1 Goals

- Set up your programming and submission environment for the course;
- Review implementation of simple sorting algorithms in `C++`;
- Practice running and writing unit tests;
- Practice running performance tests.

Note that you may use whatever environment you like for this class, but your programs must be able to compile and run using `g++` (or `clang`), `cmake`, and `make`. The department provides a remote development server (`ada.gonzaga.edu`) running Ubuntu that can be accessed using an ssh remote connection from within VS Code on your own computer. Department on your computers configuration, you may be able to install the tools you need locally to complete the homework assignments. It is important that you start assignments early in this class so that if you have questions you can ask them and get them answered with enough time before the homework deadline (to avoid late penalties).

2 Instructions

1. If you don't have one already, create a GitHub account (github.com). Once you have an account, accept the GitHub Classroom assignment via the URL posted in piazza. Note that you may need to also create a personal access token to use `git` commands.
2. Use `git clone` to clone the classroom repository created for you and to obtain the starter code (either onto the remote development server if using `ada` or locally if you are using your own machine). (Note you should create a new directory for your assignments, although it isn't required.) This will create a new directory where you can work on your code. Be sure to frequently add, commit, and push your updated files back to your GitHub repository (via `git add`, `git commit`, and `git push`).
3. Implement the three sorting functions (for selection, insertion, and bubble sort) in `simple_sorts.cpp`. As you fill in the functions, you will want to frequently run the unit tests. Not all of the unit tests are initially implemented (see the following step). See detailed instructions below on how to compile and run your programs (including the unit tests) for this assignment.
4. There are 15 total unit tests in `hw1_test.cpp`, five for each sorting algorithm. For each sorting algorithm, two of the tests are incomplete (one for partially ordered lists and one that is open-ended for you to define). Once your sorting algorithms are working for the provided tests (i.e., the tests pass), you should code the remaining six tests. The goal is for you to practice writing new unit tests and to get a sense for how the testing framework works.
5. Once your code is completed and the unit tests pass, run the provided performance tests to generate comparisons of your sorting algorithm implementations. The performance tests will generate a data file with testing results, which you will then graph using `gnuplot`. You will need to add the corresponding graphs to your assignment write up (next step). Again, see instructions below on how to run the performance tests and generate the corresponding graphs.

6. Create your assignment write up and add it as a PDF file to your assignment (GitHub) repository. You **must** save your writeup as a PDF file and name it `hw1-writeup.pdf`. (Note no capital letters, no spaces, etc.) Be sure to push all of your source code and your write up by the due date so it can be graded. (Note that you can check that everything is in your repo from the GitHub website and/or using the `git status` command.) See below for expectations of your write up file. Once you have submitted your files to your GitHub repository and are ready to submit your assignment for grading, you must fill out the grading submission form. A link to the form will be provided in piazza (in the same post as the GitHub classroom link).

3 Additional Details

Compiling and running your code. You will need to use `cmake` and `make` to compile your code. First, you will use `cmake` to configure a build script (that will be needed by `make`). Run the following `cmake` command from the command line[1]:

```
cmake CMakeLists.txt
```

To help in understanding what `cmake` is doing, you should look at the `CMakeLists.txt` file. Note that you will only need to run the `cmake` command **once** for this assignment (assuming it completes without errors). Once completed, you will compile your program using the `make` command:

```
make
```

This will use the build script created by `cmake` to create two executable files: `hw1_test` and `hw1_perf` (assuming your code compiles). Once you are able to successfully compile via `make`, you will be able to run the `hw1_test` program (see below). Note that as you modify your code, you *only* need to rerun the `make` command to recompile. Thus, you will run `cmake` once and then `make` many times.

Unit tests. For this class, we are using the Google Test framework to write basic tests to help ensure our code is working correctly. For HW1, a number of unit tests are already created for you in `hw1_test.cpp`. By running the `hw1_test` executable file, you will see the results of each unit test defined in `hw1_test.cpp`. Your job is to implement the sorting functions and get all of the corresponding tests to pass (without changing the tests in `hw1_test.cpp`). In addition, you will need to fill in the unfinished unit tests in `hw1_test.cpp` (see above). The goal for this assignment with respect to unit tests is to create some new tests to get the hang of the framework (the basics).

Performance Data and Graphs. Once all of your unit tests pass, you will need to run the performance tests over your sorting function implementations. The performance tests are implemented in `hw1_perf.cpp` (which you don't need to modify for this assignment). During compilation, the `make` script will generate the `hw1_perf` executable. Running `hw1_perf` will output the testing result data that we will graph using a provided `gnuplot` script (`plot_script.gp`). You will need to save the testing data in a file and then provide the file as input to the script. To do this, first run the performance tests using the command:

```
./hw1perf > output.dat
```

This will run the tests and store the results (via "redirection") to the file `output.dat`. Note that the performance tests also run basic correctness tests. If these fail, you may see the corresponding errors.

[1] Alternatively, you can simply run: `cmake .`

You will need to fix the errors before graphing the performance test results. Once you have generated the `output.dat` file, generate the performance graph using the following command:

```
gnuplot -c plot_script.gp
```

This will generate the graphs in a file named `simple-sort-perf.png`. This file can be included directly within your write up.

Homework Writeup. You must create a homework write up that contains the graph generated from the previous step with a description explaining why you think the performance tests came out the way they did based on what you know about the implementations. Your job is to come up with hypotheses explaining the test results, including why the test results for each implementation are different (or the same). Your write up must also contain a brief paragraph on any implementation issues and/or challenges you ran into and how you addressed them. Note that the writeup does not need to be more than a couple of short paragraphs in total. However, instead of just explaining what the performance results are (which can be seen from the graph), you only need to provide your analysis of the results (one paragraph). The second paragraph should briefly describe any challenges or issues you faced.