CSE13S Assignment 3 Writeup

Zion Kang

February 4 2023

1 Implemented Sorting Algorithms on Pseudorandom Arrays

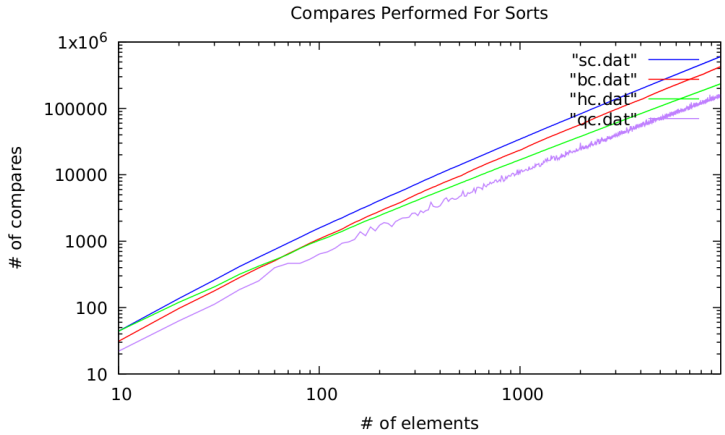

Figure 1: Number of Compares for Sorting Pseudorandom Array

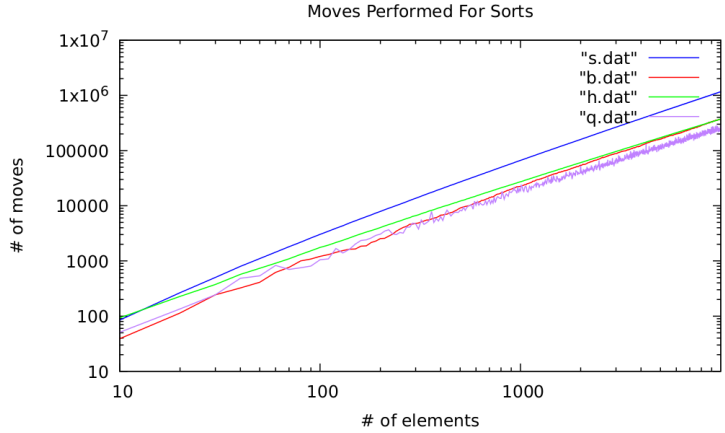

Figure 2: Number of Moves for Sorting Pseudorandom Array

When testing the implemented sorting algorithms on a pseudorandom array, it appears that quicksort consistently performs the best in terms of needing less compares and moves to complete sorting. It could

be said that it is the most efficient and preferable sort because of this. However, there seems to be some inconsistency in this trend as can be seen in the jagged line plotted for Quick Sort (purple line). The jagged line shows that it's divide and conquer method (partitioning) has some limitations in the sense that when the getting the pivot, if it is the largest or smallest, it can severely affect the performance of the sort.

Shifting our attention to the worst sort of the bunch, we see that Shell Sort consistently has the most moves and compares. With a time complexity of $O(n^2)$, it is quite clear even when thinking about how it sorts (using gaps to continuously compare items in an array), that it is the slowest sort out of the bunch.

When comparing the other two sorts, Batcher sort and Heap sort, with time complexities of $O(nlogn)$, we can see that Batcher's performance worsens as we increase the number of elements in the array (as seen in the red line in the plots above). Heap sort starts out with needing both more compares and moves but ends up performs better as we increase the number of elements in the array which is seen in the two lines intersecting (green line representing Heap sort).

2 Implemented Sorting Algorithms on Reversed Order Arrays

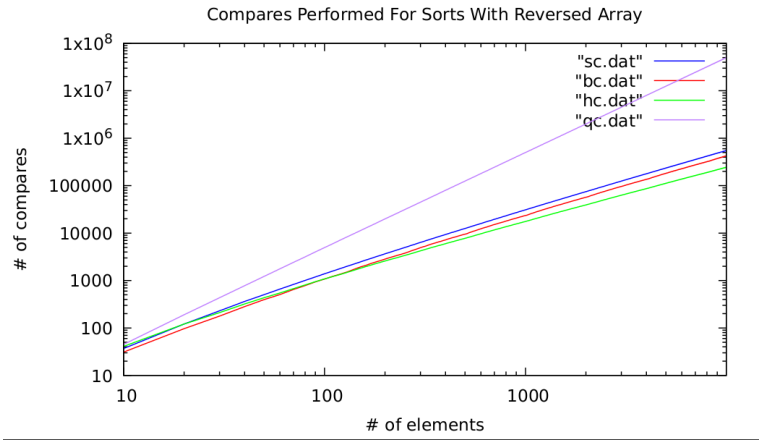

Figure 3: Number of Compares for Sorting Reversed Array

The limitation of quick sort regarding its pivot is clearly displayed in its poor performance when sorting reversed arrays. As it tries to sort based off of a pivot element that is the largest/smallest, its performance drops drastically, requiring more compares/moves as a result of more recursive calls on itself. This divide and conquer method of partitioning that quick sort uses seems to only be effective completely random orders (or at least somewhat randomized order). Based on its performance with a pseudorandom array, it could be assumed that if this sort were to contain another helper function that shuffled the array a little, it would drastically improve its performance.

When comparing the other sorts, it appeared that they followed the same general trends as sorting a psueodorandom array, at least in the plot for the number of compares needed. However, when taking a look at the plot for moves performed, it is clear that this is not entirely true. More specifically, the batcher sort appears to have no moves performed while its previous competitor, heap sort followed the same trend. I believe that this is due to the batcher_sort function not utilizing its *comparator* helper function to swap elements, resulting in the plot showing no moves for batcher.

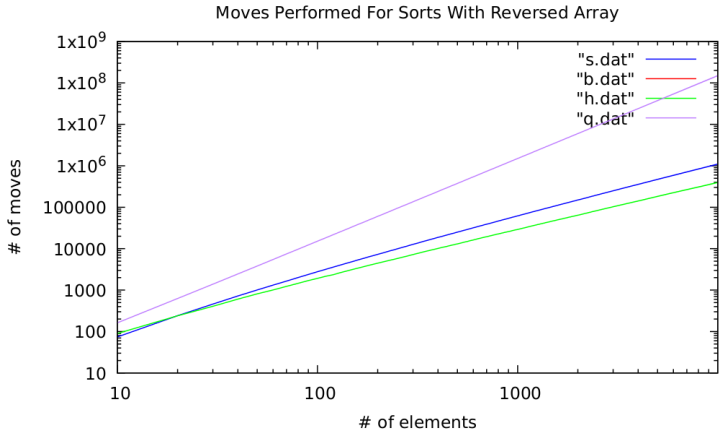

Figure 4: Number of Moves for Sorting Reversed Array