Gwendolyn Wahl
CS 5120
Project 1
10/08/2015

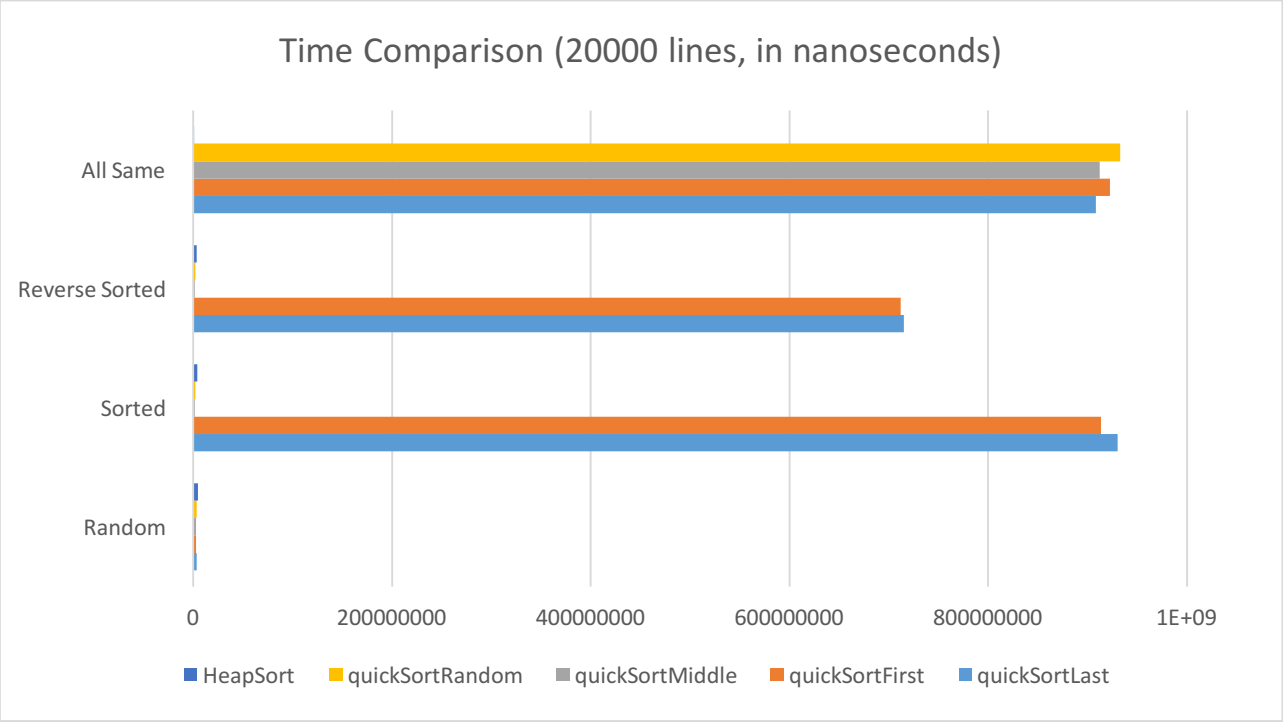
In this report I will be discussing the results of the five algorithms included within project 1. The five algorithms in question are QuickSort with the pivot as the last element, QuickSort with a first element pivot, QuickSort with the pivot set in the middle (rounded down), QuickSort with a randomized pivot, and lastly HeapSort. QuickSort in general has an average case complexity of O(nlogn), as does HeapSort. QuickSort however is known to have a worst case complexity of $O(n^2)$. It is also fairly well known that QuickSort, due to it's implementation, tends to outperform other O(nlogn) sorting algorithms in head to head trials.

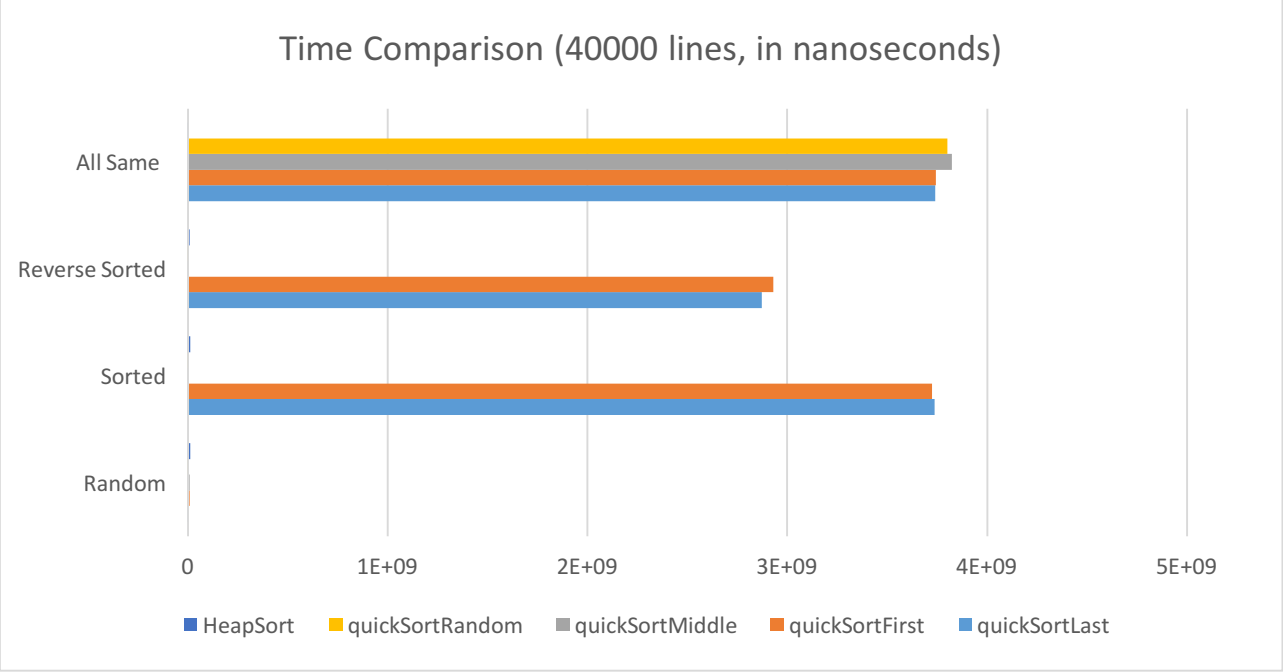
I implemented the algorithms in C++ using the clang++ compiler on OSX. I attempted to use as efficient an implementation as possible, working with primitive arrays and integers. I also utilized tail recursion to avoid any issues with stack overflow. Each algorithm can be individually called from the program, and there is a bash script that will compile and run all five on a given dataset. Collecting the data in all took about a half hour.

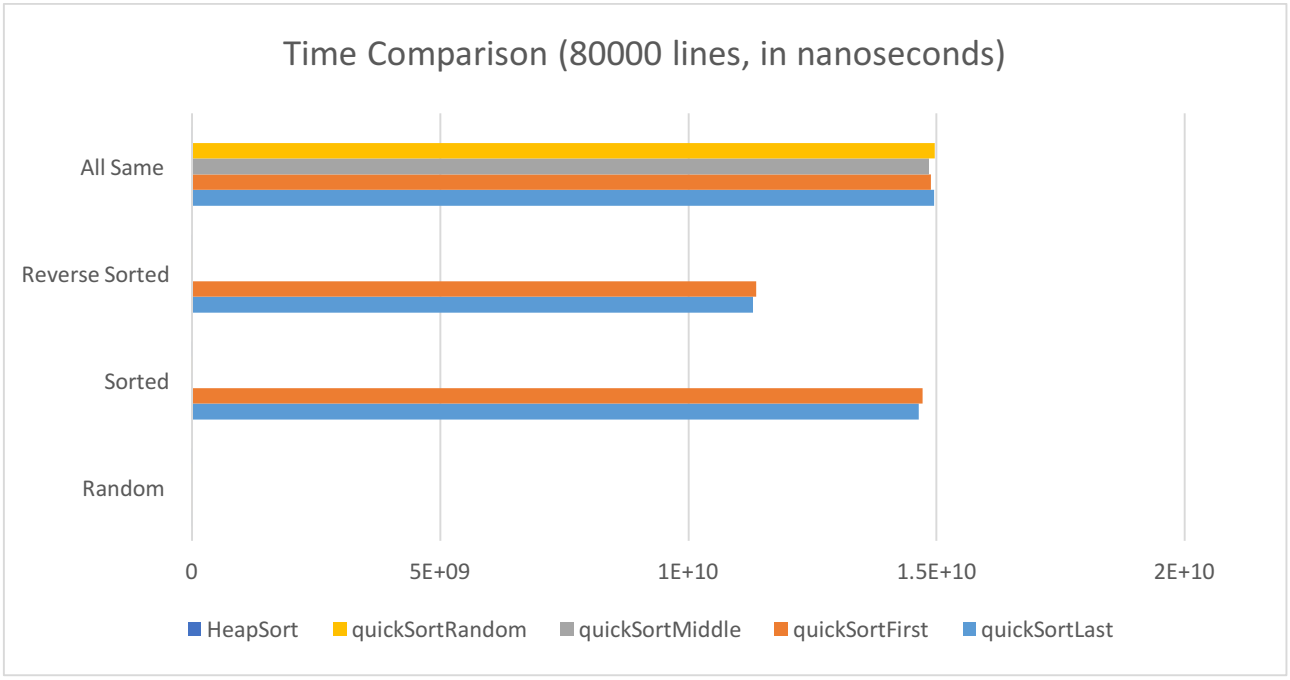
After running each algorithm against a total of 16 data sets, here are the observed results.

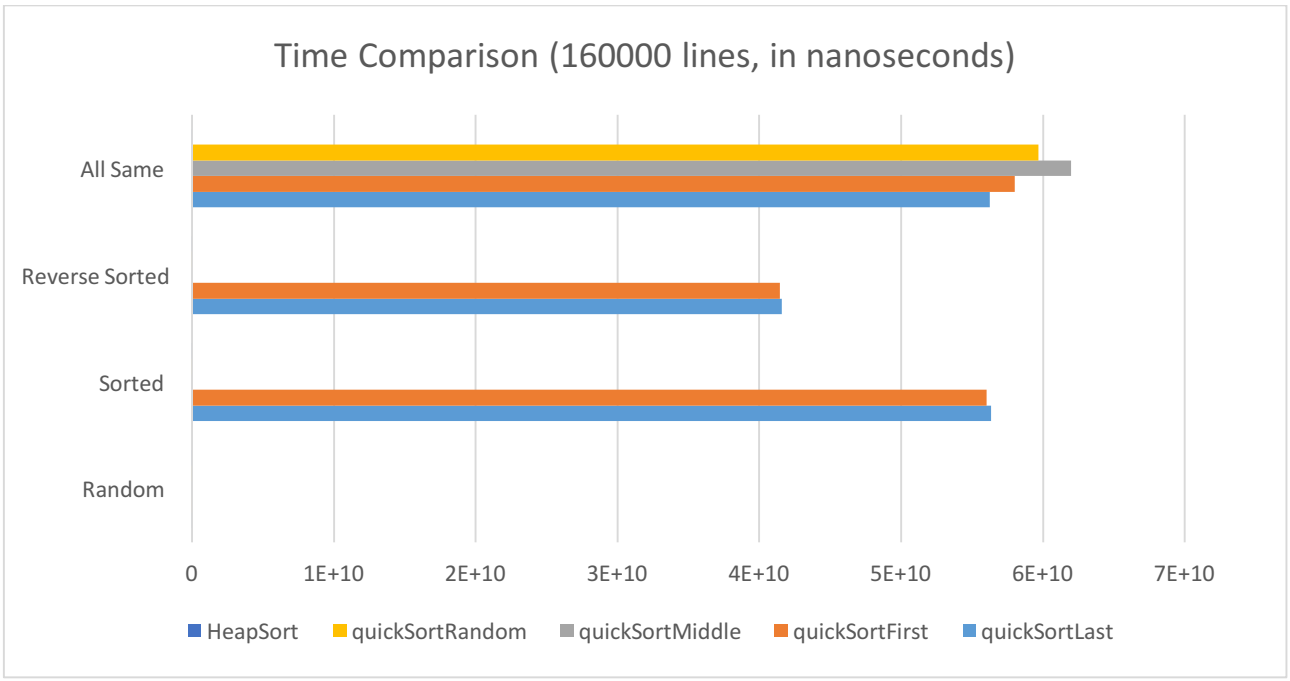
	quickSortLast	**quickSortFirst**	**quickSortMiddle**	**quickSortRandom**	**HeapSort**
Random Data	O(nlogn)	O(nlogn)	O(nlogn)	O(nlogn)	O(nlogn)
Sorted Data	$O(n^2)$	$O(n^2)$	O(nlogn)	O(nlogn)	O(nlogn)
Reverse Sorted Data	$O(n^2)$	$O(n^2)$	O(nlogn)	O(nlogn)	O(nlogn)
All Same Data	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(nlogn)

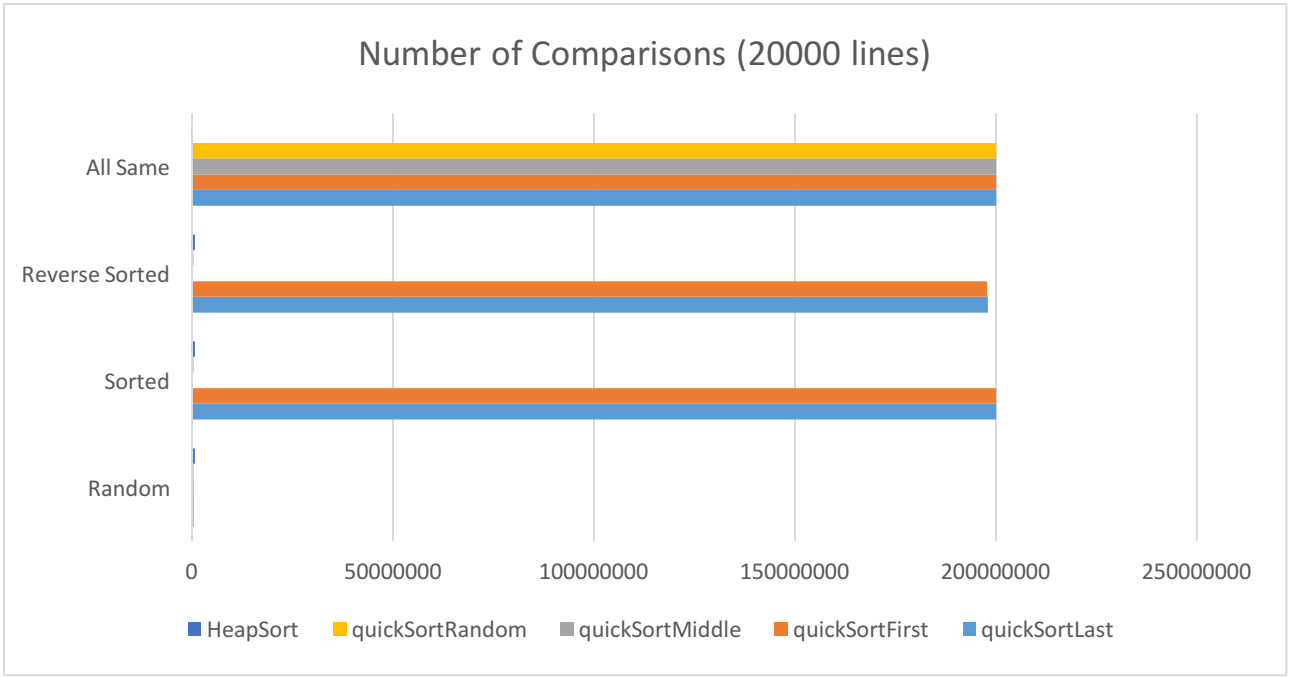
Here are a few graphs showing each algorithms performance in both time and comparisons against each data set.

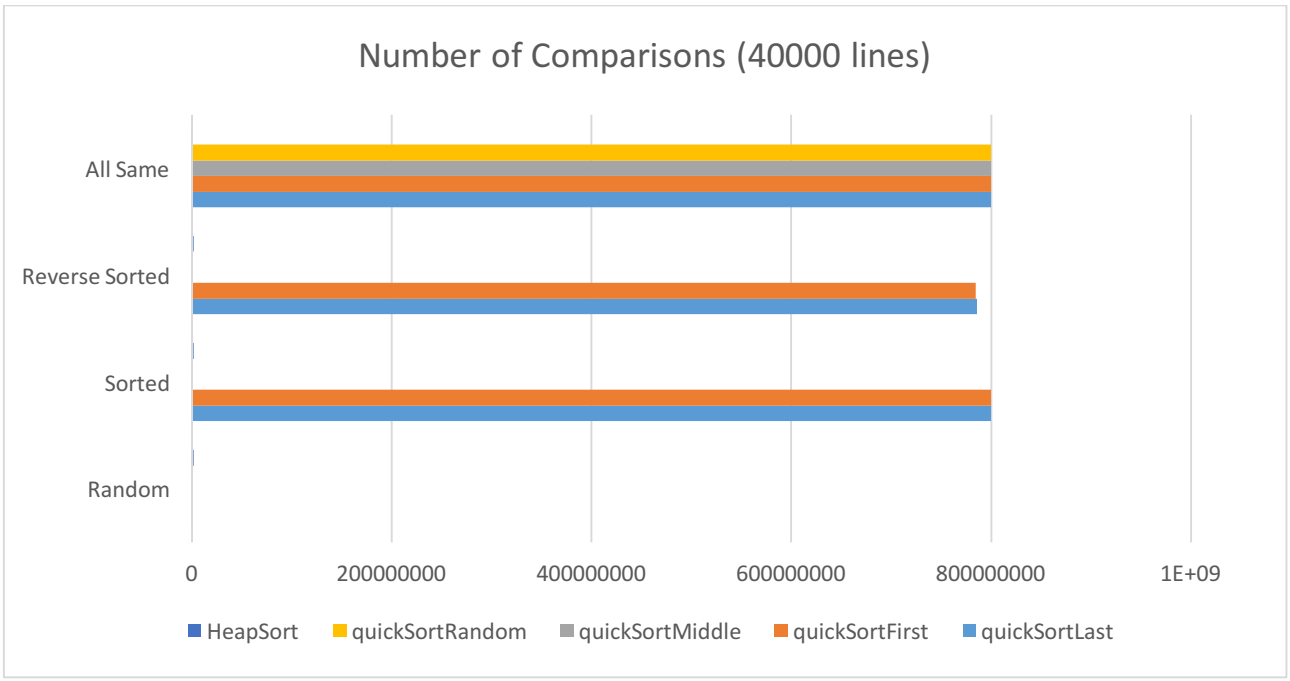


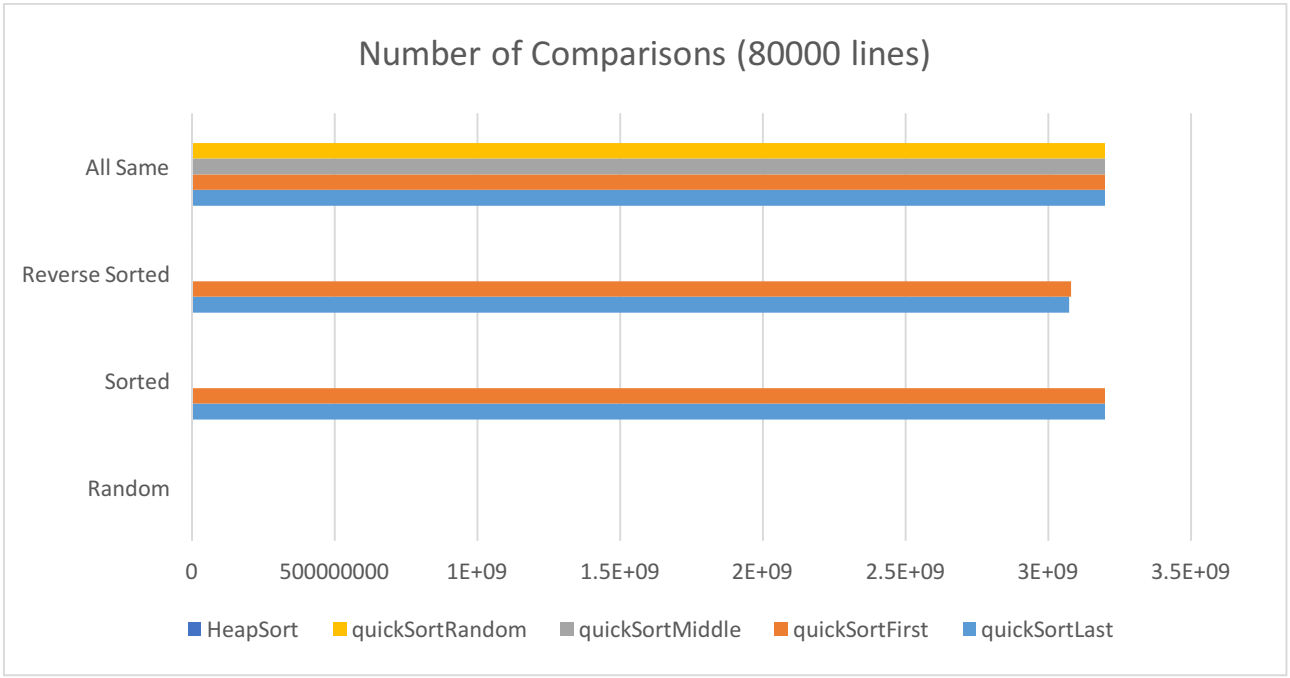


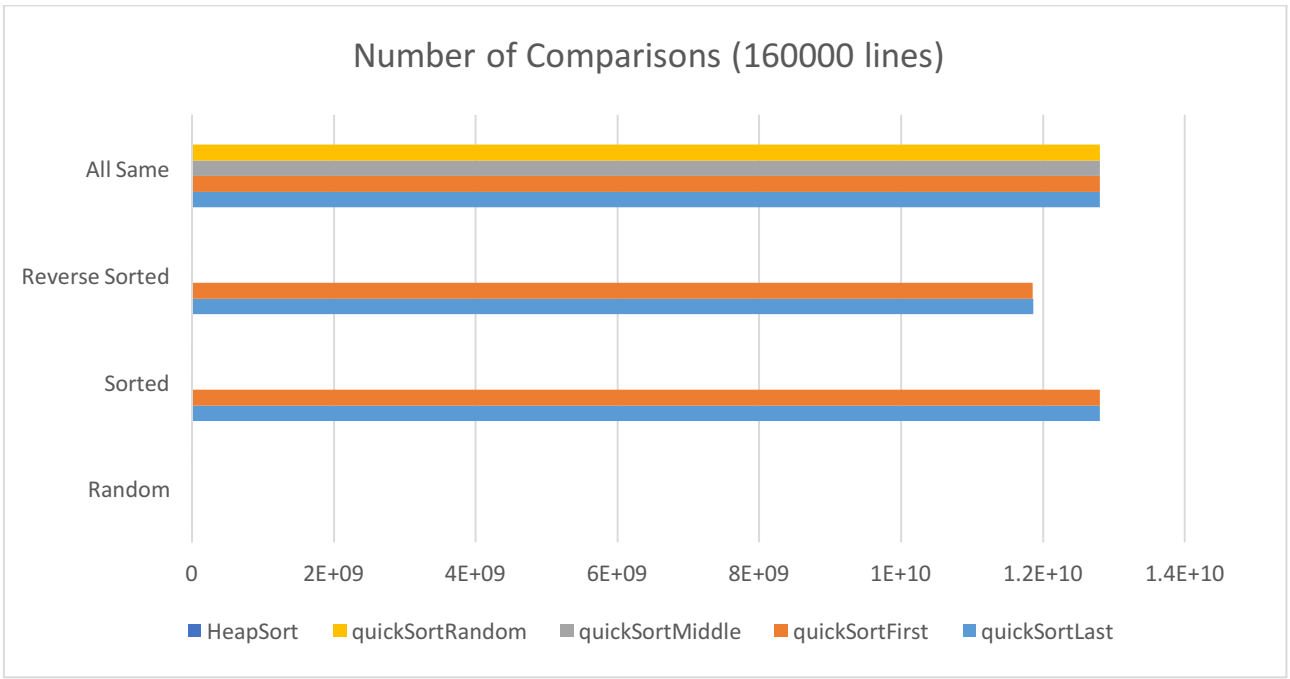

Time Comparison (80000 lines, in nanoseconds)
All Same
Reverse Sorted
Sorted
Random
0
5E+09
1E+10
1.5E+10
2E+10
HeapSort
quickSortRandom
quickSortMiddle
quickSortFirst
quickSortLast


Time Comparison (160000 lines, in nanoseconds)
All Same
Reverse Sorted
Sorted
Random
0
1E+10
2E+10
3E+10
4E+10
5E+10
6E+10
7E+10
HeapSort
quickSortRandom
quickSortMiddle
quickSortFirst
quickSortLast


Number of Comparisons (20000 lines)
All Same
Reverse Sorted
Sorted
Random
0
50000000
100000000
150000000
200000000
250000000
HeapSort
quickSortRandom
quickSortMiddle
quickSortFirst
quickSortLast


Number of Comparisons (40000 lines)
All Same
Reverse Sorted
Sorted
Random
0
200000000
400000000
600000000
800000000
1E+09
HeapSort
quickSortRandom
quickSortMiddle
quickSortFirst
quickSortLast


Number of Comparisons (80000 lines)
All Same
Reverse Sorted
Sorted
Random
0
500000000
1E+09
1.5E+09
2E+09
2.5E+09
3E+09
3.5E+09
HeapSort
quickSortRandom
quickSortMiddle
quickSortFirst
quickSortLast


Number of Comparisons (160000 lines)
All Same
Reverse Sorted
Sorted
Random
0
2E+09
4E+09
6E+09
8E+09
1E+10
1.2E+10
1.4E+10
HeapSort
quickSortRandom
quickSortMiddle
quickSortFirst
quickSortLast

As we can see from the data, quicksort definitely exhibits $O(n^2)$ complexity when the data is already sorted, particularly if the pivot is either the first or last element. The reverse sorted is slightly less bad, but still performs very poorly with a side pivot. The easiest way to avoid this worst case complexity is to use a pivot in the middle. You can randomize as well, but it doesn't provide much of an improvement over a center pivot. Some of this added time for the random algorithm may have been caused by the time it took to generate a random number. When I first implemented it I sought to get a normal distribution of random numbers, but implementing a more advanced random number library and utilizing it took twice as long! This is likely due to using C++ which already operates very efficiently, so adding any layer of complexity to the algorithm definitely stands out.

The worst case by far was if all the data is the same. Regardless of pivot, QuickSort will take every single possible branch to realize it is "sorted". Surprisingly this was the best case for HeapSort, perhaps because it was guaranteed to create a balanced tree. As such, if datasets are highly similar I would not recommend QuickSort as a sorting solution.

In the average case, all variations of QuickSort performed about the same. All of them outperformed HeapSort by about half the time in these cases. However the worst cases of QuickSort are very bad indeed, and analyzing a dataset (an O(n) walk of it) prior to selecting a sorting algorithm would greatly reduce the chance of encountering slow performance. Overall, the data came out as expected, with QuickSort outperforming HeapSort unless it encountered it's known worst case scenarios. I have definitely learned to be cautious when implementing algorithms to avoid these $O(n^2)$ scenarios, as they take quite a long time to process even on modern hardware.