Exercise 2

```
Consider a CPU that has to run a bunch of jobs. You are asked to develop a scheduler for
the CPU.
Your program should run in a loop, each iteration of which corresponds to a time slice for
the CPU.
Each job is assigned a priority, which is an integer between −20 (highest priority) and 19
(lowest priority), inclusive. From among all jobs waiting to be processed in a time slice,
the CPU must work on a job with highest priority. In this simulation, each job will also
come with a length value, which is an integer between 1 and 100, inclusive, indicating the
number of time slices that are needed to process this job. For simplicity, you may assume
jobs cannot be interrupted—once it is scheduled on the CPU, a job runs for a number of time
slices equal to its length. Moreover, each time a job has spent x time slices in the
waiting line its priority is increased by 1. Notice that after the priority of a job is
increased its waiting time restarts from 0.
Your simulator must output the name of the job running on the CPU in each time slice and
must process a sequence of commands, one per time slice, each of which is of the form “add
job name with length n and priority p” or “no new job this slice”.
```

The scheduler simulator uses an adaptable priority queue. The queue is implemented by a min-heap. Each job is represented by an object of the Job class. Its attributes are:

- priority, ranging from -20 to 19, where -20 is the maximum priority.
- length, ranging from 1 to 100.
- name of the job.
- arrival_slice, representing the time slice in which the job has arrived.
- waiting_time, which is the number of slices a job has been waiting in the queue (gets reset after X time slices have passed).
- executed_slice, representing the slice in which the job was executed.

The APQ is a queue. The value is the Job instance, while the key is priority.

How it works:

- If the queue is empty, ask for a command
- If the command is "no new job this slice" jump to the next slice, otherwise execute the job just inserted.
- If the queue is not empty and a new job is inserted, add it into the queue.
- When a job is completed, remove the minumum key element (max priority) from the queue and execute it for time slices equal to its length.

- If a job has waited more than X time slices in the queue, increase its priority by 1 (-1) and put its waiting time to 0 again.

This solution does not solve the problem where shorter jobs fall behind the longer ones in execution (average waiting time goes up). A shorter job can wait up to several times its length to be executed. *when priorities are the same*

The aging of a job avoids starvation as a job will be executed at some point for sure, since it can reach the maximun priority in a given amount of time.

Another solution is to include into the key the lenght of the job.

In this way a shorter job is executed first, when priorities are the same. But this creates another problem which is starvation. Even if a job can climb up to maximum priority, it is uncertain wheter it can be executed in a finite amount of time. A case where an indefinite amount of shorter (even by 1 time slice) jobs with maximum priority arrives to the scheduler can cause starvation, in theory. In practice, for this to happen, it would require as said above an indefinite amount of maximum priority jobs, which in practical cases should not be common.

Operation Complexity

We have as many add in the queue as remove. The operation of update of the queued jobs, which is either increase the waiting time or the priority, needs to go over every element of the queue.

Operation	Complexity
update_queued_jobs(P,tau)	Omega(n)
P.min()	O(1)
P.add(), P.remove_min()	O(logn)

How did we decide the value of X?

To determine the value of X we opted for the **exponential moving average** which is a weighted moving average. This predicts the length of the next cpu burst (*tau*) by using weights and the current CPU burst length. Weights (*alfa*) are, by default, set to *0.5*

$$tau^* = \mathrm{a} * t_i + (1 - a) * tau_i$$

At the beginning is set to 1, since there are no job queued yet. Then it is calculated everytime a job is sent to the CPU by the scheduler.

This means that if a job waits for more than the estimated CPU burst length, then its priority is increased. The *ema* is useful because it can adjust to the length of the given jobs too. A fixed value might be to big for very short jobs an viceversa.

Usage

```
python3.6 mainP.py <alfavalue>
```