ASGN3: Sorting: Putting your affairs in order

The final results for all of the sorts for moves/compares for 100 elements was:

Quick:1053, 640
Heap:1755, 1029
Insertion:2741, 2638
Shell:1183, 801

As seen, the ranking of sorts from fastest to slowest for both moves and compares would be Quick, Shell, Heap, and lastly Insertion.

We can confirm these rankings for higher numbers of elements if we increase the number of elements from 100 to 500. At 500 they are:

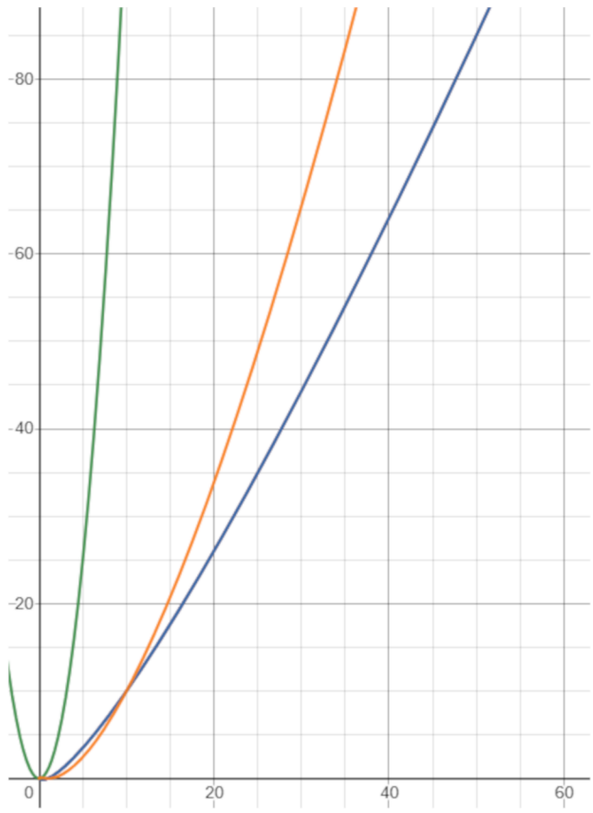
Quick: 8280, 4717
Heap: 12132, 7422
Insertion: 64858, 64354
Shell: 8318, 5640

Thus the ranking from fastest to slowest remains the same with Quick being the fastest and Insertion being the slowest.

The reasoning as to why becomes evident once we examine the average time complexity of each of the sorts. Given that for both 100 and 500 elements, the elements were completely random we can consider both instances to be average cases. The time complexity for each average case is:

Quick: Θ(n*logn)
Heap: Θ(n*logn)
Insertion: Θ(n^2)
Shell: Θ(n*(log(n))^2)

In graph form this can be seen as:



Where Green=Insertion
Orange=Shell
Blue=Quick/Heap

As seen in graph form, Quick and Heap should be the sorts with the least steps but somehow Quick and Shell are the sorts with the fewest steps in our program.

This desparity can partially be traced back to the fact that not every move and compare is actually kept track of in Long's provided binary file. As seen in my quick.c file, not every if statement has a compare nor does every variable assignment have a move. Although Those extra additions may not seem significant, they can alter the range for which one sort is better than another. For instance, looking at the graph of average time complexity we can see that for a very brief period of time Shell Sort will actually yield fewer steps on average than Quick and Heap sort. The missing extra additions from the sorting algorithms could alter the range for which rankings of sorts change.

Another reason for this disparity is rather simple, random chance. Due to the simple fact that all of these sorts operate differently, all of these sorts have varying ideal conditions for the arrays they sort. Thus a random array can simultaneously be more ideal for one sort and less ideal for another, resulting in disparities such as these.