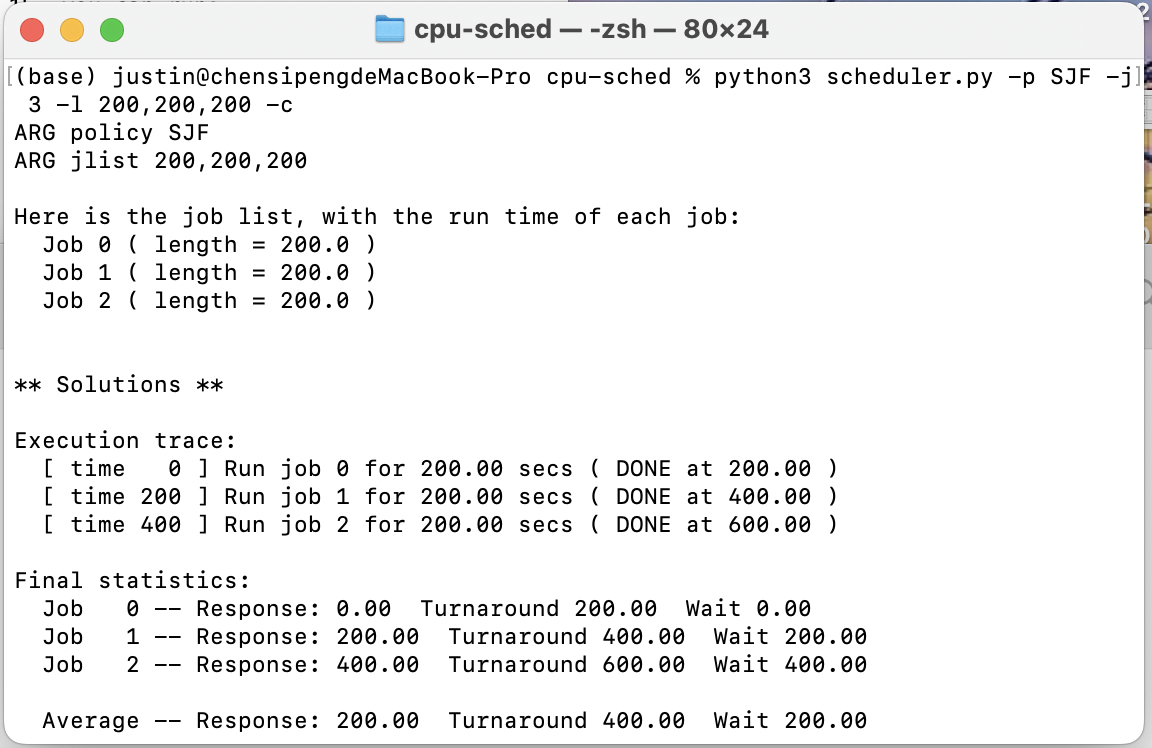
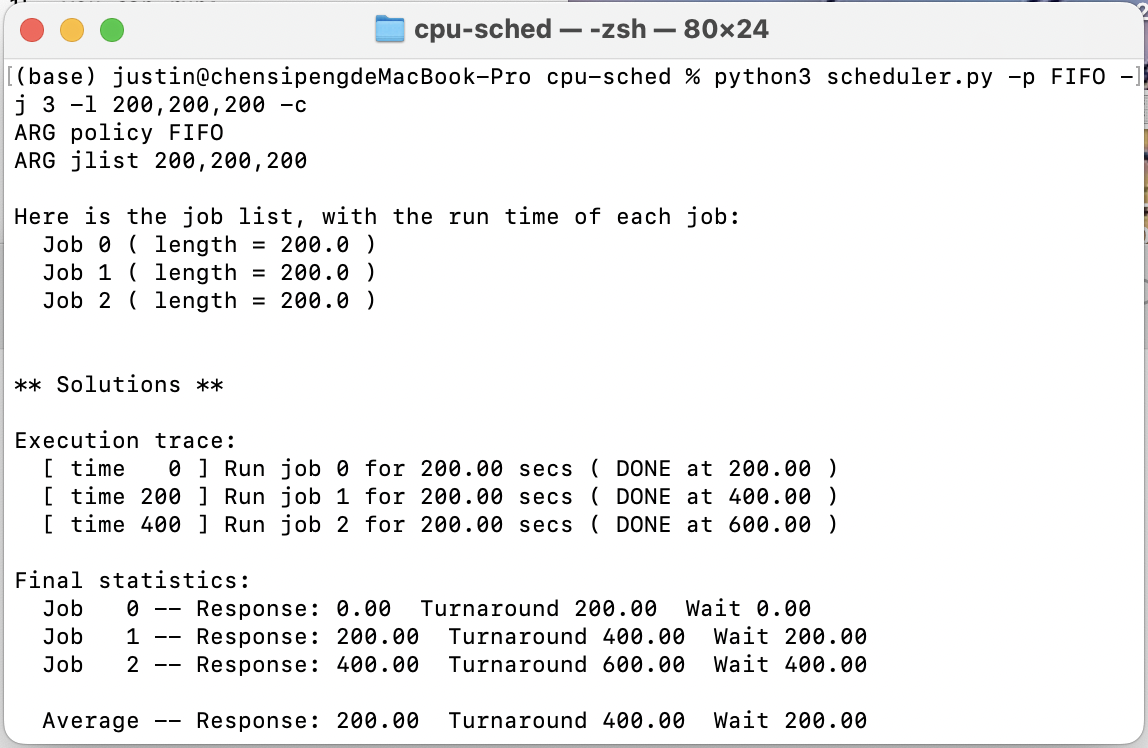
**Cpt 7**

**Homework (Simulation)**

1. **Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.**

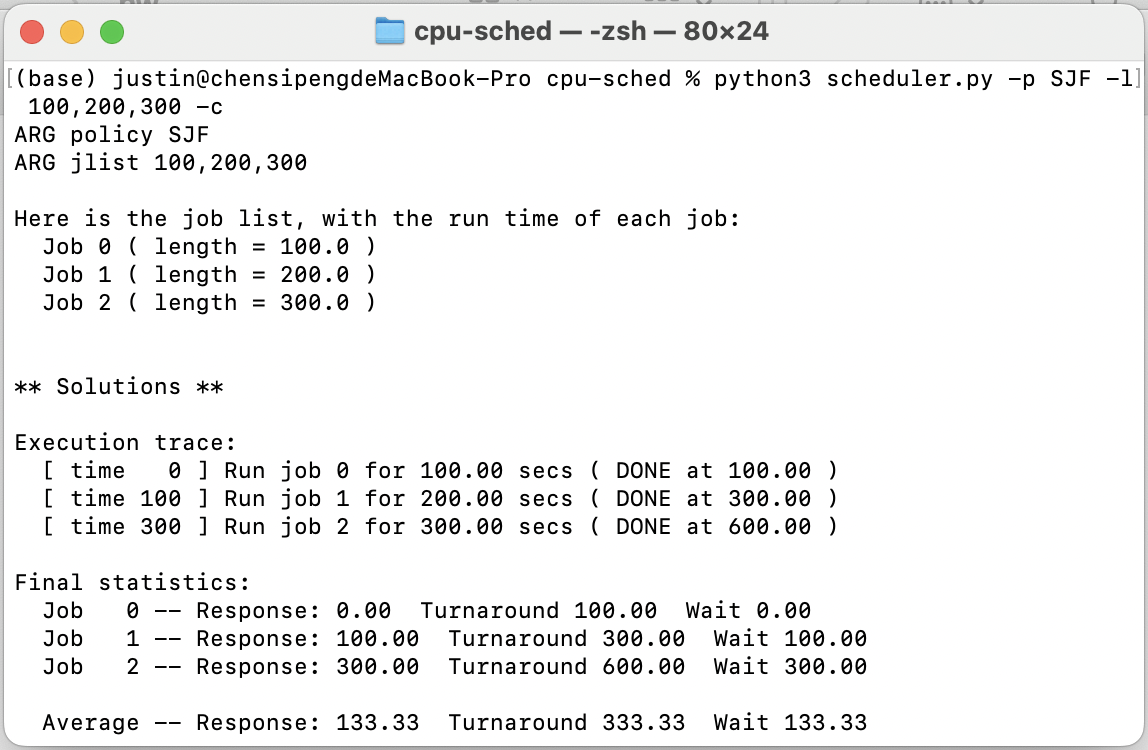
****

SJF

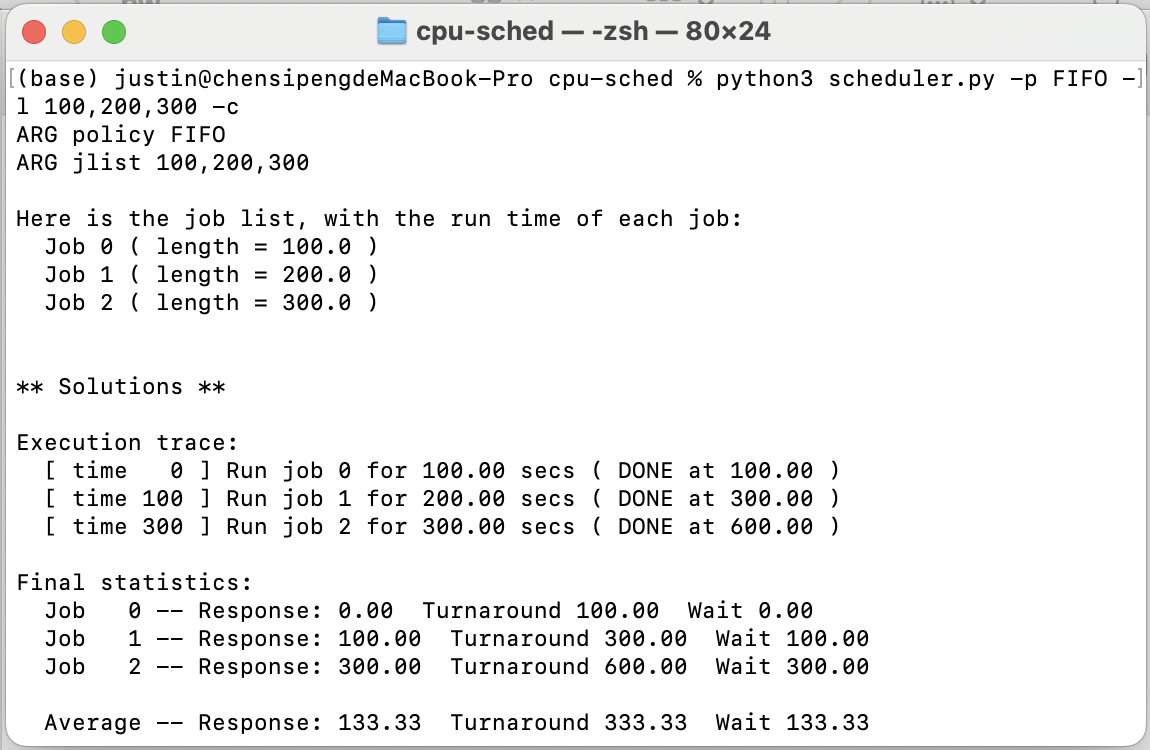
****

FIFO

1. **Now do the same but with jobs of different lengths: 100, 200, and 300.**

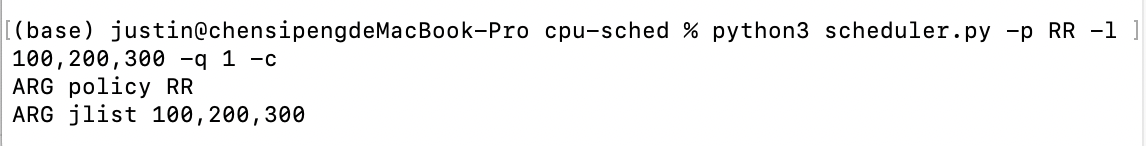
****

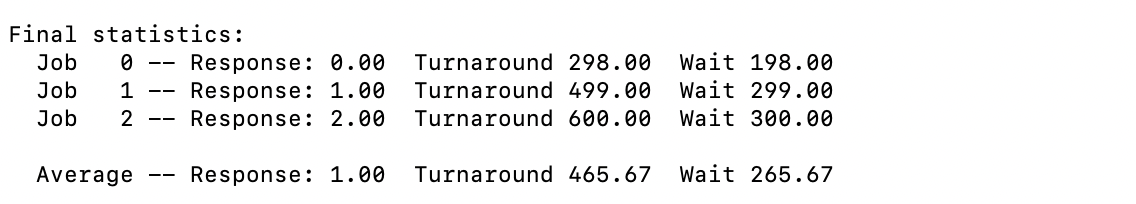
SJF

****

FIFO

1. **Now do the same, but also with the RR scheduler and a time-slice of 1.**

****

****

1. **For what types of workloads does SJF deliver the same turnaround times as FIFO?**

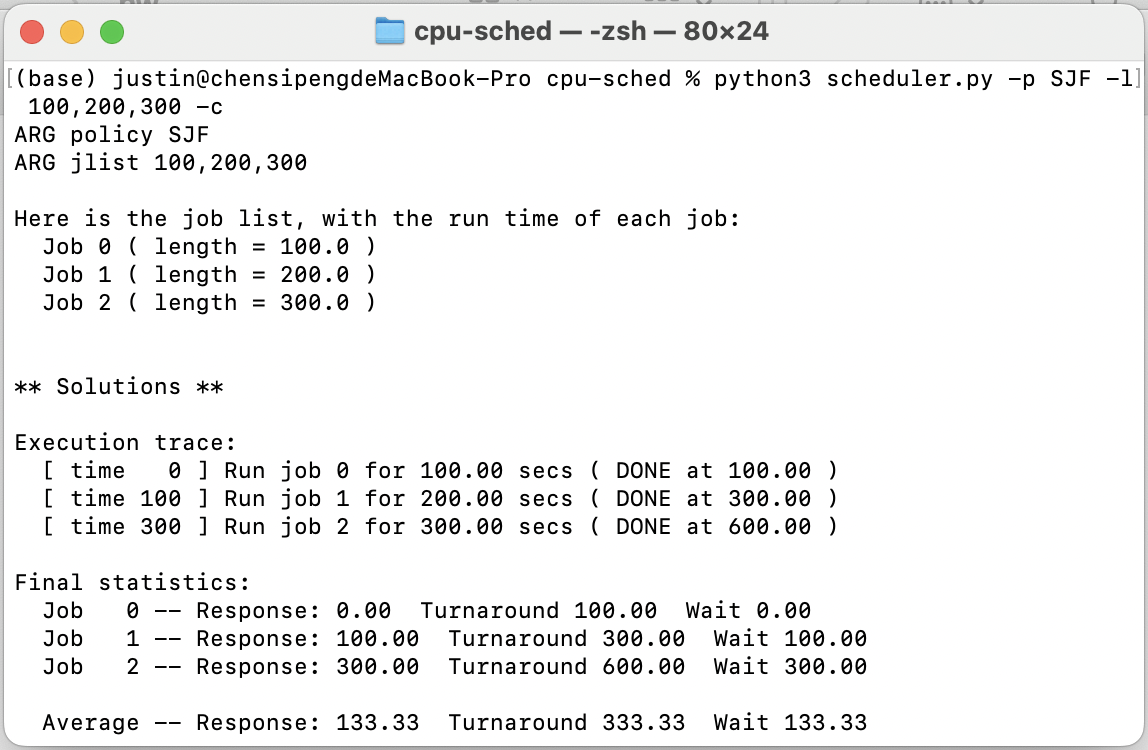
Tasks should come in in the exact order of task length from the least time-consuming to the most time-consuming one, in which case SJF will carry out an identical execution with that of FIFO. The order of tasks will be irrelevant for tasks that consume the same amount of time.

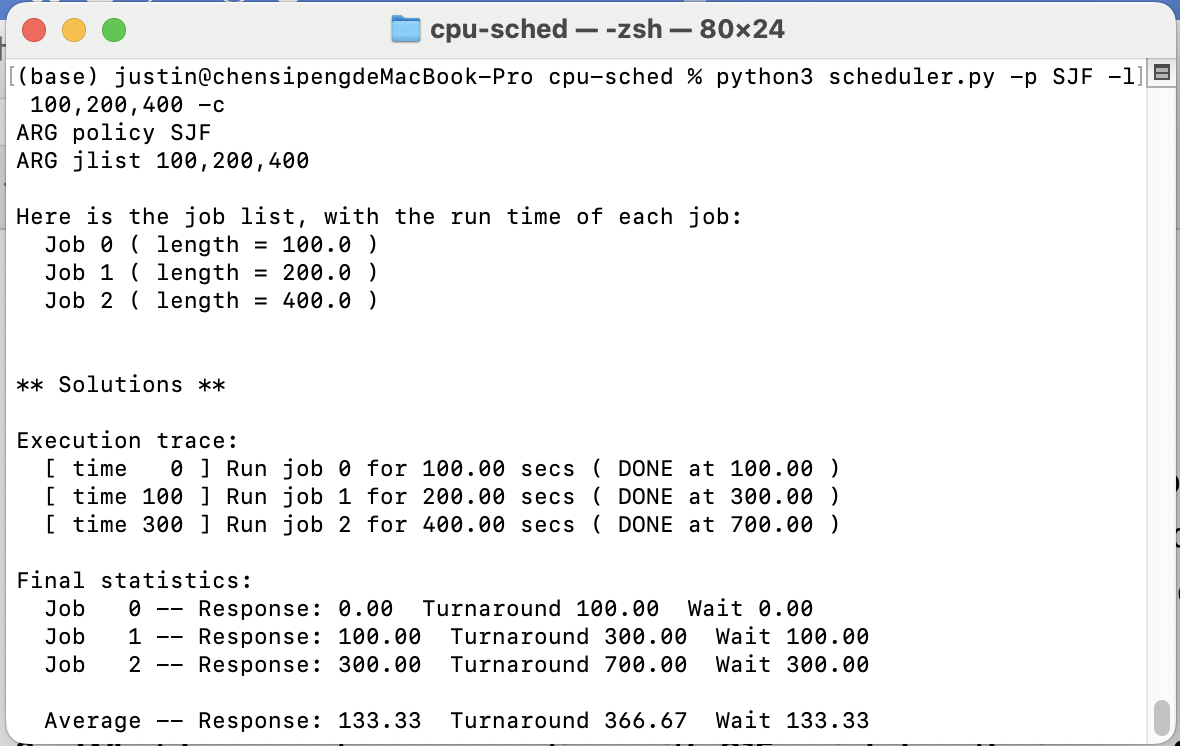
1. **For what types of workloads and quantum lengths does SJF deliver the same response times as RR?**

Generally speaking, the average response time of RR is always the amount of its time slice. Thus, if the response times are to be completely the same, tasks need to have identical lengths, and this length should also be equal to the amount of the time slice of RR.

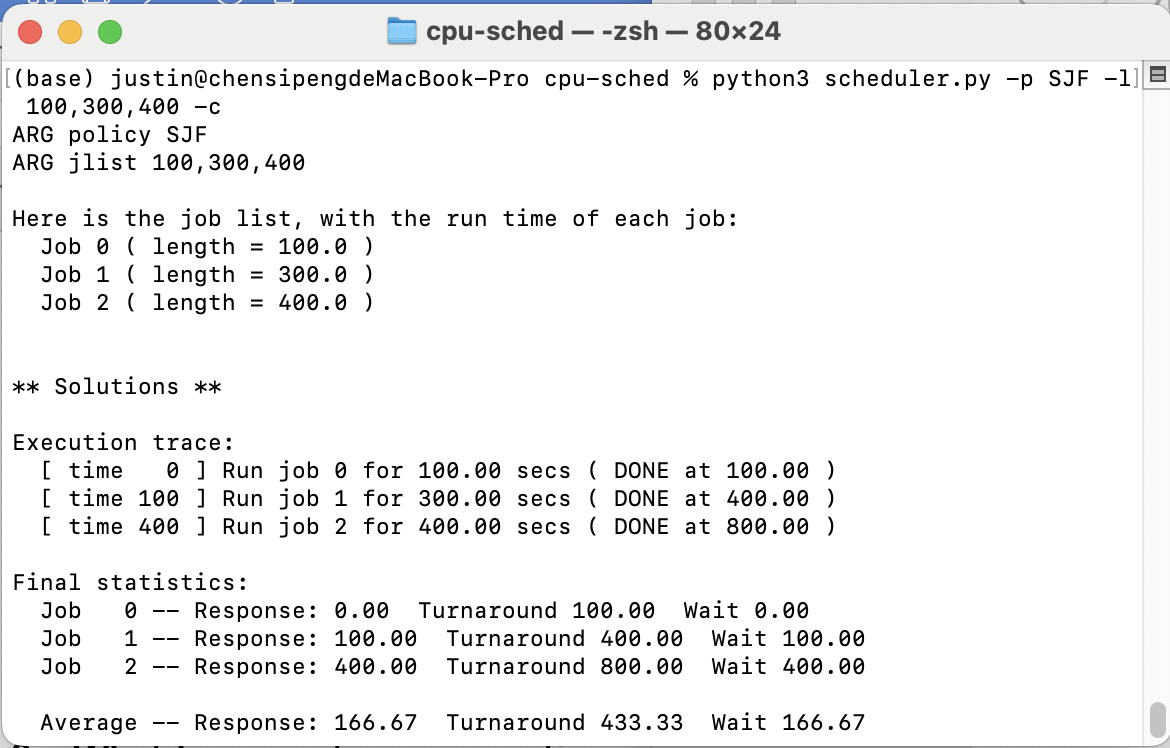
1. **What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?**

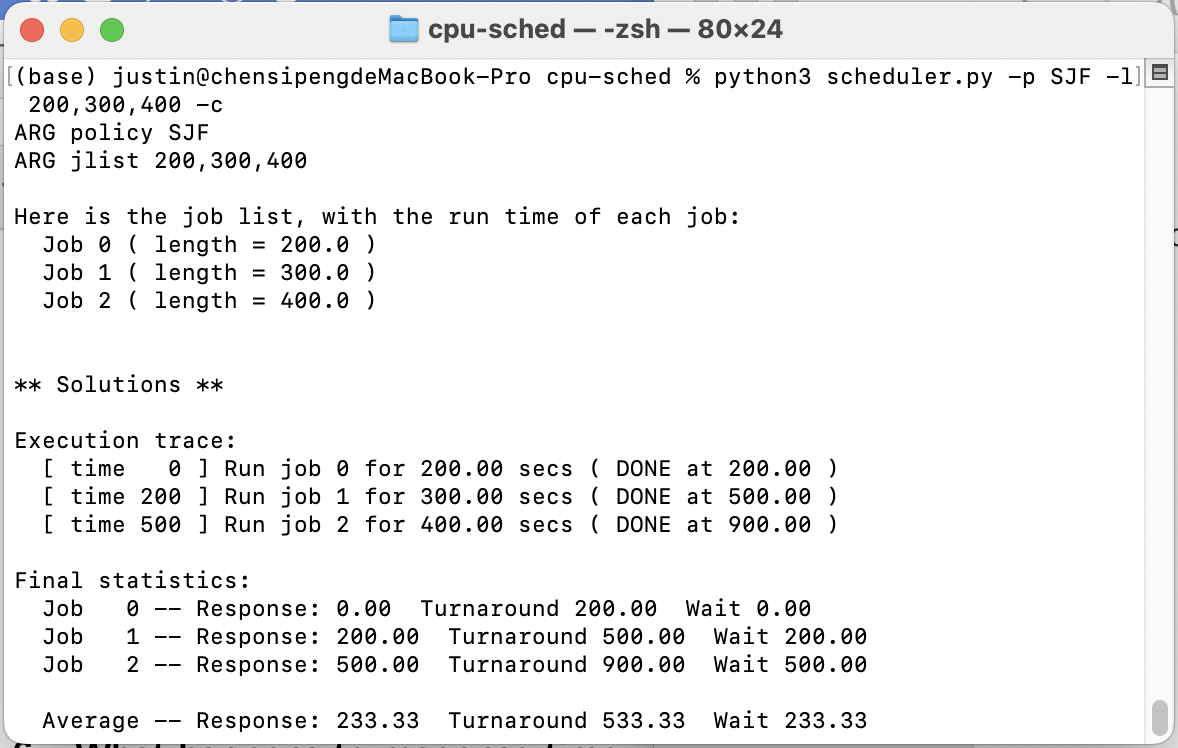
The average response time with SJF will remain identical if we increase the job length of the task the most time-consuming.





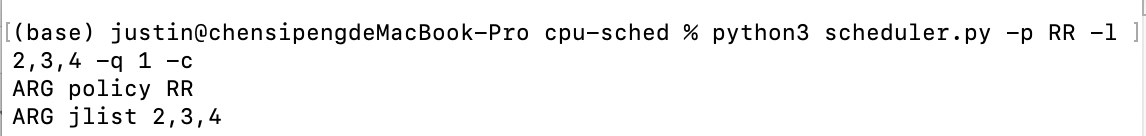
The two screenshots above shows the execution and statistics of jobs with lengths 100, 200, 300 and jobs with lengths 100, 200, 400 respectively. We can see the average response time of these two cases are both 133.33.

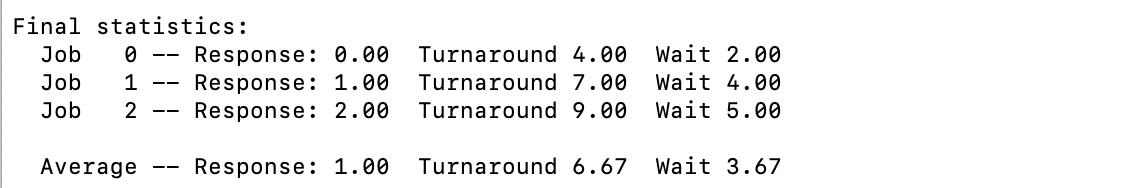


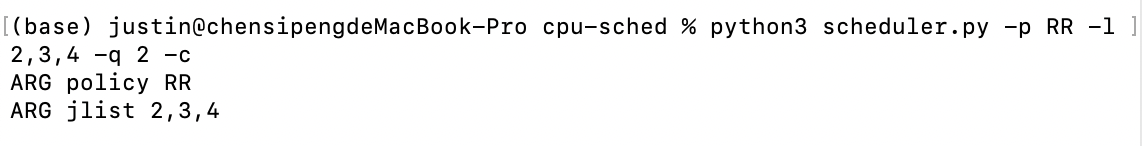


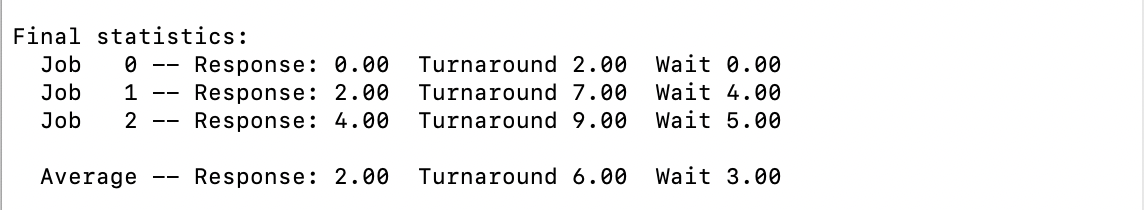
However, if we increase the length of the second task by 100, the average response time increases to 166.67. If we take a step further and increase the length of the first task by 100, the average response increases again, this time to 233.33. This is because the average response time relies on the length of the all the tasks except the last one to be carried out, and in the case of SJF the last task is always the longest one, which is coherent with what we observed from the trials above.

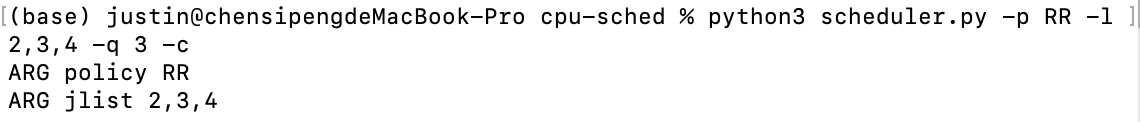
1. **What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?**

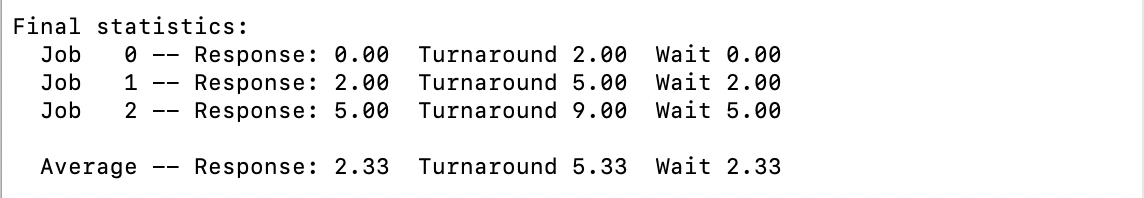
****

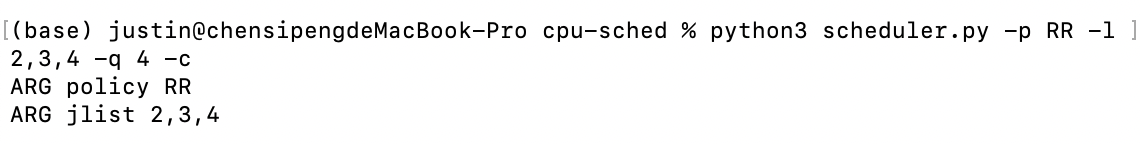
****

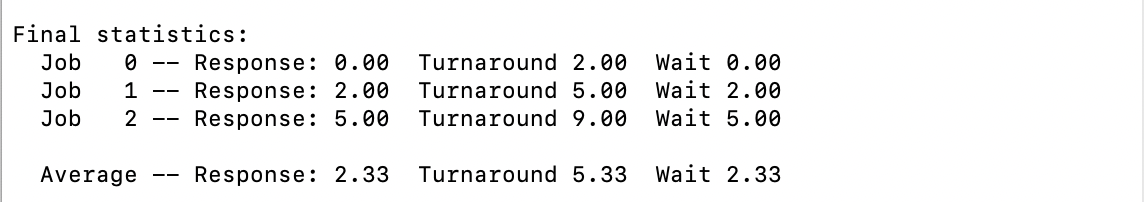
****

****

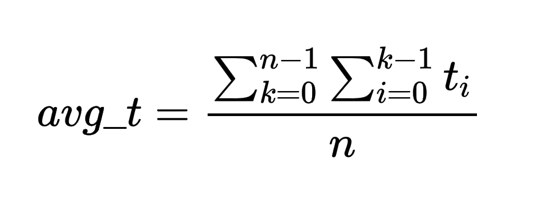
****

****

****

****

The above screenshots show the statistics of the execution of tasks with lengths 2, 3, 4 and quantum lengths from 1 to 4. The average response times are 1.00, 2.00, 2.33 and 2.33 respectively. We can see that RR has the worst response time when the quantum length is larger than all tasks except the last one, in which case RR is going to operate exactly the same as a regular FIFO will do. The exact response time can be computed with the formula below:

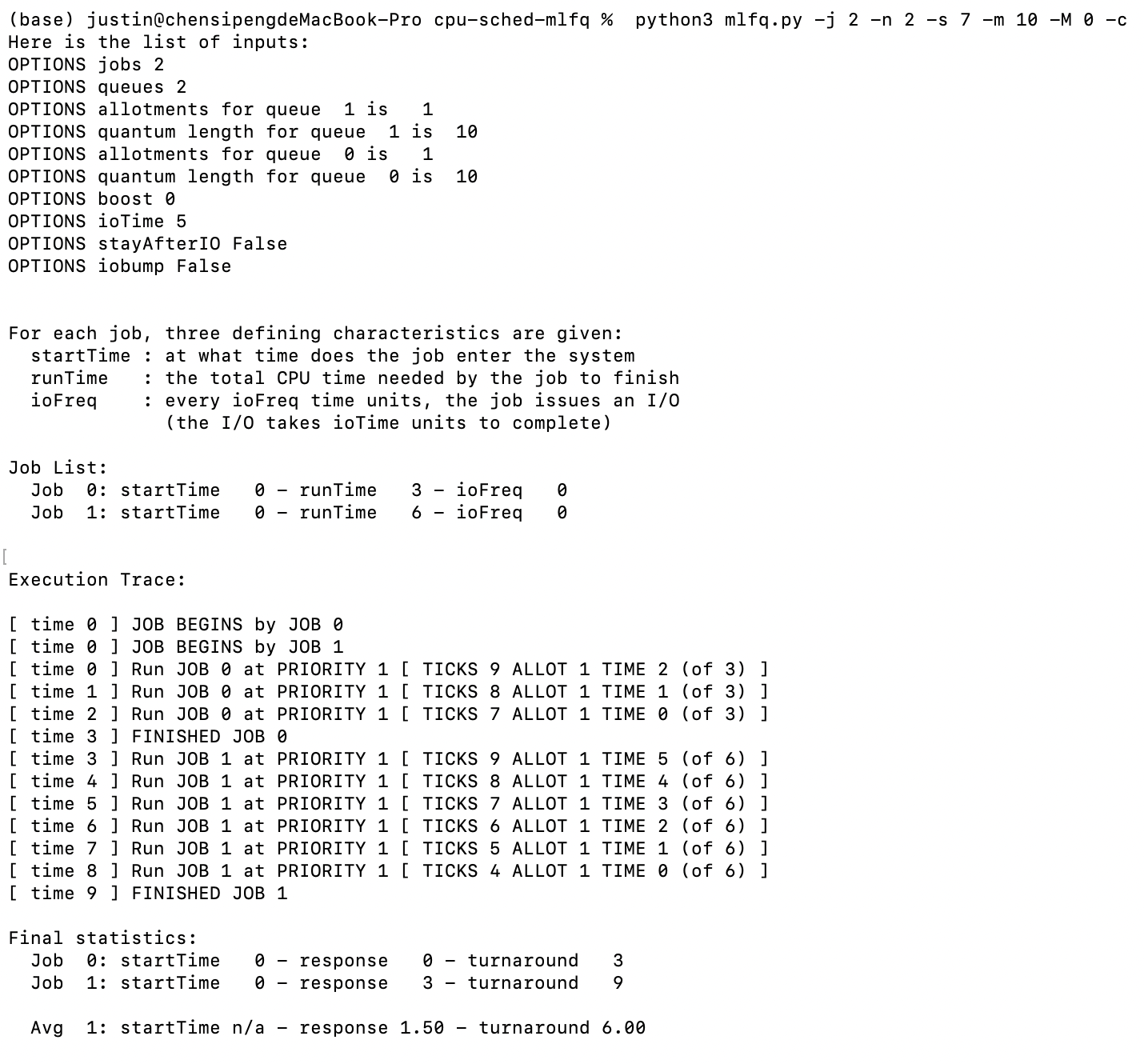


where ti stands for the length of the i-th task and n stands for the total number of tasks to be carried out.

**Cpt 8**

**Homework (Simulation)**

1. **Run a few randomly generated problems with just two jobs and two queues; compute the MLFQ execution trace for each. Make your life easier by limiting the length of each job and turning off I/Os.**

****

1. **How would you run the scheduler to reproduce each of the examples in the chapter?**

Figure 8.2: python3 mlfq.py -q 10 -n 3 -j 1 -l 0,200,0 -c

Figure 8.3: python3 mlfq.py -q 10 -n 3 -l 0,200,0:100,20,0 -j 2 -c

Figure 8.4: python3 mlfq.py -q 10 -n 3 -l 0,200,0:50,15,1 -i 10 -j 2 -S -c

Figure 8.5: python3 mlfq.py -l 0,200,0:100,50,1:100,50,1 -i 1 -c -S (no boost)

python3 mlfq.py -l 0,200,0:100,50,1:100,50,1 -i 1 -c -S -B 50 (boosted)

Figure 8.6: python3 mlfq.py -l 0,200,0:100,100,9 -i 1 -S -c (no gaming tolerance)

python3 mlfq.py -l 0,200,0:100,100,9 -i 1 -c (with gaming tolerance)

Figure 8.7: python3 mlfq.py -l 0,200,0:0,200,0 -Q 10,20,50 -c

1. **How would you configure the scheduler parameters to behave just like a round-robin scheduler?**

Simply setting only one job queue and ruling IO out of consideration will do.

1. **Craft a workload with two jobs and scheduler parameters so that one job takes advantage of the older Rules 4a and 4b (turned on with the -S flag) to game the scheduler and obtain 99% of the CPU over a particular time interval.**

python3 mlfq.py -l 0,200,0:0,200,99 -q 100 -i 1 -S -c

We can do this by setting the quantum slice 100 ms and let the second task issues an I/O request every 99 ms so that it will relinquish the CPU just when the time slice is about to end. Also, an I/O action should only last for 1 ms so that the arbitrary second task will step right in when the first task which has priority 0 runs for 1 ms, thus taking up 99% of the CPU within a single quantum slice.

1. **Given a system with a quantum length of 10 ms in its highest queue, how often would you have to boost jobs back to the highest priority level (with the -B flag) in order to guarantee that a single long-running (and potentially-starving) job gets at least 5% of the CPU?**

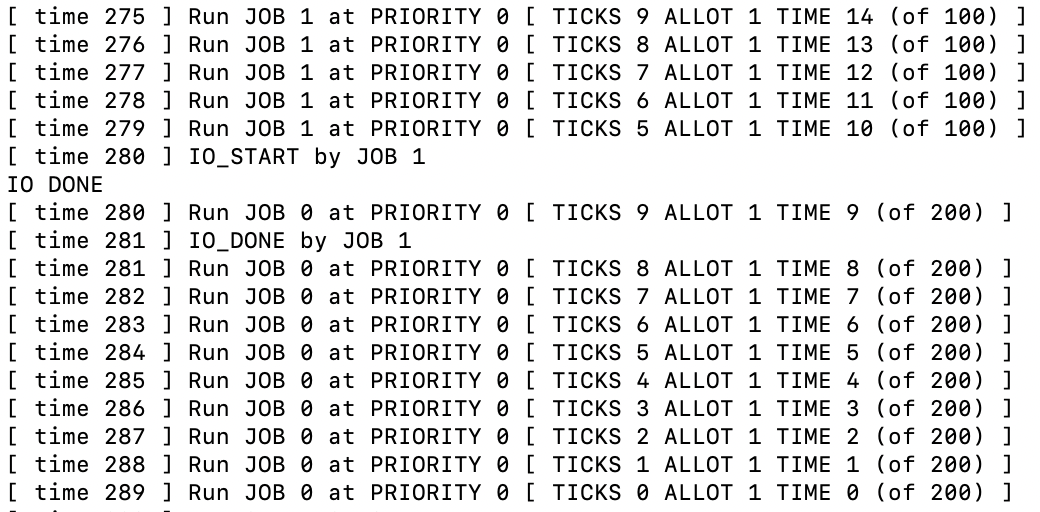
This single long-running task will have the opportunity to run for 10 ms every time it’s boosted, and then, in the worst case, will be deprived of its control over the CPU by some arbitrary tasks that use I/O to trick the CPU. In this sense, the interval between two priority boosts should be 10/5% = 200 ms.

1. **One question that arises in scheduling is which end of a queue to add a job that just finished I/O; the -I flag changes this behavior for this scheduling simulator. Play around with some workloads and see if you can see the effect of this flag.**

Take the following scheduling for instance:

python3 mlfq.py -l 0,200,0:100,100,15 -i 1 -S -c

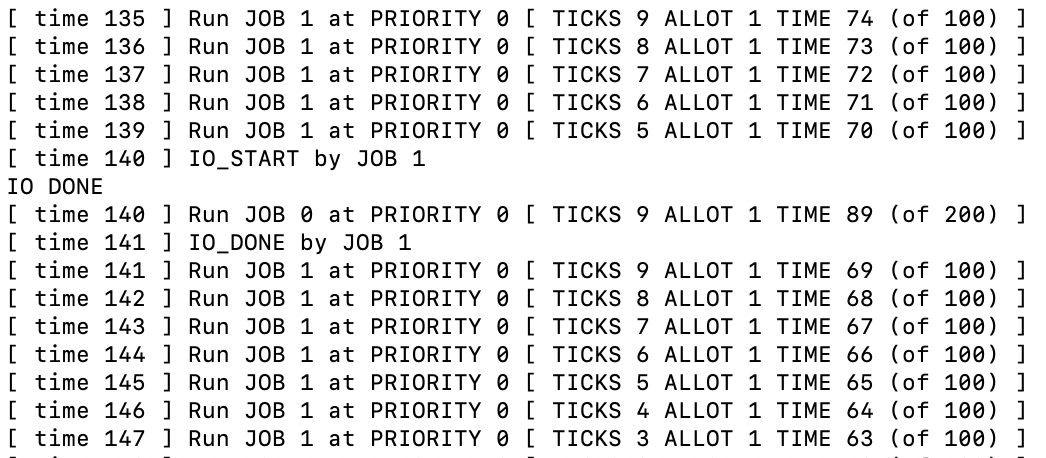
There is, as always, an honest long-running task, and one that cuts in at the 100th ms which issues an I/O request every 15 ms. Since the interval between every two I/O requests exceed the length of the quantum slice, the priority of the second task will gradually degrade until finally reaching the ground of priority 0, which is the same as the priority of the first task. Thus, when the second task finishes its I/O operation, the first task will continue running as they share the same priority level.



However, if we add the -I flag at the end of the command, which makes the command look like this:

python3 mlfq.py -l 0,200,0:100,100,15 -i 1 -S -c -I

the second task will be able to kick in as soon as it finishes its I/O.



This indicates that the -I flag will ensure that when it comes to jobs with the same level of priority, the one that just finishes I/O will always be carried out first.