Steven Lee
Project 4 analysis

In this project, I used four versions of quick sort and one version of heap sort to successfully sort arrays. To do this, I read 20 input files of varying sizes and sorting. As an enhancement, I also included an additional type of input file, which was an entirely duplicated input file, or a list consisting of just '1' the appropriate number of times. This project contains the following classes, Main, Metrics, Node, Queue, SortTimer, and Sorter. The main class handles file input, as well as statistical measurements on the time taken for each algorithm. As an enhancement, I also provide formatted, basic summary statistics as well. Finally, I also instantiate all of my other classes in Main, so that I can call them as needed. The Metrics class contains all metrics on performance for each algorithm execution, as well as calculations for summary statistics. In this class, I use array implementations since I know exactly how many input files there are, and how many algorithms there are in the program. Additionally, the Metrics class is instantiated after the file input occurs, and the length of the array is based on the count of files, so I am sure that I am not wasting space. The Node class contains basic node operations, necessary for linked implementations of data structures. The Queue class is a linked implementation of a queue data structure. This was necessary because the input files all have different lengths, so an array implementation would have wasted space. In contrast, a linked implementation can vary in size, so each input file only makes a queue that is just large enough to contain it. The SortTimer class contains all of the timing operations as each algorithm executes on each file. Additionally, it writes the sorted files to a text file once the timing operation has been stored. The Sorter class contains each sorting algorithm, along with their different implementations and helper functions.

Regarding data structures used to solve this problem, I used queues with a linked implementation to read in the files, but I converted these queues to arrays once the sorting actually took place. This is because, for each of my algorithms, I selected the versions that used array implementations. However, no space was wasted because I kept track of the number of elements in the Queue class, so the array was correctly proportioned to the number of elements.

In terms of time efficiency, this program is efficient, given the constraints. Quicksort runs in O(nlog(n)) time in the average and best cases, but we were given conditions in which quicksort would run in $O(n^2)$ because we selected the first item of the partition as the pivot, and used arrays which were sorted in ascending and descending order. By doing this, we maximize our empty pivots, which increases the number of operations. Heapsort, in contrast, has O(nlog(n)) efficiency regardless of the given input. We can see this reflected in the output statistics, where it has fairly regular performance, which is around 170,000 nanoseconds for files of size 1,000 for example. In terms of space complexity, there is a possible tradeoff that I made regarding the initial reading of the files. I knew the files had a maximum size of 5000, which meant that I could have declared arrays of size 5000, but that would have wasted space. My other option would have been to declare individual array variables for each input file, which would have wasted no space, but it would have greatly reduced the legibility of the program, since I would be declaring 20 variables, rather than one queue array.

In this project, I learned about the real consequences of making bad decisions with regards to my data structure choices. Particularly, I could see that quicksort was easily improved by simply changing the partition choice from the first element to the median of three elements. This may not guarantee that I get optimal performance, but sampling values to ensure that I avoid worst case performance is important. Additionally, I learned about how implementing Heapsort actually works, from a code perspective. Knowing the theory is important, but implementation is what makes the algorithm actually matter for real problems. Overall, I think that I prefer Heapsort over Quicksort now, because I know that it is a more robust choice, regardless of the real distribution of data. This is interesting because they are both considered to be 'efficient' sorts.

Next time, I think that I would fully read the requirements of the lab, prior to coding. I did not realize that the outputs of each sorting algorithm needed to be printed, which meant that I thought I was prematurely done. Additionally, I think that I would try to find a way to declare a ragged array in Java. I know that is possible in C, and it would have been the most efficient choice for reading in the variably sized input files, but I did not know how to do it in Java, or if it was even possible. Finally, I would have tried the iterative versions of all of the algorithms to see how much faster they worked.

In this lab, I chose the recursive implementation for all of my sorts. I did this because I think that we learned how to do the sorts in a recursive manner, and they made more intuitive sense to me as a recursive algorithm than an iterative algorithm. Since I had a fairly good idea of how to code the recursive version, and could fix any mistakes that happened more easily, I knew that the actual goals of the project could be achieved more quickly, as long as I was using either iteration or recursion in all cases.

As previously mentioned, as enhancements, I provide summary statistics on average and total runtimes for each algorithm, as well as providing additional input for unsortable data, which is a duplicate value.