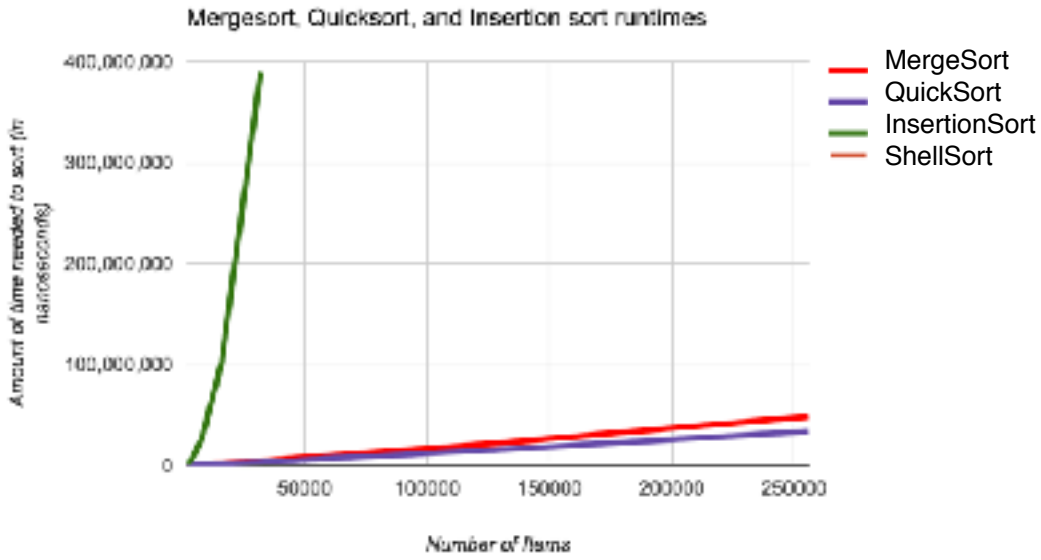


Figure 1: Run time for random arrays

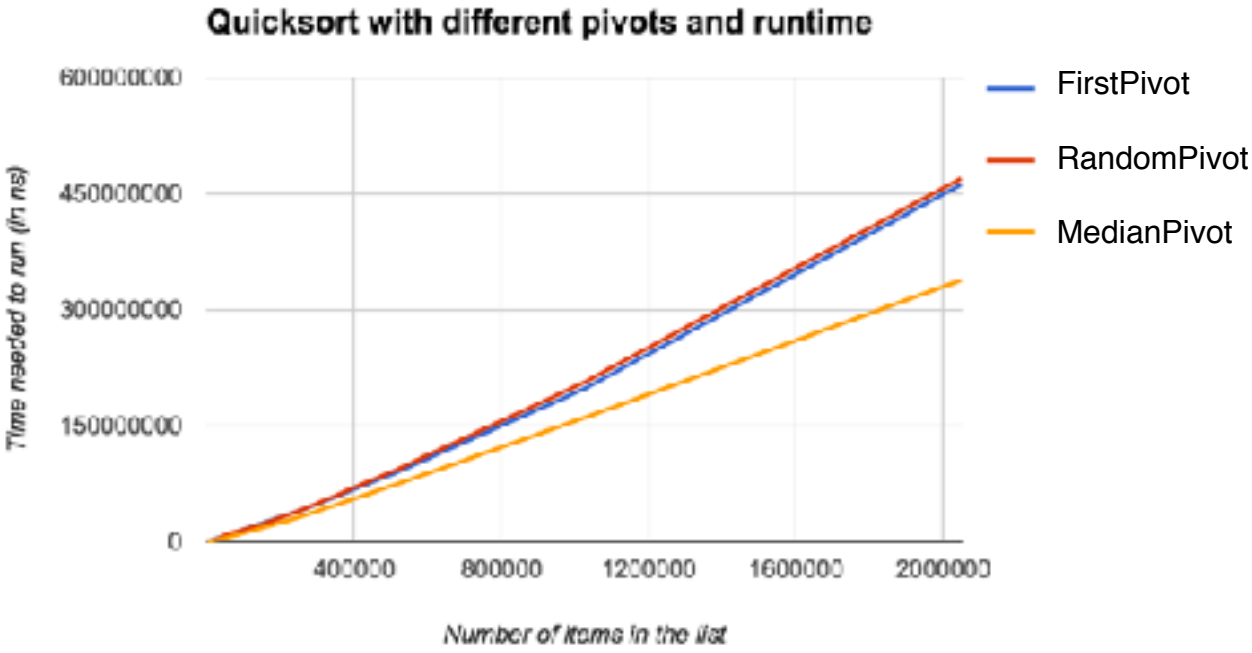
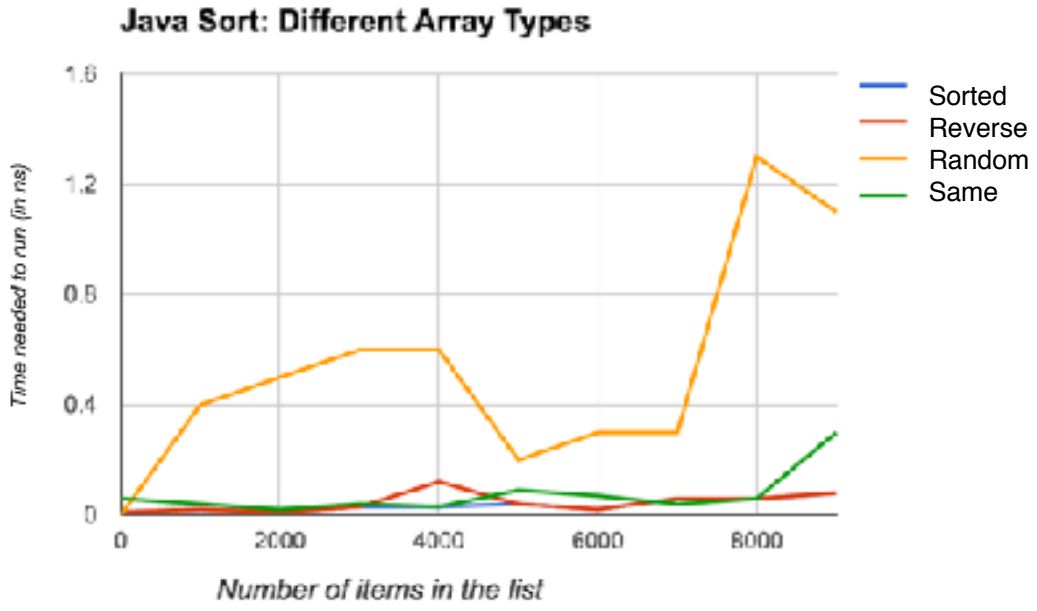
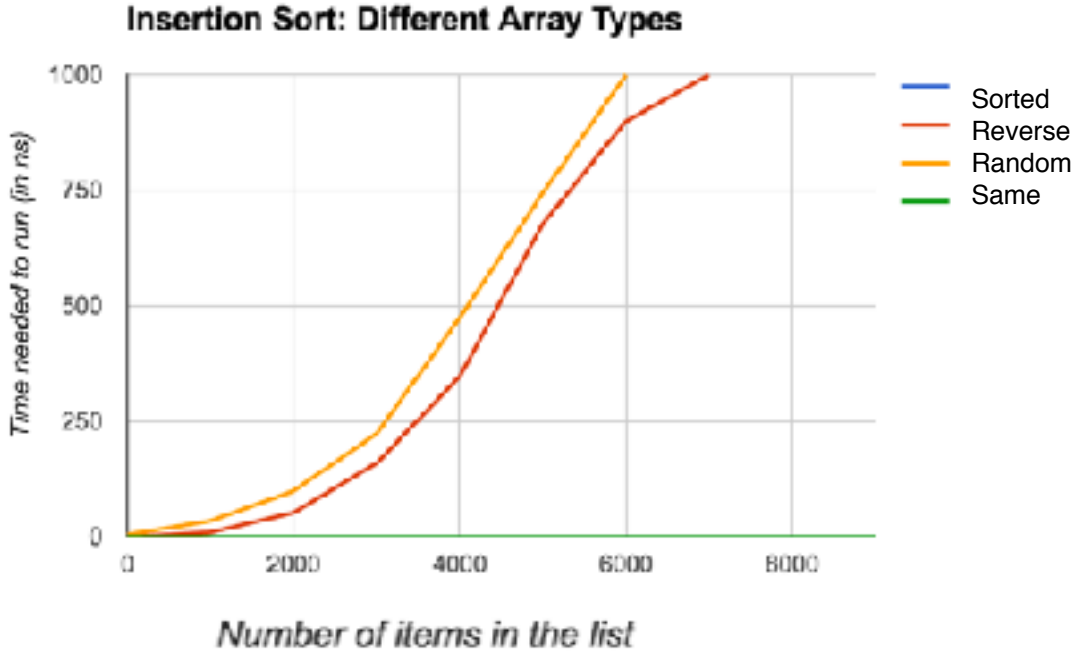


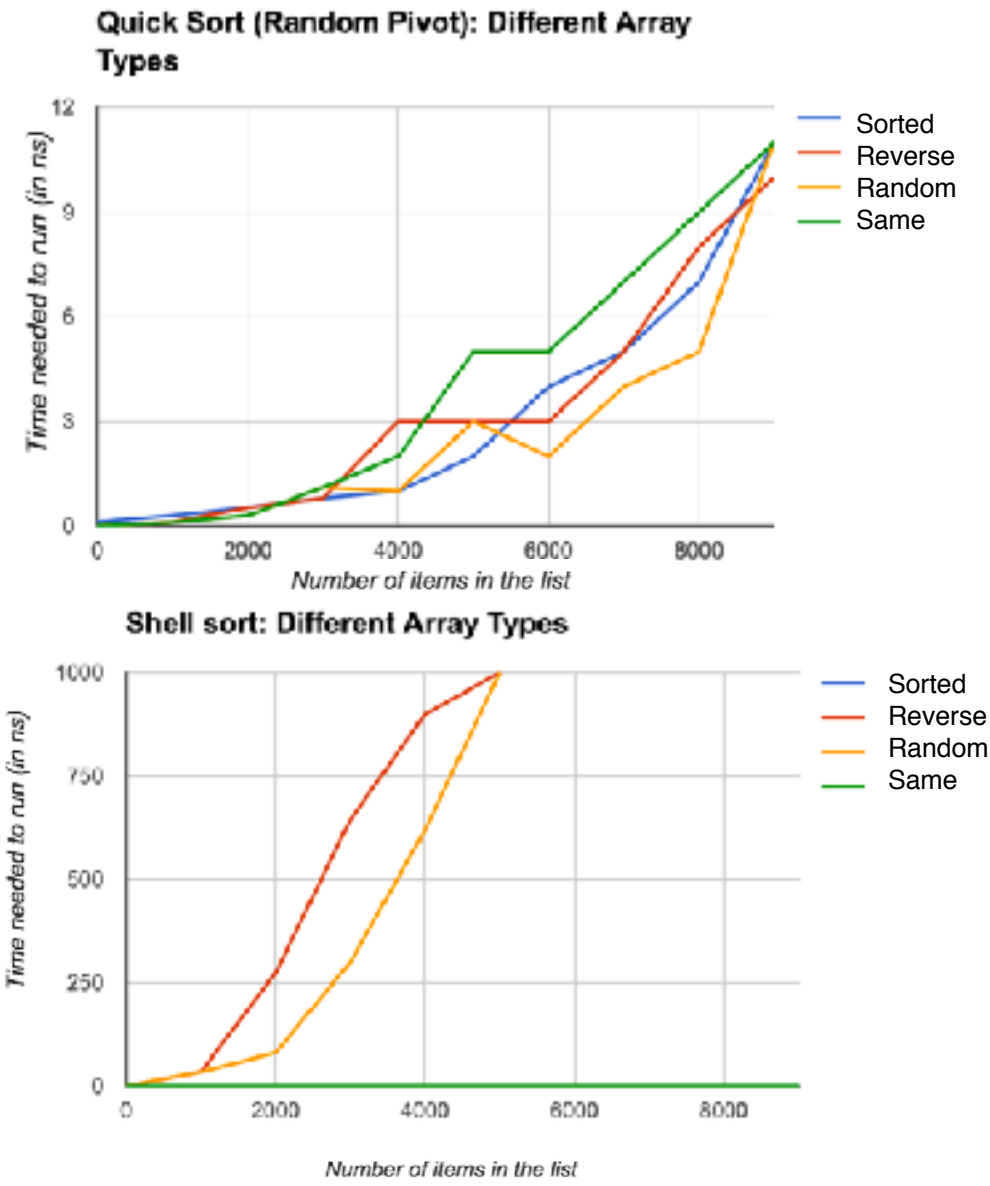
Figure 2: Run time for quick sort with different pivot points

1.) For your best implementation of each sort, plot a line for each type of test data (in-order, reverse-order, random, same).

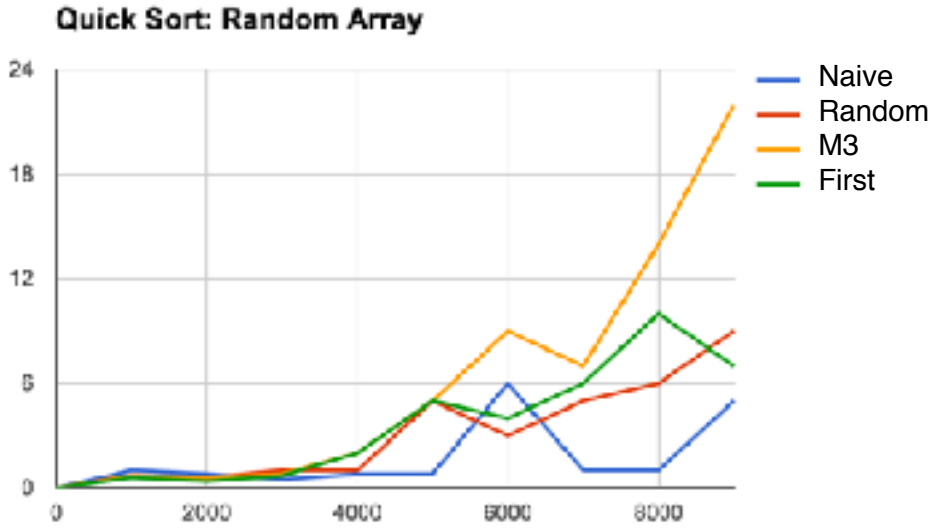

Merge Sort: Different Array Types
Time needed to run (in ns)
1000
750
500
250
0
0
2000
4000
6000
8000
Number of items in the list
Sorted
Reverse
Random
Same


Java Sort: Different Array Types
Time needed to run (in ns)
1.6
1.2
0.8
0.4
0
0
2000
4000
6000
8000
Number of items in the list
Sorted
Reverse
Random
Same

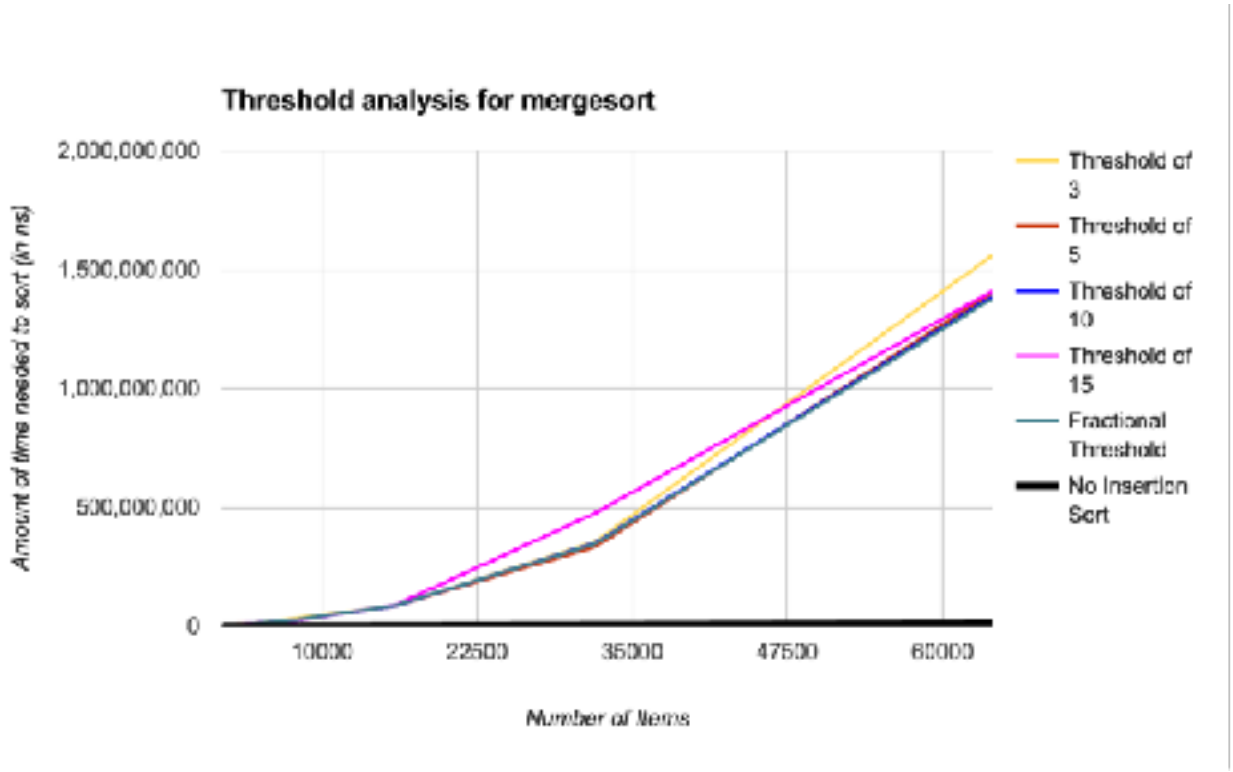

Insertion Sort: Different Array Types
Time needed to run (in ns)
1000
750
500
250
0
0
2000
4000
6000
8000
Number of items in the list
Sorted
Reverse
Random
Same



2.) For the other implementations (e.g., the naive) plot a line showing the cost for sorting random data (only).



Create a third graph for merge sort and quick sort where the only change is the insertion sort cut off value and how that affects the run time. You a reasonable amount for N when testing this.



3.) Answer this question: Quick sort works best when the partition is the middle element of the array which is (for normal data) very close to the average value of all elements in the array. Thus: A better pivot choice would seem to be to use the average value of the array. There are two flaws to this. Identify and discuss them.

One flaw with picking the average value every time is that you would get a very similar partition every time since the average value would do very well in the first partition but after that the partition would be 'diluted' since its the same value every time. Another reason the middle pivot method would be better than the average is that if you test it multiple times it will pick the median, on average, for every recursive call making it the optimal choice.

4.) Answer this question: Why does the Fisher-Yates shuffle use the following line:

If you shuffle the elements from 0 to i then you can increment i so you only have to shuffle what you have left instead of the whole array ever time which is what would happen if you went from 0 to N.

5.) Finally, discuss what you have learned about a) implementation tweaks to improve performance, and b) algorithmic choices to improve performance.

Implementation tweaks to improve performance can affect your algorithm in minor way, such as a gap size for shell sort or the switch value for the merge/quick sort. As the graph directly above shows the performance can improve but not by that much. However, different algorithm choices has major affects on performance. As figure 1 shows if you choose to implement insertion sort for a program that has to sort 50,000 elements, it will be infinitely slower than say if you chose to implement merge sort for the same program.