CS202 Homework-I
Title: Algorithm Efficiency and Sorting
Name/Surname: Ümit Yiğit Başaran
ID: 21704103
Section: 2
Assignment: 1

Question 1

(a) Show that $f(n)$=20n4+20n2+5 is $O(n5)$ by specifying appropriate c and n0 values in Big-O definition

Firstly, we need to find two positive constants: c and n_0 such that:

$0 \leq 20n^4 + 20n^2 + 5 \leq cn^5 \; for \; all \; n \geq n_0$

If we choose $c = 1 \; and \; n_0 = 100$

$\rightarrow 20n^4 + \; 20n^2 + 5 \leq n^5 \; for \; all \; n \geq 100$

Or, if we choose $c = 45 \; and \; n_0 = 1$

$\rightarrow 20n^4 + \; 20n^2 + 5 \leq 45n^5 \; for \; all \; n \geq 1$

(b) Trace the following sorting algorithms to sort the array [18, 4, 47, 24, 15, 24, 17, 11, 31, 23] in ascending order. Use the array implementation of the algorithms as described in the textbook and show all major steps.

- Selection sort

Sorting Steps	# of Key Comparisons	# of Data Moves
[18, 4, 47, 24, 15, 24, 17, 11, 31, 23]	9	3
[18, 4, 23, 24, 15, 24, 17, 11, 31, 47]	8	3
[18, 4, 23, 24, 15, 24, 17, 11, 31, 47]	7	3
[18, 4, 23, 11, 15, 17, 24, 24, 31, 47]	6	3
[18, 4, 17, 11, 15, 23, 24, 24, 31, 47]	5	3
[15, 4, 17, 11, 18, 23, 24, 24, 31, 47]	4	3
[15, 4, 11, 17, 18, 23, 24, 24, 31, 47]	3	3
[11, 4, 15, 17, 18, 23, 24, 24, 31, 47]	2	3
[4, 11, 15, 17, 18, 23, 24, 24, 31, 47]	1	3

# of total key comparisons: 45
# of total data moves: 27

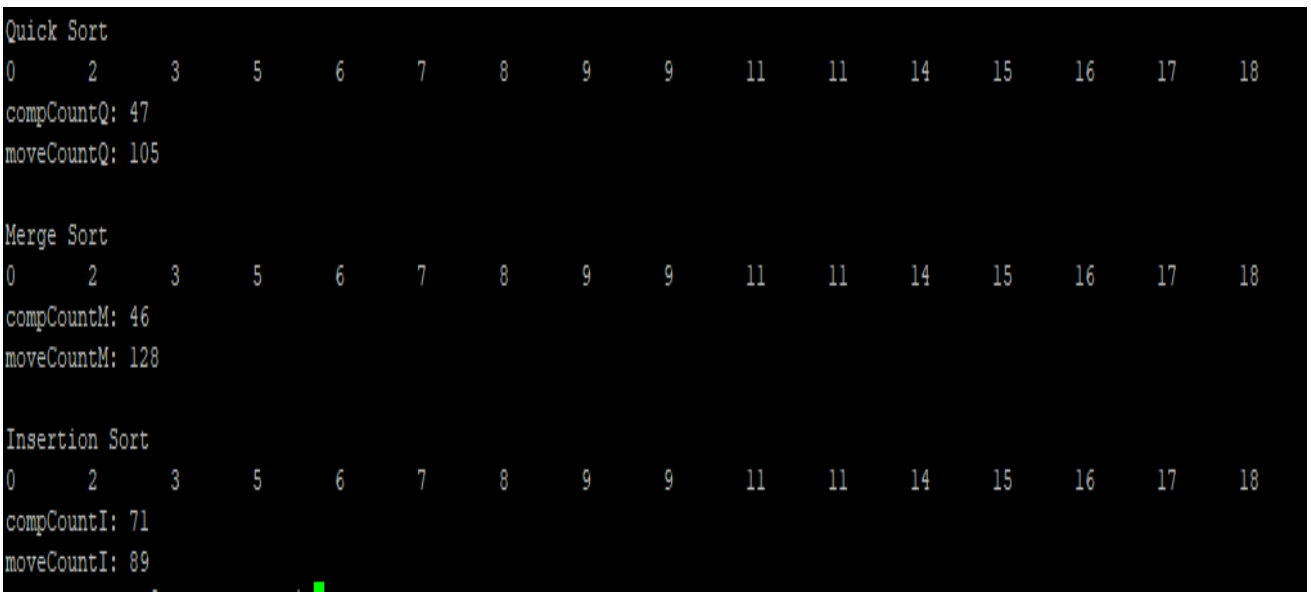
- **Bubble sort**

Sorting Steps	# of Key Comparisons	# of Data Moves
[18, 4, 47, 24, 15, 24, 17, 11, 31, 23]	9	24
[4, 18, 24, 15, 24, 17, 11, 31, 23, 47]	8	12
[4, 18, 15, 24, 17, 11, 24, 23, 31, 47]	7	12
[4, 15, 18, 17, 11, 24, 23, 24, 31, 47]	6	9
[4, 15, 17, 11, 18, 23, 24, 24, 31, 47]	5	3
[4, 15, 11, 17, 18, 23, 24, 24, 31, 47]	4	3
[4, 11, 15, 17, 18, 23, 24, 24, 31, 47]	3	0

# of total key comparisons: 42
# of total data moves: 63

Question 2

- **Screenshot of the solution (Question 2b)**



- **Question 2c**

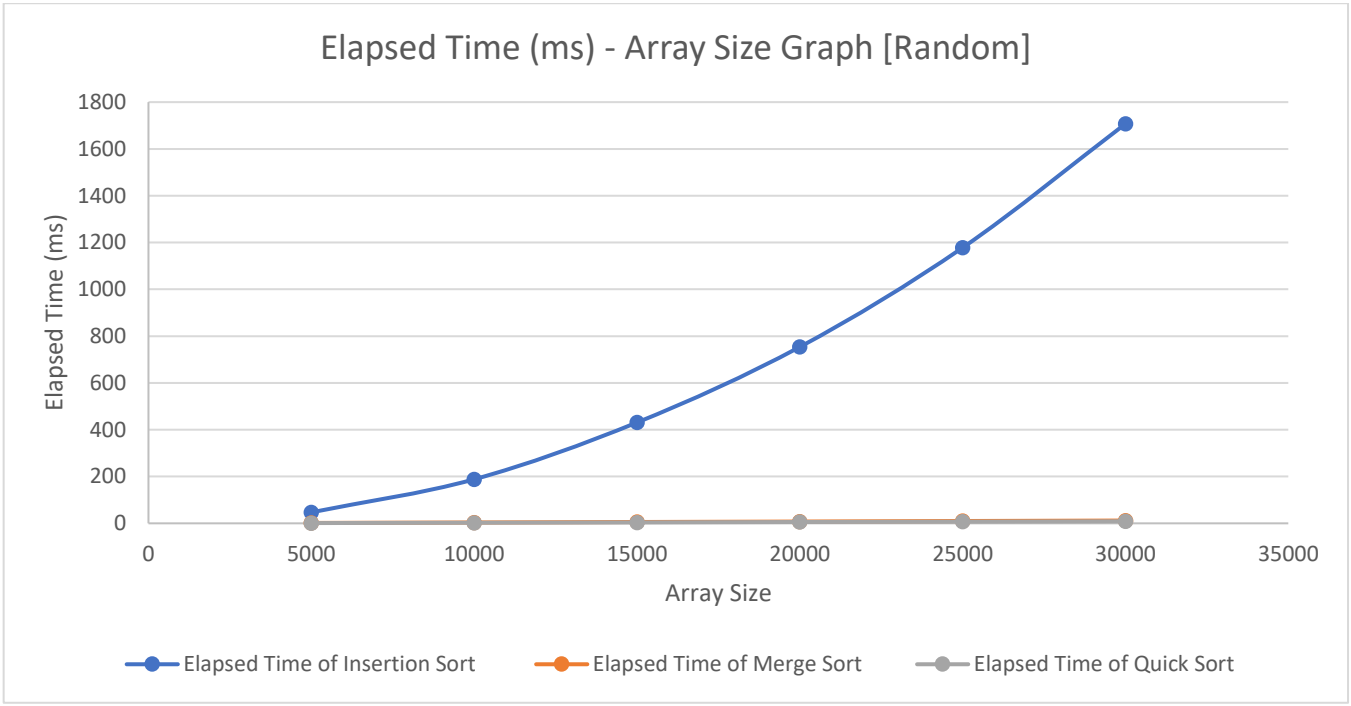
```
-------------------------------------------------------------
Part c - Time analysis of Insertion Sort - RANDOM
Array Size        Elapsed Time            compCount            moveCount
5000               47.0353 ms              6228016              6233021
10000               188.26 ms             25138065             25148074
15000               430.98 ms             56508006             56523012
20000              753.901 ms            100155084            100175091
25000              1178.09 ms            155836084            155861096
30000              1707.44 ms            225486411            225516417
-------------------------------------------------------------
Part c - Time analysis of Merge Sort - RANDOM
Array Size        Elapsed Time            compCount            moveCount
5000                1.7843 ms               55280               123616
10000               3.6679 ms              120396               267232
15000               5.7266 ms              189286               417232
20000               7.7244 ms              261025               574464
25000               9.9214 ms              333986               734464
30000               12.053 ms              408728               894464
-------------------------------------------------------------
Part c - Time analysis of Quick Sort - RANDOM
Array Size        Elapsed Time            compCount            moveCount
5000                1.2272 ms               69659               116895
10000               2.6824 ms              156974               268374
15000               4.0676 ms              232732               375999
20000               5.7428 ms              326737               579870
25000               7.1775 ms              418034               673353
30000               8.8764 ms              562384               746886
```

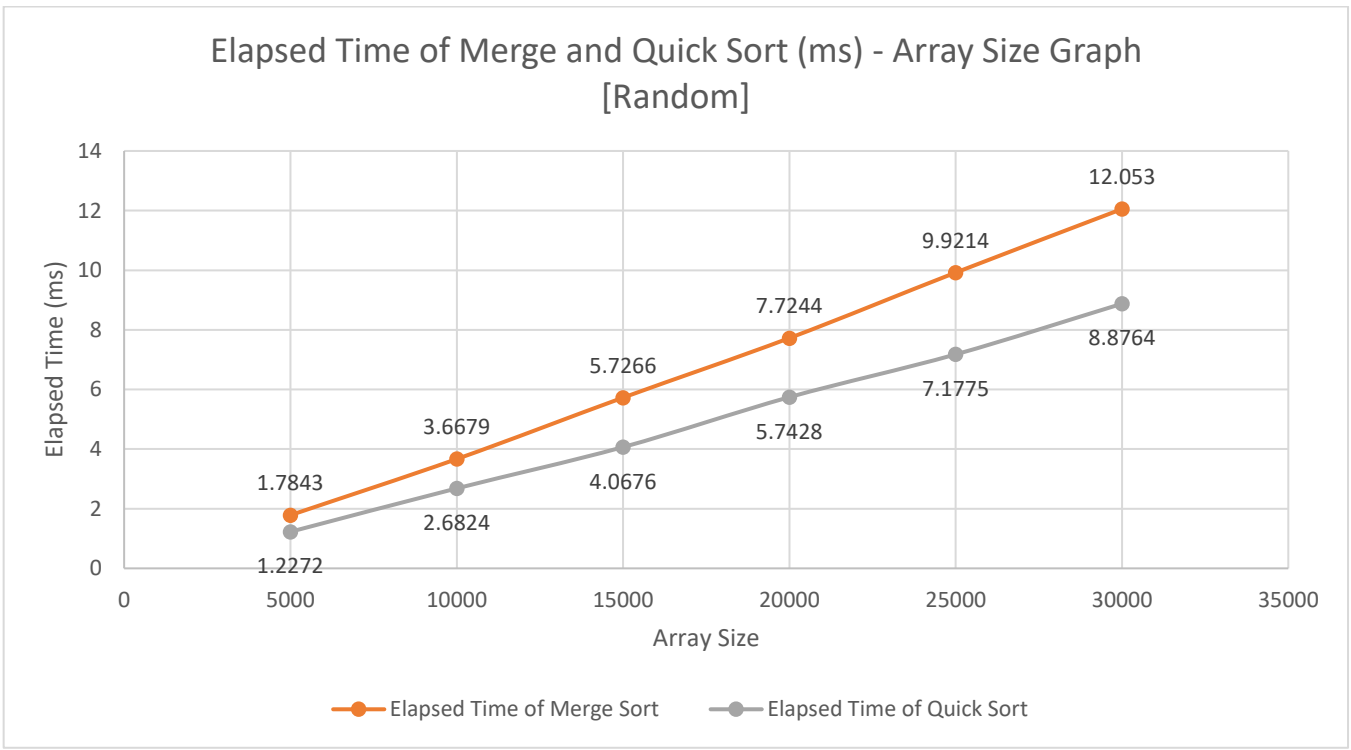
(Figure - 1)

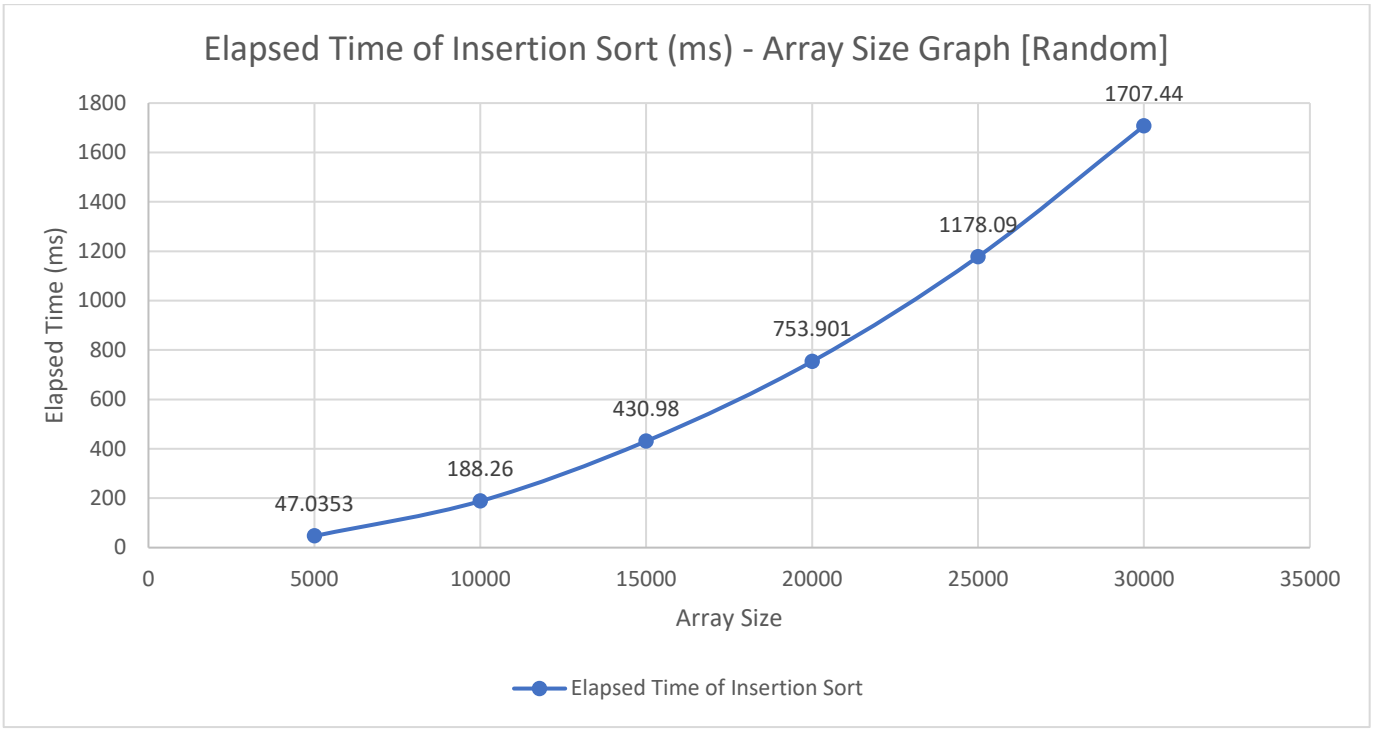
```
-------------------------------------------------------------
Part c - Time analysis of Insertion Sort - SORTED
Array Size        Elapsed Time            compCount            moveCount
5000               0.0228 ms                4999                 9998
10000               0.087 ms                9999                19998
15000              0.1558 ms               14999                29998
20000              0.2079 ms               19999                39998
25000               0.259 ms               24999                49998
30000              0.3105 ms               29999                59998
-------------------------------------------------------------
Part c - Time analysis of Merge Sort - SORTED
Array Size        Elapsed Time            compCount            moveCount
5000               0.3867 ms               32004               123616
10000              1.8587 ms               69008               267232
15000               2.951 ms              106364               417232
20000              5.5997 ms              148016               574464
25000              5.1678 ms              188476               734464
30000              6.4537 ms              227728               894464
-------------------------------------------------------------
Part c - Time analysis of Quick Sort - SORTED
Array Size        Elapsed Time            compCount            moveCount
5000               63.5909 ms            12497500                14997
10000              249.131 ms            49995000                29997
15000               554.66 ms           112492500                44997
20000              1002.02 ms           199990000                59997
25000              1550.45 ms           312487500                74997
30000              2242.05 ms           449985000                89997
```

(Figure - 2)

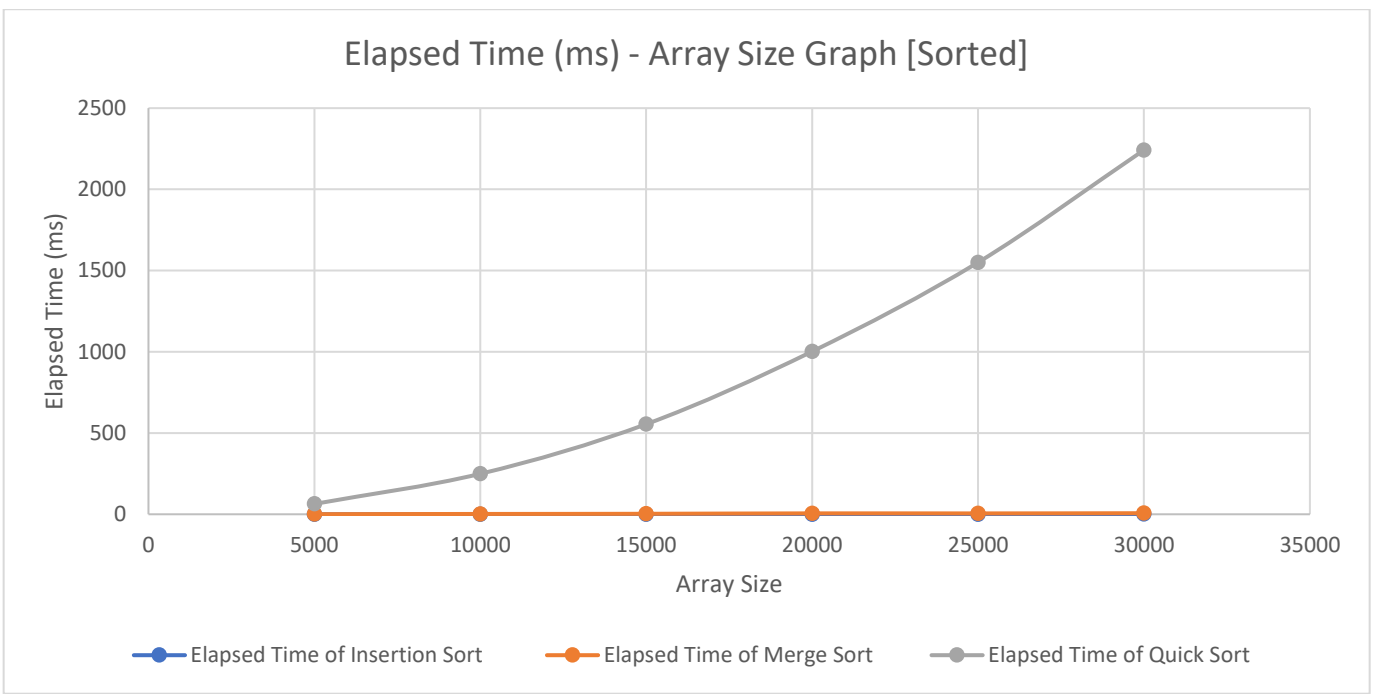
- **Graphs of arrays which are randomly created**

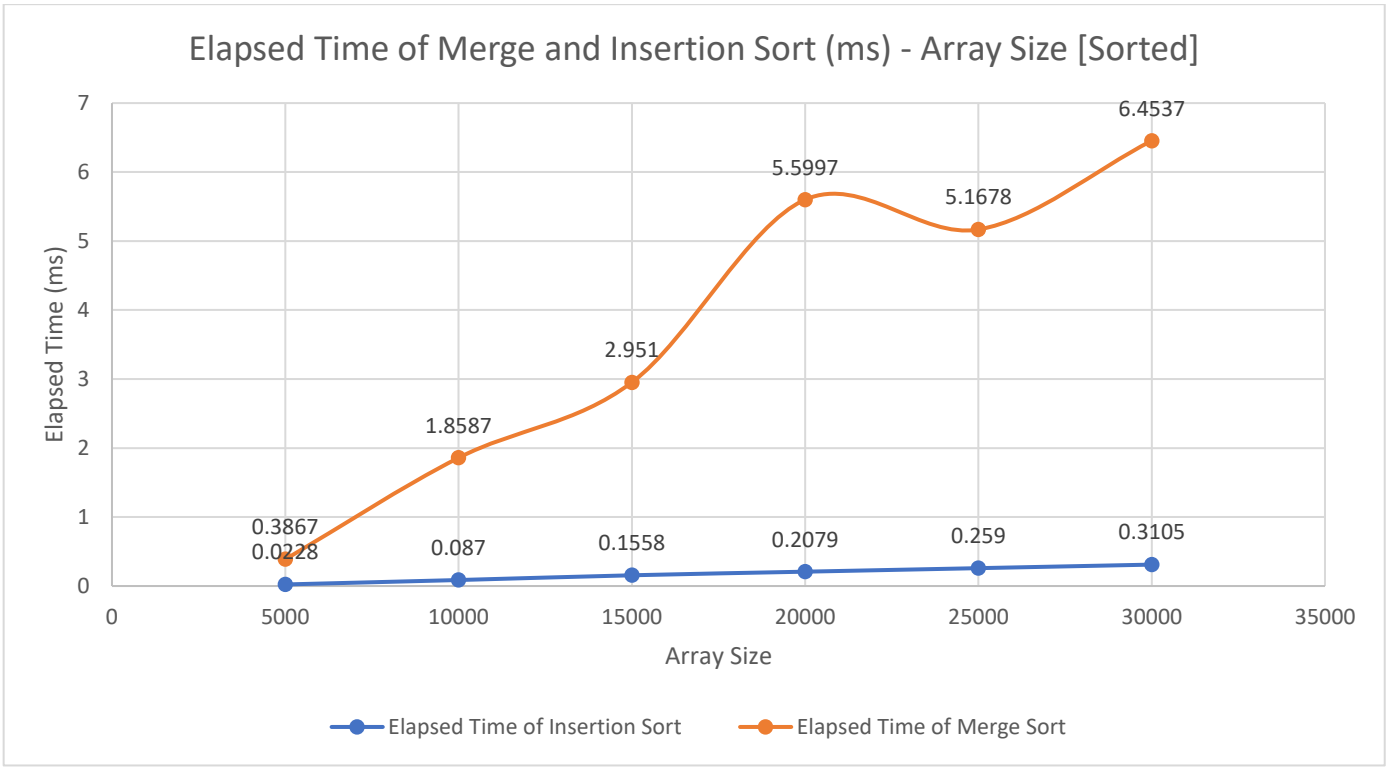


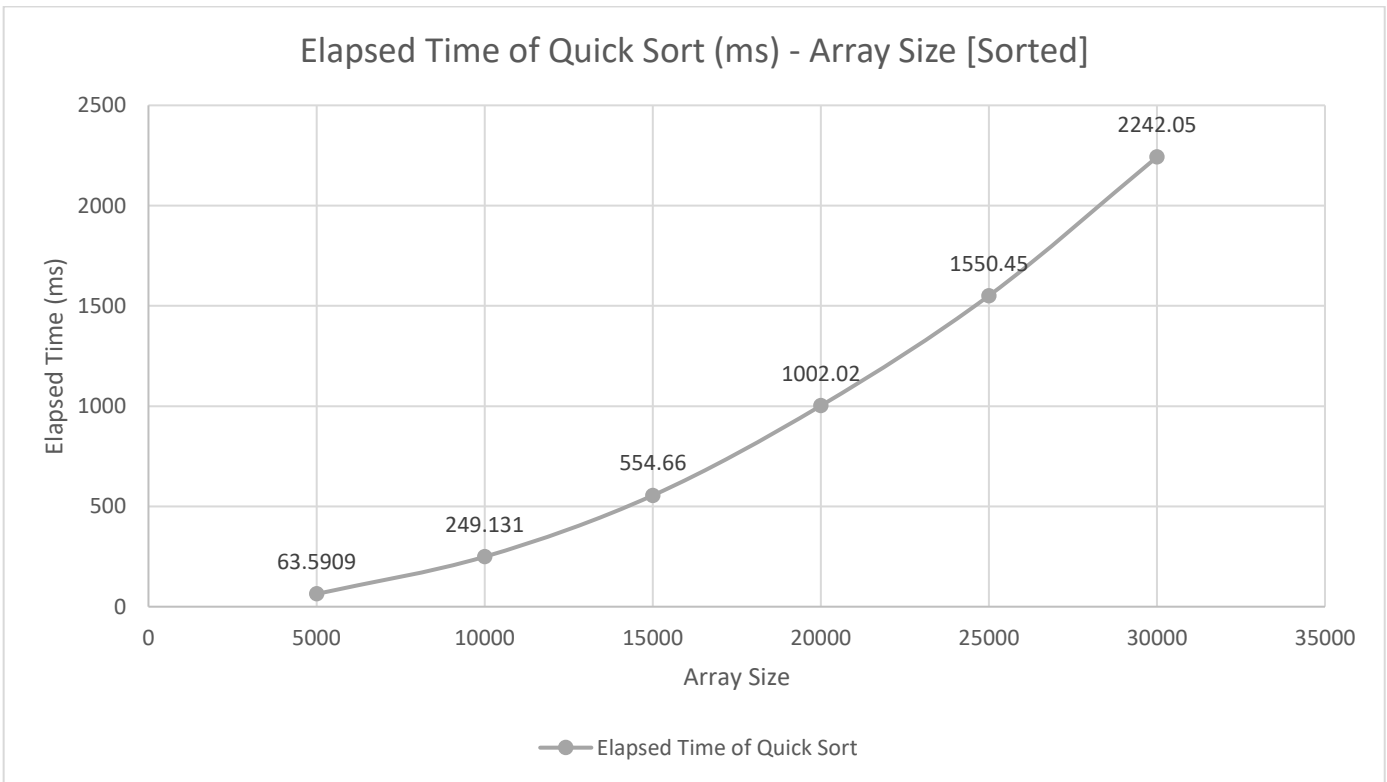




- **Graphs of arrays which are already sorted**







As a result, for randomly created arrays, we could see that insertion sort has much more large values than merge and quick sort. Similar to the theoretical results, empirical results show that insertion sort works close to the $O(n^2)$, although merge and quick sort work close to the $O(n \log n)$. For example, when array size is equal to the 30000 then elapsed time of the insertion sort is equal to the 1707.44 millisecond, however, for merge sort this number is equal to the 12.053 millisecond and for quick sort this number is equal to 8.8764 millisecond as you can see from Figure – 1 and graphs for randomly created arrays.

For already sorted arrays, we could see that quick sort has much more large values than merge and insertion sort. Similar to the theoretical results, empirical results show that quick sort works close to the $O(n^2)$, merge sort works close to the $O(n \log n)$, insertion sort works close to the $O(n)$. Quick sort works close to the $O(n^2)$ because we choose the first array element as a pivot and quick sort based on divide and conquer principle that contains unsorted and sorted parts of the arrays so that it divides the array n part and for n part it check elements n time. Merge sort works close to the $O(n \log n)$ because for merge sort it doesn't matter whether the array is sorted or not, it always works similar to $O(n \log n)$. Insertion sort works $O(n)$ because it doesn't enter the second loop to select the right index for the elements. For example, when array size is equal to the 30000 then elapsed time of the insertion sort is equal to the 0.3105, for merge sort this number is equal to the 6.4537 millisecond and for quick sort this number is equal to 2242.05 millisecond as you can see from Figure – 2 and graphs for already sorted arrays. Therefore, there is a difference between elapsed time results of already sorted arrays and randomly created arrays because number of key comparisons and number of data moves change to the situation to situation based on their algorithms. Among those three the difference can be observed based on Figure-1 and Figure-2. Additionally, for both cases, merge sort uses more data allocation rather than quick sort because of merge sorts' own algorithm.

Question 3

```
Average Distance: 0.5462
Elapsed Time: 0.6546 ms
Elapsed Time: 8.0669 ms
Elapsed Time: 6210.99 ms

Average Distance: 1.11196
Elapsed Time: 0.9569 ms
Elapsed Time: 8.2723 ms
Elapsed Time: 4978.95 ms

Average Distance: 1.62374
Elapsed Time: 1.7277 ms
Elapsed Time: 10.4228 ms
Elapsed Time: 4055.93 ms

Average Distance: 2.1273
Elapsed Time: 1.4065 ms
Elapsed Time: 8.8733 ms
Elapsed Time: 3859.94 ms

Average Distance: 2.62094
Elapsed Time: 1.632 ms
Elapsed Time: 8.7668 ms
Elapsed Time: 3303.2 ms

Average Distance: 3.13186
Elapsed Time: 1.8434 ms
Elapsed Time: 9.7583 ms
Elapsed Time: 2850.8 ms

Average Distance: 3.62546
Elapsed Time: 1.841 ms
Elapsed Time: 9.8424 ms
Elapsed Time: 2846.71 ms

Average Distance: 4.11476
Elapsed Time: 2.238 ms
Elapsed Time: 10.1049 ms
Elapsed Time: 2460.24 ms

Average Distance: 4.60624
Elapsed Time: 1.8477 ms
Elapsed Time: 9.4539 ms
Elapsed Time: 2005.63 ms

Average Distance: 5.11588
Elapsed Time: 1.9231 ms
Elapsed Time: 10.168 ms
Elapsed Time: 1827.65 ms

Average Distance: 5.59116
Elapsed Time: 2.082 ms
Elapsed Time: 9.4704 ms
Elapsed Time: 1800.98 ms
```

(Figure – 3)

```
Average Distance: 21304.1
Average Distance: 21592.3
Average Distance: 21707.8
Average Distance: 21623.3
Average Distance: 21681.8
Average Distance: 21582.4
Average Distance: 21640.4
Average Distance: 21623.1
Average Distance: 21664.5
Average Distance: 21581.4
Elapsed Time: 3760.14 ms
Elapsed Time: 14.7257 ms
Elapsed Time: 18.2519 ms

Average Distance: 21290.9
Average Distance: 21605.5
Average Distance: 21668.9
Average Distance: 21643.1
Average Distance: 21687.2
Average Distance: 21652.7
Average Distance: 21644.7
Average Distance: 21614.1
Average Distance: 21628.8
Average Distance: 21490.8
Elapsed Time: 3733.89 ms
Elapsed Time: 14.7359 ms
Elapsed Time: 16.5695 ms

Average Distance: 21309.8
Average Distance: 21636.7
Average Distance: 21666.7
Average Distance: 21633.5
Average Distance: 21630.8
Average Distance: 21659.9
Average Distance: 21569.5
Average Distance: 21696.5
Average Distance: 21657.9
Average Distance: 21643.1
Elapsed Time: 3744.61 ms
Elapsed Time: 14.7132 ms
Elapsed Time: 21.5498 ms

Average Distance: 21315.7
Average Distance: 21630.5
Average Distance: 21572.4
Average Distance: 21669.6
Average Distance: 21659.3
Average Distance: 21675.4
Average Distance: 21667.5
Average Distance: 21650.6
Average Distance: 21539.9
Average Distance: 21647.2
Elapsed Time: 3725.74 ms
Elapsed Time: 14.6498 ms
Elapsed Time: 22.9638 ms
```

(Figure – 4)

```
Average Distance: 33120.5
Average Distance: 33420.5
Average Distance: 33384.8
Average Distance: 33222.8
Average Distance: 33271.6
Average Distance: 33271
Average Distance: 33405.5
Average Distance: 33274
Average Distance: 33297
Average Distance: 33258.3
Elapsed Time: 6236.3 ms
Elapsed Time: 15.0484 ms
Elapsed Time: 11.1828 ms

Average Distance: 33286
Average Distance: 33252
Average Distance: 33261.4
Average Distance: 33379.2
Average Distance: 33363.4
Average Distance: 33424.5
Average Distance: 33379.6
Average Distance: 33314
Average Distance: 33232.9
Average Distance: 33325.4
Elapsed Time: 6352.5 ms
Elapsed Time: 15.7658 ms
Elapsed Time: 10.9325 ms

Average Distance: 33233.2
Average Distance: 33325
Average Distance: 33351
Average Distance: 33369.3
Average Distance: 33342.4
Average Distance: 33411.6
Average Distance: 33387.3
Average Distance: 33330.8
Average Distance: 33394.4
Average Distance: 33296.5
Elapsed Time: 6260.16 ms
Elapsed Time: 15.2344 ms
Elapsed Time: 11.0919 ms

Average Distance: 33333.2
Average Distance: 33246.9
Average Distance: 33331
Average Distance: 33380.6
Average Distance: 33286.7
Average Distance: 33304.5
Average Distance: 33410.1
Average Distance: 33233.1
Average Distance: 33379.9
Average Distance: 33342.9
Elapsed Time: 6236.7 ms
Elapsed Time: 15.0401 ms
Elapsed Time: 11.0615 ms
```

(Figure – 5)

Note: Elapsed times that are represented in the screenshots are for insertion sort, merge sort and quick sort respectively.

For an array with 100000 size, we have three cases:

I. K is much smaller than size (Figure - 3)
II. K is close to the size/2 (Figure - 4)
III. K is close to the size (Figure - 5)

K represents that every element in the array is at most K indices away from its original index.

As shown in the Figure – 3 when K is much smaller than the size the array, we could see that the most effective sort type is insertion sort because the growth rate of insertion sort is $O(n)$ for that case. Again, from Figure – 3, we could see that the elapsed time of insertion sort is increasing while the value of K is increasing (the elapsed time of insertion sort when the average distance is close to zero is 0.5462, the elapsed time of insertion sort when the growth rate is close to 5.5 is 2.082). When K is much smaller than the size means that the array is similar to the sorted array itself so that quick sort is the least efficient one for this case because for sorted arrays growth rate of the quick sort is $O(n^2)$. From Figure – 3, for quick sort, while average distance is increasing the elapsed time of quick sort is decreasing because the array starting to become unsorted which is a positive situation for quick sort (the elapsed time of quick sort when the average distance is close to 0 is 6210.99, the elapsed time of quick sort when the growth rate is close to 5.5 is 1800.98). For the merge sort, the elapsed times don't depend on the K values and its' growth rate is $O(n \log n)$. As a result, based on Figure – 3 for small K values, insertion sort satisfies the best case and quick sort satisfies the worst case.

As shown in the Figure – 4 when K is close to the size/2, then the most effective sort type is merge sort because the growth rate of merge sort is $O(n \log n)$ which doesn't depend on cases. However, the growth rate of the insertion sort is $O(n^2)$ and the growth rate of the quick sort is $O(n \log n)$. In this case, the growth rates are similar to the average cases of each algorithms. Based on Figure – 4, we could see that as the elapsed time of insertion sort continues to increase, the elapsed time of the quick sort continues to decrease because the array is no longer close to the sorted array. For the merge sort, the values which are the elapsed time of the merge sort between Figure – 3 and Figure – 4 are close to each other. Therefore, if we have enough space to do merge sort, the most efficient way of sorting the array with K value which is close to the size/2 is merge sort.

As shown in the Figure – 5 when K is close to the size, then the most effective sort type is quick sort because the growth rate of quick sort is $O(n \log n)$. The growth rate of quick sort is $O(n \log n)$ for that case because it use divides and conquer method and we choose the first element as a pivot so that it nearly divides all arrays to half of its' size which cause $\log n$ data usage. In this case, the growth rate of the insertion sort is $O(n^2)$ and the growth rate of the merge sort $O(n \log n)$. Again, for the insertion sort, the elapsed time of the insertion sort continues to increase as can be seen from Figure – 5. For merge sort, its' growth rate still doesn't depend on the K value and from the figures we can see the small differences of the elapsed time of the merge sort. Therefore, for K values that are close to the size, quick sort can be the most suitable and efficient sort algorithm because its' growth rate is $O(n \log n)$ and it doesn't require extra space from the memory.

To sum up, we could say that for K values that are close to the beginning, we can use insertion sort algorithm to reach most efficient time. For K values that are close to size/2, we can use merge sort algorithm to sort the array in the $O(n \log n)$ growth rate. Also, for K values that are close to the end of the array we can use quick sort algorithm for the most efficient way.