Assignment 3 Sorting: Putting your affairs in order

Writeup Document

Derrick Ko - Winter 2023

1 Summary

- What you learned from the different sorting algorithms. Under what conditions do sorts perform well? Under what conditions do sorts perform poorly? What conclusions can you make from your findings?
- Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements. Your graphs must be produced using either gnuplot or matplotlib. You will find it helpful to write a script to handle the plotting. As always, awk will be helpful for parsing the output of your program.
- Analysis of the graphs you produce.

2 What I learned

Fastest to Slowest Algorithms (tested on default settings)

1. Quick Sort
2. Heap Sort
3. Batcher Sort
4. Shell Sort

Every sorting algorithm has their strengths and weaknesses. For example, quicksort being the fastest sort we implemented has different use cases. Its worst case being O(n^2) on an already existing sorted list. Surprisingly, shell sort works the most efficiently in looking through that list with a time complexity of O(n).

2.1 Quick Sort

Quick sort has an average time complexity of O($n \times log(n)$). There are many implementations of the pivot, such as the first element, last element, random element or median element. It is an in-place sorting algorithm, which means it requires a fixed amount of memory regardless of input size. Quick sort perform s effectively on huge data sets when memory utilization is a key problem.

2.2 Heap Sort

Heap Sort has an average, best, and worst time complexity of O($n \times log(n)$). This is the same as quick sort but Quick sort on average performs better. This uses a binary heap data structure. It has a space complexity of O(1), because it sorts the elements in place, without using any additional memory.

2.3 Shell Sort

Shell sort has a best-case time complexity of O for an already sorted array (n). This is because the inner loop (insertion sort) will only run once for each array element, and each trip through the outer loop will lower the gap size by a factor of two until the gap size is one. The procedure is thus identical to a simple insertion sort, which has a best-case time complexity of O(n) for an already sorted array.

2.4 Batcher Sort

Batcher sort has an best, worst, and average time complexity of O($n(logn)^2$). Batch sort is often used in situations where the data set is too large to be sorted in memory, such as in external sorting. It can also be used to sort data sets that are too large to be transferred over a network in a single transmission, as the data can be sorted in batches and then transferred piece by piece.

3 Graphs

The two reversed graphs were created by writing another function to reverse the generated input from my original random array generator.

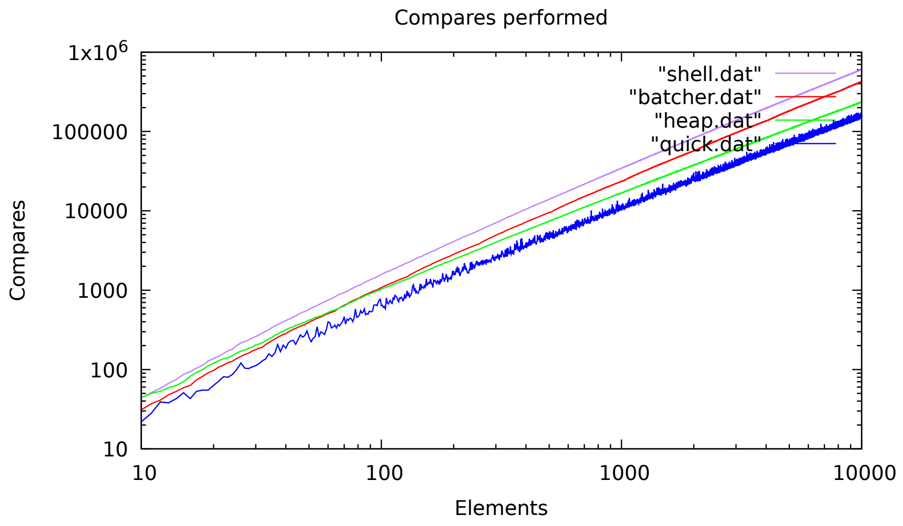

Figure 1: This is a graph of Elements and compares. Some details I noticed is that Quick sort's line is not straight. It is bumpy which is caused by implementation of the pivot

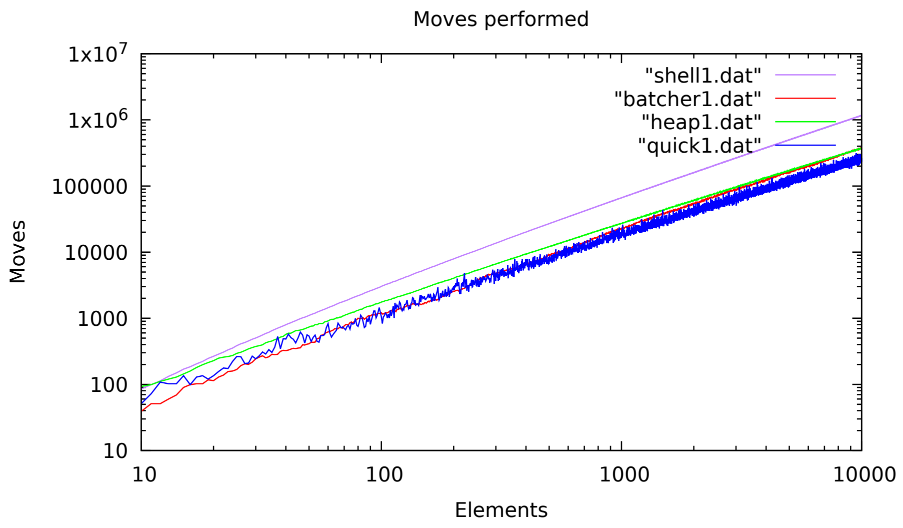

Figure 2: This is a graph of Elements and moves.

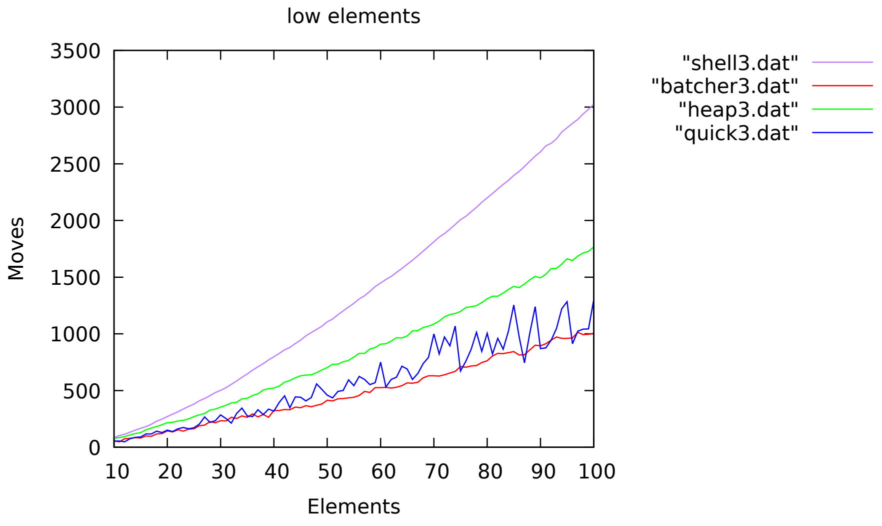

Figure 3: It seems with lower elements, batcher sort on average has lower moves than quicksort which seems like this is leaning a more worse case for quicksort performance.

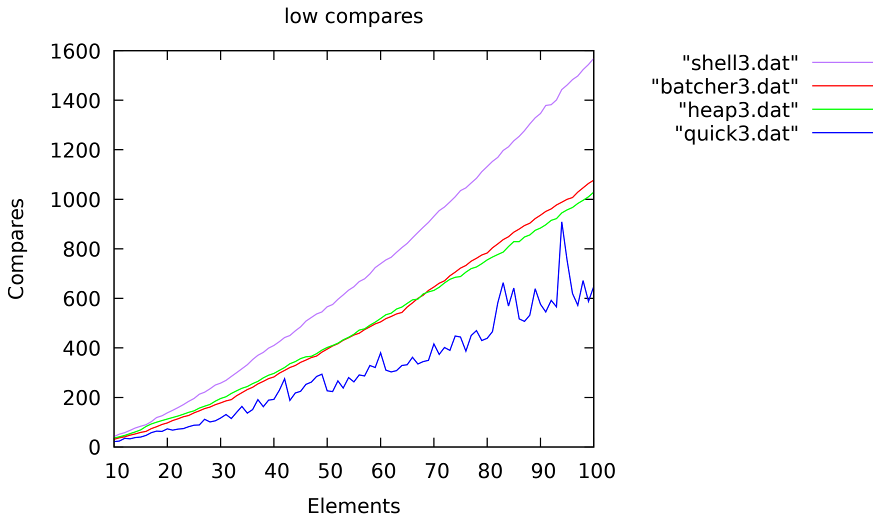

Figure 4: The compares are lower with quicksort. Batcher and Heap are very similar as well.

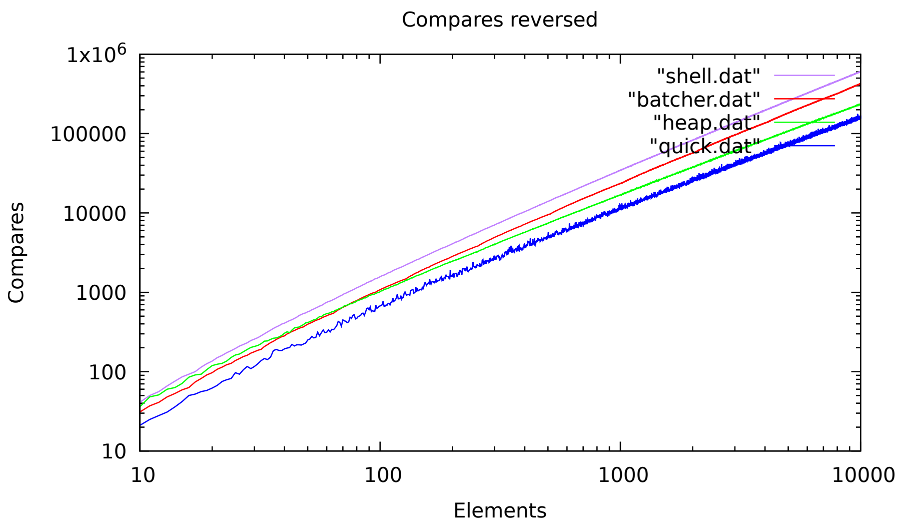

Figure 5: This looks like how the graphs would look when I ran the tests with 100 elements. Batcher and Heap have interesting data under the 100 element range. Batcher starts out with less compares than heap then they intersect and heap sort takes over.

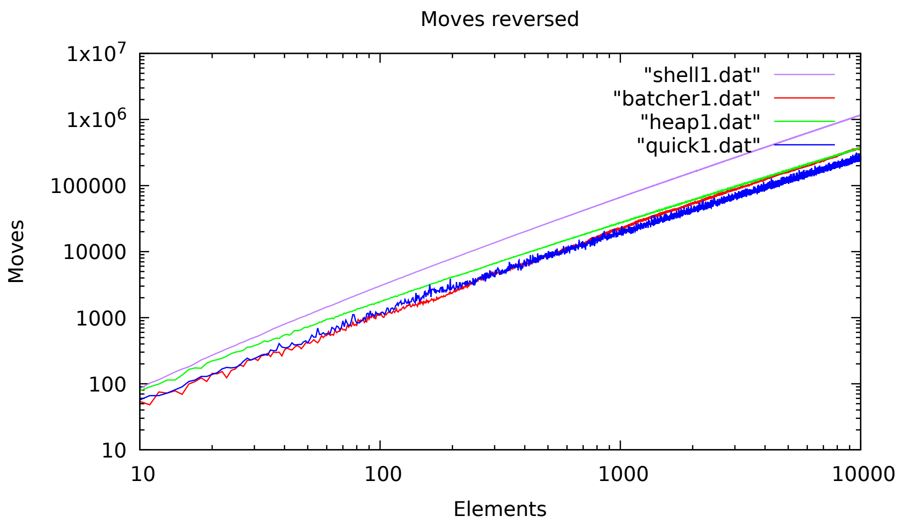

Figure 6: The reversed images are almost exactly the same. The variation between the two is barely noticeable but the difference is negligible within the data. Batcher looks like during the lower elements it has a lower moves compared to quick sort until around 300 elements where quick sort gets better.