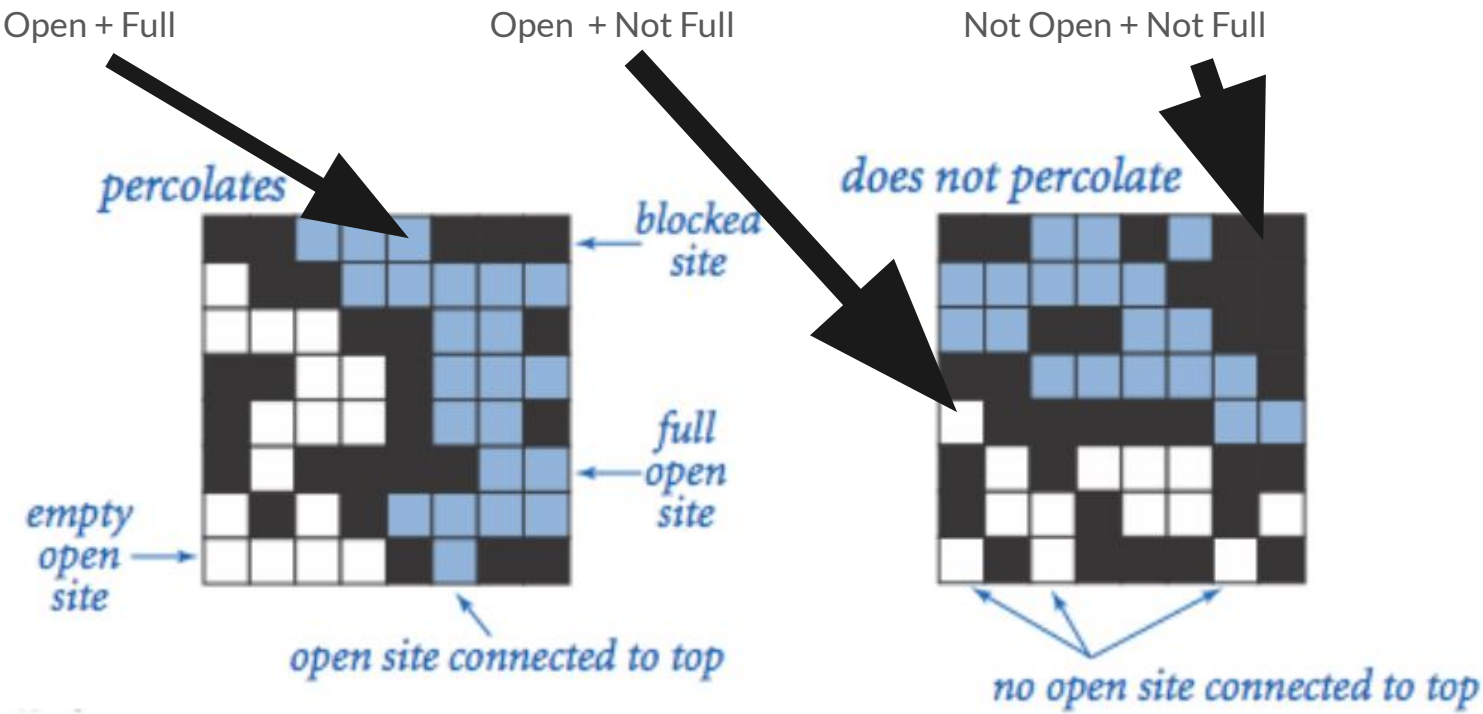
Discussion 2: The Percolation Project

Kyle Hackett

3 States: Open + Not Full, Open + Full, Not Open + Not Full



Problem 1: Array Percolation

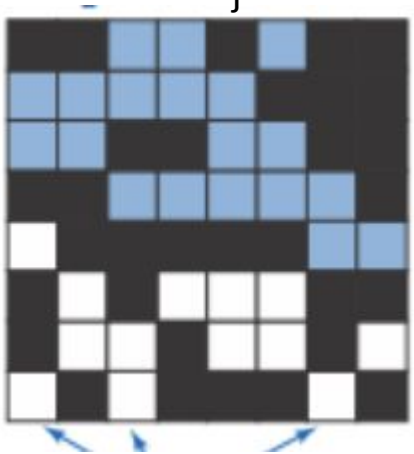
Problem 1. (*Array Percolation*) Develop a data type called `ArrayPercolation` that implements the `Percolation` interface using a 2D array as the underlying data structure.

`ArrayPercolation implements UF`	
`ArrayPercolation(int n)`	constructs an `n x n` percolation system, with all sites blocked

ixj

i

j



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Percolation

`void open(int i, int j)`	opens site `(i, j)` if it is not already open
`boolean isOpen(int i, int j)`	returns `true` if site `(i, j)` is open, and `false` otherwise
`boolean isFull(int i, int j)`	returns `true` if site `(i, j)` is full, and `false` otherwise
`int numberOfOpenSites()`	returns the number of open sites
`boolean percolates()`	returns `true` if this system percolates, and `false` otherwise

Instance Variables

- Instance variables:
  - Percolation system size, `int n`.
  - Percolation system, `boolean[][] open` (`true` $\implies$ open site and `false` $\implies$ blocked site).
  - Number of open sites, `int openSites`.

Constructor

```
// Constructs an n x n percolation system, with all sites blocked.
public ArrayPercolation(int n) {
    //...
}
```

Here you initialize the empty 2d array instance variable and the others

Remember! Any counters should be initialized to 0

this.instancevariable = yadayadayada

Setters

- Open(int i, int j)
  - Takes an i j pair
  - Manipulates the open 2d array which holds what sites are open or closed

Getters

- numberOfOpenSites() -> returns the number of open sites
- isOpen(i, j) - > returns true or false if a site is open, looking at the 2d array called open, increment counter

Do-ers

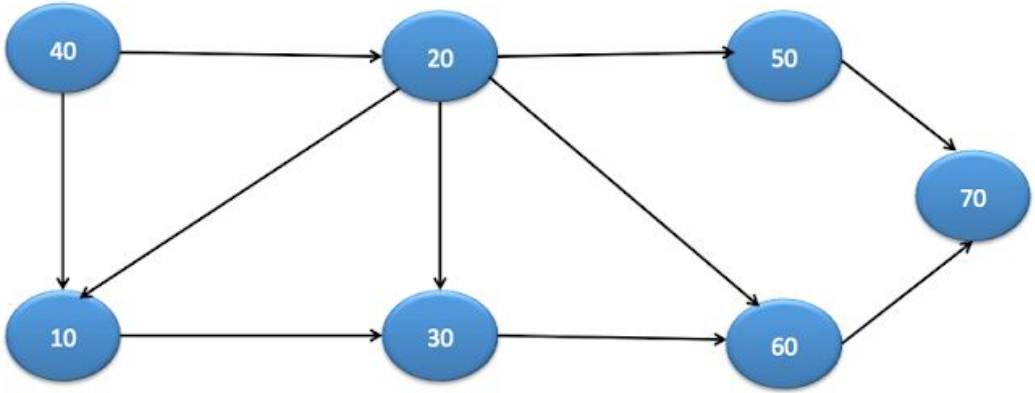
- percolates()
  - Returns true or false depending on if there is a path of full + open squares top to bottom
  - Hint: You will need to use the isFull Method here and a system percolates if the liquid reaches the bottom from the top.
- isFull(int i, int j)
  - Takes a pair
  - Create a n x n array called full -> all initialized to false
  - Call floodFill() on every site in the top row of percolation system, passing full as the first argument
  - Returns the value stored in full[i][j]

Flood Fill

- FloodFill takes int i, int j, and the 2D array full
- Check for illegal i, j, n, closed site, or already full sites
- Fill The Given IxJ site
- Then Perform Depth First Exploration, calling FloodFill
    - More on this on the next slide

Depth First Exploration

A Given (I, J) Point will have At Most 4 neighbors, and at least 2 Neighbors if in a corner



Depth first traversal of above graph can be :40,20,50,70,60,30,10

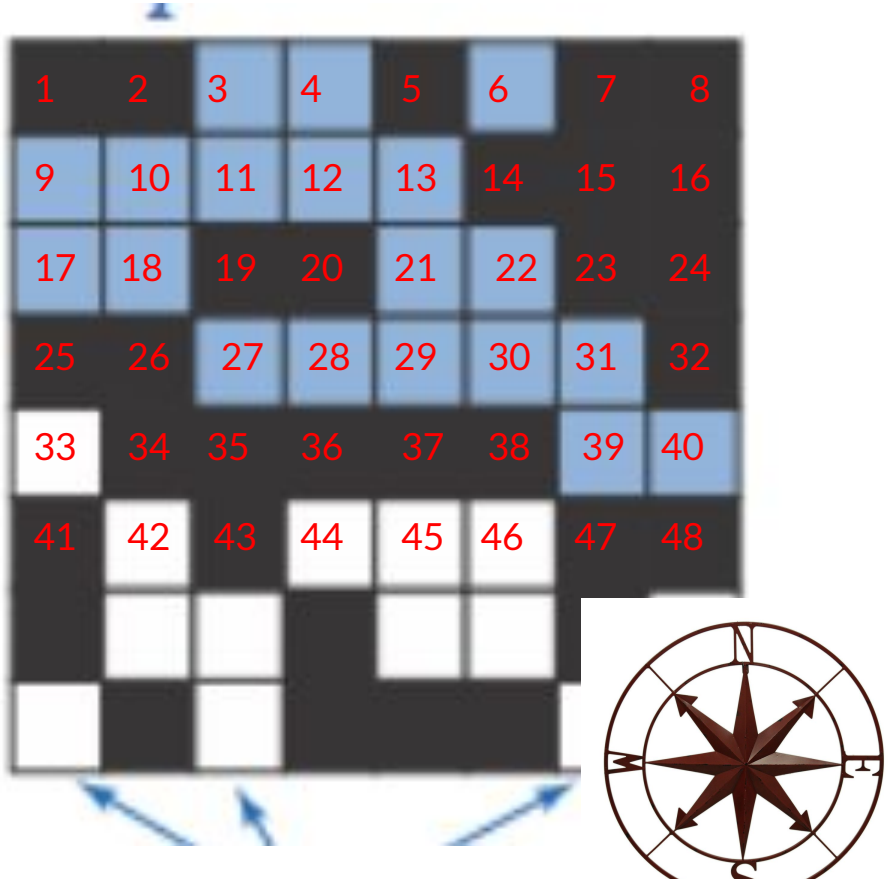
We start with node 40. It then visits node 20, node 50, node 70 respectively as they are directly connected. After that, it backtracks to node 20 and visited node 60, node 30 and node 10 respectively.

In otherwords, start on one path, and go as far as you can in that direction until you cannot continue any further.
Then you jump back to the last point that has a viable neighbor

Call floodFill() recursively on the sites next to site (i, j).

- Start at 1 Its closed. Return.
- Start at 2. Its closed. Return.
- Start at 3. Its open;
  - mark as full. Look W, closed. Back at 3. Look N, out of bounds. Back at 3. Look E, its open!
- Now at 4. Its open:
  - Mark as full, look W, its full. Back at 4. Look N, out of bounds. Back at 4. Look E, its closed. Back at 4. Look south. Its open!
- Now at 12, set to full. Same goes for 11, 10, 9. Then 17, Then 18. But now youre trapped. Backtrack to most recent site with viable neighbor.
- Back at 12. Look North, Then East, and 13 is open. So continue

In This system you will get to spot 40 and have nowhere to go. Ending the recursion and returning a 2d array named full whose bottom row has open + not full sites.



Questions?

Dont Forget to throw Exceptions from the PDF AND to pay attention to the runtimes.

T(n) ~ 1 means its just going to return a value you dont need to calculate

T(n) ~ n^2 refers to going through each element in a nxn system

T(n) ~ n refers to going through one dimension of a nxn system

And so on

Problem 2: Union Find Percolation

UFPercolation implements UF	
`UFPercolation(int n)`	constructs an `n x n` percolation system, with all sites blocked

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

This form of UF is Easier than whats been done in class, but you will still need to know the harder versions for the exams

Instance Variables

- Instance variables:
  - Percolation system size, `int n`.
  - Percolation system, `boolean[][] open`.
  - Number of open sites, `int openSites`.
  - Union-find representation of the percolation system, `WeightedQuickUnionUF uf`.

  You can add more instance variables as you see fit. This will change the space complexity but that will not be something you have to worry about in this course. However, unnecessary custom instance variables will overcomplicate your code.

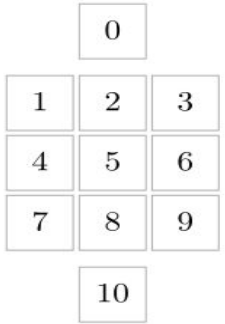
WeightedQuickUF: https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/WeightedQuickUnionUF.html

`WeightedQuickUnionUF(int n)`
Initializes an empty union-find data structure with `n` elements `0` through `n-1`.

`void`	`union(int p, int q)` Merges the set containing element `p` with the the set containing element `q`.
`boolean`	`connected(int p, int q)`

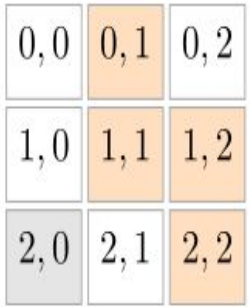
A 3×3 percolation system and its UF representation

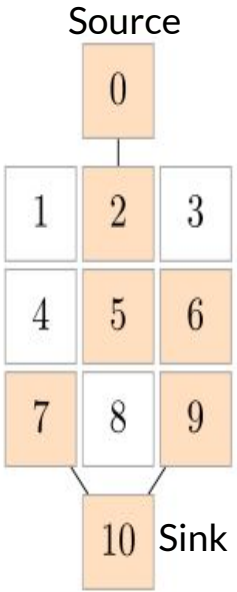
0, 0	0, 1	0, 2
1, 0	1, 1	1, 2
2, 0	2, 1	2, 2



WeightedQuickUF Continued

In the 3×3 system, consider opening the sites $(0,1)$, $(1,2)$,$(1,1)$, $(2,0)$, and $(2,2)$, and in that order; the system percolates once $(2,2)$ is opened.





Constructor

```
// Constructs an n x n percolation system, with all sites blocked.
public UFPercolation(int n) {
    //...
}
```

Initialize instance variables here!

Whats the Same?

Not much changes with these, keep the same thought process as in problem 1

```java
// Returns the number of open sites.
public int numberOfOpenSites() {
    //...
    return 0;
}
```

```java
// Returns true if site (i, j) is open, and false otherwise.
public boolean isOpen(int i, int j) {
    //...
    return false;
}
```

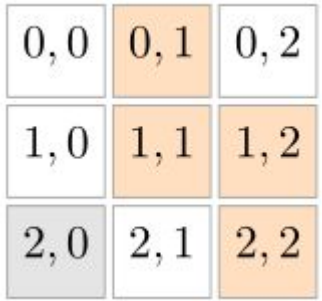
```java
// Opens site (i, j) if it is not already open.
public void open(int i, int j) {
    //...
}
```

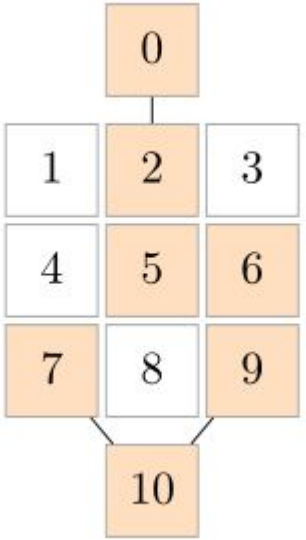
Apply what you did in the first problem for the method of the same name.

- Open the site in the 2D array,
- Open the site in the Weighted UF Structure, you will need to encode here to convert from (i,j) to UF ID
- Combine the Neighbor Logic from Problem 1 to determine what sites to union with one another

– If site `(i, j)` is not open:
  * Open the site
  * Increment `openSites` by one.
  * If the site is in the first (or last) row, connect the corresponding `uf` site with the source (or sink).
  * If any of the neighbors to the north, east, west, and south of site `(i, j)` is open, connect the `uf` site corresponding to site `(i, j)` with the `uf` site corresponding to that neighbor.

Encode(i, j)





- Look at the relationship
- Is there a Formula?
- Encode(int i, in j) takes in a xy pair, and returns the int UF ID
- Lets talk out the formula

Return (row major order index)

IsFull()

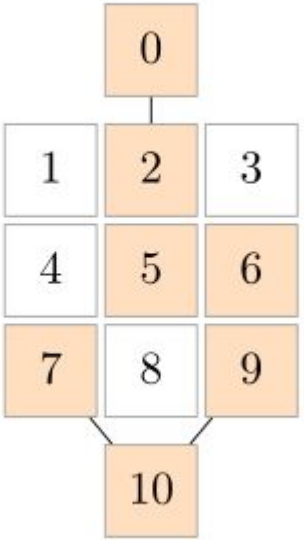
Will Return True IF the given site is Open AND the given site is connected to the source.

Percolates()

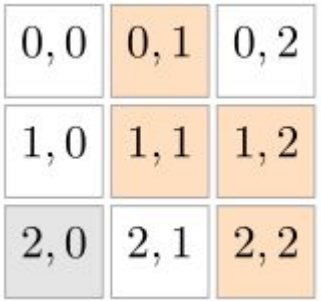
Returns True if the source is connected to the sink

BackWash Problem

Backwash problem, pretend that 9 is not connected. This would still percolate. Why?



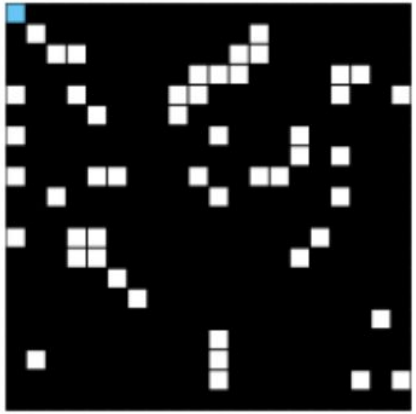
- I will not be explaining how to solve the issue
- Ask the right questions and I can answer



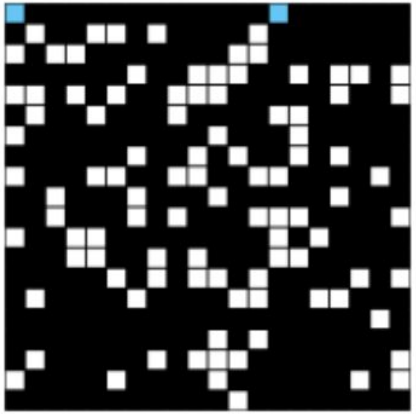
Questions on Problem 2?

Problem 3: Percolation Stats

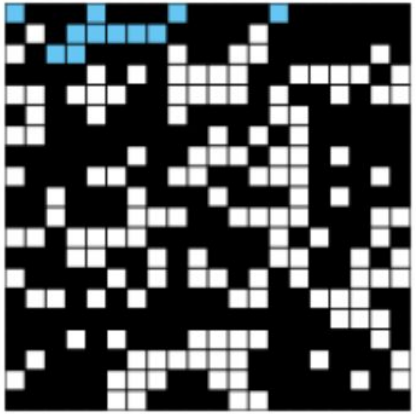
PercolationStats	
`PercolationStats(int n, int m)`	performs `m` independent experiments on an `n x n` percolation system
`double mean()`	returns sample mean of percolation threshold
`double stddev()`	returns sample standard deviation of percolation threshold
`double confidenceLow()`	returns low endpoint of 95% confidence interval
`double confidenceHigh()`	returns high endpoint of 95% confidence interval



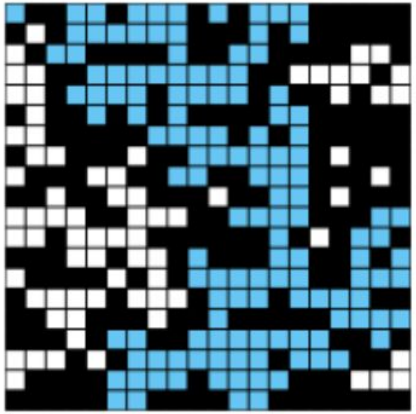
50 open sites



100 open sites



150 open sites



204 open sites

M = 4 systems

By repeating this computational experiment m times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let x1, x2, . . . , xm be the fractions of open sites in computational experiments 1, 2, . . . , m. The sample mean μ provides an estimate of the percolation threshold, and the sample standard deviation σ measures the sharpness of the threshold:

Instance Variables

You are doing m experiments, so you will want to keep track of each experiment's results

- Number of independent experiments, `int m`.
- Percolation thresholds for the `m` experiments, `double[] x`.

Constructor: PercolationStats(int n, int m)

Takes in n size of system, and m number of experiments to conduct

- `PercolationStats(int n, int m)`
  - Initialize instance variables.
  - Perform the following experiment `m` times:
    - Create an $n \times n$ percolation system (use the `UFPercolation` implementation).
    - Until the system percolates, choose a site (i, j) at random and open it if it is not already open.
    - Calculate percolation threshold as the fraction of sites opened, and store the value in `x[]`.

- `double mean()`
  - Return the mean μ of the values in `x[]`.
- `double stddev()`
  - Return the standard deviation σ of the values in `x[]`.
- `double confidenceLow()`
  - Return $\mu - \frac{1.96\sigma}{\sqrt{m}}$.
- `double confidenceHigh()`
  - Return $\mu + \frac{1.96\sigma}{\sqrt{m}}$.

Questions on Program 3?

Testing Tools

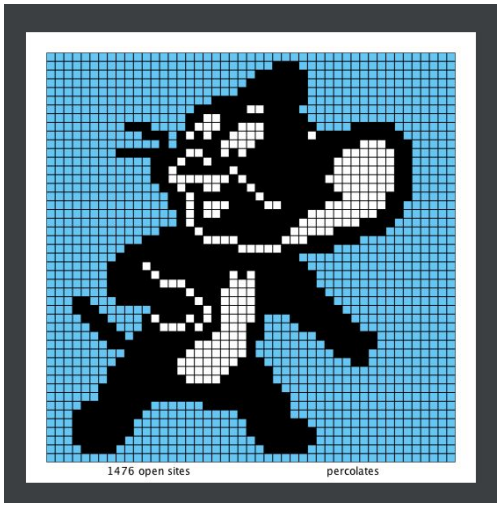
The program PercolationVisualizer accepts mode (the String “array” or “UF”) and filename (String)

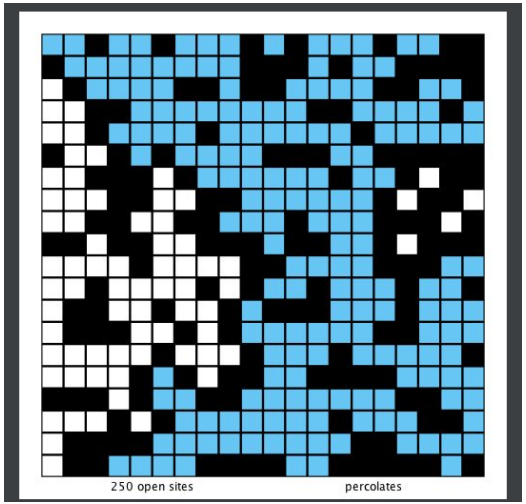
as command-line arguments, and uses ArrayPercolation or UFPercolation to determine and visually report if the system represented

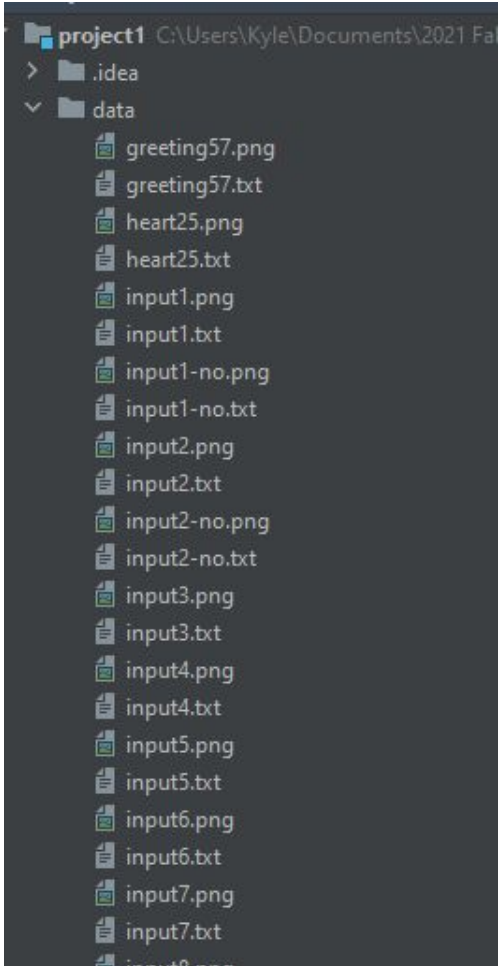
by the input file percolates or not.

```
>_ ~/workspace/project1
$ java PercolationVisualizer UF data/input10.txt
```

Different Testing File Options







Benefits of Using the Visualizer

- Gradescope will tell you what test cases you are failing and a suggested edit, but not whats happening
- The Visualizer will let you see exactly how your programs are going through the system
- Use it for Array AND UF for testing
    - Especially helpful for seeing if your issues are backwash or otherwise

Remember Corner Cases, Check_Style, Comments, Exceptions, and Runtime

The Remainder of Class will be for Questions and Essentially Office Hours. I request you stay the duration of the class and raise your hand and I will come over or you can come up to me.

All Questions are Good Questions, so Ask Ask Ask!