Overall I found this project very interesting! From the graphs I generated I noticed that overall, the schedulers performance was ranked as follows (best to worst) SRTF, HRRN, RR, FCFS. This makes sense when considering what we learned in class. In class you mentioned

Report on Results

- Scheduler Performance (best -> worst)
    - Average Turnaround time: SRTF -> HRRN -> FCFS, RR
    - Throughput: SRTF -> HRRN -> FCFS, RR
    - Average Utilization: All the same
    - Average Processes in Ready Queue: SRTF -> HRRN -> FCFS, RR
- Both RR(0.01) and RR(0.2) behaved similar to FCFS. In class you mentioned longer quantum length approaches FCFS but here it seems that any quantum length mirrors FCFS, just that a shorter time involved more context switching. Perhaps my results are incorrect?
- All curves (regardless of scheduling algorithm) are equal when lambda<16.666, and it matches what the queueing analysis says it should be. I thought this only worked for FCFS but it makes sense that it works always because the processes are being processed quicker than arrivals so there's no backing up where scheduling decisions would actually have an impact.
- SRTF has a microscopic average turnaround time compared to the other schedulers because it prioritizes short tasks
- After 16.666 each algorithm follows its own curve in each graph (except cpu utilization). These are useful because we can no longer use queueing analysis after this point, so the graphs show us how the schedulers behave under high load
- FCFS and RR throughput never goes above 16.666 (1/service time)

My Code

- Began writing FCFS from your pseudocode in class, finished pretty quick. Allowed me to get a good structure set up for the rest of the algorithms, including figuring out how to calculate all the statistics.
- Copied the general structure for handling events and began working on SRTF
- Upon moving on to HRRN, noticed that it was very similar to FCFS. Combined HRRN into FCFS to simulateNonPreemptivePriorityBased that took a custom comparator and a boolean for whether or not to use dynamic priority (like HRRN does)
- Then, noticed that my functions for non-preemptive and preemptive were very similar, so combined SRTF into non-preemptive to get simulatePriorityBased that takes a custom comparator, boolean for dynamic priority, and boolean for preemptive or not.
- Then added RR easily by adding time slice events
- Function that runs all schedulers and saves results to graph when -1 passed for scheduler
- main() parses command line arguments like desired for the project
- Sorting in the ReadyQueue class has bad time complexity, I know. Its O(n) because re-adding all elements of the set, for higher arrival rates this makes it take much longer.

Regardless of how you do it, dynamic priorities would require some sorting. Could have made it better by coding my own data structure completely from scratch, but using set underneath was convenient because of automatic insertion in sorted order.