

CSCI241, Winter 2019, Lab 2

Due Tuesday, January 22 at 9:59pm

Unlike the homework assignments for CSCI241, you are **encouraged** to work with your peers in completing the labs. However, each student must write and submit their own code—no files may be exchanged. If any of this is unclear, please ask for further clarification.

1 Overview

Testing is an important part of software development. It is done to assess whether a software product will properly serve its intended purpose. Many different parties are involved in testing over the lifetime of a software product, and there exist many different methods of software testing. This lab focuses on *unit testing*, which aims to assess the correctness/usefulness of individual components of a larger program. In other words, each test checks that a small “unit” of the system works as intended. The unit to be tested may be a class, a method, or even a particular usage of a method. The correctness of each individual unit in a larger program can be thought of as a necessary, but not sufficient, condition for the program to work. Testing individual units can also make it much easier to locate and fix bugs.

Writing tests is hard work. Often, writing a good unit test is harder than writing the unit to be tested. However, **writing tests is worth it:** finding and fixing subtle bugs can take boundless amounts of programmer time, and uncaught bugs can be exceedingly expensive. For this reason, many major software companies require tests to be written for any new functionality introduced into production code.

In this course, we will be using JUnit, a widely-used testing framework for Java. JUnit provides functionality to write and run tests that make *assertions* that verify that code behaves the way we expect it to. For A1, you have been given a test suite (a collection of unit tests) to verify that your sorting methods work correctly. The test suite relies on several helper methods that check properties of arrays; your task in this lab is to implement these helper methods, implement insertion sort, and verify that your insertion sort method passes all its tests.

2 Git and submission for Lab 2

For this lab, you will be working in your A1 repository. If you have not yet accepted the Github Classroom invitation, find the link on canvas and clone your repository as per the instructions for A1. You will submit your code for this lab by committing and pushing the changes to your SortsTest.java file to your remote A1 repository on GitHub.

It is recommended that you `git add` and `git commit` regularly while developing your code, at least once per method you implement. When you have something working, `git push` your

changes to *GitHub*.

Whether or not you finish implementing `SortsTest` during lab, make sure to commit and push whatever code you have at the end of the lab period to be sure you receive credit for attending lab.

3 The `SortsTest` Class

Gradle knows how to compile and run JUnit tests. The standard location for test code in a gradle project such as ours is `src/test/java/`; all A1 classes live in the `sort` package, so you'll find `SortsTest.java` at `src/test/java/sorts/SortsTest.java`.

`SortsTest.java` contains a number of methods preceeded by the `@Test` directive, which tells JUnit that they are test cases that should be run as part of the test suite. Each test case makes one or more assertions using methods like `assertTrue` or `assertEquals`, to check that code behaves as expected. **The test cases for A1 have been provided for you, but they call helper methods that you must implement for them to work.**

To check that a sorting method has done its job correctly, the resulting array must have two properties:

- The resulting array is sorted.
- The resulting array has exactly the same elements as the original.

Below all the test cases, you will find stubs (i.e., method headers with missing implementation) for the methods that you need to implement to make the tests work correctly. This includes two methods that check the above two properties for sorted arrays, and one method that is used in tests for the `partition` helper method for quick sort.

4 Running Tests with Gradle

You can run JUnit tests with gradle with the command `gradle test`. If your main source code or test code is not compiled, gradle will automatically run the `build` task to make sure the code is compiled before the tests are run. If all the tests pass, you'll get a list of `Tasks` that were run followed by a message that says `Build Successful`. If any tests fail (which, unless you've implemented all of the sorts correctly, some should at this point), you'll get a message saying that the build failed because some of the tests did not pass. The output from Gradle tells you which tests failed, and points you to the location of a more detailed report, found in an html file located at `build/reports/tests/test/index.html`. You can go into this directory and double-click the file open it in a browser; this will show you a stack trace for each test failure to help you track down where your code went wrong. Before writing any code, try this out—you should find that none of the tests pass, and the stack traces show that they fail because the helper methods below all return `false`.

5 Your Tasks

For each of the methods you need to implement, see the specification in the code for details of how it should behave.

1. Implement `isSorted`, which checks whether an array is sorted.

2. Implement `public static boolean sameElements`, which checks whether two arrays contain the same elements. Hint: use a `java.util.HashMap<Integer,Integer>` to keep track of how many times each value appears. The easiest way to find documentation on the `HashMap` class (or any other java class) is by googling “java 8 HashMap”

Potentially Useful Methods:

- `put(K key, V value)`
 - `get(Object key)`
 - `containsKey(Object key)`
 - `isEmpty()`
 - `remove(Object key)`
3. `public static boolean isPartitioned` checks whether the array has been correctly partitioned around a given “pivot” element.
 4. At this point, if you have not written the `insertionSort` method in `Sorts.java`, you should do so now. The algorithm was given in Lecture 1, slides for which are available on the course webpage.
 5. Finally, run `gradle test` and make sure the tests for `insertionSort` pass. Debug your code until your code passes all the tests with “Insertion” in the method name. Keep in mind that bugs may reside in `insertionSort` or in the test code itself. Use the information in the stack trace to help you out.
 6. Make sure you have committed and pushed both `Sorts.java` (with `insertionSort` completed—the other sorts need not be done until you submit A1) and `TestSorts.java` to your A1 repository on github.

Rubric

This lab is worth 10 points. The three helper methods in `SortsTest.java` are worth 3 points each, and at least an attempt at implementing `insertionSort` is worth 1 point. You are not strictly required to have all the insertion sort tests passing to get credit for this lab, but it’s recommended. Insertion sort’s correctness will be graded as a part of A1.

Deductions may be made for:

- Submission issues
- Compile or run-time errors when running tests
- Poor coding style (e.g. commenting, indentation, variable naming, etc.)

Acknowledgments

Thanks are owed to Tanzima Islam, Qiang Hao, Brian Hutchinson, Filip Jagodzinski, and others for producing and refining past labs from which this lab was adapted.