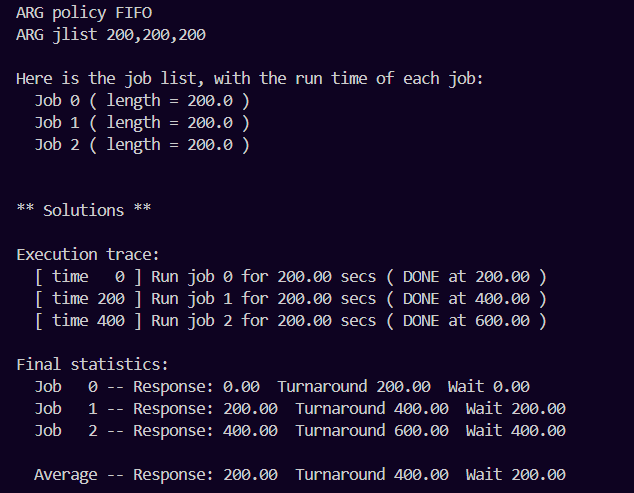
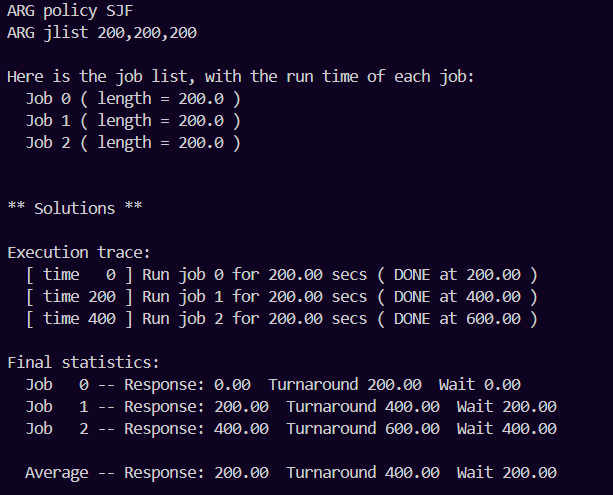
**1. Compute the response time and turnaround time when running**

**three jobs of length 200 with the SJF and FIFO schedulers.**

python ./scheduler.py -p FIFO -j 3 -l 200,200,200 -c



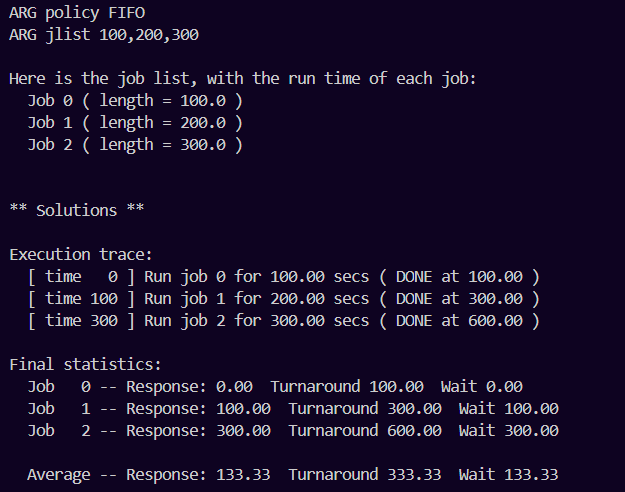
python ./scheduler.py -p SJF -j 3 -l 200,200,200 -c



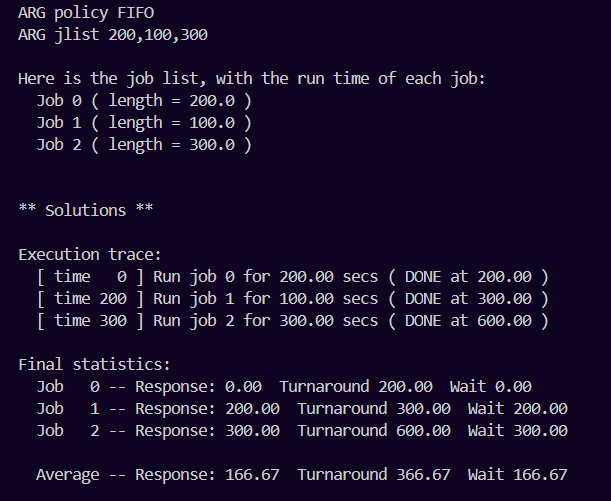
**2. Now do the same but with jobs of different lengths: 100, 200, and**

**300.**

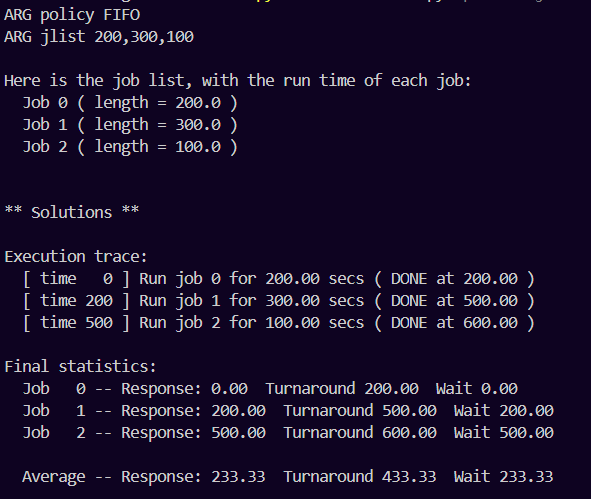
python ./scheduler.py -p FIFO -j 3 -l 100,200,300 -c

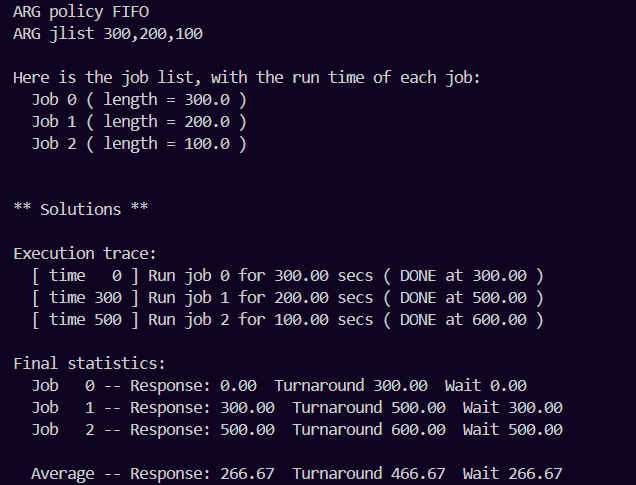


python ./scheduler.py -p FIFO -j 3 -l 200,100,300 -c

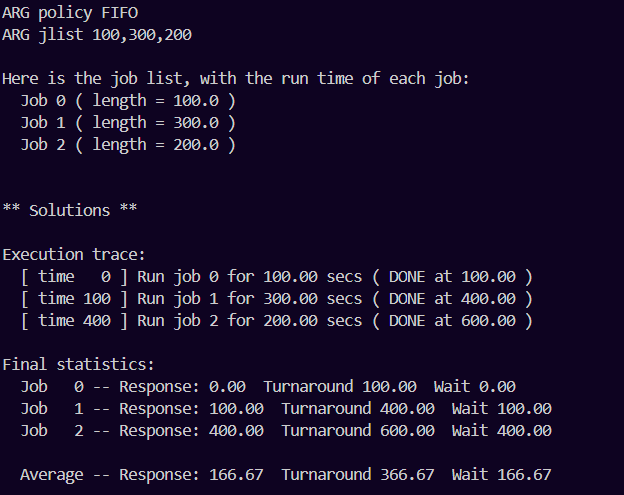


python ./scheduler.py -p FIFO -j 3 -l 200,300,100 -c

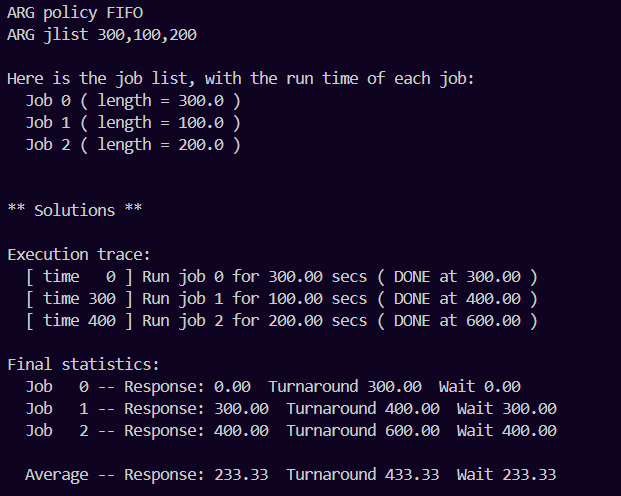


python ./scheduler.py -p FIFO -j 3 -l 300,200,100 -c

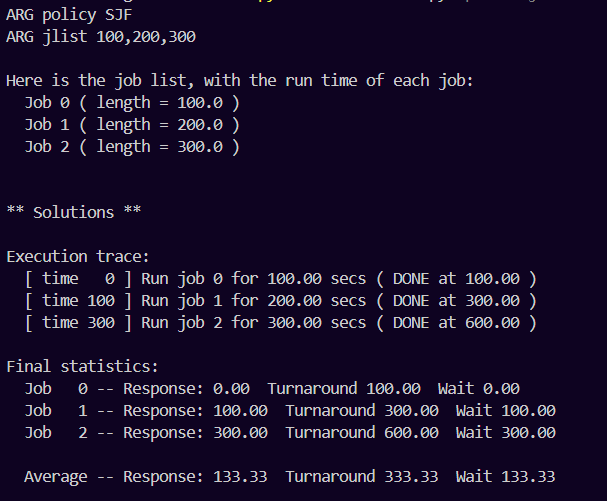
python ./scheduler.py -p FIFO -j 3 -l 100,300,200 -c



python ./scheduler.py -p FIFO -j 3 -l 300,100,200 -c



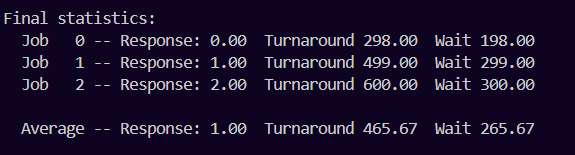
python ./scheduler.py -p SJF -j 3 -l 100,200,300 -c



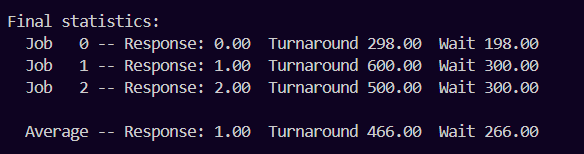
**3. Now do the same, but also with the RR scheduler and a time-slice**

**of 1.**

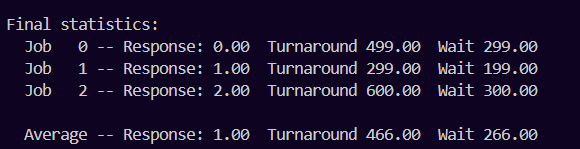
python ./scheduler.py -p RR -q 1 -j 3 -l 100,200,300 -c



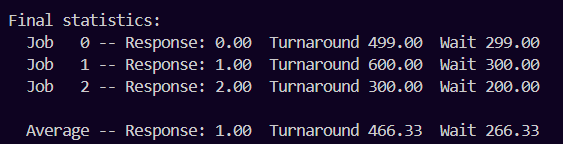
python ./scheduler.py -p RR -q 1 -j 3 -l 100,300,200 -c



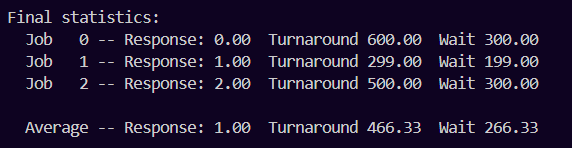
python ./scheduler.py -p RR -q 1 -j 3 -l 200,100,300 -c



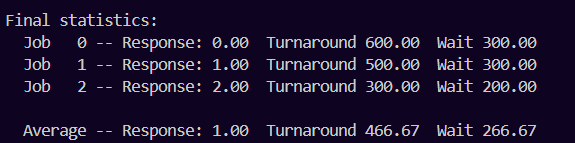
python ./scheduler.py -p RR -q 1 -j 3 -l 200,300,100 -c



python ./scheduler.py -p RR -q 1 -j 3 -l 300,100,200 -c



python ./scheduler.py -p RR -q 1 -j 3 -l 300,200,100 -c



**4. For what types of workloads does SJF deliver the same turnaround**

**times as FIFO?**

1.If jobs come in increasing completion time order. This way both policies work exactly the same.

2. If jobs arrive not simultaneously. This way both policies work exactly the same.

3. if jobs have the same completion time. This way both policies work exactly the same, because SJF differentiates jobs on the basis of their completion times.

**5. For what types of workloads and quantum lengths does SJF deliver**

**the same response times as RR?**

1. If quantum time >= max(completion time) and processes arrive in increasing completion time order. This way RR and SJF work in the same way, because a) they execute jobs in the same order, due to the 2nd condition and b) RR executes each job (just as SJF) without preemption, as quantum time is larger than actual job’s time.

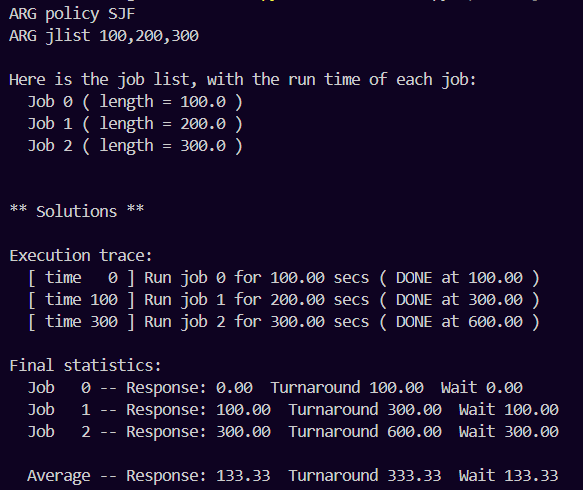
2. if all processes have the same completion time, which equals or is smaller than the quantum length. This way a) SJF calculates processes in any order, even if they arrive at the same time and it actually doesn’t matter in which one, since time is the same for every one of them and b) since SJF calculates all processes to completion (i.e. without preemption), we need to do the same for RR, and 2nd condition grants that.

**6. What happens to response time with SJF as job lengths increase?**

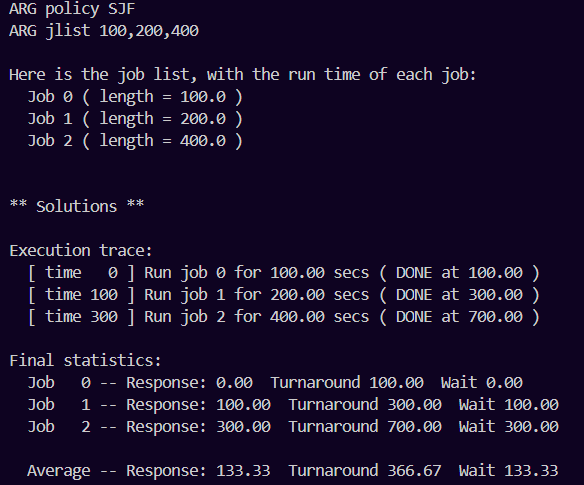
**Can you use the simulator to demonstrate the trend?**

I noticed a couple of tendencies.

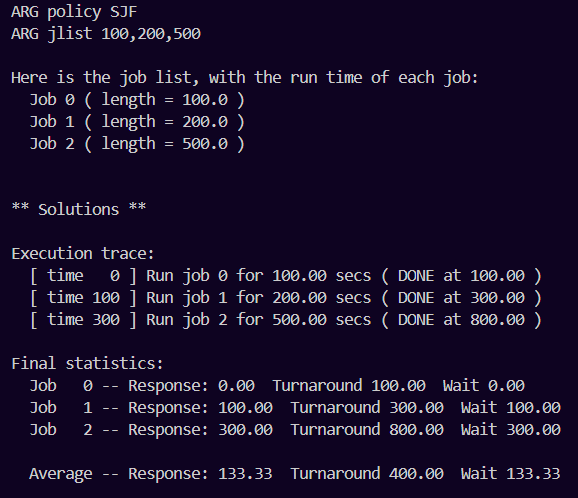
First of all, increasing the time of a job that already has the longest time to completion doesn’t change response time.

python ./scheduler.py -p SJF -j 3 -l 100,200,300 -c

python ./scheduler.py -p SJF -j 3 -l 100,200,400 -c



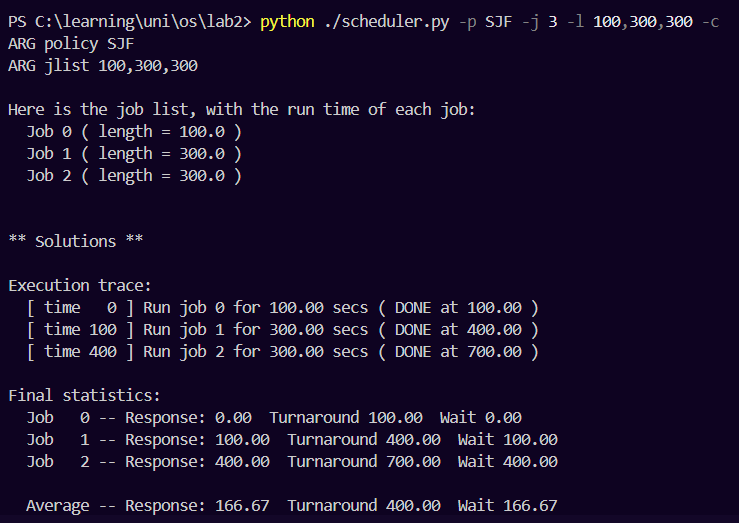
python ./scheduler.py -p SJF -j 3 -l 100,200,500 -c



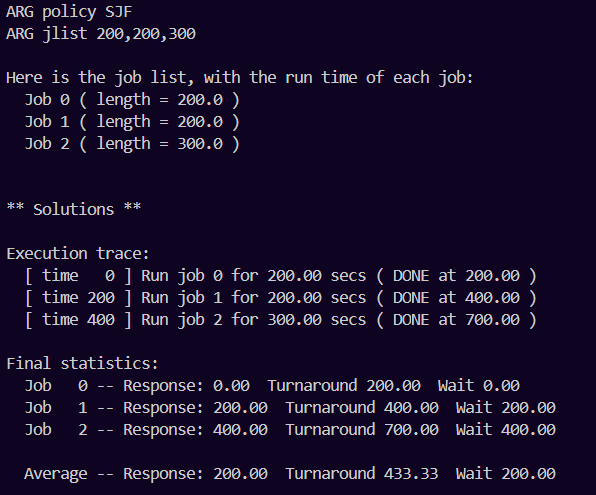
As we can see, turnaround time increases, while response time stays the same.

On the other hand, if we increase the time in other jobs (other than the longest one), response time increases. However, it doesn’t increase in the same magnitude depending on which job’s time we are increasing.

Let us first increase the time for the 2nd job (reminder: default response time was 133.33).

python ./scheduler.py -p SJF -j 3 -l 100,300,300 -c

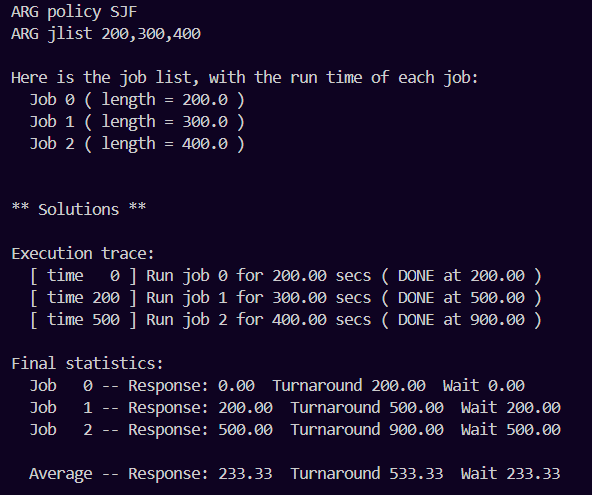
Indeed, response time is now 166.67. Now we can revert the time for the 2nd job to default (200) and try to increase the time for the 1st one.



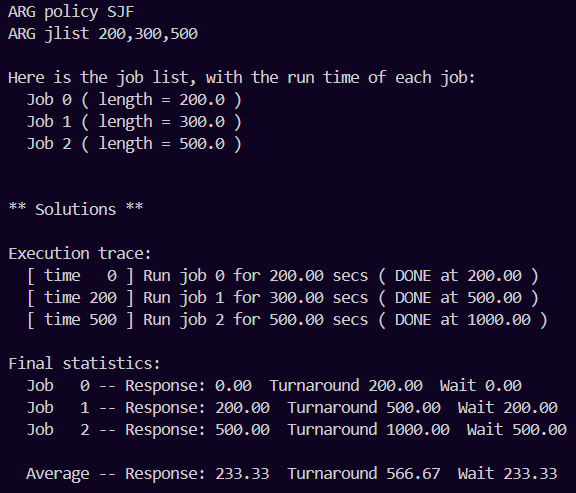
We can see, also, that response time increased. However, now it is 200.

Let’s do the same iteration again just to be sure that it is a tendency on any scale and not a coincidence (we now do this in a practical way, later we can explain the tendency). We can assume times 200,300,400 as default ones.

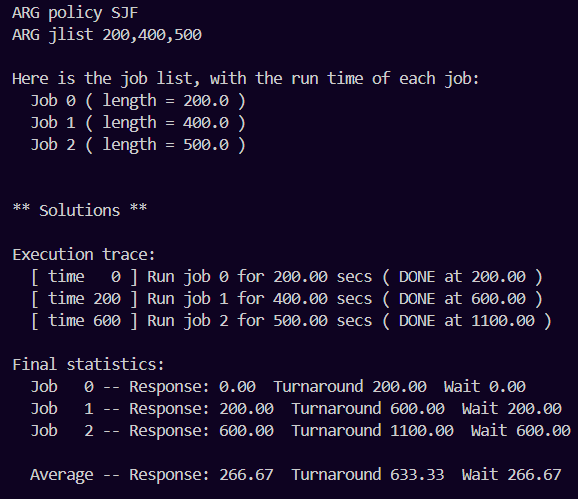
python ./scheduler.py -p SJF -j 3 -l 200,300,400 -c

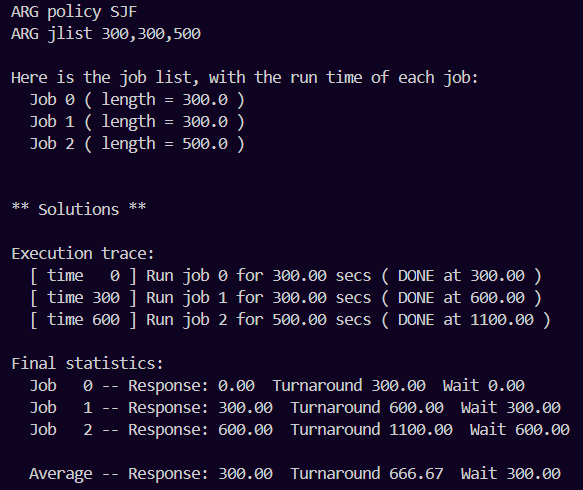


python ./scheduler.py -p SJF -j 3 -l 200,300,500 -c



python ./scheduler.py -p SJF -j 3 -l 200,400,500 -c



python ./scheduler.py -p SJF -j 3 -l 300,300,500 -c

We can see that the tendency holds.

We have shown this tendency in practice. What is an explanation?

The explanation can be found in the execution trace. We can see in both iterations that when we increase time for the 2nd job, 1st job is obviously finished as in the default case. But since we increased time for the 2nd job, it takes longer to execute, hence giving us slightly longer response time for the last job. The same logic applies to the case when we increase time for the 1st job; however, it “stacks” on one another: longer time for the 1st job gives us longer response time for the 2nd job and then for the 3rd one. It works like a snowball.  
As for the 3rd job, because it is the last one it actually doesn’t “hold back” any other jobs after it, so we can increase its time for any magnitude without increasing response time.

From what I could observe (having made some other observations which I did not put here), the magnitude of increase depends not on default values, but on values by which you increase the time of a job, which job was it and a total number of jobs. Formula should be something like this.

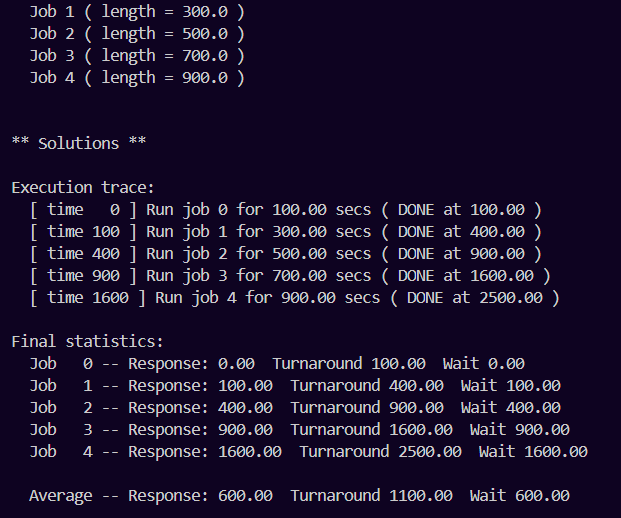
avg\_resp\_inc = x \* (m/n) for 0 <= m <= n, n>0, m∈N, n∈N. Here, x is the time by which we increase a job's time; m is a “distance” of our job from the last job; and n is the total number of jobs.   
Let’s apply it.

Defaults: 100,200,300. Let’s increase the first job’s time by 100. With our formula we should get:   
avg\_resp\_inc = 100 \* (2/3) = 66,66667. This is true, as the default value was 133.3 and became 200.

Let’s repeat for the 2nd job.

avg\_resp\_inc = 100 \* (1/3) = 33.3333. This is again true, see results above.

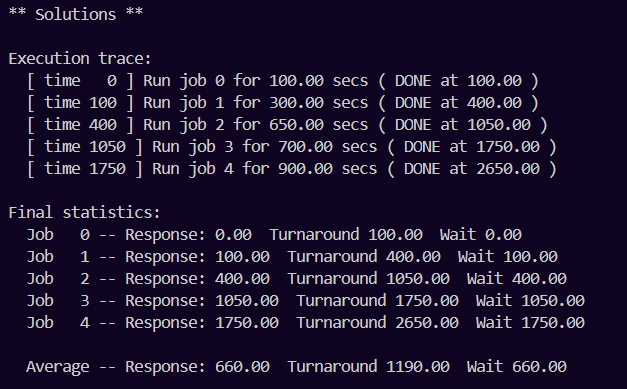
And now let’s try for a situation where we have, say, 5 jobs. Defaults will be: 100, 300, 500, 700, 900.

python ./scheduler.py -p SJF -j 3 -l 100,300,500,700,900 -c

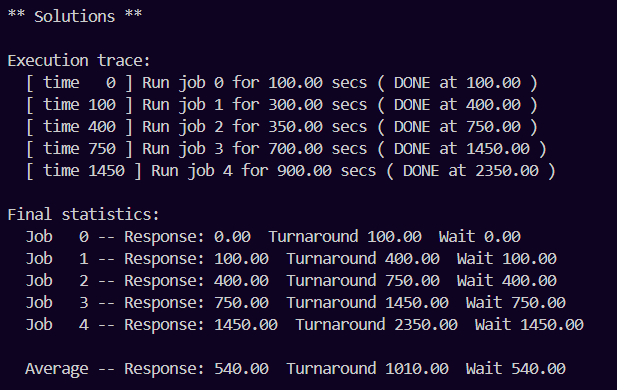
Let’s choose the 3rd job, which now has time 500, and increase it by 150.

avg\_resp\_inc = 150 \* (2/5) = 60. Let’s check.

python ./scheduler.py -p SJF -j 3 -l 100,300,650,700,900 -c



As we can see, the increase is indeed 60.

Does this formula work in reverse? If we decrease our time by 150, instead of increasing it, all that changes is a sign before x, so the result should be -60.

As we can see, it is so.

To calculate the change in average response time if we change multiple job’s completion time we can simply calculate it for every job and then add them together. I haven’t found a general formula for this, at least not yet.

**7. 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?**

Response time increases, as each process will be first started later for the same magnitude of time.

The worst case scenario here will be in the case where a) quantum time is bigger or equals the longest time for completion and b) jobs arrive in decreasing time order. Then, equation looks like:

T\_worst = ( i=1∑N-1 (T\_iresponse + t\_i) / N). Here T\_iresponse is a response time of i-sh job and t\_i is i-sh job’s time for completion. Of course, both positive.

The two conditions that we introduced are needed so that RR works like FIFO in the worst case. FIFO works fine if jobs arrive in increasing time order, but works the worst if they arrive in decreasing time order. With these conditions it is precisely our case: because quantum time is bigger or equals the longest time for completion – there is no preemption, like in FIFO; and because they arrive in decreasing time order, each job has the longest response time possible.