```
/*
 * Title: Algorithm Efficiency and Sorting
 * Author: İlknur Baş
 * ID: 21601847
 * Section: 2
 * Assignment: 1
 * CS 202, Fall 2020 Homework 1 - Algorithm Efficiency and
 * and Sorting
 * 08.03.2020
*/
```

QUESTION 1

(a)

$f(n) = 20n^4 + 20n^2 + 5$ is $O(n^5)$ by the Big-O definition. If $f(n) \le cn^5$ when$n \ge \boldsymbol{n_0}$. So the inequality is $20n^4 + 20n^2 + 5 \le cn^5$. Divide both side by n^5 and $\frac{20}{n} + \frac{20}{n^3} + \frac{5}{n^5} \le c$. Therefore, condition holds for $n \ge \boldsymbol{n_0} = 1$ and $c \ge 20 + 20 + 5 = 45$. So there exists constant c and $\boldsymbol{n_0}$ that satisfy the condition.

(b) [18, 4, 47, 24, 15, 24, 17, 11, 31, 23]

blue: is the number that will be compared and inserted to the valid position
red: is the number that compared with the blue one

18	4	47	24	15	24	17	11	31	23
18	4	47	24	15	24	17	11	31	23
4	18	47	24	15	24	17	11	31	23
4	18	47	24	15	24	17	11	31	23
4	18	47	24	15	24	17	11	31	23
4	18	24	47	15	24	17	11	31	23
4	18	24	47	15	24	17	11	31	23
4	18	24	15	47	24	17	11	31	23
4	18	24	15	47	24	17	11	31	23
4	18	24	15	24	47	17	11	31	23
4	18	24	15	24	47	17	11	31	23
4	18	24	15	24	17	47	11	31	23
4	18	24	15	24	17	47	11	31	23
4	18	24	15	24	17	11	47	31	23
4	18	24	15	24	17	11	47	31	23
4	18	24	15	24	17	11	31	47	23
4	18	24	15	24	17	11	31	47	23
4	18	24	15	24	17	11	31	23	47
4	18	24	15	24	17	11	31	23	47
4	18	24	15	24	17	11	31	23	47
4	18	24	15	24	17	11	31	23	47
4	18	15	24	24	17	11	31	23	47
4	18	15	24	24	17	11	31	23	47

4	18	15	24	24	17	11	31	23	47
4	18	15	24	17	24	11	31	23	47
4	18	15	24	17	24	11	31	23	47
4	18	15	24	17	11	24	31	23	47
4	18	15	24	17	11	24	31	23	47
4	18	15	24	17	11	24	31	23	47
4	18	15	24	17	11	24	23	31	47
4	18	15	24	17	11	24	23	31	47
4	18	15	24	17	11	24	23	31	47
4	15	18	24	17	11	24	23	31	47
4	15	18	24	17	11	24	23	31	47
4	15	18	24	17	11	24	23	31	47
4	15	18	17	24	11	24	23	31	47
4	15	18	17	24	11	24	23	31	47
4	15	18	17	11	24	24	23	31	47
4	15	18	17	11	24	24	23	31	47
4	15	18	17	11	24	24	23	31	47
4	15	18	17	11	24	23	24	31	47
4	15	18	17	11	24	23	24	31	47
4	15	18	17	11	24	23	24	31	47
4	15	18	17	11	24	23	24	31	47
4	15	17	18	11	24	23	24	31	47
4	15	17	18	11	24	23	24	31	47
4	15	17	11	18	24	23	24	31	47
4	15	17	11	18	24	23	24	31	47
4	15	17	11	18	24	23	24	31	47
4	15	17	11	18	23	24	24	31	47
4	15	17	11	18	23	24	24	31	47
4	15	17	11	18	23	24	24	31	47
4	15	17	11	18	23	24	24	31	47

4	15	11	17	18	23	24	24	31	47
4	15	11	17	18	23	24	24	31	47
4	15	11	17	18	23	24	24	31	47
4	15	11	17	18	23	24	24	31	47
4	15	11	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47

yellow: is the numbers that will be swapped

Bubble Sort: (bold lines indicate the each pass, in order words sorted part of the array)

Selection Sort: (In each iteration we are choosing the largest element of unsorted array. Bold numbers indicates the largest element. "|" indicates the sorted and unsorted part. Left hand size unsorted, right hand size sorted.)

Initial array: [18, 4, **47**, 24, 15, 24, 17, 11, 31, 23]
After 1st swap : [18, 4, 23, 24, 15, 24, 17, 11, **31** | 47]
After 2nd swap : [18, 4, 23, **24**, 15, 24, 17, 11 | 31 ,47]
After 3rd swap : [18, 4, 23, 11, 15, **24**, 17 | 24 , 31 ,47]
After 4th swap : [18, 4, 23, 11, 15, **24**, 17 | 24 , 31 ,47]
After 5th swap : [18, 4, **23**, 11, 15, 17 | 24 , 24 , 31 ,47]
After 6th swap : [**18**, 4, 17, 11, 15 | 23 , 24 , 24 , 31 ,47]
After 7th swap : [15, 4, **17**, 11 | 18 ,23 , 24 , 24 , 31 ,47]
After 8th swap : [**15**, 4, 11 | 17 ,18 ,23 , 24 , 24 , 31 ,47]
After 9th swap : [**11**, 4 | 15 ,17 ,18 ,23 , 24 , 24 , 31 ,47]
After 10th swap : [**4** | 11, 15 ,17 ,18 ,23 , 24 , 24 , 31 ,47]
Sorted array: [4, 11, 15 ,17 ,18, 23, 24, 24 , 31 ,47]

QUESTION 2

(b)

```
[ilknur.bas@dijkstra ~]$ make
g++ main.o sorting.o -o  hw1 -std=c++11
[ilknur.bas@dijkstra ~]$ make
g++ -c main.cpp -std=c++11
g++ -c sorting.cpp -std=c++11
g++ main.o sorting.o -o  hw1 -std=c++11
[ilknur.bas@dijkstra ~]$ ./hw1
Insertion sort:
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18

Comparison Count: 15 Data Moves: 89

Quick sort:
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18

Comparison Count: 120 Data Moves: 60

Merge sort:
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18

Comparison Count: 34 Data Moves: 128
```

Screenshot of the executable file hw1

(c) output of console

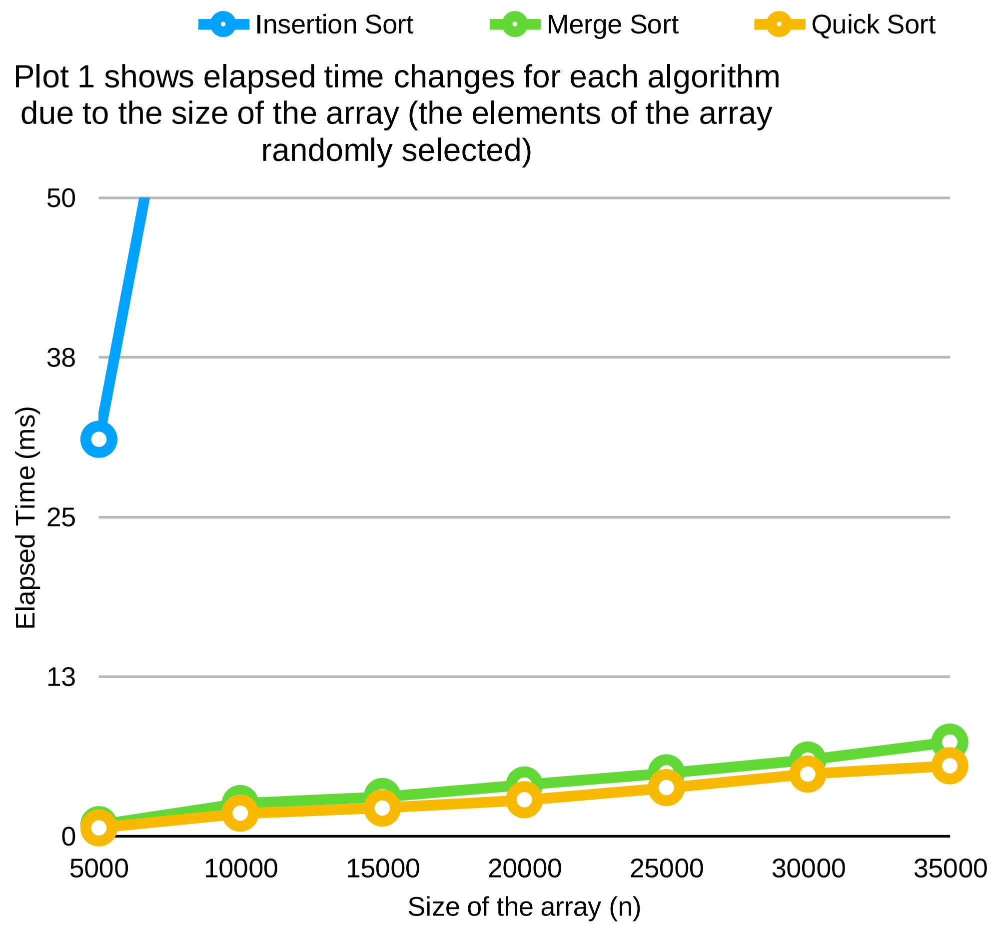
```
Random array elements****
-----------------------------------
Question 2c - Time analysis of Insertion Sort
Array size   Time Elapsed       compCount      moveCount
5000           35.7412           4999          6342296
10000           142.106           9999          25090351
15000           319.863          14999          56553669
20000           567.758          19999         100232971
25000           876.548          24999         154882719
30000           1275.21          29999         225481638
35000           1727.73          34999         306699506
-----------------------------------
Question 2c - Time analysis of Merge Sort
Array size   Time Elapsed       compCount      moveCount
5000           1.48691           55237          123616
10000           3.14716          120411          267232
15000           4.89404          189383          417232
20000           6.69711          261023          574464
25000            8.5117          333866          734464
30000           10.3362          408781          894464
35000           12.3077          484346         1058928
-----------------------------------
Question 2c - Time analysis of Quick Sort
Array size   Time Elapsed       compCount      moveCount
5000           1.0841            80591          122215
10000           2.30787          159455          254034
15000           3.46626          242093          365414
20000           4.80311          324658          534154
25000              6.12          423165          687625
30000           7.57676          534904          889511
35000            8.7435          615919          970550
```

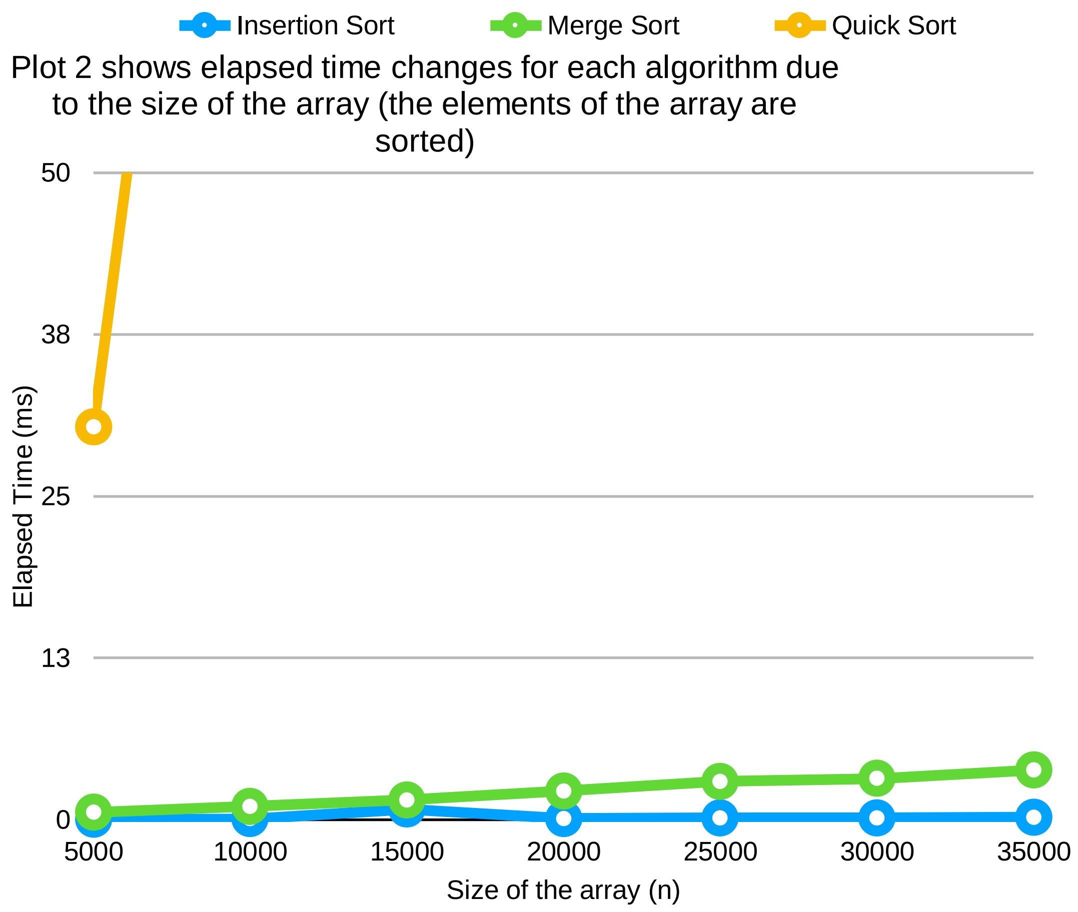
The elapsed time when the elements of the array are randomly selected

```
ilknur — ilknur.bas@dijkstra:~ — ssh ilknur.bas@dijkstra.ug.bcc.bilkent.edu.tr...
Already sorted array elements****
------------------------------------
Question 2c - Time analysis of Insertion Sort
Array size    Time Elapsed       compCount       moveCount
5000           0.050915           4999            9998
10000           0.101231           9999            19998
15000           0.150765          14999            29998
20000           0.200997          19999            39998
25000           0.250396          24999            49998
30000           0.304977          29999            59998
35000           0.351575          34999            69998
------------------------------------
Question 2c - Time analysis of Merge Sort
Array size    Time Elapsed       compCount       moveCount
5000           0.933561          32004           123616
10000            1.97425          69008            267232
15000            3.05891         106364            417232
20000            4.18648         148016            574464
25000            5.31788         188476            734464
30000            6.42859         227728            894464
35000            7.56874         269364           1058928
------------------------------------
Question 2c - Time analysis of Quick Sort
Array size    Time Elapsed       compCount       moveCount
5000            53.3502        12497500           19996
10000            212.821        49995000            39996
15000            478.932       112492500            59996
20000            851.309       199990000            79996
25000            1330.03       312487500            99996
30000             1915.2       449985000           119996
35000            2606.33       612482500           139996
```

The elapsed time when the elements of the array are already sorted

(d)





Insertion Sort
Merge Sort
Quick Sort
Plot 2 shows elapsed time changes for each algorithm due to the size of the array (the elements of the array are sorted)
50
38
25
13
0
Elapsed Time (ms)
5000
10000
15000
20000
25000
30000
35000
Size of the array (n)

Explanation

Insertion Sort

When data and plot lines for the Insertion sort are compared with other one, the elapsed time is much more larger when the array elements are selected randomly. In Plot 1, insertion sort's growth rate is faster compared to Plot 2. In other words, we can say that, Plot 1 shows the worst case of insertion sort which is when array elements is randomly picked; and plot 2 shows the best case which is sorted array elements. For instance, when array size is 35000, the elapsed time for Plot 1 is 1727.73, for Plot 2 is 0,3515. Plot 1's time is much more slow. That means, theoretical results has matched with the experimental results. (Best case of insertion is $O(n)$). Also, since insertion sort's elapsed time is much more bigger when it is compared to merge and quick sort for Plot 1, using insertion sort in Plot 1 is not good for efficiency purposes. The reason of differences in time complexities in Plot 1 & Plot 2 can be explain with the number of data moves. The data moves in plot1 ranges from 6.342.296-306.496.807, however in Plot 2 it ranges from 9.998-69.998. When the array elements are already sorted, there is no much need for to move elements to other indices, but when it is randomly selected, the number of data moves will be bigger as it has shown in the screenshot 1-2. Also, for Plot 1 & Plot 2 (this can be seen from screenshot), key comparison numbers are the same which is expected. Because, we will compare the item in the unsorted part with the all items in the sorted part. So, key comparison numbers are not affected by the distribution of the array elements.

Quick sort

When data and plot lines for the Quick sort are compared with each other, the execution time (the elapsed time) is more much larger than when the array is sorted and the first element is picked as a pivot. That means, Plot 2's growth rate of quick sort algorithm is much more faster than Plot 1. Also, it can be proven by the screen shot. The range of screen shot 1 when the elements of the array is randomly picked is 1.0841 and 8.7435; screen shot 2's is 53.3502 and 2606.33. We can say that the worst case of this algorithm correspond to the plot 2. That means, theoretical results has matched with the experimental results. In addition, for quick sort we can say that number of key comparison when the array elements are sorted has larger value. The reason is that when array is sorted, in first sort we compared $(n-1)$, then in the 2nd sort $(n-2)$ and it goes until 1(Time complexity is $O(n^2)$). However when array is randomly picked, in each partition we put the pivot in this valid index, then we sort the part which are S1 and S2. Therefore, it occurs less comparison. This lead us the fact that Plot 2 is the worst case for quick sort. To sum up, for Plot 1 quick sort seems a good fit for this case compared to others.

Merge Sort

Theoretically, merge sort algorithm's time complexity in each case is $O(nlogn)$. Plot 1 and Plot 2 is highly similar. When we consider its data, Plot 1's elapsed time ranges from 1,48691-12,3077 and Plot 2's elapsed time ranges from 0,933561-7,56874. That means, in both case merge sort algorithm's time complexity is similar. And also theoretical results has matched with the experimental results. The number of key comparisons are very similar compared to screenshot 1 and 2. So again theoretical results have matched with the experimental result. The time complexity of key comparisons are $O(nlogn)$ in each case. Also, in 2 cases, the number of moves are same.(It can be seen from screen shots.) The reason is that for instance merging two arrays of n requires 2n data moves also since merge sort uses temporary array to copy all, in total 4n data moves are required. So data moves are not affected by the distribution of the array elements. We can say that the merge sort is independent from the distribution of array elements.(it does do merge step anyway). Also, since merge uses divide and conquer algorithm, the complexity of this is better than insertion sort Plot 1.

**Also in Plot 1, For merge and quick sort are similar and we can say that merge sort's elapsed time is a little bit bigger than the quick sort. That means quick sort is a little bit faster. However, when we consider the number of comparison and data moves from the screenshot, we can see that quick sort has more comparison count and less data move count than the merge sort. Its reason could be the cost of the data moves can be more than the cost of the comparison. Also, in merge sort algorithm, additional space is needed for moving the elements so that they can be merged again.Merge is not in-place algorithm. It will also need more memory than Quick Sort.

QUESTION 3

```
ilknur — ilknur.bas@dijkstra:~ — ssh ilknur.bas@dijkstra.ug.bcc.bilkent.edu.tr...

Nearly sorted array elements k=15****
------------------------------------
Question 3 - Time analysis of Insertion Sort
Array size   Time Elapsed      compCount       moveCount
5000          0.285259          4999            34388
10000          0.571638          9999            68750
15000            0.8611         14999           103194
20000            1.1356         19999           136793
25000           1.42397         24999           171349
30000           1.71152         29999           205710
35000           1.99435         34999           239894
------------------------------------
Question 3 - Time analysis of Merge Sort
Array size   Time Elapsed      compCount       moveCount
5000           1.13495          37756          123616
10000           2.35887          80409          267232
15000           3.64943         124807          417232
20000           4.95563         170919          574464
25000            6.2655         217371          734464
30000           7.63305         264603          894464
35000           9.01927         312650         1058928
------------------------------------
Question 3 - Time analysis of Quick Sort
Array size   Time Elapsed      compCount       moveCount
5000           8.04557        1811802           44066
10000           33.4868        7724815           85105
15000           72.0687       16713045          131571
20000           128.939       29999799          174086
25000           200.497       46712537          216523
30000           291.785       68127392          259647
```

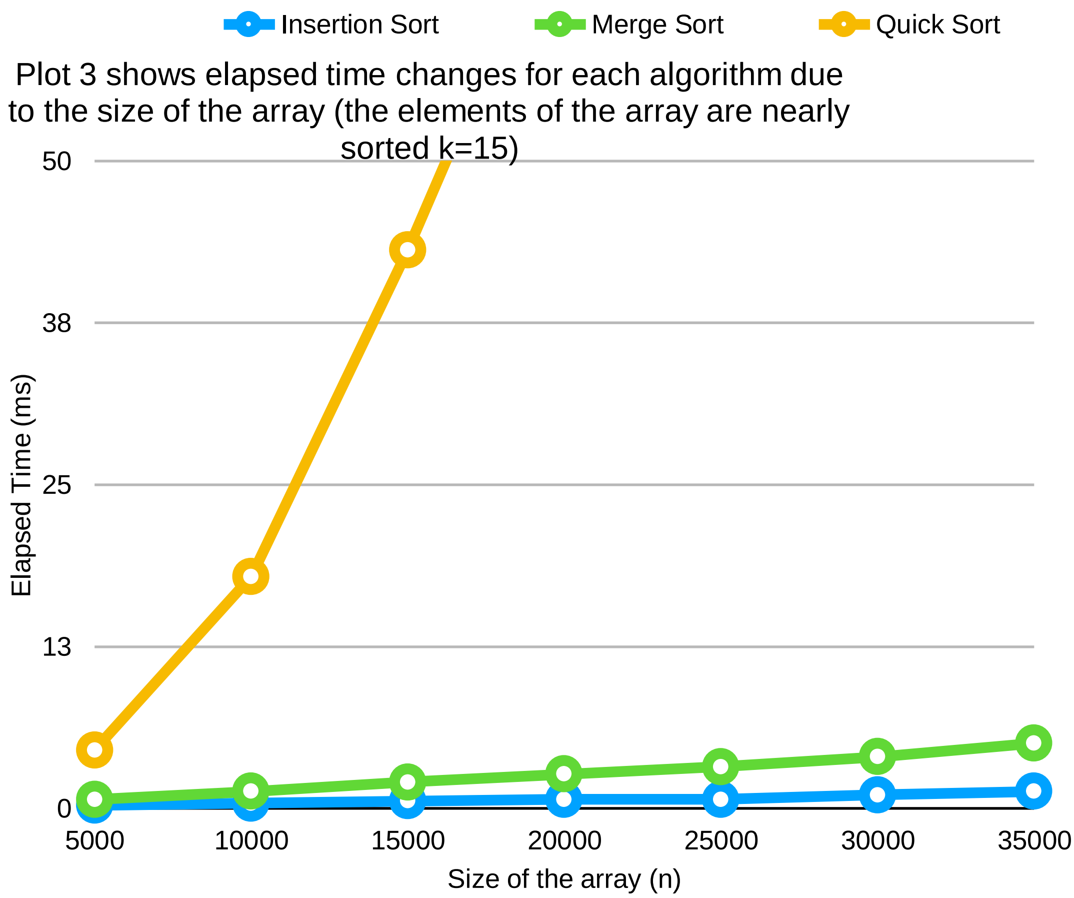
The elapsed time when the elements of the array is nearly sorted when k=15

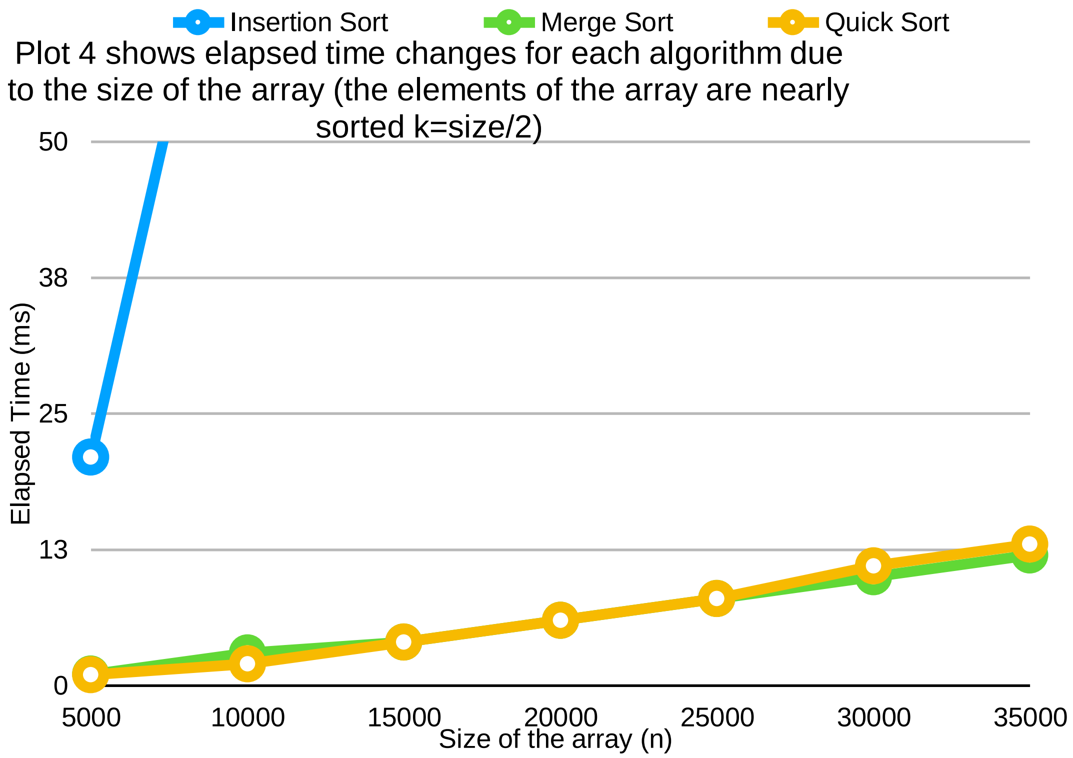
```
ilknur — ilknur.bas@dijkstra:~ — ssh ilknur.bas@dijkstra.ug.bcc.bilkent.edu.tr...
Nearly sorted array elements k=size/2****
------------------------------------
Question 3 - Time analysis of Insertion Sort
Array size   Time Elapsed      compCount       moveCount
5000            21.441           4999          3802492
10000           84.0165           9999         14904716
15000           190.473          14999         33735531
20000           336.668          19999         59696892
25000           527.839          24999         93390833
30000           759.459          29999        134495460
35000            1035.5          34999        183394400
------------------------------------
Question 3 - Time analysis of Merge Sort
Array size   Time Elapsed      compCount       moveCount
5000           1.44942          54683          123616
10000           3.06647         119678           267232
15000           4.79596         188030           417232
20000           6.55085         258829           574464
25000           8.32334         331791           734464
30000           10.1763         405981           894464
35000           12.0706         481759          1058928
------------------------------------
Question 3 - Time analysis of Quick Sort
Array size   Time Elapsed      compCount       moveCount
5000           1.13444          81906          160207
10000           2.62215         211857           373074
15000           4.81369         395029           919638
20000           6.23542         496468          1108088
25000           8.95111         752193          1764112
30000           11.1365         968586          2198647
35000           13.4461        1150884          2770435
[ilknur.bas@dijkstra ~]$
```

The elapsed time when the elements of the array is nearly sorted when k=size/2 ilknur — ilknur.bas@dijkstra:~ — ssh ilknur.bas@dijkstra.ug.bcc.bilkent.edu.tr...

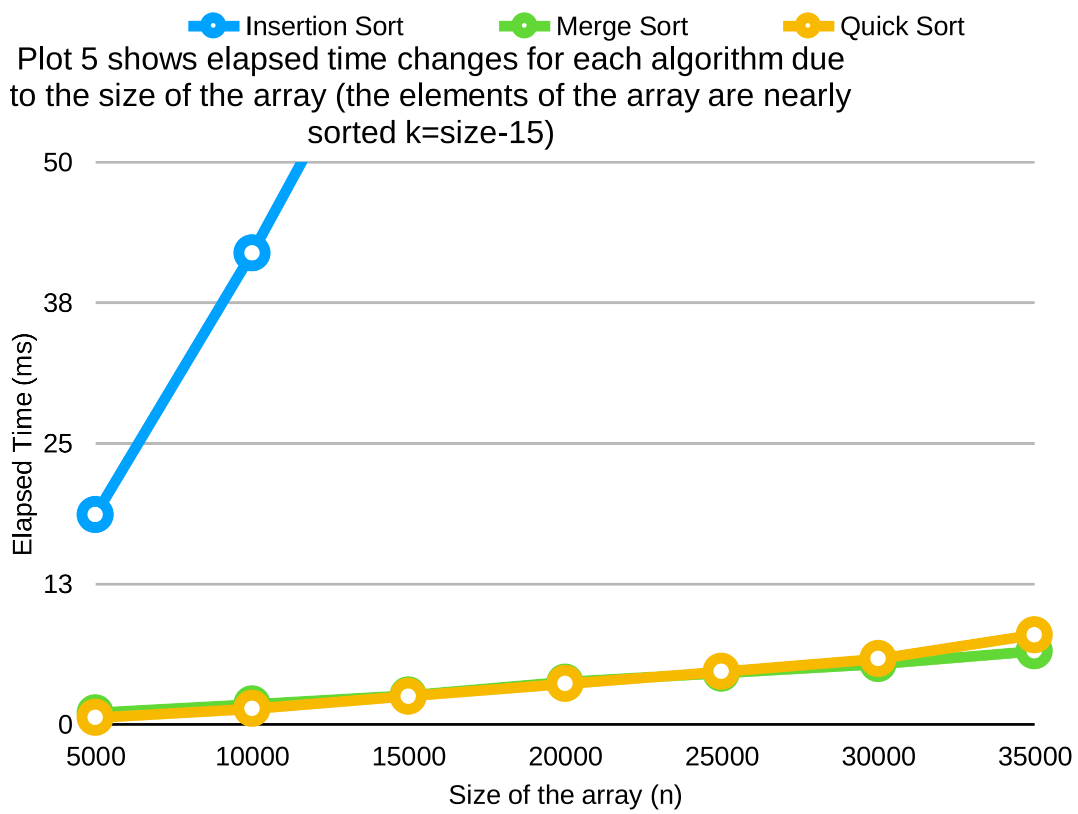
```
Nearly sorted array elements k=size-15****
------------------------------------
Question 3 - Time analysis of Insertion Sort
Array size   Time Elapsed      compCount      moveCount
5000          34.8674           4999          6200518
10000          139.758           9999         24846244
15000          317.144          14999         56197579
20000          565.992          19999        100302506
25000          889.136          24999        157487496
30000          1271.45          29999        225146393
35000          1729.13          34999        306367619
------------------------------------
Question 3 - Time analysis of Merge Sort
Array size   Time Elapsed      compCount      moveCount
5000          1.47396           55206         123616
10000          3.12768          120443          267232
15000          4.88809          189337          417232
20000          6.68183          260837          574464
25000          8.50342          334265          734464
30000          10.3713          408447          894464
35000           12.272          484627         1058928
------------------------------------
Question 3 - Time analysis of Quick Sort
Array size   Time Elapsed      compCount      moveCount
5000          1.05763           68492         123926
10000          2.29371          148988          256832
15000          3.48372          229625          365980
20000          4.87643          342258          555696
25000          6.16114          433390          659217
30000          7.58753          534738          818914
35000           9.2223          639555         1068403
```

The elapsed time when the elements of the array is nearly sorted when k=size-15





Insertion Sort
Merge Sort
Quick Sort
Plot 4 shows elapsed time changes for each algorithm due to the size of the array (the elements of the array are nearly sorted k=size/2)
50
38
25
13
0
Elapsed Time (ms)
5000
10000
15000
20000
25000
30000
35000
Size of the array (n)



Insertion Sort
Merge Sort
Quick Sort
Plot 5 shows elapsed time changes for each algorithm due to the size of the array (the elements of the array are nearly sorted k=size-15)
50
38
25
13
0
Elapsed Time (ms)
5000
10000
15000
20000
25000
30000
35000
Size of the array (n)

Explanation

When k=15

In the 3rd plot, the elements are at very close distance between their real indices when they're sorted. Again quick sort is very slow compared to other cases when the value of k is changed. It can be also seen from the screen shot, elapsed time is higher. Since the number of comparison can determine the time complexity, quick sort's is much more compared to insertion and merge. From Plot 3, we can say that insertion sort fits perfectly for this case. Its elapsed time is less.Theoretically, it is also true. If we compared the data moves for insertion sort in Plot 2 and Plot 3, we see that it is very similar. And this proves that its time complexity is $O(n)$, it is more like a best case scenario. For the merge sort algorithm, when we compare all plots (3-4 and 5), it seems the changes are not drastically different, actually pretty similar. So, again we can say that merge sort doesn't affected by the distribution of the elements of the array. Theoretically, it is also true. Because, merge sort's time complexity is $O(nlogn)$ for best, worst and average case.

When k=size/2

In the 4th plot, when we consider insertion sort, elapsed time decrease compared to when k= size-15. Because sorting elements become easier. We can consider this as a average case for insertion sort. Again we don't see many changes in merge sort compared to k=size-15 and k=15, so it can be said that it doesn't affected by the distribution of the elements of the array. For quick sort, the elapsed time values for Plot 4 are similar to Plot 5. If we consider the case when k=15 as quick sort's worst algorithm, since the elapsed times are similar when k=size-15 and k=size/2, we can say that these are the best and average cases.

When k=size-15

In this situation, for insertion sort, the number of data moves increased compared to k=15. The reason is that in this case, the elements of the array is much far from their sorted indices. Also, time elapsed is increased, that means when k=size-15 insertion is slow, compared to when k=15. Again we don't see many changes in merge sort when k=size-15. So it doesn't affected by the distribution of the elements of the array. Quick sort's elapsed time is less compared to when k=15. The reason is that quick sort is worst when array elements are sorted. For this case, we can say that its time complexity is better compared to k=15. Since, number of comparisons are less in Plot 5, that supports that the fact that k=size-15 is more close to best case for quick sort. Also, in Plot 5, plots for merge and quick sort are similar and we can say that merge sort's elapsed time is a little bit bigger than the quick sort. However, the number of key comparison of quick sort is larger than merge, and data moves are less than merge. Its reason is that for merge sort, additional space is needed.

To sum up, the theoretical and empirical results are similar. The efficiency of time complexities of sort algorithms changes according to the cases. That means for each case different algorithms should use in order to achieve best efficiency. For the merge sort, the distribution of the elements of the array doesn't affect the time complexity, it is always $O(nlogn)$. For insertion worst/average is $O(n^2)$,best is$O(n)$. Quick sort, worst case is $O(n^2)$, best and average$O(nlogn)$. Since in merge sort algorithm, additional space is needed for moving the elements, this is also an important factor for choosing the best algorithm.

The specifications (processor, RAM, operating system etc.) of the computer that I have used for this assignment.

Processor : 2 GHz Intel Core i5
RAM : 8 GB 1867 MHz LPDDR3
Operating system : macOS Sierra