Sorting Algorithm Analysis

Moore Macauley
University of California, Santa Cruz

October 23, 2022

1 Introduction

In this assignment, I was tasked with the implementation of four sorting algorithms, bubble sort, quick sort, heap sort, and shell sort, and analyzing the performance of each algorithm. In this regard, I found that quicksort was generally the best algorithm tested, making fewer moves and comparisons except in a few extreme circumstances. Heap sort was generally much more consistent in the number of moves and comparisons it makes, and made fewer comparisons than shell sort, making it a good choice if consistency was important and comparisons are faster than moves. Shell sort, on the other hand, while a bit erratic in the sorts and moves it made, did generally have fewer moves than heap sort, making it better if moves are faster than comparisons. Finally, bubble sort was almost always the worst choice, with the notable exception of when a list was already sorted, in which case it was actually the best choice. I reached these conclusions by graphing the number of moves and comparisons made by each sorting algorithm in a variety of scenarios, which can be seen below.

2 Relatively Small Array Sizes

For the first graph, which can be seen in figure 1, I simply graphed all the comparisons made by each sorting algorithm for random arrays between size 1 and size 1000.

Unfortunately, bubble sort had a few more comparisons than all the other algorithms, making the graph rather hard to read. This does, however, exemplify how much faster a worst case of $O(n^2)$ grows

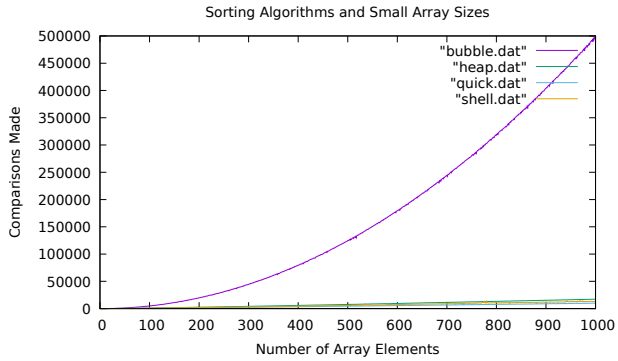

Figure 1: Not the most useful graph

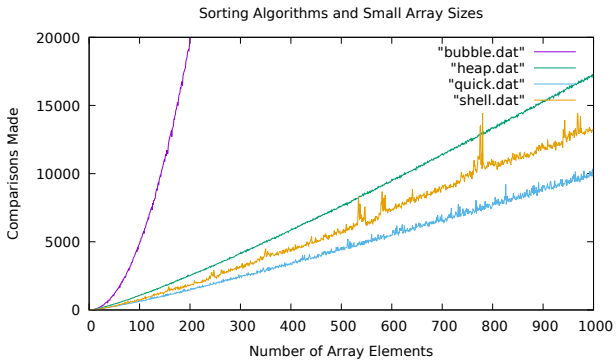

Figure 2: Figure 1 with a restricted range

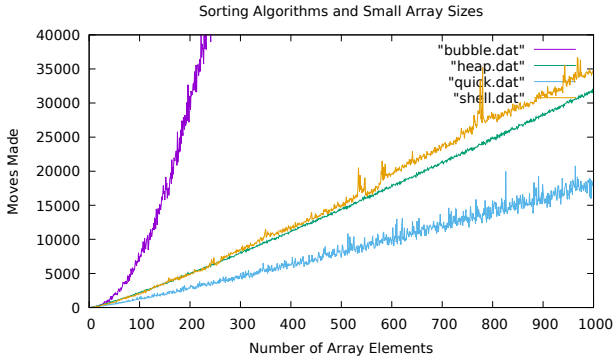

Figure 3: Moves, now

compared to O(nlog(n)), which are the theoretical growth rates for bubble sort and the other three sorts, respectively (technically speaking, quick sort has a worst case of O(n^2), but the average case is closer to nlog(n)).

However, figure 2, which is simply figure 1 with a limited range, gives us an idea of how the other three sorts behave. Quick sort is the fastest, making the least number of comparisons in all circumstances. Next is shell sort, which usually has the most comparisons, except for a few cases where the number of compares spike. Finally, heap sort makes the most comparisons, asides from shell sort, but there is almost no variation in how fast the number of compares grows, creating an almost straight line.

On the other hand, in figure 3, which shows the moves made in the same scenario, the situation is reversed. The general shape is the same, with heap sort being consistent, quick sort being the fastest, and bubble sort the slowest. However, while heap sort has more comparisons, it has fewer moves than shell sort.

The difference in compares and moves with quick and heap sort does make some sense. Fixing a heap, in particular, requires a lot of comparisons, as every left and right child must be compared and each other before they are then compared to the parent. Shell sort, on the other hand only needs to compare the value before the gap to the value after the gap, which is comparatively few. On the other hand, due to how shell sort and the gaps work, shell sort will probably be putting the swapped element into the wrong place, requiring it to be moved again, while there is much less of this in heap sort.

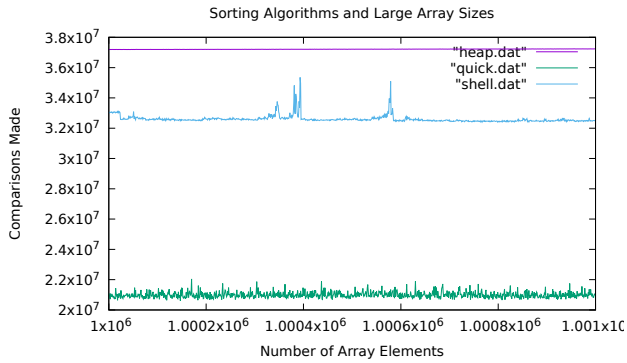

Figure 4: Large comparisons

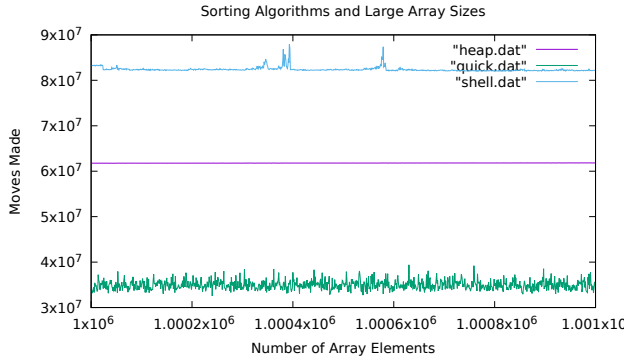

Figure 5: Large moves

Similarly, heap sort being consistent also makes sense. Fixing a heap will take a very consistent number of moves and comparisons, as the child taken from the bottom and placed at the top will almost always have to move from the top to the bottom. Similarly, building a heap always requires the entire array to be iterated over. The only part that could have any real variability would be moving an array up a heap, with some items already being the right place being fairly common. But this is still only one, relatively small portion of the algorithm, making its consistency make sense.

The spikes in the moves and compares for shell sort, however, are harder to explain. The one around 600 elements is relatively easy to explain, with the number of times a gap is calculated increasing just before 600 elements. My theory is that this spike is caused by the gap about to change, as the gap before a gap of 1 will be relatively large, causing the array to be far less sorted than it would be normally when the smallest gap is used. However, this does nothing to explain the spike at around 800 elements, as the last change in the number of gaps happened around 600 elements, and the next gap change isn't until well after 1000 elements. As such, I don't really know what causes these spikes.

3 Larger Array Sizes

In figure 4 and 5, you can find the number of comparisons and moves made by all the algorithms except bubble sort for arrays sized 1 million to 1 million, 1 thousand. Bubble sort's exclusion should be self

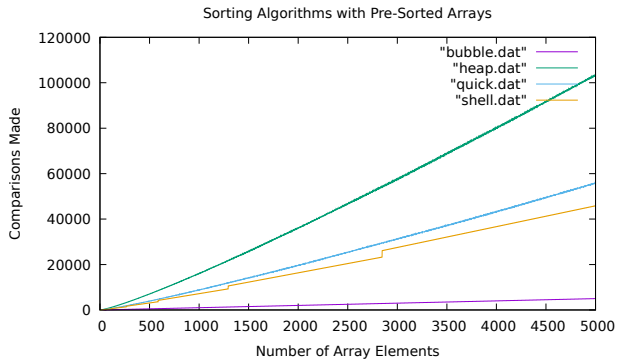

Figure 6: Pre-sorted comparisons

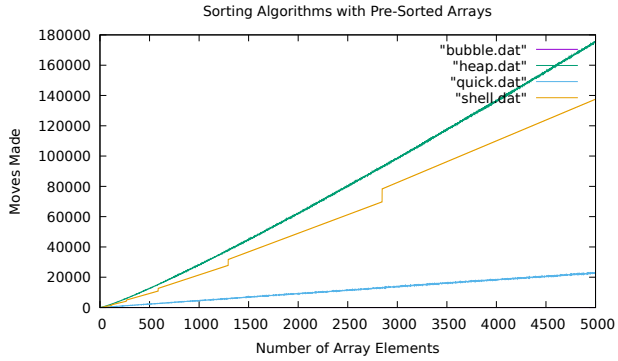

Figure 7: Pre-sorted moves

explanatory, given figure 1.

These graphs are, in all honesty, fairly uninteresting. All they serve to do is confirm the patterns noted in the small arrays have remained true, and have in fact increased, when the arrays got larger. Aside from this, the only notable thing is how the rate of growth seems to have tapered off, and the lines have become rather flat. This, however, makes sense, as with the growth being partially logarithmic, a decrease in growth as numbers get larger is exactly what we should expect.

4 Pre-Sorted Arrays

These next graphs show the performance of the sorting algorithms when passed an already sorted array, with figure 6 showing comparisons, and figure 7 showing moves.

Most surprising is that bubble sort seems to be the most effective sorting algorithm. The reason for this is fairly simple, however, as bubble sort moves at least one element to its exact right place after every iteration of its loop, it makes sense that it would perform well with lists that are already sorted or are almost sorted, as all it needs to do is iterate over the list once, move nothing, and declare it sorted.

Heap sort seems to be virtually unchanged, but this also makes sense. To put it simply, the things that made it very consistent haven't changed, so it remains consistent. A admittedly, this time it serves to make it slower, but depending on what was being done, a sort algorithm very consistent in the time it

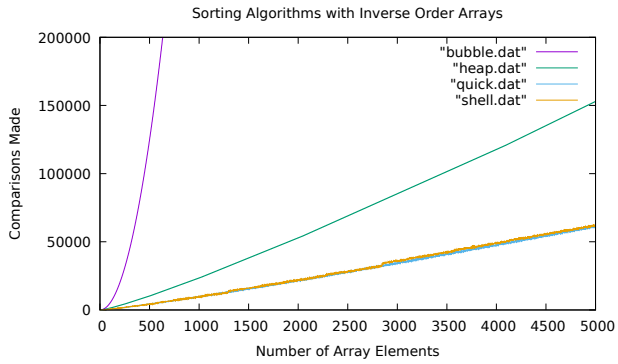

Figure 8: Reverse order comparisons

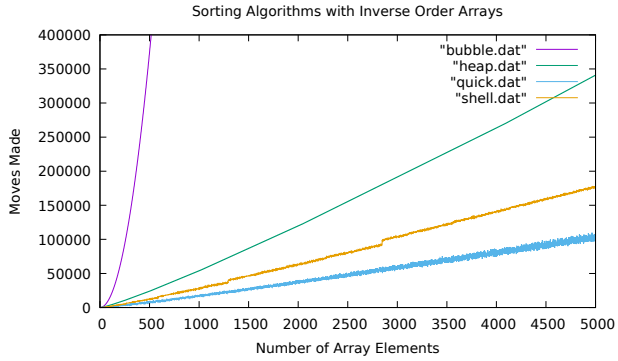

Figure 9: Reverse order moves

takes could be very useful.

Quicksort seems to have decreased the number of moves and comparisons it is making, but not by much. All in all, not too interesting.

Finally, shell sort has dramatically decreased in the number of moves and comparisons it makes, to the point where it is now making fewer comparisons than even quick sort. This makes sense, as similarly to bubble sort, though not to the same extent, it simply has less to do when the list is in order. It iterates over the list, checks to see that things are in order, and not much more. Unfortunately, however, there are some cliffs in the graph that can be observed, and similarly to the spikes in figures 2 and 3, I have no good explanation for them.

5 Reverse Order Arrays

The final set of arrays I used for testing were arrays in reverse order. This can be seen in figures 8 and 9.

Once again, bubble sort is performing the worst out of any of the sorts, and heap sort remains relatively consistent with what was seen in figures 2 and 3, which is what was expected.

Shell sort, on the other hand, gives some rather interesting data, as it seems to have almost improved with the reverse order. This does actually make some sense, as the first gap used is almost half (technically five elevenths) of the length of the array, which would effectively cause the first half of the elements

to swap with the second half. This would get the array shockingly close to in order almost immediately, greatly decreasing the amount of sorting needed to be done. Once again, however, we see the same odd cliffs, and I have no good explanation for them.

6 Conclusion

Based on all this data, we can determine when best to use different sorting algorithms. Bubble sort should only be used when the array is likely to already be in order, or very close to ordered, and should be avoided at all other times. Heap sort is very consistent in the amount of time it takes to run, and if that is important, could be very useful. Shell sort is best run when an array is likely to be in reverse order, or very close to it, and quick sort is the best option in all other scenarios.