CS115 - Computer Simulation, Assignment #1 – Train Unloading Dock
Due at START of class in the 8th Lecture of the Quarter
Note: you *must* use a non-simulation language, e.g. Python, and no simulation-specific classes (eg, no event or simulation time handling classes).

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 lie unused 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; in both of these cases we say the *loading dock* is "hogged-out". More specifically, the loading dock is hogged out if there are *any* trains in the queue but the loading dock is not actively unloading. (A bit of thought should convince you that trains in the queue behind the one at the front do not affect the hogged-out status of the unloading dock.) In other words, the loading dock can only be called "idle" if no trains are present.

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 this system for 1 million (1,000,000) 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. The population average and maximum of the time-in-system over trains.
3. The percentage of time the loading dock spent busy, idle, and hogged-out. Do these 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 1e6 # 1e6 = 1 million hours
```

or

```
$ ./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 1e6`", or Java "`java -cp . train 10.0 1e6`") 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`" containing:

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

That will allow the grader to run your program using the same syntax as above:

```
$ ./train 10.0 1000000
```

(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`")

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: *inter-arrival-time*, *unloading time*, and *remaining crew hours* (in that order), with each arrival event on a new line:

```
0.02013 3.70 8.92
12.12 4.12 10.10
8.52 3.98 7.82
... etc ...
```

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
... etc ...
```

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 1,000,000 hours), but don't stop the simulation until the last train has departed.

Output Specification: Your program should print one line for every event that gets called. We want to be able to follow what's happening in your code. Each train and each crew should be assigned an incrementing integer ID . The final statistics should come after the simulation output and closely match the specified format. Output lines should resemble the following example (lines have been split here just for human readability but shouldn't be in your output):

```
Time 10.03: train 0 arrival for 4.11h of unloading, crew 0 with 9.81h before hogout (Q=0)
Time 10.03: train 0 entering dock for 4.11h of unloading, crew 0 with 9.81h before hogout
Time 14.14: train 0 departing (Q=0)
Time 26.13: train 1 arrival for 4.24h of unloading, crew 1 with 7.49h before hogout (Q=0)
Time 26.13: train 1 entering dock for 4.24h of unloading, crew 1 with 7.49h before hogout
Time 28.94: train 2 arrival for 4.42h of unloading, crew 2 with 6.96h before hogout (Q=0)
Time 30.37: train 1 departing (Q=1)
Time 30.37: train 2 entering dock for 4.42h of unloading, crew 2 with 5.54h before hogout
Time 34.79: train 2 departing (Q=0)
…
Time 58.34: train 7 entering dock for 4.11h of unloading, crew 7 with 0.12h before hogout
Time 58.46: train 7 crew 7 hogged out during service (SERVER HOGGED)
Time 61.81: train 7 replacement crew 8 arrives (SERVER UNHOGGED)
Time 65.80: train 7 departing (Q=0)
…
Time 717.31: train 57 crew 59 hogged out during service (SERVER HOGGED)
Time 720.64: train 57 replacement crew 62 arrives (SERVER UNHOGGED)
Time 721.39: train 58 crew 60 hogged out in queue
Time 721.40: train 57 departing (Q=2)
…
Time 7201.55: simulation ended

Statistics
Total number of trains served: 721
Average time-in-system per train: 6.37h
Maximum time-in-system per train: 25.1h
Dock idle percentage: 59.95%
Dock busy percentage: 40.05%
Dock hogged-out percentage: 3.71%
Time average number of trains in queue: 0.205
Maximum number of trains in queue: 4
Histogram of hogout count per train:
[0]: 594
[1]: 122
[2]: 5
... (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 is provided to give feedback on I/O specification compliance. The script is on openlab in the directory /home/wayne/pub/cs115/A1-syntax-checker; the same directory contains sample output, plus my own executable (though its output format has debugging info that produces more output than the spec wants). To test your program's input and output, you can run one of the following commands, ensuring that you are located in your own directory containing your train executable. Note that the format checker ***takes no arguments***: instead, it expects to read the output of your program on its *standard input stream* (look up that term if you're not familiar with Linux/Unix). Thus, you could use a *pipe* as follows, assuming your program is in the current directory and is run starting with the command "train" (though other reasonable possibilities include "train.sh" or "python3 train.py"):

```
  $ ./train 10 1e6 | ~wayne/pub/cs115/A1-syntax-checker/train_format_checker.py
```

If you are not too familiar with Unix, the above is functionally equivalent to capturing your output in a temporary file called "output.txt" and then running the syntax checker on that file (this time assuming your program is in python):

```
  $ python3 train.py 10 1e6 > output.txt
  $ cat output.txt | ~wayne/pub/cs115/A1-syntax-checker/train_format_checker.py
or
  $ ~wayne/pub/cs115/A1-syntax-checker/train_format_checker.py  < output.txt
```

If you want to test the checker on *my* sample output, type the following all on one line without pressing the Enter/Return key (it's on two lines here simply because it's too long to display on one line in the PDF):

```
   $ ~wayne/pub/cs115/A1-syntax-checker/train_format_checker.py <
                                ~wayne/pub/cs115/A1-syntax-checker/output.txt
```

If your output violates the specification as checked by the above checker in a way that's easy for the grader to fix, you may or may not lose some points (depending on how annoyed the grader gets). However, if it violates the spec so badly that we can't figure out how to test your output for correctness, *we reserve the right to give you a zero.*

Submission: Submit a brief write-up of your results on GradeScope.com, along with your source code and specific sections of the output (only the first few pages and the last few dozen lines containing the statistics) for the default parameters. There should be no more than 10 pages. Be sure to also submit the PDF, along with your source code, electronically via the submit command on ICS openlab (see syllabus for details on the submit command).

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 the openlab Linux machines (for example, odin.ics.uci.edu). 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 Python), 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%.

Notes from problems running codes from previous years (READ THIS if you use Python):

Please don't use "pip" or "pip3" for *anything* unless *absolutely* necessary. Openlab has tons of version of Python, and tons of libraries. You can see which versions of Python by typing "module avail python"; it says:

```
------------ /pkg/modules/modulefiles ------------
python/2.7.10-tf     python/3.10          python/3.6.1
python/2.7.11        python/3.10.0        python/3.6.7
python/2.7.12        python/3.2.1         python/3.6.8
python/2.7.14        python/3.3.3         python/3.7.1
python/2.7.2         python/3.3.5         python/3.7.2
python/2.7.4         python/3.5.1         python/3.7.4(default)
python/2.7.8         python/3.5.2
```

Most packages you may want to intall (like NumPy, SciPy, etc) are probably part of Anaconda, which also has many versions available on openlab: "module avail conda" gives:

```
------------ /pkg/modules/modulefiles ------------
conda/2.5.0          conda/2020.02         conda/5.2.0
conda/2018.12          conda/2020.11(default)
conda/2019.07          conda/2022.10
```

To load a particular version of Python, use "module load python/XXXX"; to load a particular version of conda, use "module load conda/YYYY".

If I email you to ask for help running your program, please don't tell me to "pip install" something; doing that makes my life *harder*; experience shows that if I "pip install" something to allow your code to run, then some other student's code will break. Please use "module load" appropriately as above, and also add the appropriate "module load" commands to your shell script so I don't have to do it manually.