All exercises must be done by yourself. You may discuss questions and potential solutions with your classmates, but you may not look at their code. If in doubt, ask the instructor.

Acknowledge all sources you found useful.

Your code should compute the correct results.

Partial credit is available, so attempt all exercises.

Submit your code and answers in a single archive file (ZIP/TGZ/TBZ2, etc.) that contains your name in the filename (e.g. `JRandom_A2.zip`). Your answers should be a PDF file in the archive.

The goals of this assignment is to parallelize an existing serial code, evaluate various techniques for mutual exclusion, and to diagnose load imbalance.

The existing serial code is an implementation of the Bellman–Ford algorithm for Single Source Shortest Paths. You may use pthreads or C++11 threading to parallelize this code (though I strongly suspect that using C++11 threads will be easier).

A few good tutorials on these topics are:

1. pthreads: `https://computing.llnl.gov/tutorials/pthreads/`
2. pthreads: `http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html`
3. C++11 threading: `https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/labs/lab6/`
4. C++11 threading, the book https://proquest.safaribooksonline.com/9781933988771

The references for pthreads and C++11 threading are:

1. pthreads: `http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html` (see also manual pages on your Linux system)
2. C++11 threads: `http://en.cppreference.com/w/cpp/thread`

If you get stuck while using pthreads or C++11 threads, please post in Piazza, approach the TA or ask the instructor *as soon as possible*.

You will be asked to report execution times for some of the exercises below. Here is the procedure you should use:

- Run the program multiple times,
- Average the times obtained from these multiple runs,
- Report the average and the standard deviation of these times,
- Report the number of times you ran the program.

To minimize the variance, you should attempt to run this on as unloaded a system as you possibly can.

You should report times on `node2x14a` or `node2x18a` machines on the instructional network. Note which machine you used.

I have supplied an initial `Makefile`, adapt it so that all the binaries listed below can be compiled simply by using `make` *`binaryname`*.

Exercise 1

Parallelize both the `sssp_init` and `sssp_round` functions in the given serial code. Parallelize the outer loop (i.e. each thread should handle a subset of the outer loop's iterations).

Create threads anew for each invocation of `sssp_round` in the serial version. Use lock-free methods (i.e. atomic read–modify–writes) to update shared values.

The binary for this exercise should be called `sssp` and will be invoked as:

```
sssp graphfile resultsfile numberofthreads
```

1. Identify any data races in `sssp_init`
2. Identify any data races in `sssp_round`
3. Instead of creating threads anew for each round, use a barrier in `sssp_round` to synchronize with other threads. Call this binary `sssp_barrier`. What performance difference do you see compared to creating threads anew?

Exercise 2

Convert the lock-free code to use locks. Report the execution times as the number of threads is varied for each of the variants below.

1. Use one lock for the the entire graph, call this binary `sssp_graphlock`.
2. Use two locks per edge (i.e. a lock for the source vertex and a lock for the destination vertex). I.e. use a lock per vertex. Call this binary `sssp_nodelock`. Account for deadlocks and explain your strategy.
3. Do not use any locks. Call this binary `sssp_relaxed`. Do the semantics of SSSP allow this to work? Reason especially about reads/writes in different rounds.

   Note, this does not mean you can use ordinary loads/stores. You must use atomic memory loads/stores with relaxed memory semantics. E.g. use C++11 atomic variables with memory order relaxed loads/stores. See this discussion on the GCC page if you're using pthreads.

Exercise 3

Measure the time for each thread for a fixed number of threads when running on an input. Report these times, and explain why you observe a load imbalance (if any).

END.