Assignment 9a Writeup

Hypothesis:

We believe that the run time of the insertion sort algorithm will be faster than the quick sort algorithm when sorting a list that is already in ascending order, as the performance of insertion sort on an already sorted list is O(n). This will apply to all of the sample sizes of lists in ascending order. However, the runtime of quick sort will be superior on descending or randomly sorted lists, as its average performance of O(n log(n)) is better than that of insertion sort, which will average O(n^2).

Testing:

For our tests, we chose a number of arrays of various sizes ranging from 1 - 131072(originally, we meant to test up to 500000, but this was not possible due to hardware limitations) . Generally, our arrays doubled in size(1, 2,4,8, etc). The purpose of having so many samples was to ensure that the difference in the performances of quicksort and insertion sort would not change if the sizes of the arrays varied significantly. Additionally, we needed a number of samples to ensure that when we tested randomly filled arrays, the result wouldn't be skewed by 1 or 2 fills that happened to favor one sort over the other.

For our testing procedure, we made a number of arrays (starting with a size of 1, and gradually increasing in size). We then filled the arrays with integers organized in ascending, descending, and random order. For example, an array of size 8 would have the contents {1,2,3,4,5,6,7,8} when in ascending order, {8,7,6,5,4,3,2,1} in descending order, and an array of randomly chosen integers in random order. We would then test the amount of time it would take to complete a quick sort on the array using the std::chrono library. Then we would reset the array to its previous state. Afterwards, we would then test the amount of time it would take to complete an insertion sort on the same array using the std::chrono library. After testing the amount of time it would take to complete these 2 sorts on all of the different types of fills(ascending, descending, random), we would move on to an array of greater size. After we gathered all of our data, we would look at the amount of time taken by each sort to determine the results. A faster runtime would yield a superior performance.

Results:

Ascending Order

Size	**Quick Sort (Execution Time in Milliseconds)**	**Insertion Sort (Execution Time in Milliseconds)**
1	0.00016600	0.00004200
2	0.00012500	0.00012500
4	0.00033400	0.00016600

8	0.00062500	0.00029200
16	0.00166600	0.00033300
32	0.00570800	0.00054200
64	0.01995800	0.00083300
128	0.09520800	0.00145800
256	0.49212500	0.00275000
512	0.57862500	0.00225000
1024	2.29716700	0.00437500
2048	10.35654100	0.01054200
4096	32.57641600	0.01279100
8192	101.12779200	0.02487500
16384	403.64175000	0.04791700
32768	1620.27779200	0.09504200
65536	6517.90470800	0.19133300
131072	26100.70720800	0.40720800

Descending Order

Size	**Quick Sort (Execution Time in Milliseconds)**	**Insertion Sort (Execution Time in Milliseconds)**
1	0.00004200	0.00004100
2	0.00012500	0.00012500
4	0.00041600	0.00025000
8	0.00070800	0.00050000
16	0.00154200	0.00104200
32	0.00475000	0.00362500

64	0.01508300	0.01279200
128	0.05425000	0.04200000
256	0.26241700	0.06666700
512	0.41641600	0.25620800
1024	1.63520800	1.5168750
2048	7.81066600	4.56141700
4096	19.25679200	11.30112500
8192	72.25287500	44.07820800
16384	286.49658400	177.97183300
32768	1148.24725000	712.08454200
65536	4532.15787500	4532.15787500
131072	18195.55712500	11449.34112500

Random Order

Size	**Quick Sort (Execution Time in Milliseconds)**	**Insertion Sort (Execution Time in Milliseconds)**
1	0.00004100	0.00004200
2	0.00012500	0.00008300
4	0.00029200	0.00020900
8	0.00050000	0.00033300
16	0.00116600	0.00091700
32	0.00258300	0.00225000
64	0.00604200	0.00695800
128	0.01416700	0.02500000

256	0.01787500	0.03445800
512	0.03937500	0.13604200
1024	0.15320800	0.83958300
2048	0.23895800	2.31029200
4096	0.30362500	5.52608300
8192	0.62087500	22.24508300
16384	1.32966700	88.58025000
32768	2.81020900	357.78241700
65536	5.90837500	1415.37691600
131072	12.99645800	5812.63191600

Conclusion:

Our hypothesis was partially correct. For sorting in ascending order, insertion sort was the more efficient algorithm, as shown by its smaller runtime. In a random order, quicksort was generally faster, especially as arrays grew to a size such that all of the elements being in ascending order would be unlikely. However, insertion sort was generally faster for lists in descending order(rather than quicksort). Evidently we were mistaken in assuming that quicksort would be faster for a list in descending order.