The goal of this project was to help out understanding of the First Come First Serve (FCFS), Round Robin (RR), Shortest Process Next (SPN), Shortest Remaining Time (SRT), Highest Response Rate Next (HRRN), and the Feedback (FB) scheduling algorithms. These scheduling algorithms are important to understand so that we may comprehend the multiple different ways processes or jobs can be scheduled by the Operating System. Before this project, I had a very conceptual understanding of the algorithms but lacked an understanding of how to implement them into my code. Needless to say, I figured out how to implement them after days of trying and thinking about better approaches to the implementation. Eventually, I came to the implementation submitted with this summary. The way I approached this project, was to first develop everything other than the scheduling algorithms, and any helpers potentially needed. The structure of my program uses three classes, *Main*, *Job*, and *Scheduler*. I created a *Main* class that would act as the driver of the program. Here the file would be taken, in my case from the command line and parsed into an instance of another class, *Job* and is placed into an ArrayList to make later search-traversals less costly than something like a queue. Then, the *Main* class prompts the user with the algorithm selection prompt where then program waits for a valid input and executes the algorithm(s) based on the selection. The *Job* class is used to simulate an actual process by having attributes such as a name, start time, duration, wait time which any process would have. Each *Job* also has an array of strings called *outputArr*. *OutputArr* is one of the mainstays of my implementation, and allows me to manipulate each *Job’s* content in such as way that the output would be formatted in a way similar to the examples. The array is instantiated during each scheduling functions runtime to the size of all of the *Job’s* cumulative *duration*, which is calculated during the *Main* classes file parse. The permeability of the array allows it to scale for any number of *Jobs*, with any durations making it a functional way to format output, and maintain data. After a scheduling algorithm is run on a file of data and the outputArr is filled, the *Job.toString()* function traverses through the array and builds a String to be returned and printed. The third and final class, the *Scheduler*, acts as the logical unit for the program. Here each scheduling algorithm is implemented into an individual function that is to be called by the user input function in *Main*. The *Scheduler* contains five major components, a HashMap called *mapOut*, an ArrayList named *copy*, an integer named *time*, the scheduling algorithms and a bunch of helper functions. The HashMap is used to arrange the processes in the proper alphabetic order so to format the output correctly, and store completed processes following their full scheduling. It also acts as a fast lookup for previous processes since I was able to search for a *Job* based off of the *Job’s* *name*. This Map implementation could be tweaked easily and used for any *Job* attribute that you may want to use as a primary key. During the development of the algorithms I didn’t think about having to reuse the ArrayList of Jobs during a single execution which is needed if the ‘ALL’ choice is selected. So, the way I solved my problem was to have an ArrayList *copy* that would store a deep copy of the *Job* ArrayList before each scheduling. The *schedule\_all ()* function saves a copy the ArrayList, schedules the *Jobs*, then resets the ArrayList back to the saved copy in between each algorithm. In order to ensure the copy was in fact a deep copy helper functions, *Scheduler.properClone()*, and *Job.createClone()*, were created to facilitate the proper storage. Helper functions such as *getMin (), getMin2 (), getMaxRR (), and massIncWaitTime ()* were created to assist specific algorithms in their scheduling computation. The Shortest Process Next implements, *schedule\_spn ()*, uses the *getMin ()* function in order to get the shortest duration process left in the array. The Shortest Remaining Time implementation, *schedule\_srt ()*, uses the *getMin2 ()* function to fetch the Job with the shortest remaining time. The Highest Response Ratio Next implementation, *schedule\_hrrn ()*, uses both the *getMaxRR (),* and the *massIncWaitTime ()* functions that fetches the Job with the highest response ratio in the ArrayList, and increments the wait time of all other processes not currently running. Other helper functions which have more of a wider use are the *checkStart (), mapAdd (), printMap (),* and *instArr ()* functions. The *checkStart ()* function is used by the preemptive algorithms, and checks for any process that should be ready to enter the ready queue based on comparing the Job’s start time and the systems time attribute. The *mapAdd (), printMap (),* and *instArr ()* functions are used by every scheduling algorithm because they assist in the output and instantiation of the Job’s and their outputArr. Overall, this project was challenging at the start but grew to be more reasonable over time once my own implementation method was being set. The open-endedness of the project was something that I thoroughly enjoyed. Not being given required data structures or implementations to use really allowed me to think through the problem in a way unique to myself. The only thing that I’d wish to do differently is doing a JavaFX graphics output rather than a text based one, but unfortunately my time with other classes made it difficult to grow my time commitment to the project.