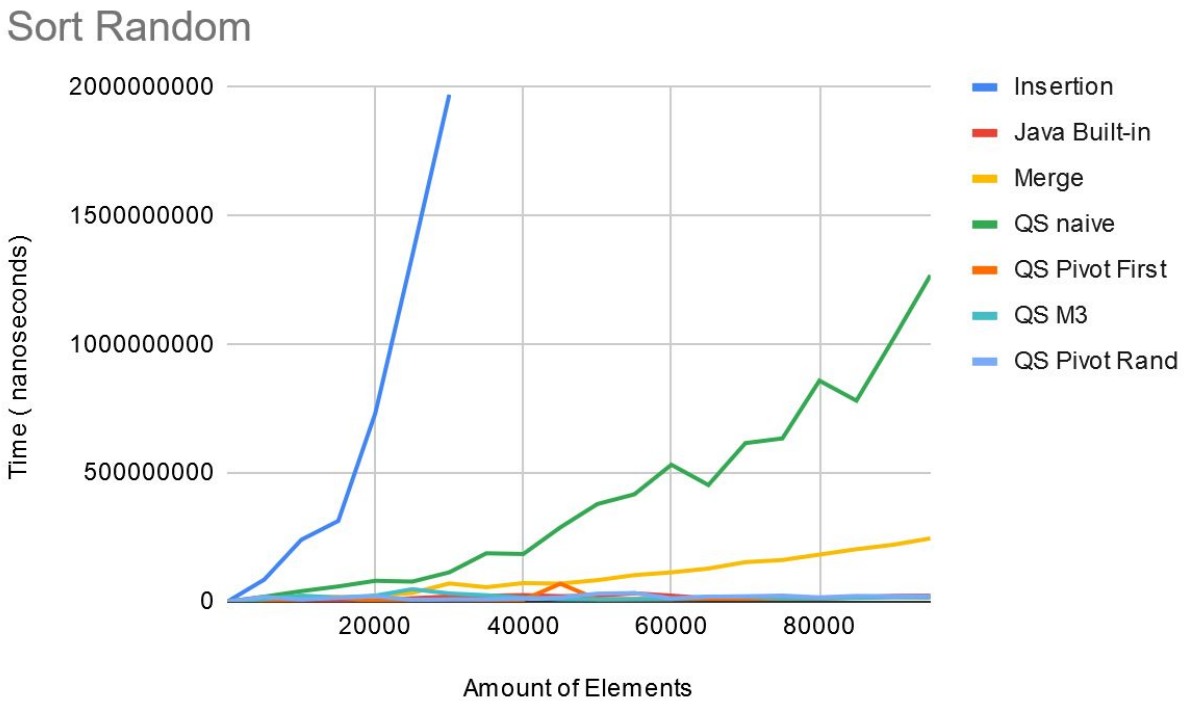
Josh Wells and Jonathan Oliveros
Prof. Swaroop Joshi
Assignment 05
Thurs. 10/3/19

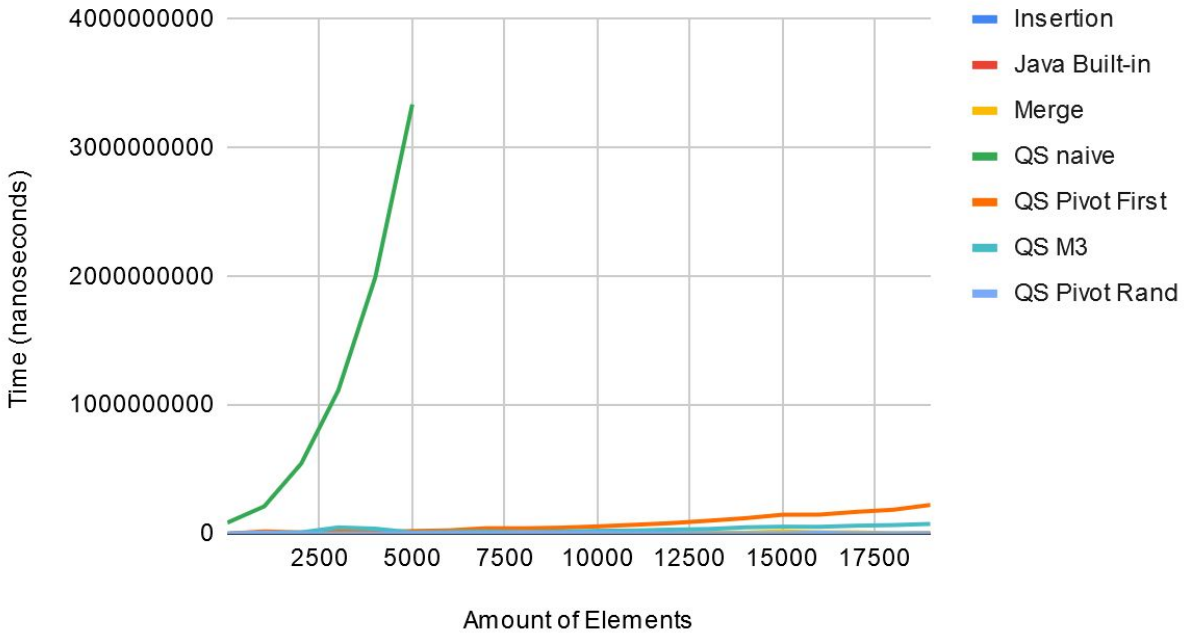
1.

The following graphs are the overall results of our sorting algorithms put under different circumstances:

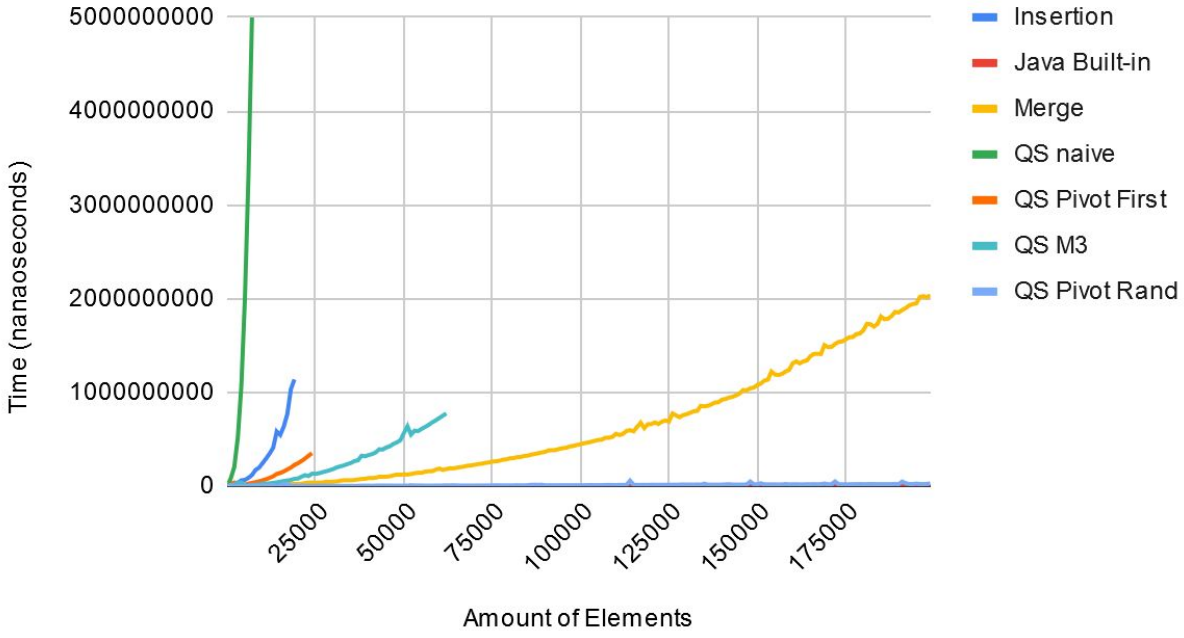


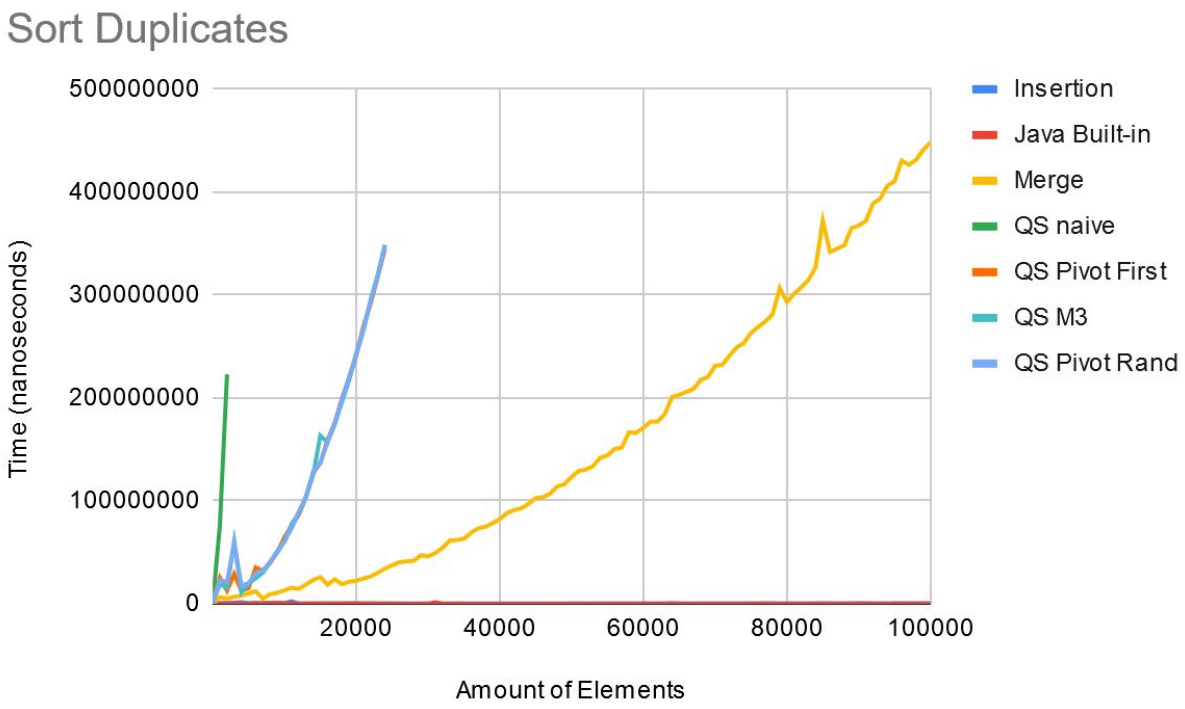
Overall, in an unsorted list the results turned out to be exactly what was expected. Insertion Sort has complexity $O(n^2)$, merge sort and and the variety of quicksorts end up with $O(n \log_2 n)$, although quicksort grows a little bit slower (disregarding quicksort naive since it is not meant to be efficient). Java built-in sorter stayed at about the same rate if not slower than quicksort.

Sort Sorted



Sort Reversed

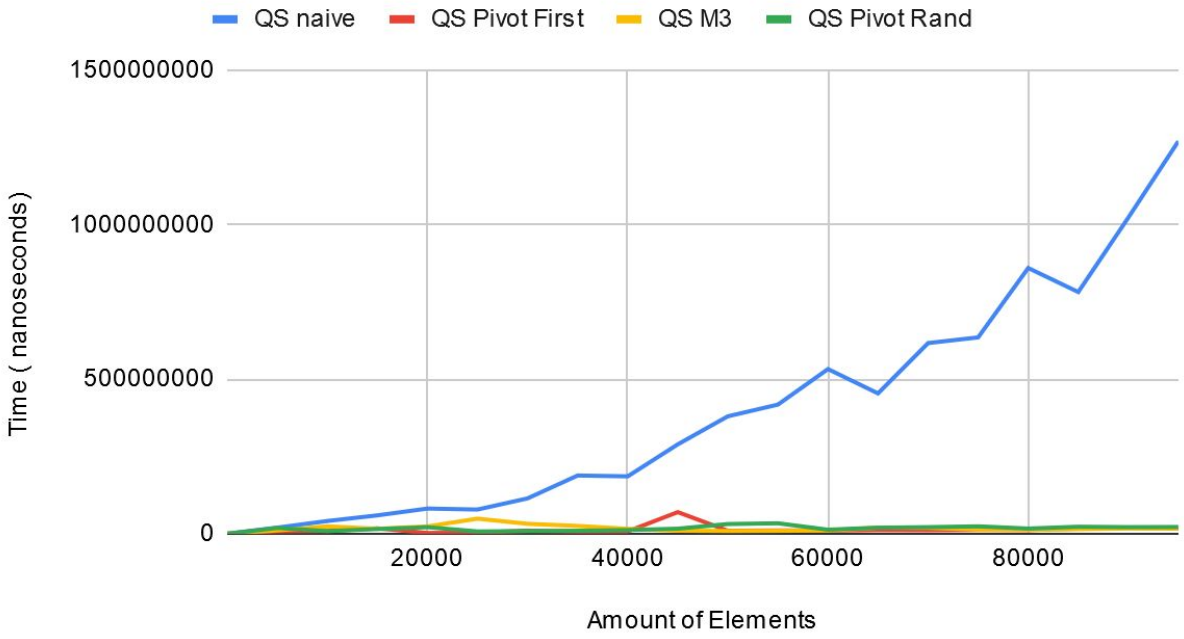




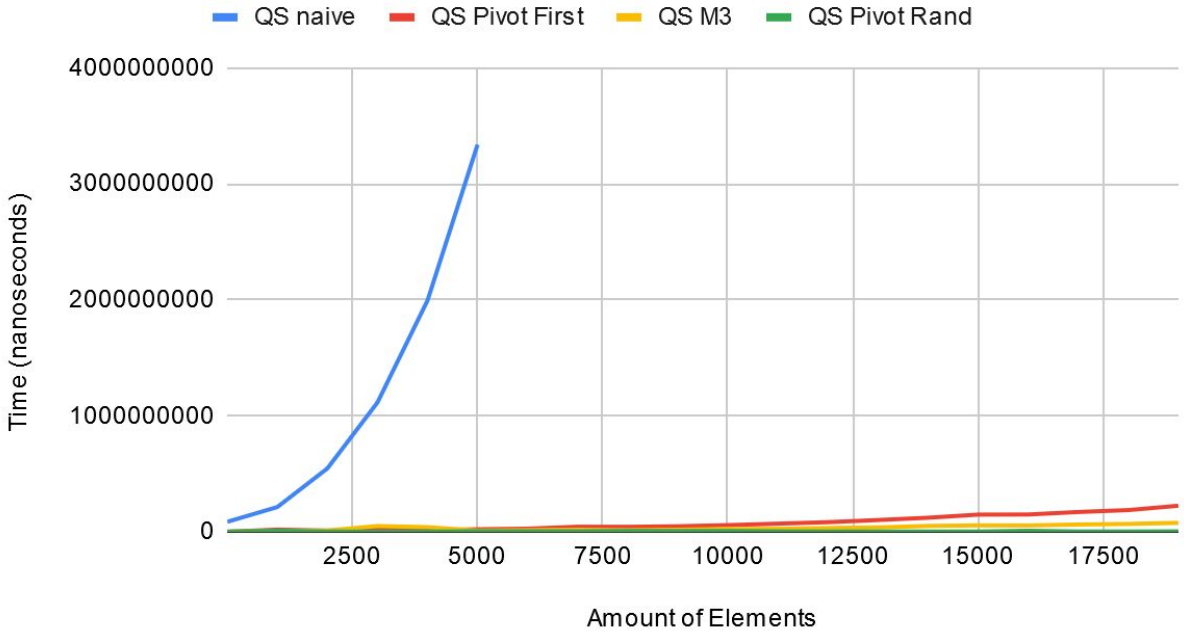
Comparing the rest of the graphs to the randomly sorted above, we can see that the results are much different. When the list is already sorted, quicksort naive does not handle it well and its complexity grows exponentially. Even worse, most of the algorithms start to break down when the list is sorted in reverse. The quicksort naive changes to O(n!) and eventually timed out because it took too long to compute each step. The different types of quicksort are not able to handle the sheer size of the elements and cause stack overflow issues. Surprisingly, merge sort handles it really well and it still grows at a $O(n \log_2 n)$ rate and Java built-in handles it the same as the randomly sorted. When they start to sort duplicate elements, the quicksort algorithms reach their worst case of $O(n^2)$ before they reach a stack overflow and mergesort stays at the $O(n \log_2 n)$. Quicksort naive and Java built-in sort stay where they were before. My hypothesis for Java built-in sort always looking like it works well is that it is a combination of sorting algorithms that get used depending on the situation, and recursion is not always used since it never has a stack overflow issue like the quicksort algorithms do.

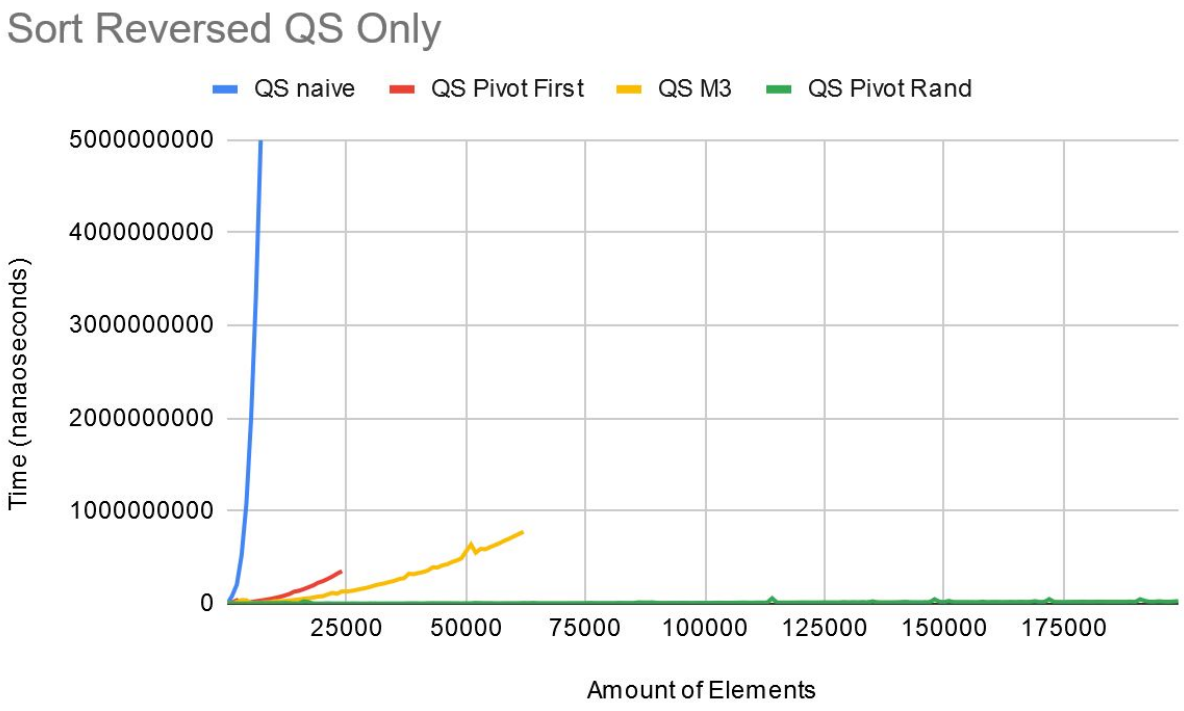
Narrowing down the results to only the quicksort algorithms are as follows:

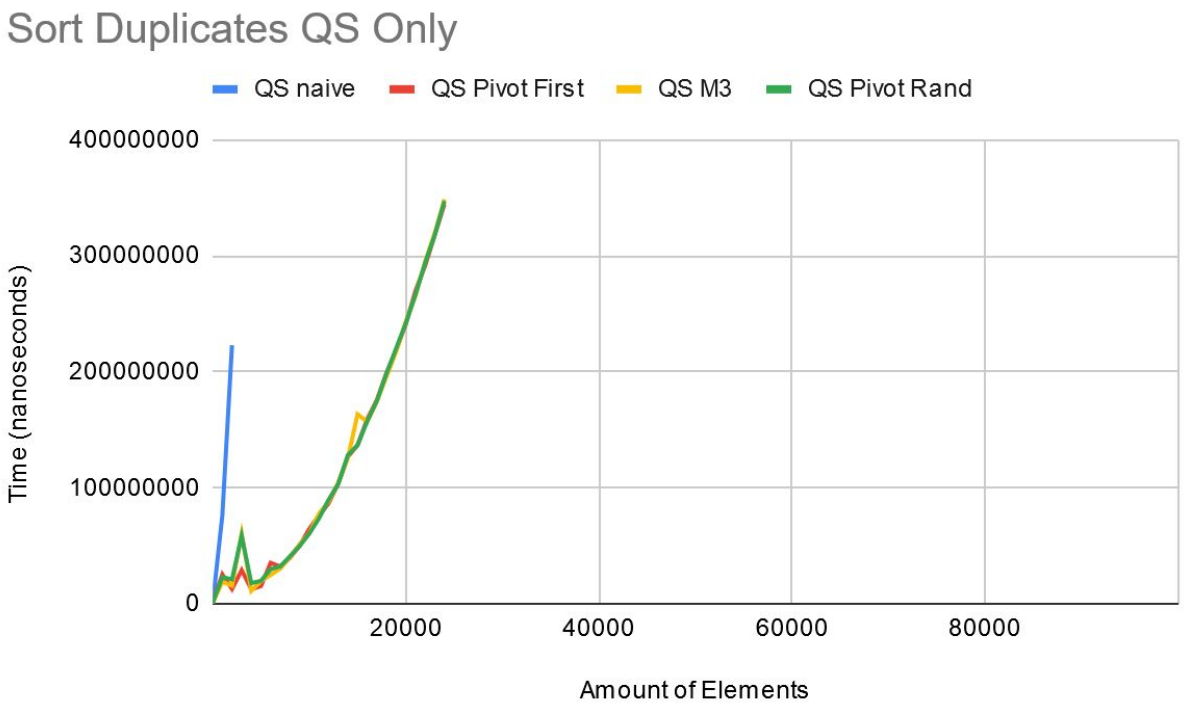
Sort Random QS Only



Sort Sorted QS Only



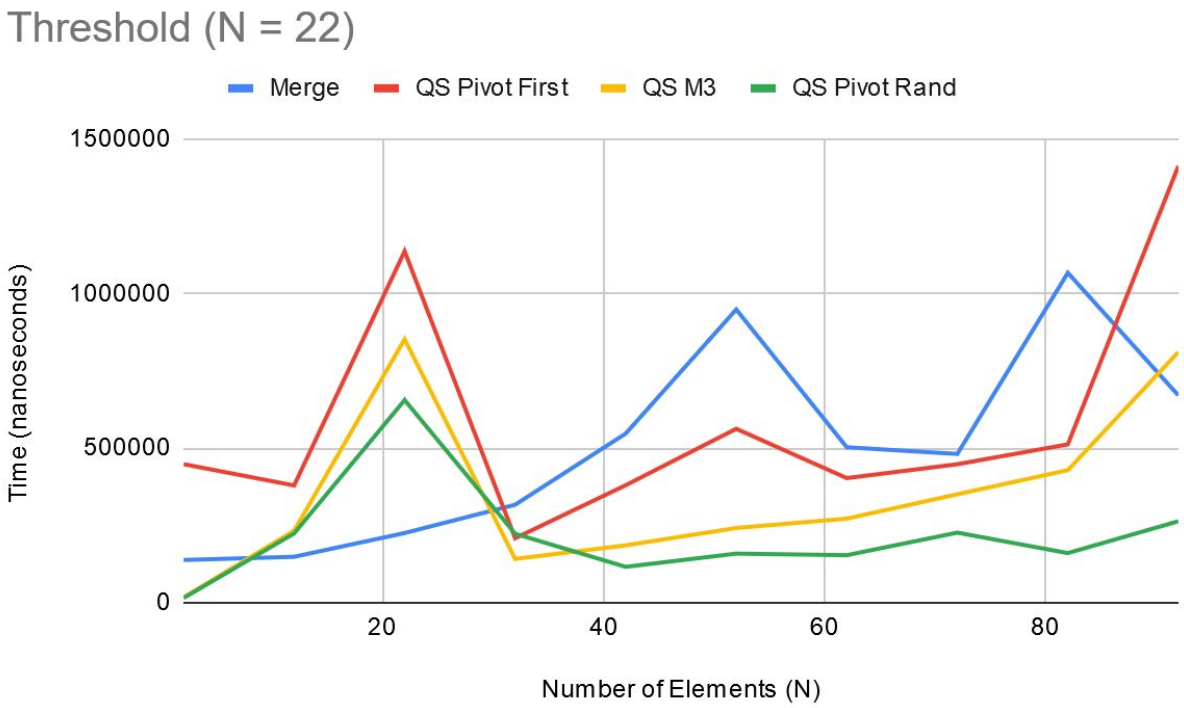




The only one out of all of the quicksorts that stayed fairly consistent is the quicksort pivot random. This makes sense since no matter the order of the list, the pivot is chosen randomly

therefore the sorting algorithm will behave as if the list was unsorted. The only situation where the algorithm doesn't work is when the list is filled with only duplicate elements.

Lastly, we zoomed in on the mergesort and quicksort algorithms to see when insertion sort ends and their own sort begins:



As we can see there is a major dropoff at 22. This is done intentionally to prevent the complexity from turning into $O(n^2)$.

2.

By finding the average, the two flaws that show up are that it would add another O(n) since it has to either compare and do a sequential search to find the middle element or add each element and divide by the number of elements in the list. Another flaw that comes up is that once the average is found, there might not exist a number in the list that would be close to it. Therefore by making a non-existent number as the pivot, it would throw off the whole algorithm.

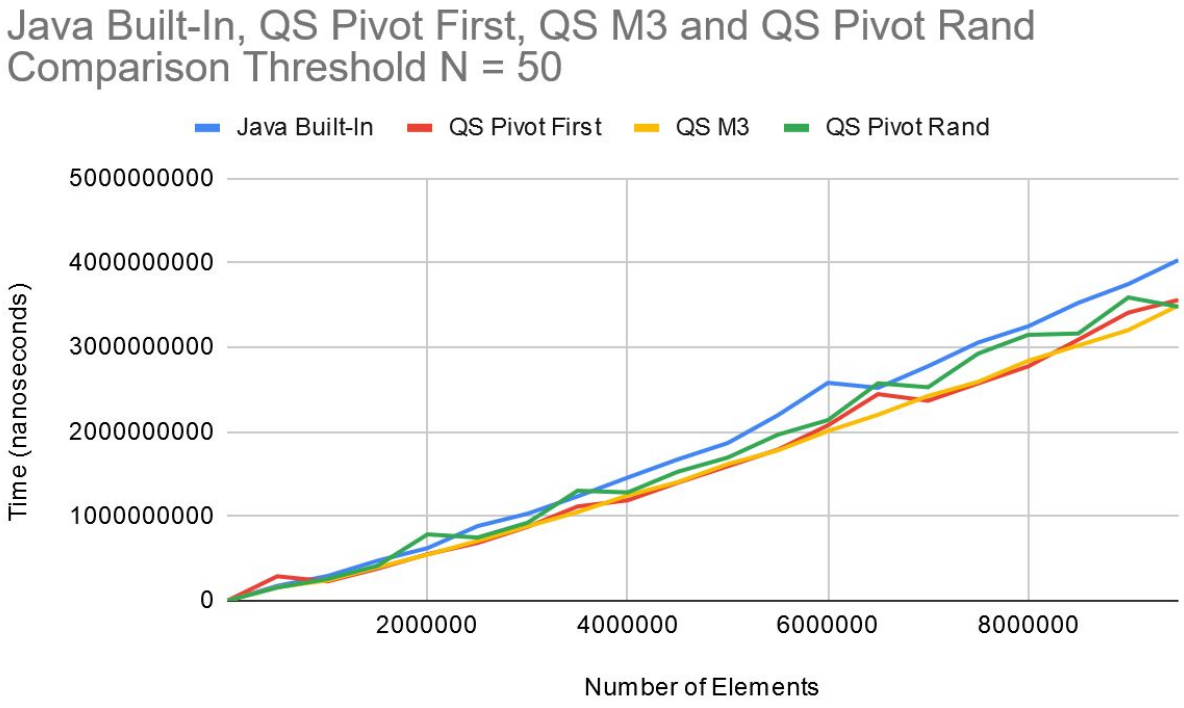
3.

a) Implementation tweaks such as a random pivot or a pivot at the median of three handle specific potential problems with the condition of the list before it is sorted. For example, if the list is sorted in a certain way, pivot at the first item could produce a partition with only one side. Pivot M3 ensures that at least one element is in each partition, but statistically

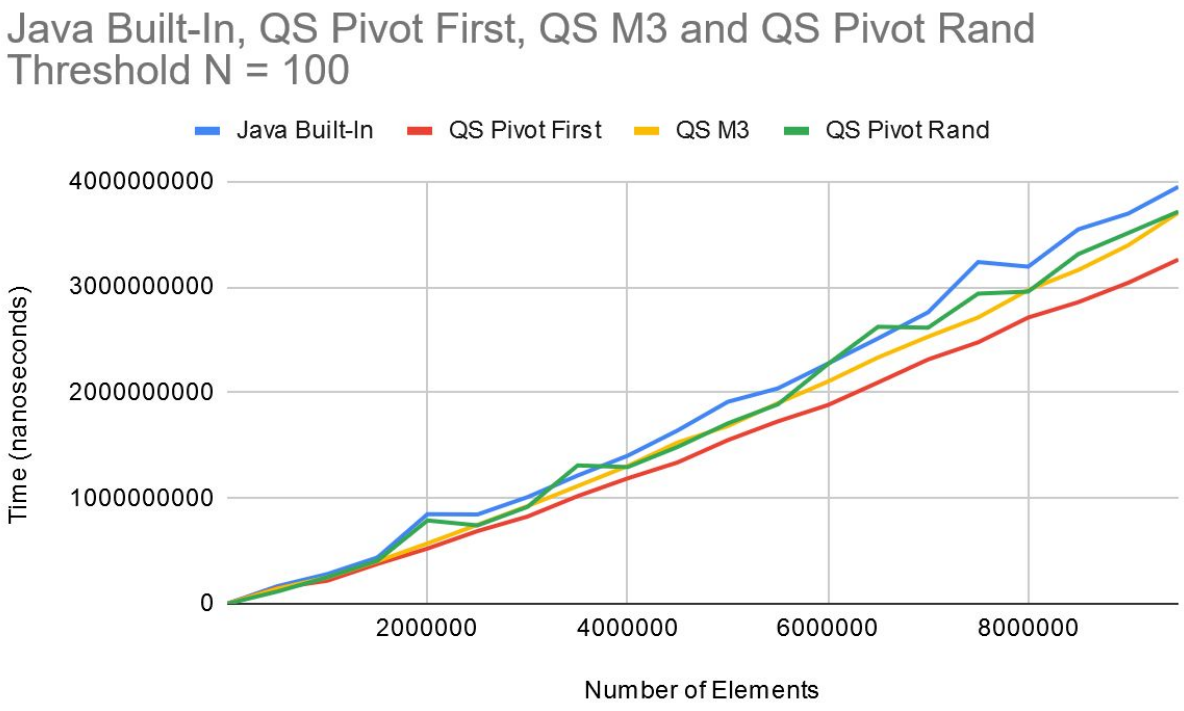
guarantees that the list will not be sorted in the least optimal way. Similarly, pivot random gives the bonus of not being a constant pivot point, and therefore statistically this will be a decent pivot most of the time. Overall, the pivot random seems to be the best in most cases, however, it struggles with duplicates in the list whereas M3 does better with duplicates in comparison to how it handles a randomly generated list.

b) Some choices we made to improve performance include creating as few objects as possible with the QuickSortAbstract methods for sorting and partitioning. This was possible because it was an in place sorting algorithm, whereas the MergeSort class required us to create multiple arrays for every recursion. Additionally, after we looked at the source code for the java built-in sort, we noticed that it uses many different sorting strategies for many different situations. It is as if they made the threshold method much more robust and able to handle different sizes, as well as situations. If we were to create an optimal sorting type, it would include a robust threshold method as well, with whatever type of sorting would be most effective for the size and condition of the list before it was sorted.

Additional Graphs using a randomly generated list

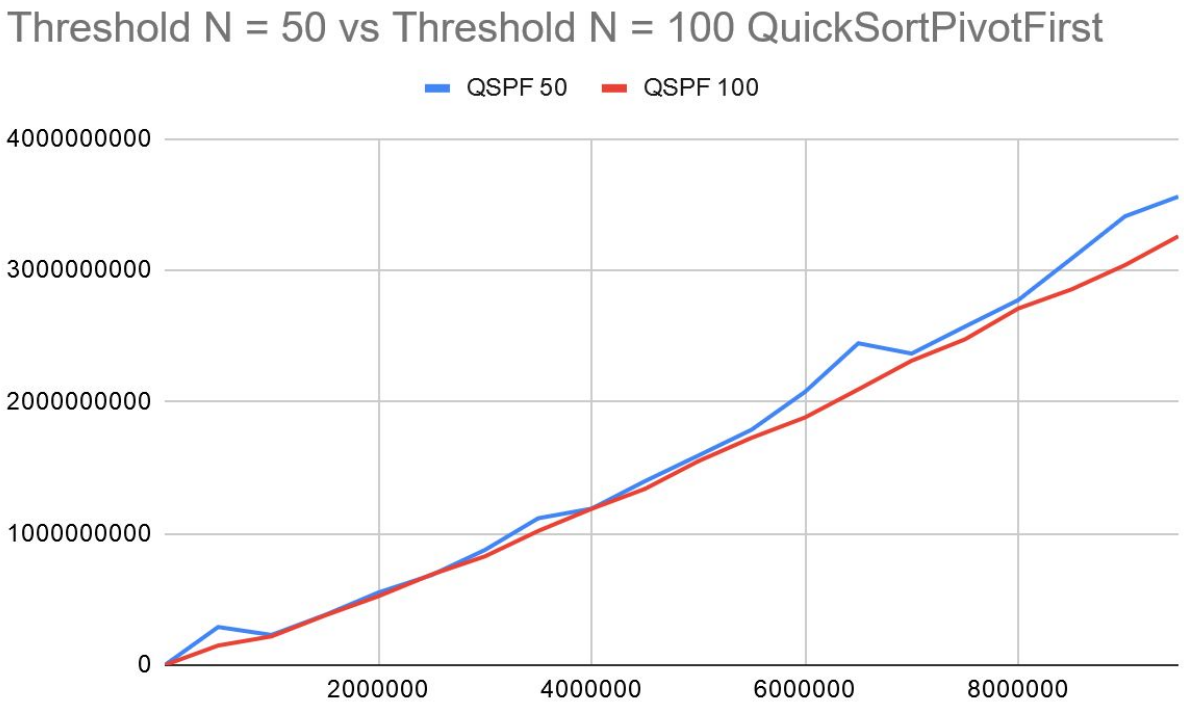


This is a graph to compare our methods to the java built in methods on a typical random list using a threshold of 50.

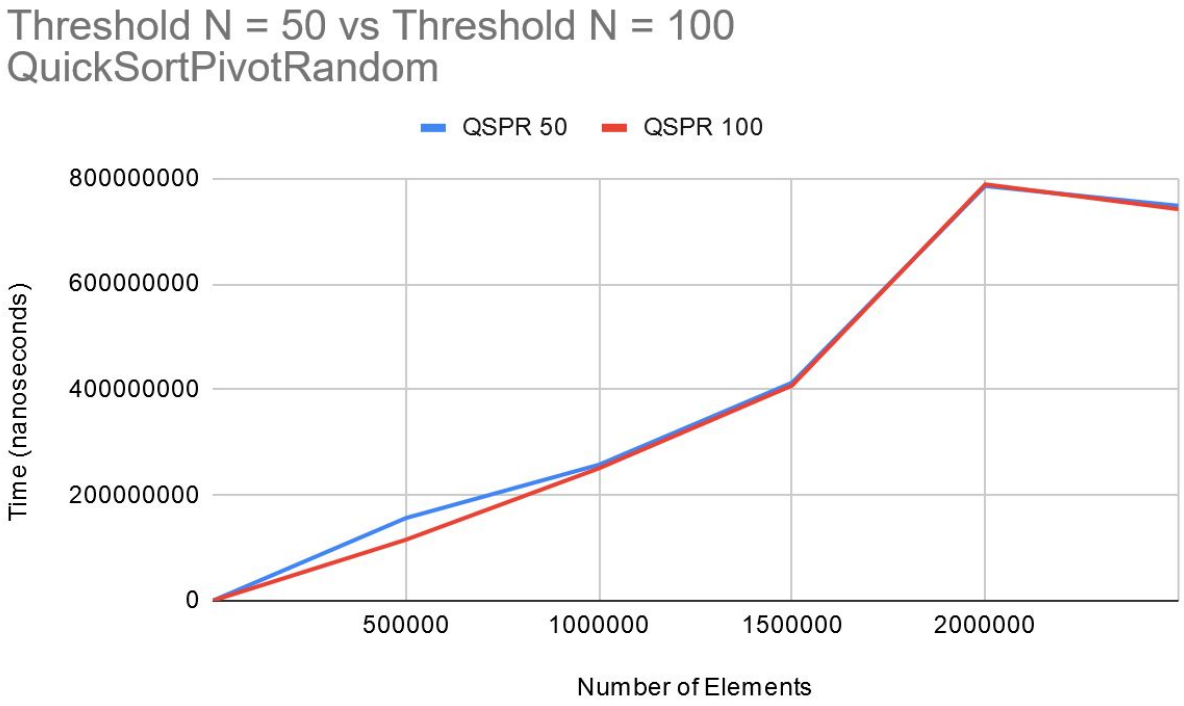


This is a graph to compare our methods to the java built in methods on a typical random list using a threshold of 100

This graph compares how the QuickSortPivotM3 object performs at threshold 50 vs 100. It performs better at a threshold of 100 for the first fourth of the graph, then the threshold at 50 performs better for the rest of the graph until ten million.



This graph compares the QuickSortPivotFirst object with a threshold of 50 and a threshold of 100. The threshold of 100 appears to be a more consistent graph which avoids random losses in performance and generally is faster at sorting.



This graph compares the QuickSortPivotRandom object with a threshold of 50 and a threshold of 100. There appears to be no significant difference other than the threshold of 100 performing marginally better for the first million elements.