2xc3 Lab Report 3

Team Information

Name: Sai Santhosh Thunga
Section: Lab 02
Student ID: 400343455
MacID: Thungas
Email: thungas@mcmaster.ca

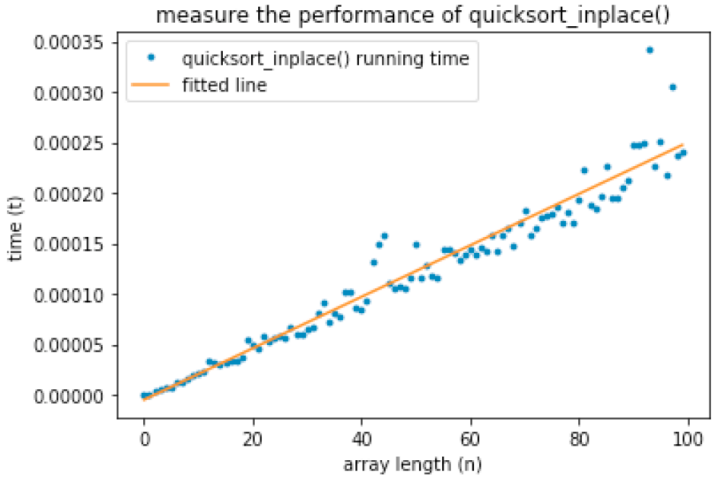
Name: Reshmii Bondili
Section: Lab 02
Student ID:400323168
MacID: Bondilir
Email: bondilir@mcmaster.ca

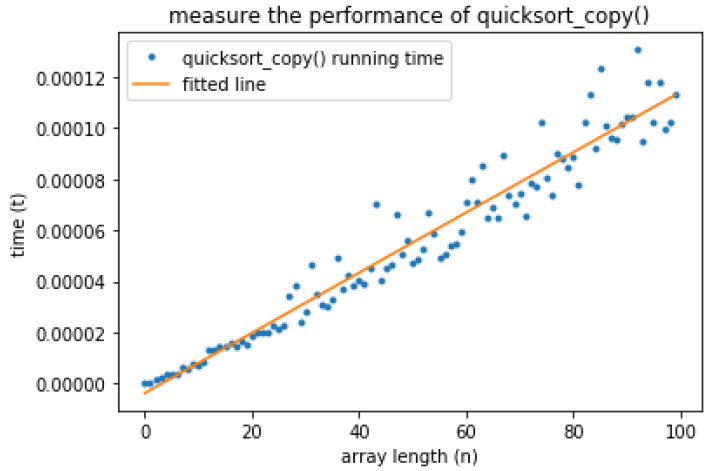
Name: Proyetei Akanda
Section:Lab 02
Student ID:400327972
MacID: Akandap
Email: akandap@mcmaster.ca

In-Place Version [20%]

Before testing the performance of your implementation, discuss any advantages it has over the implementation you are given.
We implemented quicksort using an in-place algorithm. The algorithm given to us uses a simple technique where it uses the input list as well as two other lists to store and input values. In the given implementation, two extra auxiliary arrays are used, left and right array to sort the input list. Our implementation compared to the given implementation uses no extra arrays other than the input array called *alist*. Instead, our method uses a partition function that takes the input array as well as a start and end element to partition the subarray and sorts based on where the pivot is. The “In place” algorithm for quicksort works by sorting in one array itself and not creating any other copies of the list into other sub-arrays. Because of this method, less memory is being used and hence the run-time of the program will be quicker. We predict the run time of the In-place version to be much faster than the code given to us.

Which implementation is better? By how much? Which would you use in practice?
From the graphs below, quicksort in-place algorithm is worse in terms of time complexity compared to the original implementation given to us. We noticed in-place is faster for smaller input array sizes from (10-1000). However, as the size of the list grows the average time increases more for in-place. For larger sizes, the average time for the two is fairly close but the original implementation is faster compared to the in-place version. This is due to the fact that with the in-place implementation we are optimizing our code to work with smaller memory as we perform the swaps for the sorting all within one array, thus this makes us lose efficiency in other areas, specifically time complexity. In practice, we would use the in-place version for lists with very small array input sizes and the original implementation for larger lists. We came to the conclusion that the implementation given to us is much better than the “in-place” algorithm.





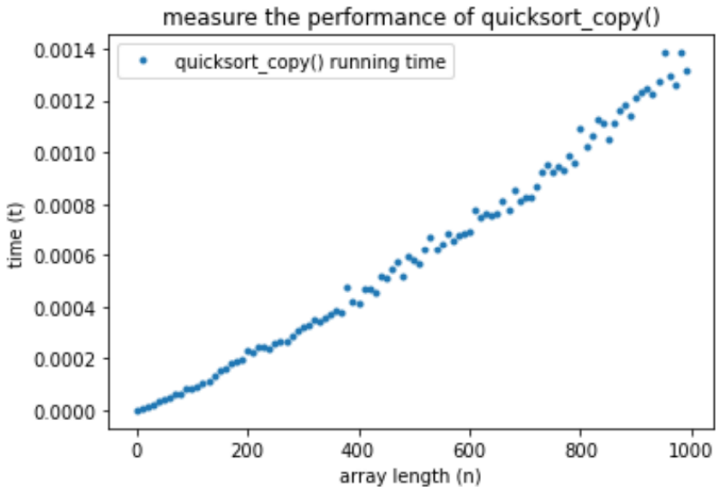
Multi-pivot [40%]

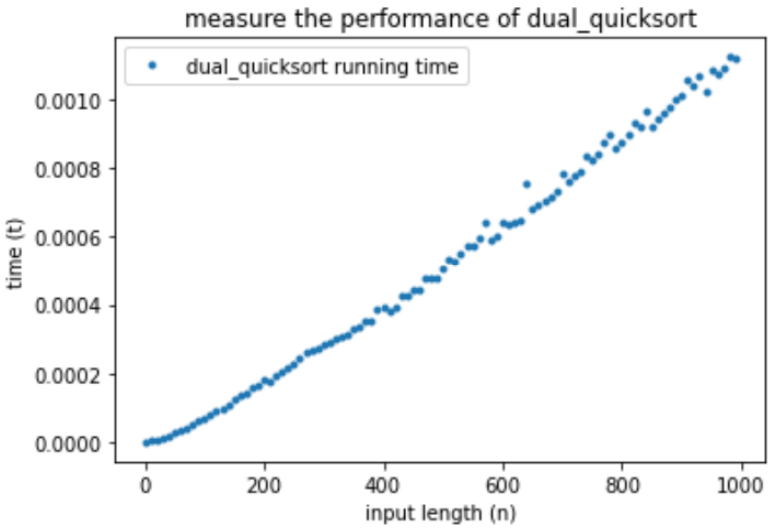
Which of the four variants do you recommend?

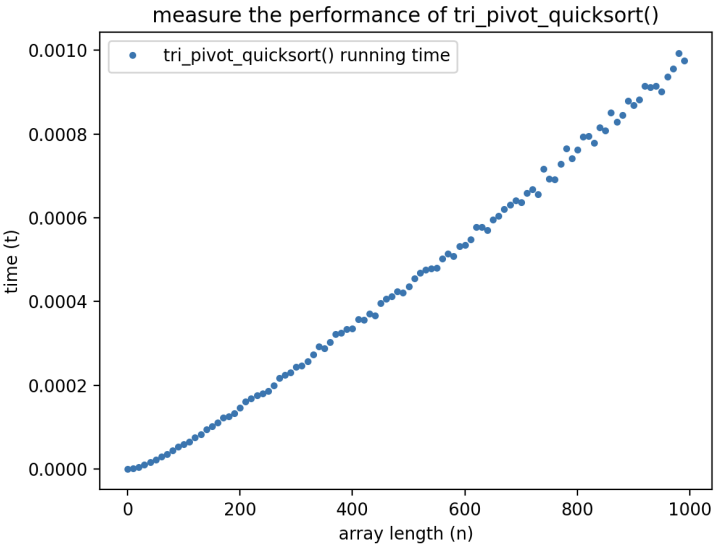
We recommend the quad_pivot_quicksort() algorithm due to it having a greater performance according to time. After testing each of the quicksorts we graphed our finding for array lengths from 0-1000. We ended up with similar-looking graphs but looking closely at the time taken for arrays with 1000 elements we got the following times:

- Single - 0.0014 s
- Dual pivot - 0.0012 s
- Tri pivot - 0.0010 s
- Quad pivot - 0.0010 s

The quicksort with a single pivot is the slowest among the other quicksort algorithms with many pivots. Second, it is clear, in the case of single, dual, triple, and quad pivots), that the more pivots are used in a quicksort algorithm, the faster its performance becomes. Also, the increase of speed by implementing more pivots in a quicksort algorithm tends to diminish slowly. The quad_pivot_quicksort is the fastest one followed by tri, dual, and single pivot quicksort. This is because the quad uses more partitions to partition the arrays.




measure the performance of dual_quicksort
dual_quicksort running time
time (t)
0.0000
0.0002
0.0004
0.0006
0.0008
0.0010
0
200
400
600
800
1000
input length (n)


measure the performance of tri_pivot_quicksort()
tri_pivot_quicksort() running time
time (t)
0.0000
0.0002
0.0004
0.0006
0.0008
0.0010
0
200
400
600
800
1000
array length (n)

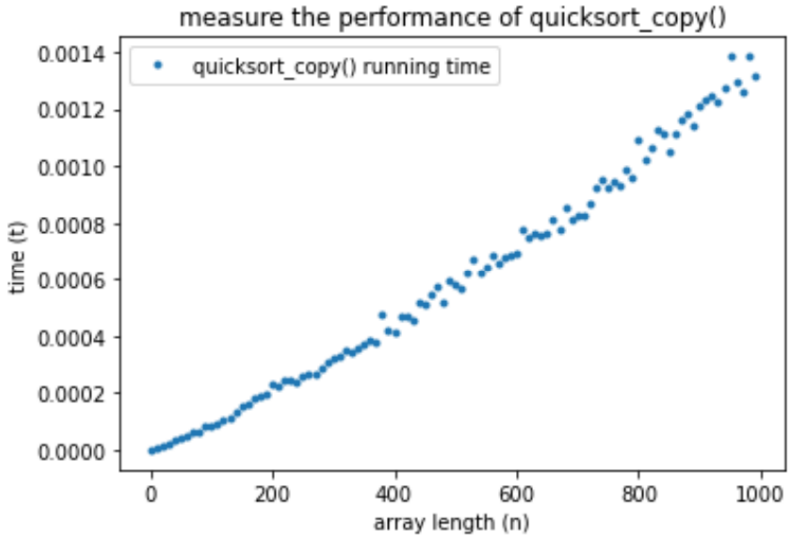

measure the performance of quad_pivot_quicksort()
quad_pivot_quicksort() running time
time (t)
0.0000
0.0002
0.0004
0.0006
0.0008
0.0010
0
200
400
600
800
1000
input length (n)

Worst-case performance [20%]

What is the worst-case performance of quicksort? Run an experiment which graphs the average case performance vs the worst case performance vs n.

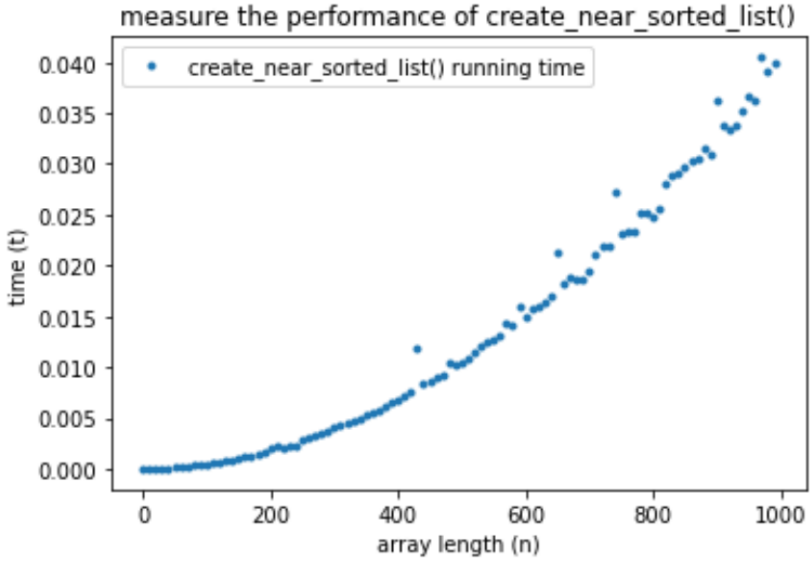
The worst-case performance for quicksort is when the running time is O(n^2) and it occurs when the picked pivot is always the smallest or largest element. This case happens when a randomly chosen pivot is selected in completely sorted order, nearly sorted order or reverse sorted order. The algorithm will split the list down to two extreme cases, one with length of zero and the other with n-1. As a result, this method of partition will cause a **~ n** recursion depth.

Average case vs n



Worst case vs n

As discussed earlier, the worst case performance for quicksort will occur when the list is nearly sorted.

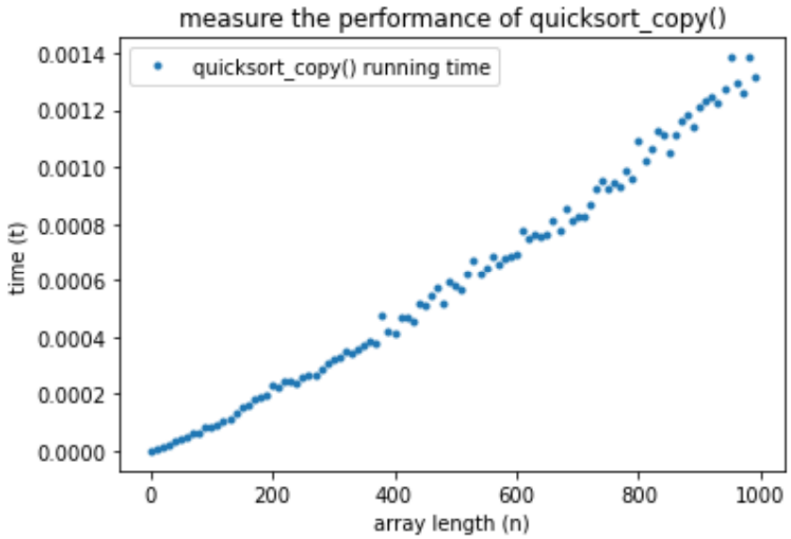


What elementary sorts/optimization (if any) would you expect to outperform your quicksort implementation?

The three elementary sorts are bubble sort, selection sort and insertion sort. When the quicksort factor is low the list is nearly sorted. From the three, we expect insertion sort to outperform quicksort for a nearly sorted list of elements. This is because quicksort has excess and indirect computation time due to it having a large number of recursive calls. Meanwhile, the fundamental method for an insertion sort is a for loop.

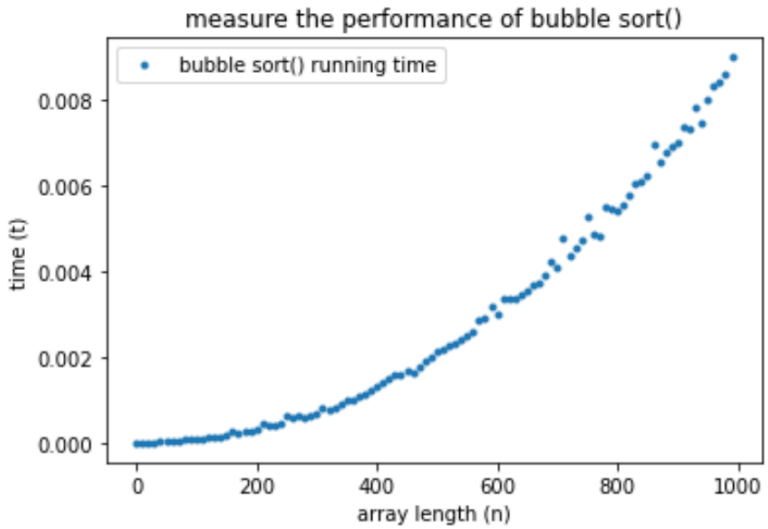
Graph the runtime of your quicksort as well as a few of the best performing elementary sorts vs the near-sorted factor.

Quicksort:

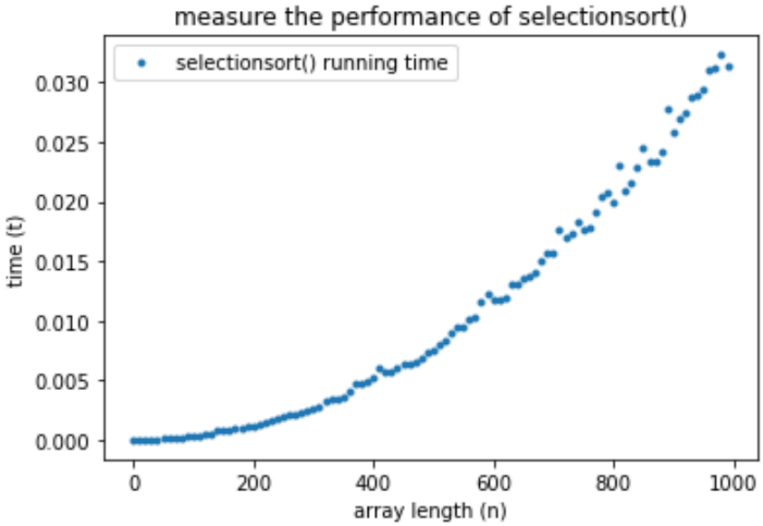


The best performing elementary sorts: (Bubble sort, selection sort, nearly sorted graph)

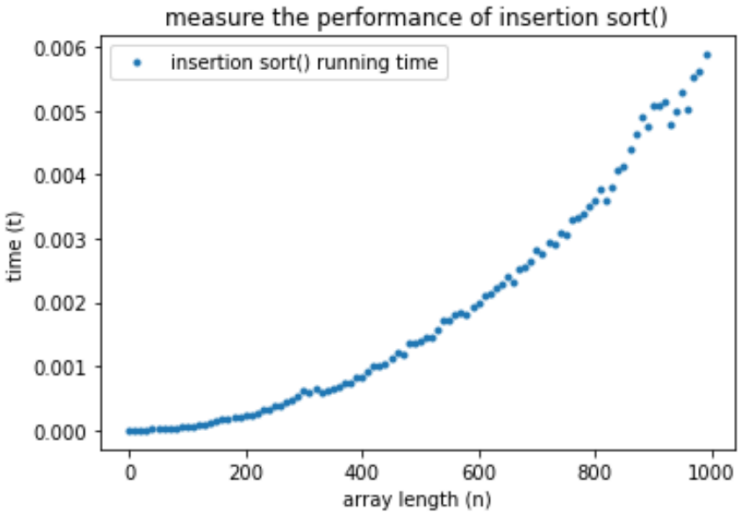
Bubble sort:

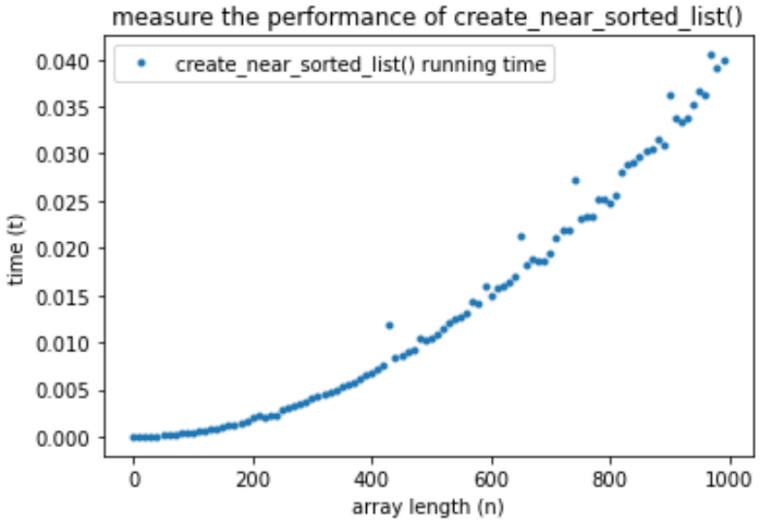


Selection sort:



Insertion sort:



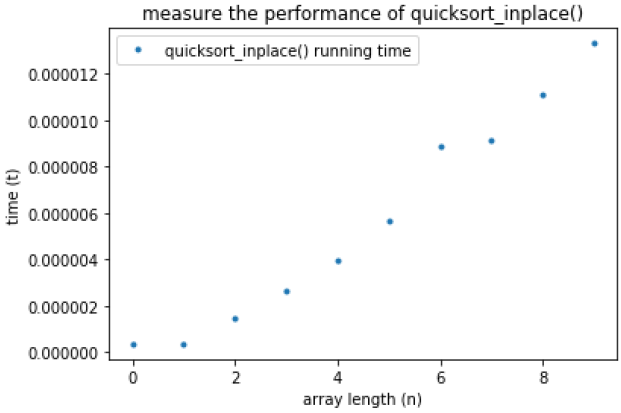


At what value(s) of this factor does quicksort begin to outperform the other sorting algorithms (if it does at all)?

From the experiment[2], we can see that despite increasing the factor to even a 1000, you get identical graphs for the performance of the elementary sorting algorithm and the quicksort algorithm, in which the insertion sort algorithm is far more efficient than quicksort . We came to a conclusion that changing the factor does not have an effect on quicksort outperforming the elementary sorting algorithms.

[2] Refer to Appendix 1 for the *results*

Small lists [20%]



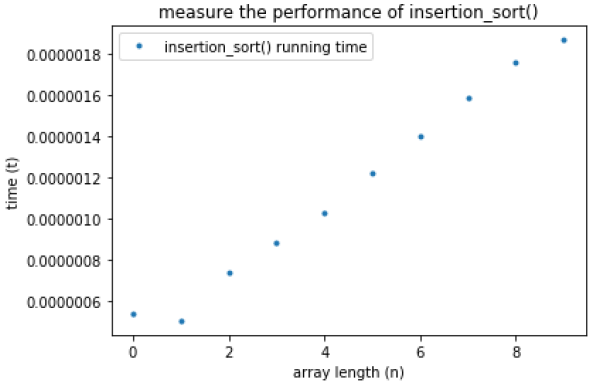


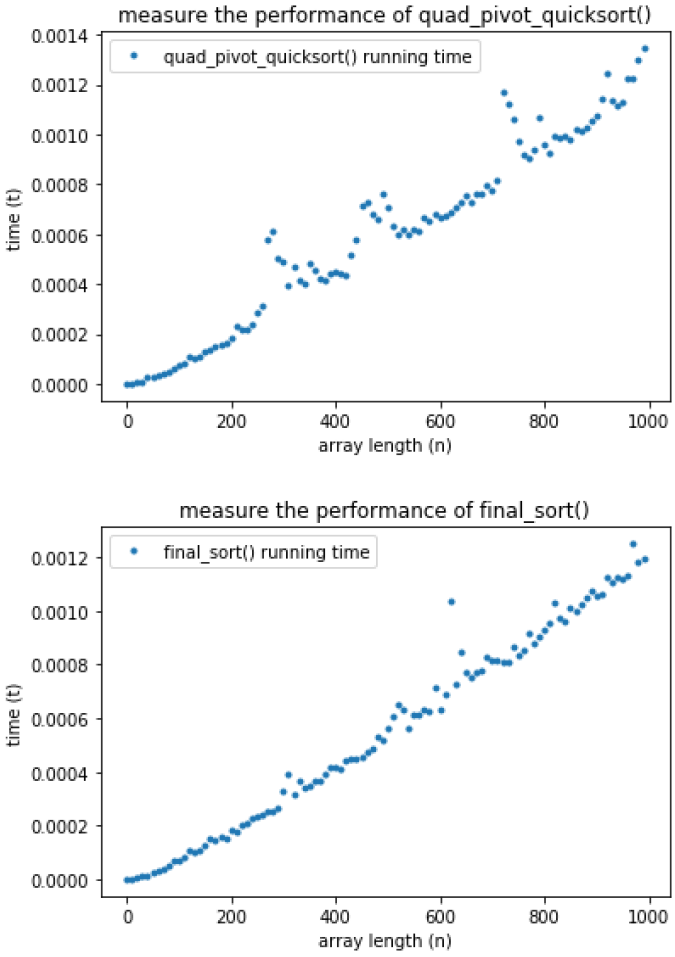
Image 4(a & b) : Measuring performance of quicksort_inplace vs insertion sort

From the above images, insertion sort performs much faster than quicksort while sorting arrays that have a small[3] number of elements. Despite quicksort having O(N*log(N)) and insertion sort having O(N^2) as the average case time complexity, from a mathematical standpoint we know that N^2 is a faster growing function and hence insertion sort would be a slower function in comparison to quicksort, but in practice insertion sort is more efficient as it has few number of comparisons and swaps when compared to quicksort. Moreover, python's recursion (i.e what quicksort uses) is slower than iteration (i.e what insertion sort uses).

So our **goal** was to create an algorithm such that it uses a combination of both quicksort and insertion sort such that it reduces the time taken to run the algorithm and reduces the recursion depth.

We built the final_sort(arr) function that uses the combination of quad_pivot_quicksort( )[4] and insertion_sort( ) functions, in which the algorithm switches to an insertion sort if the length of the sub-list is less than or equal to 10. When comparing final_sort to quad_pivot_quicksort( ), you barely see a difference in performance for lengths of lists that are between 0 to a 100. But as the list size gets large enough, we see a slight performance boost in final_sort. This can be observed in the image below.

[3] The definition of what is small can change from the program to program and from OS to OS. We found that 10 was small enough, in which insertion sort was quicker than quicksort in python.
[4] Most efficient multi-pivot sorting algorithm as seen from the above sections



The reason for final_sort being more efficient is mainly because of switching to use insertion sort on lists that are small in size, this provides faster results and reduces time taken to run sort the large list.

Image 4(c) : Measuring performance of quad_pivot_quicksort() vs insertion_sort()

Appendix 1: Finding the integer factor where quicksort out performs insertion sort for nearly sorted lists