Programming Assignment 1 Checklist: Percolation

Frequently Asked Questions (Percolation)

What are the goals of this assignment?

- Learn about a scientific application of the union–find data structure.
- Measure the running time of a program and use the doubling hypothesis to make predictions.

Can I add (or remove) methods to (or from) Percolation or PercolationStats? No. You must implement the APIs exactly as specified, with the identical set of public methods and signatures (or you will receive a substantial deduction). However, you are encouraged to add private methods that enhance the readability, maintainability, and modularity of your program.

What is unit testing? In your `main()` you are expected to show any testing you did of your code. At a minimum this includes calling all constructors and methods in the class. Any testing you did to debug your program should also be shown here. The autograder will not call it but you will fail the API check if it is not there.

Why is it so important to implement the prescribed API? Writing to an API is an important skill to master because it is an essential component of modular programming, whether you are developing software by yourself or as part of a group. When you develop a module that properly implements an API, anyone using that module (including yourself, perhaps at some later time) does not need to revisit the details of the code for that module when using it. This approach greatly simplifies writing large programs, developing software as part of a group, or developing software for use by others.

Most important, when you properly implement an API, others can write software to use your module or to test it. We do this regularly when grading your programs. For example, your `PercolationStats` client should work with our `Percolation` data type and vice versa. If you add an extra public method to `Percolation` and call them from `PercolationStats`, then your client won't work with our `Percolation` data type. Conversely, our `PercolationStats` client may not work with your `Percolation` data type if you remove a public method.

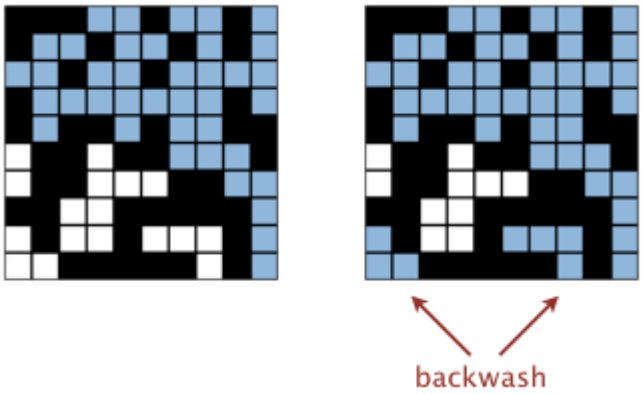
How many lines of code should my program be? You should strive for clarity and efficiency. Our reference solution for `Percolation.java` is about 80 lines, plus a test client. Our `PercolationStats.java` client is about 60 lines. If you are re-implementing the union-find data structure (instead of reusing the implementations provided), you are on the wrong track.

What should `stddev()` return if *T* equals 1? The sample standard deviation is undefined. We recommend returning `Double.NaN` but we will not test this case.

After the system has percolated, PercolationVisualizer colors in light blue all sites connected to open sites on the bottom (in addition to those connected to open sites on the

top). Is this "backwash" acceptable? While allowing backwash does not strictly conform to the `Percolation` API, it requires quite a bit of ingenuity to fix and it leads to only a one-half deduction in the grade if you don't. To put this in context, there is a one whole penalty per late day. So, treat this as a bonus challenge (if you have extra time available).

```
% java PercolationVisualizer input10.txt
```



How do I generate a site uniformly at random among all blocked sites for use in PercolationStats? Pick a site at random (by using `StdRandom` to generate two integers between 0 (inclusive) and *N* (exclusive) and use this site if it is blocked; if not, repeat.

I don't get reliable timing information in PercolationStats when *N* = 200. What should I do? Increase the size of *N* (say to 400, 800, and 1600), until the mean running time exceeds its standard deviation.

Testing

Testing. We provide two clients that serve as large-scale visual traces. We highly recommend using them for testing and debugging your `Percolation` implementation.

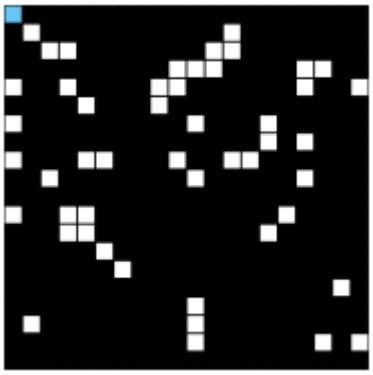
Visualization client. PercolationVisualizer.java animates the results of opening sites in a percolation system specified by a file by performing the following steps:

- Read the grid size *N* from the file.
- Create an *N*-by-*N* grid of sites (initially all blocked).
- Read in a sequence of sites (row *i*, column *j*) to open from the file. After each site is opened, draw full sites in light blue, open sites (that aren't full) in white, and blocked sites in black using *standard draw*, with site (0, 0) in the upper left-hand corner.

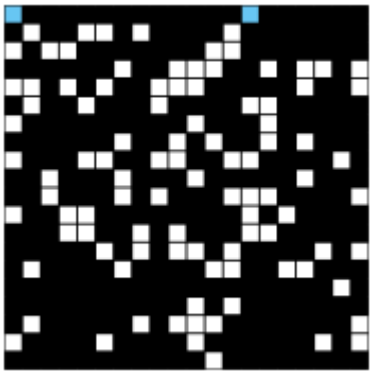
The program should behave as in this movie and the following snapshots when used with input20.txt.

```
% more input20.txt
20
 6 10
17 10
11  4
 8  4
```

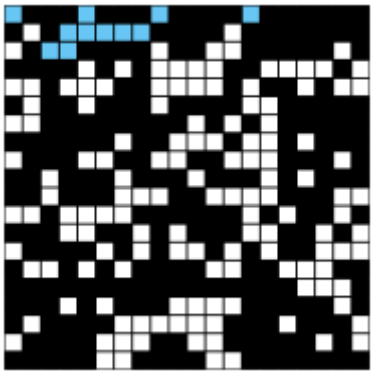
```
 4   8
 0   0
...

% java PercolationVisualizer input20.txt
```



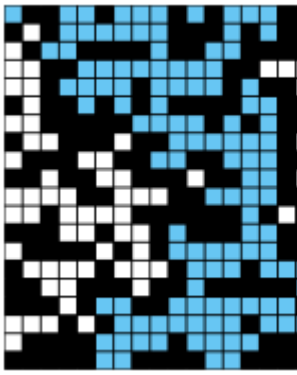
50 open sites



100 open sites



150 open sites



204 open sites

Sample data files. We supplied a set of sample files (in data folder) for use with the visualization client. More can be found in the Princeton directory percolation. Associated with most input `.txt` file is an output `.png` file that contains the desired graphical output at the end of the visualization.

InteractiveVisualization client. InteractivePercolationVisualizer.java is similar to the first test client except that the input comes from a mouse (instead of from a file). It takes a command-line integer *N* that specifies the grid size. As a bonus, it writes to standard output the sequence of sites opened in the same format used by `PercolationVisualizer`, so you can use it to prepare interesting files for testing. If you design an interesting data file, feel free to share it with us and your classmates by posting it in the discussion forums.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. **Install a Java programming environment.**
2. **Consider not worrying about backwash for your first attempt.** You can revise your implementation once you have solved the problem without handling backwash.
3. **For each method in `Percolation` that you must implement (`open()`, `percolates()`, etc.), make a list of which methods (`QuickFindUF` or `WeightedQuickUnionUF`) might be useful for implementing that method.** This should help solidify what you're attempting to accomplish. Do not write your own union-find data structure or algorithms.
4. **Using the list of methods from the API as a guide, choose instance variables that you'll need to solve the problem.** Don't overthink this, you can always change them later. Instead, use your list of instance variables to guide your thinking as you follow the steps below, and make changes to your instance variables as you go.

5. **Plan how you're going to map from a 2-dimensional (row, column) pair to a 1-dimensional union find object index.** We recommend writing a private method to convert from 2D coordinates to 1D coordinates (instead of copying and pasting the conversion formula multiples times).
6. **Write a private method for validating indices.** Since each method is supposed to throw an exception for invalid indices, you should write a private method which performs this validation process.
7. **Write the `open()` method and the `Percolation()` constructor.** The `open()` method should do three things. First, it should validate the indices of the site that it receives. Second, it should somehow mark the site as open and increment the number of open sites by one (if the site is not already open). Third, it should perform some sequence of union-find operations that links the site in question to its open neighbors. The constructor and instance variables should facilitate the `open()` method's ability to do its job.
8. **Test the `open()` method and the `Percolation()` constructor.** These tests should be in `main()`. An example of a simple test is to create a percolation object with `N = 2`, then call `open(0, 1)` and `open(1, 1)`, and finally to call `percolates()`.
9. **Write the `percolates()`, `isOpen()`, and `isFull()` methods.** These should be very simple methods. Each of the methods (except the constructor) in `Percolation` must use a constant number of union-find operations. If you have a for loop inside of one of your `Percolation` methods, you're probably doing it wrong. Don't forget about the virtual-top / virtual-bottom trick described in lecture.
10. **Test your complete implementation using the two visualization clients.** Then when submitting, your `Percolation` class should use the `WeightedQuickUnionUF` class. If you submit with either `UF` or `QuickFindUF`, your code will fail the timing tests.
11. **Write and test the PercolationStats class.** Use standard random and standard statistics.
12. **Analysis of algorithms.** Review Lecture F3 (from Aug 23), especially the section on estimating running time and the doubling hypothesis. Analyze your program using each of `QuickFindUF` and `WeightedQuickUnionUF` by answering the questions provided in the `readme.txt` file.

A video is provided for those wishing additional assistance. Be forewarned that video was made in early 2014 and is somewhat out of date. For example the API has changed.