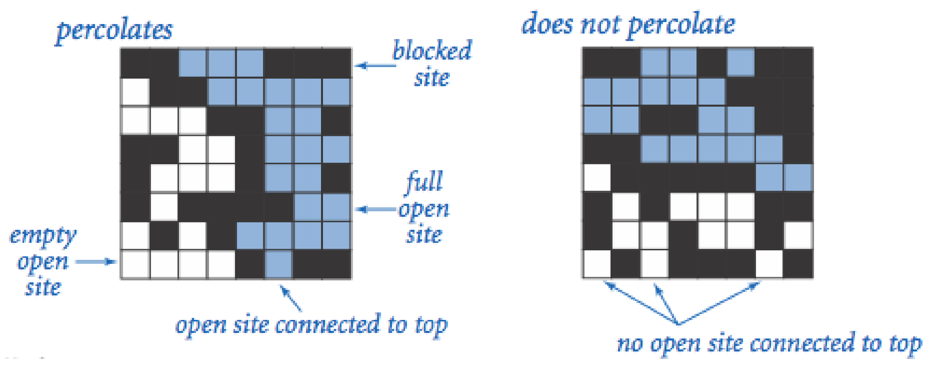
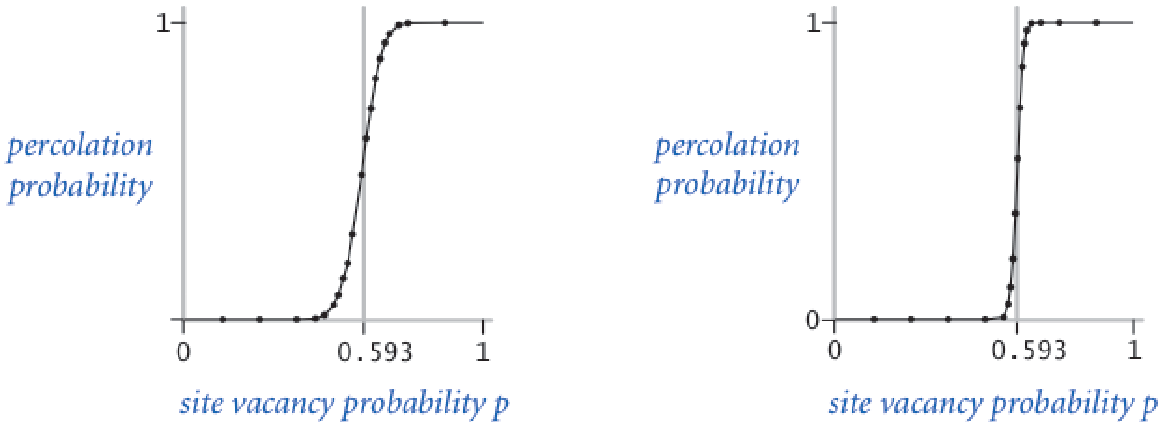
Goal Write a program to estimate the percolation threshold of a system.

Percolation Given a composite system comprising of randomly distributed insulating and metallic materials: what fraction of the system needs to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as *percolation* to model such situations.

The Model We model a percolation system using an nxn grid of sites. Each site is either open or blocked. A full site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. We say the system percolates if there is a full site in the bottom row. In other words, a system percolates if we fill all open sites connected to the top row and that process fills some open site on the bottom row. For the insulating/metallic materials example, the open sites correspond to metallic materials, so that a system that percolates has a metallic path from top to bottom, with full sites conducting. For the porous substance example, the open sites correspond to empty space through which water might flow, so that a system that percolates lets water fill open sites, flowing from top to bottom.



The Problem If sites are independently set to be open with probability p (and therefore blocked with probability $1 - p$), what is the probability that the system percolates? When p equals 0, the system does not percolate; when p equals 1, the system percolates. The plots below show the site vacancy probability p versus the percolation probability for 20×20 random grid (left) and 100×100 random grid (right).



When n is sufficiently large, there is a threshold value $p^\star$ such that when $p < p^\star$ a random $n \times n$ grid almost never percolates, and when $p > p^\star$, a random $n \times n$ grid almost always percolates. No mathematical solution for determining the percolation threshold $p^\star$ has yet been derived. Your task is to write a computer program to estimate $p^\star$.

Percolation API To model a percolation system, we define an interface called `Percolation`, supporting the following API:

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

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

Corner cases:

- `ArrayPercolation()` should throw an `IllegalArgumentException("Illegal n")` if $n \leq 0$.
- `open()`, `isOpen()`, and `isFull()` should throw an `IndexOutOfBoundsException("Illegal i or j")` if i or j is outside the interval $[0, n-1]$.

Performance requirements:

- `open()`, `isOpen()`, and `numberOfOpenSites()` should run in time $T(n) \sim 1$.
- `percolates()` should run in time $T(n) \sim n$.
- `ArrayPercolation()` and `isFull()` should run in time $T(n) \sim n^2$.

```
>_ ~/workspace/project1
$ java ArrayPercolation data/input10.txt
10 x 10 system:
  Open sites = 56
  Percolates = true
$ java ArrayPercolation data/input10-no.txt
10 x 10 system:
  Open sites = 55
  Percolates = false
```

Directions:

- 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`.
- `private void floodFill(boolean[][] full, int i, int j)`
  - Return if `i` or `j` is out of bounds; or site `(i, j)` is not open; or site `(i, j)` is full (ie, `full[i][j]` is `true`).
  - Fill site `(i, j)`.
  - Call `floodFill()` recursively on the sites to the north, east, west, and south of site `(i, j)`.
- `public ArrayPercolation(int n)`
  - Initialize instance variables.
- `void open(int i, int j)`
  - If site `(i, j)` is not open:
    * Open the site.

    * Increment `openSites` by one.

- `boolean isOpen(int i, int j)`
  - Return whether site `(i, j)` is open or not.
- `boolean isFull(int i, int j)`
  - Create an $n \times n$ array of booleans called `full`.
  - Call `floodFill()` on every site in the first row of the percolation system, passing `full` as the first argument.
  - Return `full[i][j]`.
- `int numberOfOpenSites()`
  - Return the number of open sites.
- `boolean percolates()`
  - Return whether the system percolates or not — a system percolates if the last row contains at least one full site.

Problem 2. (*Union Find Percolation*) Develop a data type called `UFPercolation` that implements the `Percolation` interface using a `WeightedQuickUnionUF` object as the underlying data structure.

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

Corner cases:

- `UFPercolation()` should throw an `IllegalArgumentException("Illegal n")` if $n \leq 0$.
- `open()`, `isOpen()`, and `isFull()` should throw an `IndexOutOfBoundsException("Illegal i or j")` if i or j is outside the interval $[0, n-1]$.

Performance requirements:

- `isOpen()` and `numberOfOpenSites()` should run in time $T(n) \sim 1$.
- `open()`, `isFull()`, and `percolates()` should run in time $T(n) \sim \log n$.
- `UFPercolation()` should run in time $T(n) \sim n^2$.

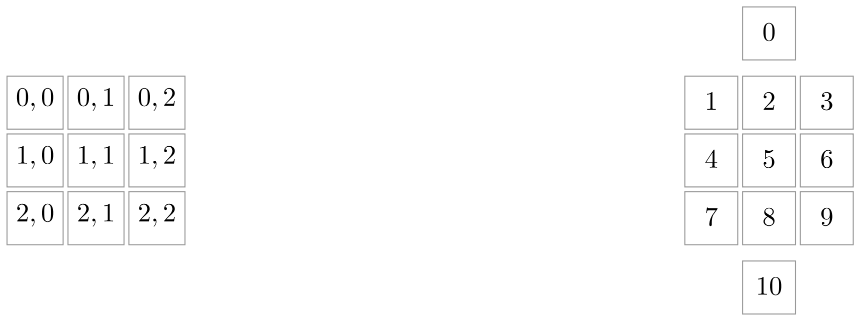
```
>_ ~/workspace/project1
```

```
$ java UFPercolation data/input10.txt
10 x 10 system:
  Open sites = 56
  Percolates = true
$ java UFPercolation data/input10-no.txt
10 x 10 system:
  Open sites = 55
  Percolates = false
```

Directions:

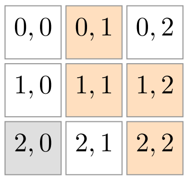
- Model percolation system as an $n \times n$ array of booleans (`true` $\implies$ open site and `false` $\implies$ blocked site).
- Create an `UF` object with $n^2 + 2$ sites and use the private `encode()` method to translate sites $(0, 0), (0, 1), \ldots, (n-1, n-1)$ of the array to sites $1, 2, \ldots, n^2$ of the `UF` object; sites 0 (source) and $n^2 + 1$ (sink) are virtual, ie, not part of the percolation system.

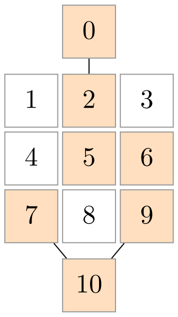
- A 3×3 percolation system and its `UF` representation



- 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`.
- `private int encode(int i, int j)`
  - Return the `UF` site $(1, 2, \ldots, n^2)$ corresponding to the percolation system site `(i, j)`.
- `public UFPercolation(int n)`
  - Initialize instance variables.
- `void open(int i, int j)`
  - 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.
- `boolean isOpen(int i, int j)`
  - Return whether site `(i, j)` is open or not.
- `boolean isFull(int i, int j)`
  - Return whether site `(i, j)` is full or not — a site is full if it is open and its corresponding `uf` site is connected to the source.
- `int numberOfOpenSites()`
  - Return the number of open sites.
- `boolean percolates()`
  - Return whether the system percolates or not — a system percolates if the sink is connected to the source.
- Using virtual source and sink sites introduces what is called the *back wash* problem.

- 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.



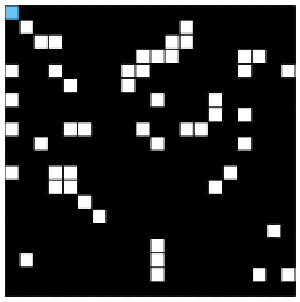


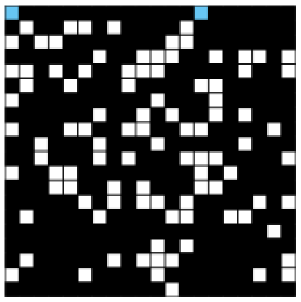
- The site $(2,0)$ is technically not full since it is not connected to an open site in the top row via a path of neighboring (north, east, west, and south) open sites, but the corresponding `uf` site (7) is connected to the source, so is incorrectly reported as being full — this is the back wash problem.
- To receive full credit, you must resolve the back wash problem.

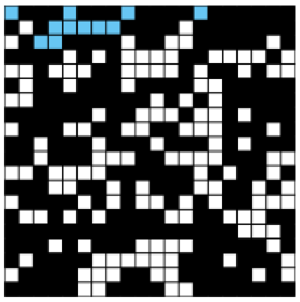
Problem 3. (*Estimation of Percolation Threshold*) To estimate the percolation threshold, consider the following computational (Monte Carlo simulation) experiment:

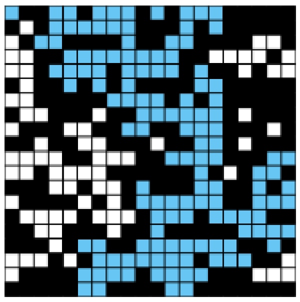
- Create an $n \times n$ percolation system (use the `UFPercolation` implementation) with all sites blocked.
- Repeat the following until the system percolates:
  - Choose a site (row i, column j) uniformly at random among all blocked sites.
  - Open the site (row i, column j).
- The fraction of sites that are open when the system percolates provides an estimate of the percolation threshold.

For example, if sites are opened in a 20×20 grid according to the snapshots below, then our estimate of the percolation threshold is $204/400 = 0.51$ because the system percolates when the 204th site is opened.


50 open sites


100 open sites


150 open sites


204 open sites

By repeating this computational experiment m times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let $x_1, x_2, \ldots, x_m$ be the fractions of open sites in computational experiments $1, 2, \ldots, m$. The sample mean μ provides an estimate of the percolation threshold, and the sample standard deviation σ measures the sharpness of the threshold:

$$\mu = \frac{x_1 + x_2 + \cdots + x_m}{m}, \quad \sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_m - \mu)^2}{m-1}.$$

Assuming m is sufficiently large (say, at least 30), the following interval provides a 95% confidence interval for the percolation threshold:

$$\left[\mu - \frac{1.96\sigma}{\sqrt{m}}, \mu + \frac{1.96\sigma}{\sqrt{m}}\right].$$

To perform a series of computational experiments, create an immutable data type called `PercolationStats` that supports the following API:

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

The constructor perform m independent computational experiments (discussed above) on an $n \times n$ grid. Using this experimental data, it should calculate the mean, standard deviation, and the 95% confidence interval for the percolation threshold.

Corner cases:

- The constructor should throw an `IllegalArgumentException("Illegal n or m")` if either $n \leq 0$ or $m \leq 0$.

Performance requirements:

- `mean()`, `stddev()`, `confidenceLow()`, and `confidenceHigh()` should run in time $T(n, m) \sim m$.
- `PercolationStats()` should run in time $T(n, m) \sim mn^2$.

```
>_ ~/workspace/project1
$ java PercolationStats 100 1000
Percolation threshold for a 100 x 100 system:
  Mean               = 0.592
  Standard deviation  = 0.016
  Confidence interval = [0.591, 0.594]
```

Directions:

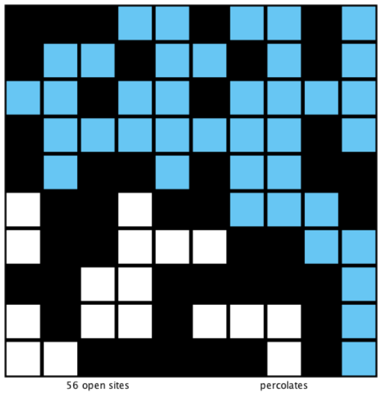
- Instance variables:
  - Number of independent experiments, `int m`.
  - Percolation thresholds for the `m` experiments, `double[] x`.
- `PercolationStats(int n, int m)`
  - Initialize instance variables.
  - Perform the following experiment `m` times:
    * Create an $n \times n$ percolation system (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 $\mu + \frac{1.96\sigma}{\sqrt{m}}$.

Data The `data` directory contains some input `.txt` files for the percolation visualization programs, and associated with each file is an output `.png` file that shows the desired output. For example

```
>_ ~/workspace/project1
$ cat data/input10.txt
10
9 1
1 9
...
7 9
```

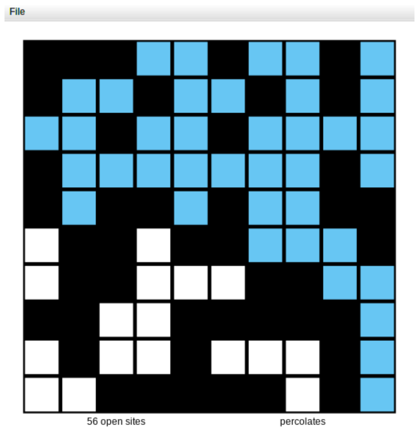
The first number specifies the size of the percolation system and the pairs of numbers that follow specify the sites to open.

```
>_ ~/workspace/project1
$ display data/input10.png
```



Visualization Programs 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.

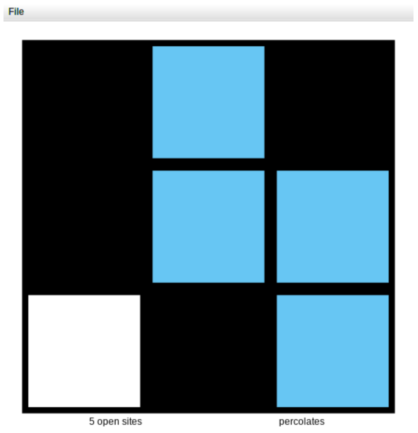
```
>_ ~/workspace/project1
$ java PercolationVisualizer UF data/input10.txt
```



The program `InteractivePercolationVisualizer` accepts *mode* ("array" or "UF") and n (int) as command-line arguments, constructs an $n \times n$ percolation system using `ArrayPercolation` or `UFPercolation`, and allows you to interactively open sites in the system by clicking on them and visually inspect if the system percolates or not.

```
>_ ~/workspace/project1
$ java InteractivePercolationVisualizer UF 3
3
0 1
```

```
1 2
1 1
2 0
2 2
```



Acknowledgements This project is an adaptation of the Percolation assignment developed at Princeton University by Robert Sedgewick and Kevin Wayne.

Files to Submit

1. `ArrayPercolation.java`
2. `UFPercolation.java`
3. `PercolationStats.java`
4. `notes.txt`

Before you submit your files, make sure:

- You do not use concepts outside of what has been taught in class.
- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You edit the sections (`#1` mandatory, `#2` if applicable, and `#3` optional) in the given `notes.txt` file as appropriate. Section `#1` must provide a clear high-level description of the project in no more than 200 words.