Krisha Sharma

CSE 13s

Winter 2022 Long

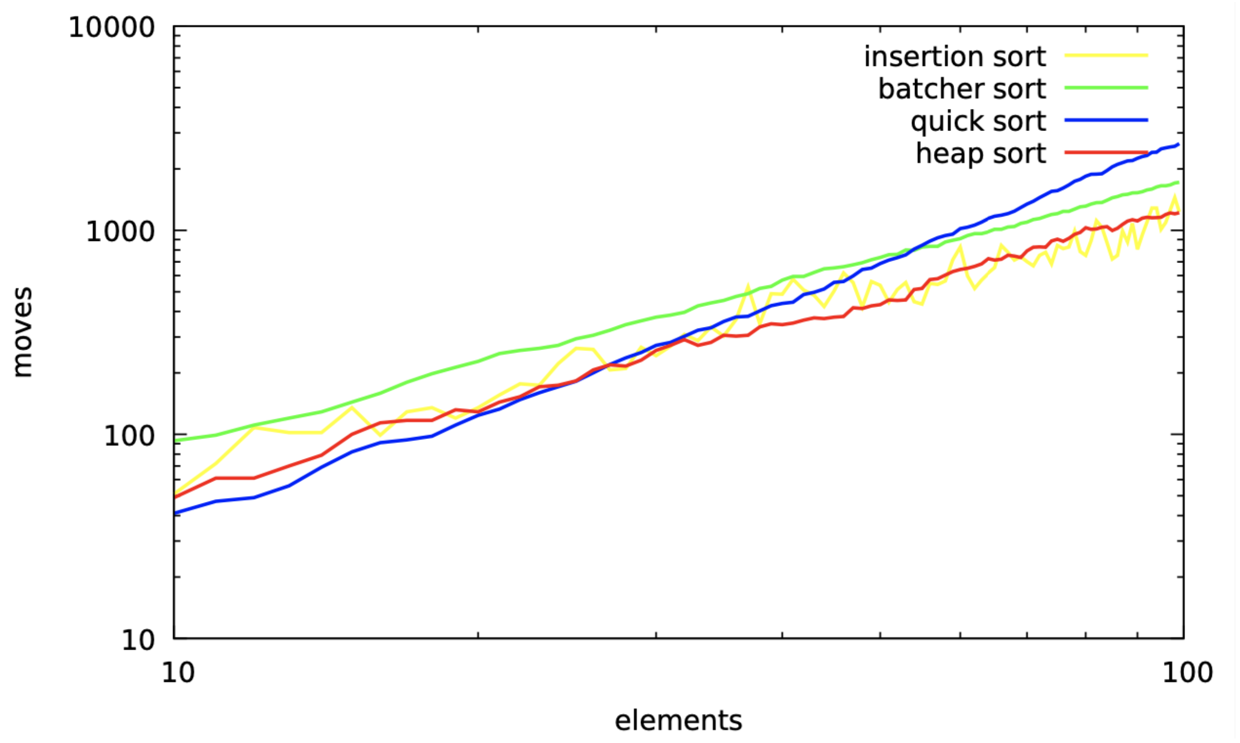
Assignment 3: Sorting WRITEUP

Introduction:

Assignment 3 is focused on sorting, specifically, it is focused around 4 different sorting algorithms. Batchers even-odd merge sort, Heapsort, Quick Sort, and Insertion Sort. Our task for this assignment is to implement each of these sorting algorithms based on the python pseudocode provided in the assignment document PDF. After implementing Insertion Sort, Heapsort, Batchers Sort, and the recursive Quicksort, the assignment asks us to create a test harness that creates an array of pseudo-random elements and tests each of the sorts. The test harness we will create also needs to support the command-line options specified in the assignment document. This assignment also requires us to use a set to track which command-line options are specified when the program is run. The final part of this assignment is to gather the statistics about each sort and its performance, such as the size of the array, and the number of moves and comparisons required. This assignment has the overall purpose of getting us fully familiarized with each sorting algorithm and for us to get an overall understanding of computational complexity.

General Graph Analysis:

Each of these 4 sorting algorithms/networks (Heapsort, Batchers Sort, Insertion Sort, and Quicksort) all work in different ways to sort their given inputs. Because they all differ in the ways that they choose to sort, the data behind each sorting algorithm differs as well. The amount of “moves” and “compares” that each algorithm completed was recorded and collected in order to analyze. Each of the algorithms end up sorting the array's; however, the time complexity between them differs along with the amount of times an element in the array is transfered or compared. Each of these algorithms introduced the idea of computational complexity and the analysis of the graphs only furthered my understanding.



As you can see in this graph above, all the sorting algorithms are shown alongside with the amount of moves they performed and the amount of elements in the array. Across all the sorting algorithms, Batcher, Quick, Heap, and Insertion, it is clear to see that the amount of elements in the array directly affects how many moves each sorting algorithm is making.

When the number of elements is lower and closer to 10, the number of moves stays small for all of the sorting algorithm, they roughly stay around 50-100 moves depending on the algorithm. However, when the number of elements in the array increases, so does the number of moves. When the number of elements in an array reaches closer to 100, the number of moves for all the algorithms nears 1000. This is because when there are a smaller number of elements in a given array, the number of moves, transfers, or swaps needed to sort the array will be less compared to an array with a larger size, therefore it makes sense that the number of moves increases as the number of elements in an array increases.

We can also reach the conclusion that based on the data collected and the graph above, Quicksort is the fastest sorting algorithm, followed by Batcher Sort and Insertion Sort, with Heapsort coming in last.

Insertion Sort:

Insertion sort is a sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The first step involves the comparison of the element in question with its adjacent element.

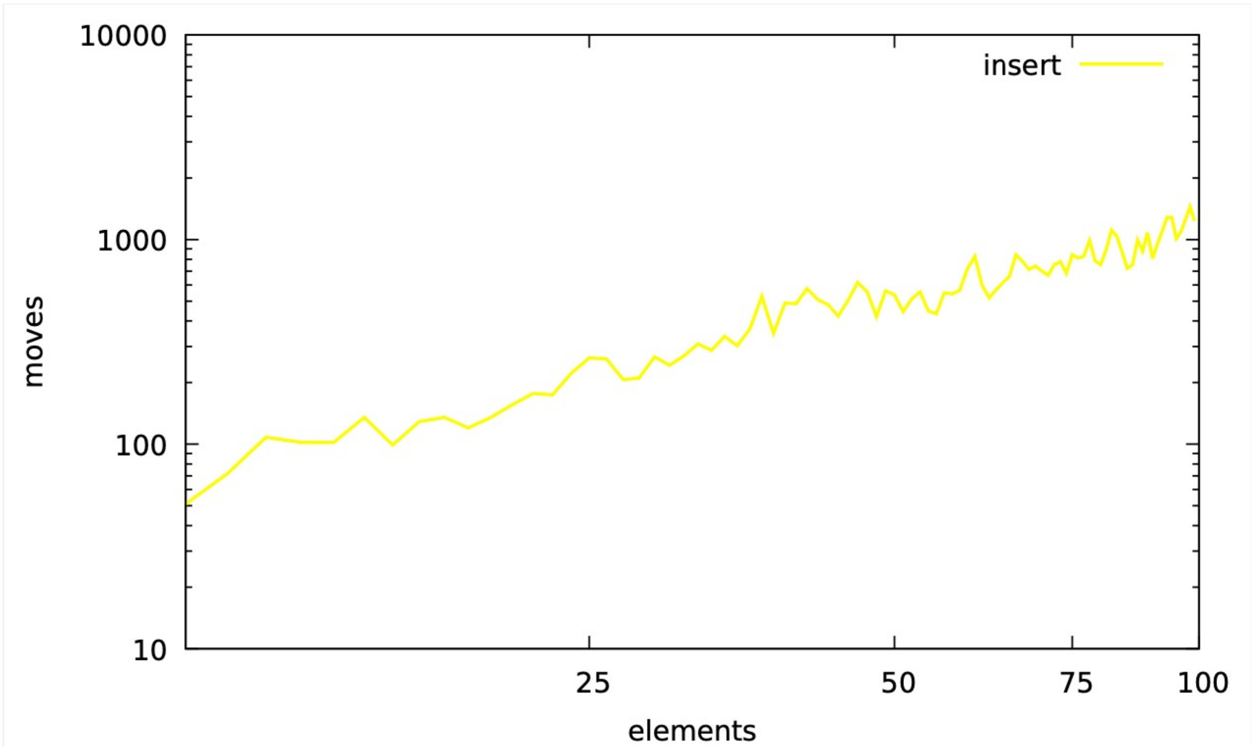
The sorting algorithm generally follows this structure of steps:

1. The first step involves the comparison of the element in question with its adjacent element.
2. If every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other element's one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the elements in the array are at their apt position.

Insertion Sort is an algorithm based on the one assumption that a single element is always sorted. I found out that the running time for an algorithm is generally the execution time of each line of the algorithm. The time complexity for Insertion Sort is as follows:

- Worst case time complexity: $o(n^2)$
- Average case time complexity: $o(n^2)$
- Best case time complexity: $o(n)$

In terms of memory usage for this algorithm, there is no extra space needed since all this algorithm does is rearrange the input array to reach the desired output. Searching for the correct position of an element and swapping are the two main operations in this algorithm. This algorithm could possibly be optimized further by possibly using a binary search to help with the complexity or using a doubly linked list rather than an array instead.



Looking at the graph of the Insertion Sort sorting algorithm individually, we can see that the same thing that we concluded above, while looking at the graph of all the algorithms, can be concluded for Insertion Sort. Here we can more clearly see that as the number of elements in the array increases, so does the amount of moves the algorithm makes to sort the said array.

Looking at this close up graph however, can show us something that we might not have seen in the earlier graph. In this graph, it is very clear that the line graphing out the algorithm's performance is jagged and going up and down, rather than being straight. While the graph still does show a linear correlation between the number of elements and amount of moves performed, it also shows us that there is a slight fluctuation in terms of moves and elements.

At times in the graph it is clear that the amount of moves jumps up at a certain number of elements but then back down when the size of the array increases by a bit. Overall the graph moves linearly upwards, increasing as the amount of elements and number of moves increases.

Batcher's Sort:

It is important to make the distinction that the sorting algorithm we implemented for Bathcer's sort uses merge exchange sort as well meaning that it is more closely Batcher's merge-exchange sort. This odd-even mergesort is a merging network, not a sorting network. However, since the design can scale to larger and smaller merge networks, a sorting network is easily constructed with it. Batcher's odd even mergesort is a generic construction that was devised for sorting networks of size $O(n(log\ n)^2$ and depth of $O((log\ n)^2)$. Batcher's Sort is based on a merge algorithm that merges two sorted halves of a sequence to a completely sorted sequence.

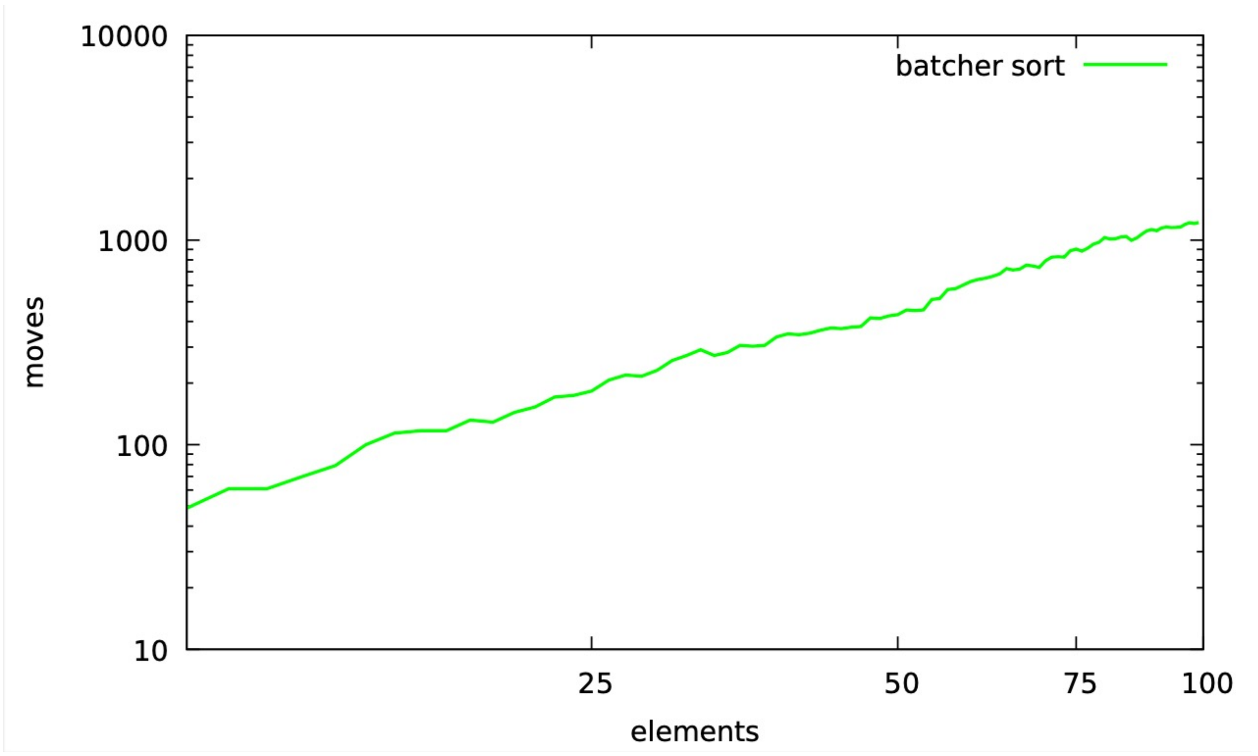
The sorting algorithm generally works like this:

Given the input of a sequence $\mathbf{a_0, ..., a_{n-1}}$ (n a power of 2), if $n > 1$ then:

1. apply Batcher's odd-even mergesort $(n/2)$ recursively to the two halves $\mathbf{a_0, ..., a_{n/2-1}}$ and $\mathbf{a_{n/2}, ..., a_{n-1}}$ of the sequence

2. Finally odd-even merge (n)

In our implementation of Batcher's sort we were instructed to combine it with merge exchange sort. Along with this, our implementation of Batcher's Sort is to run sequentially meaning that it will be sorting the input over several rounds. Below is the graph of the Batcher's Sort algorithm's performance in terms of moves and elements.



We can clearly see here that as the number of elements in the array increases, so did the amount of moves the algorithm performed, reaching the same conclusion we reached will Insertion Sort. This graph differs from the Insertion Sort graph in that the lines are more "straight" and generally less shaky or jagged. This means that there is very little variation in the algorithm and that the amount of moves is more closely and directly related to the number of elements in the array.

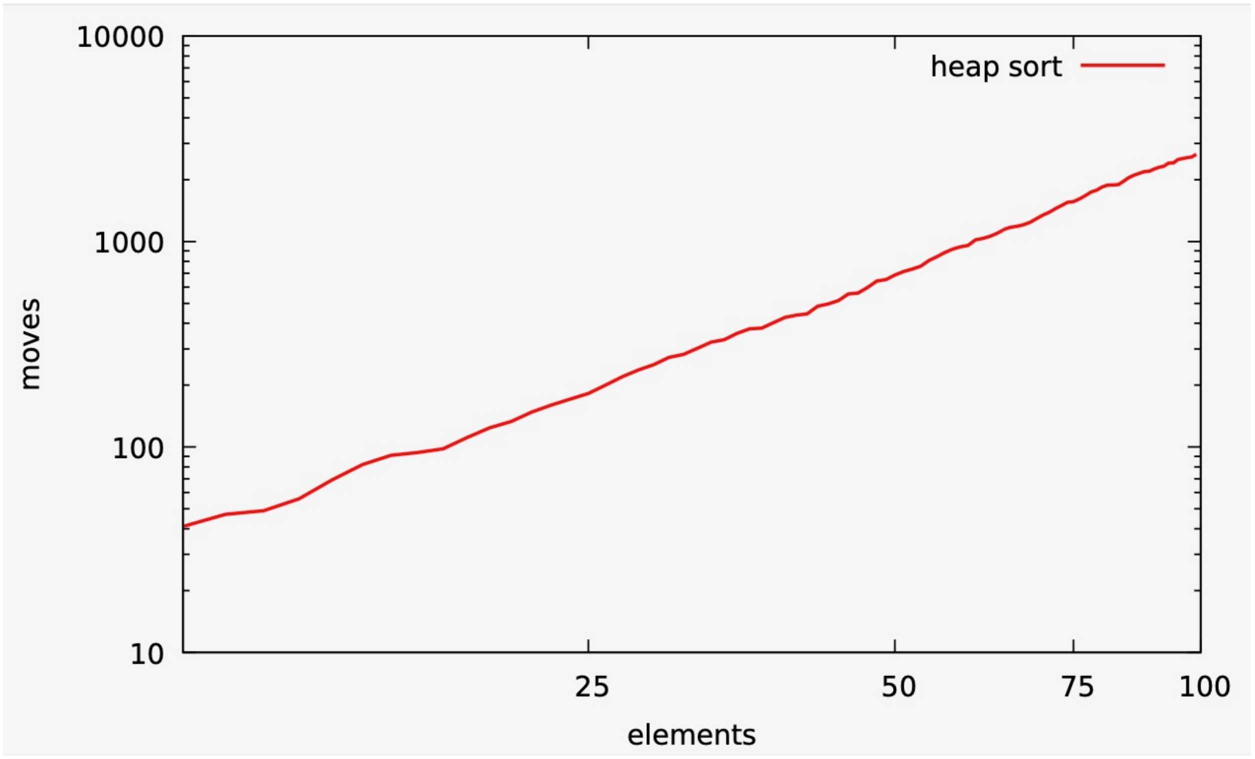
Parts of the line jump up and down; however, it is very minute and less noticeable when compared to the graph displayed for Insertion Sort. Overall this graph shows us that the amount of elements in the array directly correlates with the number of moves being performed by the algorithm. This graph specifically shows how this algorithm is more stable in terms of the fact that if needed, we could more accurately predict how many moves the algorithm would make based on the number of elements due to the fact that that graph is much more linear and straight.

Heapsort:

Heapsort is one of the sorting algorithms that we had to implement for this assignment and it works by processing the elements of an array and creating the min-heap or max-heap using the elements of the given array. Heapsort structure is based on a concept called a binary tree. Binary trees are rooted at some node, and while in a binary tree any node can have at most 2 “children”; it is also possible for a node to have one “child” or no “children”.

In Heapsort, you can have a max-heap and a min-heap. For a max-heap, any parent node must have a value that is greater than or equal to the value of their “children”. For a min-heap, any parent node must have a value that is less than or equal to the value of their “children”.

Heapsort recursively performs two main operations: first, it builds a heap using the elements of an array and second, it will repeatedly delete the root element of the heap formed in the first phase. The overall concept of Heapsort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.



This is the graph that displays the performance of the Heapsort algorithm, and in it we can once again reach the conclusion that as the number of elements in the array increases, so does the amount of moves the algorithm performed. Out of all the graphs we have seen so far, Heapsort by far shows a performance that is the move stable and linear. There is almost zero variation in the line and when there is, it is located closer toward the end of the line, when the elements in the array increase in number. When the number of elements stay small the moves the algorithm made to sort the array stays relatively linear as well.

We can also reach the conclusion that Heapsort is an efficient, however, unstable sorting algorithm with an average, best-case, and worst-case time complexity of *O(n log n)*. Heapsort is significantly slower than Quicksort and Merge Sort, so Heapsort is thus most likely less commonly encountered in practice.

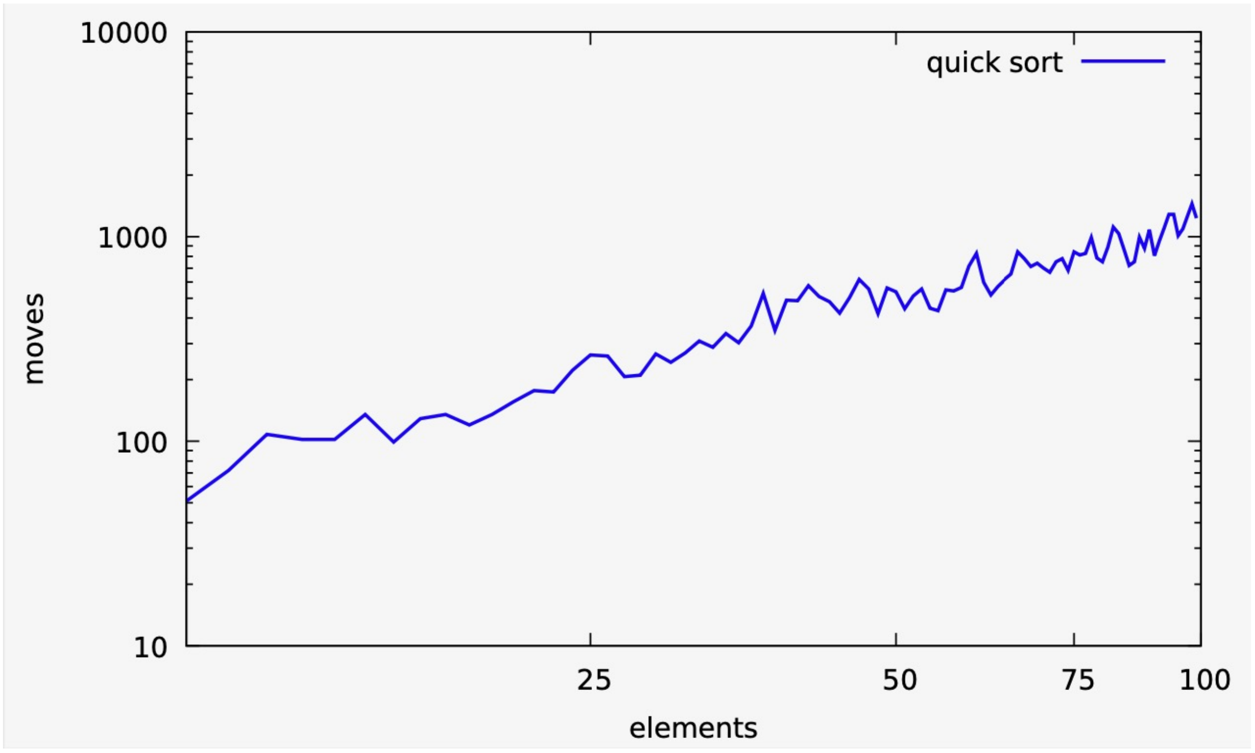
Quicksort:

Quicksort (sometimes called a partition-exchange sort) works as a divide-and-conquer algorithm. Quicksort creates two empty arrays to hold elements less than the pivot value and elements greater than the pivot value, and then recursively sort the sub-arrays. There are two basic operations in the algorithm, swapping items in place and partitioning a section of the array. To start, it partitions an array into two subsequent arrays by selecting an element from an array and designating it as a pivot, then elements that are less than the pivot are placed to the left-sub array, and elements that are greater than or equal go to the right-sub array.

The overall general steps for quick sort are as follows:

1. Find a “pivot” item in the array. This item is the basis for comparison for a single round.
2. Start a pointer (the left pointer) at the first item in the array.
3. Start a pointer (the right pointer) at the last item in the array.
4. While the value at the left pointer in the array is less than the pivot value, move the left pointer to the right (add 1). Continue until the value at the left pointer is greater than or equal to the pivot value.
5. While the value at the right pointer in the array is greater than the pivot value, move the right pointer to the left (subtract 1). Continue until the value at the right pointer is less than or equal to the pivot value.
6. If the left pointer is less than or equal to the right pointer, then swap the values at these locations in the array.
7. Move the left pointer to the right by one and the right pointer to the left by one.

8. If the left pointer and right pointer don’t meet, go to step 1.



The graph above displays the performance of the Quicksort algorithm in terms of moves and elements in an array. In this graph it is clear to see that the number of moves the algorithm makes to sort correlates to the elements in the array. Similar to all the other sorting algorithms, as one increases, so does the other. This graph is different when compared to all the other sorting algorithms because there seems to be more variability and instability in the line for Quicksort.

It is important to note that the instability increases as the number of elements in the array increases as well. Meaning that when the size of the array is bigger and more moves are required to sort it, there is more variation and instability. Quicksort generally has two main disadvantages, the first being that in the worst case (with elements sorted in descending order), its time complexity is $O(n^2)$. The second disadvantage is that Quicksort is not a stable sorting algorithm. QuickSort is an unstable algorithm because we do swapping of elements according to pivot's position; however, we do all of this without considering their original positions. I recently learned that the stability of a sorting algorithm is concerned with how the algorithm treats equal (or repeated) elements. Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't. In other words, stable sorting maintains the position of two equal elements relative to one another. Quicksort fails to do this and is thus an unstable sort.

Error Handling:

- I ran into an infinite loop error while trying to implement Batchers Sort. I realized that my infinite loop was because I had an incorrect argument statement and needed a less than 0 where I had a less than or equal to 0.
- I ran into another infinite loop error while trying to print out the command line outputs. The while loop I implemented to print the array values never stopped quitting and I realized it was because I didn't have an if statement actually breaking out of the loop.
- I ran into a segmentation fault error along with a few other syntax errors while trying to implement Heap Sort and sorting.c; however, when I fixed the syntax error, the segmentation fault error went away. I was told that by typing Valgrind into the command line, I would be able to see my issue but I didn't get the chance to try since my error was resolved.
- I ran into a memory leak error but I fixed it quickly by implementing free() in my code.

Citations:

- Throughout this assignment, I did high-level pseudocode collaboration with my sister Twisha Sharma (tvsharma). We bounced ideas off of each other and generally talked out the best ways to go about implementing each of the sorting algorithms.

- Professor Long is cited for the python pseudocode I based my sorting algorithms code off of. He is also credited for all the additional files he supplied for this assignment that helped me understand how to implement parts of my code.
- I watched Eugene’s recorded section video that he posted to Yuja to help me get started on this assignment. The section was on the 21st of January, and in it, Eugene talks about assignment 3, sorting, time complexity, sets, and dynamic memory allocation [malloc()/calloc()]
- I attended Brian’s section on 12/26 for help with a segmentation fault error I ran into. Brain told me that Valgrind would help me identify the problem and that I should start there. He also told me that it was ok for me to pass my “helper” functions through Stats *stats in order to be able to use the swap, move, and compare functions when needed. He said that since the functions keep track of the moves and comparisons and increase the count automatically I would be better off using them rather than hardcoding the parallel assignment. This bit of advice helped me begin to print out the number of moves and comparisons each of my sorting algorithms was doing.
- I attended an MSI LSS tutoring session with Kat on 1/26 and during it, she helped me check to see if I had any infinite loops in my Batcher’s Sort program since when I would go to test it on the command line, the program would never quit running or terminate.
- I attended Audrey’s tutoring session on 1/26/2022 for help with implementing sets into the assignment. I wanted clarity on how to properly implement it and during the session, Audrey showed me how sets works and told me how bool flags are no longer needed for the case statements when implementing sets.

Conclusions:

Over the course of this assignment, I learned a lot about sorting algorithms and computational complexity, and what I learned got me thinking about what the best way to apply these sorting algorithms would be. Specifically, I was thinking about the best way to implement Insertion Sort. The sorting algorithm could be used in sorting small lists or used in sorting "almost sorted" lists. It could also be effective when used to sort smaller subproblems in Quick Sort. I also learned a lot about how time complexity and stability play a role in sorting and what computational complexity really was. At first, I never thought to think about the resources required to run a simple algorithm, but this assignment had me focusing on that. I learned that overall, the computational complexity of an algorithm is the number of resources required to run it. Particular focus is given to time and memory requirements. The complexity of a problem is the complexity of the best algorithms that allow solving the problem. I also learned how Quicksort is an unstable sorting algorithm but one of the fastest.

While completing this assignment and learning about sorting algorithms and time complexity, I generated a question in regards to Quicksort and Heapsort. As we know, the quicksort performance is $o(n(log(n)))$ on average but the heapsort performance is $o(n(log(n)))$ on average too. So the question is why Quicksort is faster on average?