Q1:
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.

Command: ***python3 mlfq.py -s 1 -n 2 -j 2 -m 20 -M 0***

```
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime   3 - ioFreq   0
  Job  1: startTime   0 - runTime  15 - ioFreq   0

Compute the execution trace for the given workloads.
If you would like, also compute the response and turnaround
times for each of the jobs.

Use the -c flag to get the exact results when you are finished.
```

Answer:

At time 0, job 0 and job 1 are both in priority 1. Job 0 will run first. At time 2 job 0 is done. Notice job 0 use 3 out of 10 time slice, and each time slice has 10 quantum length. At time 3 CPU start run the job 1, at time 12 the job 1 used up entire 10 out of 10 time slice, and each level of the queue has one allotments. Thus, at time 13 the job 1 will demote to priority 0. At time 18, job 1 finished.

Command: ***python3 mlfq.py -s 2 -n 2 -j 2 -m 20 -M 0***

```
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  19 - ioFreq   0
  Job  1: startTime   0 - runTime   2 - ioFreq   0

Compute the execution trace for the given workloads.
If you would like, also compute the response and turnaround
times for each of the jobs.

Use the -c flag to get the exact results when you are finished.
```

Answer:

At time 0, job 0 and job 2 arrive to priority 1. Job 0 first run, at time 9 the job 0 used up all the allotments and get demoted to the priority 0. At time 10, job 1 start to run. At time 12 job 1 finish the execution. There is no process in priority 1 left, thus, cpu start execute job 0 at priority 0. At time 21, job 0 done. All the jobs done.

Q2:
How would you run the scheduler to reproduce each of the examples in the chapter?
Example 1 Figure 8.2:
Command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0**
This command will run the scheduler in 3 level of priority queues(-n 3), with time slice of 10(-q 10), only one job with start time 0, runtime 200, and 0 I/O chance(-l 0,200,0).

Example 2 Figure8.3:
Command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0:100,20,0**
This command will run the scheduler in 3 level of priority queues(-n 3), with time slice of 10(-q 10), first job start at time 0, runtime 200, and 0 I/O change, and the second job start at time 100, with runtime 20, 0 chance I/O (-l 0,200,0:100,20,0)

Example 3 Figure8.4:
Command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0:50,20,1 -S**
This command will run the scheduler in 3 level of priority queues(-n 3), with time slice of 10(-q 10), first job start at time 0, runtime 200, and 0 I/O change, and the second job start at time 50, with runtime 20, 100% chance I/O (-l 0,200,0:50,20,1). And each time process doing IO will reset allotment(-S).

Figure 8.5:
Left one command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0:100,100,1:100,100,1 -S -i 1**
The job 0 will start at 0 with runtime of 200, and 0 chance of I/O. At time 100 the job 1 and job 2 arrive with run time of 100 and 100% chance of I/O. We set I/O time to 1 (-i 1), and rest allotment each time process doing i/o(-S).

Right one command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0:100,100,1:100,100,1 -S -i 1 -B 50**
The job 0 will start at 0 with runtime of 200, and 0 chance of I/O. At time 100 the job1 and job 2 arrive with run time of 100 and 100% chance of I/O. We set I/O time 1 (-I 1), and rest allotment each time process doing I/O(-s). Every 50 ms all the process pump into priority 2.

Figure 8.6:
Left one command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0:75,100,9 -S -i 1**
The job 0 start at 0 with runtime of 200, 0 chance of I/O. At time 75, the job1 arrive with runtime of 100, and each 9ms will run I/O to reset the allotment (-l 0,200,0:75,100,9), each I/O time is 1 (-i 1), and each I/O will reset allotment(-S)

Right one command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0:75,100,9 -i 1**
The job 0 start at 0 with runtime of 200, 0 chance of I/O. At time 75, the job1 arrive with runtime of 100, and each 9ms will run I/O (-l 0,200,0:75,100,9), each I/O time is 1 (-i 1). This time I/O won't reset the same priority level.

Figure 8.7:
Command: python3 mlfq.py -n 3 -Q 10,20,40 -l 0,150,0:0,150,0 -A 2,2,2
This time we have 3 level priorities, with top one 10 ms time slice, middle 20 ms time slice, bottom 40 ms times slice(-Q 10,20,40), and each level have two allotment(-A 2,2,2). There are two jobs arrive at same time at 0, and with runtime 150, and 0 ms of doing I/O out of it runtime.

Q3:
How would you configure the scheduler parameters to behave just like a round-robin scheduler?

Command: **python3 mlfq.py -n 1 -q 10 -M 0**
In this way we only have one level of queue, and all the job into one level and every 10 ms time slice need change to switch the turn. We also set I/O frequency to 0.

Q4:
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.

Command: **python3 mlfq.py -n 3 -q 10 -l 0,200,0:75,100,9 -S -i 1**
The job 0 start at 0 with runtime of 200, 0 chance of I/O. At time 75, the job1 arrive with runtime of 100, and each 9ms will run I/O to reset the allotment (-l 0,200,0:75,100,9), each I/O time is 1 (-i 1), and each I/O will reset allotment(-S)

In this way one job 1 arrive at time 75, it will first run 9 ms, then at time 84 will run I/O to reset the timeslice, then the job 0 will one ms, the job 1 finish the I/O takeover the CPU time again and so forth. Thus, between time 75 to time186. The job 1 will 99% of the CPU time.

Q5:
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 longrunning (and potentially-starving) job gets at least 5% of the CPU?

so basically this question is asking how long should long running process to wait until have another 10 ms to run? And this proportion should be 5%. We can easily set formula as follow:

$$\frac{10}{x} = 5\%$$
$$\frac{10}{x} = \frac{5}{100}$$
$$x = 200$$

Thus, every 200 ms we should boost the jobs. Our command should be as follow
Command: **python3 mlfq.py -n 3 -q 10 -l 0,2000,0:0,1000,1:0,1000,1 -S -i 1 -B 200**
Job 0 will get at 5% of cpu time. In ever 200ms, it will have 10ms to run on CPU.

Q6:
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.

Command: **python3 mlfq.py -n 1 -q 10 -l 0,10,1:0,30,0 -i 3 -c**
We first try don't use -I flag, with only one level of queue, and two jobs, both jobs at time 0. Job 0 with runtime 10 and all of them is I/O. Job 1 with run time 30, all of them is CPU time. Lastly, we set I/O take 3 ms to finish. In this case we observer that once I/O is finished for job 0, the CPU won't immediately to run the next instruction of job 0. Instead, It will keep running job 1 until the job run out of time slice then it start to run the job 0 again so on and so forth. At the end, job 1 will finish first, and cpu will wait job 0 back from I/O again and again. Average turn around time is 51.5

Command: **python3 mlfq.py -n 1 -q 10 -l 0,10,1:0,30,0 -i 3 -I -c**
This time we use the -I flag, than make job 0 immediately back to cpu and execute next I/O instruction. In this way even though, the cpu utilize more time. The average turnaround time is 38.5. which less than the 51.5