# Homework Assignment #5 [30 pts] – Java Threads

You are expected to do your own work on all homework assignments. (See the statement of Academic Dishonesty on the Syllabus.)

Check the Syllabus for the late assignment policy for this course.

## How to turn in?

Assignments need to be turned in via Laulima. Check the Syllabus for the late assignment policy for the course.

## What to turn in?

You should turn in a tarred archive named `ics332_hw5_USERNAME.tar` that contains a single top-level directory called `ics332_hw5_USERNAME`, where `USERNAME` is your UH username. In that directory you should have all the files **named exactly** as specified in the questions below.

Expected contents of the `ics332_hw5_USERNAME` directory:

- `README.txt`: your report in plain ASCII text (for Exercise #1)
- `paths`: a directory with all code for Exercise #1:
  - `ComputePaths.java`: Source code you have to write (based on provided starter code)
  - `FloydWarshall.java`: Floyd-Warshall implementation (provided to you and not to be modified)

## Environment

Exercise #1 can be done on any environment. As usual, we'll test everything on Linux.

# Exercise #1: Multi-threading for Speed [30 pts]

## Overview

You are hired by a company and tasked with setting up a driving directions app in Java. Like Google Map, the idea is to pre-compute a bunch of all-pair-shortest-paths so that user queries can be answered quickly. The concern is that the compute time will be problematic because, as you know from ICS311, the Floyd-Warshall algorithm has complexity $O(n^3)$ for a graph with n vertices.

can run.

This exercise corresponds to the simplest possible “more speed with threads” scenario.

## Question #1 [2 pts]

You are given two Java source files that form a package called paths:

- ComputePaths.java: the main class, which runs Floyd-Warshall on 2520 random graphs
- FloydWarshall.java: an implementation of the Floyd-Warshall algorithm that prints the sum of all shortest path lengths

Download these files to a directory called paths, and you can compile and run from the command-line as follows (or use whatever IDE, etc.):

```
% ls -R
paths

./paths:
ComputePaths.java  FloydWarshall.java

% javac paths/*.java

% java paths.ComputePaths
Usage: java ComputePaths <graph size> <# threads>
```

The program requires two command-line arguments:

- a graph size: how big should the random graphs be
- a number of threads: how many threads should be used (this argument is ignored in the code provided to you)

Running the program for graphs with 250 vertices (and passing 1 for the number of threads, which is ignored anyway) produces this output on my laptop:

```
% java paths/ComputePaths 250 1
Ran Floyd-Warshall on Graph #   0: 68790
Ran Floyd-Warshall on Graph #   1: 65671
Ran Floyd-Warshall on Graph #   2: 64633
Ran Floyd-Warshall on Graph #   3: 64193
[...]
Ran Floyd-Warshall on Graph #2519: 124981
```

By trial-and-error (i.e., doing a human-driven binary search), determine the graph size that leads this code to run in between 25 and 30 seconds on your machine. Note that repeated runs will show some variation.

*Warning:* You should run your code on a "quiescent" system, i.e., without running any other applications at the same time.

## Question #2 [20 pts]

Enhance ComputePaths.java so that it uses the specified number of threads. The idea is to split up the work among threads. For instance, when using 2 threads each thread should compute 2520/2 = 1260 graphs; using 3 threads, each thread should compute 2520/3 = 840 graphs; etc. Note that I picked 2520 because it is divisible by 1, 2, 3, ..., 10.

For instance, on my laptop, repeating the run from the previous question but specifying 2 threads, the output from the enhanced program looks like:

```
% java paths/ComputePaths 250 2
Ran Floyd-Warshall on Graph #   0: 68790
Ran Floyd-Warshall on Graph #1260: 93759
Ran Floyd-Warshall on Graph #1261: 93785
Ran Floyd-Warshall on Graph #   1: 65671
[...]
Ran Floyd-Warshall on Graph #2519: 124981
All graphs computed in 23.964 seconds
```

You should observe that using 2 threads your program goes faster. If not, you most likely have a bug. Make sure your program uses the number of threads that's specified.

Create a "thread" class called `Worker` that's nested in class `ComputePaths` that implements the `Runnable` interface. Do not use the `ExecutorService`.

*Hints:*

- The "Ran Floyd-Warshall..." messages will appear out of order, which is fine
- Using an array of threads is a good idea
- Because 2520 is divisible by all integers up to and including 10, and because we won't run this code with more than 10 threads, it's absolutely trivial to divide up the work among the threads.
- The only method you have to modify in the `ComputePaths` class is `compute()`.
- Your code should still work if one specifies 1 thread (and use only only thread).

## Question #3 [8 pts]

Run your code for 1, 2, 3, ..., 10 threads. For each number of threads, run your code 5 times and compute the average execution time. In your report (`README.txt`) list the average execution time for each number of threads. Then answer the following questions:

- [q1] How much faster does the program run with 2 threads when compared to using 1 thread? (e.g., 1.95x faster)

- [q4] What is the largest acceleration factor you observe in your results when compared to the 1-thread case, and with how many threads?