Advanced Programming final project

Percolation simulation

Modeling the flow of liquid through random grids

May, 2025

Introduction to percolation

What is percolation?

Process of **liquid seeping through porous material**

Real world examples

Water seeping through soil

Spread of fire or disease

Importance

Physics

Materials science

Percolation in programming

a **simulation** of a percolation process with a grid of open and blocked sites

Goal: determine if a path exists from one side of the grid to the other through open

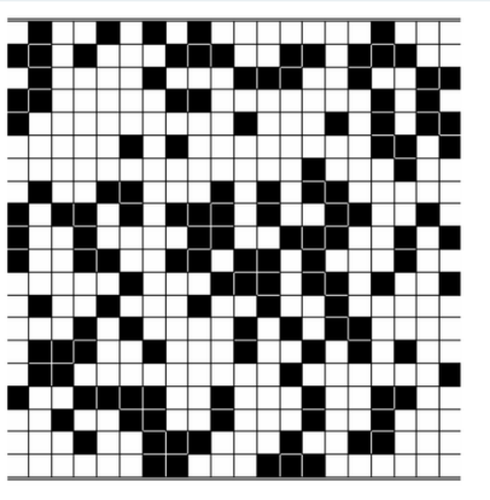
Objective

What is the goal?

Develop Python code to simulate percolation

Steps

- Simulate percolation in 2D grids
- Generate random grids with open (1) and closed (0) cells
- Detect if there's a connected path from top to bottom
- Analyze how likelihood of percolation changes with open-cell probability



Matrix generation

- Parameters:
  - n for the size of the matrix
  - prob is the ratio of how many grids should be open out of n*n
- We created a list containing prob*n*n 1s , the rest are 0s, and then shuffled it so the places of 1s are random
- then reshaping it as an n*n matrix

```python
import matplotlib.pyplot as plt
import numpy as np
def generate_matrix(n, prob=0.5):
    size = n+2
    num_grid = n**2
    open_ratio = prob
    num_open = int(num_grid*open_ratio)

    flat = np.array([0]*(n**2-num_open)+[1]*num_open)

    np.random.shuffle(flat)
    matrix = np.zeros((size,size))

    matrix[1:-1, 1:-1] = flat.reshape((size - 2, size - 2))

    return matrix
```

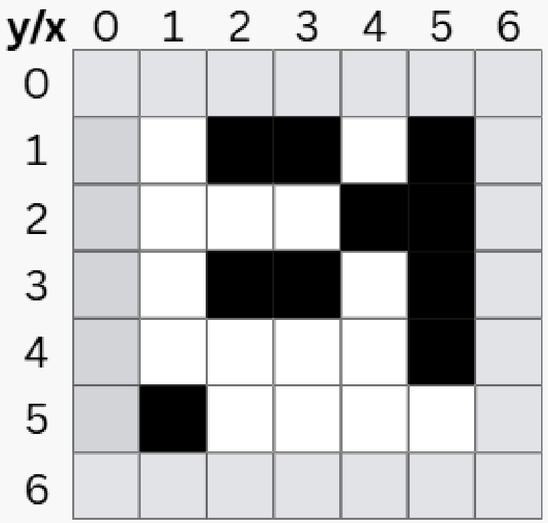
```
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1]
```

```
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 1. 0. 0.]
 [0. 0. 1. 0. 1. 1. 0.]
 [0. 1. 1. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
```

Matrix generation

- We added an outer layer of 0s to our matrix so when we loop through the grids we dont have to worry about getting out of bounds

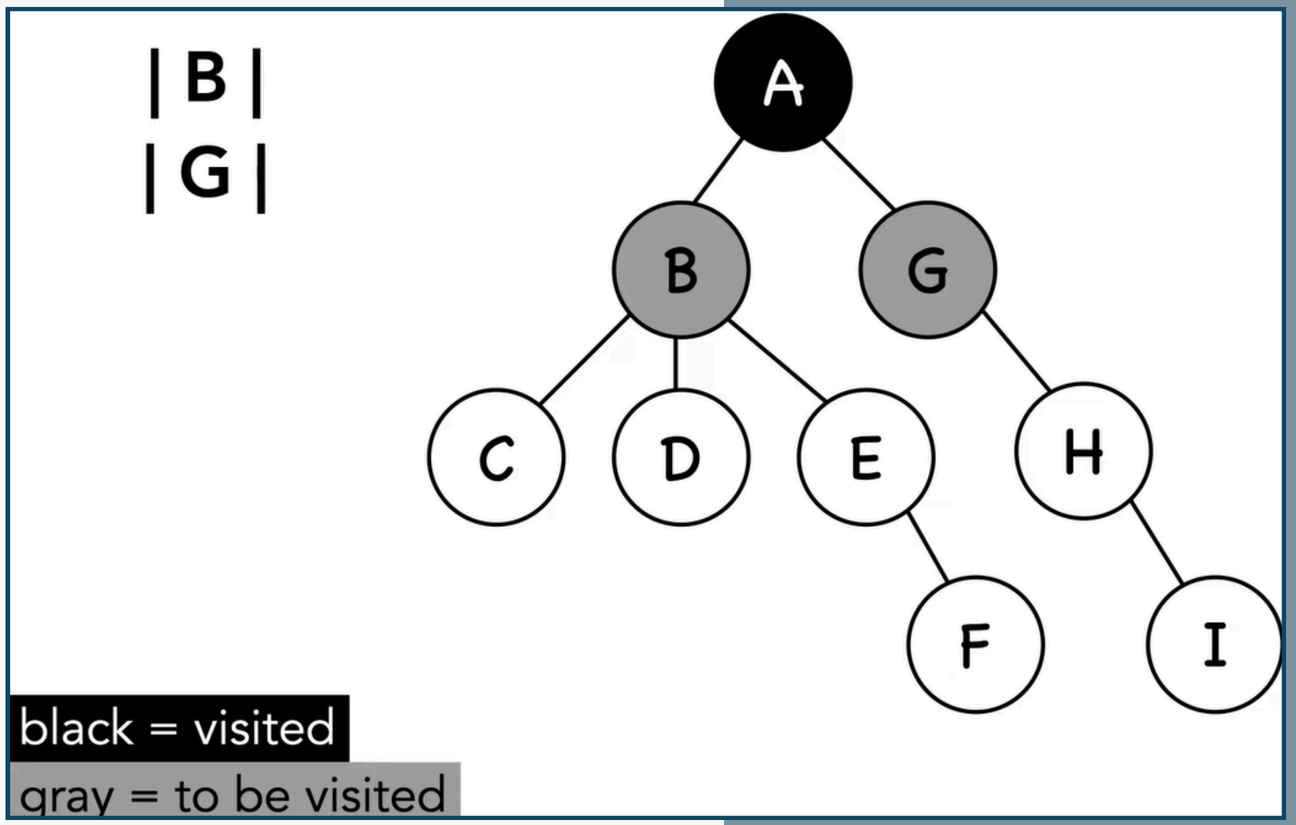
- In this case n=5 so we have a 7*7 matrix that has a 5*5 matrix with a layer of 0s surrounding it




```python
import matplotlib.pyplot as plt
import numpy as np
def generate_matrix(n, prob=0.5):
    size = n+2
    num_grid = n**2
    open_ratio = prob
    num_open = int(num_grid*open_ratio)

    flat = np.array([0]*(n**2-num_open)+[1]*num_open)

    np.random.shuffle(flat)
    matrix = np.zeros((size,size))

    matrix[1:-1, 1:-1] = flat.reshape((size - 2, size - 2))

    return matrix
```

Depth-first search (DFS)

- Mostly used for path finding on Graphs
- It explores as far down a branch as possible before backtracking
- Consists of two main lists:
  - Visited list
  - Stack list



Visited list

- Each element that is popped, is then appended to the Visited list so we know which elements we've checked

Stack list

- We **loop** through this list until it has no more elements.
- We check the surroundings of the current element and also pop it

Percolation detection: finding a path from top to bottom

Stack-based DFS (stack_check()):

- Use a stack to simulate Depth-First Search (DFS) instead of recursion
- Start from all open cells (1) in the top row.

Exploration from top row:

- Push top row open cells into the stack.
- Explore neighboring cells (up, down, left, right) if **they are open (1)** & **not visited**.

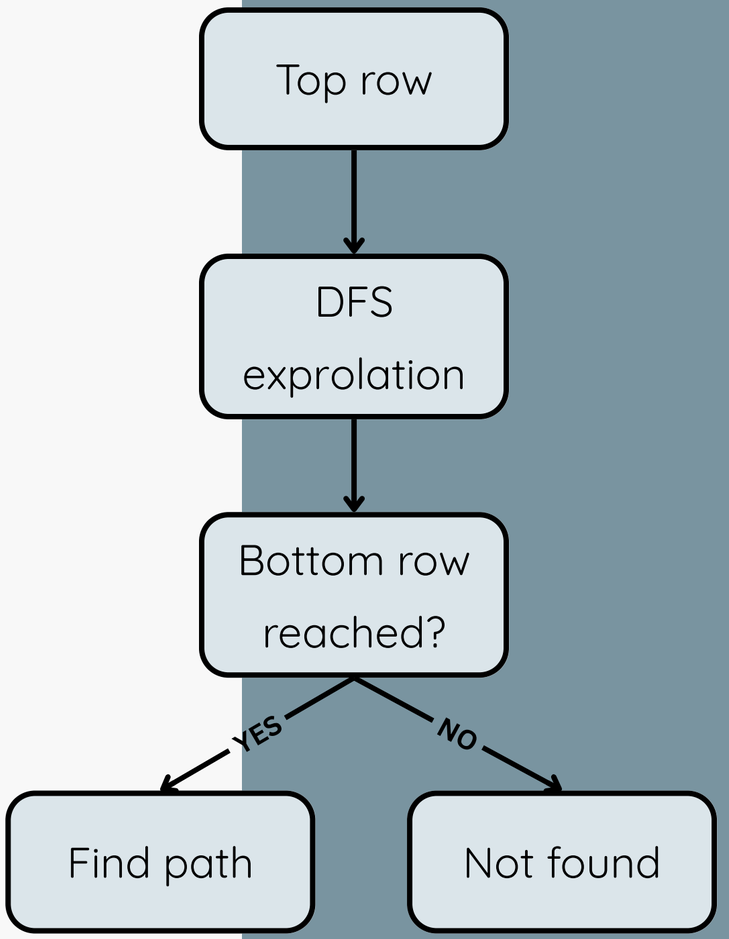
```python
def stack_check(matrix):
    n = len(matrix)
    stack = []
    visited = set()

    for x in range(1,n-1):
        if matrix[1][x]:
            stack.append((1,x))
            visited.add((1,x))

    directions = [(0,1),(1,0),(-1,0),(0, -1)]

    while stack:
        y,x = stack.pop()

        for dy,dx in directions:
            nexty,nextx=y+dy,x+dx
            if matrix[nexty][nextx] and (nexty,nextx) not in visited:
                stack.append((nexty,nextx))
                visited.add((nexty,nextx))
    return visited
```

Percolation detection:
Finding a path from top to bottom

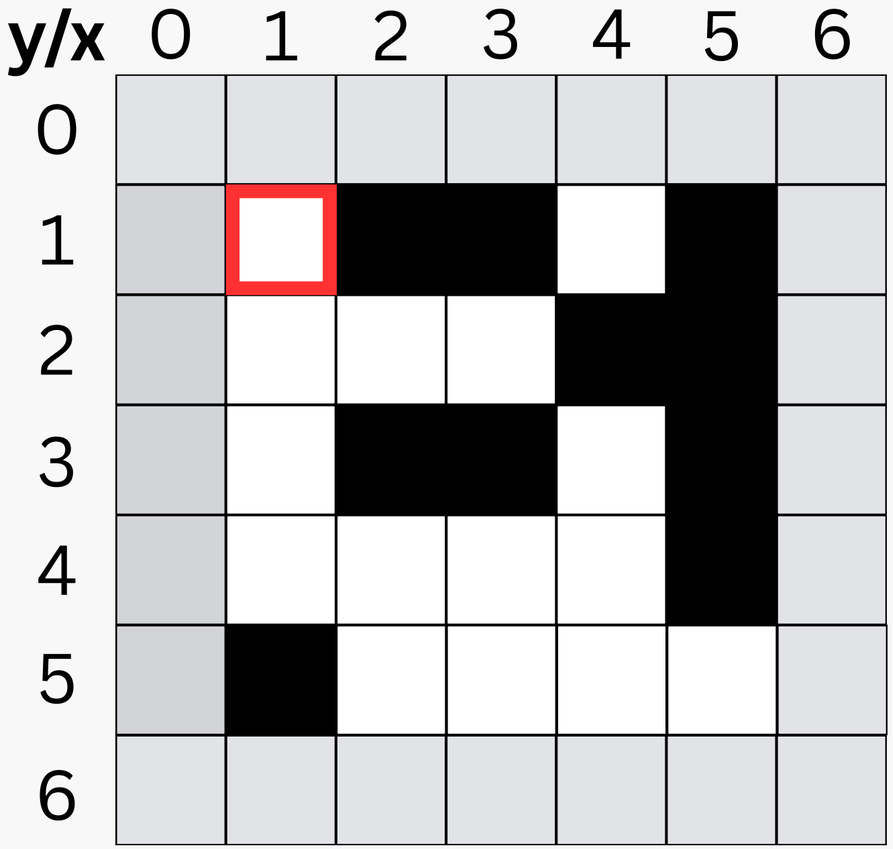
Checking reachability (find_path()):

- During DFS, if reach any open cell on the bottom row, percolation occurs (path exists from top to bottom).
- If no such path is found after exploration, the system does not percolate.

```python
def find_path(matrix, visited):
    n = len(matrix)
    expect = [(n-2,x) for x in range(1,n-1)]
    for i in visited:
        if i in expect:
            return True #if you need the percola
    return False
```



First for Loop: Check 1st(Top) Row n=5



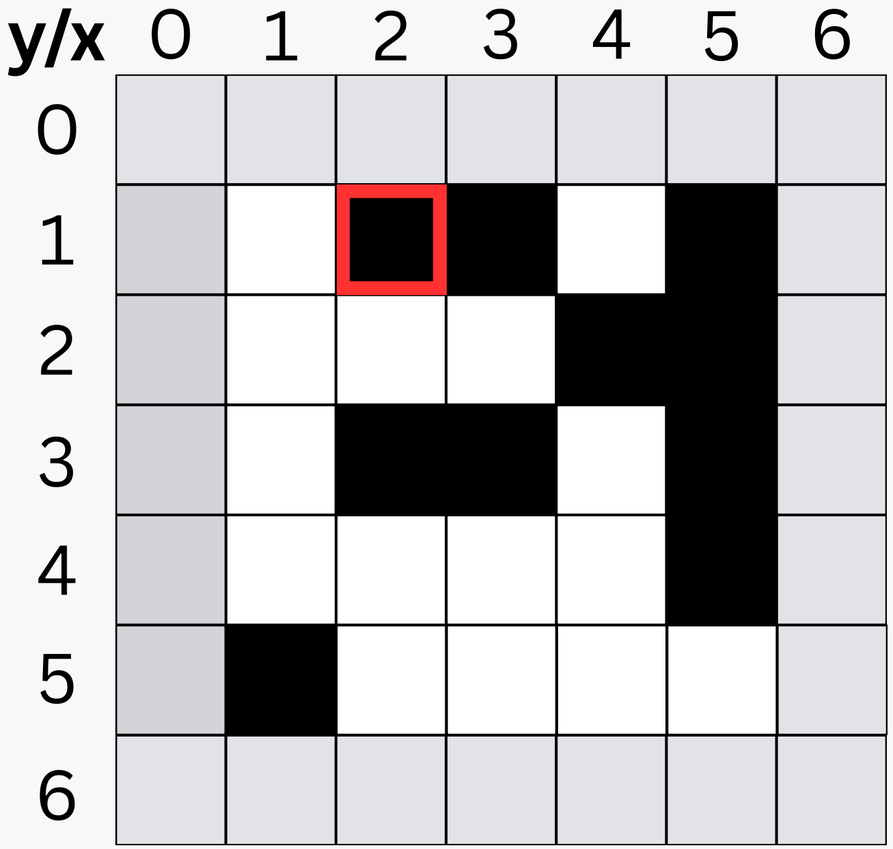
stack(list)

Current cell(y,x)

(1,1)

visited(set) = {(1,1)

}



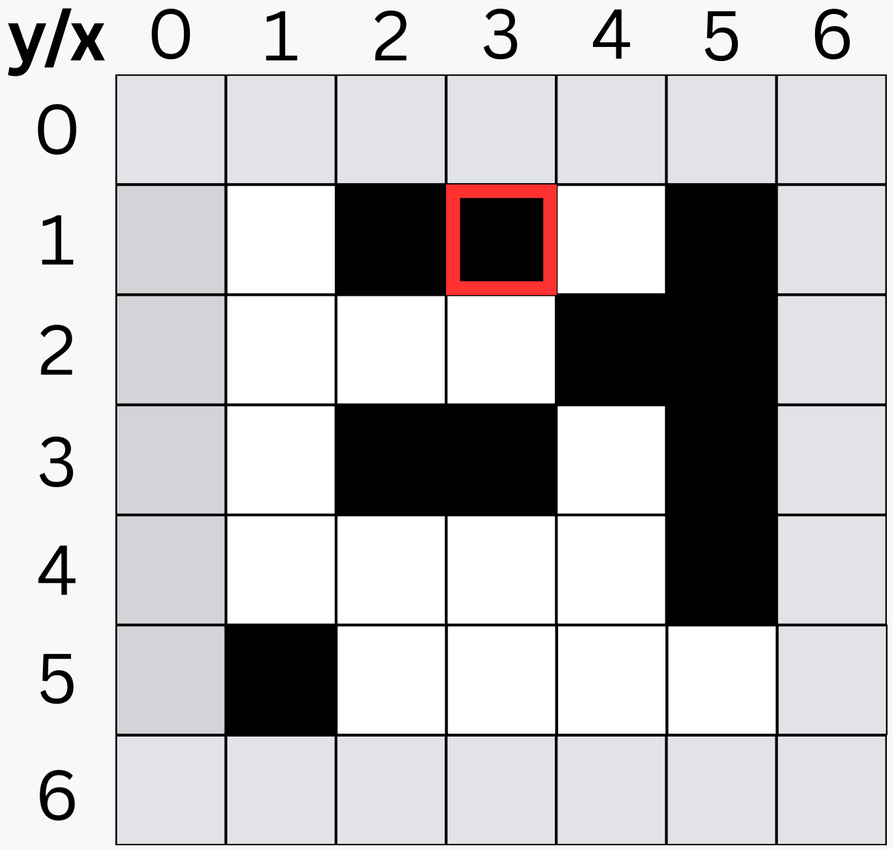
stack(list)

Current cell(y,x)

(1,2)

visited(set) = {(1,1)

}

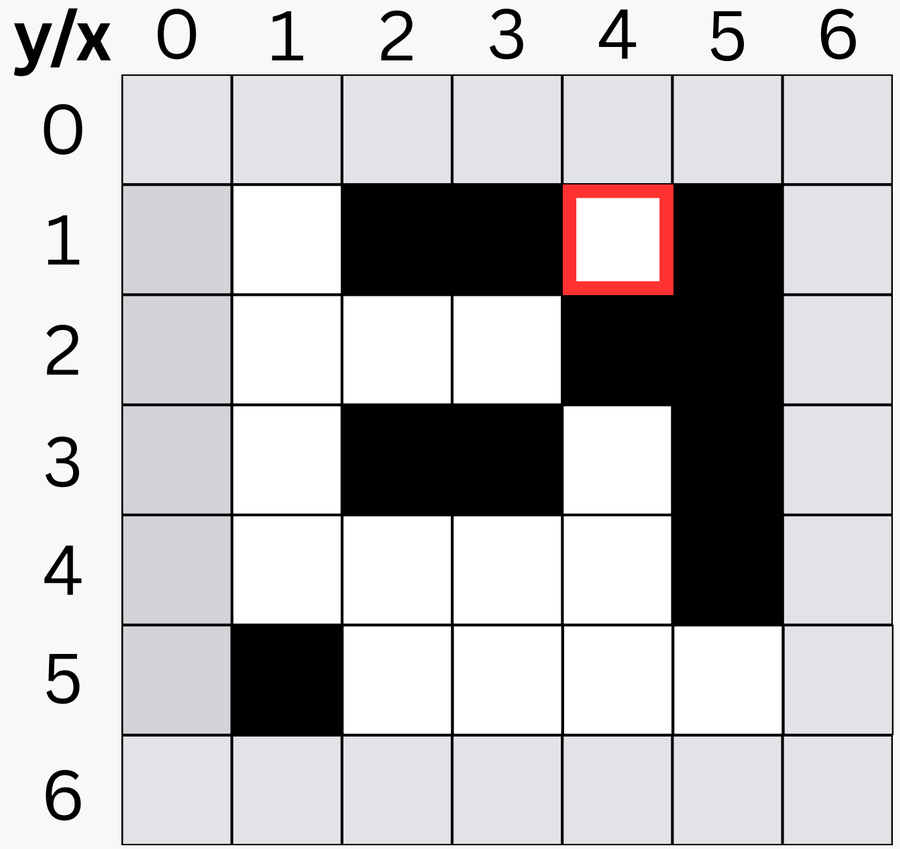


stack(list)

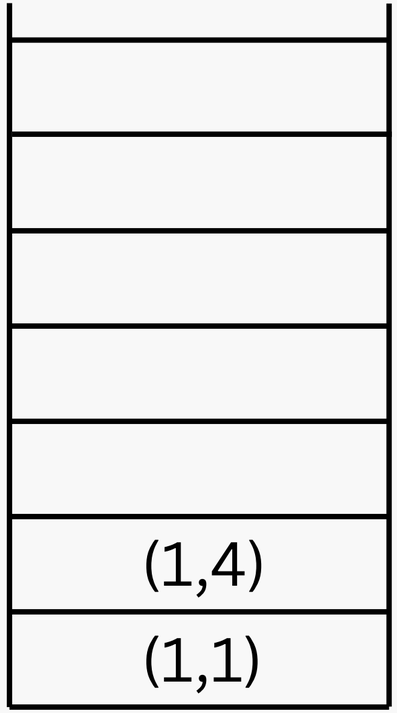
Current cell(y,x)
(1,3)

visited(set) = {(1,1)

}



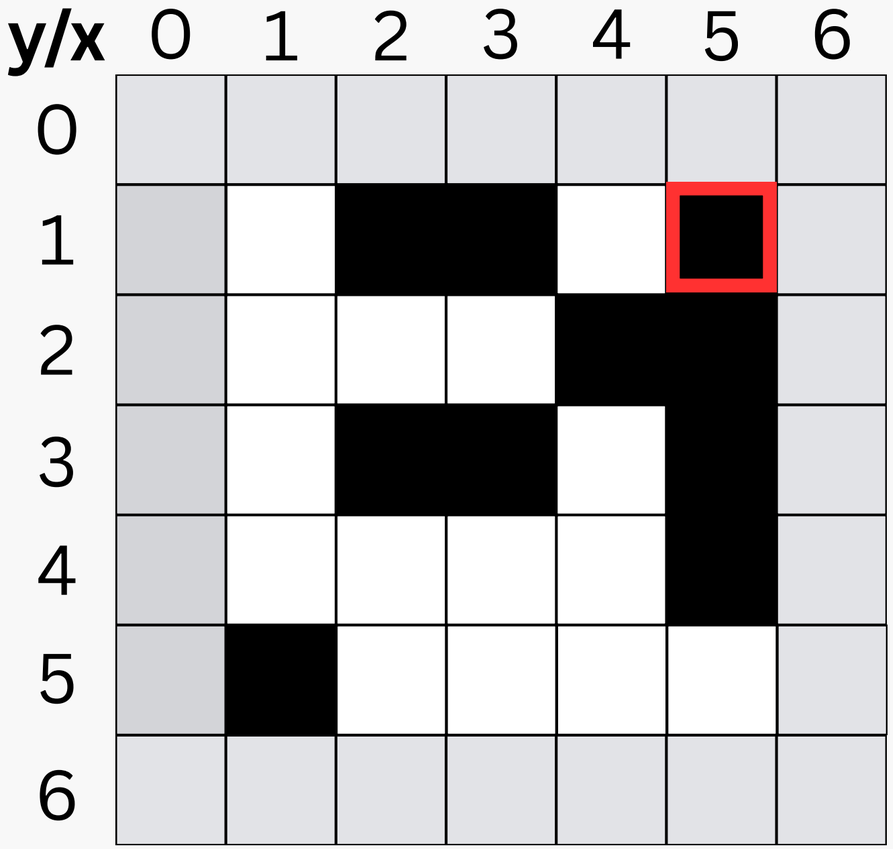
stack(list)



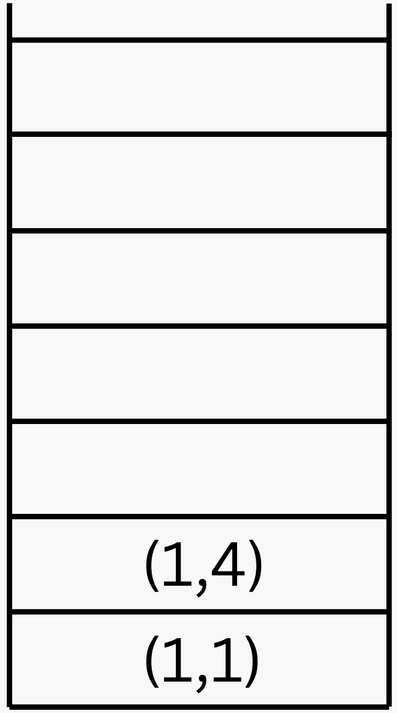
Current cell(y,x)
(1,4)

visited(set) = {(1,1),(1,4)

}



stack(list)

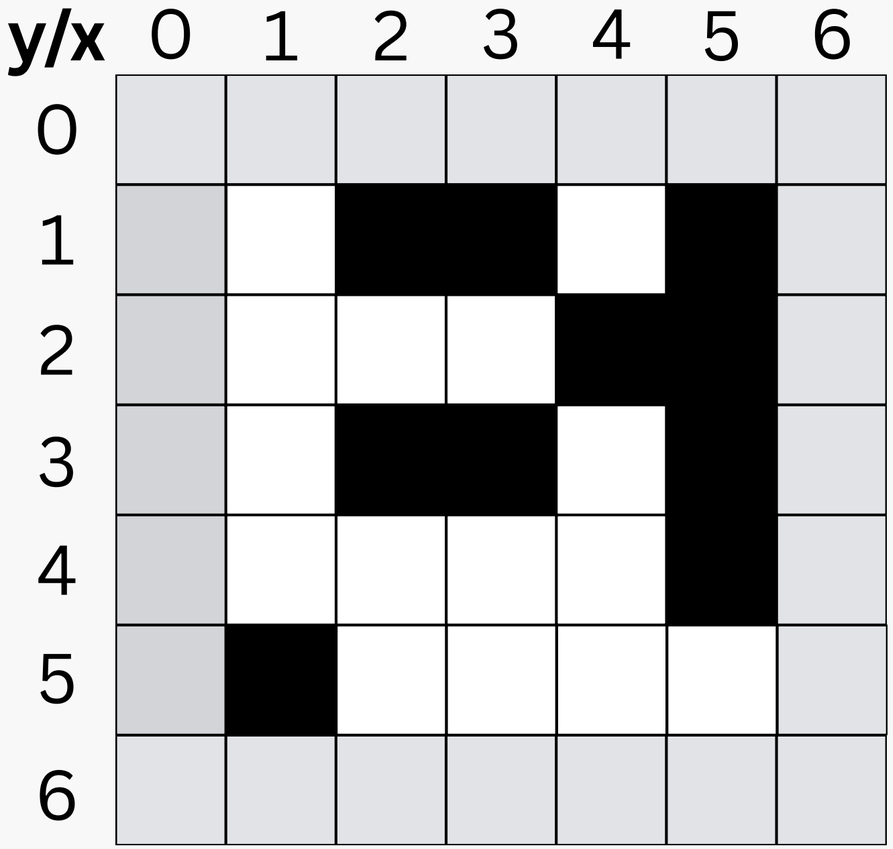


Current cell(y,x)
(1,5)

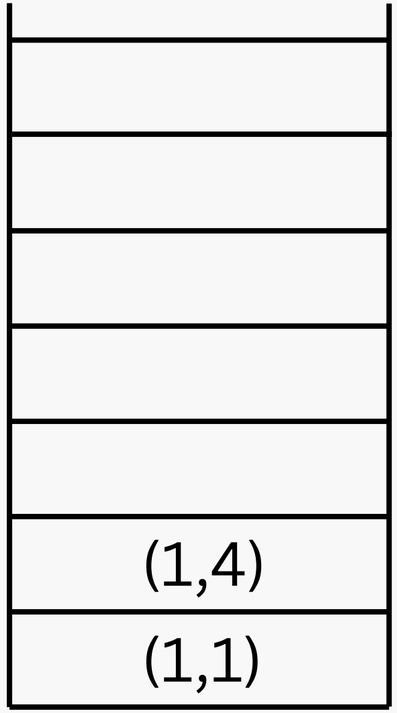
visited(set) = {(1,1),(1,4)

}

Second for Loop: Check 4 Directions Until The Stack Is Empty (DFS exprolation)



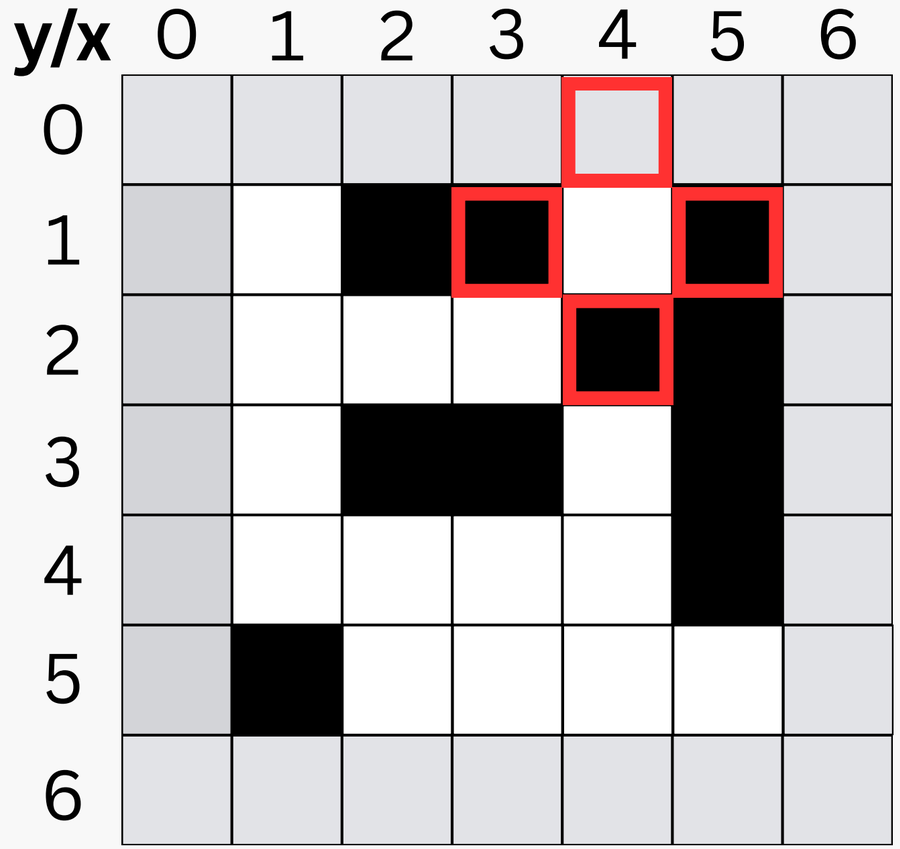
stack(list)



<- pop from last

Current cell(y,x)
(1,4)

visited(set) = {(1,1),(1,4) }



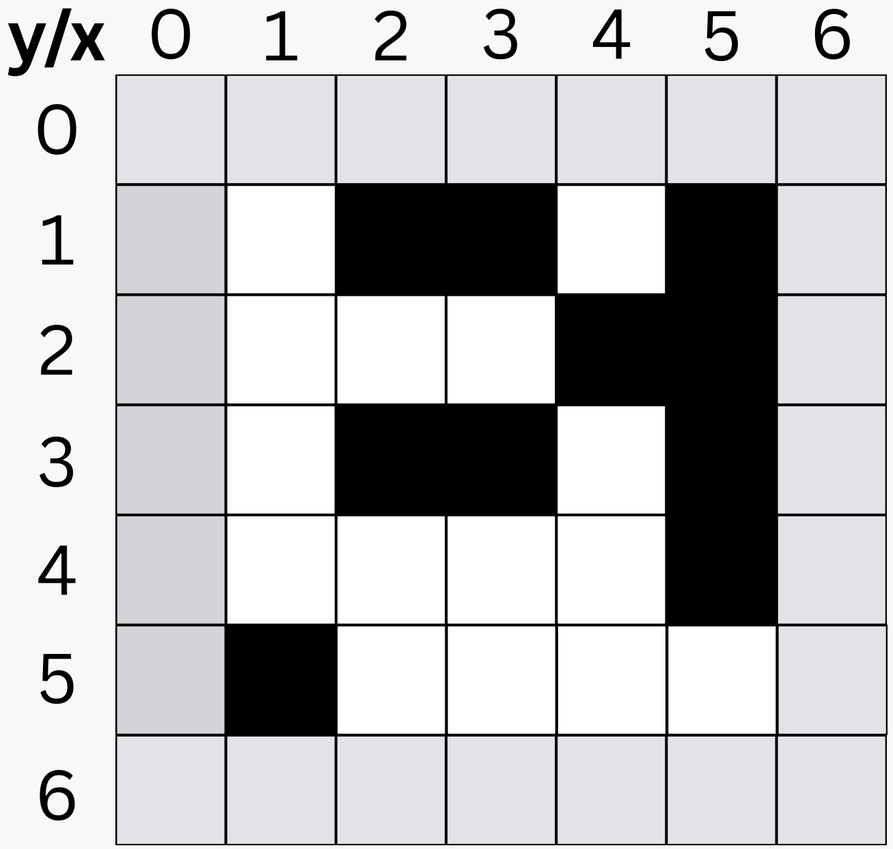
stack(list)

Current cell(y,x)

(1,4)

visited(set) = {(1,1),(1,4)

}



stack(list)

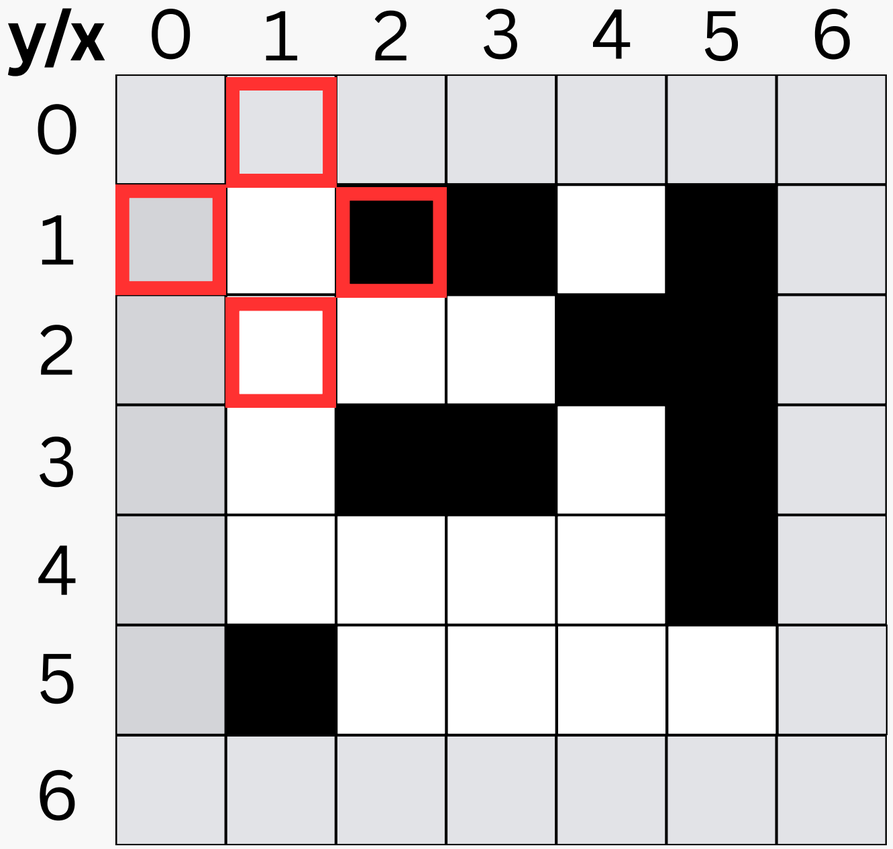
<- pop from last

Current cell(y,x)

(1,1)

visited(set) = {(1,1),(1,4)

}



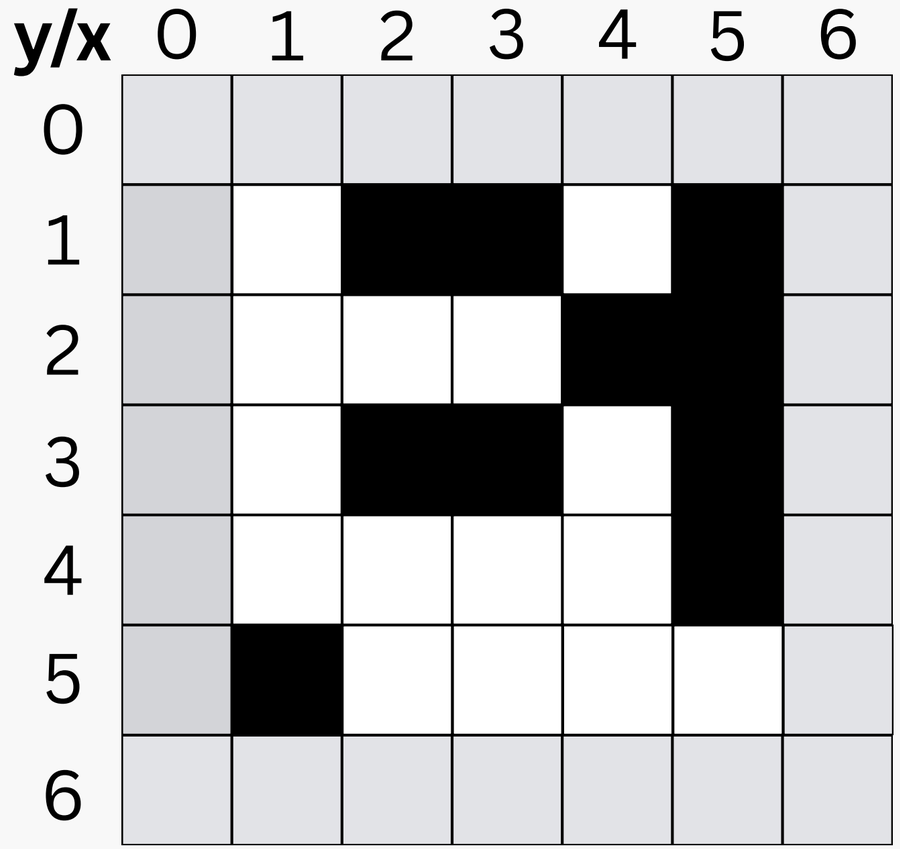
stack(list)

Current cell(y,x)

(1,1)

visited(set) = {(1,1),(1,4),(2,1)

}



stack(list)

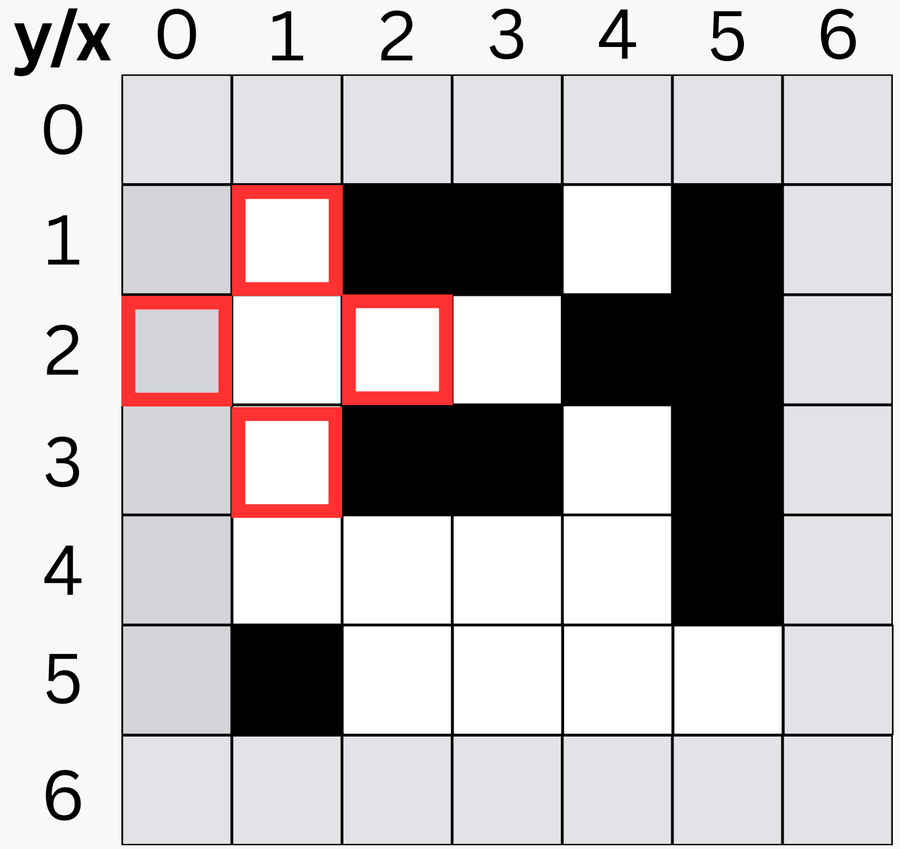
<- pop from last

Current cell(y,x)

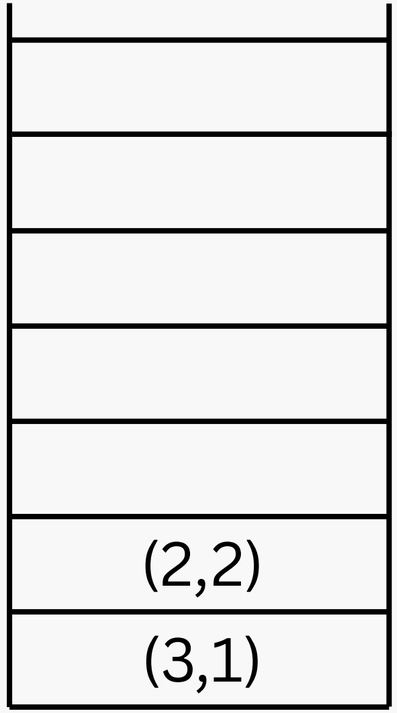
(2,1)

visited(set) = {(1,1),(1,4),(2,1)

}

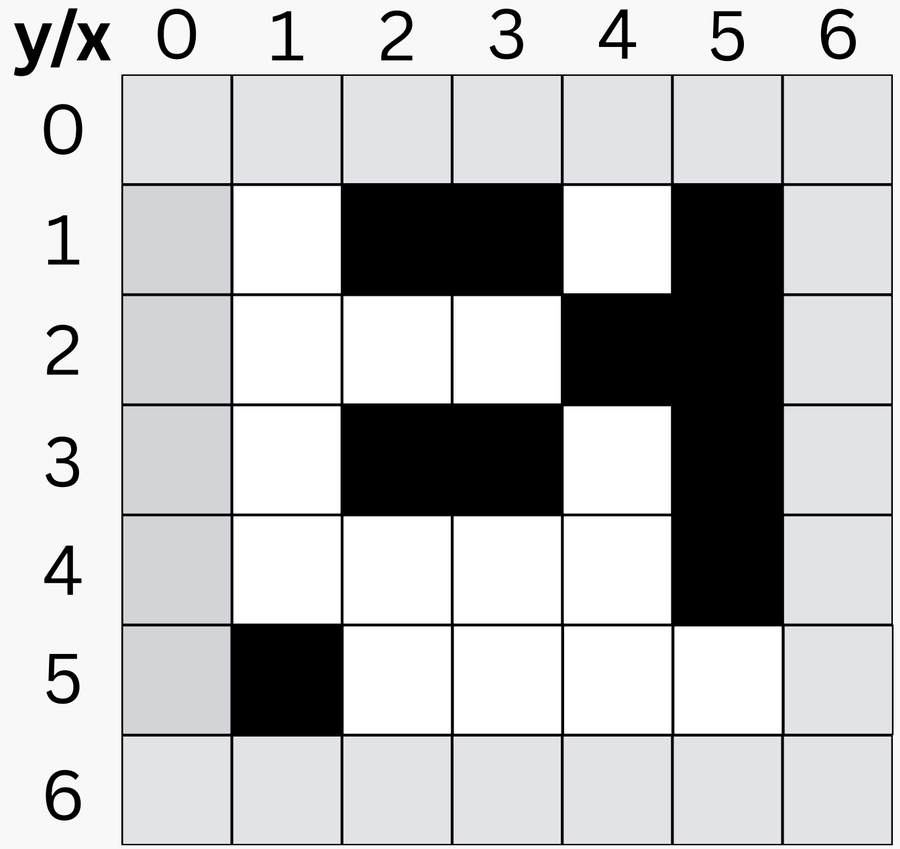


stack(list)

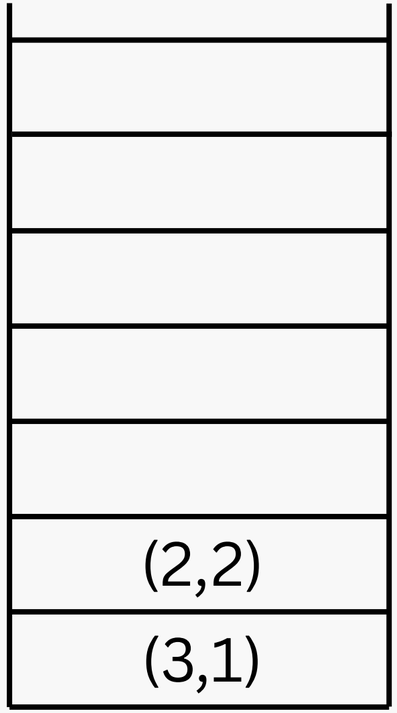


Current cell(y,x)
(2,1)

visited(set) = {(1,1) ,(1,4) ,(2,1) ,(3,1) ,(2,2) }



stack(list)

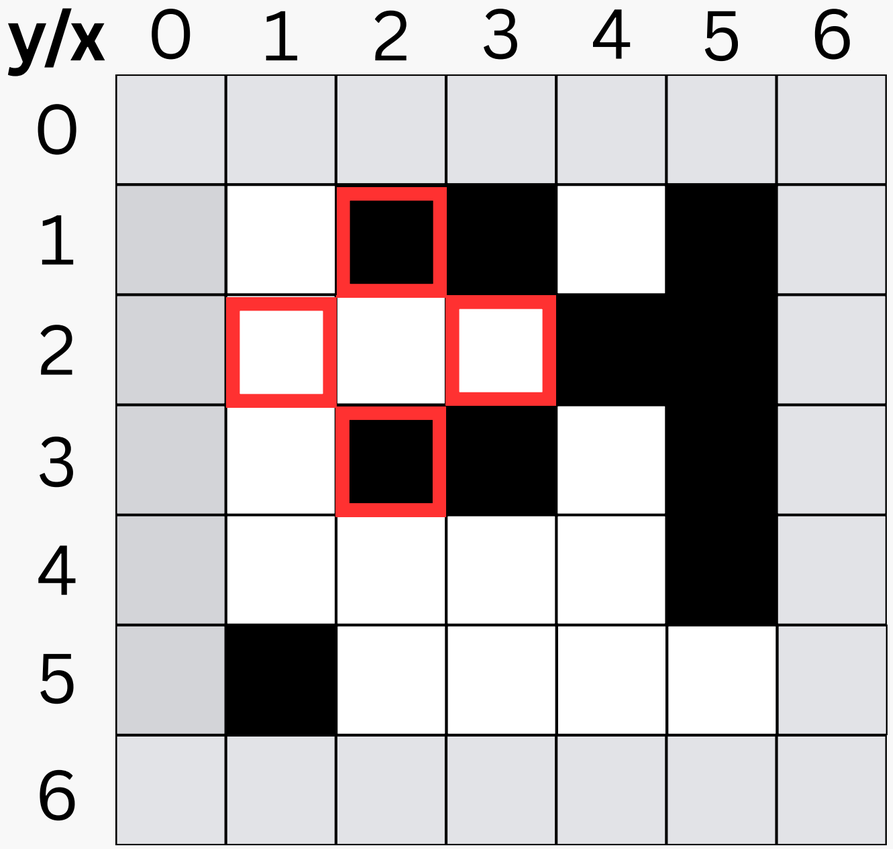


<- pop from last

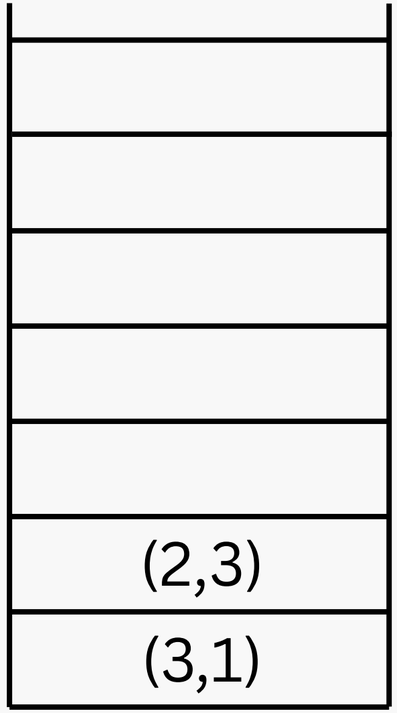
Current cell(y,x)

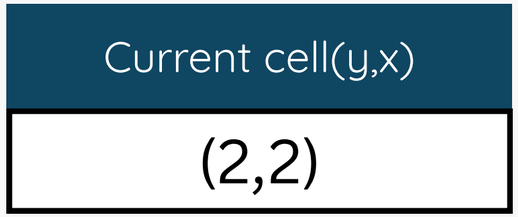
(2,2)

visited(set) = {(1,1) ,(1,4) ,(2,1) ,(3,1) ,(2,2) }



stack(list)

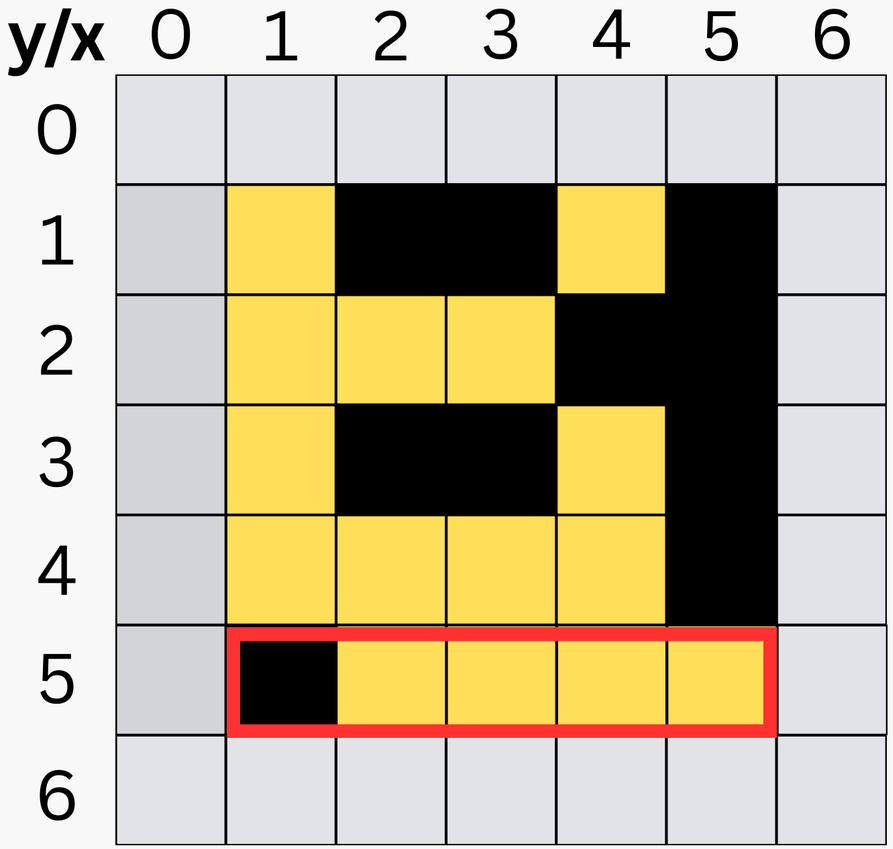




Continue until stack is empty…

visited(set) = {(1,1),(1,4),(2,1),(3,1),(2,2),(2,3) }

Bottom Row Reached?



stack(list)

Current cell(y,x)

visited(set) = {(1,1),(1,4),(2,1),(3,1),(2,2)(2,3),(4,1),(4,2),(5,2),(4,3),(5,3),(4,4),(5,4),(3,4),(5,5)}

Percolation exists

Simulation design

```
def simulate_percolation(grid_sizes=[10, 20, 50, 100],
prob=np.arange(0.3, 1.01, 0.01), trials=100):

    for n in grid_sizes:
        results = []
        for p in prob:
            c= 0
            for i in range(trials):
                matrix = generate_matrix(n, p)
                visited = stack_check(matrix)
                if find_path(matrix, visited):
                    c+= 1
            results.append(c/trials)
#ratio of success percolation out of all trials
```

*Focused only on the simulation on values around the percolation threshold (p>0.3) to avoid wasting computational resources because the success rate would be very close to 0

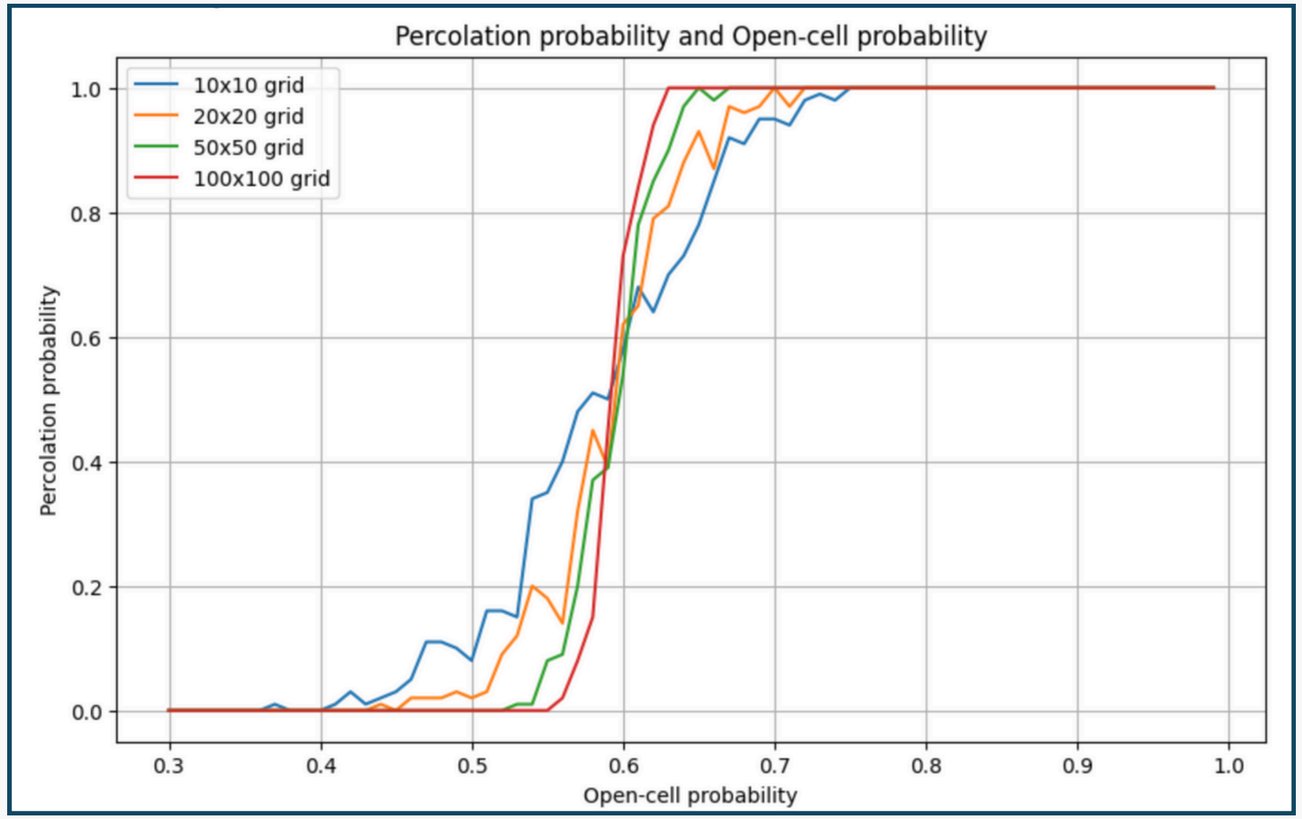
Simulated for various

- Grid sizes (e.g., 10x10, 20x20, 50x50, 100x100)

- Open-cell probabilities (0.3* to 1.0)

- Repeated each simulation multiple times (100 trials)
to ensure stable results

Data analysis



The percolation threshold becomes clearer and more precise as grid size increases

Conclusions

- Low p (< 0.5): Percolation is unlikely
- Around p ≈ 0.59: Sharp transition (critical threshold)
- High p (> 0.7): Percolation is almost certain

Grid size effects:

- Smaller grids (10x10): Transition is smoother due to finite-size effects—randomness dominates
- Larger grids: Transition is sharper, more like a step function
- Similar trends
- This sharpness converges toward the theoretical percolation threshold as grid size increases
- Bigger the grid, smaller the fluctuation

Data analysis

Conclusions

- Time increases nonlinearly with grid size
- Approximate scaling suggests quadratic or worse complexity (likely $O(n^2)$ to $O(n^3)$), due to:

1. Matrix generation: $O(n^2)$
2. Pathfinding algorithm (DFS): can be $O(n^2)$ or worse

20 trials, prob=(0.5, 1.0, 0.1)

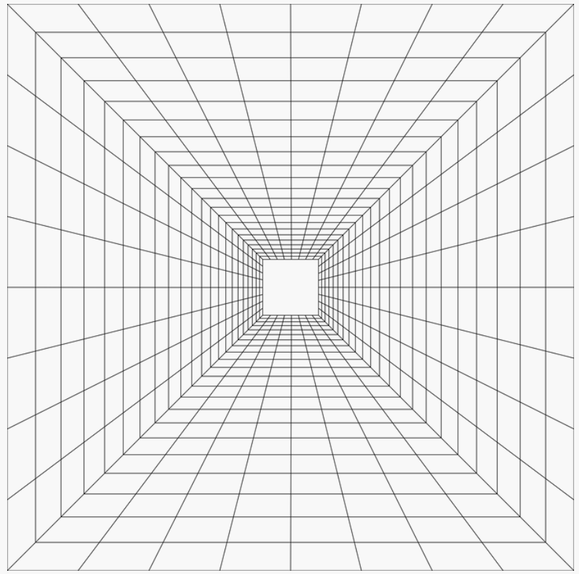
```
Time taken for grid size 1000x1000: 76.22 seconds
```

100 trials, prob=(0.3, 1.0, 0.01)

```
Time taken for grid size 10x10: 0.35 seconds
Time taken for grid size 20x20: 1.31 seconds
Time taken for grid size 50x50: 7.92 seconds
Time taken for grid size 100x100: 32.06 seconds
```

Simulation time scales steeply with grid size, so optimising matrix generation and pathfinding is key for performance on large grids

Optimisation and variants



Stack-based DFS advantage:

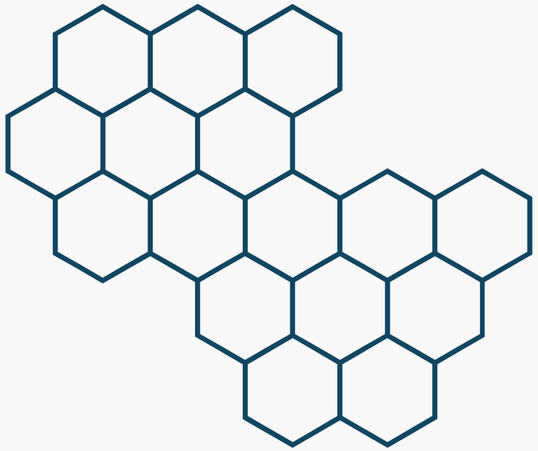
- Avoids recursion limit issues in Python (no RecursionError for large grids).
- Manages memory better for deep explorations.

Dynamic programming possibility:

- Store visited paths or memoize reachable cells to avoid redundant checks.
- Especially useful if checking percolation multiple times with slight changes.

Possible extensions:

- **3D Grids**: Instead of 2D matrices, simulate percolation in 3D cubes.
- **Hexagonal Grids**: Use hexagonal neighbors (6 directions) instead of square (4 directions).



Challenges faced

01 Visualisation clarity

02 Ensuring randomness

03 Efficient algorithm and simulation for large grids

Future work

Animate the percolation process

Add GUI for interactive simulation

Explore other percolation types (bond, directed)

Use real-world terrain data

Conclusion

·Successfully **simulated and visualised percolation**

·Found **relationship between grid size and percolation** threshold

·Learned **key techniques** in simulation, search algorithms, and data visualisation