CS 202 - 03
Barış Tan Ünal
22003617

ASSIGNMENT 1: ALGORITHM EFFICIENCY AND SORTING

Question 1.a:

Show that $f(n) = 8n^4 + 5n^3 + 7$ is $O(n^5)$ by specifying appropriate c and n_0 values in Big-O definition.

- $f(n)$ is $O(n^5)$ if $f(n) \leq c * n^5$ for some value $n_0 \leq n$.
- If $8n^4 + 5n^3 + 7 \leq c * n^5$ satisfies; then $\frac{8}{n} + \frac{5}{n^2} + \frac{7}{n^5} \leq c$.
- Therefore, choosing $n_0 = 4$ where $n_0 \leq n$ and choosing $c = 3$ holds for our big-Oh notation $8n^4 + 5n^3 + 7$ is $O(n^5)$.

Question 1.b:

<u>Selection Sort:</u>

There will be an index called lastSorted which will start at index -1. We will traverse the array and find the minimum element and swap it with the $(lastSorted + 1)^{th}$ element while updating lastSorted value. Sequence of traversals should stop when the lastSorted element equals to (arraySize – 1).

Initial state of array:
{ 22, 8, 49, 25, 18, 30, 20, 15, 35, 27 }
lastSorted = -1;
1st traversal:
  min = 8;
  swap (22, 8);

{ 8, 22, 49, 25, 18, 30, 20, 15, 35, 27 }
lastSorted = 0;
2nd traversal:
  min = 15;
  swap (22, 15);

{ 8, 15, 49, 25, 18, 30, 20, 22, 35, 27 }
lastSorted = 1;
3rd traversal:
    min = 18;
    swap (49, 18);

{ 8, 15, 18, 25, 49, 30, 20, 22, 35, 27 }
lastSorted = 2;
4th traversal:
    min = 20;
    swap (25, 20);

{ 8, 15, 18, 20, 25, 30, 49, 22, 35, 27 }
lastSorted = 3;
5th traversal:
    min = 22;
    swap (25, 22);

{ 8, 15, 18, 20, 22, 30, 49, 25, 35, 27 }
lastSorted = 4;
6th traversal:
    min = 25;
    swap (30, 25);

{ 8, 15, 18, 20, 22, 25, 49, 30, 35, 27 }
lastSorted = 5;
7th traversal:
    min = 27;
    swap (49, 27);

{ 8, 15, 18, 20, 22, 25, 27, 30, 35, 49 }
lastSorted = 6;
8th traversal:
    min = 30;
    swap (30, 30); // no swap

{ 8, 15, 18, 20, 22, 25, 27, 30, 35, 49 }
lastSorted = 7;
9th traversal:
    min = 35;
    swap (35, 35); // no swap

{ 8, 15, 18, 20, 22, 25, 27, 30, 35, 49 }
lastSorted = 8;
10th traversal:
    min = 35;
    swap (35, 35); // no swap

{ 8, 15, 18, 20, 22, 25, 27, 30, 35, 49 }
lastSorted = 9;
(lastSorted == size – 1) is true; no more traversals.

Bubble Sort:

There will be a counter called pass, which counts the number of traversals in order to prevent redundant traversals, and two indexes called cur and next, which points to two successive indexes and traverses the array from 0 to (arraySize – pass). If the element sitting at the cur index is greater than the element at the next index, two elements are swapped. Whenever a traversal without any swap operation is performed, bubble sort algorithm stops.

Initial state of array:
{ 22, 8, 49, 25, 18, 30, 20, 15, 35, 27 }
pass = 1;
1st traversal:
    swap (22, 8);
    swap (49, 25);
    swap (49,18);
    swap (49, 30);
    swap (49, 20);
    swap (49, 15);
    swap (49, 35);
    swap (49, 27);

{ 8, 22, 25, 18, 30, 20, 15, 35, 27, 49 }
pass = 2;
2nd traversal:
    swap (25, 18);
    swap (30, 20);
    swap (30, 15);
    swap (35, 27);

{ 8, 22, 18, 25, 20, 15, 30, 27, 35, 49 }
pass = 3;
3rd traversal:

    swap (22, 18);
    swap (25, 20);
    swap (25, 15);
    swap (30, 27);

{ 8, 18, 22, 20, 15, 25, 27, 30, 35, 49 }
pass = 4;
4th traversal:

    swap (22, 20);
    swap (22, 15);

{ 8, 18, 20, 15, 22, 25, 27, 30, 35, 49 }
pass = 5;
5th traversal:

    swap (20, 15);

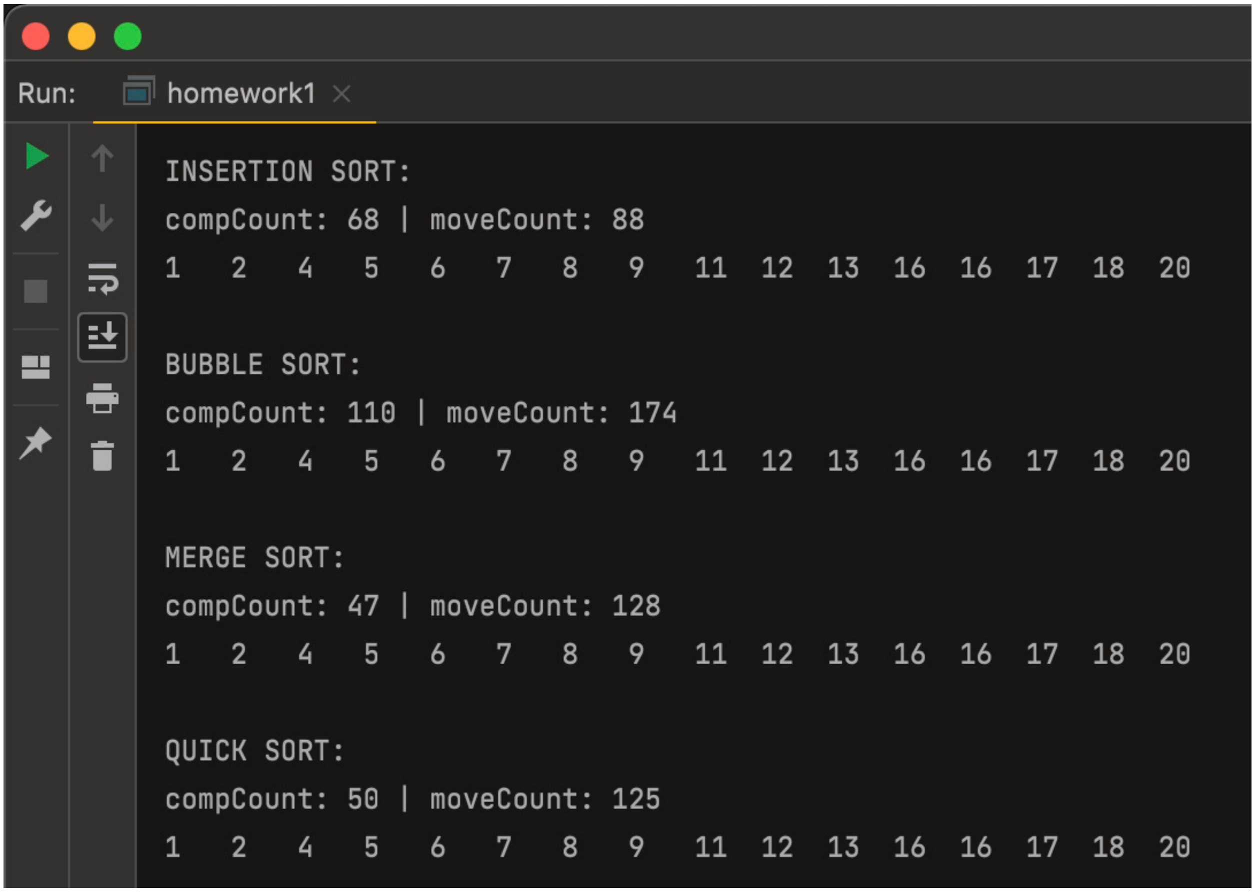
{ 8, 18, 15, 20, 22, 25, 27, 30, 35, 49 }
pass = 6;
6th traversal:

    swap (18, 15);

{ 8, 15, 18, 20, 22, 25, 27, 30, 35, 49 }
pass = 7;
7th traversal:

    // no swaps
    sorted = true;

Question 2.c:


Run:
homework1
INSERTION SORT:
compCount: 68 | moveCount: 88
1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20
BUBBLE SORT:
compCount: 110 | moveCount: 174
1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20
MERGE SORT:
compCount: 47 | moveCount: 128
1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20
QUICK SORT:
compCount: 50 | moveCount: 125
1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20

Question 2.d:

Run: homework1

```
-----------------------------------------------------
ANALYSIS OF INSERTION SORT

Random Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            19.472ms        6152786         6157796
10000           69.018ms        24794443        24804453
15000           154.631ms       55738423        55753433
20000           270.208ms       99060813        99080823
25000           425.089ms       155135646       155160656
30000           609.248ms       224391307       224421317
35000           823.086ms       305325073       305360083
40000           1071.5ms        398152868       398192878

Almost Sorted Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            2.225ms         813026          818028
10000           8.4ms           3118616         3128618
15000           18.94ms     6990154         7005154
20000           32.634ms        12080752        12100754
25000           51ms        18808870        18833870
30000           75.972ms        28404740        28434740
35000           112.371ms       38747576        38782578
40000           131.403ms       48944186        48984186

Almost Unsorted Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            31.612ms        11694440        11699468
10000           133.641ms       46896333        46906378
15000           283.771ms       105532309       105547342
20000           502.02ms        187949210       187969242
25000           793.065ms       293728574       293753626
30000           1132.41ms       421640223       421670256
35000           1541.51ms       573804890       573839918
40000           1945.73ms       751115761       751155810
```

Run: homework1

```
---------------------------------------------------------
ANALYSIS OF BUBBLE SORT

Random Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            69.857ms        12491395        18443394
10000           313.676ms       49991840        74353365
15000           716.563ms       112489725       167170305
20000           1299.66ms       199975122       297122475
25000           2055.24ms       312479499       465331974
30000           2993.64ms       449978097       673083957
35000           4089.26ms       612367540       915870255
40000           5788.05ms       799906464       1194338640

Almost Sorted Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            38.414ms        12491059        2424090
10000           147.838ms       49988559        9325860
15000           335.403ms       112466849       20925468
20000           594.868ms       199889872       36182268
25000           929.068ms       312155795       56351616
30000           1364.52ms       449507247       85124226
35000           1814.46ms       611916984       116137740
40000           2388.89ms       799848672       146712564

Almost Unsorted Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            77.591ms        12497499        35068410
10000           323.844ms       49995000        140659140
15000           700.391ms       112492499       316552032
20000           1221.49ms       199990000       563787732
25000           1902.98ms       312487500       881110884
30000           2728.08ms       449985000       1264830774
35000           3764.64ms       612482499       1721309760
40000           4838.52ms       799980000       -2041739860
```

Run: homework1

ANALYSIS OF MERGE SORT

Random Arrays:

Array Size	Elapsed Time	compCount	moveCount
5000	0.774ms	55257	123616
10000	1.622ms	120524	267232
15000	2.39ms	189342	417232
20000	3.076ms	260990	574464
25000	3.875ms	334092	734464
30000	4.823ms	408647	894464
35000	5.672ms	484570	1058928
40000	6.494ms	561822	1228928

Almost Sorted Arrays:

Array Size	Elapsed Time	compCount	moveCount
5000	0.502ms	51468	123616
10000	1.01ms	111262	267232
15000	1.581ms	175628	417232
20000	2.159ms	243209	574464
25000	2.826ms	311776	734464
30000	3.366ms	379446	894464
35000	3.836ms	451287	1058928
40000	4.546ms	526342	1228928

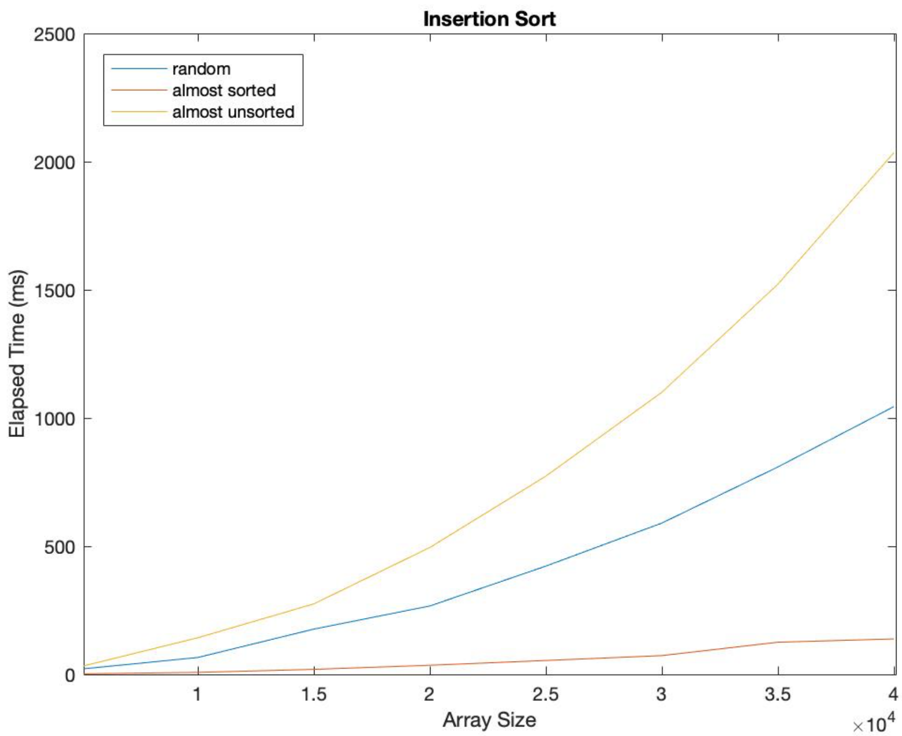
Almost Unsorted Arrays:

Array Size	Elapsed Time	compCount	moveCount
5000	0.5ms	48263	123616
10000	1.092ms	108051	267232
15000	1.552ms	171265	417232
20000	2.103ms	236310	574464
25000	2.7ms	305159	734464
30000	3.314ms	374112	894464
35000	3.97ms	441870	1058928
40000	4.393ms	512192	1228928

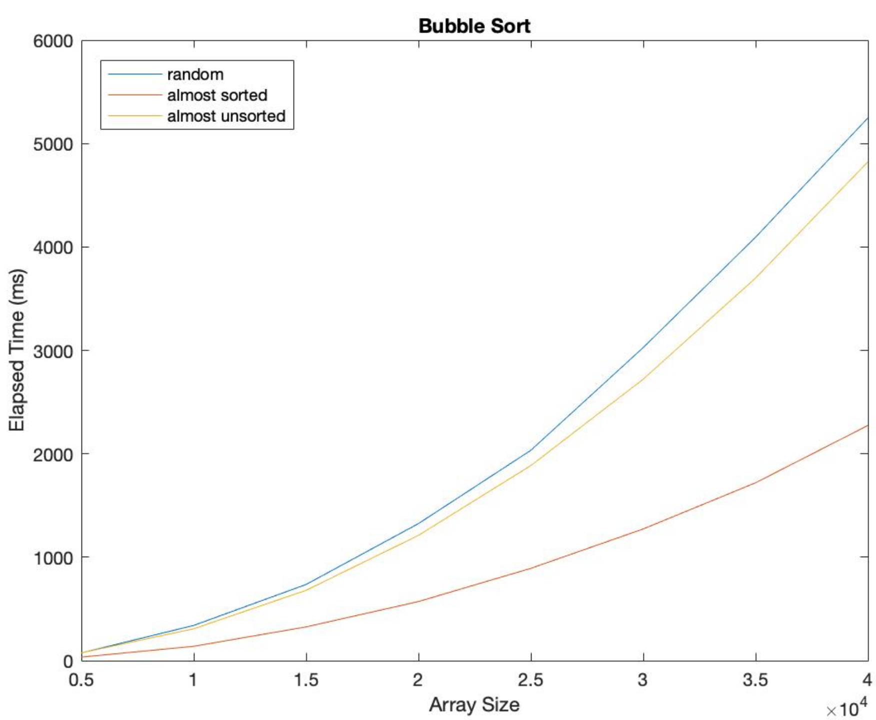
Run: homework1

```
40000           4.393ms         512192          1228928

-----------------------------------------------------
ANALYSIS OF QUICK SORT

Random Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            0.527ms         72564           99503
10000           1.095ms         169497          208830
15000           1.667ms         295960          306308
20000           2.446ms         439544          397006
25000           3.382ms         647901          505758
30000           3.918ms         840689          644690
35000           5.015ms         1054499         740965
40000           5.768ms         1351715         1030530

Almost Sorted Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            0.689ms         179939          196117
10000           1.71ms          461039          459810
15000           2.888ms         810160          795682
20000           3.872ms         1049737         1219863
25000           5.203ms         1682810         1268116
30000           5.676ms         1712424         1430410
35000           6.53ms          1812192         1972086
40000           9.381ms         2903172         2594186

Almost Unsorted Arrays:
Array Size      Elapsed Time    compCount       moveCount
5000            0.578ms         114189          193174
10000           1.145ms         232063          385950
15000           1.947ms         396339          670522
20000           2.531ms         547933          902016
25000           4.15ms          915332          1498110
30000           4.907ms         1080027         1725220
35000           4.474ms         950816          1578560
40000           8.135ms     1919505         3096569
```

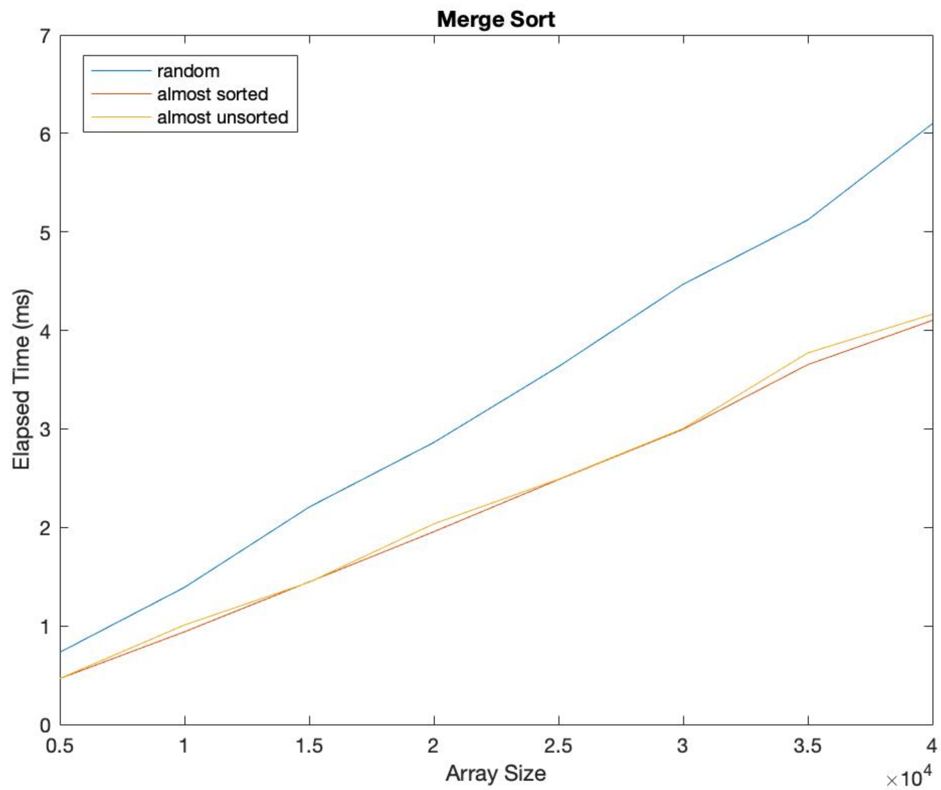
Question 3:

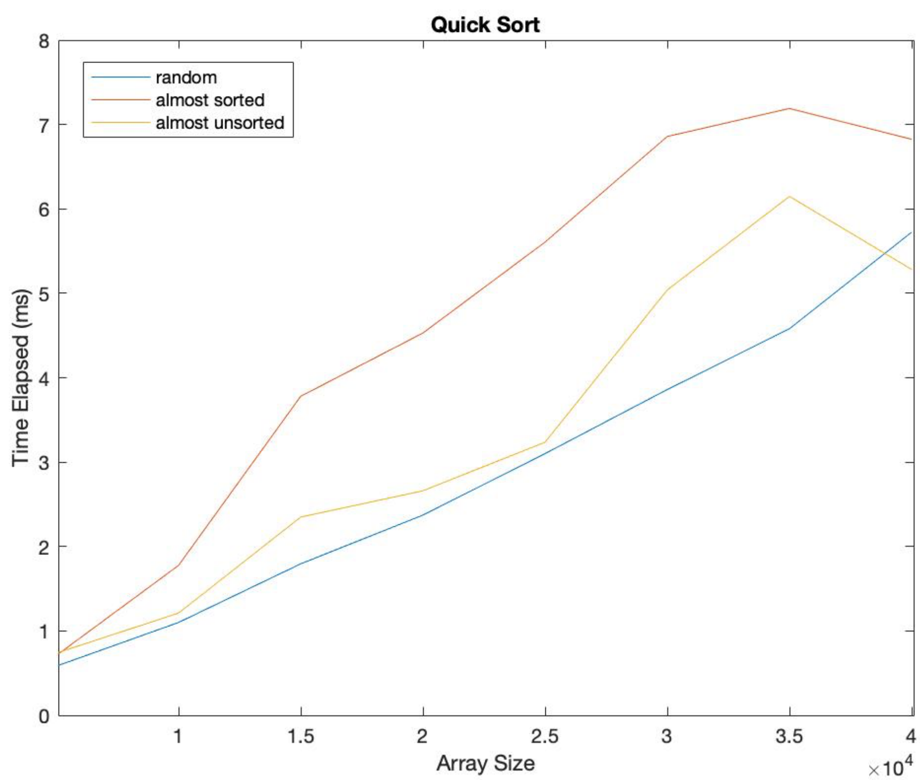


Insertion Sort
random
almost sorted
almost unsorted
Elapsed Time (ms)
0
500
1000
1500
2000
2500
1
1.5
2
2.5
3
3.5
4
Array Size
$\times 10^4$



Bubble Sort
random
almost sorted
almost unsorted
Elapsed Time (ms)
0
1000
2000
3000
4000
5000
6000
0.5
1
1.5
2
2.5
3
3.5
4
Array Size
$\times 10^4$


Merge Sort
random
almost sorted
almost unsorted
Elapsed Time (ms)
0
1
2
3
4
5
6
7
0.5
1
1.5
2
2.5
3
3.5
4
Array Size
×10^4


Quick Sort
random
almost sorted
almost unsorted
Time Elapsed (ms)
0
1
2
3
4
5
6
7
8
1
1.5
2
2.5
3
3.5
4
Array Size
×10^4

Question 3. Conclusion:

Ideally, **insertion sort** should be Ω(n) for best case. Our experimental result is matching the theoretical value as shown in the first graph with a red line which is nearly linear. The theoretical complexity in average case is $\theta(n^2)$, which roughly satisfies the experimental result as shown with the blue line. The reason behind the line being a bit more linear compared to a n^2 parabola is the lack of sample size, or size of the arrays in other words. If this experiment was conducted with a sample size of 20 for instance – maximum array size being 100,000 – the n^2 parabola would have been more visible. When it comes to worst case which is represented by the almost unsorted array line which is yellow, it is much more accurate to the theoretical complexity of $O(n^2)$ compared to best and average case, since insertion sorting an almost unsorted array requires significantly more work, hence more time. The ideal complexity is n^2 for average and worst case because there is a nested for loop in the insertion sort algorithm. On the other hand, if we have a completely sorted array, only one iteration is sufficient.

For **bubble sort**, both worst and average case should be $O(n^2)$ and $\theta(n^2)$ respectively. As shown in the graph above, blue representing the average case and yellow representing the worst case are in fact increasing approximately quadratically, meeting our expectations. When it comes to the best case, we expect the complexity to be Ω(n). Although the red line on the graph is not absolutely linear, one should keep in mind that the sample array is not completely sorted but the 10% of it was altered, meaning we could not expect the line to be perfectly linear, but a bit more parabola-like. The reason behind worst- and best-case complexities generating different lines is the bubble sort algorithm, which has a nested for loops. In the best case only one traversal is sufficient, however in the case of the worst case, we should make sure that the biggest value of the unsorted part is transferred to the end of the array, hence the sorted part. This operation requires n traversals at most.

Merge sort algorithm should have the same time complexities $O(n * \log(n))$, $\theta(n * \log(n))$, and $\Omega(n * \log(n))$ for worst, average, and best cases. $y = n(\log(n))$ is nearly a linear line, which justifies the graphs above, all of which are practically linear. The time required for the merge sort algorithm, which outperforms bubble and insertion sort algorithms is significantly less due to its efficiency. Its complexity is $O(* \log(n))$, because *mergeSort* function recursively calls itself for both halves, dividing itself to two sections every time it is called. This corresponds to $\log(n)$ times for number of *mergeSort* function calls. There is a helper function, called *merge*, which iterates through every sub array it was given, corresponding to n number of iterations.

The case for **quick sort** is more complicated. We should expect time complexities of $\Omega(n * \log(n))$ and $\theta(n * \log(n))$ for best and average cases, whereas $O(n^2)$ for worst case. This is because *quickSort* function again recursively calls itself, just as *mergeSort*. However, this time the hard work is done before the recursive call, meaning the *partition* function is called before the recursive call. For the worst case, the pivot input could be the greatest or the least element of the array every time it was recursively called. This is possible with a completely sorted or completely unsorted array. Although, the graph is right with showing almost sorted and unsorted arrays taking more time than a random array; the red and yellow lines does not match our time complexity expectations completely. The justification for it could be the lack of sample size, therefore the arrays size. The sorting operation only take a few milliseconds which is not adequate for an accurate graph. For the best case, the value of the pivot index should be as close as it can be to the median of the array for each recursive call which would result as $(\log(n))$ calls. Again, partition is iterating over the array which corresponds to n. Blue line which represents average case is more accurate since it is close to a $(n * \log(n))$ line.