Expand All Collapse All

Java

Which version of Java must I use?

Which Java programming environment should I use?

How do I configure *IntelliJ* to access the libraries in `algs4.jar`?

What's the Project link?

I haven't programmed in Java in a while. Which material do I need to remember?

I've never programmed in Java before. Should I continue with the course?

Where can I find the Java source code and documentation for the algorithms, data structures, and I/O libraries from lecture and the textbook?

Submission and Feedback

How can I receive personalized feedback on this assignment?

How do I create a zip file for submission to Coursera?

Can I add (or remove) methods to (or from) the prescribed APIs?

Why is it so important to implement the prescribed API?

Can my public constructor and methods have side effects that are not described in the API?

Which style and bug checkers does the autograder use? How can I configure my system to use them?

Will I receive a deduction if I don't adhere to the course rules for formatting and commenting my code?

Percolation

Can my Percolation data type assume the row and column indices are between 0 and n−1?

How do I throw a `IndexOutOfBoundsException`?

How many lines of code should my program be?

After the system has percolated, my `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?

PercolationStats

What should `stddev()` return if *trials* equals 1?

How do I generate a site uniformly at random among all blocked sites?

How precisely must the output in `main()` match the format given?

I don't get reliable timing information when *n* = 200. What should I do?

TESTING

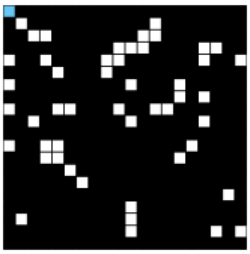
Testing. We provide two clients that serve as large-scale visual traces. We highly recommend using them for testing and debugging your `Percolation` implementation.

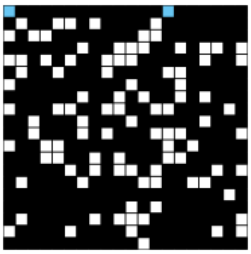
Visualization client. PercolationVisualizer.java animates the results of opening sites in a percolation system specified by a file by performing the following steps:

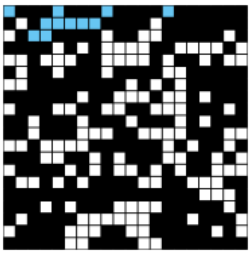
- Read the grid size *n* from the file.
- Create an *n*-by-*n* grid of sites (initially all blocked).
- Read in a sequence of sites (row *i*, column *j*) to open from the file. After each site is opened, draw full sites in light blue, open sites (that aren't full) in white, and blocked sites in black using *standard draw*, with with site (1, 1) in the upper left-hand corner.

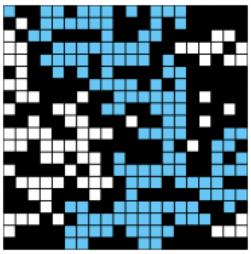
The program should behave as in this movie and the following snapshots when used with input20.txt.

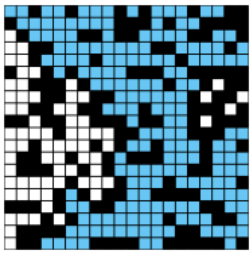
```
% java PercolationVisualizer input20.txt
```


50 open sites


100 open sites


150 open sites


204 open sites


250 open sites

Sample data files. The zip file percolation.zip contains some sample files for use with the visualization client. Associated with each input `.txt` file is an output `.png` file that contains the desired graphical output at the end of the animation.

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 an integer command-line argument *n* that specifies the lattice 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. **Consider not worrying about backwash for your first attempt.** If you're feeling overwhelmed, don't worry about backwash when following the possible progress steps below. You can revise your implementation once you have a better handle on the problem and have solved the problem without handling backwash.
2. **For each method in `Percolation` that you must implement (`open()`, `percolates()`, etc.), make a list of which `WeightedQuickUnionUF` methods might be useful for implementing that method.** This should help solidify what you're attempting to accomplish.
3. **Using the list of methods above 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. Hint: At minimum, you'll need to store the grid size, which sites are open, and which sites are connected to which other sites. The last of these is exactly what the union–find data structure is designed for.
4. **Plan how you're going to map from a 2-dimensional (row, column) pair to a 1-dimensional union find object index.** You will need to come up with a scheme for uniquely mapping 2D coordinates to 1D coordinates. We recommend writing a private method with a signature along the lines of `int xyTo1D(int, int)` that performs this conversion. You will need to utilize the percolation grid size when writing this method. Writing such a private method (instead of copying and pasting a conversion formula multiple times throughout your code) will greatly improve the readability and maintainability of your code. In general, we encourage you to write such modules wherever possible. Directly test this method using the main() function of Percolation.
5. **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.
6. **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. Third, it should perform some sequence of `WeightedQuickUnionUF` 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.
7. **Test the `open()` method and the `Percolation()` constructor.** These tests should be in `main()`. An example of a simple test is to call `open(1, 1)` and `open(1, 2)`, and then to ensure that the two corresponding entries are connected (using two calls to `find()` in `WeightedQuickUnionUF`).
8. **Write the `percolates()`, `isOpen()`, and `isFull()` methods.** These should be very simple methods.
9. **Test your complete implementation using the visualization clients.**
10. **Write and test the PercolationStats class.**

Programming Tricks and Common Pitfalls

1. **Do not write your own union–find data structure. Use `WeightedQuickUnionUF` instead.**
2. **Your `Percolation` class must use `WeightedQuickUnionUF`.** Otherwise, it will fail the timing tests, as the autograder intercepts and counts calls to methods in `WeightedQuickUnionUF`.
3. **It's OK to use an extra row and/or column to deal with the 1-based indexing of the percolation grid.** Though it is slightly inefficient, it's fine to use arrays or union–find objects that are slightly larger than strictly necessary. Doing this results in cleaner code at the cost of slightly greater memory usage.
4. **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.