Performance of Sorts on Variety of Inputs

Alex Yeh

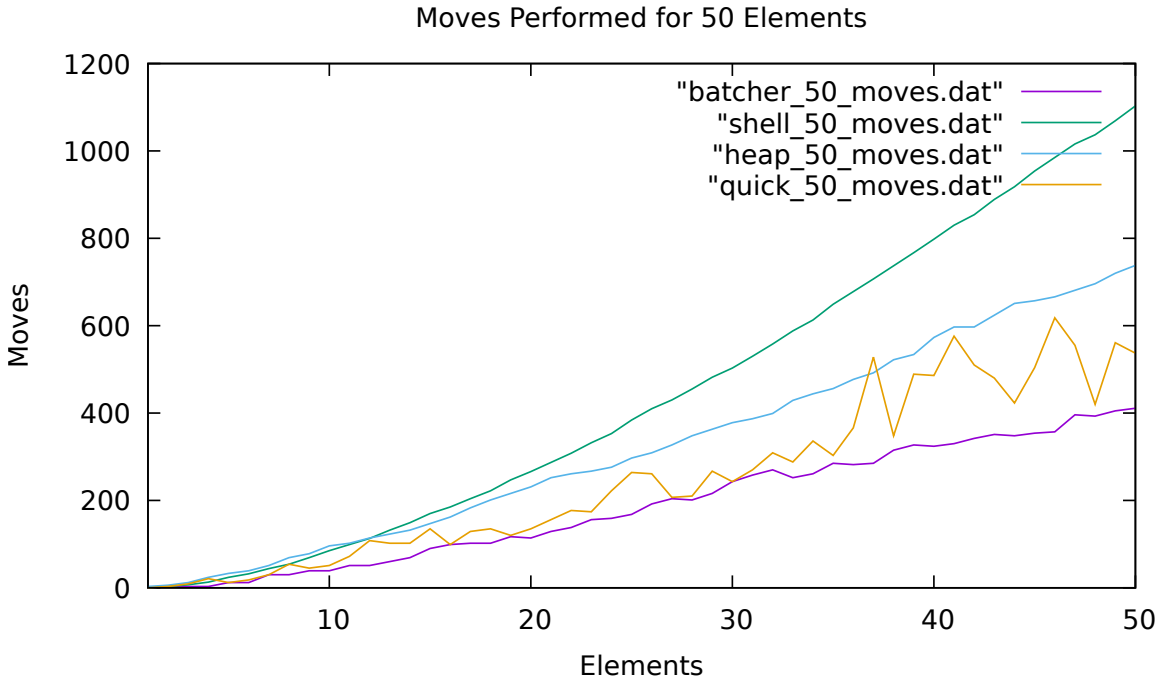
February 4, 2023

1 What I learned about the Different Sorting Algorithms

Sorts perform well if there are less elements in the array which leads to less moves and compares. Sorts perform poorly if there are more elements in the array which leads to more moves and compares. When I ran all of the sorts with 50 elements, shell sort used the most amount of moves and compares (1103 and 575 respectively). Batcher sort used the least amount of moves (411) and quick sort used the least amount of compares (252). When I ran all of the sorts with 1000 elements, shell sort used the most moves and compares (66253 and 34407 respectively). Quick sort used the least amount of moves and compares (18642 and 10531 respectively). From my findings, I conclude that batcher sort is the best sort for arrays with less elements and quick sort is the best sort for arrays with more elements.

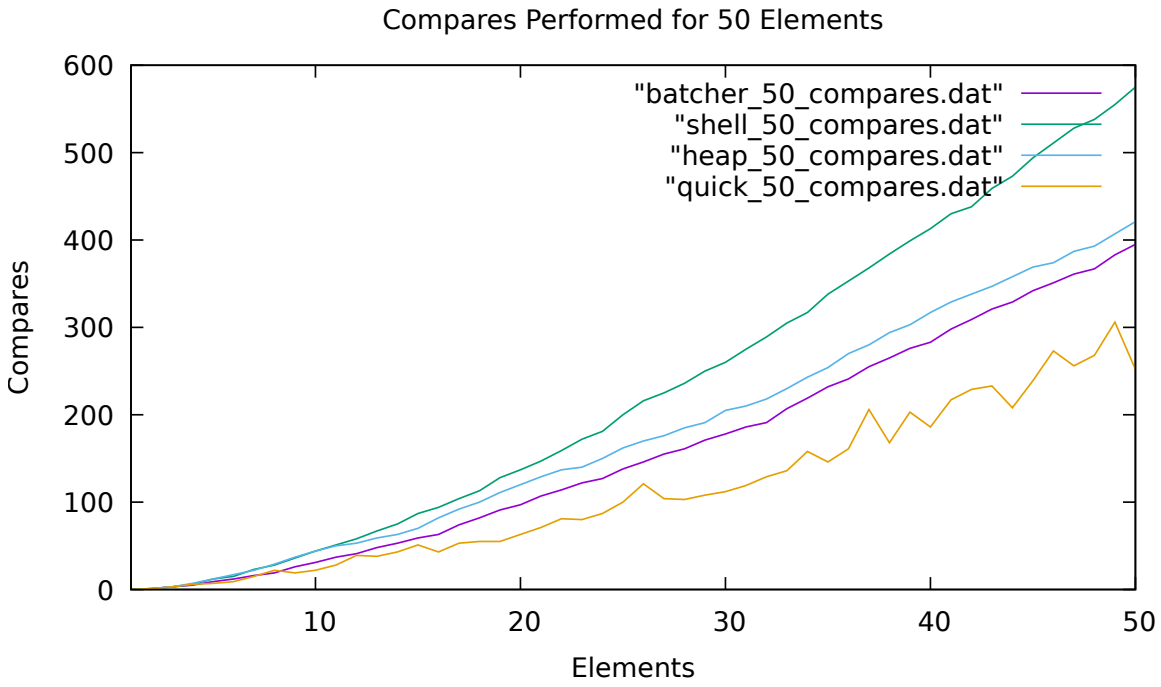
2 Graphs of the Performance of the Sorts on a Variety of Inputs

2.1 Moves Performed for 50 Elements



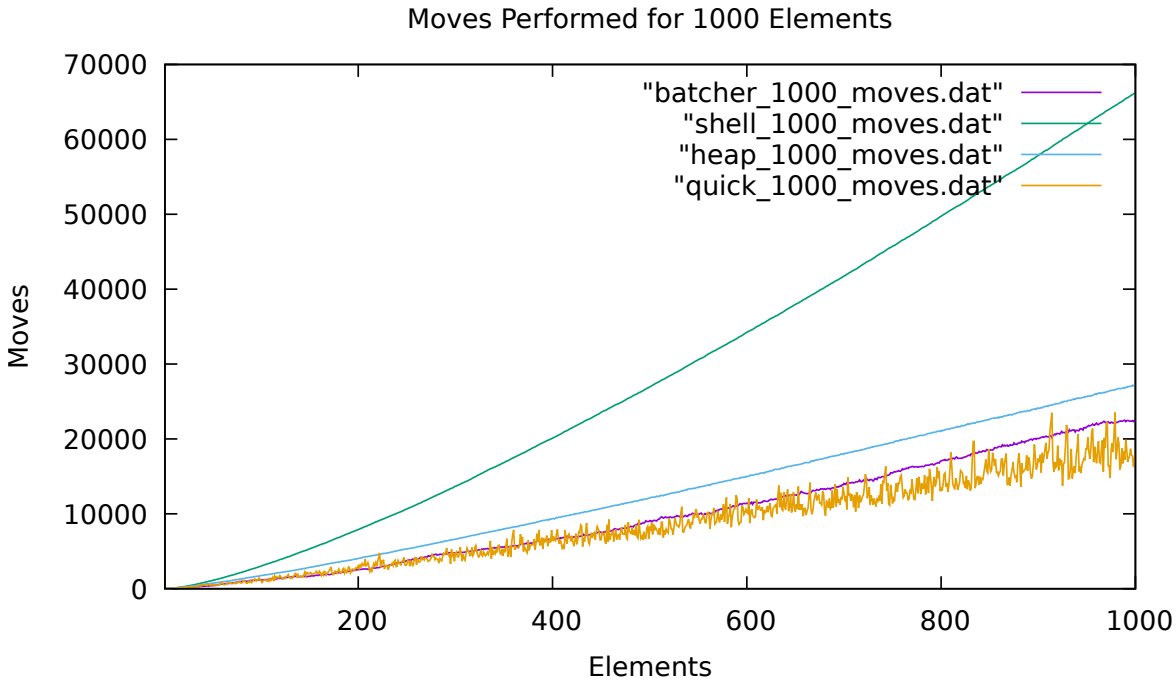
The graph above shows my four sorting algorithms, batcher sort, shell sort, heap sort, and quick sort, sorting an array up to 50 elements and the number of moves required for each number of elements. This represents arrays with a small number of elements to observe how the number of moves changes as the number of elements remains small. As we can see from the graph, shell sort uses the most moves for the majority of elements. We can also see that batcher sort uses the least amount of moves for the majority of elements. An additional observation we can make is that not all of the lines are smooth. The use of swapping in the sorts could be causing this as swapping uses three moves. From seeing how high the shell sort line goes and how low the batcher sort line goes, we can conclude that shell sort is the least efficient in terms of moves usage for sorting arrays with less elements and batcher sort is the most efficient in terms of moves usage for sorting arrays with less elements.

2.2 Compares Performed for 50 Elements



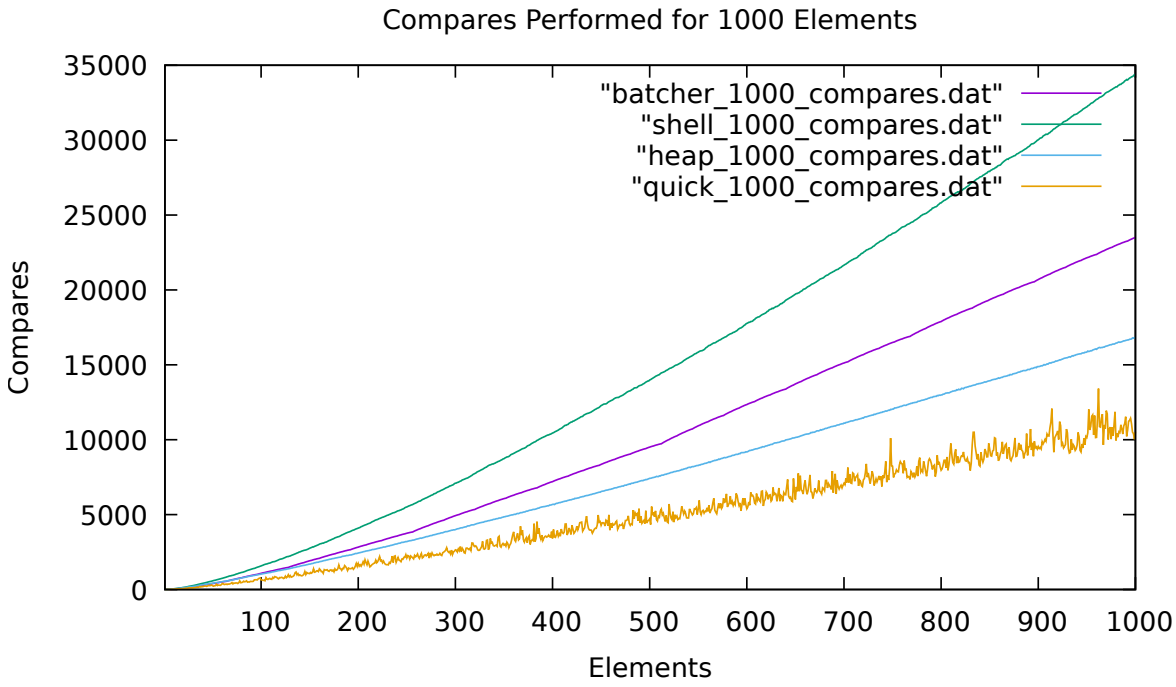
The graph above shows my four sorting algorithms, batcher sort, shell sort, heap sort, and quick sort, sorting an array up to 50 elements and the number of compares required for each number of elements. This represents arrays with a small number of elements to observe how the number of compares changes as the number of elements remains small. As we can see from the graph, shell sort uses the most compares for the majority of elements. We can also see that quick sort uses the least amount of compares for the majority of elements. An additional observation we can make is that not all of the lines are smooth. The use of swapping in the sorts could be causing this as swapping uses three moves. From seeing how high the shell sort line goes and how low the quick sort line goes, we can conclude that shell sort is the least efficient in terms of compares usage for sorting arrays with less elements and quick sort is the most efficient in terms of compares usage for sorting arrays with less elements.

2.3 Moves Performed for 1000 Elements



The graph above shows my four sorting algorithms, batcher sort, shell sort, heap sort, and quick sort, sorting an array up to 1000 elements and the number of moves required for each number of elements. This represents arrays with a large number of elements to observe how the number of moves changes as the size of the array increases. As we can see from the graph, shell sort uses the most moves for the majority of elements by a large margin compared to the other three sorts. We can also see that quick sort uses the least amount of moves for the majority of elements. An additional observation we can make is that not all of the lines are smooth. The use of swapping in the sorts could be causing this as swapping uses three moves. From seeing how high the shell sort line goes and how low the quick sort line goes, we can conclude that shell sort is the least efficient in terms of moves usage for sorting arrays with more elements and quick sort is the most efficient in terms of moves usage for sorting arrays with more elements.

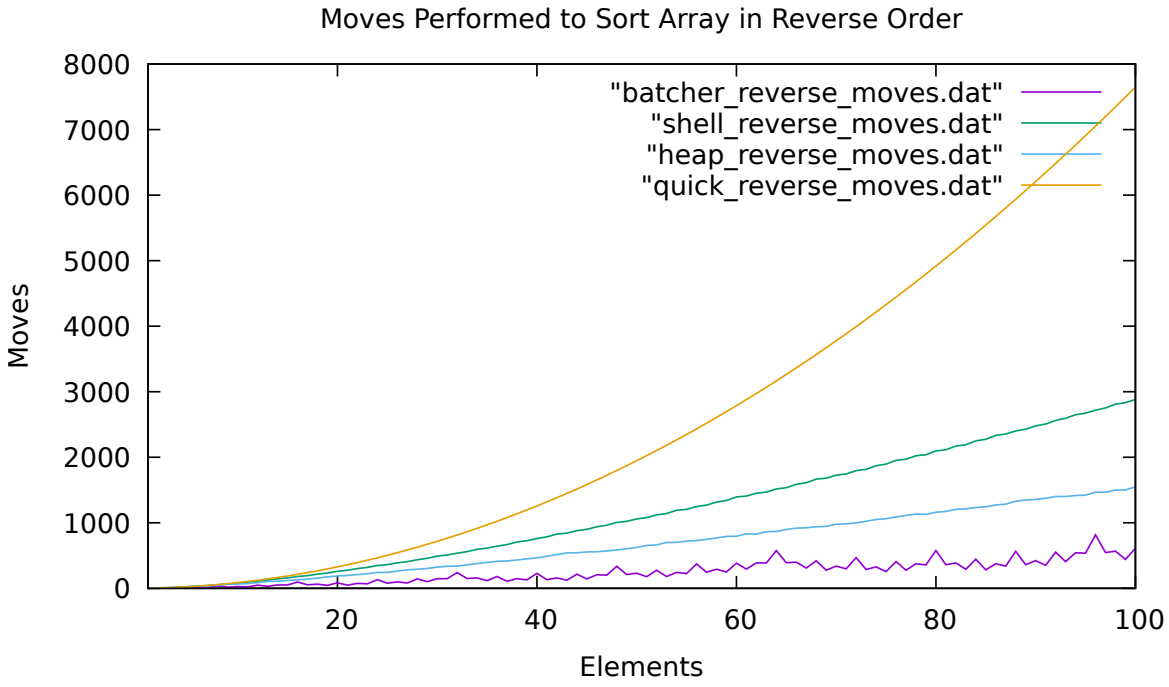
2.4 Compares Performed for 1000 Elements



The graph above shows my four sorting algorithms, batcher sort, shell sort, heap sort, and quick sort, sorting an array up to 1000 elements and the number of compares required for each number of elements. This represents arrays with a large number of elements to observe how the number of compares changes as the size of the array increases. As we can see from the graph, shell sort uses the most compares for the majority of elements compared to the other three sorts. We can also see that quick sort uses the least amount of compares. An additional observation we can make is that not all of the lines are smooth. The use of swapping in the sorts could be causing this as swapping uses three moves. From seeing how high the shell sort line goes and how low the quick sort line goes, we can conclude that shell sort is the least efficient in terms of compares usage for sorting arrays with more elements and quick sort is the most efficient in terms of compares usage for sorting arrays with more elements.

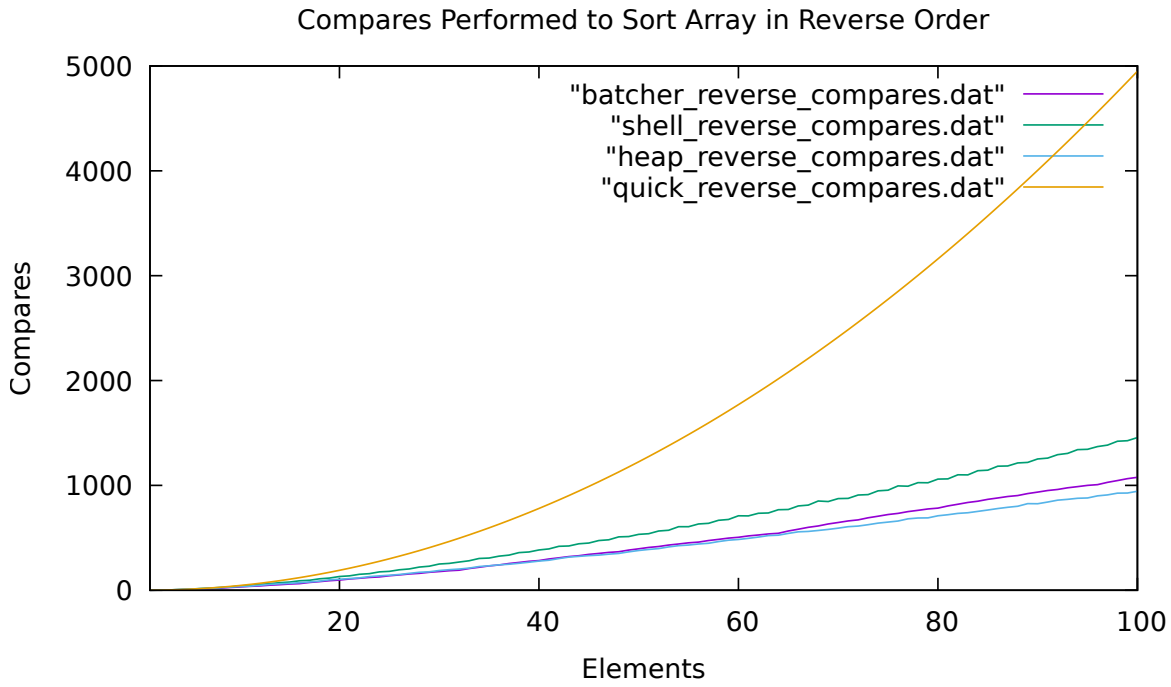
3 Graphs of Sorting Arrays in Reverse Order

3.1 Moves Performed by Sorting Arrays in Reverse Order



The graph above shows my four sorting algorithms, batcher sort, shell sort, heap sort, and quick sort, sorting an array in reverse order from 100 to 1 and the number of moves required for each number of elements. As we can see from the graph above, quick sort uses the most moves by far for the majority of elements. We can also see that batcher sort uses the least amount of moves for the majority of elements. An additional observation we can make is that not all of the lines are smooth. The use of swapping in the sorts could be causing this as swapping uses three moves. From seeing how high the quick sort line goes and how low the batcher sort line goes, we can conclude that quick sort is the least efficient in terms of moves usage for sorting arrays in reverse order and batcher sort is the most efficient in terms of moves usage for sorting arrays in reverse order.

3.2 Compares Performed by Sorting Arrays in Reverse Order



The graph above shows my four sorting algorithms, batcher sort, shell sort, heap sort, and quick sort, sorting an array in reverse order from 100 to 1 and the number of compares required for each number of elements. As we can see from the graph above, quick sort uses the most compares by far for the majority of elements. We can see that for elements 1-around 65, batcher sort and heap sort use around the same amount of compares. It is only after around 65 elements that heap sort starts to use less compares than batcher sort. An additional observation we can make is that not all of the lines are smooth. The use of swapping in the sorts could be causing this as swapping uses three moves. From seeing how much higher the quick sort line goes, we can conclude that quick sort is the least efficient in terms of compares usage for sorting arrays in reverse order. Because the line of heap sort gets lower than batcher sort starting around element 65, we can conclude that heap sort is the most efficient in terms of compares usage for sorting arrays in reverse order.

4 Conclusion

The main takeaway that I had completing this assignment was learning about how different sorts are implemented and why some are more and less efficient than others. I also learned how to use sets and to keep track of stats (moves and compares) when running my sorting algorithms. In addition, I was able to understand more about arrays since that is what I was sorting this entire assignment. Overall, this assignment was very interesting to complete, especially

looking at all the graphs of the sorts side by side to see how efficient or inefficient they were compared to each other.