All students should read chapter 5 up to (but not including) §5.1.3, then §5.3–5.3.2 for this assignment as well as future lectures.

This LGA is structured differently than others before it. The reading from the text describes the important steps in creating a **Next Event Simuation** (Algorithm 5.1.2). This assignment asks you to apply these principles to two systems we can now readily imagine: a single server queue (SSQ) **with feedback** and a (**daily!** simple inventory system (SiS) with **delivery delays**).

Within your learning group, each student should pick a simulation and language from the following crossed sets:[1]

$$(\text{SSQ}, \text{SiS}) \times (\text{C++}, \text{Python}, \text{Java})$$

For this LGA, we aren't so concerned about "double coverage" — go for breadth of coverage instead. For this assignment **one simulation per student is probably sufficient** — there are a total of six choices, so even a group of five can achieve wide coverage without duplicating efforts.

For any language, you will need to decide on what your event data type looks like. There are at least two data elements it needs: one to hold the **time the event should activate** and another to identify the **type of event** it is (pseudo code uses *.at* and *.type* for these values). Depending on the event type there may be extra data elements required as well. For instance an `OrderDelivery` event in the SiS would need a `NumberOfCars` data element to represent how many cars are being delivered. Here are lists of event types and the additional data elements I've used in writing this assignment.

Simulation	Event Type	Extra Data Members
SSQ	Job Arrival	Service Time
SSQ	Feedback Arrival	Service Time
SSQ	Job Completion	(None)
SiS	Demand	(None)
SiS	Inventory Review	(None)
SiS	Order Delivery	Number of Cars

All of your implementations will use the same type of **event list**, a simple linked list (C++ `std::list`, Python `list` aka `[ "a", "b", 1, "d" ]`, Java: `java.util.LinkedList`). You can either:

i. insert events into the list wisely, keeping the list in activation time sorted order ($O(n)$),

ii. insert events naively and search the full list for the imminent event ($O(n)$),

iii. or insert events naively and use the linked list's sort functionality before extracting the imminent next event (probably $O(n \log n)$).

Option (iii) is shown in the pseudo code for this assignment.

On the following two pages are pseudo code for SSQ and SiS systems implemented using **next event simulation** techniques. The pseudo code is commented with respect to the steps in Algorithm 5.1.3. For simplicity, both systems will use a termination time of $\tau = 1000$ — not permiting new arrival events to be added after this simulation time.

Implement your simulations using your language's built-in random number generator, you may need to implement some of the variants we have seen so far such as $Uniform(a,b)$, $Exponential(\mu)$, $Geometric(p)$, and $Equilikely(a,b)$. The distributions for critical values in the simulations can be read from the pseudo code.

Be sure to print out ("report") the values for your metrics tracked in the simulation. **Even better**: use your group's `binner` from `lga-sim-correlations.pdf` to show density histograms for these results.

Some notes about each pseudo code example follow on the next page, the pseudo code listings are on the last two pages of this assignment write-up. I tend to write "mathy" pseudo code, and avoid = since it has different meanings in code as opposed to math. In keeping with mathy notation, $|\cdot|$ (absolute value bars) is used for the size (number of elements) in a container.[2]

[1] You may insert another language of your *group*'s choice, but there must be at least two group members for whom it is not a "new" language to them, the majority of group members must agree to its use, and it must be a language suitable for course projects.

[2] At least I didn't throw the proper "mathy" $\neg$ and $\wedge$ operators at you as well.

Students aren't required to follow the example logic verbatim. As long as you stick to the instructions on this first page, and end up with a working, correct simulation, we'll be good.

SSQ Notes

- The simulation state is maintained by a true queue data structure (for holding delayed jobs) and a flag variable that tracks if the service component of the SSQ is currently processing a job.
- Any completed job has a 25% chance of **feeding back** into the SSQ, the time it takes to return is $Exponential(1)$ and its next service time is always one-half of its last service time.
- Notice that we don't put 1000 job arrival events into the event list and then sort, we put only one (non feedback) job into the event list at a time. When it arrives and is processed we determine the arrival time of the job following it and schedule it with the event list.
- Like the pseudo code, your simulation should track and report a couple of simple SSQ metrics.

SiS Notes

- The S and s variables come from the textbook's original SiS presentation $S = 80$, $s = 20$. Don't confuse $S = 80$ with the script $\mathcal{S}$ in the text and lecture slides; the latter is a collection of variables called the **system state**.
- The simulation state is maintained by an inventory variable and an "on order but not delivered" variable.
- The timing of inventory reviews is not probabilistic, it happens every week and is hard-coded as 7 days.
- Notice that we don't put 1000 demand arrival events into the event list and then sort, we put only one demand into the event list at a time. When it arrives and is processed we determine the arrival time of the **next customer (demand)** and schedule it with the event list.
- Like the pseudo code, your simulation should track and report a couple of simple SiS metrics.

```
# SSQ with feedback NES pseudo code
// INITIALIZE
eventList ← empty List
t ← 0                # simulation clock
τ ← 1000      # no new events after τ
# SSQ specific state
jobQueue ← empty Queue
inService ← False
# for reporting
newJobCount ← 0
jobsServiced ← 0
# kick things off with a job arrival at some time in the future
# aka "priming the pump"
firstArrival.at ← Exponential(2.0)
firstArrival.type ← ArrivalEvent
firstArrival.service ← Uniform(0,2)
eventList.insert(firstArrival)
while( |eventList| > 0 ) do (
    eventList.sort( by .at)
    event ← extract event from eventList with smallest .at time
    t ← event.at // PROCESS EVENT
    if( event.type is ArrivalEvent or is FeedbackEvent ) then ( // PROCESS EVENT
        if( event.type is ArrivalEvent ) then (
            increment newJobCount
            # schedule the next job arrival
            nextArrival.at ← t + Exponential(2.0)
            if( nextArrival.at < τ ) then ( // SCHEDULE NEW EVENT
                nextArrival.type ← ArrivalEvent
                nextArrival.service ← Uniform(0,2)
                eventList.insert(nextArrival)
            )
        )
        # place into job queue
        jobQueue.enqueue(event)
    ) else if( event.type is JobCompleteEvent ) then ( // PROCESS EVENT
        # any job is a candidate for feedback with probability 0.25
        if( Uniform(0,1) < 1/4 ) then ( // SCHEDULE NEW EVENT
            feedbackArrival.at ← t + Exponential(1.0)
            feedbackArrival.type ← FeedbackEvent
            feedbackArrival.service ← event.service / 2
            eventList.insert(feedbackArrival)
        }
        inService ← False
    )
    if( not inService and |jobQueue| > 0 ) then (
        event ← jobQueue.dequeue()
        event.at ← t + event.service // SCHEDULE NEW EVENT
        event.type ← JobCompleteEvent
        eventList.insert(event)
        inService ← True
        increment jobsServiced
    )
)
// TERMINATE
report t, newJobCount, jobsServiced
```

```
# daily SiS with delivery delay NES pseudo code
// INITIALIZE
eventList ← empty List
t ← 0                          # simulation clock
τ ← 1000                  # no new events after τ
# SiS specific state
inventory ← S
onOrder ← 0
# for reporting
backorderCount ← 0 # number of cars purchased when inventory <= 0
orderCount ← 0
# schedule an inventory review at 7 days
firstReview.at ← 7
firstReview.type ← ReviewEvent
eventList.insert(firstReview)
# schedule a first demand arrival
firstDemand.at ← Exponential(2/3)
firstDemand.type ← DemandEvent
eventList.insert(firstDemand)
while( |eventList| > 0 ) do (
    eventList.sort( by .at)
    event ← extract event from eventList with smallest .at time
    t ← event.at // PROCESS EVENT
    if( event.type is ReviewEvent ) then ( // PROCESS EVENT
        # schedule the next review
        nextReview.at ← t + 7
        if( nextReview.at < τ ) then ( // SCHEDULE NEW EVENT
            nextReview.type ← ReviewEvent
            eventList.insert(nextReview)
        )
        # re order?
        if( inventory + onOrder ≤ s ) then (
            deliveryEvent.at ← t + Exponential(7) // SCHEDULE NEW EVENT
            deliveryEvent.type ← DeliveryEvent
            deliveryEvent.count ← S − (inventory + onOrder)
            increment onOrder by deliveryEvent.count
            eventList.insert(deliveryEvent)
            increment orderCount
        )
    ) else if( event.type is DeliveryEvent ) then ( // PROCESS EVENT
        decrement onOrder by event.count
        increment inventory by event.count
    ) else if( event.type is DemendEvent ) then ( // PROCESS EVENT
        # schedule the next demand
        nextDemand.at ← t + Exponential(2/3)
        if( nextDemand.at < τ ) then ( // SCHEDULE NEW EVENT
            nextDemand.type ← DemandEvent
            eventList.insert(nextDemand)
        )
        purchased ← Geometric(3/4)
        if( purchased > 0 ) then (
            if( inventory ≥ purchased ) then (
                decrement inventory by purchased
            ) else (
                if( inventory > 0 ) then (
                    decrement purchased by inventory
                    inventory ← 0
                )
                decrement inventory by purchased
                increment backorderCount by purchased
            )
        )
    )
)
# flow balanced inventory
if( inventory < S ) then (
    if( inventory < 0 ) increment backorderCount by |inventory|
    increment orderCount
    t ← t + Exponential(7)
)
report t, orderCount, backorderCount // TERMINATE
```