This homework covers OSTEP chapter 9. It is due in class Friday, February 20. Hand in a hardcopy of your answers in class.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Preliminaries

- Copy the directory `cpu-sched-lottery` (and its contents) from `/classes/cs331` to your `~/cs331` directory. (For this part you won't be writing any code, so the `cpu-sched-lottery` directory doesn't need to go into your `workspace` directory.)
- The `cpu-sched-lottery` directory contains a python program (`lottery.py` and a `README` with instructions for using the program. You can also find the text of the `README` in the appendix section A at the end of this document.

Exercises

Chapter 9 homework problems (at the very end of chapter 9) —

1. Do #1 for practice — make sure you understand the concepts (how the lottery scheduler works) and how to run and use `lottery.py`, but you don't need to hand in your solutions. (Remember that you can check your work by running `lottery.py` with the `-c` option.) Read through the example in the `README` to understand what "compute the solutions for simulations" means.

For the rest of the chapter 9 problems, run `lottery.py` with the `-c` option to see how the jobs are scheduled rather than working it out for yourself.

2. Do #2. Run `lottery.py` once as specified, then reason about the scheduler behavior in order to answer the questions rather than answering them empirically. However, if you do want to experiment with running `lottery.py` multiple times to observe the results, remember that the random seed determines the ticket numbers picked — you need to specify different seeds each time or else it will always do exactly the same thing.

3. Do #3–4. Use the two-job fairness metric from section 9.4: F is the completion time of the first job to finish divided by the completion time of the second job to finish. Hint: a brief dive into shell scripting can greatly facilitate gathering the data to answer these questions. See the appendix section B for more information.

A lottery.py

This program, lottery.py, allows you to see how a lottery scheduler works. As always, there are two steps to running the program. First, run without the -c flag: this shows you what problem to solve without revealing the answers.

```
prompt> ./lottery.py -j 2 -s 0
...
Here is the job list, with the run time of each job:
  Job 0 ( length = 8, tickets = 75 )
  Job 1 ( length = 4, tickets = 25 )

Here is the set of random numbers you will need (at most):
Random 511275
Random 404934
Random 783799
Random 303313
Random 476597
Random 583382
Random 908113
Random 504687
Random 281838
Random 755804
Random 618369
Random 250506
```

When you run the simulator in this manner, it first assigns you some random jobs (here of lengths 8, and 4), each with some number of tickets (here 75 and 25, respectively). The simulator also gives you a list of random numbers, which you will need to determine what the lottery scheduler will do. The random numbers are chosen to be between 0 and a large number; thus, you'll have to use the modulo operator to compute the lottery winner (i.e., winner should equal this random number modulo the total number of tickets in the system).

Running with -c shows exactly what you are supposed to calculate:

```
prompt> ./lottery.py -j 2 -s 0 -c
...
** Solutions **
Random 511275 -> Winning ticket 75 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:8 tix:75 ) (* job:1 timeleft:4 tix:25 )
Random 404934 -> Winning ticket 34 (of 100) -> Run 0
  Jobs:  (* job:0 timeleft:8 tix:75 ) (  job:1 timeleft:3 tix:25 )
Random 783799 -> Winning ticket 99 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:7 tix:75 ) (* job:1 timeleft:3 tix:25 )
```

```
Random 303313 -> Winning ticket 13 (of 100) -> Run 0
  Jobs:  (* job:0 timeleft:7 tix:75 ) (  job:1 timeleft:2 tix:25 )
Random 476597 -> Winning ticket 97 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:6 tix:75 ) (* job:1 timeleft:2 tix:25 )
Random 583382 -> Winning ticket 82 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:6 tix:75 ) (* job:1 timeleft:1 tix:25 )
--> JOB 1 DONE at time 6
Random 908113 -> Winning ticket 13 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:6 tix:75 ) (  job:1 timeleft:0 tix:--- )
Random 504687 -> Winning ticket 12 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:5 tix:75 ) (  job:1 timeleft:0 tix:--- )
Random 281838 -> Winning ticket 63 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:4 tix:75 ) (  job:1 timeleft:0 tix:--- )
Random 755804 -> Winning ticket 29 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:3 tix:75 ) (  job:1 timeleft:0 tix:--- )
Random 618369 -> Winning ticket 69 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:2 tix:75 ) (  job:1 timeleft:0 tix:--- )
Random 250506 -> Winning ticket 6 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:1 tix:75 ) (  job:1 timeleft:0 tix:--- )
--> JOB 0 DONE at time 12
```

As you can see from this trace, what you are supposed to do is use the random number to figure out which ticket is the winner. Then, given the winning ticket, figure out which job should run. Repeat this until all of the jobs are finished running. It's as simple as that – you are just emulating what the lottery scheduler does, but by hand!

Just to make this absolutely clear, let's look at the first decision made in the example above. At this point, we have two jobs (job 0 which has a runtime of 8 and 75 tickets, and job 1 which has a runtime of 4 and 25 tickets). The first random number we are given is 511275. As there are 100 tickets in the system, 511275 \% 100 is 75, and thus 75 is our winning ticket.

If ticket 75 is the winner, we simply search through the job list until we find it. The first entry, for job 0, has 75 tickets (0 through 74), and thus does not win; the next entry is for job 1, and thus we have found our winner, so we run job 1 for the quantum length (1 in this example). All of this is shown in the print out as follows:

```
Random 511275 -> Winning ticket 75 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:8 tix:75 ) (* job:1 timeleft:4 tix:25 )
```

As you can see, the first line summarizes what happens, and the second simply shows the entire job queue, with an * denoting which job was chosen.

The simulator has a few other options, most of which should be self-explanatory. Most notably, the -l/–jlist flag can be used to specify an exact set of jobs and their ticket values, instead of always using randomly-generated job lists.

```
prompt> ./lottery.py -h
Usage: lottery.py [options]

Options:
  -h, --help
      show this help message and exit
  -s SEED, --seed=SEED
      the random seed
  -j JOBS, --jobs=JOBS
      number of jobs in the system
  -l JLIST, --jlist=JLIST
      instead of random jobs, provide a comma-separated list
      of run times and ticket values (e.g., 10:100,20:100
      would have two jobs with run-times of 10 and 20, each
      with 100 tickets)
  -m MAXLEN, --maxlen=MAXLEN
      max length of job
  -T MAXTICKET, --maxtick=MAXTICKET
      maximum ticket value, if randomly assigned
  -q QUANTUM, --quantum=QUANTUM
      length of time slice
  -c, --compute
      compute answers for me
```

B A Mini Introduction to `bash`

`bash` (the shell running in your terminal window) can do more than just run single-line commands — you can, for example, write loops and conditionals. For example: (type this all on one line in the terminal; it is written on multiple lines below for readability)

```
for f in {1..10} ; do
  ./lottery.py -l 100:100,100:100 -c -s $f ;
done
```

What is going on: `f` is a loop variable that counts through the values 1, 2, 3, ..., 10; `$f` references the value of `f`. The loop above is equivalent to:

```
./lottery.py -l 100:100,100:100 -c -s 1
./lottery.py -l 100:100,100:100 -c -s 2
./lottery.py -l 100:100,100:100 -c -s 3
./lottery.py -l 100:100,100:100 -c -s 4
./lottery.py -l 100:100,100:100 -c -s 5
./lottery.py -l 100:100,100:100 -c -s 6
./lottery.py -l 100:100,100:100 -c -s 7
./lottery.py -l 100:100,100:100 -c -s 8
./lottery.py -l 100:100,100:100 -c -s 9
./lottery.py -l 100:100,100:100 -c -s 10
```

Now, `lottery.py` produces a lot of output, and for the two problems in question, the only information needed is when each job completes. These lines have the form

```
--> JOB 0 DONE at time 192
```

in the output. Since they are the only ones containing the string "DONE", `grep` can be used to pick out just those lines:

```
for f in {1..10} ; do
  ./lottery.py -l 100:100,100:100 -c -s $f | grep "DONE" ;
done
```

The vertical bar `|` is a pipe, as seen in class — it redirects the output from `lottery.py` to be the input for `grep`.

As a final step, it would be nice to separate the output from each run of `./lottery.py`. Printing a blank line after each run does this:

```
for f in {1..10} ; do
  ./lottery.py -l 100:100,100:100 -c -s $f | grep "DONE" ;
  echo ;
done
```

`echo` echoes its argument followed by a newline to standard output; without an argument, just a blank line is printed.