

See the Assessment Guide for information on how to interpret this report.

Want to receive personalized feedback on this submission? You can pay to have a teaching assistant read and provide personalized feedback on your submission at <https://mooc.codepost.io>.

ASSESSMENT SUMMARY

Compilation: PASSED
API: PASSED

SpotBugs: PASSED
PMD: PASSED
Checkstyle: FAILED (0 errors, 1 warning)

Correctness: 34/38 tests passed
Memory: 8/8 tests passed
Timing: 20/20 tests passed

Aggregate score: 93.68%
[Compilation: 5%, API: 5%, Style: 0%, Correctness: 60%, Timing: 10%, Memory: 20%]

ASSESSMENT DETAILS

The following files were submitted:

3.8K Mar 3 18:17 Percolation.java
2.7K Mar 3 18:17 PercolationStats.java

* COMPILING

% javac Percolation.java
*-----

% javac PercolationStats.java
*-----

=====

Checking the APIs of your programs.
*-----
Percolation:

PercolationStats:

=====

* CHECKING STYLE AND COMMON BUG PATTERNS

% spotbugs *.class
*-----

=====

% pmd .
*-----

=====

% checkstyle *.java
*-----

% custom checkstyle checks for Percolation.java
*-----
[WARN] Percolation.java:1: We recommend defining at least one private helper method, e.g., to validate the row and column indices or to map from 2D to 1D
Checkstyle ends with 0 errors and 1 warning.

% custom checkstyle checks for PercolationStats.java
*-----

=====

* TESTING CORRECTNESS

Testing correctness of Percolation

*-----

Running 21 total tests.

Tests 1 through 7 create a Percolation object using your code, then repeatedly open sites by calling `open()`. After each call to `open()`, it checks the return values of `isOpen()`, `percolates()`, `numberOfOpenSites()`, and `isFull()` in that order. Tests 12 through 15 create a Percolation object using your code, then repeatedly call the methods `open()`, `isOpen()`, `isFull()`, `percolates()`, and, `numberOfOpenSites()` in random order with probabilities $p = (p_1, p_2, p_3, p_4, p_5)$. The tests stop immediately after the system percolates.

Tests 18 through 21 test backwash.

Except as noted, a site is opened at most once.

Test 1: open predetermined list of sites using file inputs

```
* filename = input6.txt
* filename = input8.txt
* filename = input8-no.txt
* filename = input10-no.txt
* filename = greeting57.txt
* filename = heart25.txt
```

==> passed

Test 2: open random sites until the system percolates

```
* n = 3
* n = 5
* n = 10
* n = 10
* n = 20
* n = 20
* n = 50
* n = 50
```

==> passed

Test 3: open predetermined sites for $n = 1$ and $n = 2$ (corner case test)

```
* filename = input1.txt
* filename = input1-no.txt
* filename = input2.txt
* filename = input2-no.txt
```

==> passed

Test 4: check predetermined sites with long percolating path

```
* filename = snake13.txt
* filename = snake101.txt
```

==> passed

Test 5: open every site

```
* filename = input5.txt
```

==> passed

Test 6: open random sites until the system percolates,
allowing `open()` to be called on a site more than once

```
* n = 3
* n = 5
* n = 10
* n = 10
* n = 20
* n = 20
* n = 50
* n = 50
```

==> passed

Test 7: open random sites with large n

```
* n = 250
* n = 500
* n = 1000
* n = 2000
```

==> passed

Test 8: call methods with invalid arguments

```
* n = 10, (row, col) = (-1, 5)
* n = 10, (row, col) = (11, 5)
* n = 10, (row, col) = (0, 5)
* n = 10, (row, col) = (5, -1)
* n = 10, (row, col) = (5, 11)
* n = 10, (row, col) = (5, 0)
* n = 10, (row, col) = (-2147483648, -2147483648)
* n = 10, (row, col) = (2147483647, 2147483647)
```

==> passed

Test 9: call constructor with invalid argument

```
* n = -10
* n = -1
* n = 0
```

==> passed

Test 10: create multiple Percolation objects at the same time
(to make sure you didn't store data in static variables)

==> passed

Test 11: open predetermined list of sites using file inputs,
but permute the order in which methods are called

```
* filename = input8.txt; order = isFull(), isOpen(), percolates()
* filename = input8.txt; order = isFull(), percolates(), isOpen()
* filename = input8.txt; order = isOpen(), isFull(), percolates()
* filename = input8.txt; order = isOpen(), percolates(), isFull()
* filename = input8.txt; order = percolates(), isOpen(), isFull()
* filename = input8.txt; order = percolates(), isFull(), isOpen()
```

==> passed

Test 12: call `open()`, `isOpen()`, and `numberOfOpenSites()`
in random order until just before system percolates

```
* n = 3, trials = 40, p = (0.4, 0.4, 0.0, 0.0, 0.3)
* n = 5, trials = 20, p = (0.4, 0.4, 0.0, 0.0, 0.3)
* n = 7, trials = 10, p = (0.4, 0.4, 0.0, 0.0, 0.3)
* n = 10, trials = 5, p = (0.4, 0.4, 0.0, 0.0, 0.3)
* n = 20, trials = 2, p = (0.4, 0.4, 0.0, 0.0, 0.3)
* n = 50, trials = 1, p = (0.4, 0.4, 0.0, 0.0, 0.3)
==> passed
```

```
Test 13: call open() and percolates() in random order until just before system percolates
* n = 3, trials = 40, p = (0.5, 0.0, 0.0, 0.5, 0.0)
* n = 5, trials = 20, p = (0.5, 0.0, 0.0, 0.5, 0.0)
* n = 7, trials = 10, p = (0.5, 0.0, 0.0, 0.5, 0.0)
* n = 10, trials = 5, p = (0.5, 0.0, 0.0, 0.5, 0.0)
* n = 20, trials = 2, p = (0.5, 0.0, 0.0, 0.5, 0.0)
* n = 50, trials = 1, p = (0.5, 0.0, 0.0, 0.5, 0.0)
==> passed
```

```
Test 14: call open() and isFull() in random order until just before system percolates
* n = 3, trials = 40, p = (0.5, 0.0, 0.5, 0.0, 0.0)
* n = 5, trials = 20, p = (0.5, 0.0, 0.5, 0.0, 0.0)
* n = 7, trials = 10, p = (0.5, 0.0, 0.5, 0.0, 0.0)
* n = 10, trials = 5, p = (0.5, 0.0, 0.5, 0.0, 0.0)
* n = 20, trials = 2, p = (0.5, 0.0, 0.5, 0.0, 0.0)
* n = 50, trials = 1, p = (0.5, 0.0, 0.5, 0.0, 0.0)
==> passed
```

```
Test 15: call all methods in random order until just before system percolates
* n = 3, trials = 40, p = (0.2, 0.2, 0.2, 0.2, 0.2)
* n = 5, trials = 20, p = (0.2, 0.2, 0.2, 0.2, 0.2)
* n = 7, trials = 10, p = (0.2, 0.2, 0.2, 0.2, 0.2)
* n = 10, trials = 5, p = (0.2, 0.2, 0.2, 0.2, 0.2)
* n = 20, trials = 2, p = (0.2, 0.2, 0.2, 0.2, 0.2)
* n = 50, trials = 1, p = (0.2, 0.2, 0.2, 0.2, 0.2)
==> passed
```

```
Test 16: call all methods in random order until almost all sites are open
        (with inputs not prone to backwash)
* n = 3
* n = 5
* n = 7
* n = 10
* n = 20
* n = 50
==> passed
```

```
Test 17: substitute WeightedQuickUnionUF data type that sets root nondeterministically;
        call all methods in random order until almost all sites are open
        (with inputs not prone to backwash)
* n = 3
* n = 5
* n = 7
* n = 10
* n = 20
* n = 50
==> passed
```

```
Test 18: check for backwash with predetermined sites
* filename = input20.txt
  - isFull() returns wrong value after 231 sites opened
  - student   isFull(18, 1) = true
  - reference isFull(18, 1) = false

* filename = input10.txt
  - isFull() returns wrong value after 56 sites opened
  - student   isFull(9, 1) = true
  - reference isFull(9, 1) = false

* filename = input50.txt
  - isFull() returns wrong value after 1412 sites opened
  - student   isFull(22, 28) = true
  - reference isFull(22, 28) = false

* filename = jerry47.txt
  - isFull() returns wrong value after 1076 sites opened
  - student   isFull(11, 47) = true
  - reference isFull(11, 47) = false

* filename = sedgewick60.txt
  - isFull() returns wrong value after 1577 sites opened
  - student   isFull(21, 59) = true
  - reference isFull(21, 59) = false

* filename = wayne98.txt
  - isFull() returns wrong value after 3851 sites opened
  - student   isFull(69, 9) = true
  - reference isFull(69, 9) = false

==> FAILED
```

```
Test 19: check for backwash with predetermined sites that have
        multiple percolating paths
* filename = input3.txt
  - isFull() returns wrong value after 4 sites opened
  - student   isFull(3, 1) = true
  - reference isFull(3, 1) = false

* filename = input4.txt
  - isFull() returns wrong value after 7 sites opened
  - student   isFull(4, 4) = true
  - reference isFull(4, 4) = false

* filename = input7.txt
  - isFull() returns wrong value after 12 sites opened
  - student   isFull(6, 1) = true
```

```
- reference isFull(6, 1) = false
```



```
Test 20: call all methods in random order until all sites are open
         (these inputs are prone to backwash)
```

```
* n = 3
- isFull() returns wrong value after 5 sites opened
- student isFull(3, 3) = true
- reference isFull(3, 3) = false

- failed on trial 2 of 40

* n = 5
- isFull() returns wrong value after 14 sites opened
- student isFull(5, 5) = true
- reference isFull(5, 5) = false

- failed on trial 2 of 20

* n = 7
- isFull() returns wrong value after 35 sites opened
- student isFull(7, 7) = true
- reference isFull(7, 7) = false

- failed on trial 1 of 10

* n = 10
- isFull() returns wrong value after 60 sites opened
- student isFull(4, 1) = true
- reference isFull(4, 1) = false

- failed on trial 1 of 5

* n = 20
- isFull() returns wrong value after 245 sites opened
- student isFull(19, 10) = true
- reference isFull(19, 10) = false

- failed on trial 1 of 2

* n = 50
- isFull() returns wrong value after 1491 sites opened
- student isFull(35, 9) = true
- reference isFull(35, 9) = false

- failed on trial 1 of 1
```



```
Test 21: substitute WeightedQuickUnionUF data type that sets root nondeterministically;
         call all methods in random order until all sites are open
         (these inputs are prone to backwash)
```

```
* n = 3
- isFull() returns wrong value after 4 sites opened
- student isFull(3, 1) = true
- reference isFull(3, 1) = false

- failed on trial 4 of 40

* n = 5
- isFull() returns wrong value after 15 sites opened
- student isFull(4, 2) = true
- reference isFull(4, 2) = false

- failed on trial 2 of 20

* n = 7
- isFull() returns wrong value after 27 sites opened
- student isFull(4, 7) = true
- reference isFull(4, 7) = false

- failed on trial 1 of 10

* n = 10
- isFull() returns wrong value after 50 sites opened
- student isFull(10, 3) = true
- reference isFull(10, 3) = false

- failed on trial 1 of 5

* n = 20
- isFull() returns wrong value after 236 sites opened
- student isFull(13, 3) = true
- reference isFull(13, 3) = false

- failed on trial 1 of 2

* n = 50
- isFull() returns wrong value after 1536 sites opened
- student isFull(32, 16) = true
- reference isFull(32, 16) = false

- failed on trial 1 of 1
```



```
Total: 17/21 tests passed!
```

```
=====
*****
* TESTING CORRECTNESS (substituting reference Percolation)
```

Testing correctness of PercolationStats

*-----

Running 17 total tests.

Test 1: check formatting of output of main()

```
% java-algs4 PercolationStats 20 10
mean          = 0.566500
stddev        = 0.059397
95% confidence interval = [0.529685, 0.603315]
```

```
% java-algs4 PercolationStats 200 100
mean          = 0.593132
stddev        = 0.009317
95% confidence interval = [0.591306, 0.594958]
```

=> passed

Test 2: check that methods in PercolationStats do not print to standard output

```
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 3: check that mean() returns value in expected range

```
* n = 2, trials = 10000
* n = 5, trials = 10000
* n = 10, trials = 10000
* n = 25, trials = 10000
```

=> passed

Test 4: check that stddev() returns value in expected range

```
* n = 2, trials = 10000
* n = 5, trials = 10000
* n = 10, trials = 10000
* n = 25, trials = 10000
```

=> passed

Test 5: check that PercolationStats constructor creates trials Percolation objects, each of size n-by-n

```
* n = 15, trials = 15
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 6: check that PercolationStats.main() creates trials Percolation objects, each of size n-by-n

```
* n = 15, trials = 15
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 7: check that PercolationStats calls open() until system percolates

```
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 8: check that PercolationStats does not call open() after system percolates

```
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 9: check that mean() is consistent with the number of intercepted calls to open() on blocked sites

```
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 10: check that stddev() is consistent with the number of intercepted calls to open() on blocked sites

```
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 11: check that confidenceLo() and confidenceHigh() are consistent with mean() and stddev()

```
* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
```

=> passed

Test 12: check that exception is thrown if either n or trials is out of bounds

```
* n = -23, trials = 42
* n = 23, trials = 0
* n = -42, trials = 0
* n = 42, trials = -1
* n = -2147483648, trials = -2147483648
```

=> passed

Test 13: create two PercolationStats objects at the same time and check mean()

```

        (to make sure you didn't store data in static variables)
* n1 = 50, trials1 = 10, n2 = 50, trials2 = 5
* n1 = 50, trials1 = 5, n2 = 50, trials2 = 10
* n1 = 50, trials1 = 10, n2 = 25, trials2 = 10
* n1 = 25, trials1 = 10, n2 = 50, trials2 = 10
* n1 = 50, trials1 = 10, n2 = 15, trials2 = 100
* n1 = 15, trials1 = 100, n2 = 50, trials2 = 10
==> passed

```

Test 14: check that the methods return the same value, regardless of the order in which they are called

```

* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
==> passed

```

Test 15: check that no calls to StdRandom.setSeed()

```

* n = 20, trials = 10
* n = 20, trials = 10
* n = 40, trials = 10
* n = 80, trials = 10
==> passed

```

Test 16: check distribution of number of sites opened until percolation

```

* n = 2, trials = 100000
* n = 3, trials = 100000
* n = 4, trials = 100000
==> passed

```

Test 17: check that each site is opened the expected number of times

```

* n = 2, trials = 100000
* n = 3, trials = 100000
* n = 4, trials = 100000
==> passed

```

Total: 17/17 tests passed!

```

=====
*****
* MEMORY (substituting reference Percolation)
*****

```

Analyzing memory of PercolationStats

```

*-----
Running 4 total tests.

```

Test 1a-1d: check memory usage as a function of T trials for n = 100
(max allowed: 8*T + 128 bytes)

	T	bytes
=> passed	16	48
=> passed	32	48
=> passed	64	48
=> passed	128	48

=> 4/4 tests passed

Estimated student memory = 48.00 (R² = 1.000)

Total: 4/4 tests passed!

```

=====
*****
* TIMING (substituting reference Percolation)
*****

```

Timing PercolationStats

```

*-----
Running 4 total tests.

```

Test 1: Call PercolationStats constructor and instance methods and count calls to StdStats.mean() and StdStats.stddev().

```

* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
==> passed

```

Test 2: Call PercolationStats constructor and instance methods and count calls to methods in StdRandom.

```

* n = 20, trials = 10
* n = 20, trials = 10
* n = 40, trials = 10
* n = 80, trials = 10
==> passed

```

Test 3: Call PercolationStats constructor and instance methods and count calls to methods in Percolation.

```

* n = 20, trials = 10
* n = 50, trials = 20
* n = 100, trials = 50
* n = 64, trials = 150
==> passed

```

Test 4: Call PercolationStats constructor and instance methods with trials = 3 and values of n that go up by a multiplicative factor of sqrt(2).

```
The test passes when n reaches 2,896.

The approximate order-of-growth is  $n^{\log \text{ratio}}$ 

n  seconds  log ratio
-----
724    0.18    2.5
1024    0.47    2.7
1448    1.22    2.7
2048    2.97    2.6
2896    6.95    2.4
==> passed

Total: 4/4 tests passed!

=====

*****
* MEMORY
*****

Analyzing memory of Percolation
*-----
Running 4 total tests.

Test 1a-1d: check that total memory <= 17 n^2 + 128 n + 1024 bytes

n      bytes
-----
=> passed    64      37048
=> passed   256     590008
=> passed   512    2359480
=> passed  1024    9437368
==> 4/4 tests passed

Estimated student memory = 9.00 n^2 + 0.00 n + 184.00   (R^2 = 1.000)

Test 2 (bonus): check that total memory <= 11 n^2 + 128 n + 1024 bytes
- bonus available only if solution passes backwash correctness test
==> FAILED

Total: 4/4 tests passed!

=====
```

```
*****
* TIMING
*****

Timing Percolation
*-----
Running 16 total tests.

Test 1a-1e: Creates an n-by-n percolation system; open sites at random until
the system percolates, interleaving calls to percolates() and open().
Count calls to connected(), union() and find().

n      union()      2 * connected()
                        + find()      constructor
-----
=> passed    16         240              344              1
=> passed    32         725              1222             1
=> passed    64        2748              4716             1
=> passed   128       12114             19868             1
=> passed   256       47011             78402             1
=> passed   512      179773            306672             1
=> passed  1024     734115           1240482             1
==> 7/7 tests passed

If one of the values in the table violates the performance limits
the factor by which you failed the test appears in parentheses.
For example, (9.6x) in the union() column indicates that it uses
9.6x too many calls.

Tests 2a-2f: Check whether the number of calls to union(), connected(), and find()
is a constant per call to open(), isOpen(), isFull(), and percolates().
The table shows the maximum number of union() and find() calls made
during a single call to open(), isOpen(), isFull(), and percolates().
One call to connected() counts as two calls to find().

n      per open()      per isOpen()      per isFull()      per percolates()
-----
=> passed    16         4              0              2              2
=> passed    32         4              0              2              2
=> passed    64         4              0              2              2
=> passed   128         4              0              2              2
=> passed   256         4              0              2              2
=> passed   512         4              0              2              2
=> passed  1024         4              0              2              2
==> 7/7 tests passed
```

Running time (in seconds) depends on the machine on which the script runs.

Test 3: Create an n-by-n percolation system; interleave calls to percolates() and open() until the system percolates. The values of n go up by a factor of sqrt(2). The test is passed if n >= 4096 in under 10 seconds.

The approximate order-of-growth is $n^{\log \text{ratio}}$

n	seconds	log ratio	union-find operations	log ratio
1024	0.10	2.5	2745560	2.1
1448	0.27	2.9	5444522	2.0
2048	0.68	2.6	10817972	2.0
2896	1.82	2.9	21646960	2.0
4096	4.09	2.3	43282814	2.0

=> passed

Test 4: Create an n-by-n percolation system; interleave calls to open(), percolates(), isOpen(), isFull(), and numberOfOpenSites() until the system percolates. The values of n go up by a factor of sqrt(2). The test is passed if n >= 4096 in under 10 seconds.

n	seconds	log ratio	union-find operations	log ratio
1024	0.12	2.2	3925122	2.0
1448	0.30	2.8	7960924	2.0
2048	0.77	2.7	15839924	2.0
2896	1.95	2.7	31647398	2.0
4096	4.35	2.3	63376680	2.0

=> passed

Total: 16/16 tests passed!

=====