coursera

Assessment Guide

After you submit your code, it will be automatically graded by a comprehensive suite of tests which will analyze your program's style, correctness, memory usage, and timing. The results are then compiled into a report, which you can view by clicking on the score which appears next to the submit button after the grading script has finished.

The report consists of two major sections, the **Assessment Summary** and the **Assessment Details**. The Assessment Details section is further subdivided into **Compilation**, **API**, **Spotbugs**, **PMD**, **Checkstyle**, **Correctness**, **Memory**, and **Timing**.

Assessment Summary

The report begins with the Assessment Summary. An example is shown below.

```
Compilation: PASSED
API:         PASSED

Spotbugs:    PASSED
PMD:         FAILED (1 warning)
Checkstyle:  FAILED (0 errors, 3 warnings)

Correctness: 13/14 tests passed
Memory: 0/4 tests passed
Timing: 2/5 tests passed

Raw score: 73.71% [Compilation: 5%, API: 5%, Spotbugs: 0%, PMD: 0%, Checkstyle:
0%, Correctness: 65%, Memory: 10%, Timing: 25%]
```

The Compilation, API, Spotbugs, PMD, and Checkstyle tests report either PASSED or FAILED, along with the number of warnings.

- The **Compilation** tests check whether your Java programs compile without warning or error. If the autograder cannot successfully compile your programs, you will not receive any credit or additional feedback.
- The **API** tests verify that the APIs of your classes exactly match the APIs prescribed in the assignment specification. If there is any disagreement, you will not receive any credit or additional feedback.
- The **Spotbugs** and **PMD** tests inspect your program for common bug patterns. The results neither count toward your score nor prevent additional analysis.
- The **Checkstyle** tests check that your programs adhere to a set of style rules (see below for details). The results neither counts toward your score nor prevents additional analysis.

The correctness, memory, and timing tests report the number of tests passed.

- The **Correctness** tests check whether your programs are correct.
- The **Memory** tests analyze the memory usage of your programs.
- The **Timing** tests measure the running times of your programs.

The Compilation, API, Correctness, Memory, and Timing components are combined into an aggregate score, using the weights 5%, 5%, 60%, 10%, and 20%, respectively. In the example above, the student score is 5% + 5% + (13/14 * 60%) + (0/4 * 10%) + (2/5 * 20%), for a total of 73.71%.

Assessment Details

After the Assessment Summary, the Assessment Details follow; they begin with a listing of the files submitted, as shown below.

```
files submitted
----------------------------------
3.4K Aug 16 00:03 Percolation.java
3.4K Aug 16 00:01 PercolationStats.java
```

Compilation

The autograder uses a Java 8 compiler to compile your Java programs, but access to certain packages (such as java.net and java.util.concurrent) is restricted. The autograder includes the named package version of algs4.jar in the classpath. It also uses the -encoding UTF-8 option, which rejects input files that are not encoded in UTF-8. Here is an example.

```
******************************************************************************
*  compiling
******************************************************************************


% javac Percolation.java
*-----------------------------------------------------------
================================================================

% javac PercolationStats.java
*-----------------------------------------------------------
PercolationStats.java:3: <identifier> expected
    StdOut.printf("% java PercolationStats ");
                 ^
PercolationStats.java:3: illegal start of type
    StdOut.printf("% java PercolationStats ");
                  ^
```

The program Percolation.java compiles cleanly but the program PercolationStats.java does not. The autograder displays any compiler errors or warnings. You should see similar messages when you compile locally on your machine.

API

The API tests ensure that your program obeys the exact API prescribed in the assignment specification (including the class definition, constructor, instance variables, and instance methods). Your program may contain other constructors, instance variables, and instance methods, but they must be declared private. The one exception is public static void main(String[] args), which you are always free to include, even if it is not specified. Also, as with Java, the autograder assumes a default no-argument constructor if no other constructor is specified in the API. Finally, due to a technical limitation of the API checker, if the API specifies the name of a generic type parameter, you must use the same name in your program.

```
Testing the APIs of your programs.
*-----------------------------------------------------------
Percolation:
The following fields should be made private:
  *  public WeightedQuickUnionUF gridUF

The following methods should be removed or made private:
  *  public int xyToUF(int,int)


PercolationStats:
The following methods are missing:
  *  public double mean()


================================================================
```

In the example above, the API checker reports two API violations in Percolation.java. The first is a public instance variable named gridUF; the second is a public instance method named xyToUF(), neither of which appears in the assignment specification. The API checker reports one API violation in PercolationStats.java: it is missing the mean() method. Until you correct all API violations, your program will not be graded.

Spotbugs

Spotbugs scans Java bytecode and looks for common bug patterns. The autograder uses Spotbugs 3.1.3 with the configuration file spotbugs.xml. Here is a list of bug descriptions. Occasionally, Findbugs reports false positives, so the autograder does not count the results toward your score.

```
% spotbugs *.class
*-----------------------------------------------------------
H B ES_COMPARING_STRINGS_WITH_EQ ES: Compares two strings for reference equality
using '==' or '!='. Use the 'equals()' method for object equality (to check
whether two strings correspond to the same sequence of characters).  At
HorseSorter.java:[line 128]

Warnings generated: 1
================================================================
```

In the example above, Spotbugs reports one issue: two strings are being compared using the == operator instead of the equals() method. Usually, this is not what you want, so Spotbugs flags it as suspicious.

PMD

PMD scans Java source code and looks for common bug patterns. The autograder uses PMD 6.3.0 with the configuration file pmd.xml. Here is a list of bug descriptions. Occasionally, PMD reports false positives, so the autograder does not count the results toward your score.

```
% pmd .
*-----------------------------------------------------------
PercolationStats.java:112: Avoid unused method parameters such as 'alpha'.
[UnusedFormalParameter]
PMD ends with 1 warning.
================================================================
```

In the example above, PMD reports one issue: the formal parameter alpha is never used by its method.

Checkstyle

Checkstyle scans Java source code and verifies that the code adheres to a set of style guidelines. The autograder uses Checkstyle 8.11 with the configuration file checkstyle-coursera.xml. Here is a list of available checks. As style is inherently subjective, you may disagree with the style checker. In such cases, you are free to ignore it; the autograder does not count style toward your score.

```
% checkstyle *.java
*-----------------------------------------------------------
[WARN] Percolation.java:17:33: The instance variable 'grid_connections' must
start with a lowercase letter and use camelCase. [ParameterName]
[WARN] Percolation.java:12:1: File contains tab characters (this is the first
instance). Configure your editor to replace tabs with spaces. [FileTabCharacter]
Checkstyle ends with 0 errors and 2 warnings.
================================================================
```

In the example above, the style checker reports two warnings. The first is that the instance variable grid_connections does not use camel case. The second is that the file contains tab characters. To fix the first issue, rename the variable to gridConnections. To fix the second issue, configure your IDE or text editor to replace tabs with spaces.

Correctness

The correctness tests ensure that your program behaves as required by the assignment specification. Correctness tests vary widely in their operation and output formatting. An example is shown below.

```
Testing methods in Percolation
*-----------------------------------------------------------
Running 15 total tests.

Tests 1 through 8 create a Percolation object using your code, then repeatedly
open sites by calling open(). After each call to open(), we check the return
values of isOpen(i, j) for every (i, j), the return value of percolates(),
and the return value of isFull(i, j) for every (i, j), in that order.

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 just before system percolates
  *  N = 3
  *  N = 5
  *  N = 10
  *  N = 10
  *  N = 20
  *  N = 20
  *  N = 50
  *  N = 50
==> passed
```

For each assignment, you'll need to read the Assessment Details to understand each test. Every test is marked at the bottom as "==> passed" or "==> FAILED". When a test is failed, information is given to the user to assist with debugging. However, this feedback is not always thorough, and may not be sufficient to solve your problem alone. You will often need to write your own tests to uncover the bug.

An example of a failed test is shown below.

```
Test 11: Create multiple Percolation objects at the same time
         (to make sure you didn't store data in static variables)
     isOpen(6, 9) returns wrong value [after 1 site opened]
     - student   = true
     - reference = false
     isOpen(2, 4) returns wrong value [after 1 site opened]
     - student   = true
     - reference = false
     java.lang.ArrayIndexOutOfBoundsException: 12

     Percolation.isOpen(Percolation.java:68)
     TestPercolation.checkIsOpen(TestPercolation.java:46)
     TestPercolation.check(TestPercolation.java:82)
     TestPercolation.twoPercolations(TestPercolation.java:407)
     TestPercolation.test11(TestPercolation.java:438)
     TestPercolation.main(TestPercolation.java:704)

==> FAILED
```

The test fails because Java reports an IndexOutOfBoundsException. In the source code corresponding to this report (code not shown), the problem is that the size of the Percolation object is stored in a static variable, causing a problem when more than one Percolation object is created. However, the error message only hints as to what is wrong.

After all of the tests for a single .java file are completed, the total number of tests passed for that file is reported, as shown in the example below.

```
Total: 14/15 tests passed!


================================================================
```

We see that Percolation.java passed 14 of the 15 tests.

Memory

The Memory test ensures that your program utilizes memory efficiently. As with Correctness tests, memory tests vary widely in operation and formatting. The autograder uses classmexer to measure the memory of an object. We execute with the -XX:-UseCompressedOops command-line option to ensure that the memory model is consistent with the 64-bit memory model from lecture.

An example of a failed test is shown below.

```
Computing memory of Percolation
*-----------------------------------------------------------
Running 4 total tests.

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


                 n        bytes
--------------------------------------------
=> FAILED       64        94192   (1.2x)
=> FAILED      256      1358320   (1.2x)
=> FAILED      512      5337584   (1.2x)
=> FAILED     1024     21160432   (1.2x)
==> 0/4 tests passed

Estimated student memory = 20.00 n^2 + 184.00 n + 496.00 bytes  (R^2 = 1.000)

Total: 0/4 tests passed!

================================================================
```

In this example, the autograder creates Percolation objects of dimension 64, 256, 512, and 1024, and outputs the total memory in tabular format. Each row is marked as either "=> passed" or "=> FAILED". In this case, the submission fails all 4 subtests. If a subtest is failed, the degree of failure is marked in parentheses. Above, we see that each row is marked by (1.2x). This means that the program uses 1.2 times too much memory. The fact that this factor is constant as n grows means that the error is probably fairly minor.

For Tests 1a–1d, the autograder provides a formula for the maximum amount of memory you're allowed to use as a function of n (though this is not always the case).

Timing

This course is built around the notion of efficient algorithms. Thus, we expect that your programs should not only solve the problems at hand, but should also do so in an efficient manner. The timing tests typically measure temporal efficiency in two ways. First, it measures the raw time your program needs to complete a set of tasks. For many assignments, the timing test will also count the number of elementary operations that your program makes to solve a problem of a given size. As a example, consider the timing test for Percolation.java below.

```
Timing Percolation
*-----------------------------------------------------------
Running 5 total tests.

Tests 1a-1e: Measuring runtime and counting calls to union() and find() in
WeightedQuickUnionUF. Note that connected() makes two calls to find().

For each n, a percolation object is generated and sites are randomly opened until
the system percolates. If you do not pass the correctness tests, these results may
be meaningless.


                 n   seconds     # union()              # find()
--------------------------------------------------------------------------------
=> passed        8     0.00         126                    302
=> passed       32     0.02        1530                  16646
=> FAILED      128     0.07       22670                 771594   (2.6x)
=> FAILED      512     0.50      371032               47158700   (9.4x)
=> FAILED     1024     2.15     1458534              363126570  (18.2x)
==> 2/5 tests passed

The running time (in seconds) depends on the machine on which the script runs, and may
vary each time that you submit. If one of the values in the table violates the
performance limits, the factor by which you failed the test appears in parentheses,
e.g. the 9.4x in the find() column indicates your test uses 9.4x too many calls.

Total: 2/5 tests passed!
================================================================
```

Running time in seconds depends on the machine on which the script runs, and may vary each time that you submit. If one of the values in the table violates the performance limits, the factor by which you failed the test appears in parentheses, e.g. (9.6x) in the union() column would indicate your test uses 9.6x too many calls.

As with the **Memory** and **Correctness** tests, the format for the **Timing** tests vary with each assignment. For Percolation.java, the testing is primarily focused on operation counts, i.e. how many total calls are made to union() and find(), though you will also fail the test if your program is unable to solve an instance of size n = 1024 in the allotted time.

In the example above, we see that the submission fails for n = { 128, 512, 1,024 }. The problem is that there are too many calls to find(), as noted in parentheses next to the find() count. For example, if n = 128, the number of find calls is 2.6x higher than allowed. Note that the failure factor grows as n becomes larger, which means that the order of growth of the running time with respect to n is growing more rapidly than allowed in the assignment instructions. This is a sure sign that there is a fundamental flaw in this submission's approach, which will most likely require substantive changes.

Test aborted. Ran out of time or crashed before completion

If the autograder is unable to complete a correcting, timing, or memory test in its allotted CPU time, it will abort and report *Test aborted. Ran out of time or crashed before completion*. When the autograder aborts, it is (currently) unable to award partial credit for the number of tests passed before aborting, so you will receive a score of 0 for this component. This usually indicates a serious performance bug. You can also get the same error message if the autograder crashes (usually because the submitted program is throwing some kind of exception that was unexpected), but this is extremely rare.