Arjun Aggarwal
2024AIB1289
Department : Artificial Intelligence and Data Engineering
CS205
Assignment 1 : Sorting Algorithms

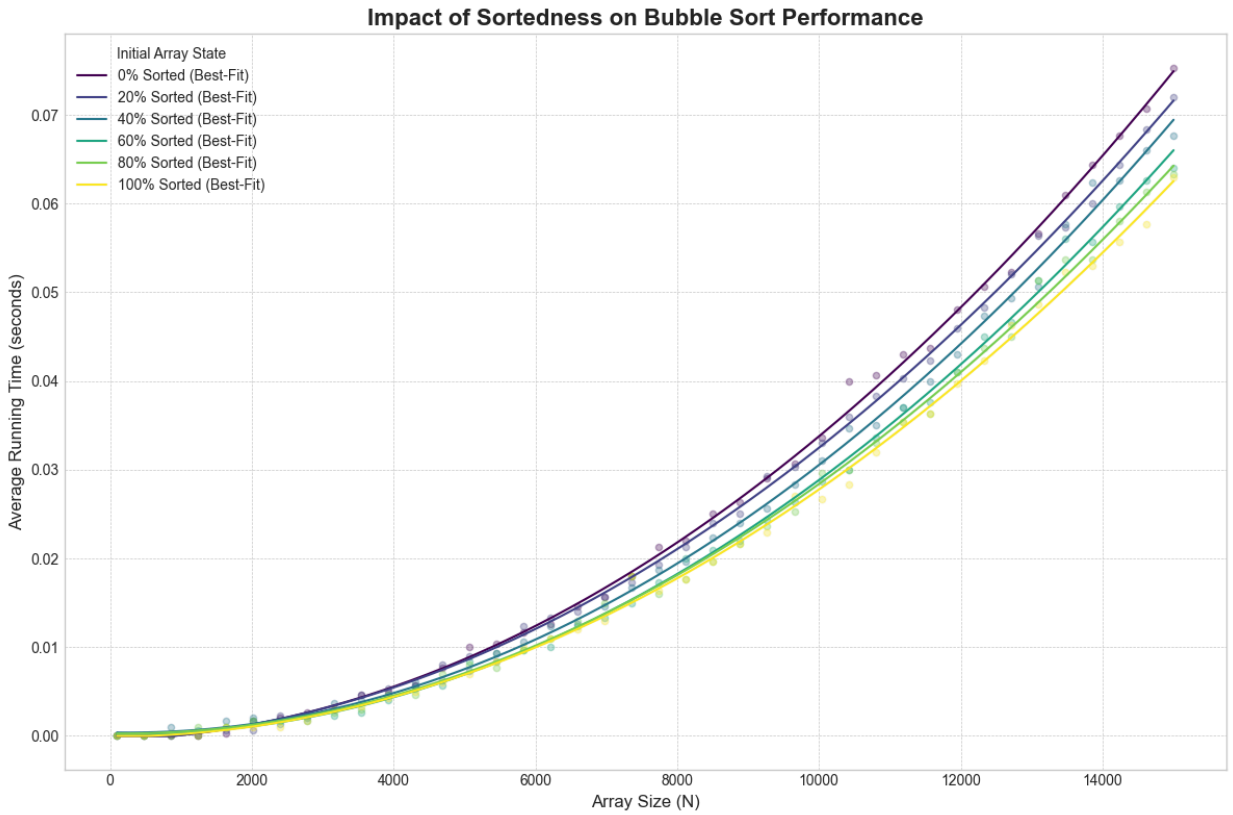
System Specifications
Processor : AMD Ryzen 7 8845HS w/ Radeon 780M Graphics (3.80 GHz)
RAM : 16.0 GB (13.8 GB usable)
System Type : 64-bit operating system, x64-based processor

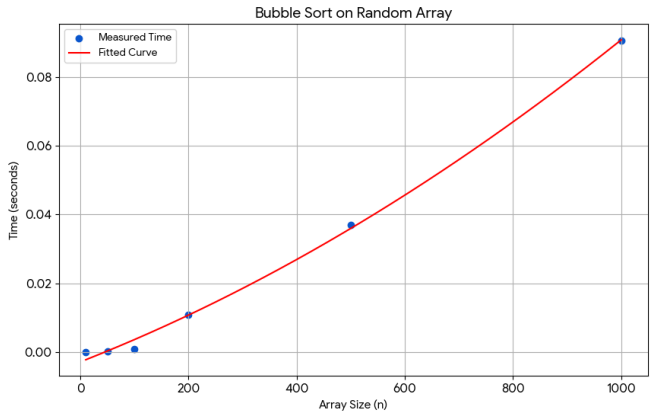
**Points to be noted*
The plots displayed have been generated using matplotlib and scipy. Rather than mapping discrete points, a quadratic curve has been fitted to the data to obtain a continuous distribution.
Each datapoint is averaged over three calculations.
The code in C language is compiled with O2 optimization.

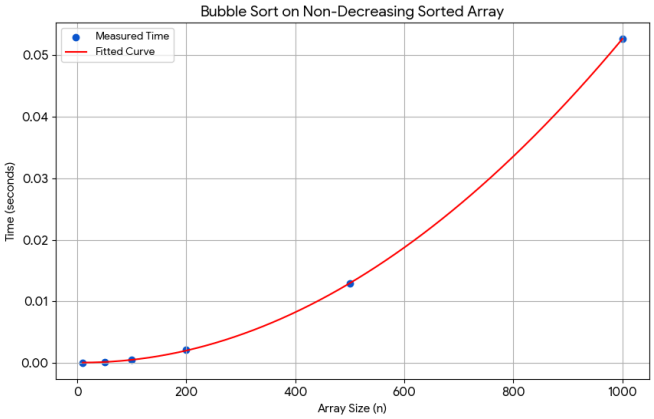
In this section, we will be exploring the five sorting algorithms, analyzing their best, average and worst case time complexity. Plots will be accompanied, if required.

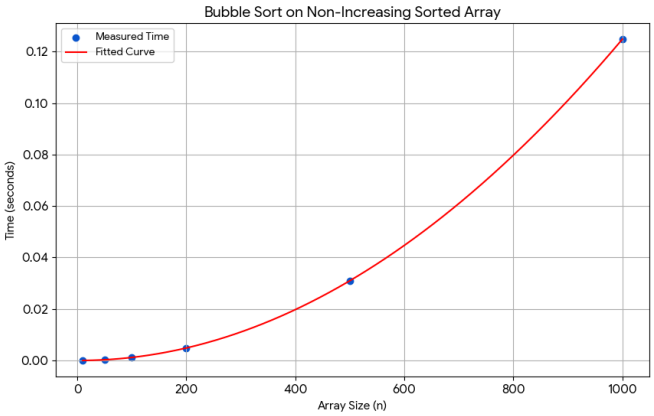
1) ***Bubble Sort***

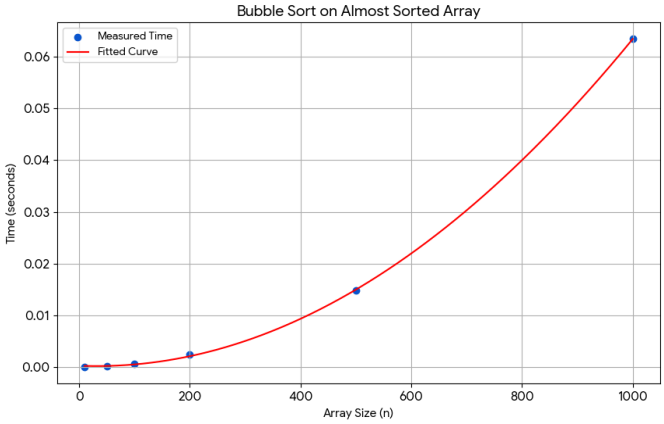
In this sorting algorithm, the number of operations are approximately of the order of $O(n^2)$, n being the array size. Best, average and worst case complexity is of order $O(n^2)$.
The number of operations does not vary with respect to the sortedness level of the array as still the outer loop runs for (n-1) times, and the inside loop compares elements in order of O(n).
So, the program terminates only after making all the comparisons, even if the array is fully sorted.
The plot below demonstrates how little does sortedness affect the time performance of bubble sort algorithm. Though there is clearly descent in $O(n^2)$-type curve as sortedness increases, it is mostly due to decrease in swaps as sortedness increases.







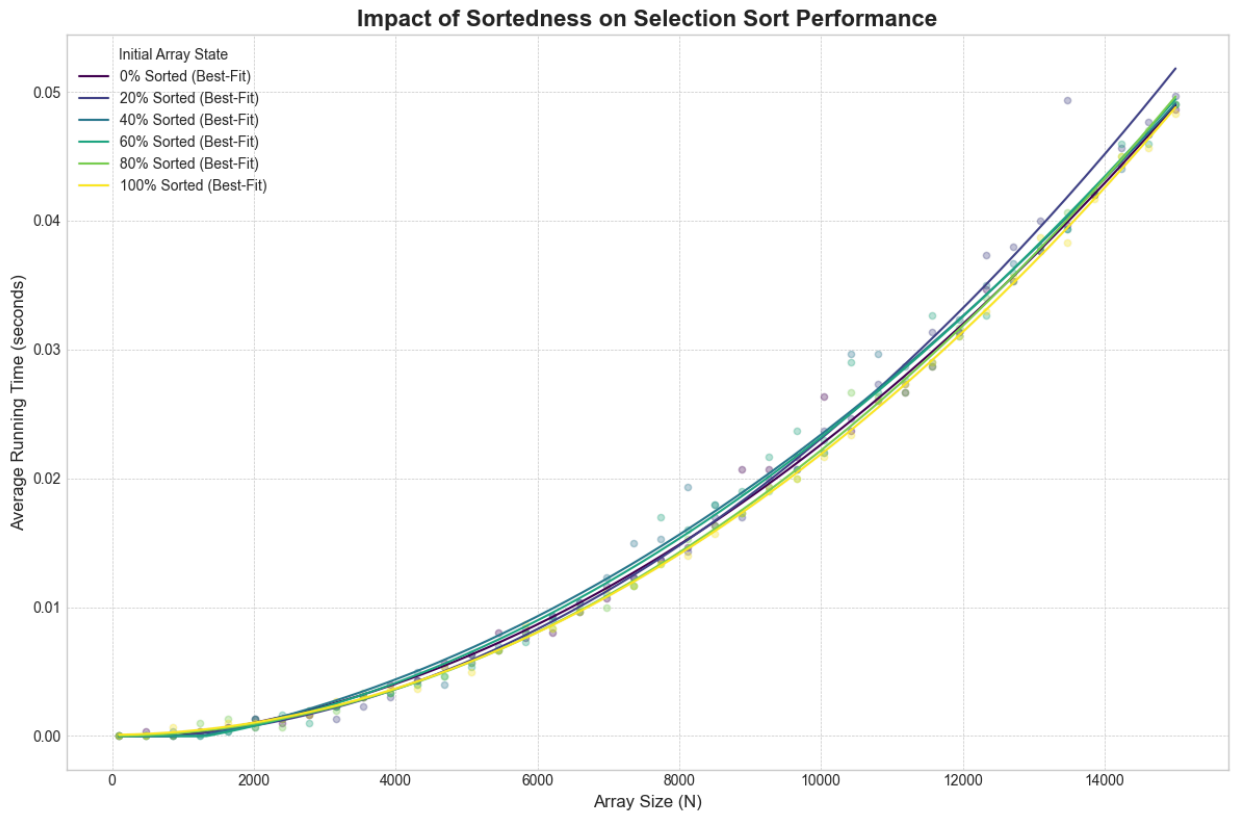


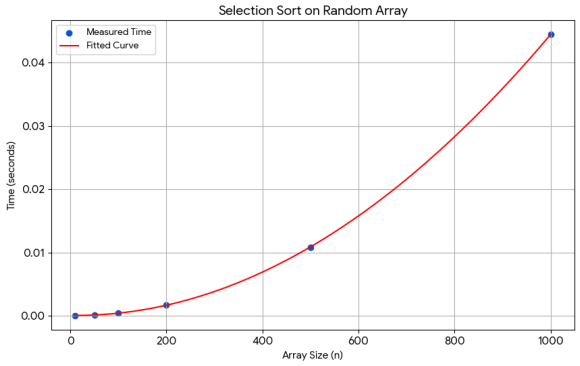


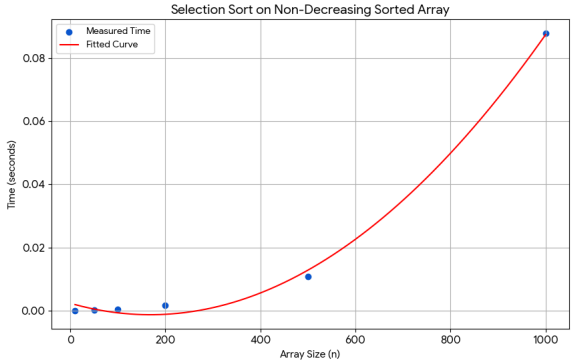
2) *Selection Sort*

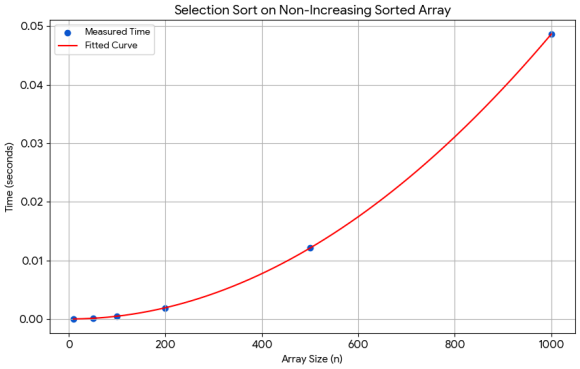
Similar to bubble sort, selection sort is also independent of the sortedness of the array since the outer loop runs for (n-1) times and the inner loop searches for the minimum from the remaining array in O(n), there is a maximum one swap per iteration of the outer loop.

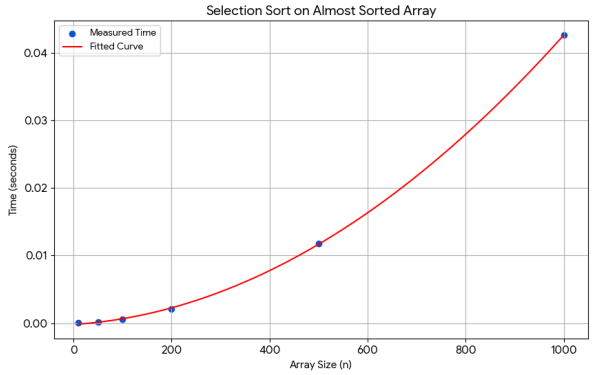
Unlike bubble sort, the sortedness even less impacts time performance as swaps are also very less. So, the best, average and worst case complexity is the order of $O(n^2)$.


Impact of Sortedness on Selection Sort Performance
Initial Array State
0% Sorted (Best-Fit)
20% Sorted (Best-Fit)
40% Sorted (Best-Fit)
60% Sorted (Best-Fit)
80% Sorted (Best-Fit)
100% Sorted (Best-Fit)
Average Running Time (seconds)
Array Size (N)


Selection Sort on Random Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)


Selection Sort on Non-Decreasing Sorted Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)


Selection Sort on Non-Increasing Sorted Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)


Selection Sort on Almost Sorted Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)

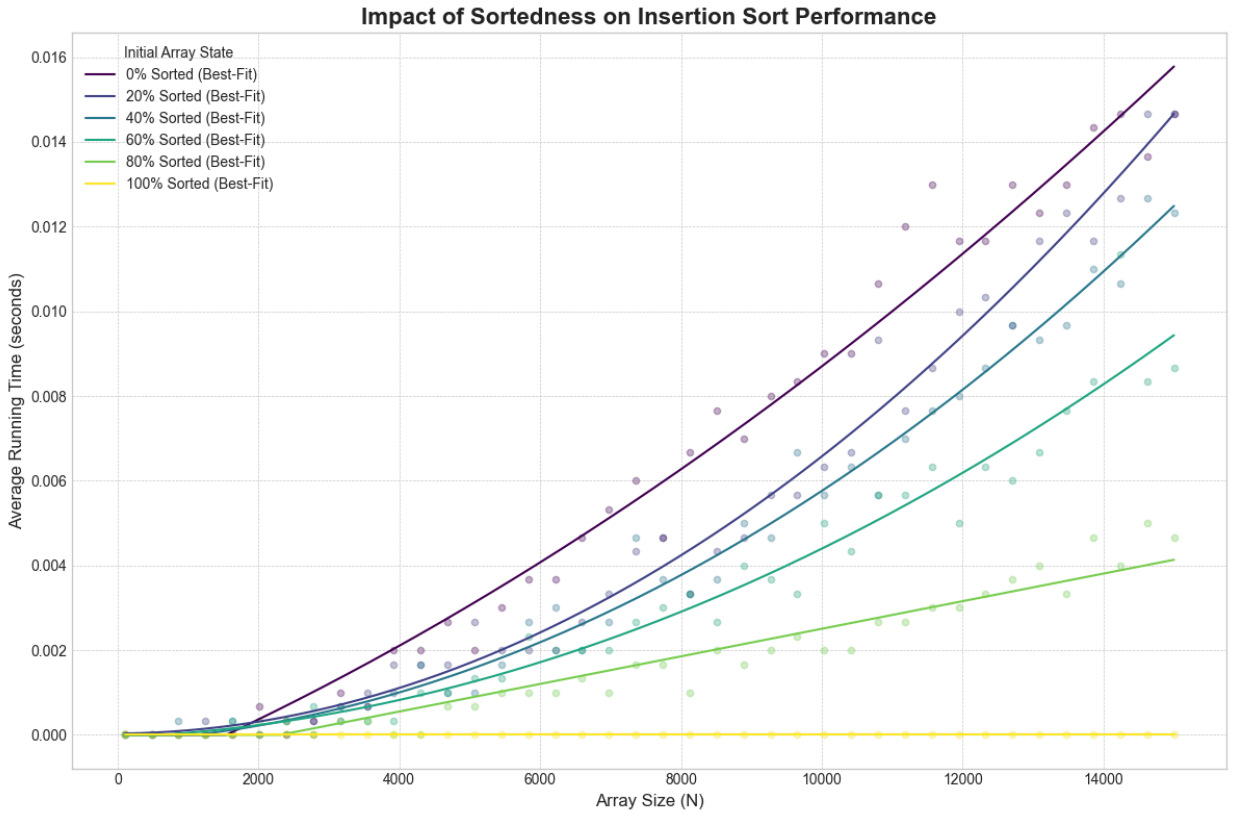
3) *Insertion Sort*

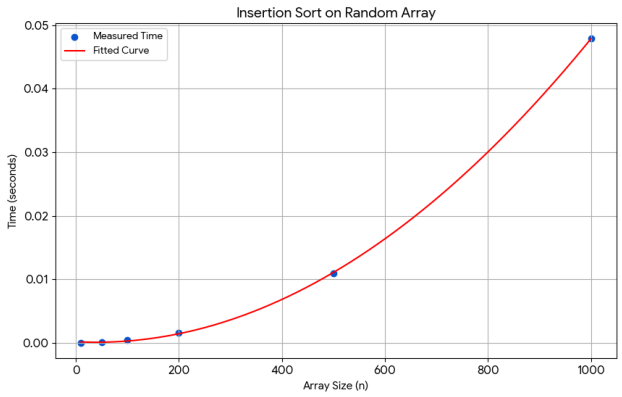
Though, considering average time complexity, the algorithm's running time is pretty similar to bubble sort and selection sort, i.e. $O(n^2)$, there are fundamental differences in this sorting algorithm, where sortedness of the array significantly influences the running time of the algorithm.

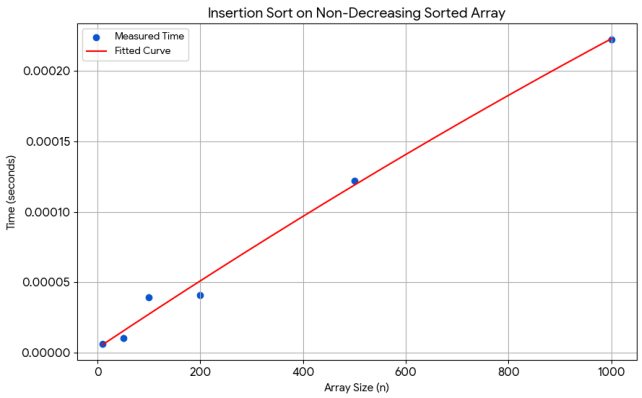
The best case running time complexity of the algorithm is of order O(n). This happens when the array is fully sorted. The inside 'while' loop always compares two elements and continues to the next iteration of the outer loop which runs O(n) times.

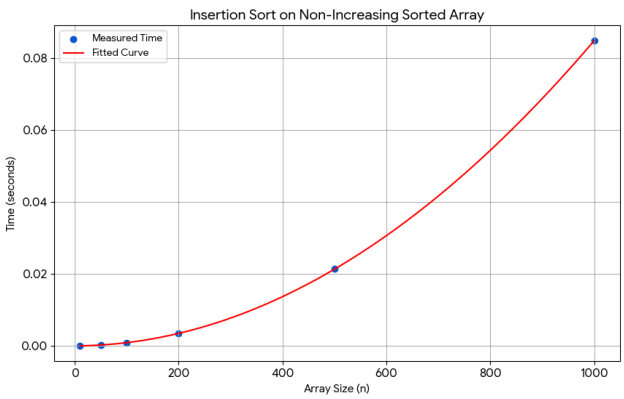
The worst case time complexity is $O(n^2)$ since while loop could have to perform O(n) operations.

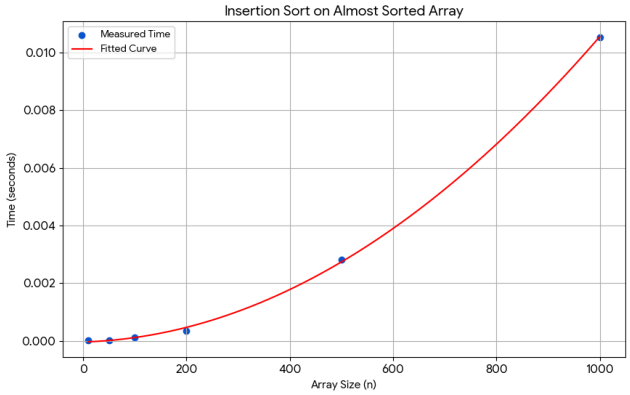
The graph demonstrates how linear curve for full sorted array and transition to $O(n^2)$ complexity as sortedness decreases.











The next two listed algorithms are fundamentally different from the above three. This is because both the algorithms use recursion. At each step of recursion, basically at a certain depth of the recursion tree, both the algorithms perform O(n) operations. The depth approximately of the recursion tree is approximately $\log_2 n$ as the array is divided into two parts at each level. Equality of these segments depends on the specific algorithms. Hence the average time complexities of both type of algorithms is $O(n*\log_2 n)$. Worst case and best case complexities will be discussed in specific sections.

4) *Merge Sort*

In this algorithm, given array is divided into two almost equal parts. Basically a mid = (l+r)/2 is chosen and merge sort on left and right subarray is called. This division occurs till only one element is left. Basically, first the array is just divided till it reaches the leaves of the recursion tree.

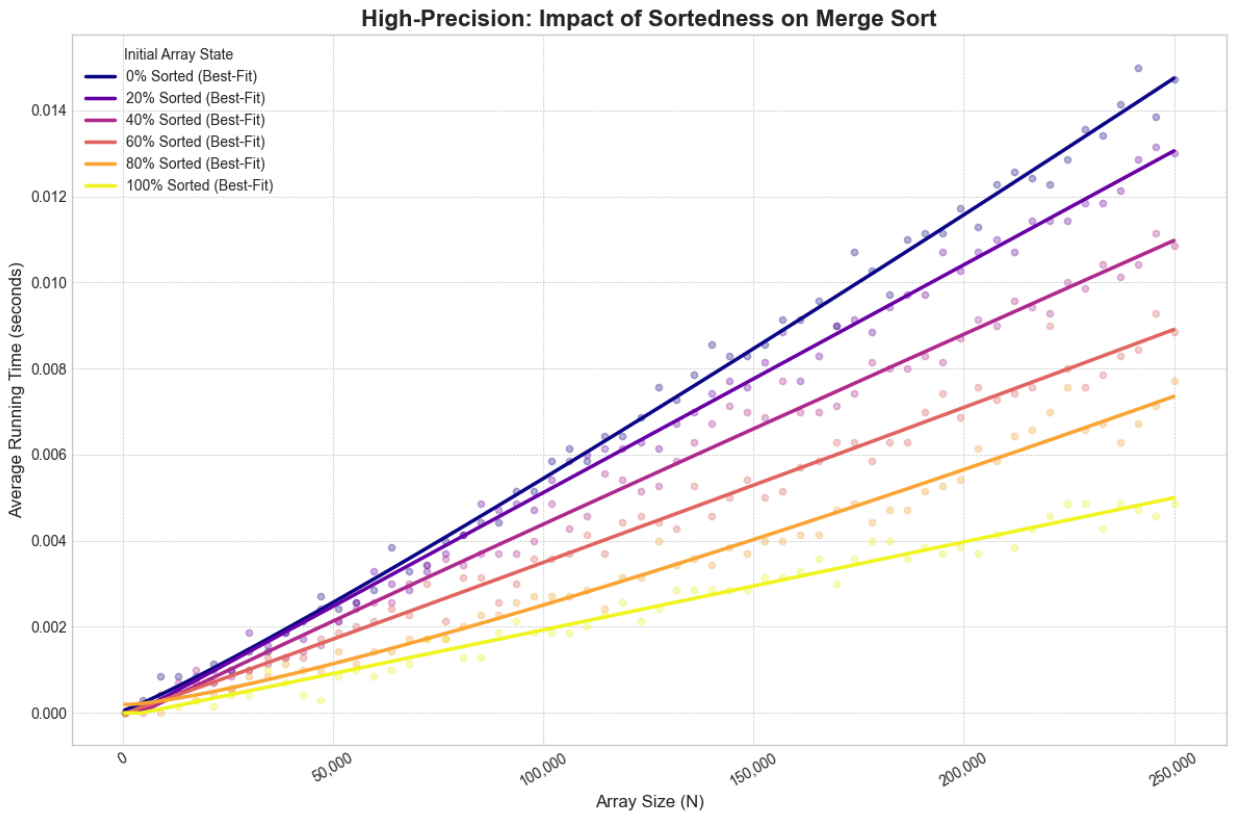
Once the algorithm touches the leaves of the recursion tree, at each level merge sort that was called on the left and right subarray returns the sorted array. So, then we combine two sorted arrays into one sorted array by using a two pointer technique.

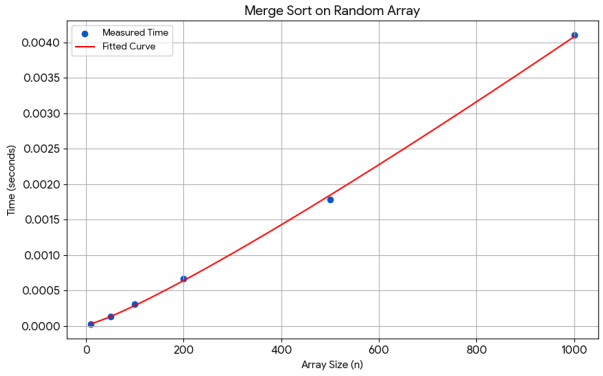
*At each depth of the recursion tree, this operation take O(2*n) time. O(n) for creating a sorted array in a shallow copy and then O(n) for copying them into the original array.*

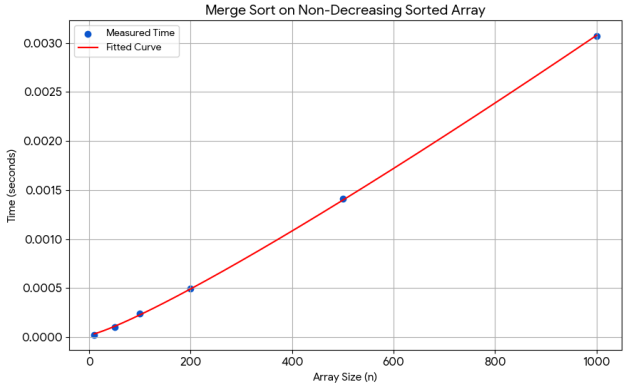
As the algorithm is not dependent on the any certain number the best case, average case and worst case time complexities are equal i.e. $O(n*\log_2 n)$

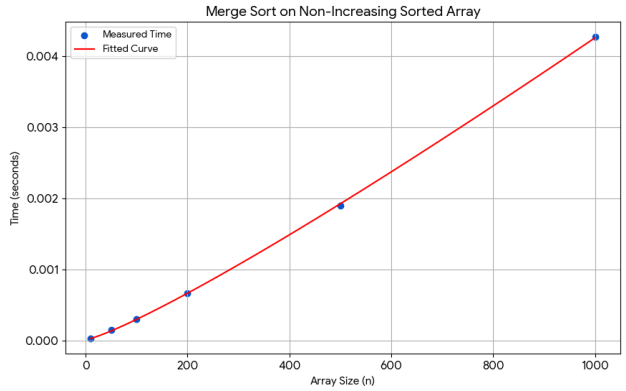
The given graph compares running times of the merge sort algorithm on the arrays of different levels of sortedness. All curves obtain a $O(n*\log_2 n)$ type nature. The subtle difference in curvatures might be due to not switching the pointer type again and again if sortedness increases. Basically $\log_2 n$ depth is traversed by all, the subtle differences might be due to hardware level optimizations.

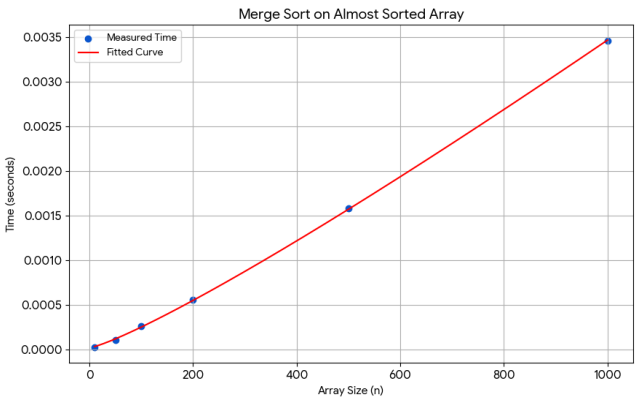
Space Complexity : It requires shallow copy each time combining two sorted subarrays that is O(n) and recursion stack memory.


High-Precision: Impact of Sortedness on Merge Sort
Initial Array State
0% Sorted (Best-Fit)
20% Sorted (Best-Fit)
40% Sorted (Best-Fit)
60% Sorted (Best-Fit)
80% Sorted (Best-Fit)
100% Sorted (Best-Fit)
Average Running Time (seconds)
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
50,000
100,000
150,000
200,000
250,000
Array Size (N)


Merge Sort on Random Array
Measured Time
Fitted Curve
Time (seconds)
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
200
400
600
800
1000
Array Size (n)


Merge Sort on Non-Decreasing Sorted Array
Measured Time
Fitted Curve
Time (seconds)
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
200
400
600
800
1000
Array Size (n)


Merge Sort on Non-Increasing Sorted Array
Measured Time
Fitted Curve
Time (seconds)
0.004
0.003
0.002
0.001
0.000
0
200
400
600
800
1000
Array Size (n)


Merge Sort on Almost Sorted Array
Measured Time
Fitted Curve
Time (seconds)
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
200
400
600
800
1000
Array Size (n)

5) ***Quick Sort***

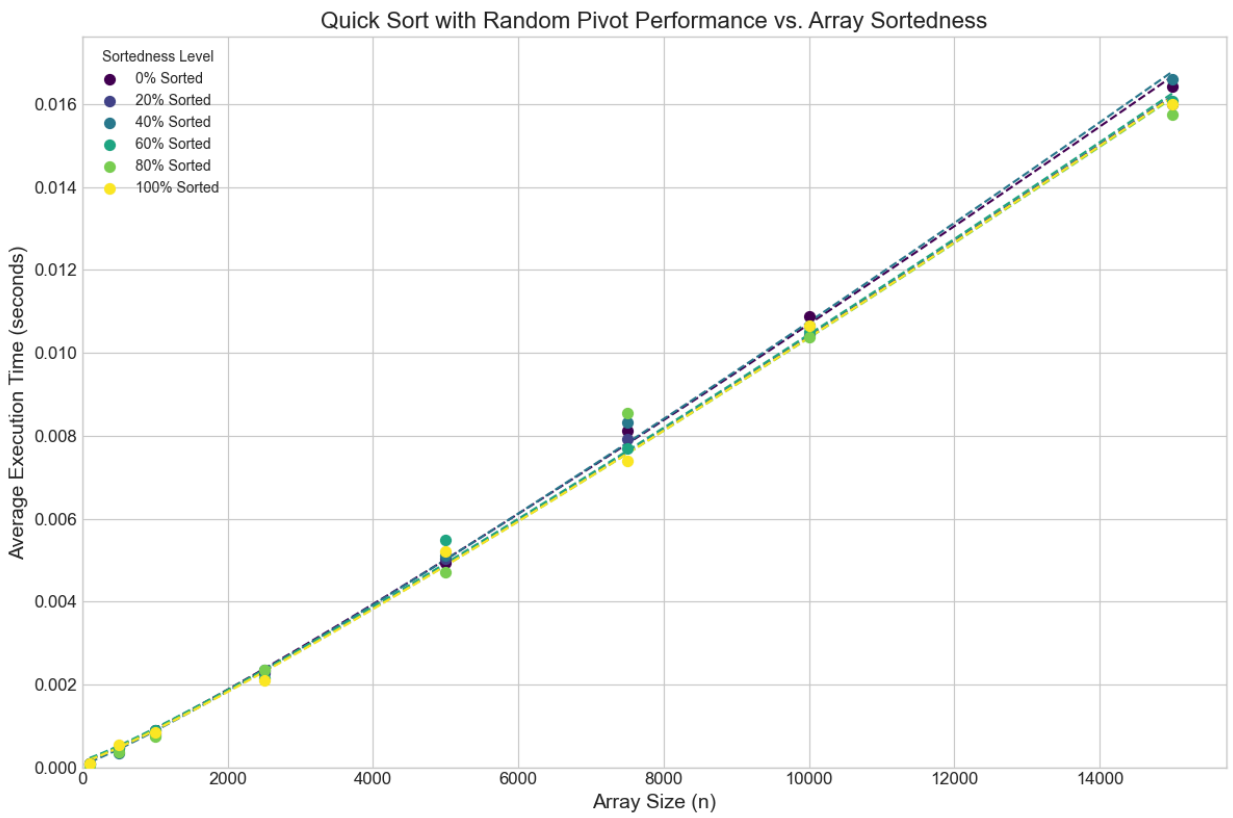
The algorithmic complexity and the recursion equation of this algorithm has high resemblance with merge sort with few subtle differences. This algorithm is highly dependent on the chosen pivot. This algorithm works by first partitioning the array in place by shifting all smaller elements to the pivot to its left side and greater, to the right side. Once partitioned, a quick sort function is called on the left and right parts to the pivot.

Like merge sort, at each depth level of the recursion tree, O(n) takes to partition the array. But the depth of the tree is dependent on the chosen pivot and the sortedness of the array. In the case of random pivot, any sortedness of the array gives consistent results as on average half the elements are smaller and half, greater than the pivot. So the average depth of the recursion tree is $log_2 n$. So time complexity is $O(n*log_2 n)$. The best case time complexity arises when the chosen pivot partitions the array into two almost equal halves which is similar to merge sort. Hence the average time complexity and best case time complexity is $O(n*log_2 n)$. On the other hand, if the random pivot chosen lies on the extremes,the division of the array is highly unequal and the depth of the tree is about O(n). Hence, the worst case time complexity is $O(n^2)$. But this case is highly unfavourable when choosing a random pivot.

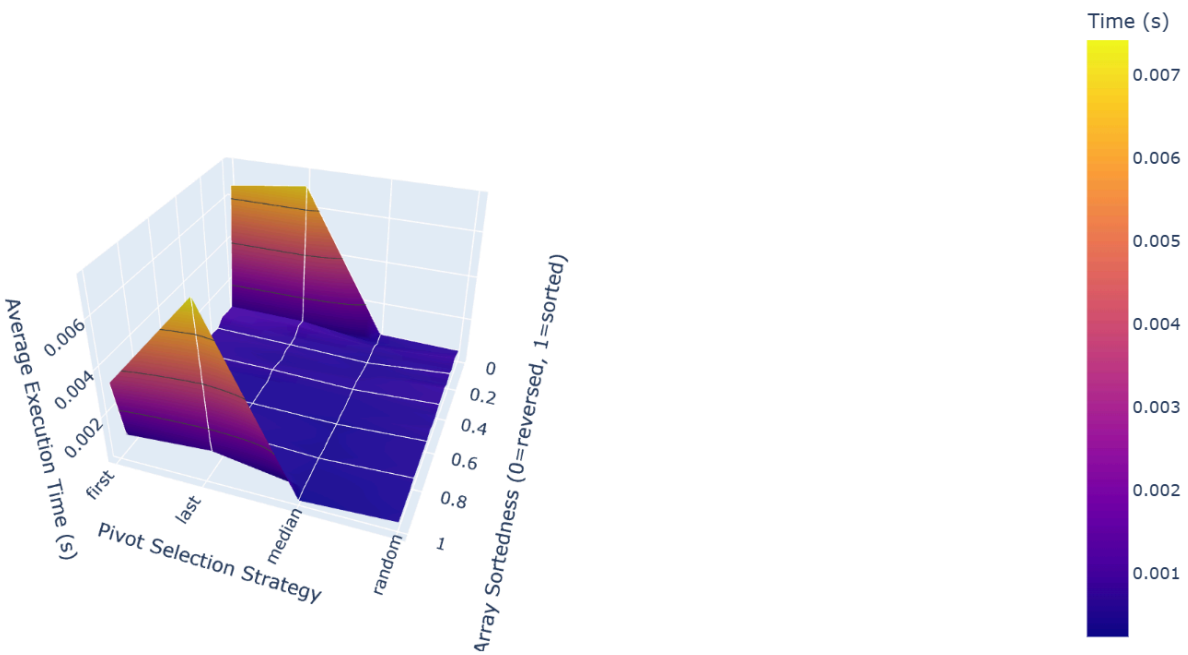
Similarly if the array is totally random, the choice of pivot does not affect time complexity as on average at each step, there is an equalized partition.

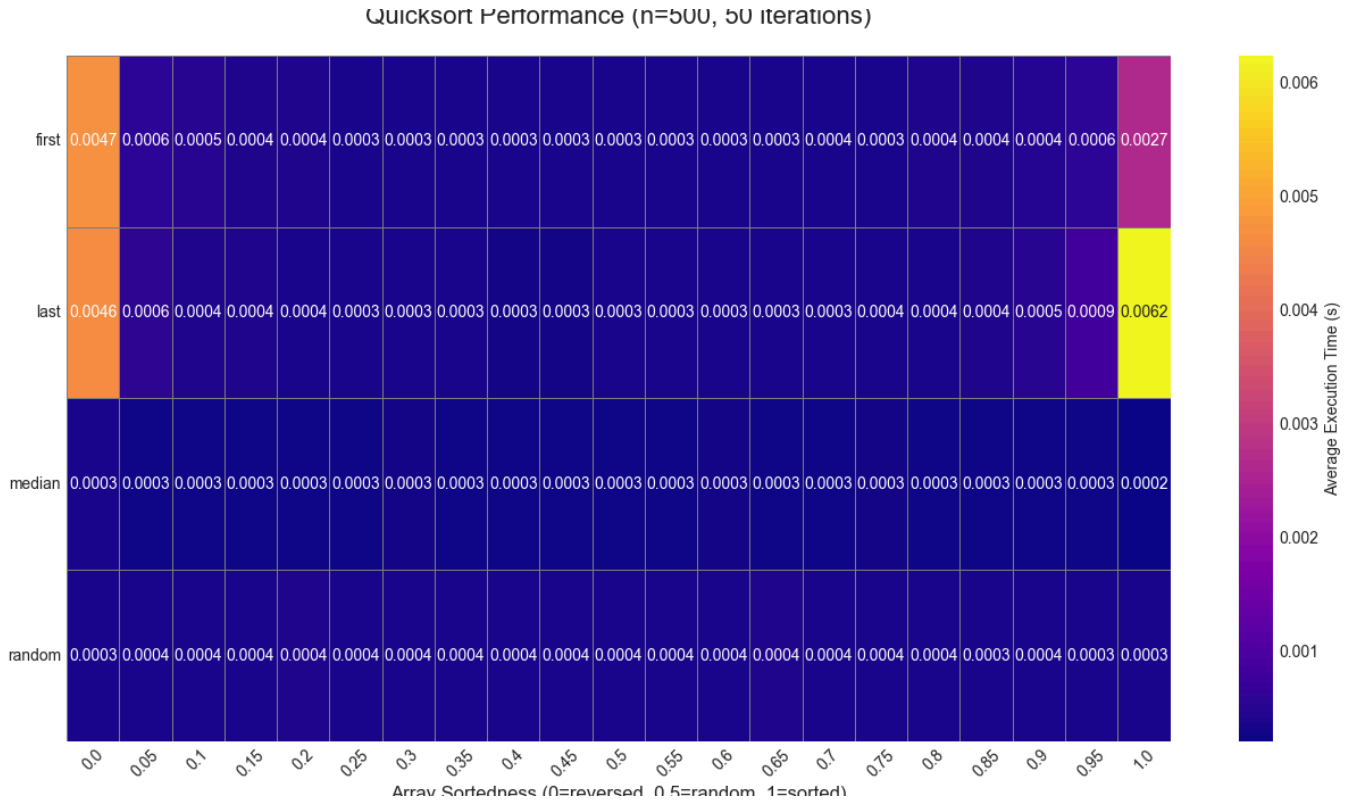
The worst case triggers when the array is fully sorted and you choose the first or last element as pivot which leads to a highly unbalanced partition.

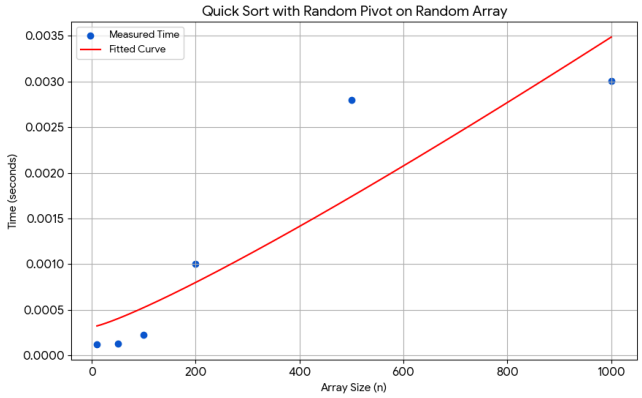
Following is a graph for pivot choice random for different sortedness of the array. As the array is random, we can observe how time complexity is unaffected by the sortedness of array.

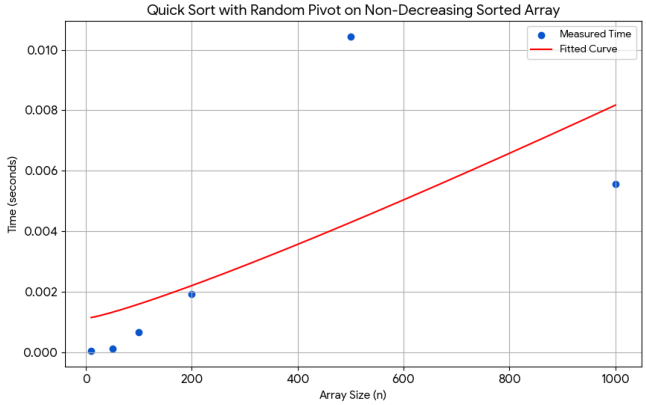


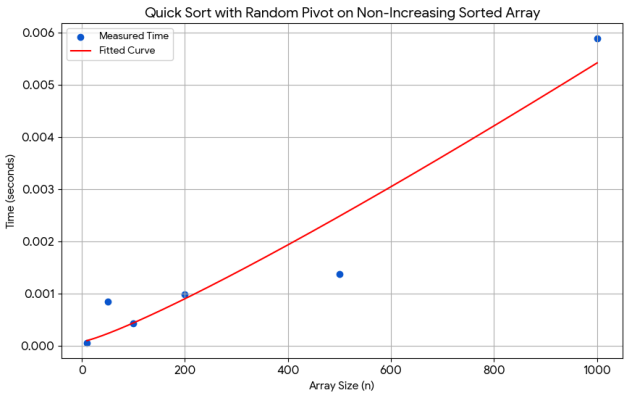
**The following plots are done using python library seaborn*
Since time complexity depends on the sortedness of the array and choice of pivot, here is the 3D map and a heatmap illustrating the worst case complexity of the algorithm.

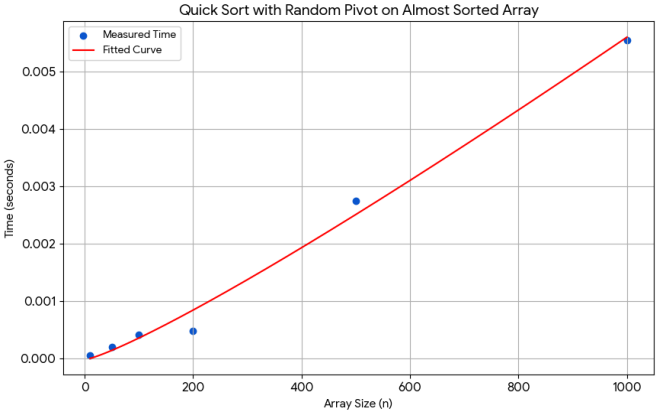



Quicksort Performance (n=500, 50 iterations)
first 0.0047 0.0006 0.0005 0.0004 0.0004 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0004 0.0003 0.0004 0.0004 0.0004 0.0006 0.0027
last 0.0046 0.0006 0.0004 0.0004 0.0004 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0003 0.0004 0.0004 0.0004 0.0005 0.0009 0.0062
median 0.0003 0.0002
random 0.0003 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0004 0.0003 0.0004 0.0003 0.0003
0.0 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7 0.75 0.8 0.85 0.9 0.95 1.0
Array Sortedness (0=reversed, 0.5=random, 1=sorted)
Average Execution Time (s)
0.006
0.005
0.004
0.003
0.002
0.001


Quick Sort with Random Pivot on Random Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)


Quick Sort with Random Pivot on Non-Decreasing Sorted Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)


Quick Sort with Random Pivot on Non-Increasing Sorted Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)

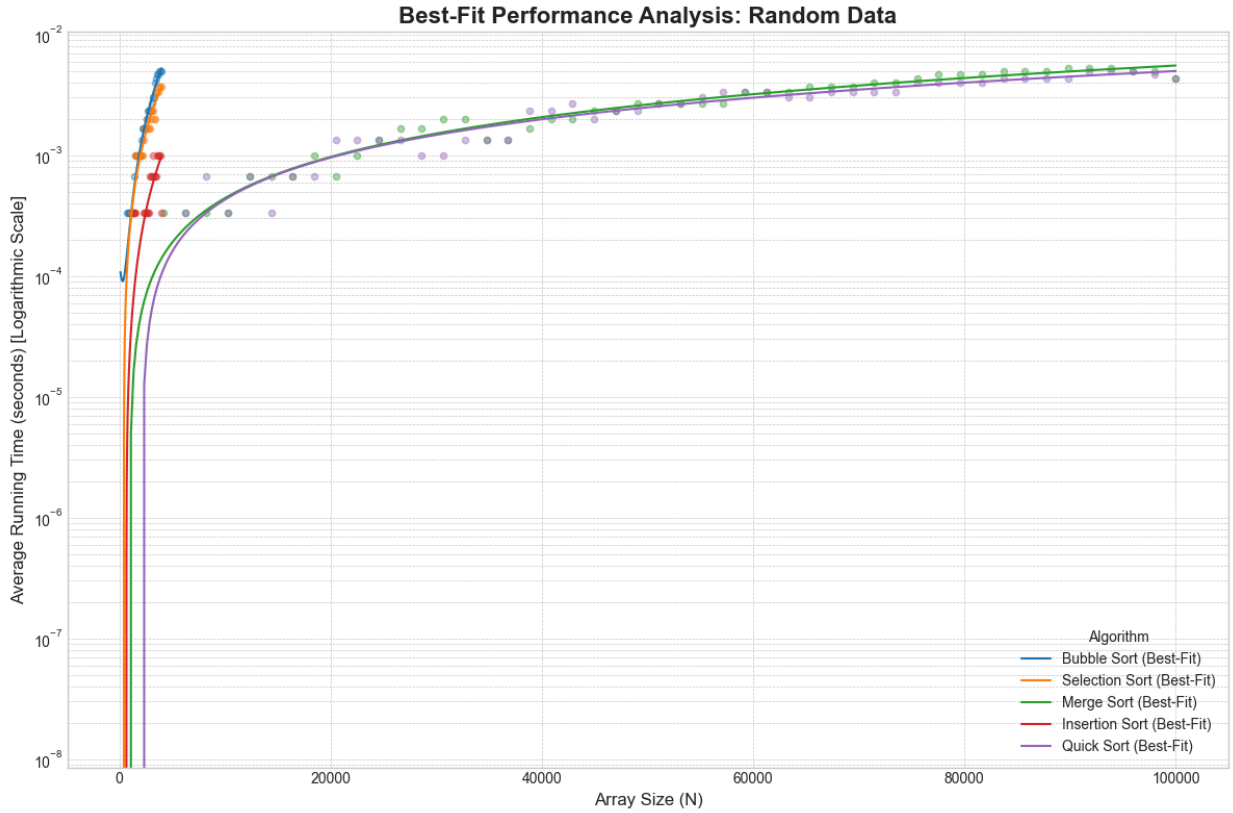

Quick Sort with Random Pivot on Almost Sorted Array
Measured Time
Fitted Curve
Time (seconds)
Array Size (n)

A very interesting algorithmic difference between merge sort and quick sort is that in merge sort, the array is sorted in the backtracking step of recursion, the forward step just divides the array in half. On the other hand, in quick sort, the array is sorted in the forward feed. Once the quick sort algorithm reaches the recursion tree leaves, the array is fully sorted, it backtracks then.

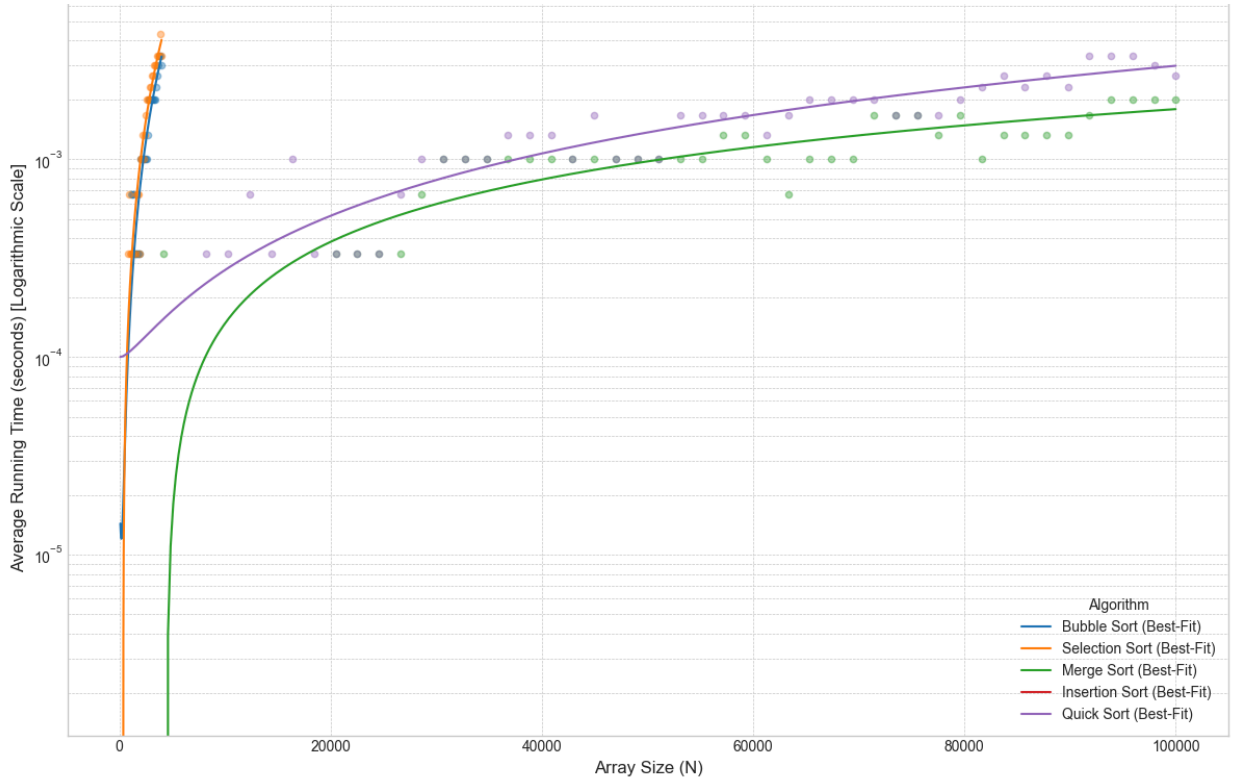
In the further section, the plots help us compare the average time complexity of different algorithms.

**Note that quick sort algorithm has random pivot*

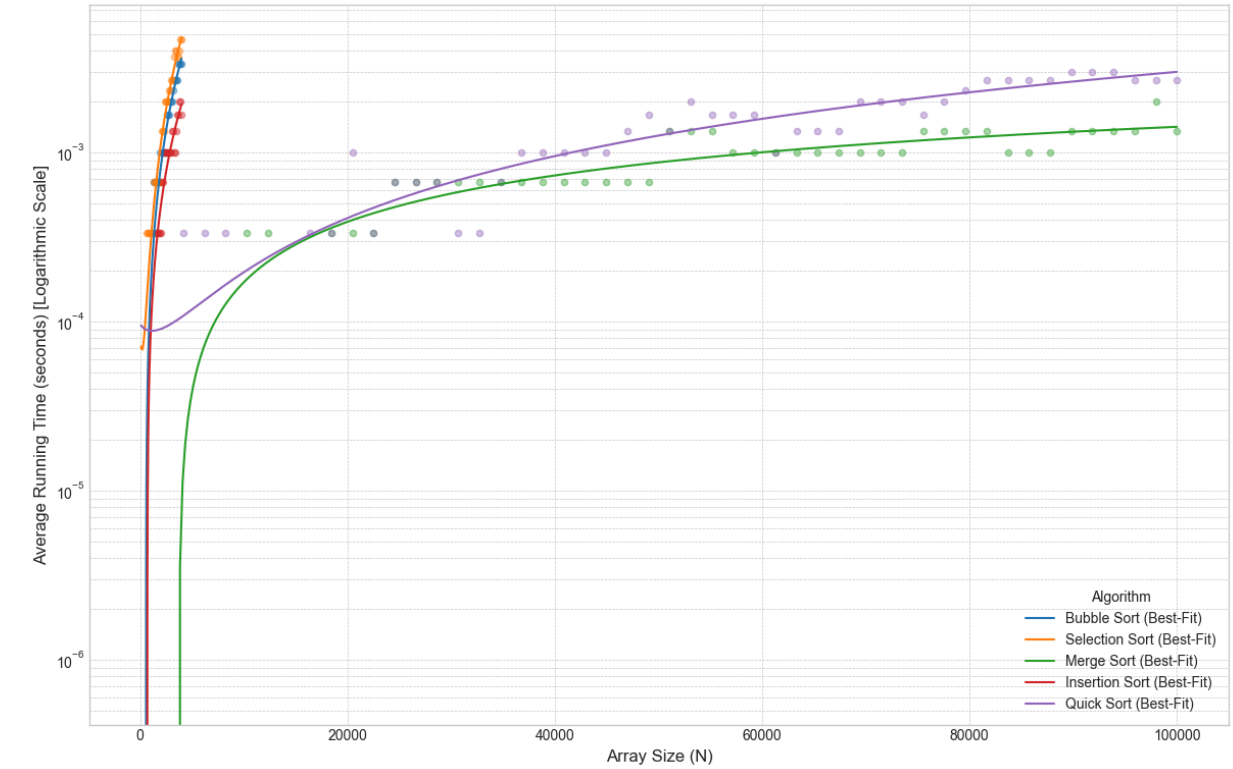
Plot 1 : Random Data
We can observe that the time complexities of both merge sort and quick sort are approximately $O(n*\log_2 n)$ compared to $O(n^2)$ time complexities of the other three algorithms. We can observe that insertion does a little better compared to selection sort or bubble sort.



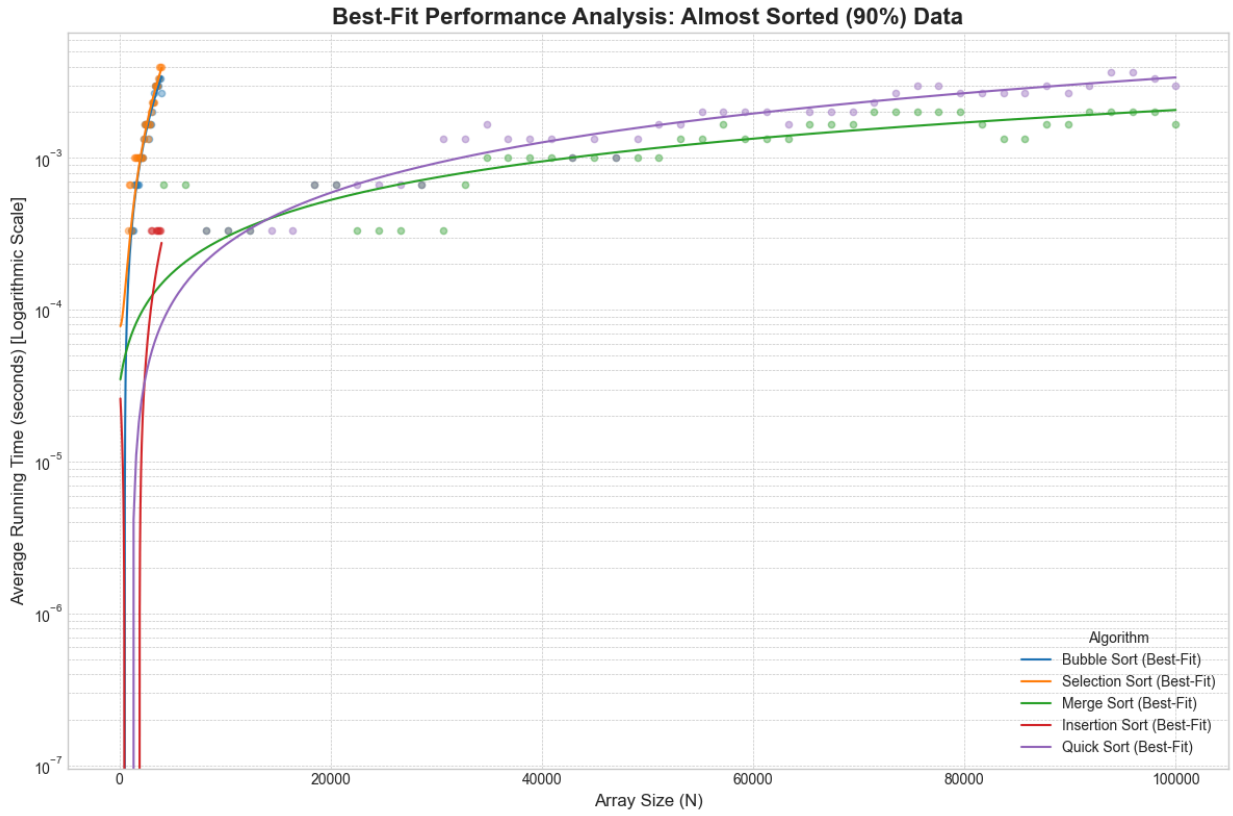
Here are three other plots which convey the same thing
Plot 2 : Non Decreasing Data :



Plot 3 : Non Increasing Data



Plot 4 : Almost Sorted (90%)



On the basis of the above plots, we can conclude that bubble sort, selection sort and insertion sort work almost the same in terms of time complexity, with insertion sort a little better compared to the other. But the complexities of these three $O(n^2)$ are much higher compared to merge sort and quick sort i.e. $O(n*\log_2 n)$.
As the worst case complexity of merge sort is equal to the best case complexity of quick sort, we can notice that quick sort performs a little poorer compared to merge sort.