CS115 - Computer Simulation, Assignment #1 – Train Unloading Dock
Due at START of class in the 6th Lecture of the Quarter
Note you CANNOT use CSIM – must use a non-simulation language

In this assignment, you will write a simulation of a train unloading dock. Trains arrive at the station as a Poisson process on average once every 10 hours. Each train takes between 3.5 and 4.5 hours, uniformly at random, to unload. If the loading dock is busy, then trains wait in a first-come, first-served queue outside the loading dock for the currently unloading train to finish. Negligible time passes between the departure of one train, and the entry of the next train (if any) into the loading dock---unless the entering train has no crew (see below).

There are a number of complications. Each train has a crew that, by union regulations, cannot work more than 12 hours at a time. When a train arrives at the station, the crew's remaining work time is uniformly distributed at random between 6 and 11 hours. When a crew abandons their train at the end of their shift, they are said to have "hogged out". A train whose crew has hogged out cannot be moved, and so if a hogged-out train is at the front of the queue and the train in front finishes unloading, it cannot be moved into the loading dock until a replacement crew arrives (crews from other trains cannot be used). Furthermore, a train that is already in the loading dock cannot be unloaded in the absence of its crew, so once the crew hogs out, unloading must stop temporarily until the next crew arrives for that train. This means that the unloading dock can be idle even if there is a train in it, and even if it is empty and a (hogged-out) train is at the front of the queue.

Once a train's crew has hogged out, the arrival of a replacement crew takes between 2.5 and 3.5 hours, uniformly at random. However, due to union regulations, the new crew's 12-hour clock starts ticking as soon as they are called in for replacement (i.e., at the instant the previous crew hogged out); i.e., their 2.5-3.5 hour travel time counts as part of their 12-hour shift.

You will simulate for 72,000 hours (plus the time it takes for all remaining trains to depart), and output the following statistics at the end:

1. Total number of trains served.
2. Average and maximum of the time-in-system over trains.
3. The percentage of time the loading dock spent busy, idle, and hogged-out (i.e., when prevented from unloading a train that has hogged-out) (does this add to 100%? Why or why not?)
4. Time average and maximum number of trains in the queue.
5. A histogram of the number of trains that hogged out 0, 1, 2, etc times.

Input Specification: We *will* run your code, but to make it easy for the grader, we need everybody to adhere to the following guidelines: create a makefile that builds your program (if necessary), and your program should take either two or three command-line arguments. When given three arguments, the first argument should just be "-s", your program should read the second argument as the *file path* to the *pre-determined train arrival schedule* (described below), and it should read the third argument as the *file path* to *the pre-determined travel-times for new crews* (also described below). When only given two arguments, the first argument must be the *average train inter-arrival time,* and the second argument must be the *total simulation time*. Thus, for example, if your program is written in C and compiled to an executable called "`train`", then to run it with the default parameters above, I should be able to run it on my Unix command line as:

```
$ make
$ ./train 10.0 72000.0
```

or

```
      $ make
      $ ./train –s schedule.txt traveltimes.txt
```

If you are using a language that does not run like the above (e.g. Python "`python train.py 10.0 72000.0`", or Java "`java -cp . train 10.0 72000.0`") create a shell script or program wrapper that takes in the arguments and runs your code as above. For example, if you are using Python, you can create a shell script named "`train.sh`" containing:

```
#!/usr/bin/env bash
python train.py $@
```

That will allow the grader to run your program using:

```
      $ ./train.sh 10.0 72000.0
```

(Note: If you want to use this yourself on GNU/Linux, you'll need to mark it as executable using the command "`chmod –x train.sh`")

Also, if you are using a language that does not require building/compiling (e.g. Python), just create a makefile with no targets:

```
target: ;
```

The pre-determined train arrival schedule contains three space-delimited columns: *arrival times*, *unloading times,* and *remaining crew hours* (in that order), with each arrival event on a new line:

```
0.02013 3.70 8.92
8.12 4.12 10.10
12.52 3.98 7.82
...
1210.0 4.12 9.21
```

The pre-determined travel-times for new crews contains a single column of data: the *travel-time for new crews*. It would be safe to assume there could be more rows in this file than the previous file.

```
2.51
3.0001
...
2.89
```

When using a pre-determined schedule and there are no more train arrivals scheduled, end the simulation after the very last train has departed; when using random values, stop adding arrival events after the total simulation time has passed as specified by the command arguments (e.g., at 72,000 hours). The simulation will stop after the last train has departed in this case as well.

Output Specification: Your program should print one line for every event that gets called. I want to be able to follow what's happening in your code. Each train and each crew should be assigned an increasing integer ID. The final statistics should come after the simulation output and closely match the specified format. Output lines should resemble the following example:

```
Time 0.00: train 0 arrival for 4.45h of unloading,
           crew 0 with 7.80h before hogout (Q=0)
Time 0.00: train 0 entering dock for 4.45h of unloading,
           crew 0 with 7.80h before hogout
Time 0.56: train 1 arrival for 4.33h of unloading,
           crew 1 with 6.12h before hogout (Q=1)
Time 3.72: train 0 departing (Q=1)
Time 3.72: train 1 entering dock for 4.33h of unloading,
           crew 1 with 2.96h before hogout
Time 6.68: train 1 crew 1 hogged out during service (SERVER HOGGED)
Time 9.43: train 1 replacement crew 2 arrives (SERVER UNHOGGED)
...
Time 319.64: train 33 entering dock for 3.51h of unloading,
             crew 39 with 3.09h before hogout (Q=2)
Time 322.73: train 33 crew 39 hogged out during service (SERVER HOGGED)
Time 323.63: train 34 crew 40 hogged out in queue
Time 325.78: train 33 replacement crew 41 arrives (SERVER UNHOGGED)
Time 326.20: train 33 departing (Q=1)
Time 326.20: train 34 crew 42 hasn't arrived yet,
             cannot enter dock (SERVER HOGGED)
Time 326.82: train 34 replacement crew 42 arrives (SERVER UNHOGGED)
Time 326.82: train 34 entering dock for 4.32h of unloading,
             crew 42 with 8.81h before hogout (Q=1)
...
Time 1234.00: simulation ended

Statistics
----------
Total number of trains served: 512
Average time-in-system per train: 3.141593h
Maximum time-in-system per train: 6.283186h
Dock idle percentage: 64.00%
Dock busy percentage: 32.00%
Dock hogged-out percentage: 16.00%
Time average of trains in queue: 0.1234
Maximum number of trains in queue: 2
Histogram of hogout count per train:
[0]: 512
[1]: 128
[2]: 32
... (show as many bins as necessary)
```

Numbers can have any number of decimal places (valid examples include 13 and 3.14159), but no scientific notation.

A script to automatically check the output formatting will be provided to give feedback on I/O specification compliance. To run it with your program's output, you can run the following commands:

```
    $ ./train –s schedule.txt traveltimes.txt > output.txt
    $ ./python3 checker.py output.txt
```

Submission: Submit a brief write-up of your results, along with your source code and specific sections of the output (only the first few pages and the last page) for the default parameters. There should be no more than 10 pages. Submit both a paper printout to me in class, and also electronically via the submit command on ICS openlab**.**

Grading: You can use any non-simulation language (ie., any language not already designed for simulation), but it must allow command-line execution similar to the above, and I must be able to run it on my Linux box. This probably eliminates most proprietary languages, such as Matlab. However, if you want to use anything "weird" (i.e., anything other than Pascal, Fortran, C, C++, Java, Lisp, or Scheme), please clear it with me first. In the worst case, I may ask you to run it for me, in my office, in front of my eyes, if I can't figure out how to run your language myself.

Your code should be "pretty", which means easily understood and maintainable by another programmer. This is of course subjective, but it should be well indented, and well commented inside, such that, if we were fellow employees and I had to change your code, I wouldn't be cursing your birth after several hours (or days) of trying to figure it out. It should be easy-to-read; the variables should have meaningful (but not overly verbose) names; the comments should clarify tricky points but not obvious ones. (I've seen the comment "add one to x" beside "x++"; that qualifies as an unnecessary comment. A better comment would tell us WHY x is being incremented, if it's not obvious.) The prettiness (i.e., understandability, readability, and maintainability) of your code, by the grader's judgment alone, will count as 25% of your grade.

Your code should be correct. We will judge the correctness of your code both by reading it, and based on tracing its output events and final statistics. Correctness of the traces and the output will count for 50% of the grade. The write-up will count for the remaining 25%.