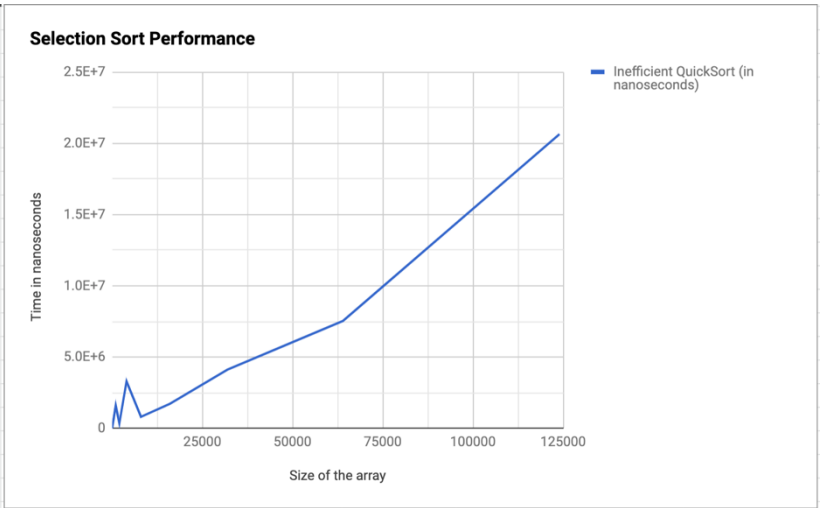
Inefficient implementation of the quicksort algorithm

QuickSort sorts an array by selecting a pivot from the array and splitting the array according to what's larger or smaller than the pivot. This continues creating sub-arrays until only 1 element remains in each sub-array. This following implementation of the algorithm is a basic version of it with no features added to improve efficiency.

Table and Graphical representation of timings found

N	Inefficient QuickSort (in nanoseconds)
8	13330.00
1000	1603846.00
2000	371396.00
4000	3304604.00
8000	815342.00
16000	1727015.00
32000	4129477.00
64000	7539548.00
124000	20634544.00



We can see that this graph is following the growth rate of O(n log(n)).

Efficient implementation of the quicksort algorithm

This implementation of the quicksort algorithm is made to be more efficient. It does this by making use of three different features I added.

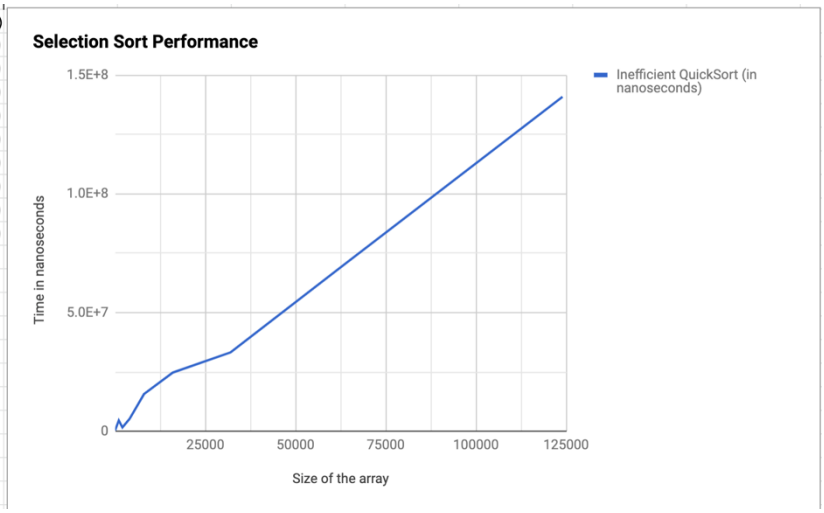
1. Shuffles the inputted array to ensure it is not already sorted so we can avoid the worst case scenario with the QuickSort algorithm. The time complexity of QuickSort, in its worst case, is O(n^2). For instance if we take the last element of an array sized 100, every element below the pivot will be found to be less, creating a sub-list with 99 elements. This continues for every sub-list created causing 100 sub-lists to be created and consequentially put back together.

2. If a sub-array has less than the given cut-off point, then the sub-array will instead be passed to my insertion sort algorithm to be sorted instead. This is because insertion sort, in its best case when dealing with small sized array, has a linear time complexity. This saves time rather than passing the small sub-array to be sorted by the QuickSort algorithm which even in its best case, has a time complexity of O(n log(n)).

3. Use quicksort with median-of-three partitioning. The pivot is selected as the median between the first element, the last element, and the middle element. Choosing a pivot where the value is near the middle or exactly the middle of the elements in the arrays values means the sort will perform better.

Table and Graphical representation of timings

N	Inefficient QuickSort (in nanoseconds)
8	450437.00
1000	4642956.00
2000	1708093.00
4000	5336719.00
8000	15753038.00
16000	24816855.00
32000	33310415.00
64000	70845064.00
124000	140811201.00



As we can see, while the growth rate still follows a O(n log(n)) growth rate, it is much more stable when dealing with smaller sized arrays. In the inefficient version of this algorithm, smaller sized arrays resulted in more spikes with the time the algorithm takes to sort the array. By looking at the times of the smaller sized arrays, it's clear that a growth rate of $O(n^2)$ begun to form.