Assignment 3: Writeup

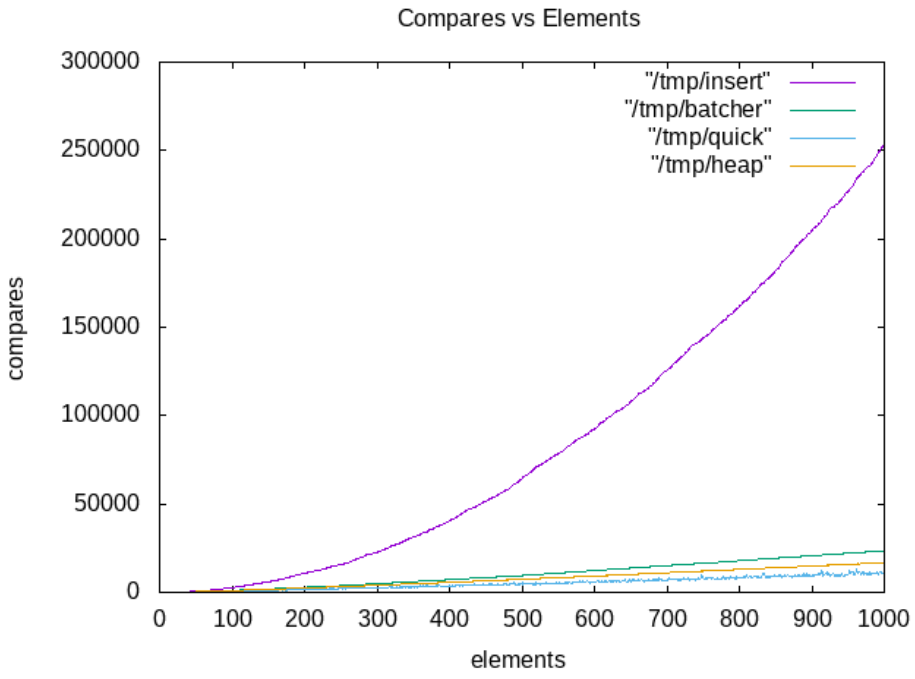
Madison Ormsby

January 27, 2022

1 Introduction

This document will be analyzing the trends found in the differing sort methods implemented in the file sorting.c. Each of the sorting methods have wildly different methods used and each of the steps have been tracked using the file stats.c and printed out after each sort along with the sorted array using ./sorting. However, some of the sorting methods excel under certain circumstances due to their layout while others struggle. Some of the things I struggled with in this project were function pointers, creating arrays, and specifically the heap.c function. A few key points that will be analyzed in this essay are how many compares are made based on the amount of elements, how many moves are made based on the amount of elements, and the ratio of moves to compares for each sort at any predetermined amount of elements in the array.

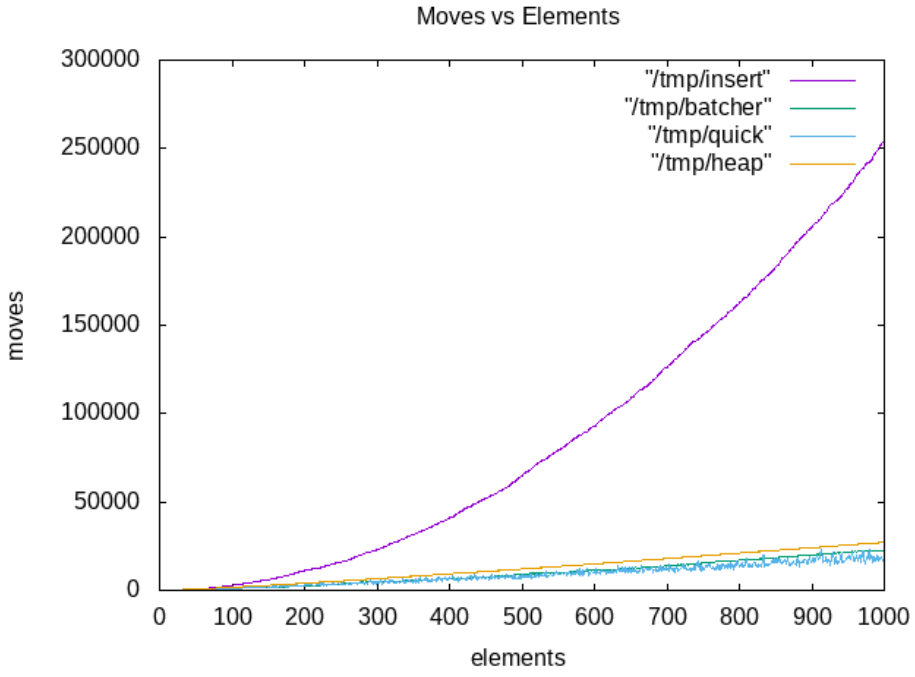
2 Compares to Elements



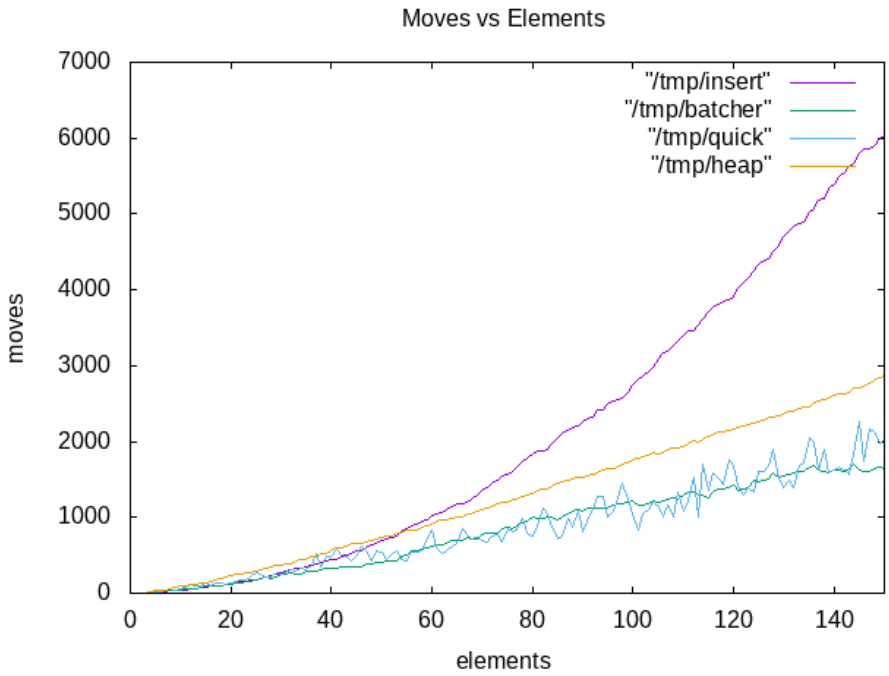
As shown above, as the number of elements grows, so does the number of compares. However, it can be shown that insertion sort takes up a much larger amount of compares than any of the other sorts. In addition to that, it can be shown that batcher sort takes up the second most, heap sort taking the third most, with quick sort taking up the rear. However insert sort and quick sort are much bumpier than the

other two. As the number of elements in the arrays grow, insertion sort takes an exponentially larger amount of compares for each element as its curved line demonstrates. However batcher and heap sort seem to grow linearly without much visible change in slope. Unlike all of them however insert seems to be a jagged line that steadily rises but doesn't really pass quick. Around 100 to 200 elements it can be seen that heap sort actually beats batcher sort by just a small of compares while quick dodges between the two. It can be concluded that when looking to sort with the smallest amount of compares insertion sort will never be the quickest solution and for smaller amounts of elements, it doesn't really matter which of the other three is used.

3 Moves to Elements

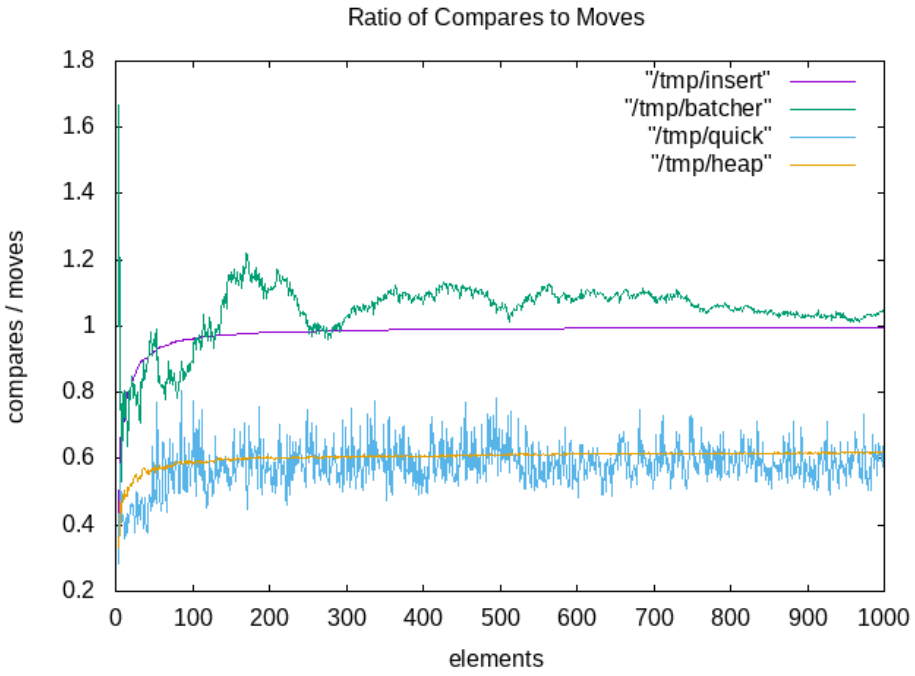


Very similarly to the "Compares vs Elements" graph, insertion sort absolutely knocks it out of the park with how many moves it requires. In fact, the insertion sort line looks near identical to the line in the previous graph (this will be explained in the next section). After insertion sort though, heap sort takes up a larger amount of moves than both batcher and quick sort, with the latter two trading quite frequently for third place. Once again quick sort is a very jagged line that seems to spike in seemingly random areas of the graph, overlapping with batcher sort. From elements 0 to around 600, the two lines for batcher sort and quick sort are near overlapping and if you were looking for efficiency in moves, it would not seemingly matter which of the two is used.



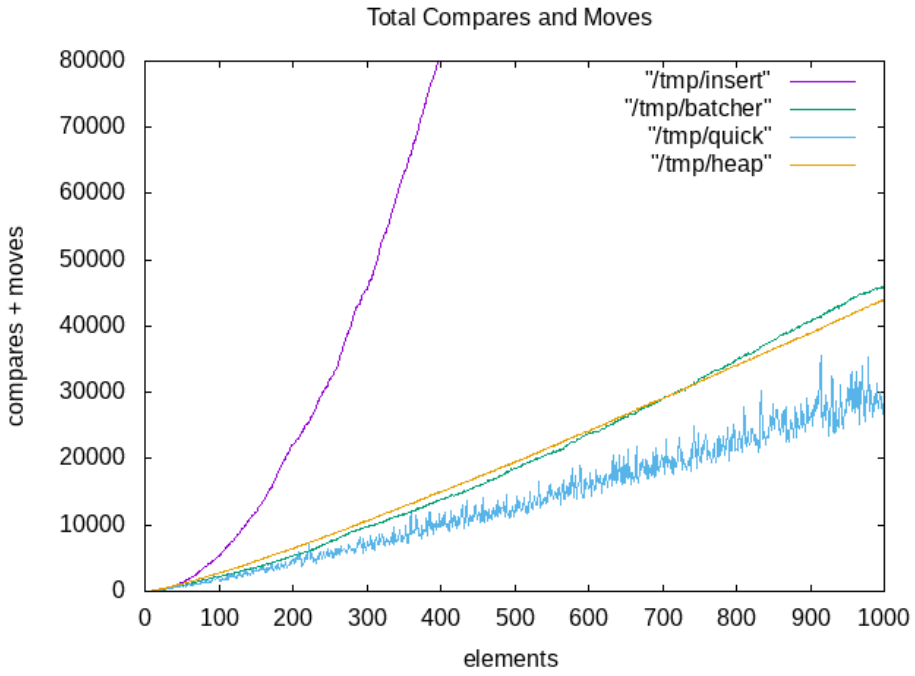
As shown above it can be seen that even among smaller arrays, insertion sort does poorly. However, for arrays smaller than around 58 elements, insertion sort actually performs better than heap sort. Shown up close, it can also be seen that batcher sort is not the smooth line that it seemed to be from a far away distance but as a matter of fact also seems to be quite jagged.

4 The Ratio



So from the past two graphs it can be seen concluded that insertion sort always loses, batcher and heap are relatively linear, and quick sort is very very jagged and does not offer much reliability in sorts with smaller elements. Although there is seemingly no pattern to many of these sorts, when comparing the amount of compares vs the amount of moves very specific lines start to appear. For example, the exponential trend of insertion sort shows a line similar to the sqrt() graph, with two asymptotes at $y = 1$ and $x = 0$. The more elements insertion sort has, the closer the amount of compares gets to the amount of moves until they are equal. A very similar trend is shown with heap sort, but rather than reaching an asymptote at $y = 1$, it levels off at $y = 0.6$. This means that no matter how many elements are input into the heap sort, there will always be fewer compares to the amount of moves in heap sort. Along the heap sort line, quick sort is once again a jagged mess. However it almost seems to follow a trend of bobbing up and down the heap sort line. By looking at this we can see that in terms of efficiency for compares and moves, although quick sort may be the quickest, it is also the least reliable so you may get your result faster or slower than heap sort.

5 Conclusion



From the graphs shown above, it can be concluded that for smaller arrays, insertion sort is not necessarily a bad idea. However, as the amount of elements grow larger, so does insertion sort's processing time and thus can be concluded a less efficient sort than the other three. Meanwhile for the most efficient of all the sorts, appears to be quick sort. Even as the number of elements rises, quick sort maintains a steady gap between it and batcher sort and heap sort after around 250 elements. However, it can be seen that between batcher sort and heap sort, heap sort is less efficient than batcher sort until around 750 elements. After that it becomes more efficient and works better for larger amounts of elements.

6 Credits

- Brian Mak noticed I had an extra parenthesis in my heap.c along with helping me implement stats.c during his office hours on 01.24.2022 and 01.21.2022
- Ben's gdb document helped when I was having many segmentation faults in my code