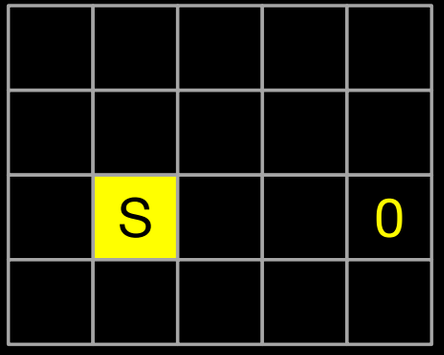
1. Choose a random cell, mark it as "visited"
2. Choose another random unvisited cell as a starting point
3. Perform a **loop-erased random walk**, choosing a random neighbor at each step
4. If the random neighbor has been "visited", the walk is over
    1. Mark all the cells from the walk as "visited"
    2. Link all of the cells to each other, in the order they were chosen, and to the already-visited cell that ended the walk
5. Repeat steps 2-4 until all cells are visited

The Algorithm

1. Choose a random cell, mark it as "visited"



2. Choose a random unvisited cell as a starting point

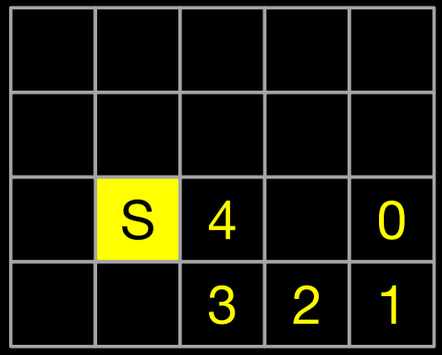


The Algorithm

Ignore for now

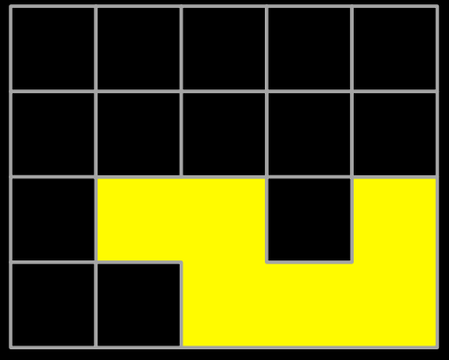
3. Perform a ~~**loop-erased**~~ **random walk**, choosing a random neighbor at each step

This walk randomly chose: "South Neighbor", "West Neighbor", "West Neighbor", "North Neighbor" and "West Neighbor"



The Algorithm

4. If the random neighbor has been "visited", the walk is over
    1. Mark all the cells from the walk as "visited"
    2. Link all of the cells to each other, in the order they were chosen, and to the already-visited cell that ended the walk

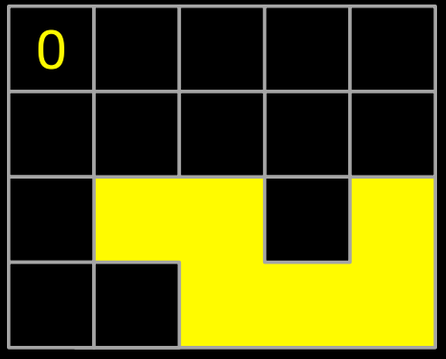


5. Repeat steps 2-4 until all cells are visited

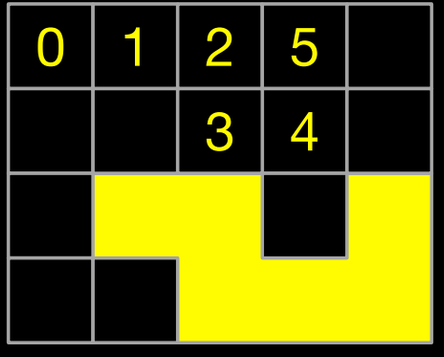
Not all cells are visited, so we have to repeat from Step 2

The Algorithm

2. Choose a random unvisited cell as a starting point



3. Perform a **loop-erased random walk**, choosing a random neighbor at each step

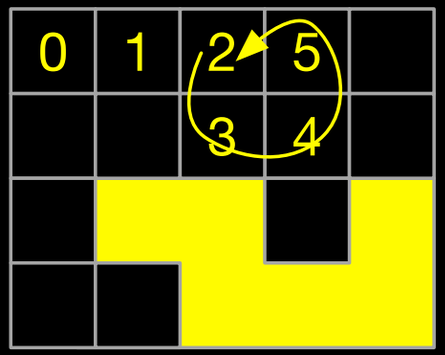


So far, this walk randomly chose: "East Neighbor", "East Neighbor", "South Neighbor", "East Neighbor" and "North Neighbor"

The Algorithm

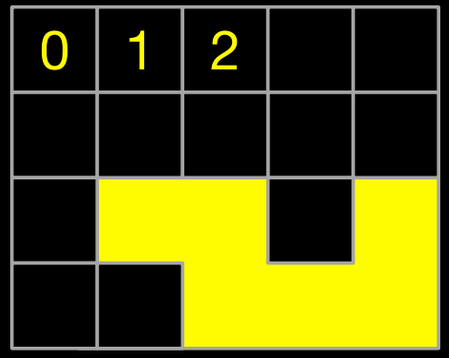
3. Perform a **loop-erased random walk**, choosing a random neighbor at each step

Still walking...



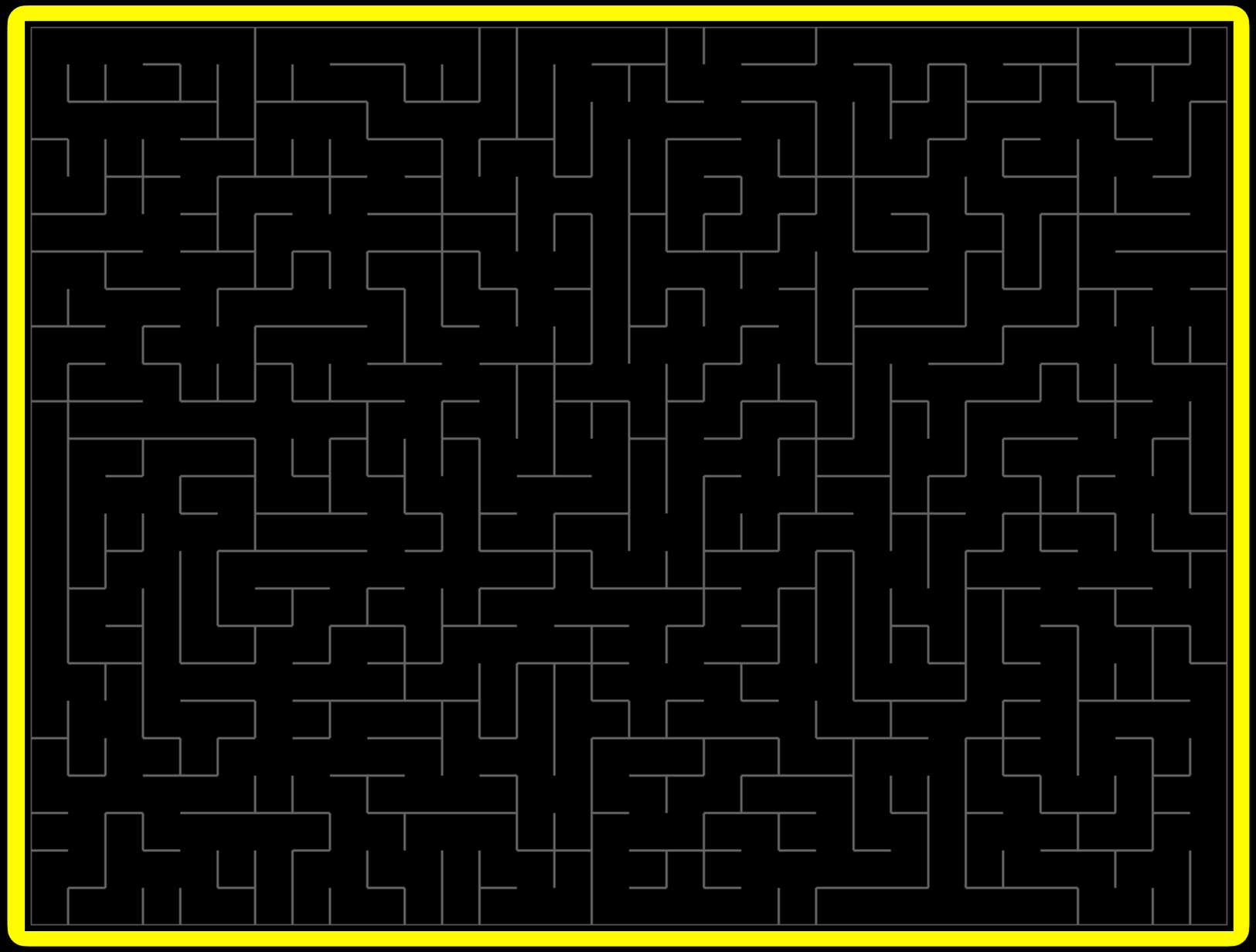
If the next neighbor choice is "West Neighbor", that creates a loop

Erase any loop, retaining the beginning of the loop (i.e. Cell#2)



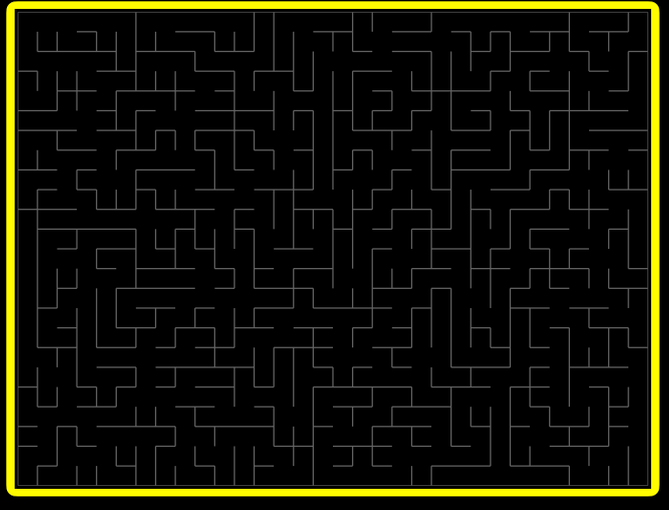
Result: Wilson's

Looks great!



Judgement

- Again, no bias
  - You can't tell from the output that this was a Wilson's vs an Aldous-Broder generated maze
- Again, performance:
  - This maze has 768 cells, but 4739 random cells were chosen
  - 1288 loops were removed



- In the beginning, it is hard to find a random walk back to that starting point without looping
  - But, the end is very quick to walk to a visited cell

recursive backtracker

Recursion

- Defining a thing in terms of itself
- Often confusing for intro CS students
- Don't be scared: we don't have to write this algorithm recursively (but you can!)

The Algorithm

- Choose a random starting cell
- Do a random walk: randomly choose a previously unvisited neighbor
  - Link to and move to that neighboring cell
- Anytime there is no unvisited neighbor, back up along your path until you do have one, which you pick

State

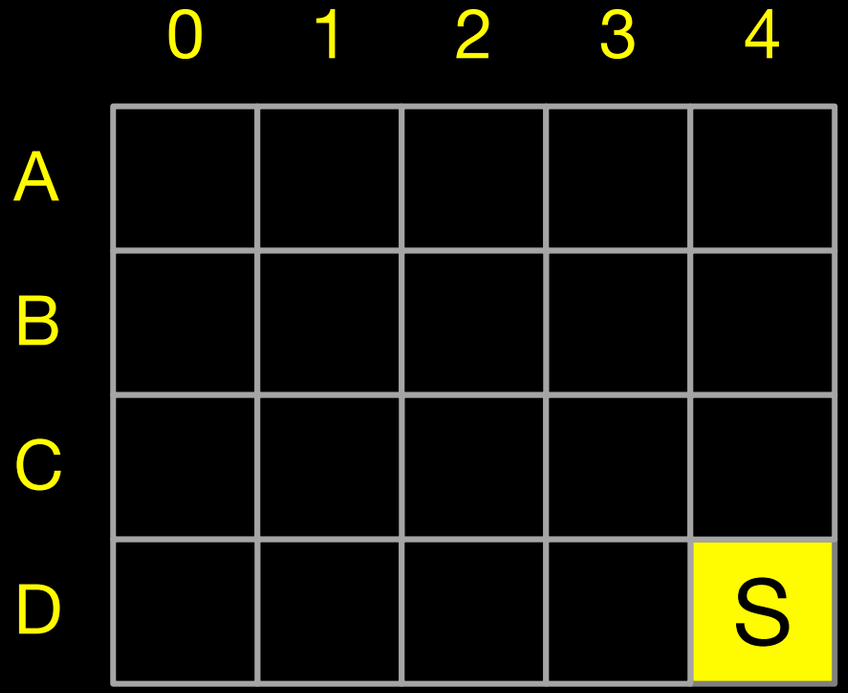
- The recursive backtracker algorithm requires some additional state -- you need to keep track of the cells you have visited, in reverse order
  - A **stack**: a last-in, first-out data structure
- Do we have those in Python?

```
# A Python stack
stack = []

# PUSH: Add to the stack (always to the end)
stack.append("Mazes")
stack.append("are")
stack.append("amazing!")

# POP: Remove from the stack (always from the end)
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

Notice: We add/remove from the **back**
A stack is an ADT, you shouldn't care how it stores internal data

The Algorithm

- Choose a random starting cell

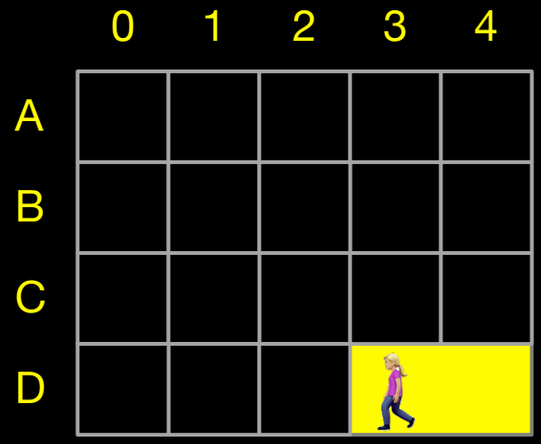


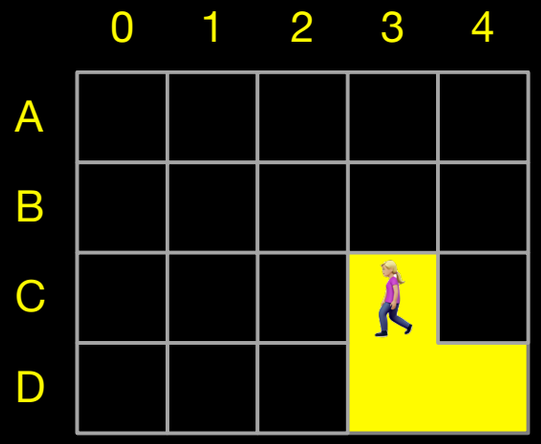
`stack=[D4]`

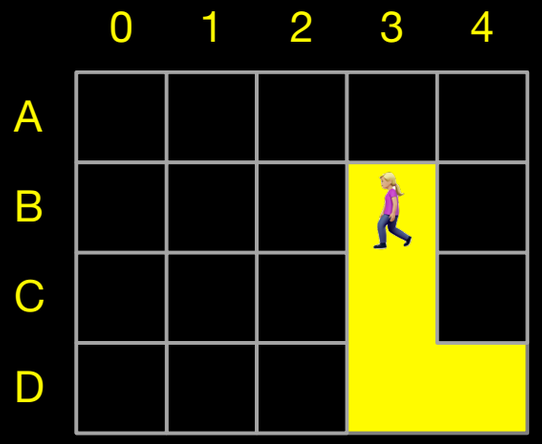
I will visit this cell, so put it on the stack

The Algorithm

- Do a random walk: randomly choose a previously unvisited neighbor
  - Link to and move to that neighboring cell

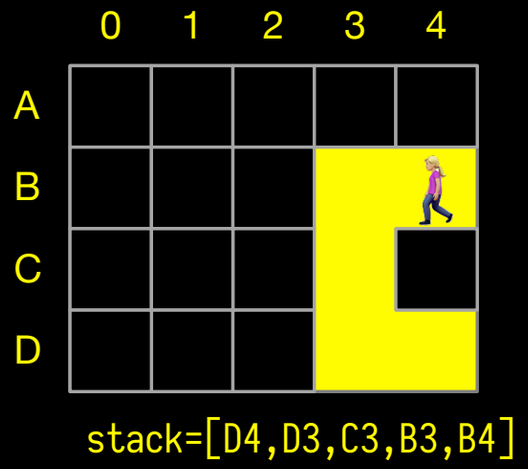


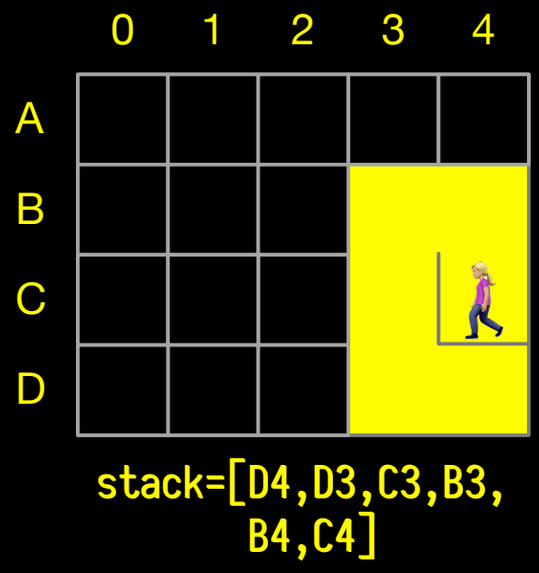




The Algorithm

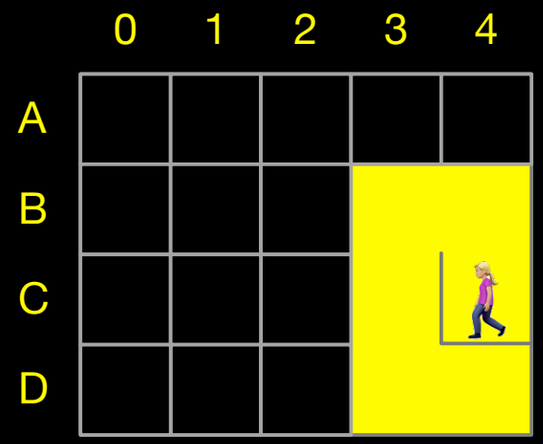
- Do a random walk: randomly choose a previously unvisited neighbor
  - Link to and move to that neighboring cell



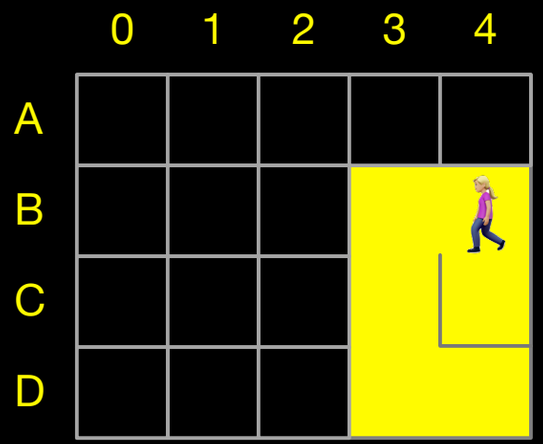


The Algorithm

- Anytime there is no unvisited neighbor, back up along your path until you do have one, which you pick

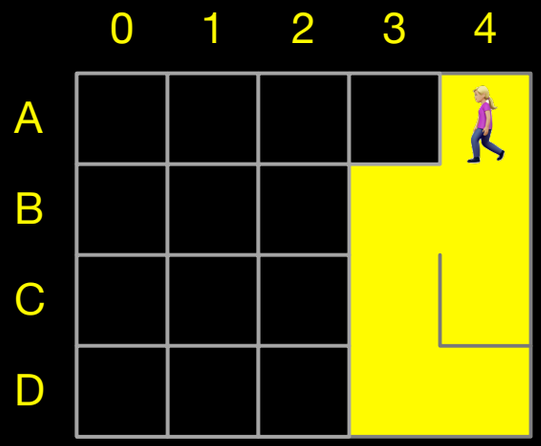


stack=[D4,D3,C3,B3,
B4,C4]

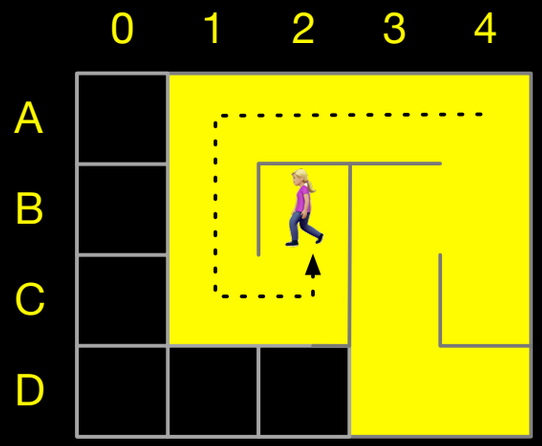


stack=[D4,D3,C3,B3,B4]

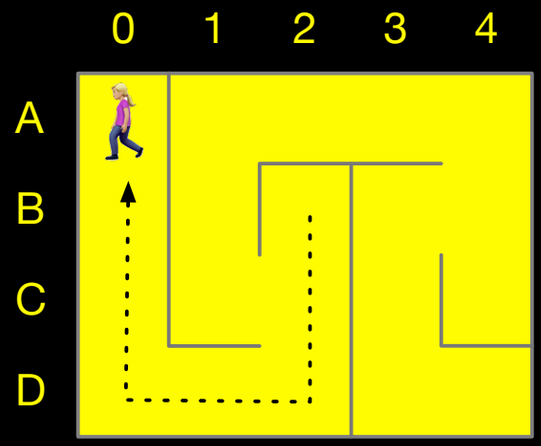
The Rest



stack=[D4,D3,C3,B3,
B4,A4]



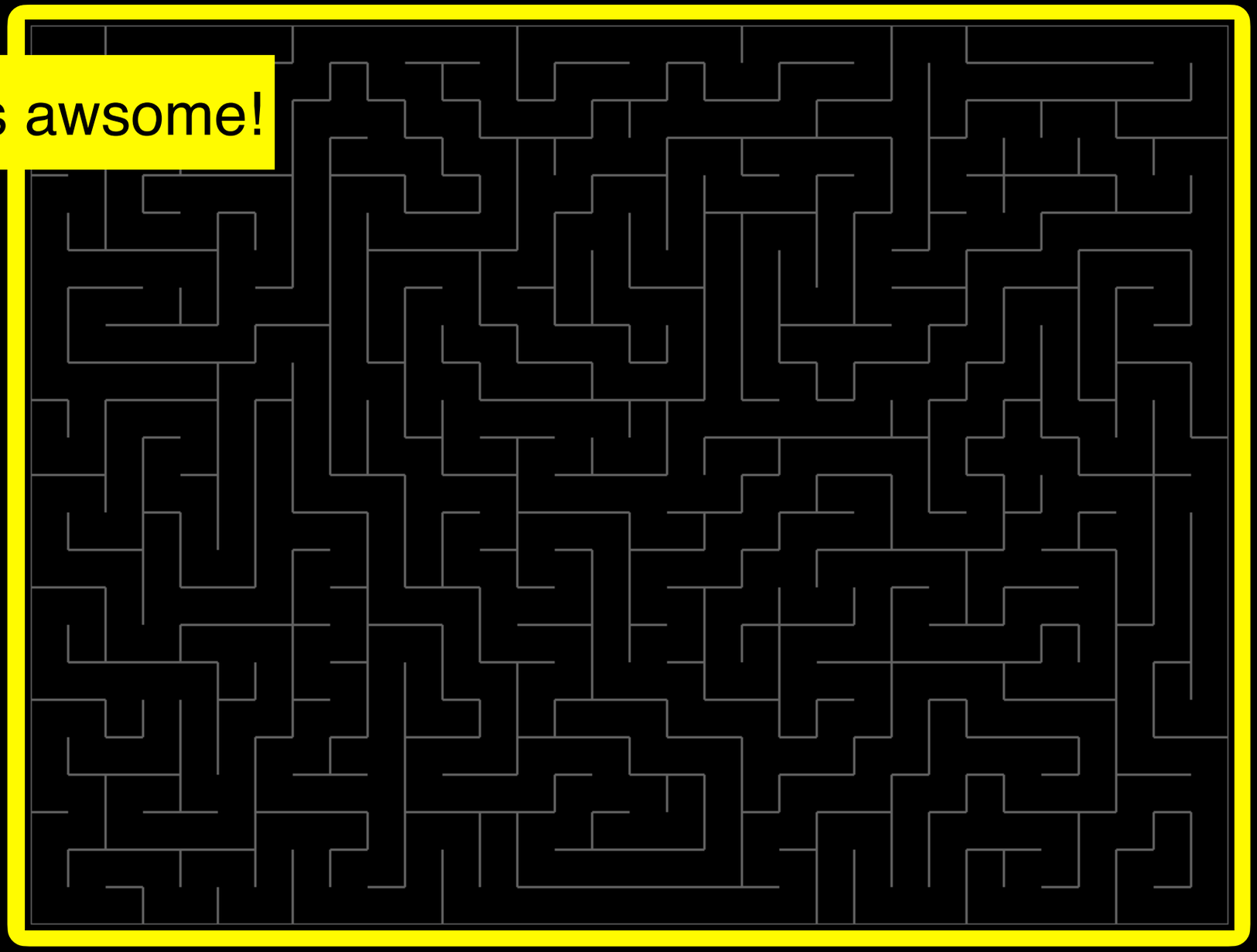
stack=[D4,D3,C3,B3,
B4,A4,A3,A2,
A1,B1,C1,C2,
B2]



stack=[D4,D3,C3,B3,
B4,A4,A3,A2,
A1,B1,C1,C2,
D2,D1,D0,C0
B0,A0]

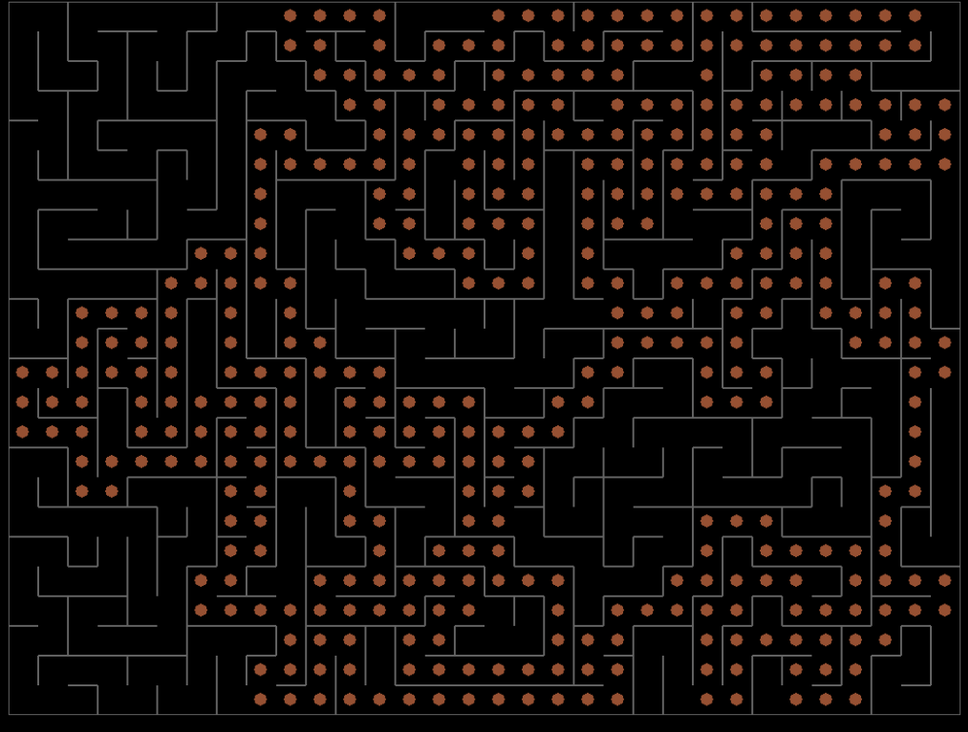
Result: Recursive Backtracker

Looks awsome!

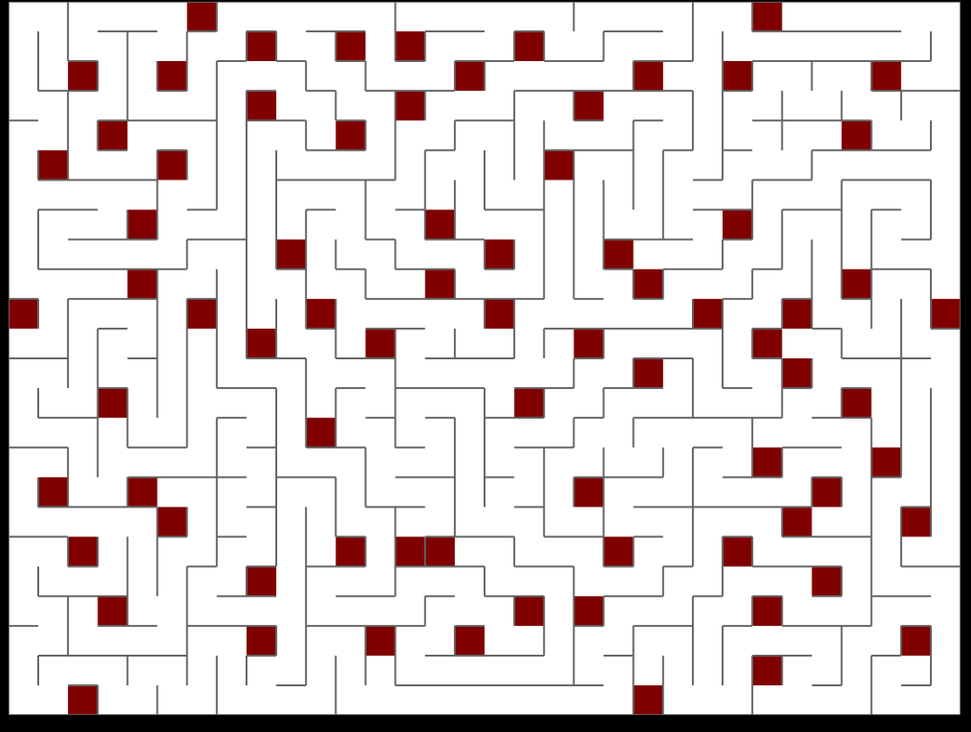


Judgement

- Recursive Backtracker leads to mazes with long, twisty paths with few dead ends (yep, it's biased)



Same maze, longest path is marked



Same maze, all the dead ends

Judgement

- The algorithm is not very memory efficient
  - It has to keep the stack, which may contain every cell in the entire maze
- The stack may be explicit (i.e. a list called "stack")
  - Or implicit: use the function call stack and write the algorithm in a recursive manner
- For large mazes, explicit is probably better

maze solution algorithms

Solution Algorithms

- Now that we've created mazes, can't we solve them? Of course we can
- Our solution algorithms will have access to the Grid, and thus can find out about a cell of interest
  - Primarily if it is linked to its neighbors
- Beware: Solution algorithms can't see the entire maze, like we can
  - Need to be stated very precisely, not just "follow the left-hand wall"

Various Algorithms

- Wall Follower: At each junction turn right (or left)
  - Can solve any maze if the entry and exit are on the edges
- Dead End Filler: works for perfect mazes
- Recursive Backtracker: won't always find the shortest solution
- Trémaux Algorithm: can be used by a human inside the maze, marking passages 1 or 2 times

Dijkstra's Algorithm

- A very famous graph algorithm, named after a very famous Dutch computer scientist
- Used in network routing algorithms, map navigation, robot movement planning, and many others
- We will start off with a simplified version that works well for our mazes (a "breadth-first search")
    - In sessions to come, we will revisit it several times

Goal: Least-cost path

- It finds the "least-cost" path between some starting cell and every other cell in the maze
  - For our maze, the cost to travel from one cell to another is the same:
    - 1 if linked
    - ∞ if not linked
- In general, it works even when the cost is different
  - Useful, for instance, to determine if a car should go a long distance on the highway or a shorter distance on surface (and slower) streets

The Algorithm

- We must have a way to mark up the grid -- to place a value in each cell (thus, the Markup class)

1. Mark the root cell (the starting point) with zero
    - It costs 0 to get from the starting point to this cell
    - All other cells have an infinite (or blank or `None`) value

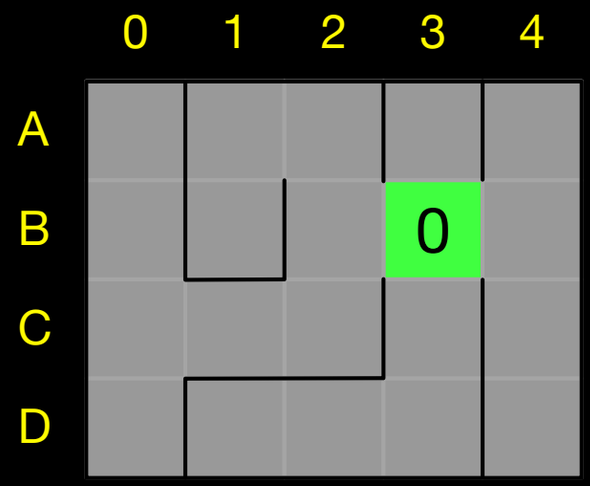
2. Create a collection (set, list, ...) to hold cells and put the root cell in it. We will call this the **frontier**

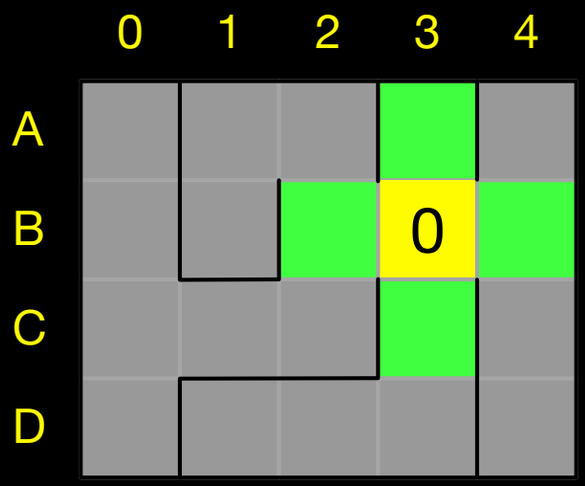
   - The frontier is the "edge" of all the cells whose least-cost path you already know

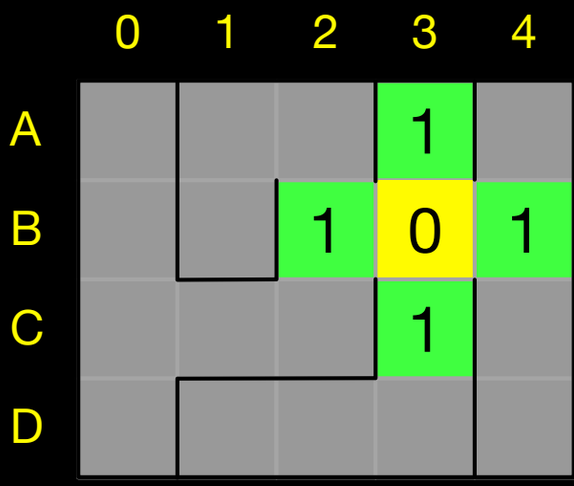
   - For this simplified maze version of Dijkstra's, all cells in the frontier during each phase have the same value

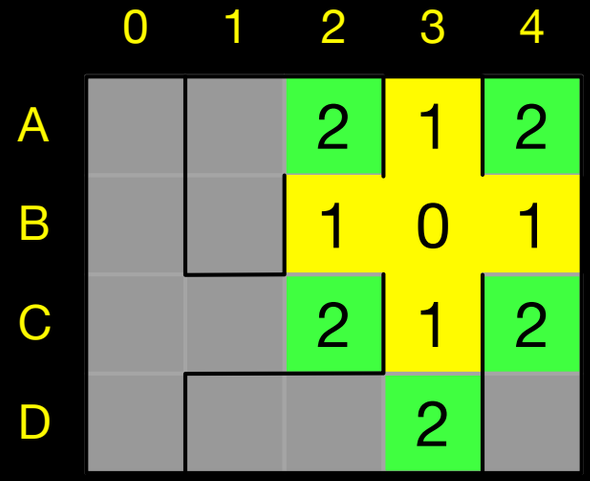
3. Repeat until the frontier is empty

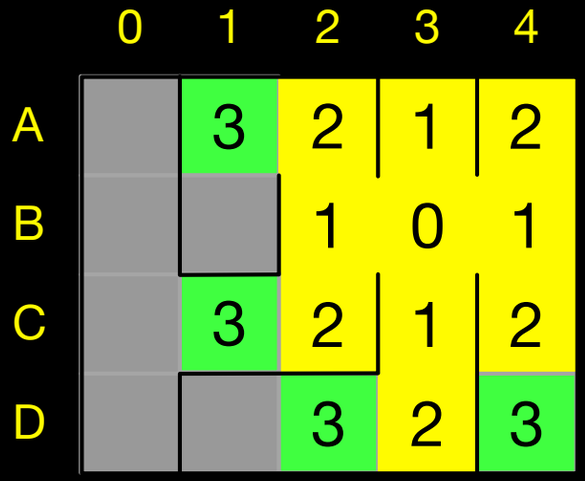
   - Find the cell in the frontier with the smallest value, we will call it cell `c`
   - Remove `c` from the frontier
   - Find all neighbors of `c` which are not already marked
     - Mark them with 1 + `c`'s value
     - Add them to the frontier

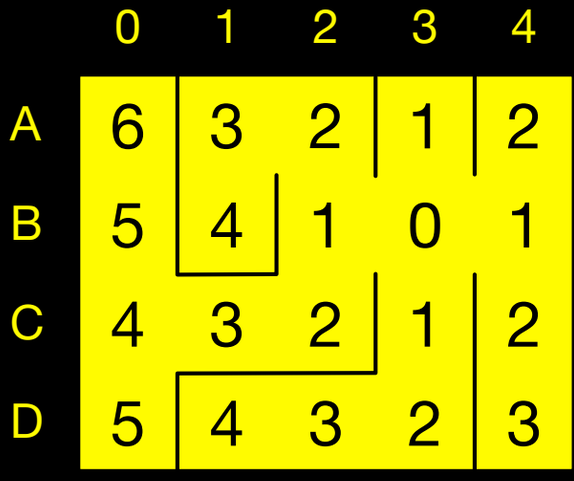

0 1 2 3 4
A
B
C
D
0


0 1 2 3 4
A
B
C
D
0


0 1 2 3 4
A
B
C
D
1
1 0 1
1


0 1 2 3 4
A
B
C
D
2 1 2
1 0 1
2 1 2
2


0 1 2 3 4
A
B
C
D
3 2 1 2
1 0 1
3 2 1 2
3 2 3


0 1 2 3 4
A
B
C
D
6 3 2 1 2
5 4 1 0 1
4 3 2 1 2
5 4 3 2 3

What to do with the values?

- Each value represents the number of cells from it to the root cell
- If the root cell is the exit, you can find a path from the entrance (or any other cell)

    1. Initialize a list called path. Add your target cell to the path
    2. Repeat until at the root (i.e. until the marked value is zero):

        a) Of all the neighbors of the first cell in the path, choose the one with the smallest marked value

        b) Add it to the front of the path

Longest Path

- If you want to know the longest path in the entire maze (i.e. longest among all paths),
- Do Dijkstra's algorithm twice
  - Pick a random cell for the root (let's call it A)
    - Run Dijkstra's on A
  - You now know the distance from that cell to the furthest cell away from A (it has the highest value in the markup). Choose that cell (B)
    - Run Dijkstra's on B.
  - Find the cell with the highest markup on this second run (C)
- Longest path is B → C

- Longest path can be very useful
  - The longer it is, perhaps the more complex the maze
    - i.e. a *metric* for how hard the maze is
  - You can use it to place entrance/exit for longer adventures
    - either on the ends, or "20% from the end"

A look ahead

- We will visit Dijkstra's algorithm again
- We can make it more efficient (don't visit every cell in the maze) for navigational tasks

What did you learn today?

- Maze Algorithms
  - Generation via: Binary Tree, Sidewinder, A-B, Wilsons, Recursive Backtracking
  - Solution via Dijkstra's Algorithm