2XB3 Lab 3

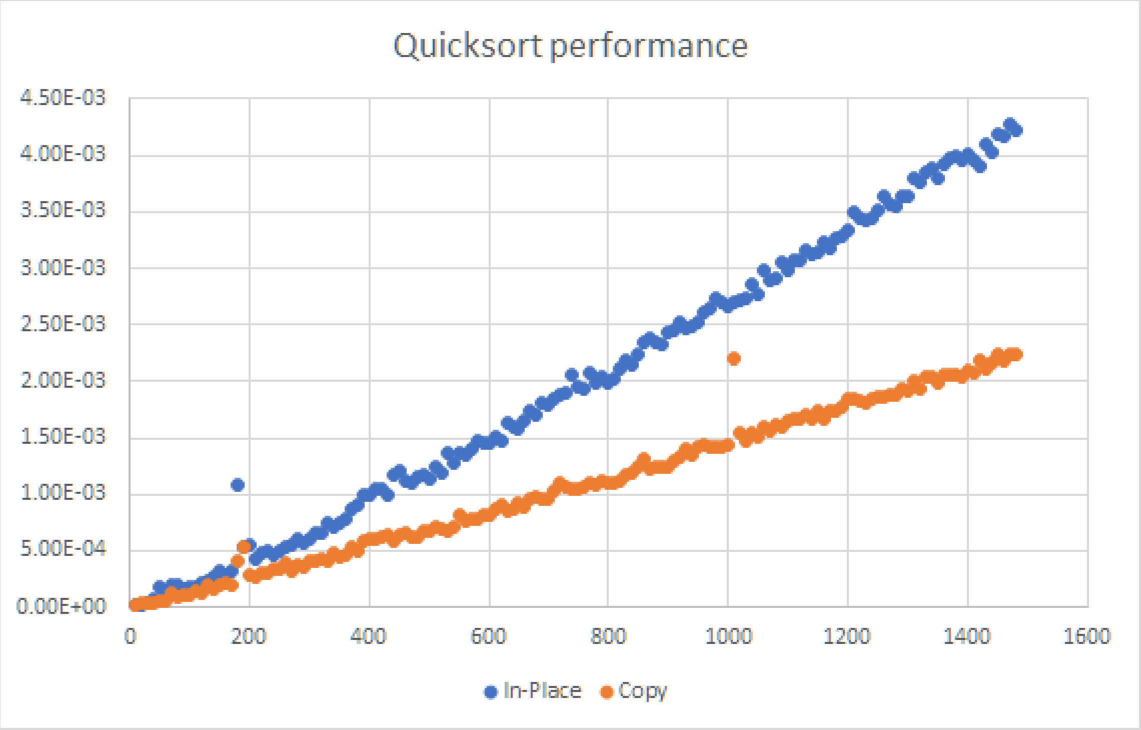
Casey Doede – 001223493 – doedecd@mcmaster.ca – L01

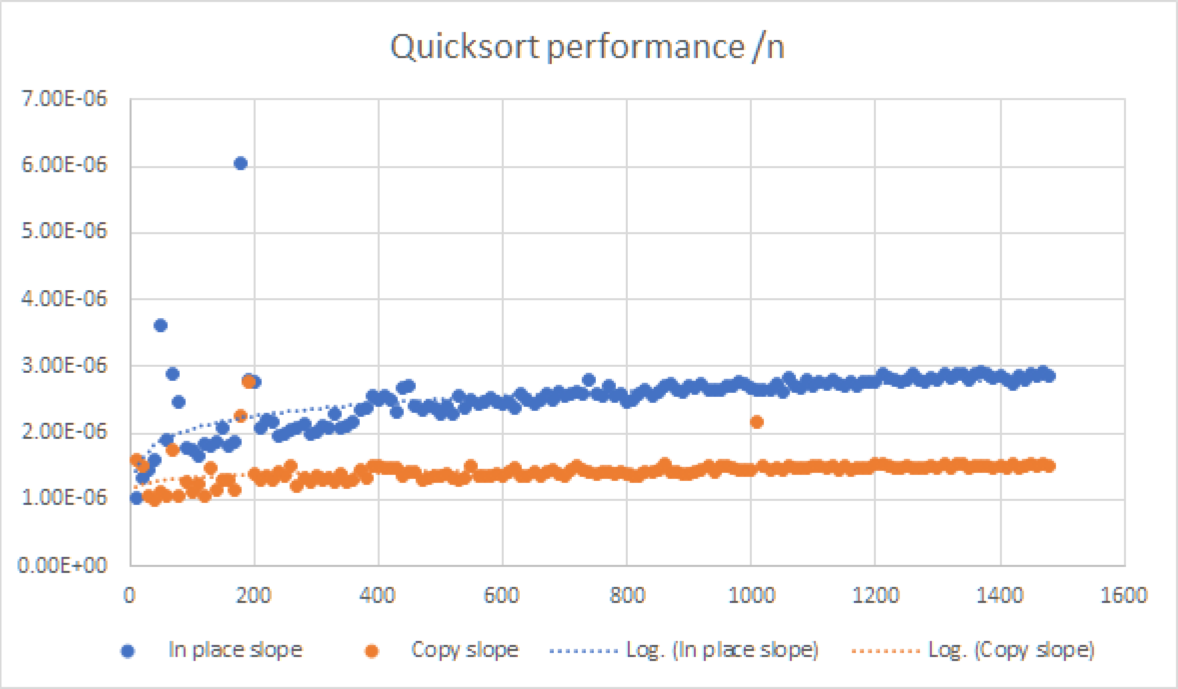
Amaan Ahmad Khan – 400230523 – khana251@mcmaster.ca – L01

Alex Axenti – 400268807 – axentia@mcmaster.ca - L04

In Place Quicksort:

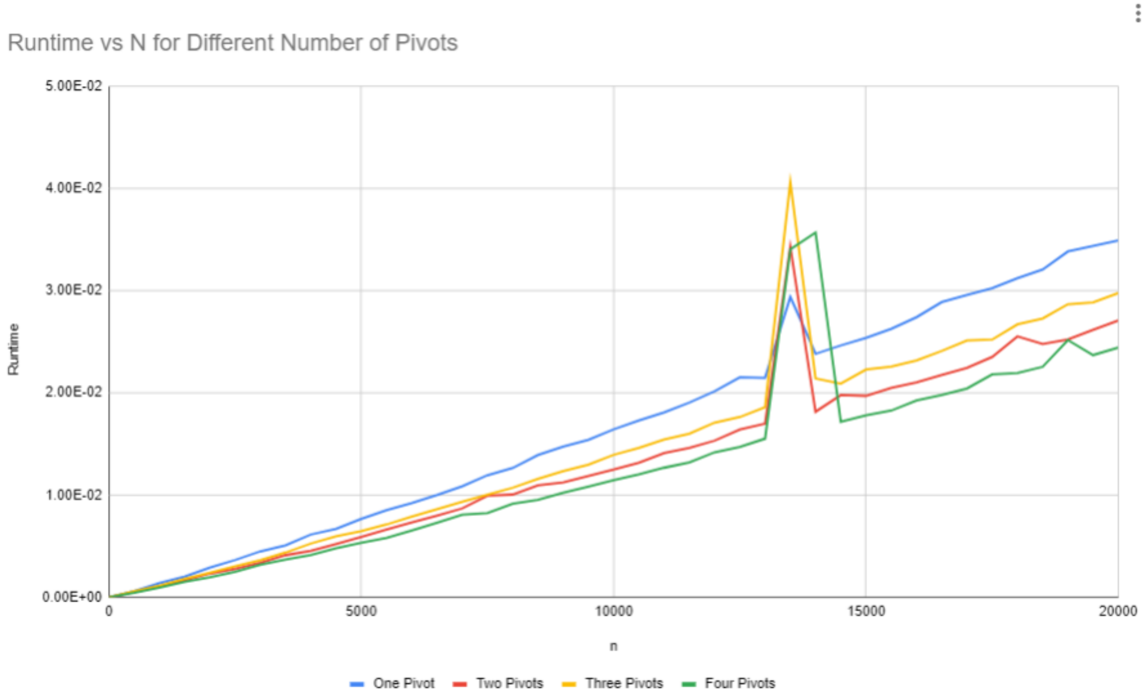
We assumed in place quicksort would suffer in performance in comparison with a quicksort that utilizes copying of arrays. In place requires repeated swapping of elements in the list to allow the quicksort algorithm to be recursively performed on the list that has been sorted via the pivot element. Copy on the other hand would suffer in the realms of memory performance as it requires the creation of multiple sub arrays.





After testing it was found that the copy iteration of quicksort does indeed perform better by the factor of some constant, appearing to be somewhere around 2. One can observe that both algorithms are still nlogn in complexity for time, but the copy version of quicksort would use O(n logn) memory as each recursive call copies each array to two arrays of size n/2, which is done log n times on average.

Multi-Pivot

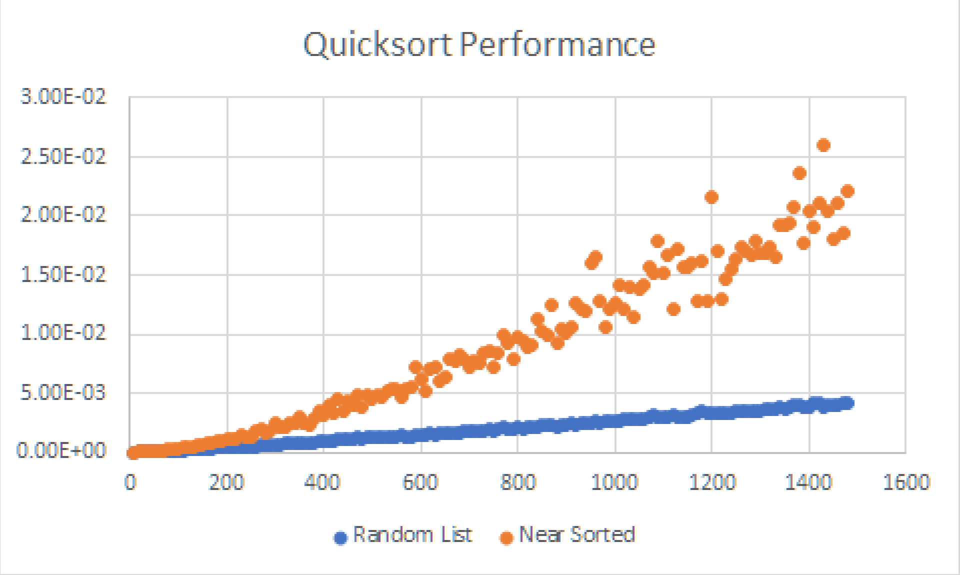


This is the result of comparing one to four pivot quicksort. The large spike was likely due to some background pc complications and is just an anomaly. For the rest of the graph though, four and two pivots were shown to be the fastest. The sorts were tested on 10 random lists for each iteration, so this data is for the average case. Four pivots was the fastest of the four quicksorts. As a result, the four pivot is the recommended quicksort of the four variants. In addition, another interesting result from the analysis was that the two pivots was faster than the three pivots.

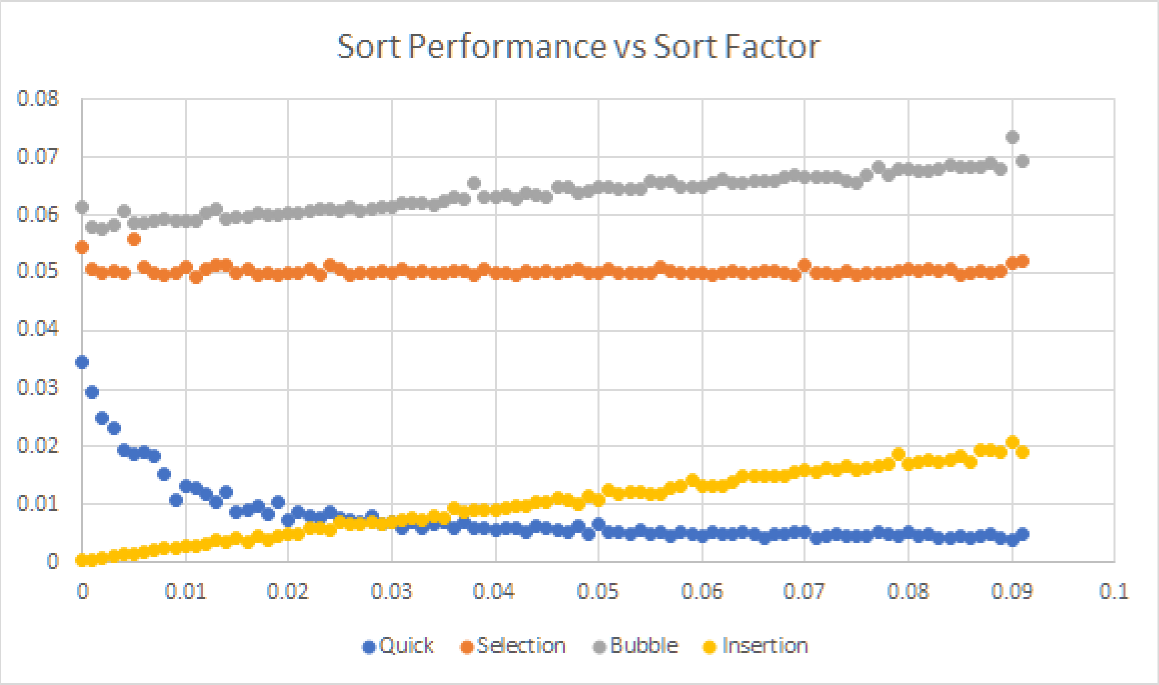
Finally for the remainder of the lab we chose to analyze a single pivot quicksort, while the performance is worse than the multi pivot quicksorts, each of their implementations utilized the copying of arrays, greatly increasing their memory complexity compared to an in-place quicksort.

Worst case quicksort

In the worst-case scenario with quicksort, a list would be sorted or nearly sorted. As then choosing the pivot to be the first or last element of a list would involve unequal partitioning and a much greater recursive depth to divide the list. We utilized a factor of 10 to emulate a random list, and a factor of .01 to emulate a near sorted list. The performance was then measured, and we observed a massive difference in performance. As n grows the performance suffers greatly, where it can be quite similar for small lists. This demonstrates the worst-case O(n^2) performance of quicksort



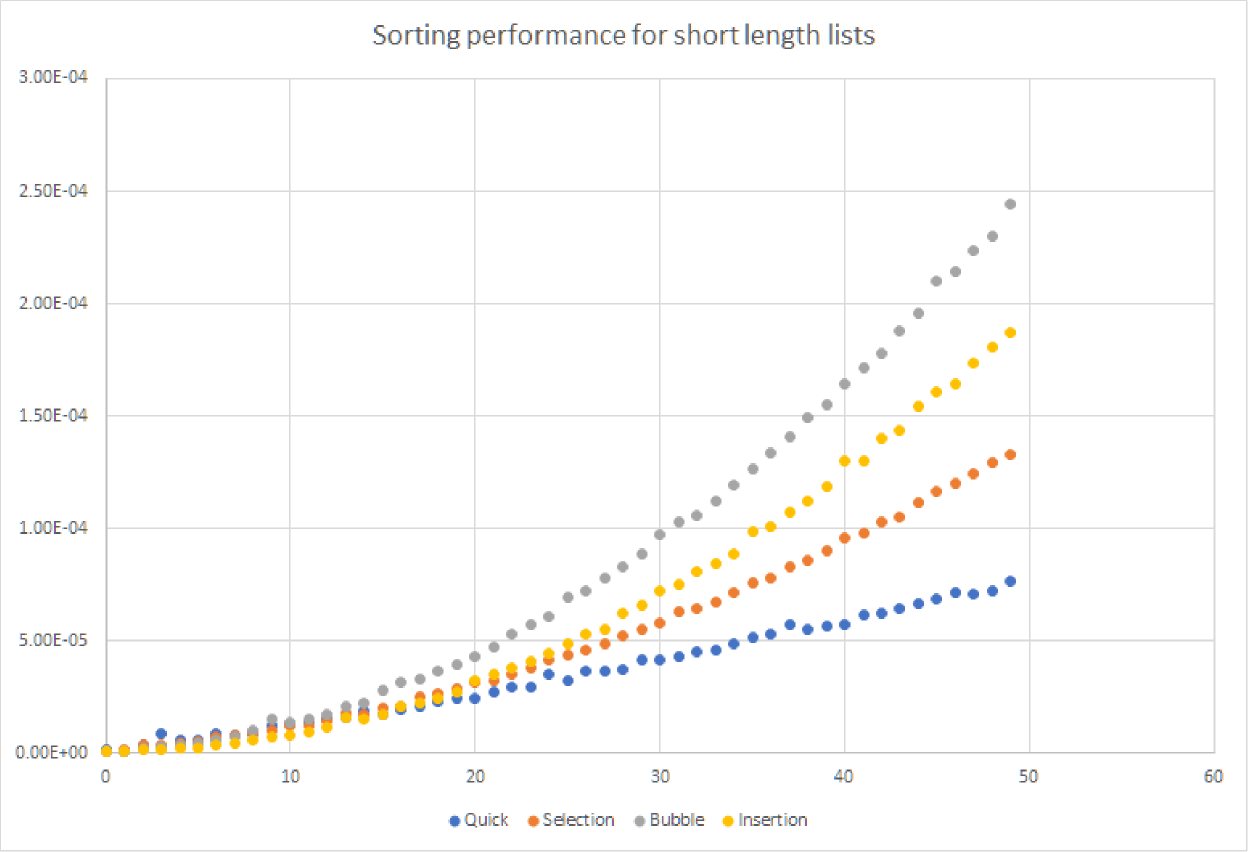
Insertion sort can have O(n) performance when a list is nearly sorted, and we will compare quicksort to elementary sorting algorithms. We fixed the list size at 1000 and varied the sorting factor from 0 to .1 via .001 increments. Analyzing this data one can observe the poor performance of quicksort with near sorted lists, and the excellent performance of insertion sort for near sorted lists. After a factor of .029 was reached quicksort overtook insertion sort. One can also observe that insertion sort still maintains improved performance compared to bubble and selection sort for these relatively sorted lists of length 1000.



\

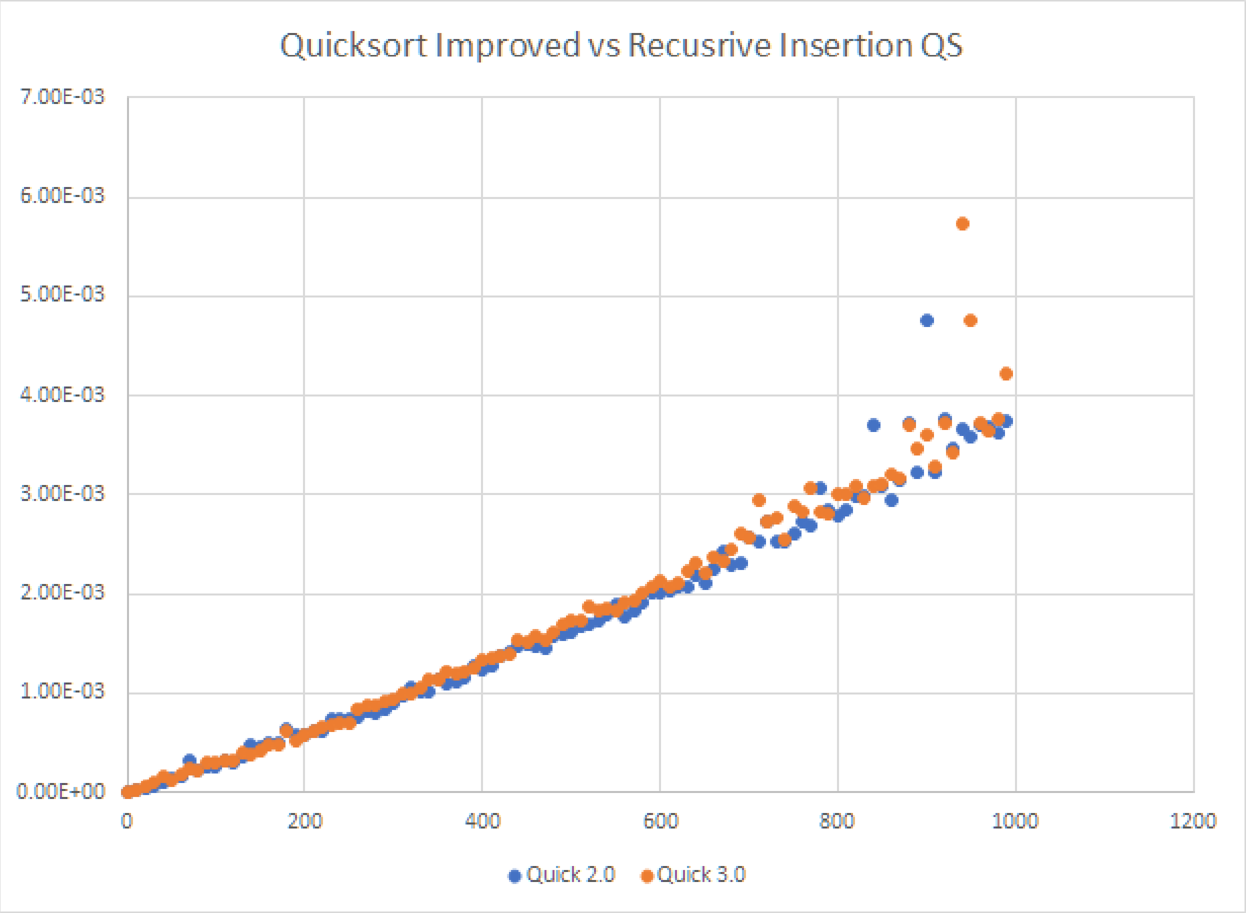
Small Lists:

Comparing quicksort with selection sort, bubble sort, and insertion sort we observed that both selection and insertion sort outperformed quicksort for lists with lengths less than 16. Bubble sort performed slightly worse, outperforming quicksort only until the lists were of length 8.

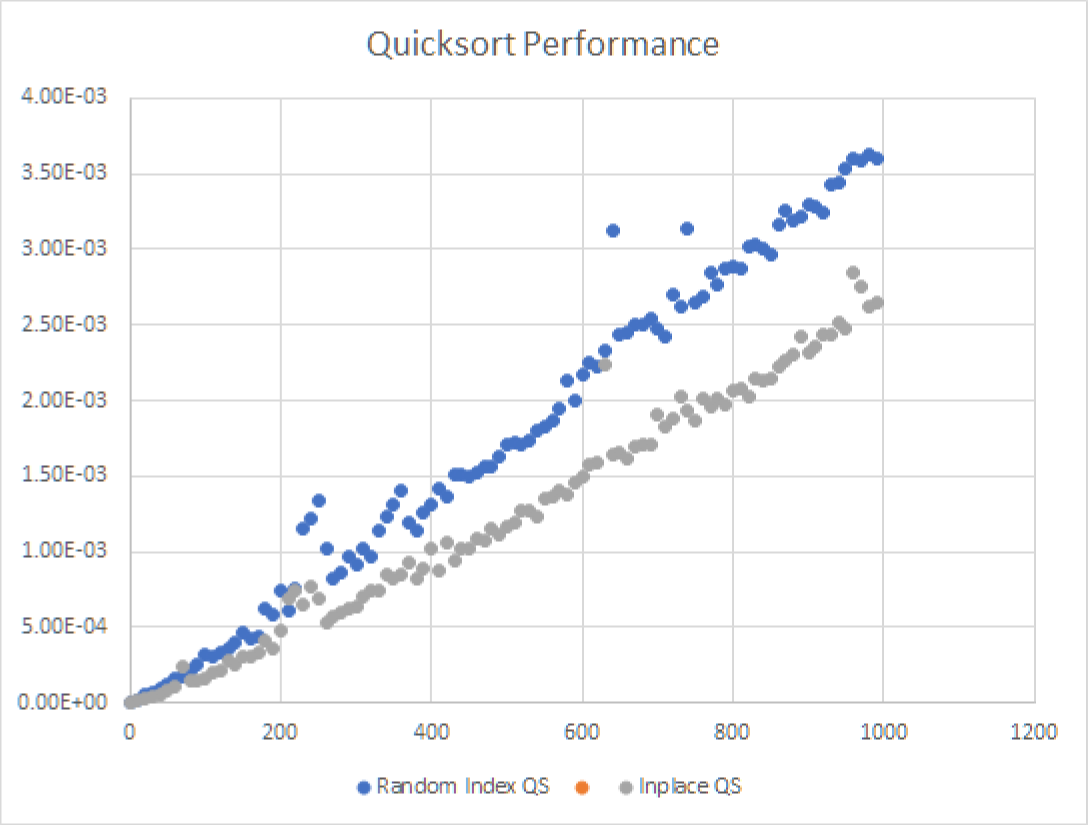


We then added an if statement to call insertion sort if the length of the list was less than 16, this improved performance compared to our chosen quicksort, though still slightly lagged behind insertion sort due to the added complexity and comparisons.

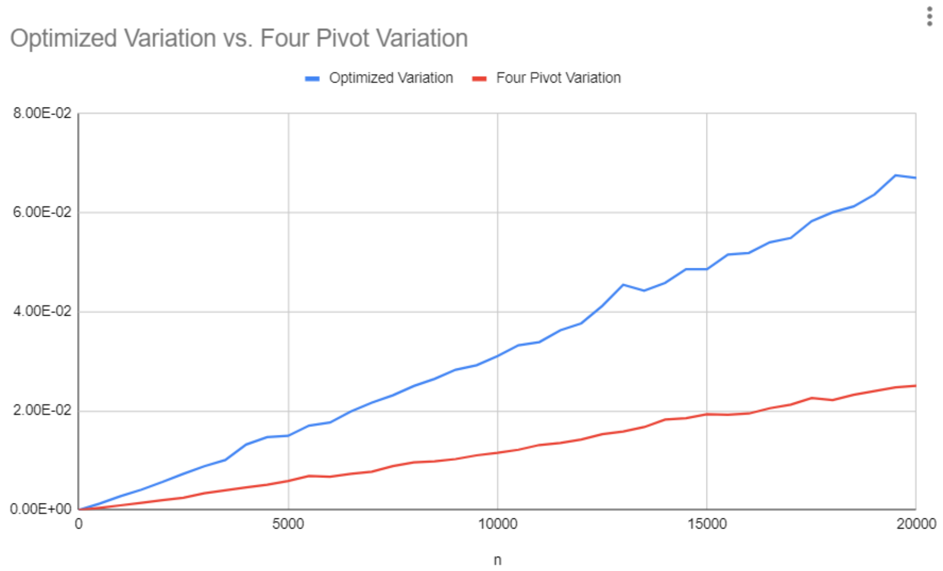
Afterwards, we tried another variation where insertion sort was called within the quicksort, whenever the pivot splits were less than length 15. Although, this did not have any noticeable result on the average case, due to the extra comparisons throughout.

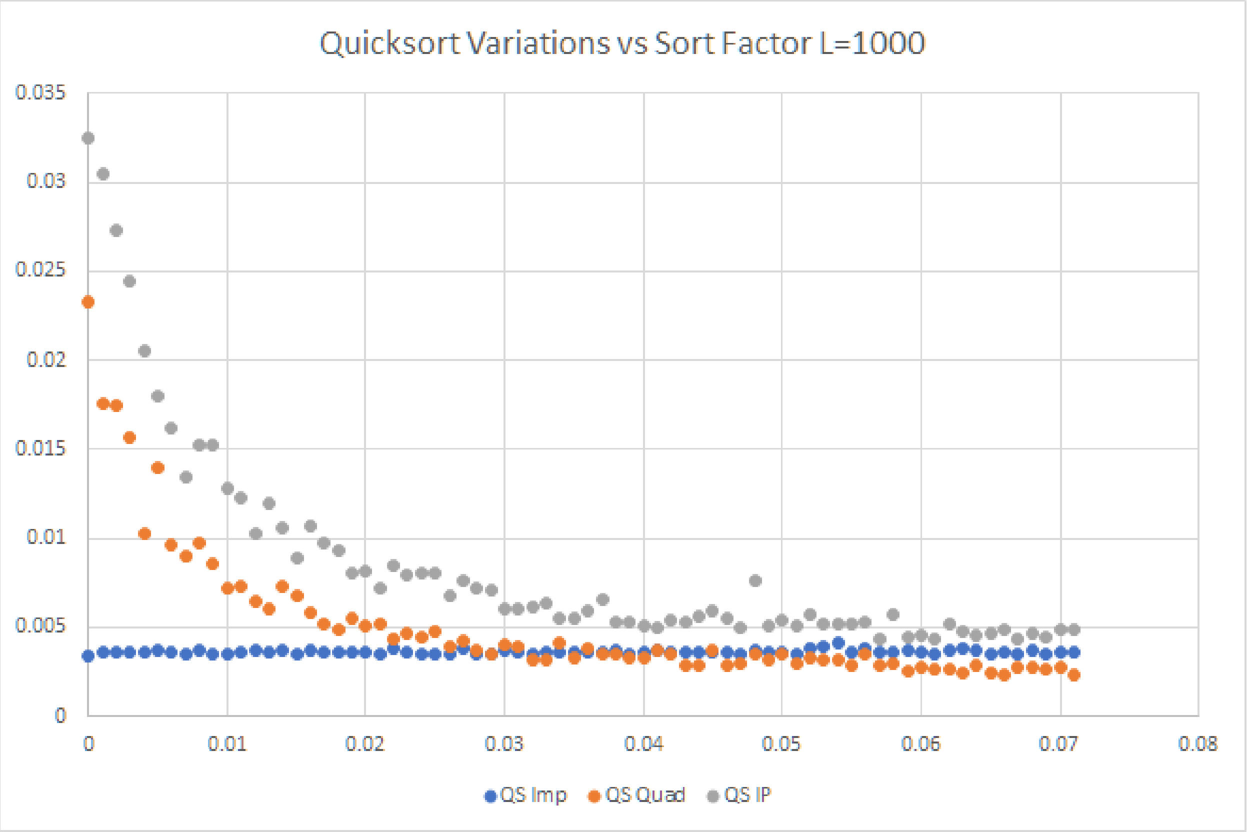


We further improved quicksort by changing the pivot values to be at random indexes in the list. We did this to try to improve the worst-case scenario where a list is near sorted, and the pivots were being chosen as the beginning index and creating uneven splits. Choosing a random pivot greatly reduces the chance of this occurring unless the randomly chosen value was the same as the greatest or smallest value in the list. Though we found that this made the average case take longer due to the extra operations. We feel this tradeoff is overall a reasonable one to make as a nearly sorted list is a relatively common occurrence, though one could choose to utilize a traditional quicksort algorithm if they knew their data would always be random, though this is not usually the case in the real world.



Finally, we compared our improved version of quicksort with the 4-pivot implementation from earlier. It was found that the 4-pivot quicksort still performs better than our optimized version for random lists, though the space complexity is still far greater. With near sorted lists our improved quicksort can outperform a quad pivot quicksort until the sorting factor reaches .029, similar to that of insertion sort. Overall, we felt a single pivot optimized quicksort is the best performing version of the quicksort algorithm for general use. A multi pivot option can be preferrable but the complexity increases for implementation and utilizing a copy version of quicksort with multiple pivots will still utilize far greater amounts of memory than a single pivot in-place quicksort.


Optimized Variation vs. Four Pivot Variation
Optimized Variation
Four Pivot Variation
8.00E-02
6.00E-02
4.00E-02
2.00E-02
0.00E+00
0
5000
10000
15000
20000
n


Quicksort Variations vs Sort Factor L=1000
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
● QS Imp
● QS Quad
● QS IP