In this assignment, you will write a program to estimate the value of the *percolation threshold* via Monte Carlo simulation[1]. The learning outcomes of this assignment are:

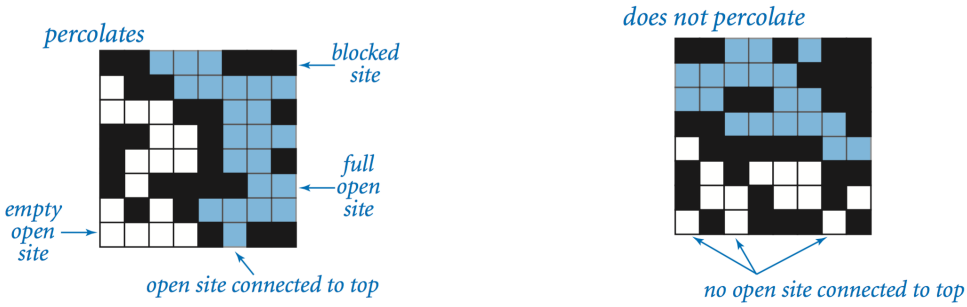
- Apply the union-find data structure to a scientific application.
- See the enormous impact of an efficient algorithm on real-world performance.

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.

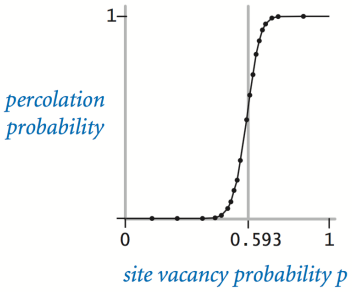
We model a percolation system using an n-by-n grid of *sites*. Each site is either *open* or *blocked*. A *full* site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. We say the system *percolates* if there is a full site 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.)

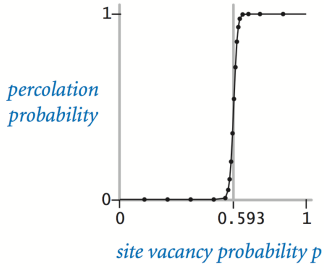


The problem.

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 random 20-by-20 grids (left) and 100-by-100 grids (right).

[1] *Adapted from* `https://introcs.cs.princeton.edu/java/assignments/percolation.html`





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:

```
class Percolation {
  public:
    // creates n-by-n grid, with all sites initially blocked
    Percolation(int n);

    // opens the site (row, col) if it is not open already
    void open(int row, int col);

    // is the site (row, col) open?
    bool isOpen(int row, int col);

    // is the site (row, col) full?
    bool isFull(int row, int col);

    // returns the number of open sites
    int numberOfOpenSites();

    // does the system percolate?
    bool percolates();

    // unit testing (required)
    static void test();
};
```

Corner cases. By convention, the row and column indices are integers between 0 and $n - 1$, where $(0, 0)$ is the upper-left site.

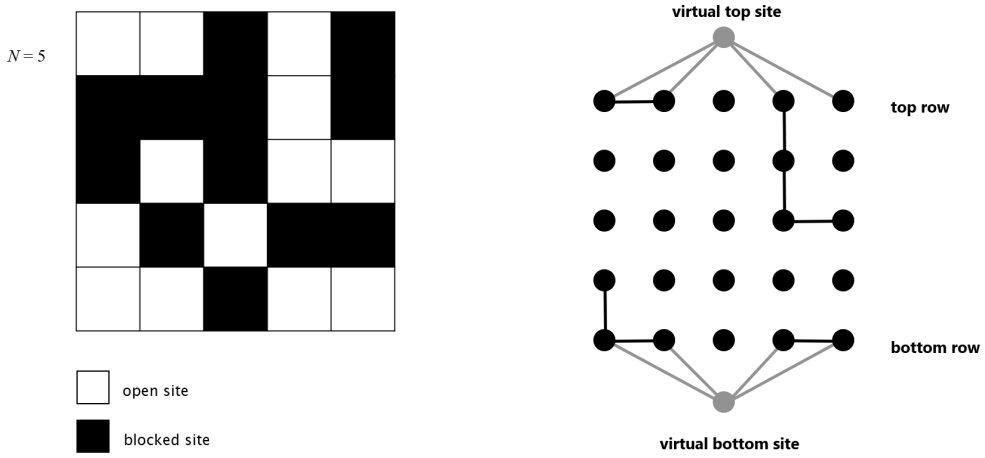
- Throw an `std::invalid_argument` if any argument to `open()`, `isOpen()`, or `isFull()` is outside its prescribed range.

- Throw an `std::invalid_argument` in the constructor if $n \leq 0$.

Unit testing. Your `test()` method must call each public constructor and method directly and help verify that they work as prescribed (e.g., by printing results to standard output).
Performance requirements. The constructor must take $\Theta(n^2)$ time; all instance methods must take $\Theta(1)$ time plus $\Theta(1)$ calls to `union()` and `find()`.

Percolation with Virtual Sites: An Efficient Approach

Determining whether a system percolates can be simplified by introducing two virtual sites: a virtual top site and a virtual bottom site. The virtual top site is connected to every open site in the top row, and the virtual bottom site is connected to every open site in the bottom row. This allows for a more efficient algorithm for the `percolates()` method. Instead of iterating through all open sites in the bottom row to see if they are full, the system percolates if and only if the virtual top site is connected to the virtual bottom site. This check requires only a single call to the `connected()` function, greatly improving performance.

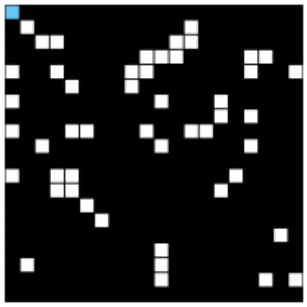


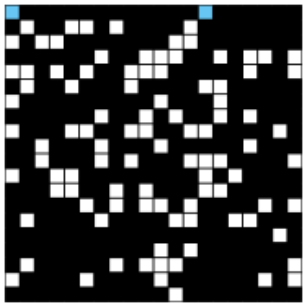
Monte Carlo simulation

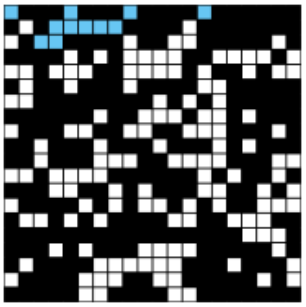
To estimate the percolation threshold, consider the following computational experiment:

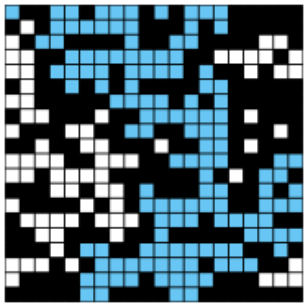
- Initialize all sites to be blocked.
- Repeat the following until the system percolates:
  - Choose a site uniformly at random among all blocked sites.
  - Open the site.
- The fraction of sites that are opened when the system percolates provides an estimate of the percolation threshold.

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 $\overline{x}$ provides an estimate of the percolation threshold; the sample standard deviation s measures the sharpness of the threshold.

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

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

$$\left[\overline{x} - \frac{1.96s}{\sqrt{T}}, \overline{x} + \frac{1.96s}{\sqrt{T}}\right]$$

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

```
  class PercolationStats {
  public:
  // perform independent trials on an n-by-n grid
   PercolationStats(int n, int trials);

  // sample mean of percolation threshold
  double mean();

  // sample standard deviation of percolation threshold
  double stddev();

  // low endpoint of 95% confidence interval
  double confidenceLow();

  // high endpoint of 95% confidence interval
  double confidenceHigh();

 // test client (see below)
  static void test(s);
};
```

The constructor takes two arguments n and T, and perform T independent computational experiments (discussed above) on an n-by-n grid. Using this experimental data, it calculates the mean, standard deviation, and the 95% confidence interval for the percolation threshold.

Standard libraries. Use `<random>` to generate random numbers; use *Stopwatch.hpp* to measure the running time.
Corner cases. Throw an `std::invalid_argument` in the constructor if either $n \leq 0$ or $T \leq 0$.
Test client. The test client takes two command-line arguments n and T and prints the relevant statistics for T computational experiments on an n-by-n grid.

```
> PercolationStats 200 100
mean()           = 0.592993
stddev()         = 0.008770
confidenceLow()  = 0.591275
confidenceHigh() = 0.594712
elapsed time     = 0.373

> PercolationStats 200 100
mean()           = 0.592877
stddev()         = 0.009991
confidenceLow()  = 0.590919
confidenceHigh() = 0.594835
elapsed time     = 0.364

> PercolationStats 2 100000
mean()           = 0.666948
stddev()         = 0.117752
confidenceLow()  = 0.666218
confidenceHigh() = 0.667677
elapsed time     = 0.087
```

Comparison of two data structures

Implement Percolation using the quick-find algorithm, then using the weighted quick-union algorithm. Compare the performance of the two algorithms.

- First, implement Percolation using *quick-find*. What is the largest values of n that `PercolationStats` can handle in less than one minute on your computer when performing $T = 100$ trials?
- Next, implement Percolation using *weighted quick-union*. What is the largest values of n that `PercolationStats` can handle in less than one minute on your computer when performing $T = 100$ trials?

Submission.

- `Percolation.hpp` with the Weighted Quick-Union implementation

- `PercolationStat.hpp`
- `main.cpp` which will run and test the program
- `Comparison.txt` which will include your results and comparison between the two Union Find implementations.

Grading rubric.

file	*marks*
`Percolation.hpp`	22
`PercolationStats.hpp`	12
`Comparison.txt`	6
	40