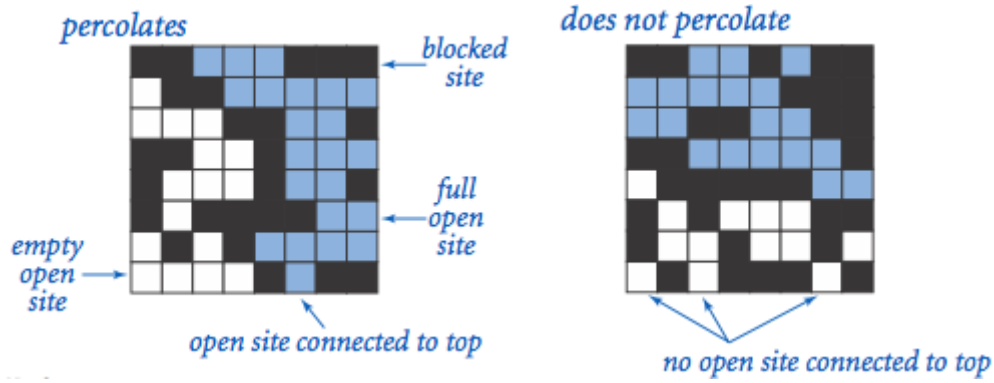
Programming Assignment 1: Percolation

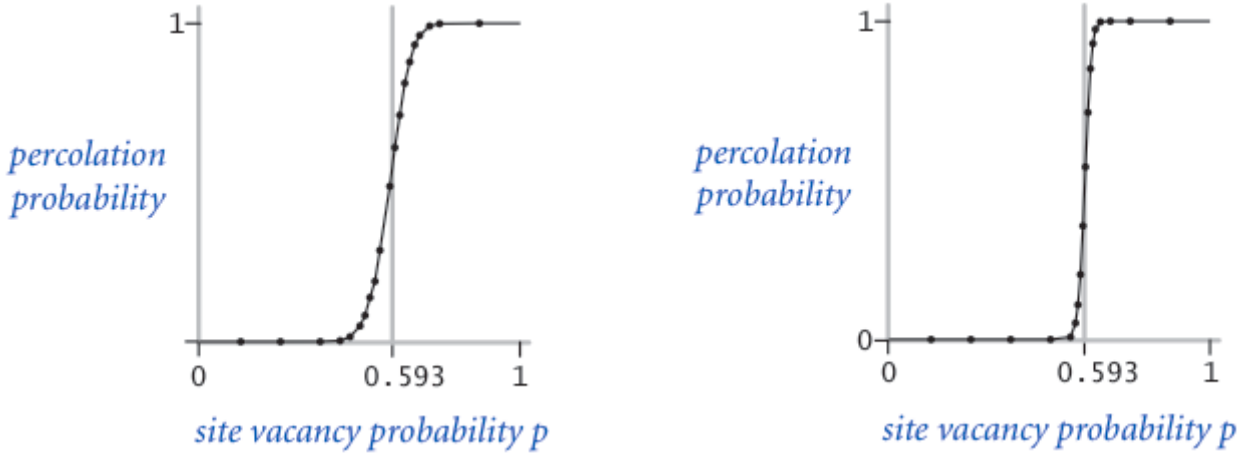
Write a program to estimate the value of the *percolation threshold* via Monte Carlo simulation.

Percolation. Given a composite systems comprised of randomly distributed insulating and metallic materials: what fraction of the materials need 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 N-by-N 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. In a famous scientific problem, researchers are interested in the following question: 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-by-20 random grid (left) and 100-by-100 random grid (right).



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

Percolation data type. To model a percolation system, create a data type `Percolation` with the following API:

```
public class Percolation {
   public Percolation(int N)                // create N-by-N grid, with all
                                                 sites initially blocked
   public void open(int row, int col)       // open the site (row, col) if
                                                 it is not open already
   public boolean isOpen(int row, int col)  // is the site (row, col) open?
   public boolean isFull(int row, int col)  // is the site (row, col) full?
   public int numberOfOpenSites()           // number of open sites
   public boolean percolates()              // does the system percolate?
   public static void main(String[] args)   // unit testing (required)
}
```

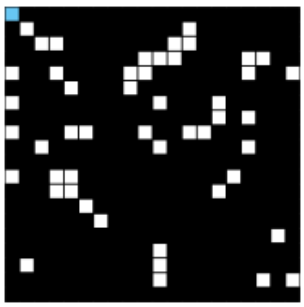
Corner cases. By convention, the row and column indices are integers between 0 and $N - 1$, where (0, 0) is the upper-left site: Throw a `java.lang.IndexOutOfBoundsException` if any argument to `open()`, `isOpen()`, or `isFull()` is outside its prescribed range. The constructor should throw a `java.lang.IllegalArgumentException` if $N \leq 0$.

Performance requirements. The constructor should take time proportional to N^2; all methods should take constant time plus a constant number of calls to the union-find methods `union()`, `find()`, `connected()`, and `count()`.

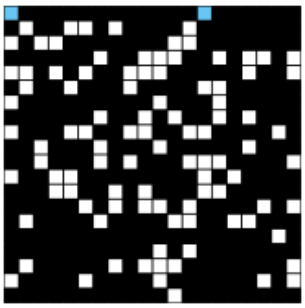
Monte Carlo simulation. To estimate the percolation threshold, consider the following computational experiment:

- Initialize all sites to be blocked.
- Repeat the following until the system percolates:
  - Choose a site uniformly at random among all blocked sites.
  - Open the site.
- The fraction of sites that are opened when the system percolates provides an estimate of the percolation threshold.

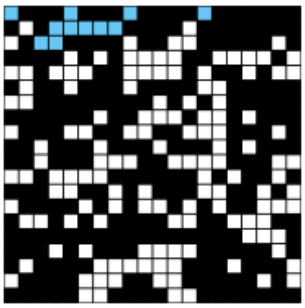
For example, if sites are opened in a 20-by-20 grid according to the snapshots below, then our estimate of the percolation threshold is 204/400 = 0.51 because the system percolates when the 204th site is opened.



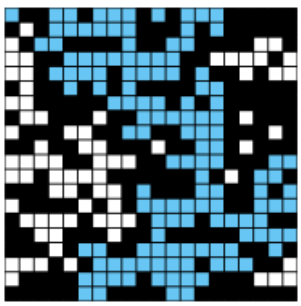
50 open sites



100 open sites



150 open sites



204 open sites

By repeating this computation experiment T times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let x_t be the fraction of open sites in computational experiment t. The sample mean μ provides an estimate of the percolation threshold; the sample standard deviation σ measures the sharpness of the threshold.

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

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

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

To perform a series of computational experiments, create a data type `PercolationStats` with the following API.

```
public class PercolationStats {
   public PercolationStats(int N, int T)    // perform T independent
experiments on an N-by-N grid
   public double mean()                     // sample mean of percolation
threshold
   public double stddev()                   // sample standard deviation of
percolation threshold
   public double confidenceLow()            // low  endpoint of 95%
confidence interval
   public double confidenceHigh()           // high endpoint of 95%
confidence interval
}
```

The constructor should throw a `java.lang.IllegalArgumentException` if either $N \leq 0$ or $T \leq 0$.

The constructor should take two arguments *N* and *T*, and perform *T* independent computational experiments (discussed above) on an *N*-by-*N* grid. Using this experimental data, it should calculate the mean, standard deviation, and the *95% confidence interval* for the percolation threshold. Use `StdRandom` to generate random numbers; use `StdStats` to compute the sample mean and standard deviation.

```
Example values after creating PercolationStats(200, 100)
mean()                  = 0.5929934999999997
stddev()                = 0.00876990421552567
confidenceLow()         = 0.5912745987737567
confidenceHigh()        = 0.5947124012262428

Example values after creating PercolationStats(200, 100)
mean()                  = 0.592877
stddev()                = 0.009990523717073799
confidenceLow()         = 0.5909188573514536
confidenceHigh()        = 0.5948351426485464


Example values after creating PercolationStats(2, 100000)
mean()                  = 0.6669475
stddev()                = 0.11775205263262094
confidenceLow()         = 0.666217665216461
confidenceHigh()        = 0.6676773347835391
```

Analysis of running time.

- Implement the `Percolation` data type using the *quick-find* algorithm in `QuickFindUF`. Use `Stopwatch` to measure the total running time of `PercolationStats` for various values of *N* and *T*. How does doubling *N* affect the total running time? How does doubling *T* affect the total running time? Give a formula (using tilde notation) of the total running time on your computer (in seconds) as a single function of both *N* and *T*.
- Now, implement the `Percolation` data type using the *weighted quick-union* algorithm in `WeightedQuickUnionUF`. Answer the same questions in the previous bullet.

Describe the experiments following the template and questions in the attached readme.txt file. You may use another format, but be sure to address all issues raised there, including some in comments.

Deliverables. Submit: A) `Percolation.java` (using the weighted quick-union algorithm from `WeightedQuickUnionUF`) and B) `PercolationStats.java` on Mooshak. Your submission may not call

library functions except those in `StdIn`, `StdOut`, `StdRandom`, `StdStats`, and `java.lang`. Also, submit on Canvas both programs, along with the readme file answering all questions there.