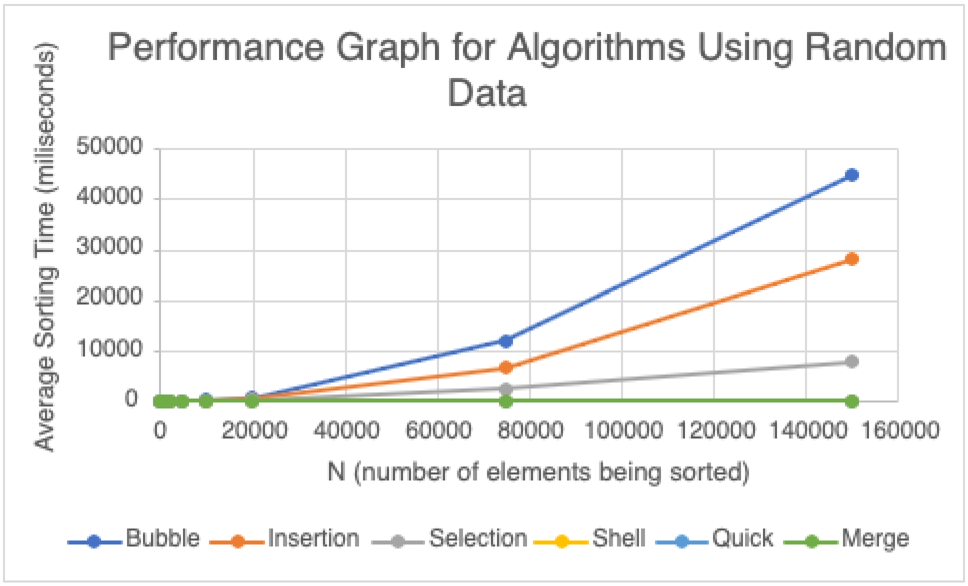
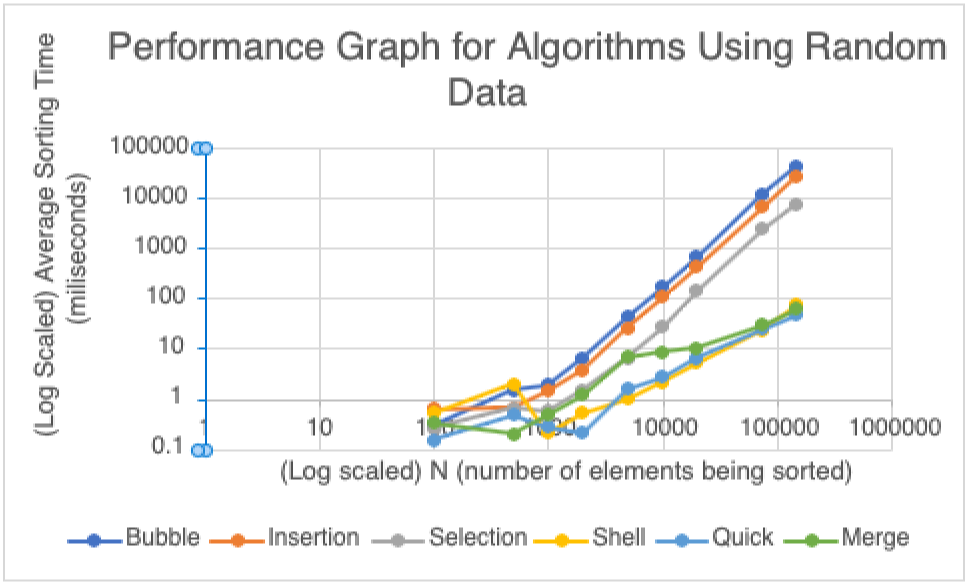
Problem 9:

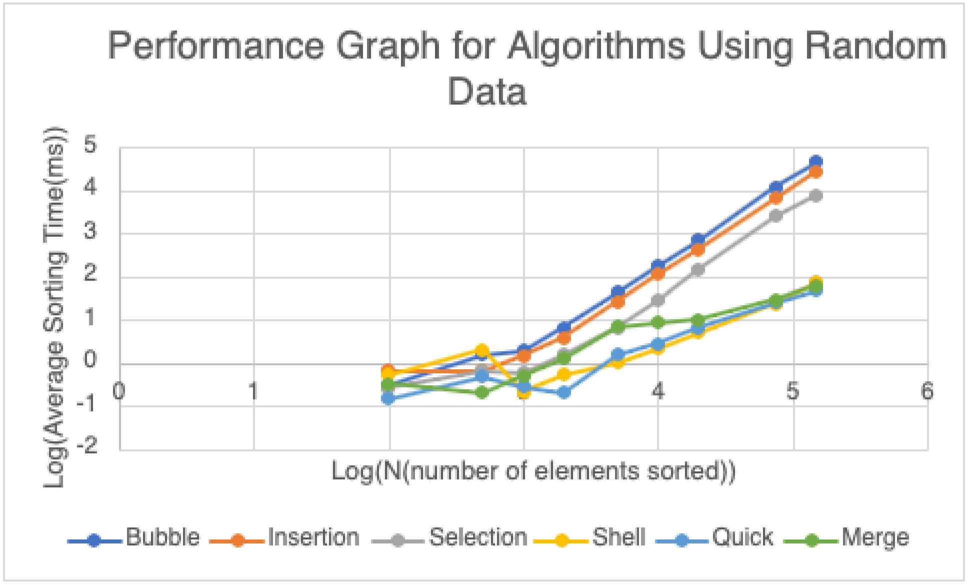


This is the first graph I made using my data. As seen in the legend, there are all six sorting algorithms on here. There is a big issue, which is that the data sizes for bubble, insertion, and selection sort are so large that it is "pushing" the graph higher, and making merge sort seem linear, and by consequence hiding the results of shell and quick sort. Basically, this means that the sorting time of bubble, insertion, and selection sort are so large compared to those of merge, shell, and quick sort that their times and results are very difficult to see and appear linear, or don't appear at all. The large difference skews the y-axis to make it more top heavy, which makes it harder to see the smaller results since the y scale leaning is so much higher, and the y gaps are very large.

This is my attempt to fix that issue. By taking the log of the y axis and the x axis, this helps to 'straighten' the curve and reduce the growth scale visually. By simple using the excel feature to log scale both axes, this graph is created, with the original values kept, but the lines are adjusted to fit if it were log of that value:



This now lets us view all the algorithms and allows us to compare them. Its original values for y and x are kept for each algorithm, so there is no need to convert the values from log to get their original form, since the axes are log formatted. The only downside is that the y axis has a very weird scale where it jumps from 1 to 10 to 100 and so on, which makes it hard to get exact numbers from looking. To confirm this is the right shape, and the lines are in proper order, I made this graph where I used the log of x and log of y to make the graph:

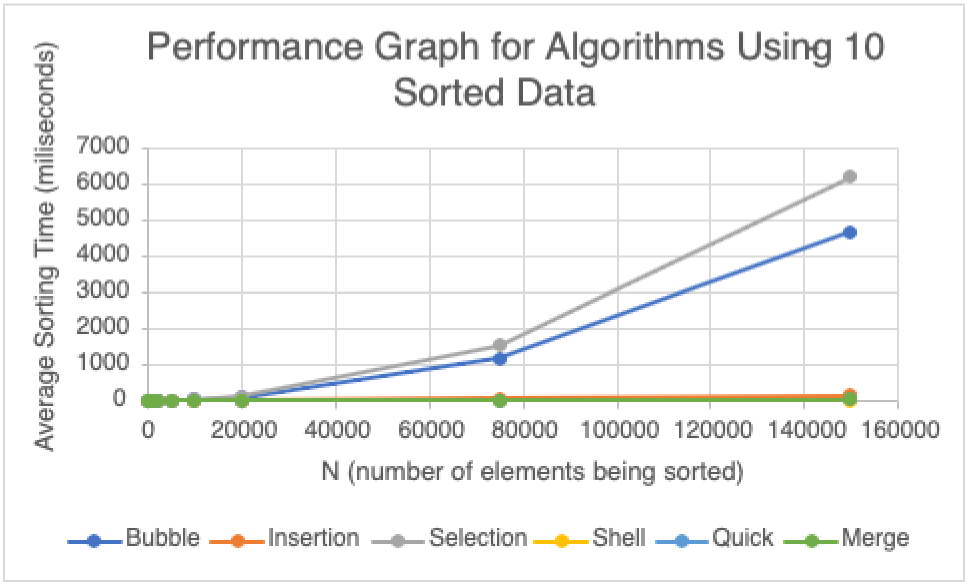


So as seen above, both have the same shape and everything is correct. The advantage of this graph is that the y axis does not have any weird jumps and is very consistent. The downside is that a conversion from log is needed to get the original values.

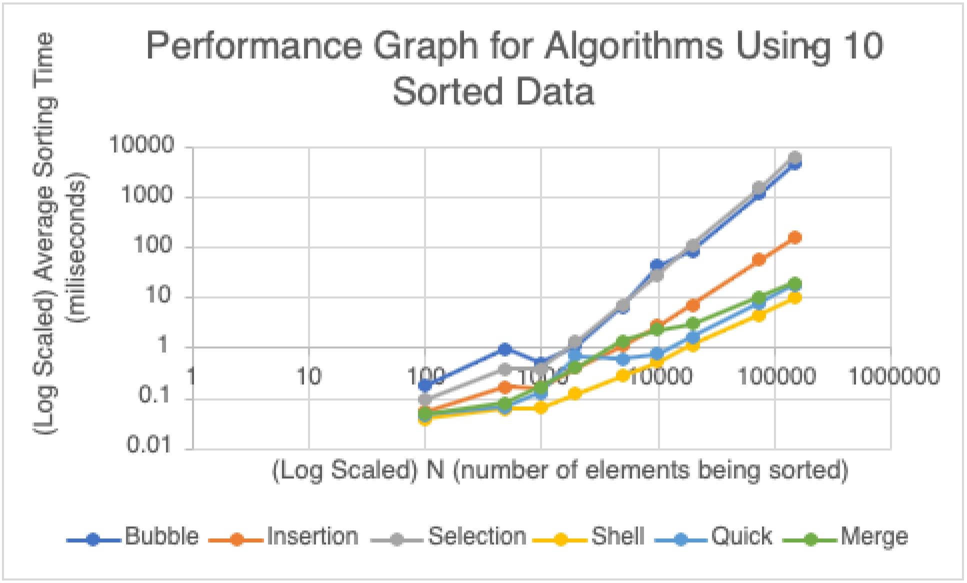
Problem 10:
In my experiments I observed that bubble, insertion, and selection sort all took the longest compared to shell, quick, and merge sort. Those three took tens of minutes to finish sorting the larger sizes, while shell, quick, and merge sort finished sorting it very quickly, usually in the amount of time it took the previous three to sort 1000 or 2000 elements. I also noticed that most of the time all six sorting algorithms had a very similar average sorting time at 100 and 500 elements ranging from 0.5 to 1 millisecond. After that though, the difference between them could start to be seen. At 5000 elements bubble and insertion sort took tens of milliseconds, selection took only a few milliseconds, while the rest did not take much longer than two milliseconds. Then as the element size grew, do did the difference in time between many of them. Bubble and insertion sort took tens of thousands of milliseconds; selection took several thousand, but the rest never took longer than eighty milliseconds to sort 150000 elements. So, selection sort was the middle ground between all the algorithms, since it was faster than bubble and insertion, while being slower than quick and merge sort. Bubble and insertion sort were the slowest out of all of them. Based on the experiments, the performance of quick sort was always the fastest, then merge sort, then shell sort, then selection sort, then insertion sort, and finally bubble sort. The actual performance almost never matched my prediction in question 5. The closest result to my prediction was at the beginning when all of them were about the same, after that though, it was very clear which ones were faster than the others. The reason for this difference is because of the number of swaps and not being the worst case. The five-way tie between shell, quick, bubble, insertion, and selection were wrong for many reasons. The reason it was wrong for shell and quick sort was because they did not take the worst case and performed a lot faster since they seemed to perform closer to their best case, which was O(nlog2n) and O(nlogn), then they were their worst case. Like merge sort, both do separate the list in their own way to sort the list and make it faster to sort or closer to proper order, which made it faster than going through each element one at a time like selection, bubble, and insertion. Compared to shell sort, quick sort was fast because of its divide and conquer approach allowing it to be more efficient than shell sort since it can 'simultaneously' sort two halves. The reason why quick sort performed the best even against merge sort even though merge sort is always O(nlogn) is because merge sort creates many arrays and must fill them up, which makes it take longer. Bubble, insertion, and selection sort, did not fully match my tie ranking, even though all of them almost go through the loop one by one because of the swaps. Bubble and insertion being tied were very close to right, since they consistently were in a similar range, with bubble being

considerably higher. The difference between bubble and insertion regarding time was probably because of the break I included in insertion which would make leaving the loop each time much faster. However, selection sort being faster than them despite always being O(n^2) is because it made less swaps than them on average. Selection sort only does one swap each iteration of the outer loop whereas the others could do multiple swaps in one iteration of the outer loop. This difference is because bubble swaps until the highest value is at the top, insertion compares and swaps until it finds the right placement, but selection just find the smallest and swaps it with the current starting index. This is why the selection was much faster than both of the previous ones. This is why the rankings did not match the results I gathered. From the experiments, this would be my ranking now: Quick, Merge, Shell, Selection, Insertion, and Bubble.

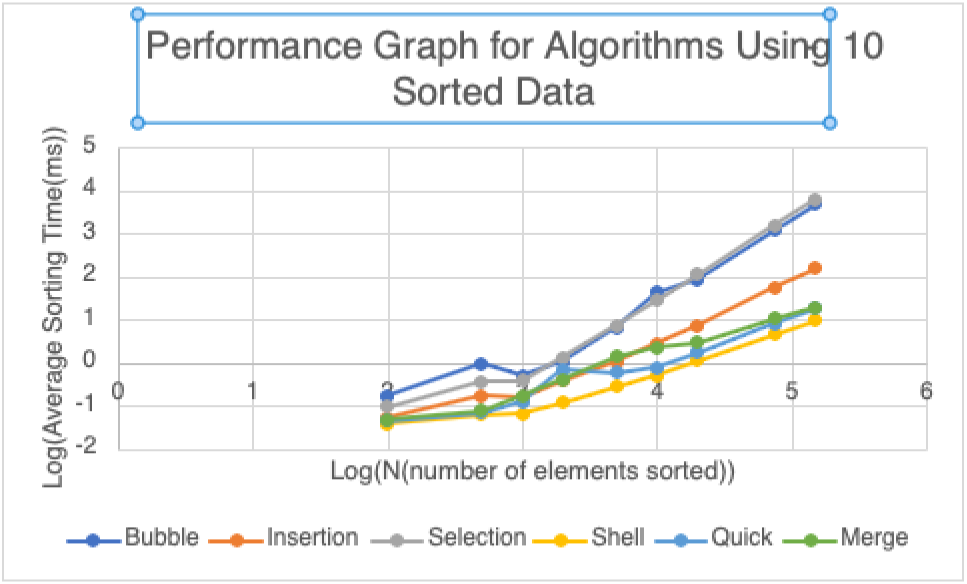
Problem 12:



This is the initial graph of the runtime of the six algorithms using 10-sorted data. There is an issue in this which is that Selection and Bubble sort have a runtime that is much higher than the run time of everything else, so it makes the scale of the graph unbalanced on the y axis. This can be seen because they are in the thousands, while the other sorting algorithms are nowhere close to the thousands on the y axis. So, this unbalance is making it difficult to see or even distinguish the smaller values from the larger ones and themselves. Insertion is slightly above merge, but quick and shell cannot be seen even though they are there. The solution to this problem is the same as it is for problem 9, and it is to log scale or take the log of both axes so straighten the graph since it reduces the growth scaling, to be more visible. This is the result of log scaling:



So, the results appear more linear, their original values are kept the same they just take the form of their logged axes, and you can see all algorithms runtimes. This makes it easy to tell them apart; it's just that the scaling of the axes on the y side is not the best and has large gaps making it a bit difficult to get exact data. There is also the graph where log is taken of both sides which gives:



Both have the same shape, but the advantage to this is that the y axis is scaled well and does not have a weird jump in values. The disadvantage to this is that to get the original value you have to undo the log.

In my experiments, I observed that the overall runtime of this for all algorithms was much faster than it was for random data. First off, none of them were at ten thousand milliseconds for 150000 elements, with the quickest at that time being around 10 milliseconds. All six sorting algorithms had a very similar runtime up until about 2000 to 5000 elements, but even so they were still within 10 milliseconds of each other most of the time, which is much better than before. Even so, at the largest size, the largest run time never reaches ten thousand like it did previously. Shell, quick, and merge sort were the fastest like before, but this time the amount of time it took for them to sort 150000, was the amount of time it took the other algorithms to sort 10000 to 20000 elements, except for bubble sort, which was about the same with 5000. This is significantly better than before since it would only take half or even a fourth of the number of elements for their total completion time to be faster than sorting x elements with the other methods. Overall, every algorithm performed significantly faster than before with 10-sorted random data. Bubble, insertion, and shell sort performed significantly differently. This time quick and merge sort were some of the fastest, neither the fastest, but still in the expected range. Since all of them performed better than before their time decrease of about 30 milliseconds is big but not the most significant, since we know that merge always run at O(nlogn) we know that its current time is about O(nlogn) for 10-sorted data. Quick sorts run time is very similar to that so we know that it is also about O(nlogn), which is its best case. So, both are not too significant or surprising. Selection sort took less time, which all of them did but it's not very significant, since it still took several thousand milliseconds to sort 150000, which is close to its time for random data . Bubble, insertion, and shell sort took much less time than with random data. Bubble took about a tenth of the time it needed for random data to sort 150000 elements, and the reason for this is because 10-sorted ensured everything was roughly within 10 positions of its correct place, so it needs significantly less swaps to get the highest and next highest values to the top, which is why it took less time. Insertion sort was much better than before now only needing a couple hundred milliseconds to finish sorting 150000 elements, this is again because it got to do less swaps. Since 10-sorting ensures everything is about 10 positions away from its correct position, insertion sort would only have to swap a number 10 positions down maximum, while before it could have to swap it down hundreds of times. This cap makes it significantly faster to sort through everything and put it in the right position, and makes it do less comparisons. Finally, shell sort is the fastest this time and took close to 10 milliseconds, which is much faster than before since it is now faster than quick and merge sort. The reason for this is because 10 sorting the data already makes things in 10 positions of where they should be, and since shell sort performs insertion sort on its interleaves, making it faster than pure insertion sort, this speeds it up since it also has to do less swaps. Furthermore, giving it 10-sorted data makes sorting its interleaves faster since everything is within ten positions of the right place the larger interleaves get passed much faster since everything is nearly sorted. So, everything is very near to being sorted at the end. Overall, everything performed much faster than random data but did not have the same ranking as random data. My original ranking was merge first and the rest tied for last, and that is still wrong here since shell is first, then merge and quick are about tied, then insertion, then bubble, then selection. This is also different from my revised list from random data. My revised list was quick, merge, shell, selection, insertion, then bubble sort. Many of them are swapped around, shell moves quick and merge down the ranking. Then insertion and bubble move up the ranking as selection sort is dropped to last. This is also the ranking that is backed by my data. So the ranking for k-sorted is different from the ranking of randodm data and my prediction.