Sorting:
Which House Do They Belong To?

Lucais Sanderson

5 February 2023

1 Intro

Sorting algorithms are instrumental in computer science. They're useful in a wide scope of operations such as their use in abstract data types. In this assignment, we implement four sorting algorithms in C: Batcher Odd-Even Merge Sort, Shell Sort, Quick Sort, and Heap Sort.

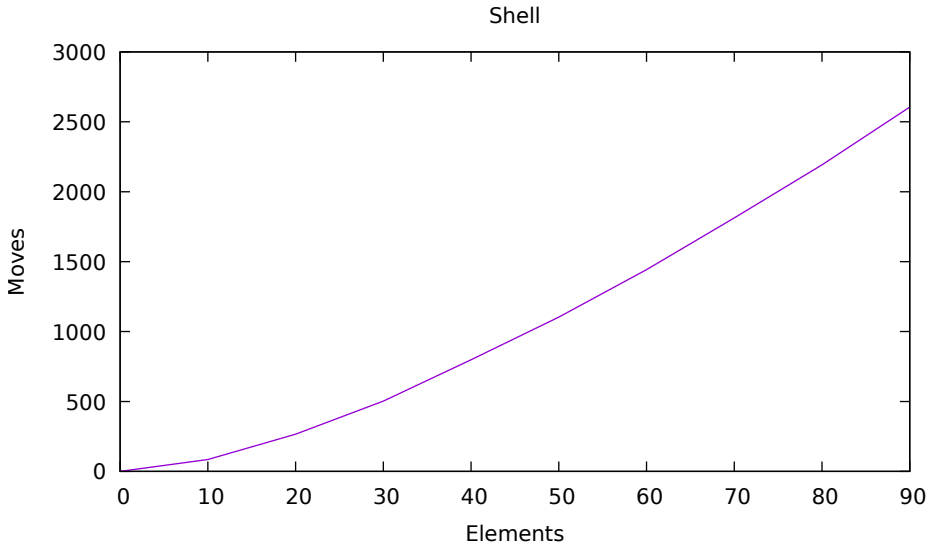
We then ran each sort with small values and large ones to test their efficiencies and limits. Below, we will analyze our findings and compare each sorting method's results.

2 Shell Sort

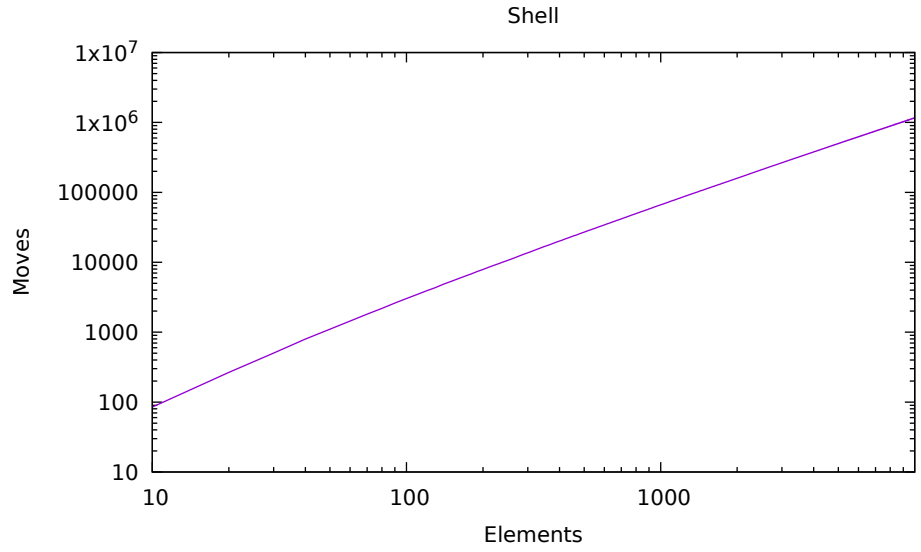
Shell Sort is arguably the simplest out of the mix. Designed by Donald L. Shell, Shell Sort compares elements some **gap** apart. This gap starts very large, and continuously grows smaller and smaller, swapping elements where needed.

This method performs better than the typical nearest neighbor sort method, however, its actual time complexity is determined by the gap **sequence** the algorithm uses. In our case, we'll be using the *Pratt sequence*, given by $2^q 3^q$, also referred to also as a 3-smooth.

First, let's look at Shell's performance with a small number of elements.

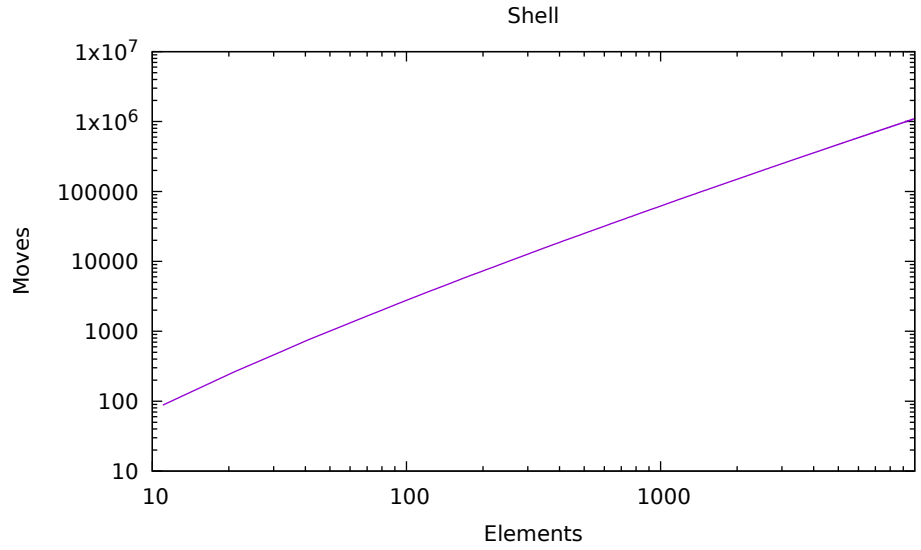


The line appears pretty smooth, staying predictable, and almost takes on a parabolic shape. Next, let's see how it does on larger scales.



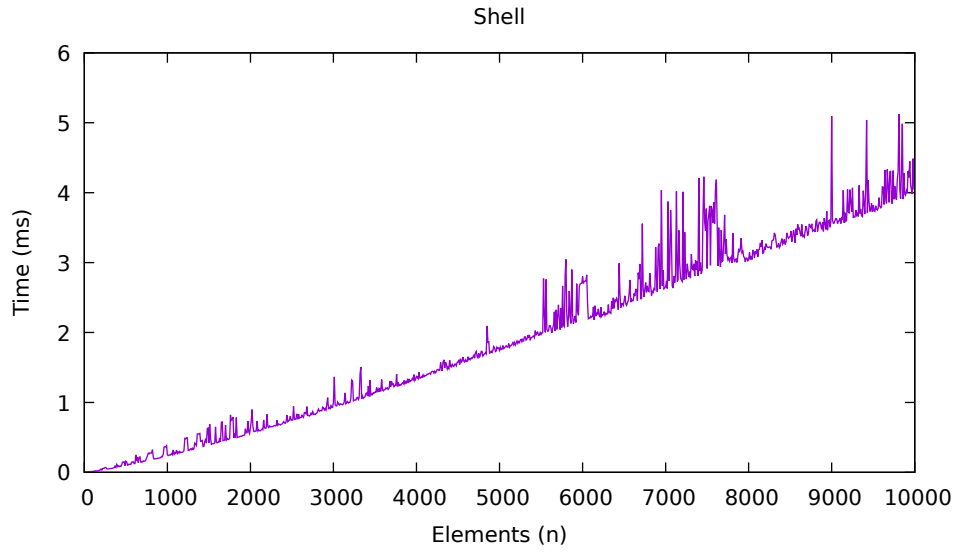
Shell seems to keep its consistency in regard to its operations (moves). The line is smooth and still predictable. However, the parabolic shape goes away and we obverse more a linear or $n \times log(n)$ (n : number of elements) curve. This shows us that Shell is probably better suited for larger sets of elements because $n \times log(n)$ is better than n^2.

An interesting test for any sorting method is to see how it does for an array of elements in reverse order. Below are the results of just that.



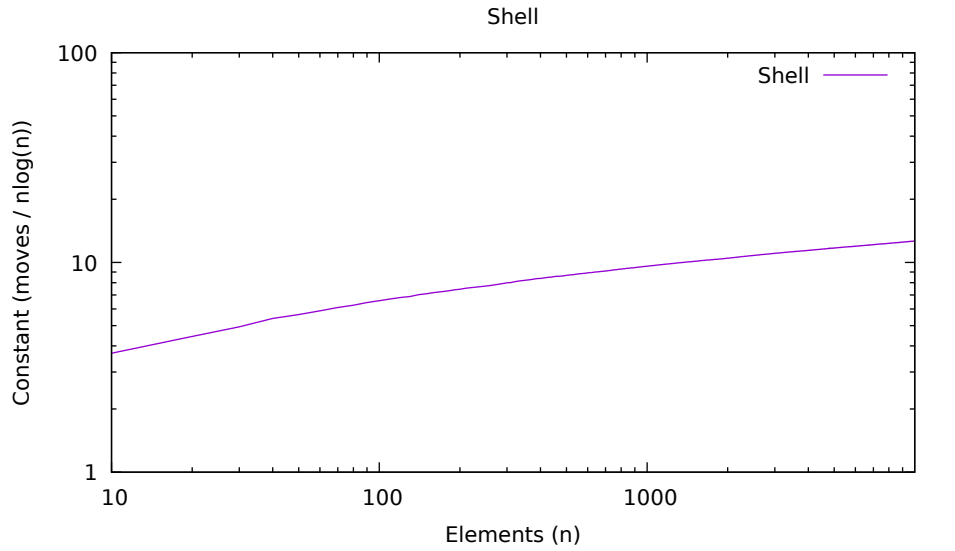
Giving Shell Sort an array in reverse order doesn't negatively affect the operation efficiency it seems. In fact, it doesn't seem to deviate much at all from just feeding in a random set of elements.

Now, let's focus our attention on the time taken for each number of elements.



For the most part, Shell's time stays consistent (in relation to other methods shown later). At around 5,500 elements, we see the time start to spike here and there.

Shell Sort performs on a $n \times log(n)$ complexity for the best case. (via Wikipedia) Dividing our number of moves by this complexity, we can obtain the constant for Shell Sort:



With the given sample size (we end with 10,000 elements), we see the constant converge around 10.

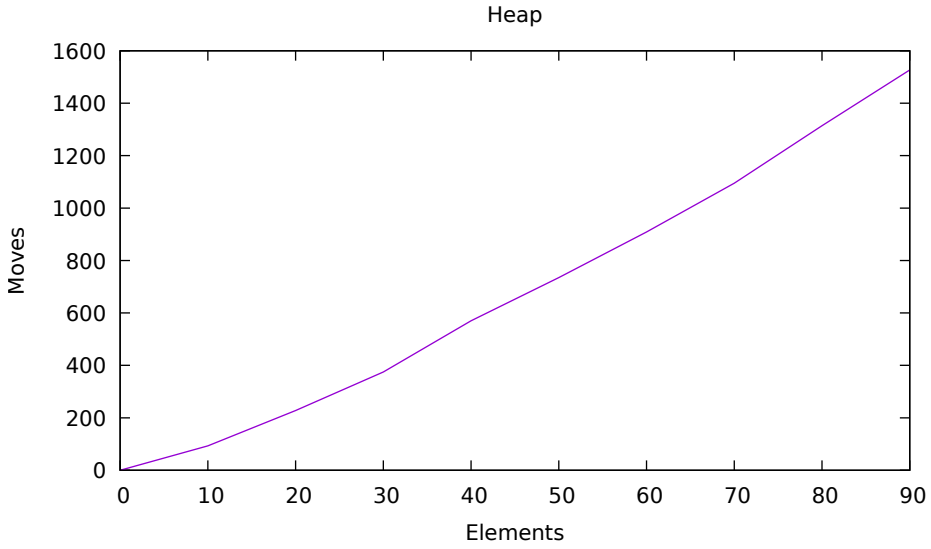
Overall, Shell Sort is a very stable method and our implementation seems to take on the expected $n \times log(n)$ complexity. Simple and effective.

3 Heap Sort

Heap Sort, developed by J. W. J. Williams in 1964, is a sorting algorithm using a special type of binary tree. Specifically, we will use a max heap in which each parent node is greater than its child

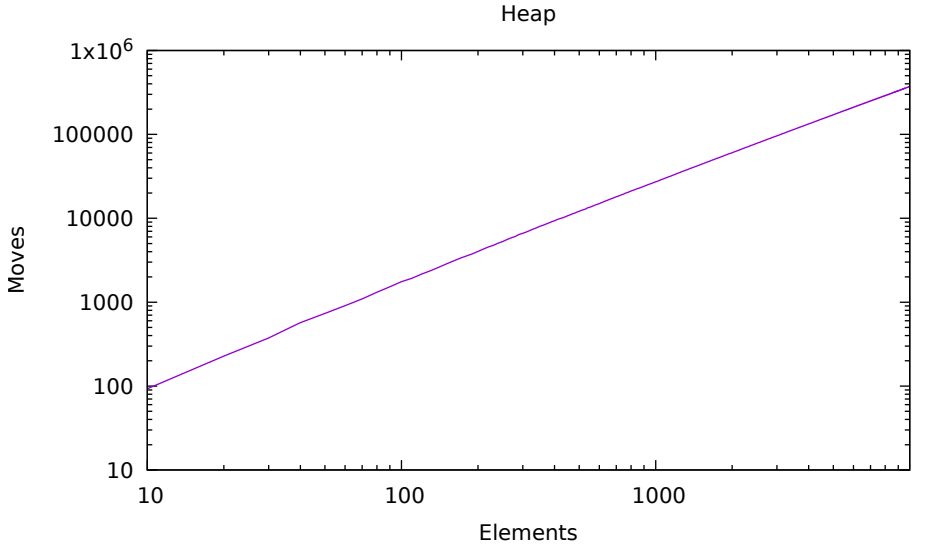
nodes. First, the heap has to be built from the array. Then the root node (topmost node) is moved to the end as it will be the greatest. After that, the heap must be rebuilt. This process continues until the array is sorted.

Here is Heap Sort on a small scale:



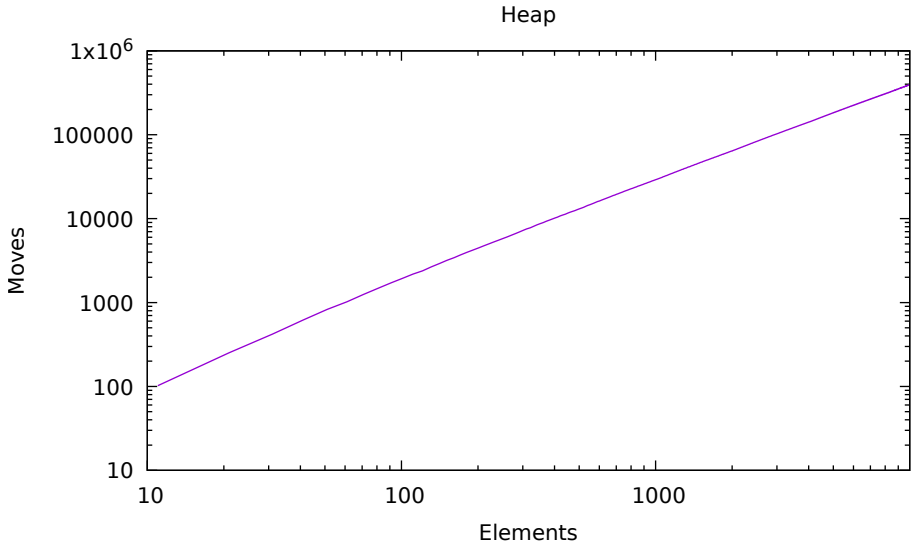
Again, the line appears smooth and almost takes on an x^2 curve, however it's probably closer to a linear curve.

Scaling up to very large element sets, we get:

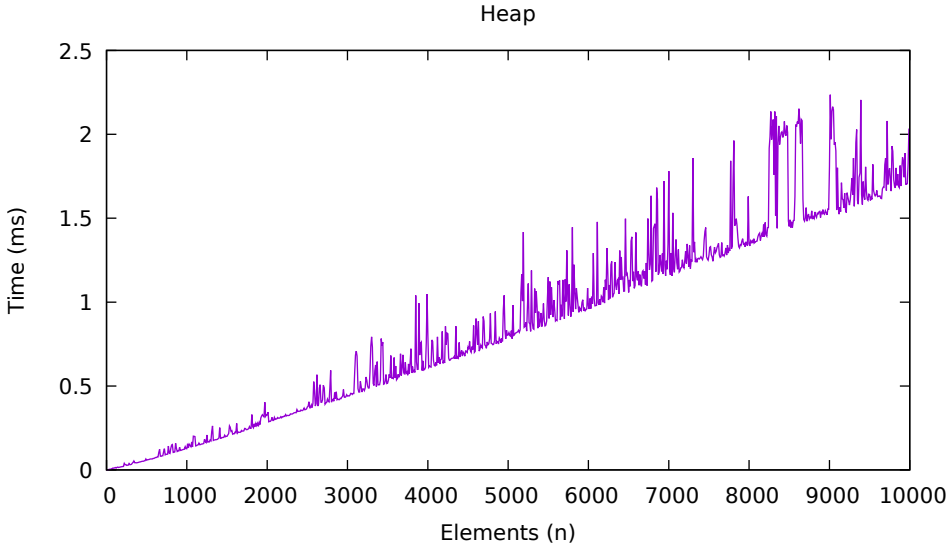


We yield smooth lines for the most part and although it looks linear it's probably closer to an $n \times log(n)$ curve.

Next, we test its reaction to a reverse-ordered set of elements.

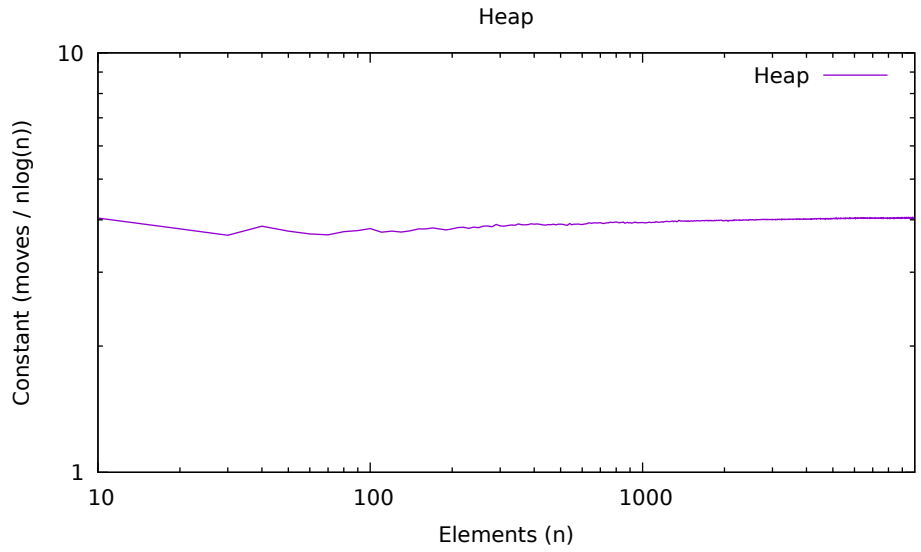


There doesn't seem to be much of a change from just a randomized set of elements. Let's compare the time it takes.



Overall, the time Heap takes is more efficient than Shell. However, it seems to be less stable over the domain.

The best-case complexity of Heap Sort is $n \times log(n)$ (via Wikipedia), and we divide the moves by this to obtain Heap's constant:

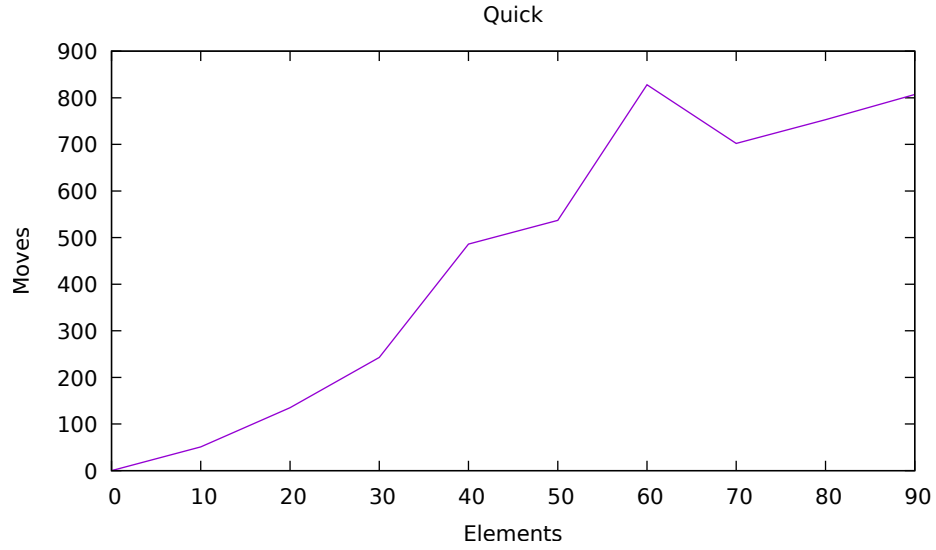


This one converges a little more nicely, coming towards 1,000.

4 Quick Sort

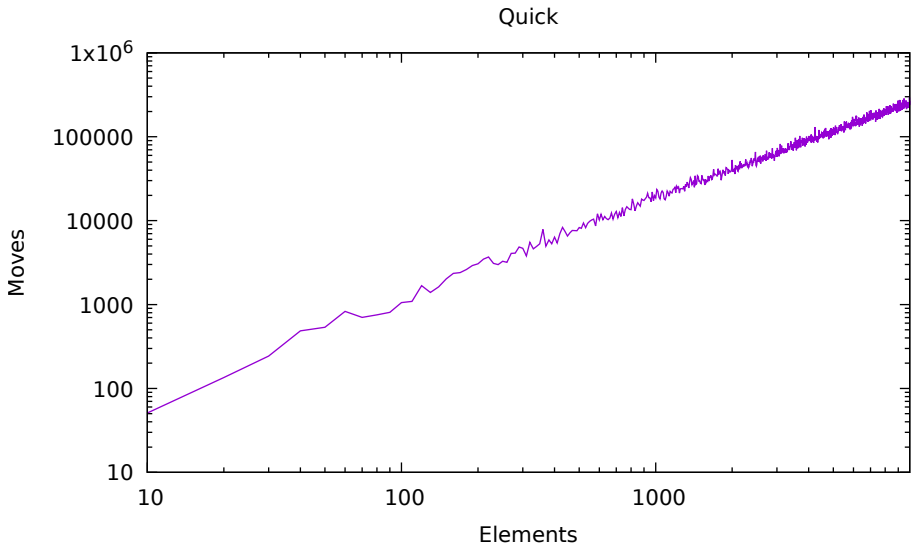
Quick Sort, designed by C.A.R. "Tony" Hoare, sorts an array by partitioning it into two sub-arrays, in between which is the pivot. This partition is done recursively until the array is sorted.

First off, let's see how it reacts to small array sizes.



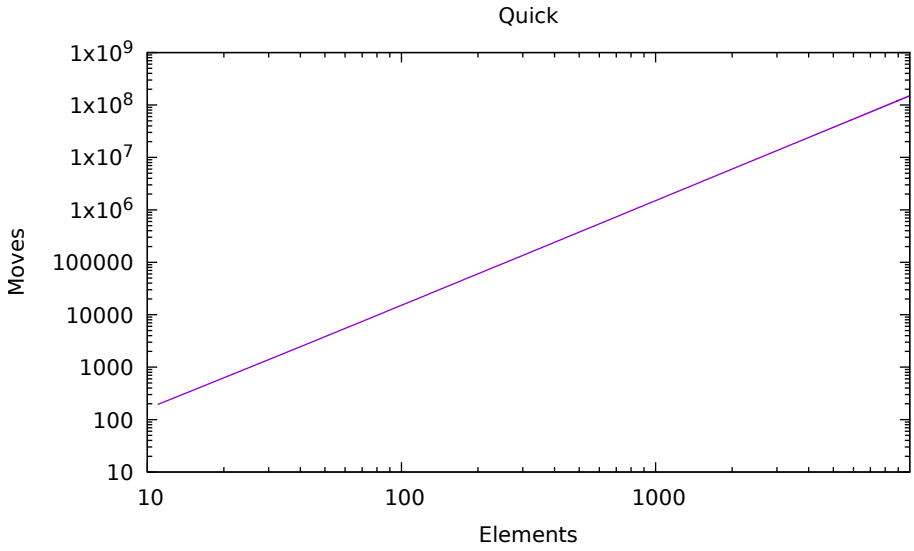
Unlike Heap and Shell sort before it, Quick Sort tends to look a little more volatile. The lines aren't nearly as smooth, *however*, the `moves` count is *far* less than its predecessors.

This trend continues as the element count scales.



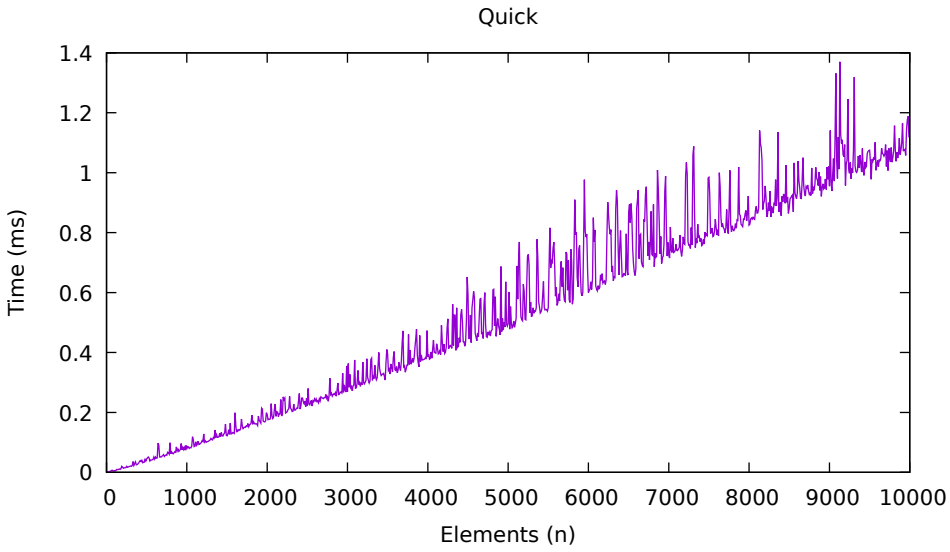
With greater element counts, we can see Quick Sort does well to keep `moves` down compared to Shell and Heap. Despite this, its lines are far less smooth and have a bit more shakiness to them.

However, when we throw a reverse-ordered array at it:



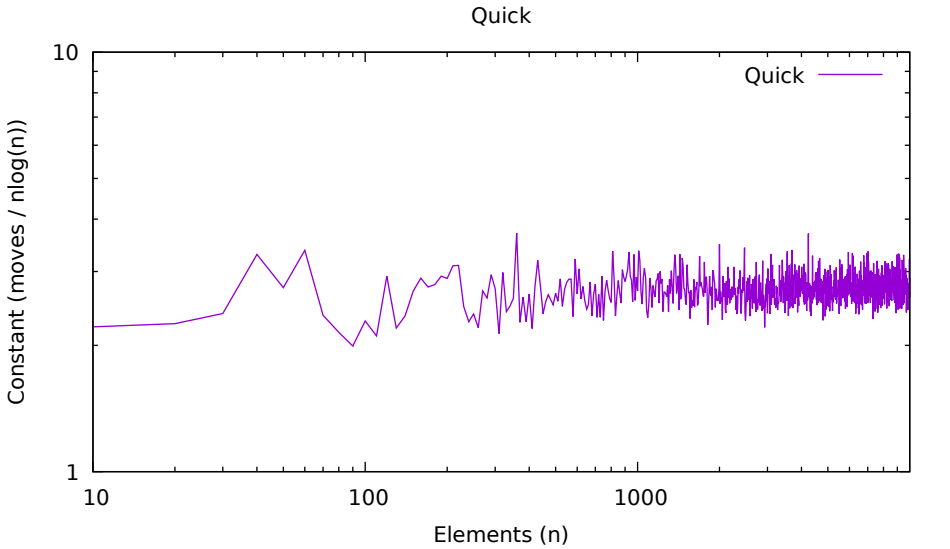
Interestingly, with reverse-ordered elements, Quick's lines smooth out but the moves (at 10,000 elements) increased by almost 3 orders of magnitude! The smoothness of the lines could just be due to the fact our y-axis is scaled up by 3 orders of 10. A possible explanation for this could be due to Quick utilizing recursion, combined with sorting reverse-ordered lists meaning you need to move every single element resulting in this increased number of `moves`.

Let's analyze its time efficiency.



From what we've seen thus far, Quick Sort lives up to its name; it's the fastest with time peaking at 1.4 milliseconds at around 9,000 elements.

Now we'll compare it with its respective best-case complexity $n \times log(n)$.

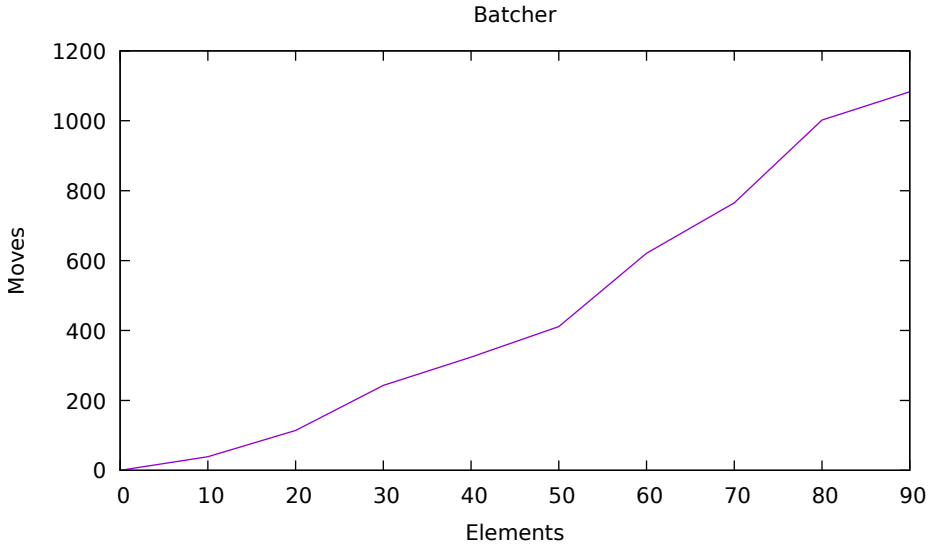


In this case, the convergence is apparent but more difficult to pinpoint. It seems to somewhat oscillate around 100, so that will be Quick Sort's constant.

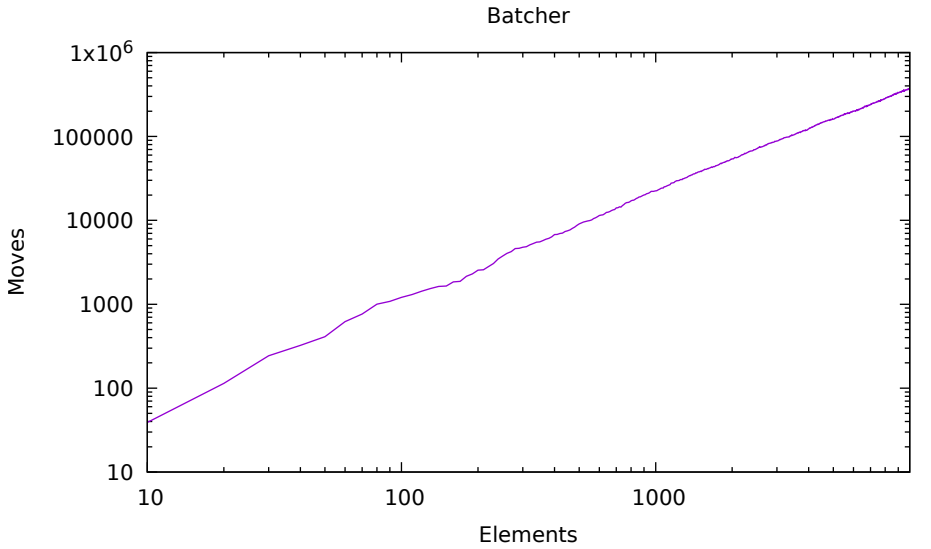
Overall, Quick Sort seems like it might sacrifice move counts (and thus resources) for time. It **is** the quickest of the bunch (as seen later) but when it comes to reverse-ordered lists, it struggles in the operations category.

5 Batcher Odd-Even Merge Sort

Batcher's Sort is unique from the rest because it is a sorting *network*.

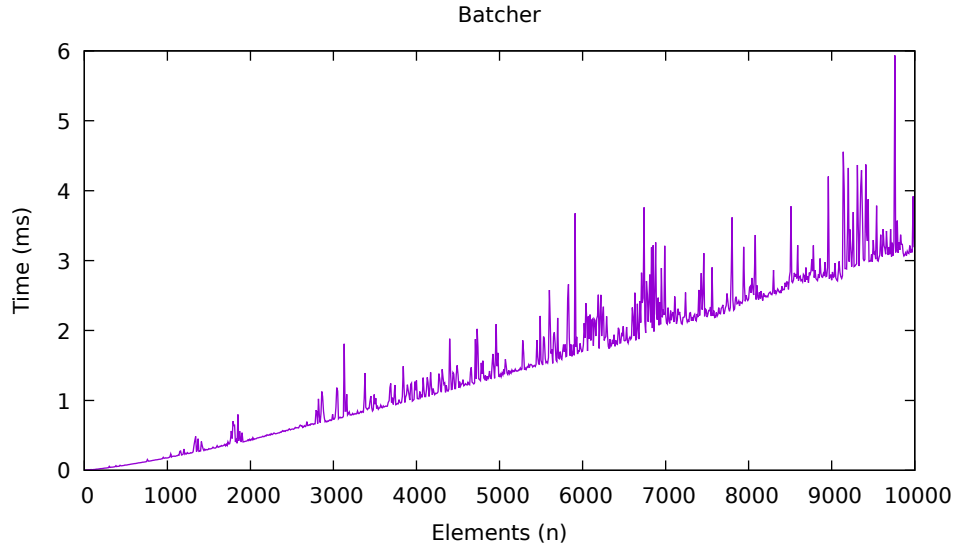


On a small scale, Batcher uses an average (in relation to others) number of moves. The line isn't as smooth as Heap or Shell but seems to be more stable than Quick Sort. However, Batcher does worse in terms of moves compared to sort.

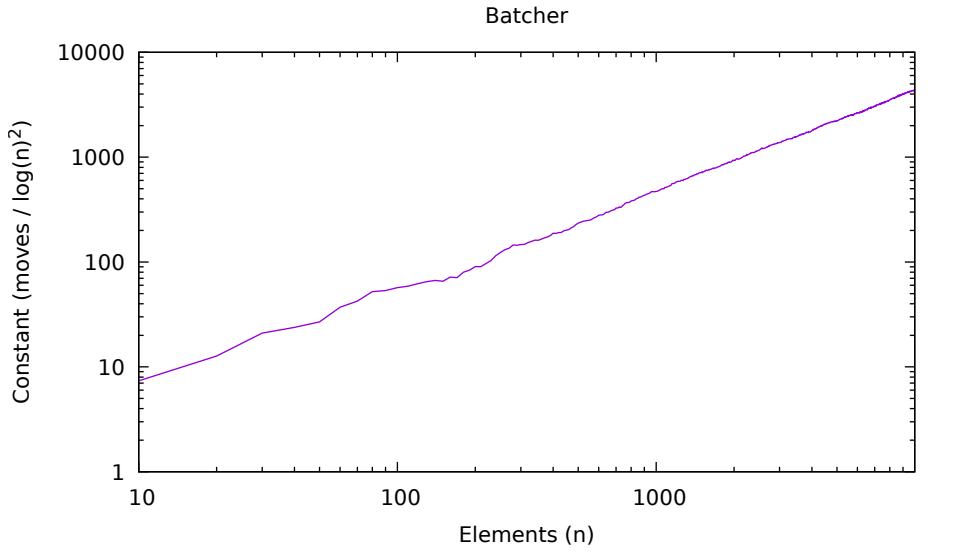


Increasing elements by a lot, we observe that Batcher becomes even more stable and its curve seems to follow (loosely) a $n \times log(n)$ line. In terms of moves, it does as well as Quick and Heap.

Note: *When running tests for reverse-ordered arrays, Batcher kept throwing errors about memory and I couldn't figure them out in time. Sorry.*



The above graph depicts the time taken to complete the sort on a given a number of elements.

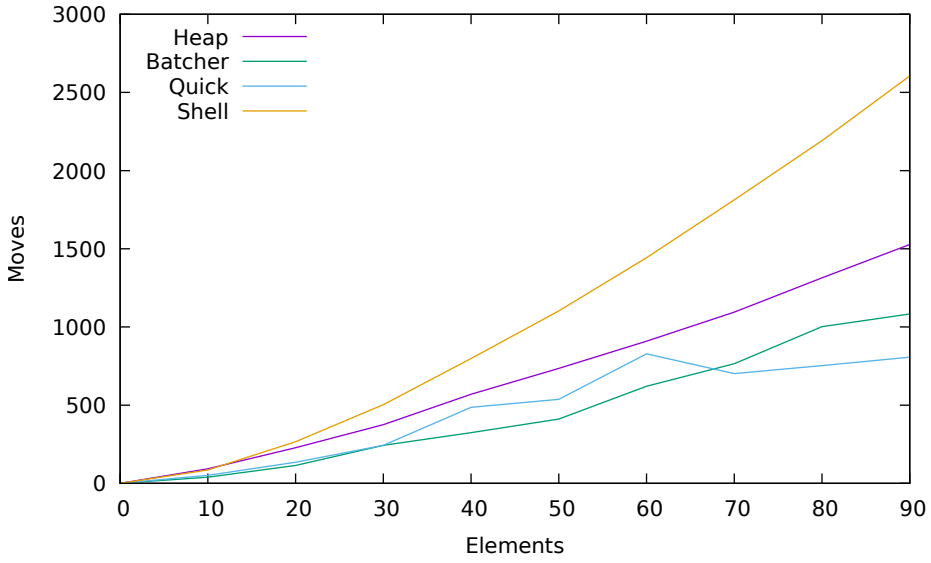


Lastly, we plot the approximation of Batcher's constant however, it doesn't seem to converge onto any value. I used the complexity found at Wikipedia to estimate, $log(n)^2$. I was not able to resolve why this relationship doesn't converge.

6 Wrap-Up

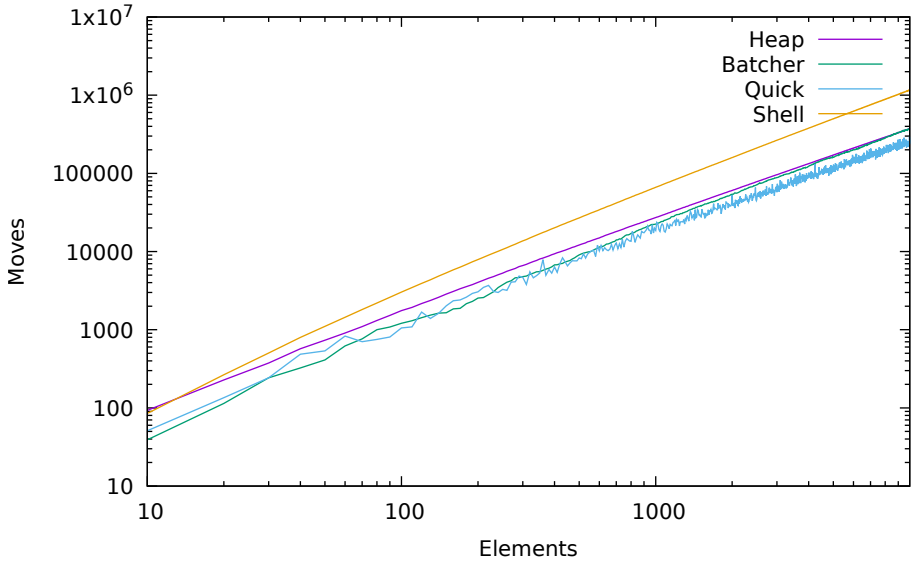
- **Comparing the Sorting Algorithms**

  The below graph shows each sorting algorithm on a small scale.

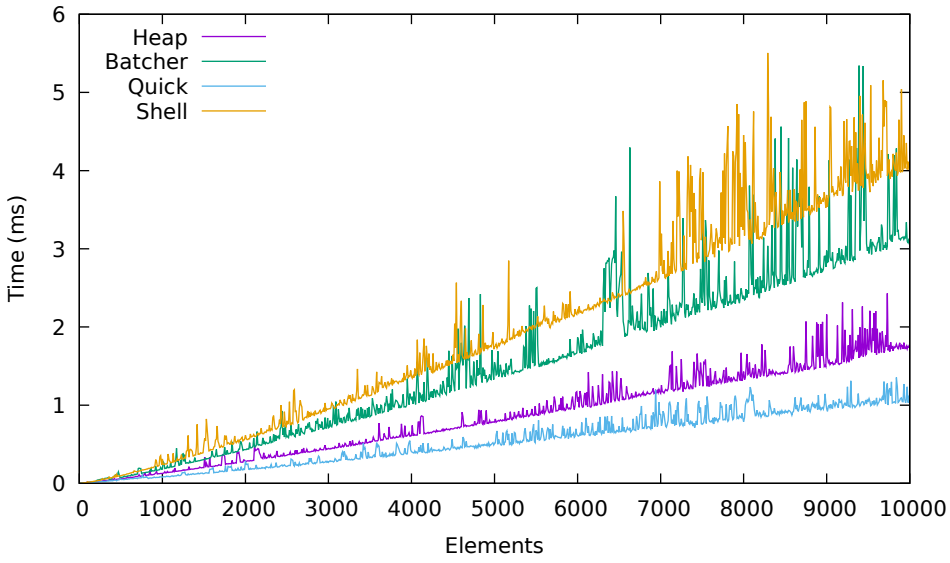


Shell and Heap, as mentioned before, appear smooth and consistent. This is juxtaposed with Batcher and Heap which look a little more volatile. But, both Shell and Heap perform *worse* than the latter two, completing with more moves per element.

Next, we turn our attention to a greater sample size of elements.



The same is seen on a larger scale. Heap actually tends to become more effective than Batcher around 10,000 elements. This is while Shell, while consistent, diverges from the rest becoming the least efficient, operation-wise.



Lastly, we look at a comparison of the time taken for each algorithm. Again, Shell comes in last taking the longest, especially in the latter half. However, Batcher is a close second in this regard. In some cases, it's apparent Batcher performs *worse* than Shell for a given number of elements.

Heap and Quick sort both keep more consistent ranges and are far lower than the previous two. Quick, as stated before, is the quickest algorithm, not reaching past 2 milliseconds at any point.

- **Take-Away**

  I come away from this assignment understanding some of the more common sorting algorithms as well as interpreting Python code with C code.