



## PERCOLATION

Spec

FAQ

Project

Submit

Expand All

Collapse All

## Java

## ▼ Which Java programming environment should I use?

We recommend our customized *IntelliJ* programming environment for [Mac OS X](#) , [Windows](#) , and [Linux](#) . If you followed our step-by-step instructions, then everything should be configured and ready to go.

If you prefer to use another environment (such as Eclipse or Sublime), that's perfectly fine—just be sure that you know how to do the following:

- Add `algs4.jar` to your Java classpath.
- Enter command-line arguments.
- Use standard input and standard output (and, ideally, redirect them to or from a file).

## ▼ Which version of Java must I use?

Any version of Java 8, Java 9, Java 10, or Java 11 should be fine. Our autograder uses Java 8, so, if you use a more recent version, be sure to use only Java 8 features. We have configured the *IntelliJ* projects to enforce this constraint.

## ▼ What's the Project link?

It contains an *IntelliJ* project that you can use to develop your program. It may also contain supplemental data files (which you will use even if you do not use *IntelliJ*).

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

For a review of our Java programming model (including our input and output libraries), read Sections 1.1 and 1.2 of *Algorithms, 4th Edition*.

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

That depends. You will need to write your programs in Java to receive feedback from the autograder. Perhaps you can use this course as an opportunity to learn Java.

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

They are in `algs4.jar`. Here are links to the [source code](#) and [Javadoc](#).

## Submission and Feedback

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

Here are three approaches:

- *Mac OS X*.
  - Select the required files in the *Finder*.
  - Right-click and select *Compress 5 Items*.
  - Rename the resulting file to `percolation.zip`.
- *Windows*.
  - Select the required files in Windows Explorer.
  - Right-click and select *Send to -> Compressed (zipped) folder*.
  - Rename the resulting file to `percolation` (the `.zip` extension is automatic).
- *Command line (Linux or Mac OS X or Windows Git Bash)*.
  - Change to the directory containing the required java files.
  - Execute the command: `zip percolation.zip *.java`.

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

No. You must implement each API exactly as specified, with the identical set of public methods and signatures or your assignment will not be graded. However, you are encouraged to add private methods that enhance the readability, maintainability, and modularity of your program. The one exception is `main()`—you are always permitted to add this method to test your code, but we will not call it unless we specify it in our API.

## ▼ 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.

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

The autograder uses the following tools:

- [Checkstyle 9.21](#) checks the style of your Java programs. Here is a list of available [Checkstyle checks](#) and the configuration file [checkstyle-coursera.xml](#) that the autograder uses.
- [SpotBugs 4.1.3](#) identifies common bug patterns in your code. Here is a summary of [SpotBugs bug descriptions](#) and the configuration file [spotbugs.xml](#) that the autograder uses.

Here is some guidance for installing on your system.

- *IntelliJ*. Our custom *IntelliJ* environment is preconfigured to use *Checkstyle* and *SpotBugs*.
- *Command line*. Our custom *IntelliJ* environment installs command-line versions of these tools.
- *Eclipse*. there is a [Checkstyle plugin for Eclipse](#) and a [SpotBugs plugin for Eclipse](#).

Note that *Checkstyle* inspects the source code; *SpotBugs* inspects the compiled code.

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

The autograder (and *IntelliJ*) provide style feedback to help you become a better programmer. The autograder, however, does not typically deduct for stylistic flaws.

## Percolation

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

No. The API specifies that valid row and column indices are between 1 and  $n$ .

▼ How do I throw a `IndexOutOfBoundsException`?

Use a `throw` statement such as the following:

```
if (i <= 0 || i > n) throw new IndexOutOfBoundsException("row index i out of bounds");
```

Your code should not attempt to catch any exceptions—this will interfere with our grading scripts.

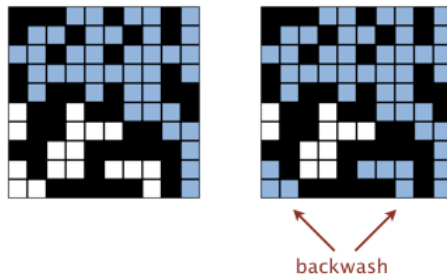
## ▼ How many lines of code should my program be?

You should strive for clarity and efficiency. Our reference solution for `Percolation.java` is about 70 lines, plus a test client. If you are re-implementing the union-find data structure (instead of reusing the implementations provided), you are on the wrong track.

▼ 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?

No, this is likely a bug in `Percolation`. It is only a minor deduction (because it impacts only the visualizer and not the experiment to estimate the percolation threshold), so don't go crazy trying to get this detail. However, many students consider this to be the most challenging and creative part of the assignment (especially if you limit yourself to one union-find object).

```
~/Desktop/percolation> java-algs4 PercolationVisualizer input10.txt
```



## PercolationStats

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

The sample standard deviation is undefined. We recommend returning `Double.NaN`.


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

Pick a site at random (by using `StdRandom` to generate two integers between 1 and  $n$ ) and use this site if it is blocked; if not, repeat.


▼ I don't get reliable timing information 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.

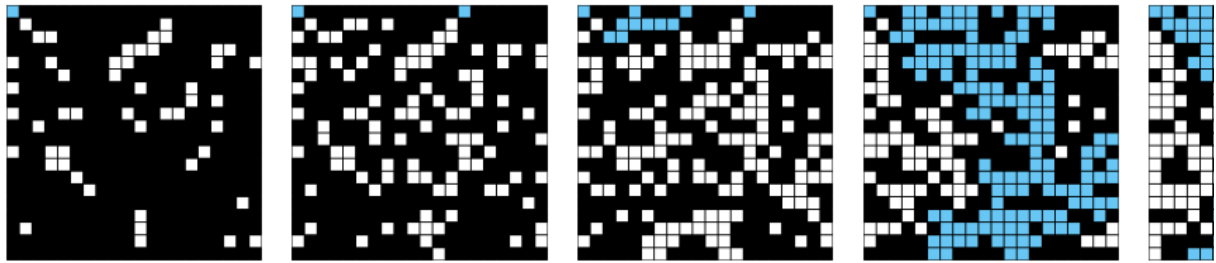
**Testing.** We provide two clients that serve as large-scale visual traces. We highly recommend using them for testing and debugging your `Percolation` implementation.

**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 (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

**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.

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 `.connected()` 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.**

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.