COMPSCI 2XB3 L04 Lab Report #3

Mahad Aziz - azizm17 - 400250379 - azizm17@mcmaster.ca
Talha Amjad - amjadt1 - 400203592 - amjadt1@mcmaster.ca
Logan Brown - brownl33 - 400263889 - brownl33@mcmaster.ca
McMaster University

COMPSCI/SFWRENG 2XB3: Binding Theory To Practice

Dr. Vincent Maccio

TA: Amir Afzali

February 5, 2021

In-Place Version:

Advantages of quicksort_inplace():

The big advantage of quicksort in-place is that it doesn't require extra memory space to manipulate data. However, it would only use that extra space for recursive calls. This makes sense as creating copies or temporary lists require the algorithm to use more space.

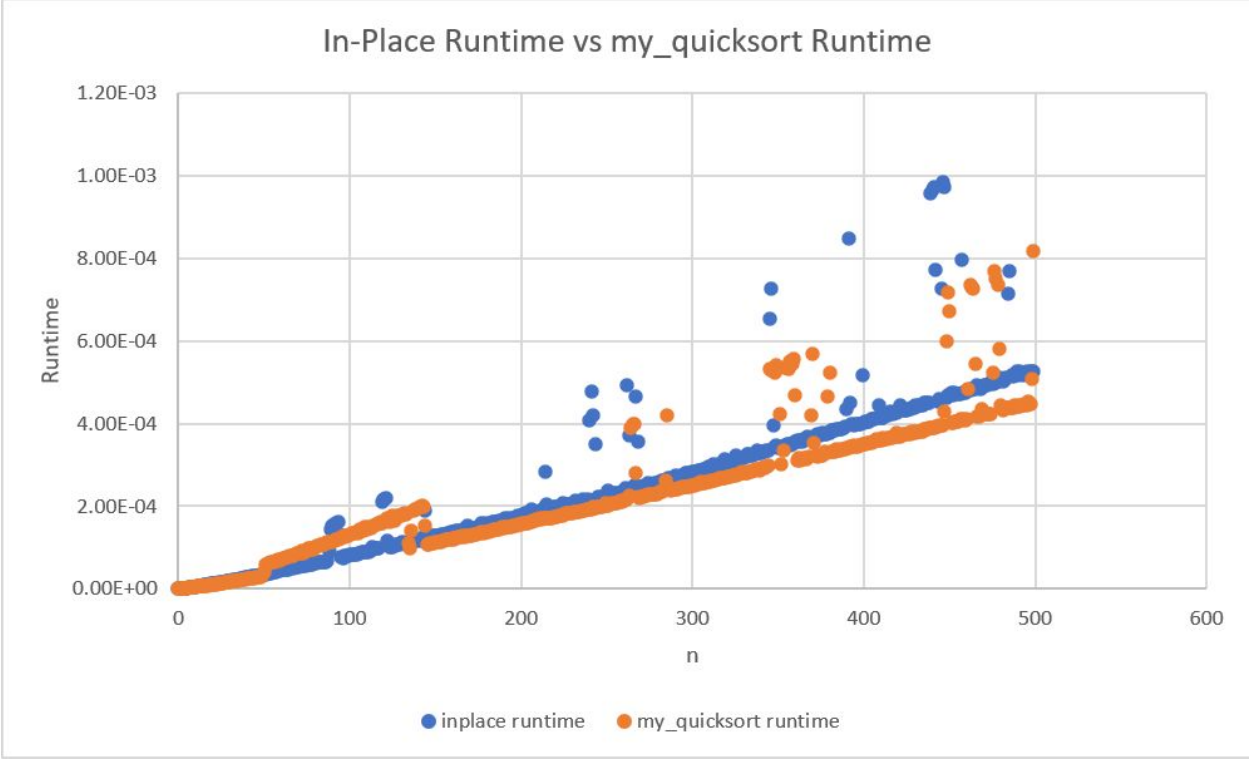
Which implementation is better? And by how much?

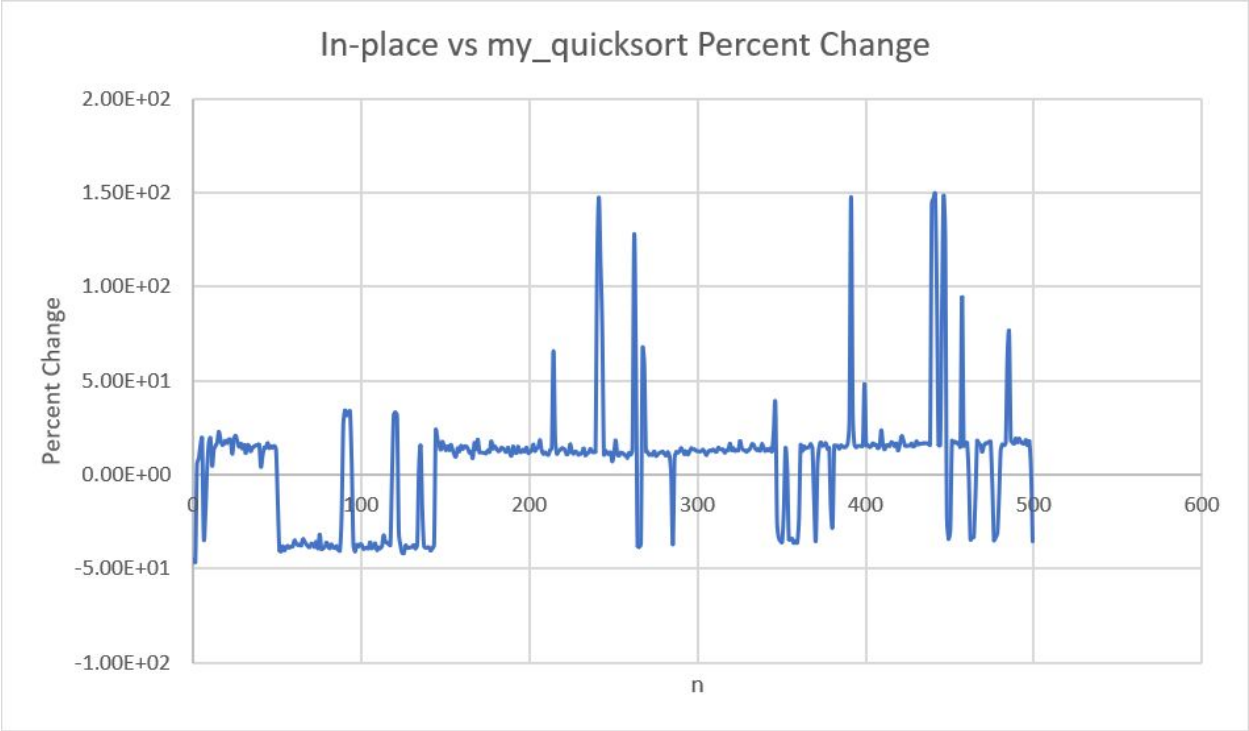
The in-place version is better because of its efficiency as it uses way less memory space to manipulate data compared to the normal version. After graphing the data we collected for both implementations, we found out that the implementation provided to us was faster than the in-place implementation we created. Using the percent change formula on the runtimes of the in-place quicksort and the my_quicksort function we can see that our in-place quicksort takes on average about ten to twenty percent more runtime to sort a list compared to the my_quicksort function. The abnormal shifts in the percent change of runtime for these functions could be a result of an anomaly in the hardware.

Which would you use in practice?

The trade-off is between memory and speed for these two implementations. If you wanted to sacrifice more speed and have extra memory space available, then the normal implementation is the way to go. However, the in-place implementation is much more efficient if you wanted to save memory space. As a result, we would use the normal implementation to get more speed since we currently do not run into limited memory problems.

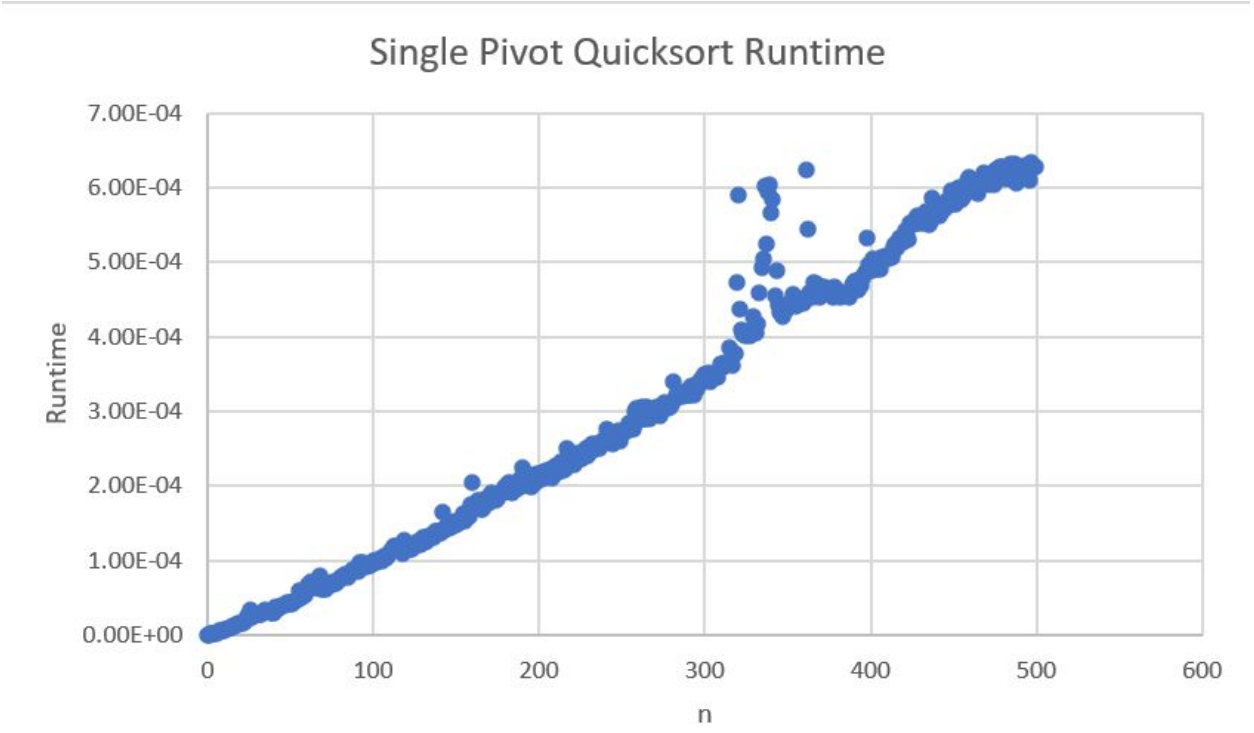
Testing:

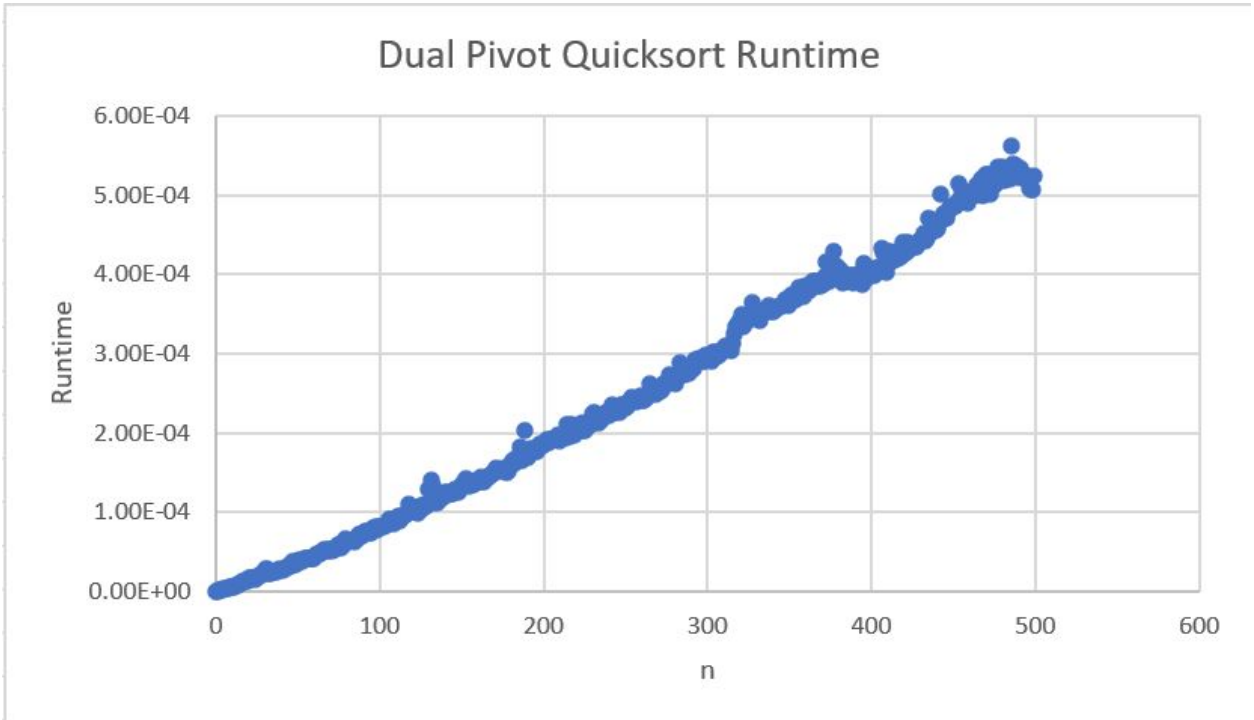

In-Place Runtime vs my_quicksort Runtime
1.20E-03
1.00E-03
8.00E-04
6.00E-04
4.00E-04
2.00E-04
0.00E+00
Runtime
0
100
200
300
400
500
600
n
inplace runtime
my_quicksort runtime

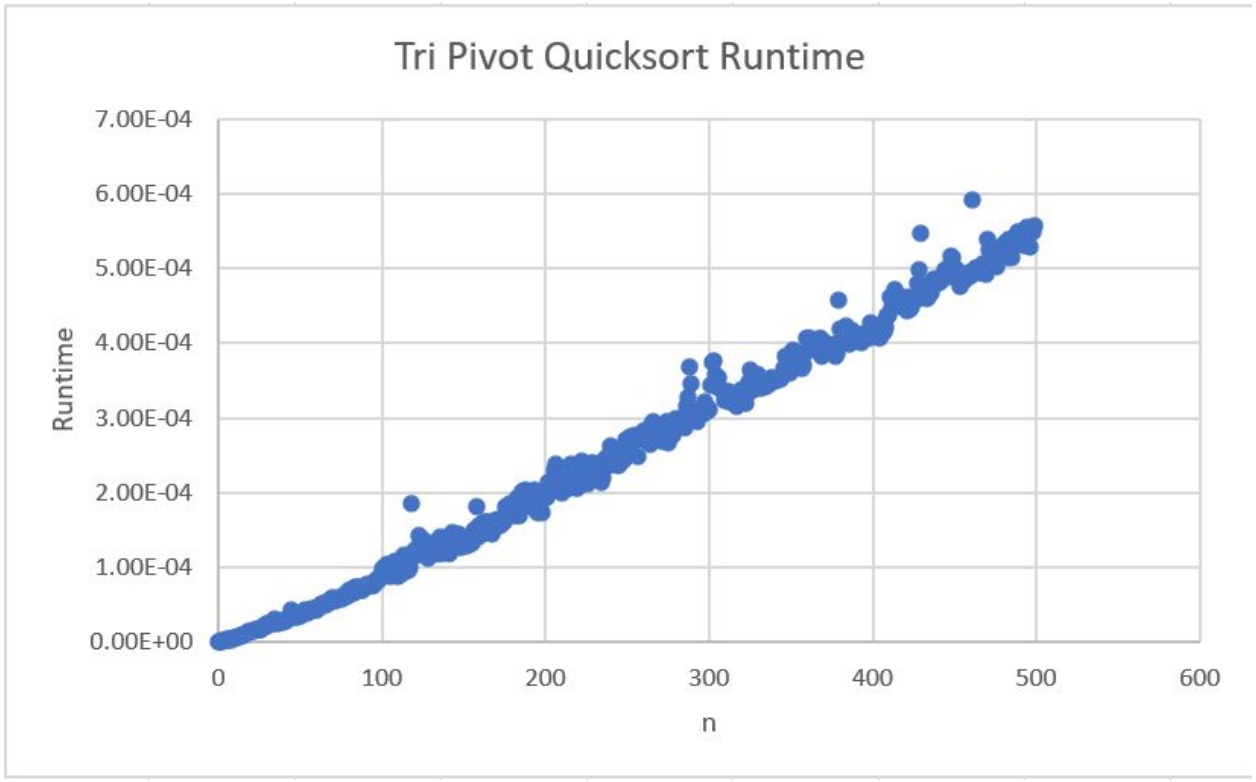

In-place vs my_quicksort Percent Change
2.00E+02
1.50E+02
1.00E+02
5.00E+01
0.00E+00
-5.00E+01
-1.00E+02
Percent Change
0
100
200
300
400
500
600
n

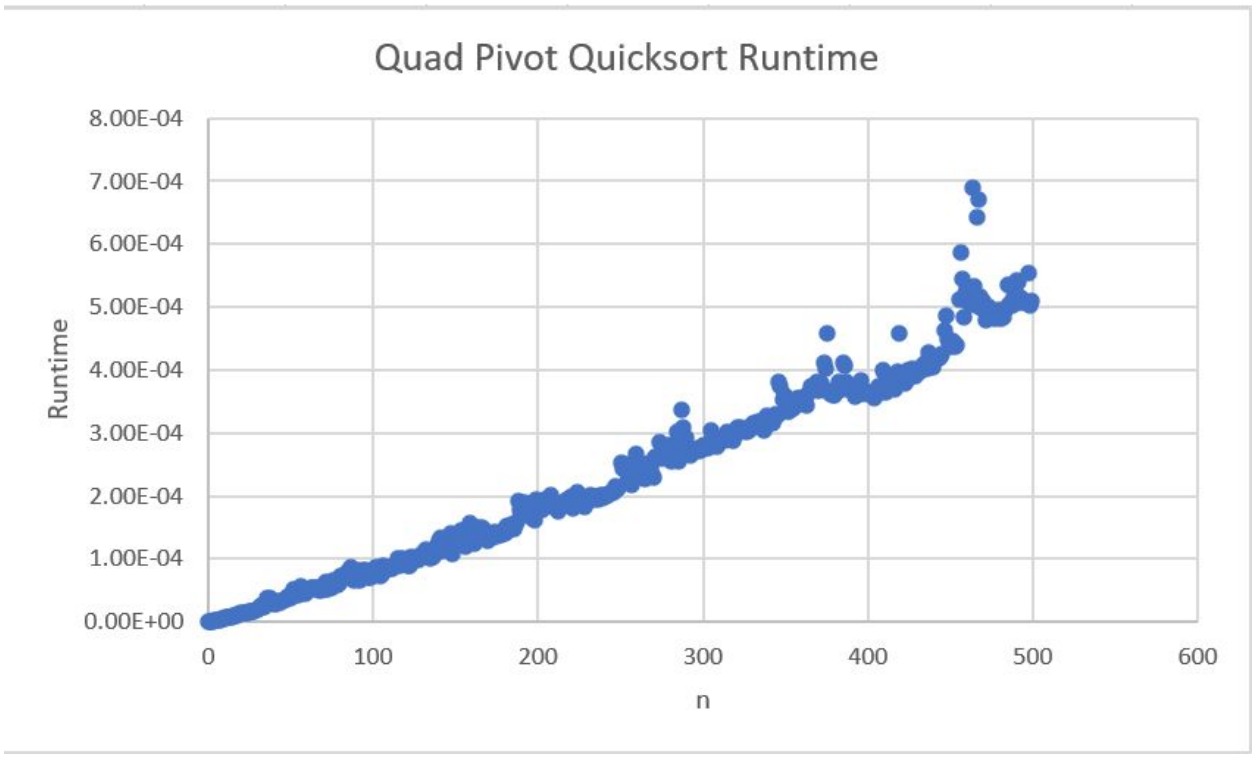
Multi-Pivot:

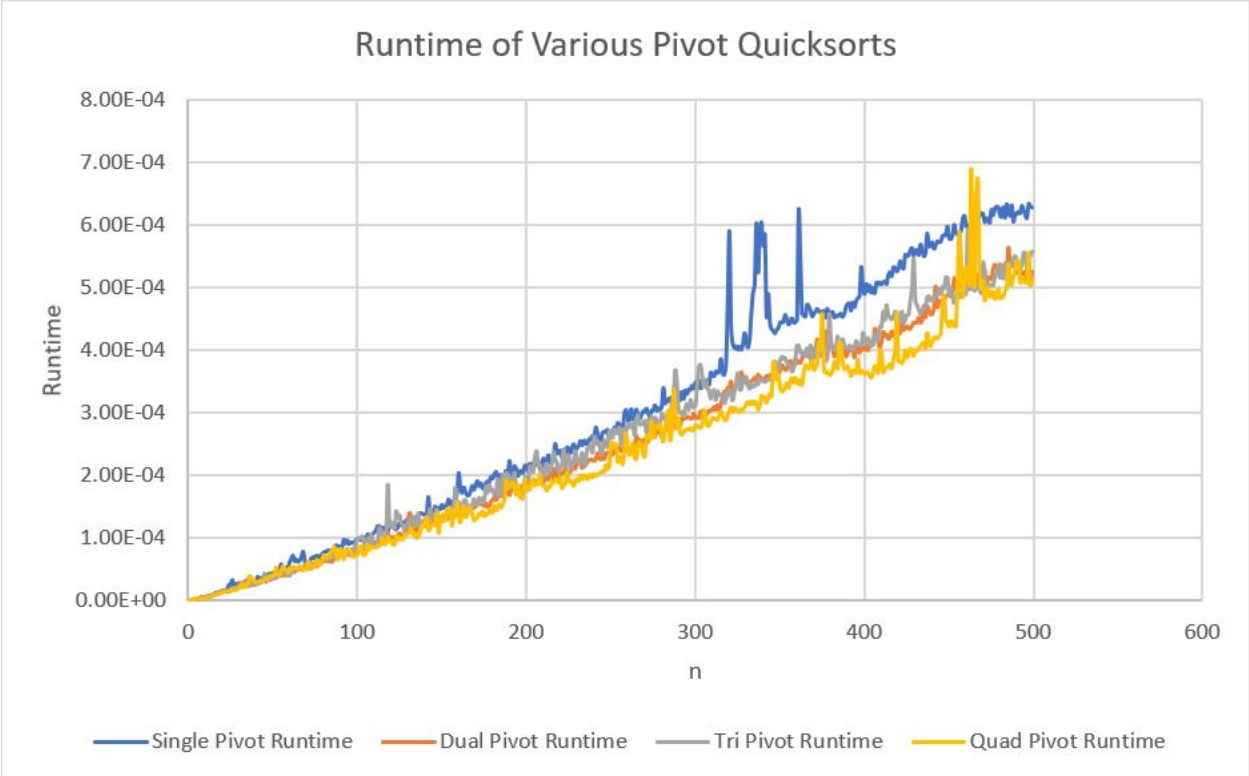
Testing results:






Tri Pivot Quicksort Runtime
7.00E-04
6.00E-04
5.00E-04
4.00E-04
3.00E-04
2.00E-04
1.00E-04
0.00E+00
Runtime
0
100
200
300
400
500
600
n


Quad Pivot Quicksort Runtime
8.00E-04
7.00E-04
6.00E-04
5.00E-04
4.00E-04
3.00E-04
2.00E-04
1.00E-04
0.00E+00
Runtime
0
100
200
300
400
500
600
n



Testing method:

To test the four algorithms, we used random lists of increasing length 'n' and compared the graphs to see which one was fastest. On average, we found quad-pivot quicksort to be the fastest compared to the other quicksort functions that utilized fewer pivots.

Which of the four variants do you recommend?

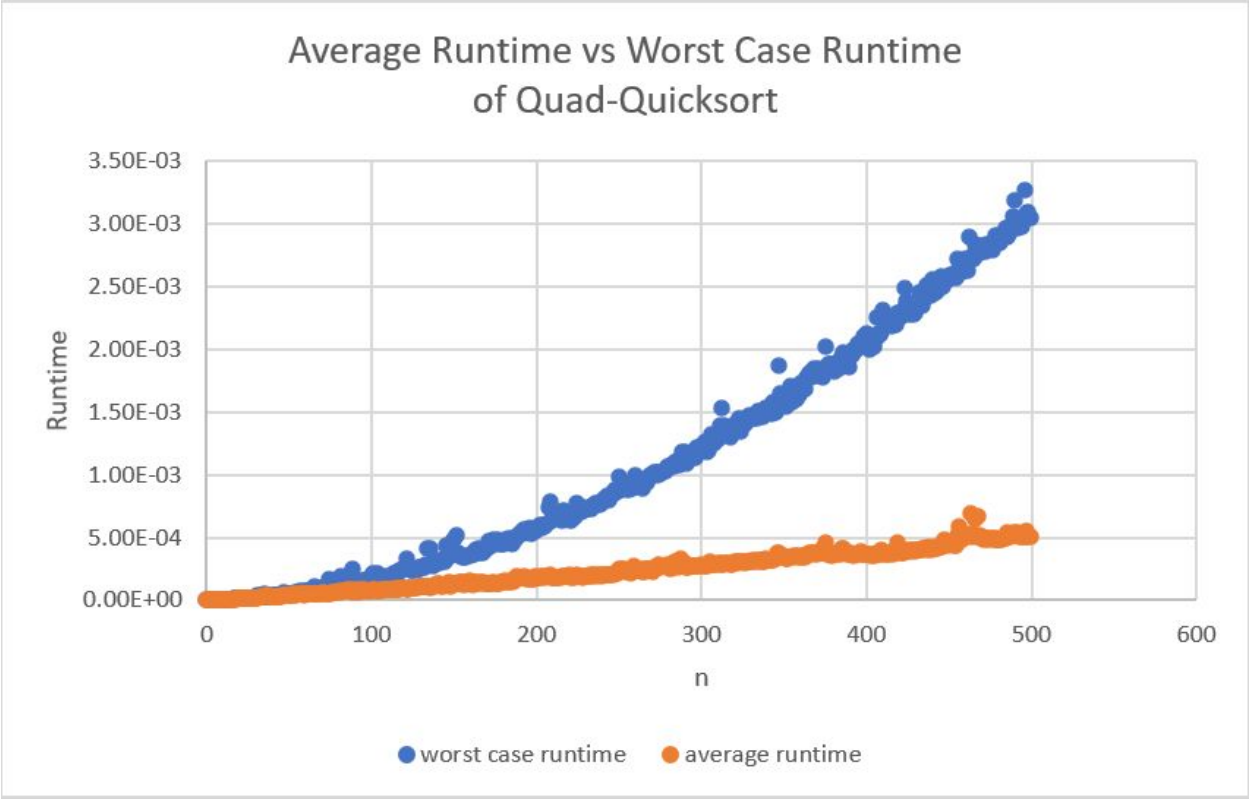
If you want the fastest algorithm, we recommend the quad-pivot quicksort because the size of the lists passed on each recursion call is smaller compared to the size of the lists passed on each recursion call on the other functions with fewer pivots. If the easiest implementation is your goal then we recommend the single-pivot algorithm since it's still very close in speed and it is the easiest function to implement.

Worst-Case Performance:

What is the worst-case performance of quicksort?

The worst-case performance of quicksort is $O(n^2)$. We would expect insertion sort to outperform quicksort when a list is nearly sorted since insertion sort would only need a couple of linear passes through the list to sort it whereas quicksort would still use O(nlogn) passes.

Testing:

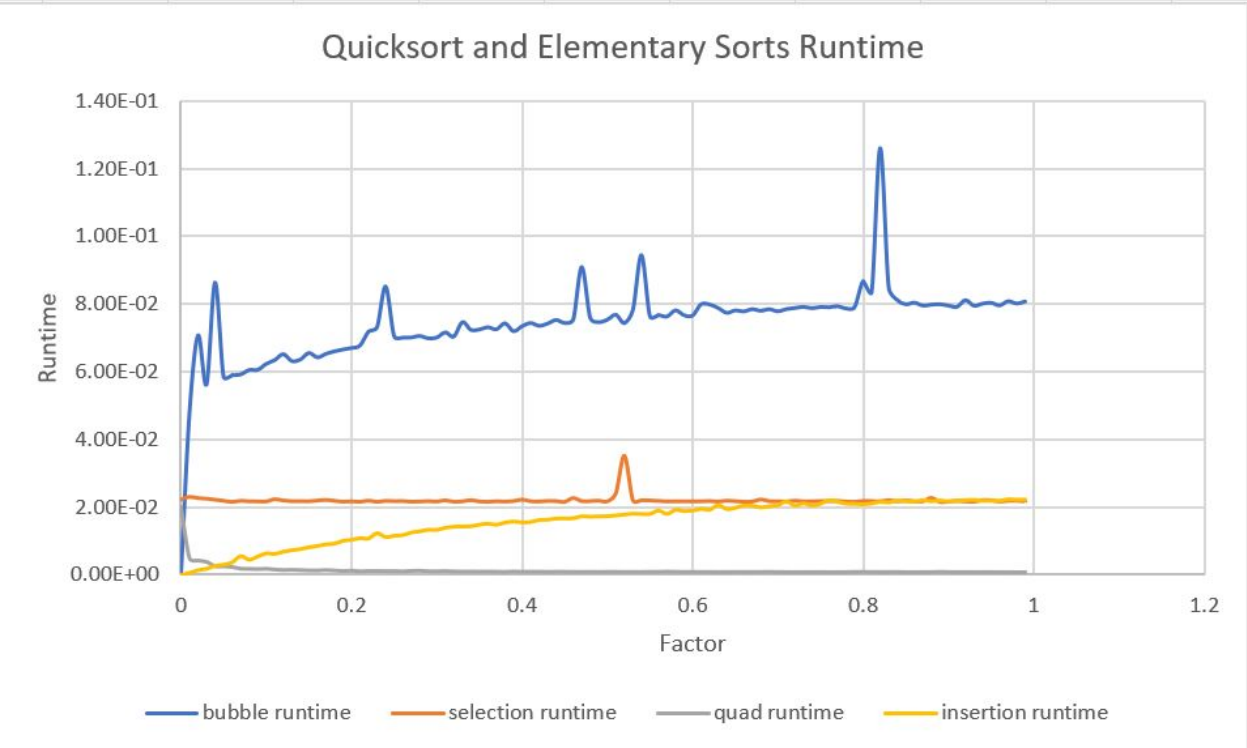


As you can see in the graph above, our quicksort algorithm's average runtime was much faster compared to its worst case. According to the StackOverflow post, the worst case for quicksort occurs when the arrays are already sorted in either ascending or descending order. As a result, we decided to create a function that would sort and reverse the randomly generated list. Furthermore, the average case of quicksort can be tested when the given array is randomly generated. This allows all values in the array to have an equal chance to be selected as the pivots. Knowing this, we used the `create_random_list` function to let our algorithm receive random

lists up to a size of 500 as inputs. As a result, you can see a significant difference in runtime compared to the worst-case.

https://stackoverflow.com/questions/4019528/quick-sort-worst-case

https://stackoverflow.com/questions/59267864/average-case-of-quick-sort

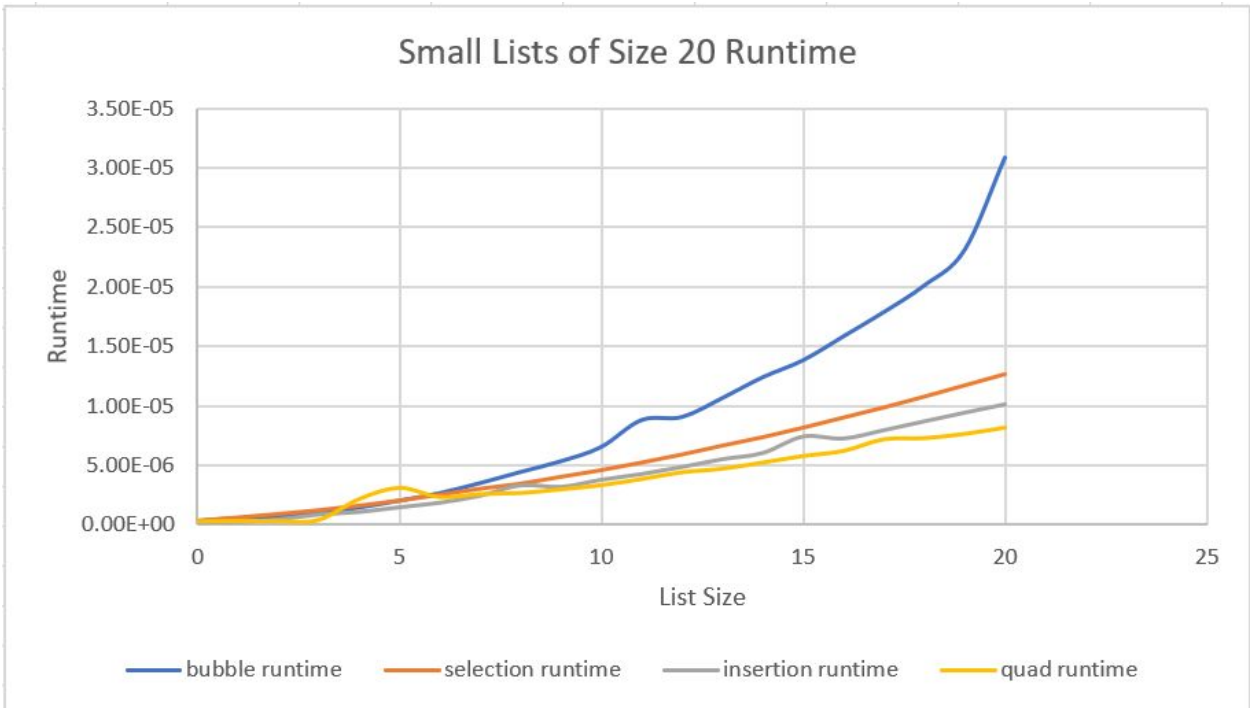


When this factor is low, i.e. the list is close to sorted, what elementary sorts/optimization (if any) would you expect to outperform your quicksort implementation?

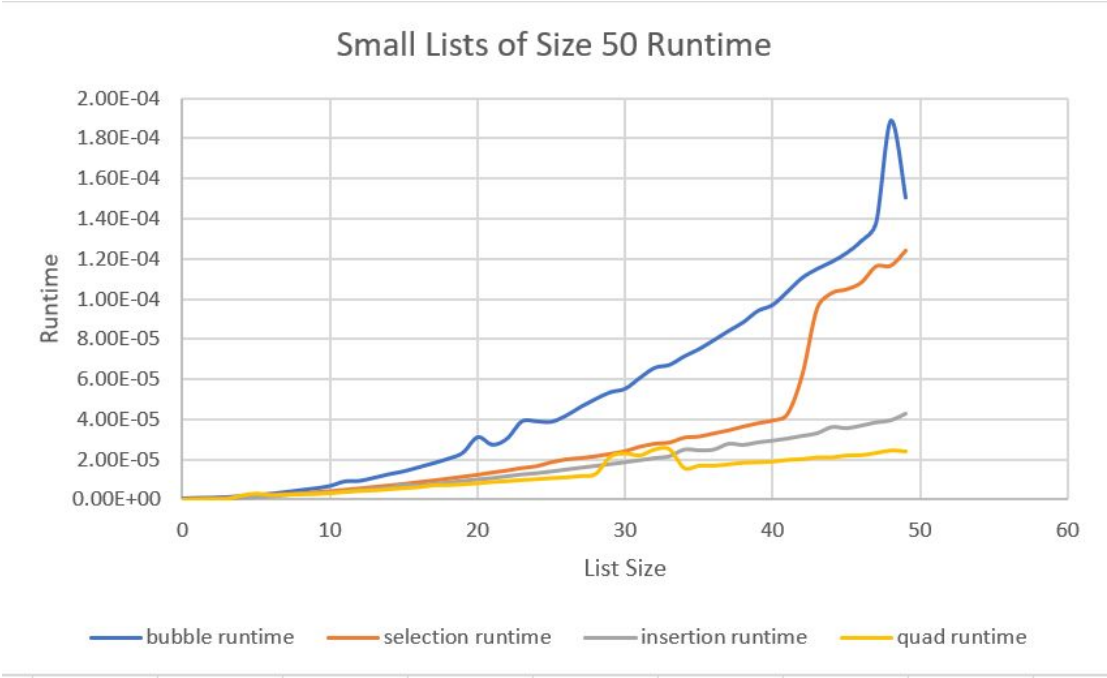
We expect elementary sorts like bubble-sort and insertion-sort to outperform our quicksort implementation since those elementary sorting algorithms have great performance for lists that are small and close to sorted. Furthermore, knowing how bubble sort works, it was expected to be the least performant algorithm for large factors.

As you can see, bubble sort and insertion sort outperform quicksort initially, but bubble sort rapidly increases. Insertion sort is faster than quicksort before a factor of around 0.05 and after this quick sort outperforms every other algorithm by a big margin. As a result, this data makes sense since insertion sort is known to have great performance on lower factors where the lists are more sorted.

Small Lists:



As you can see in the graph above, most of the elementary sorts outperform our quicksort implementation. However, as the list size increases, our quicksort algorithm begins to outperform them.



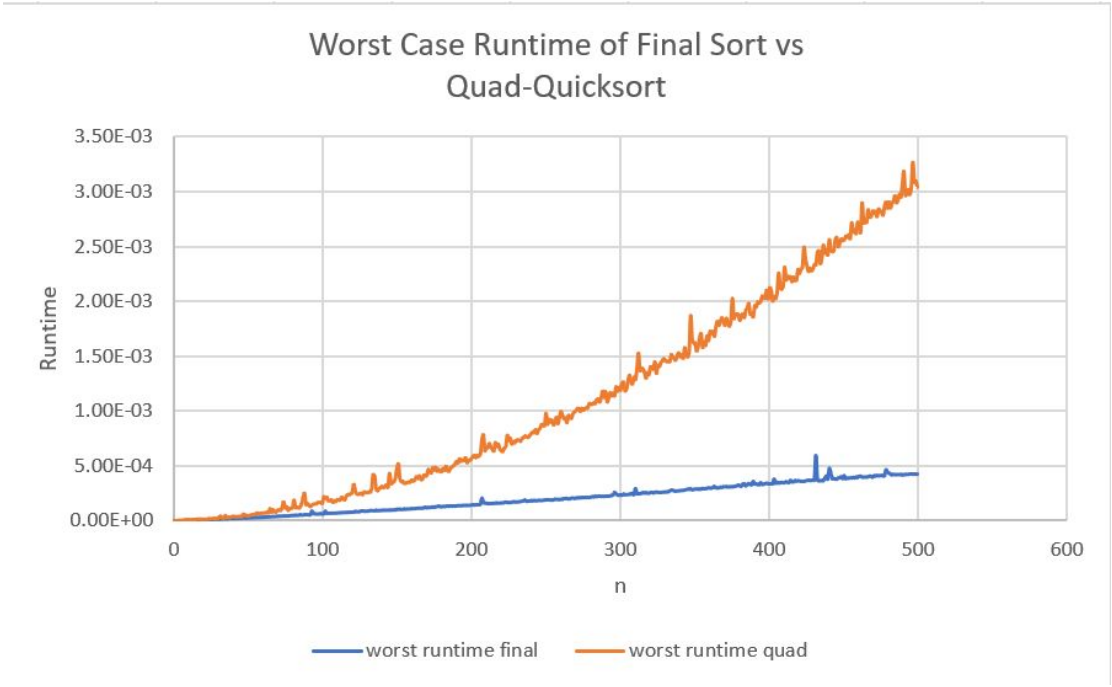
Again, the graph above shows that our quicksort algorithm outperforms the elementary quicksort algorithms. However, we see an unexpected increase in runtime for our quicksort algorithm when the list sizes are around 28-35. We believe this to be an anomaly caused by hardware issues.

final_sort() design decisions:

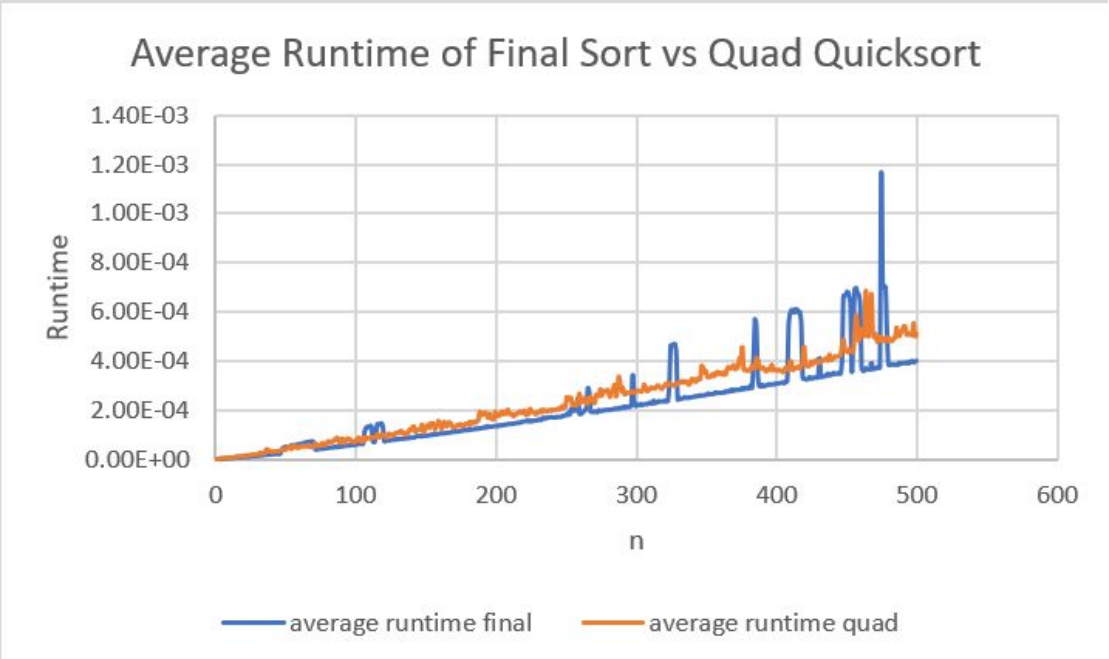
We implemented various changes to our quad-pivot quicksort based on our testing results. The first of these changes was to separate the smallest lists of length 0 and 1 to be returned immediately so that we wouldn't have any unnecessary comparisons slowing it down. From our testing, we concluded that lists of length 3 and lower are sorted faster when insertion sort rather than is used so we again separated those lists and used the insertion sort algorithm on it. Finally, for lists greater than length 3 we used our quad-pivot quicksort since it is the fastest at these lengths of lists. A small change we also made to quad quicksort was to randomize the pivot points in the list based on their index within the list so the chance of the worst-case pivots being chosen is much lower.

What goals were you trying to achieve?

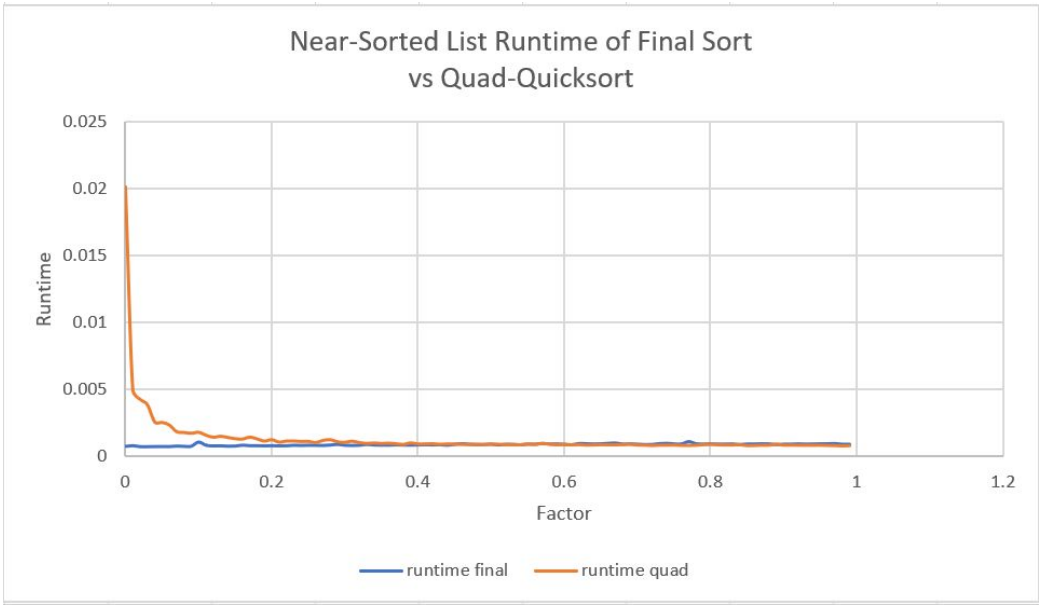
The main goals we were trying to achieve was to minimize the average and worst-case runtime and reduce the possibility of the occurrence of the worst case. Another goal we were trying to achieve was to reduce the runtime of near-sorted lists with a very small factor. We also decided to use the quad-pivot quicksort to reduce the recursion depth of our function which would lead to faster runtime. After creating our final_sort quicksort function we were quite satisfied with the results. We saw a general improvement in the average runtime excluding a few anomalies which could be because of the hardware. We saw a great improvement in the worst-case runtime and the runtime for near-sorted lists. The worst-case runtime for quicksort is O(n2) which is what we got for quad-pivot quicksort, however, our final_sort gave us a much better runtime of O(nlogn). For the near-sorted lists runtime, we were attempting to improve the runtime on lists with a small factor and we were able to do that with our final_sort function. The only thing that our final_sort did not improve on was the runtime on small lists and it had generally the same runtime for list sizes up to 50.



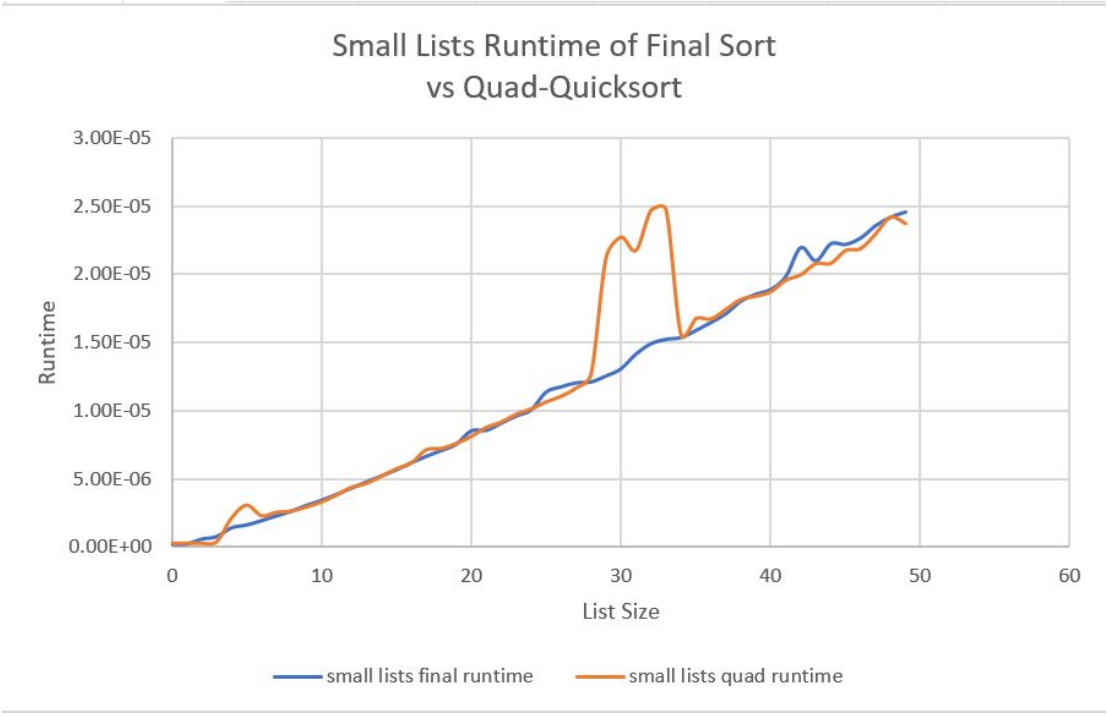
In the graph above, you can see that we've drastically improved the worst-case runtime of our final sorting algorithm due to the change of spacing out the pivots.



The above graph shows the general improvement in average runtime, this again can be attributed to the better selection of pivots by spacing them out.



In this graph, we can see that a big improvement has been made to sorting nearly sorted lists. This is in part due to including insertion sort in our implementation.



In this graph, we see an improvement in stability for smaller lists. The spike at around a list size of 30 for the quad-pivot sort is most likely due to a hardware anomaly and does not reflect the changes in final_sort.

References

Optimizing elementary sorts:

https://www.programiz.com/dsa/bubble-sort#:~:text=Optimized%20Bubble%20Sort&text=The%20code%20can%20be%20optimized,we%20can%20prevent%20further%20iterations.

Information regarding quick-sort:

https://stackoverflow.com/questions/4019528/quick-sort-worst-case

https://stackoverflow.com/questions/59267864/average-case-of-quick-sort