CS 202

Homework 1

Sec 03

Kutay Tire

22001787

08.03.2022

TA: Berat Biçer

Instructor: Aynur Dayanık

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.

Solution:

In order to solve the problem, some value of c and n_0 have to be found such *that cf(n) >= T(n)* for all n >= n_0 where $T(n) = 8n^4 + 5n^3 + 7$.

$cn^5 \geq 8n^4 + 5n^3 + 7$ dividing by n^5, inequality becomes

$c \geq \frac{8}{n} + \frac{5}{n^2} + \frac{7}{n^5}$

Choosing c = 20 and $n_0 = 1$ will give the answer since $20n^5 \geq 8n^4 + 5n^3 + 7 \; for \; all \; n \geq 1$

b.) Trace the following sorting algorithms to sort the array [22, 8, 49, 25, 18, 30, 20, 15, 35, 27] in ascending order. Use the array implementation of the algorithms as described in the textbook and show all major steps

Solution:

Selection Sort

In selection sort, the largest element will be found from the unsorted sub-list in each step. After that, the largest element will be added at the last index of the array becoming part of the sorted sub-list. Since the given array has 10 elements, the sorting will be completed in 9 passes. Blue shaded boxes mark the selected item whereas grey sorted boxes represent sorted part of the list.

Initial Array

22	8	49	25	18	30	20	15	35	27

After 1st Swap (49 and 27 are swapped, next largest element is 35)

22	8	27	25	18	30	20	15	35	49

After 2nd Swap (35 is already in order so it remained at the same index, next largest element is 30)

22	8	27	25	18	30	20	15	35	49

After 3rd Swap (30 and 15 are swapped, next largest element is 27)

22	8	27	25	18	15	20	30	35	49

After 4th Swap (27 and 20 are swapped, next largest element is 25)

22	8	20	25	18	15	27	30	35	49

After 5th Swap (25 and 15 are swapped, next largest element is 22)

22	8	20	15	18	25	27	30	35	49

After 6th Swap (22 and 18 are swapped, next largest element is 20)

18	8	20	15	22	25	27	30	35	49

After 7th Swap (20 and 15 are swapped, next largest element is 18)

18	8	15	20	22	25	27	30	35	49

After 8th Swap (18 and 15 are swapped, next largest element is 15)

15	8	18	20	22	25	27	30	35	49

After 9th Swap (15 and 8 are swapped, there is no need for the next largest because the remaining element “8” becomes automatically the smallest in the list)

8	15	18	20	22	25	27	30	35	49

Bubble Sort

Similar to selection sort, in bubble sort the list will be divided into two sub-lists. On each pass, an element will be moved from unsorted part of the list to the sorted part. In the following steps the green shaded boxes represent pairs that are already sorted, blue shaded ones represent pairs to be swapped whereas the grey shaded ones represent the sorted elements.

Pass 1

22	8	49	25	18	30	20	15	35	27

8	22	49	25	18	30	20	15	35	27

8	22	49	25	18	30	20	15	35	27

8	22	25	49	18	30	20	15	35	27

8	22	25	18	49	30	20	15	35	27

8	22	25	18	30	49	20	15	35	27

8	22	25	18	30	20	49	15	35	27

8	22	25	18	30	20	15	49	35	27

8	22	25	18	30	20	15	35	49	27

8	22	25	18	30	20	15	35	27	49

Pass 2

8	22	25	18	30	20	15	35	27	49

8	22	25	18	30	20	15	35	27	49

8	22	25	18	30	20	15	35	27	49

8	22	18	25	30	20	15	35	27	49

8	22	18	25	30	20	15	35	27	49

8	22	18	25	20	30	15	35	27	49

8	22	18	25	20	15	30	35	27	49

8	22	18	25	20	15	30	35	27	49

8	22	18	25	20	15	30	27	35	49

Pass 3

8	22	18	25	20	15	30	27	35	49

8	22	18	25	20	15	30	27	35	49

8	18	22	25	20	15	30	27	35	49

8	18	22	25	20	15	30	27	35	49

8	18	22	20	25	15	30	27	35	49

8	18	22	20	15	25	30	27	35	49

8	18	22	20	15	25	30	27	35	49

8	18	22	20	15	25	27	30	35	49

Pass 4

8	18	22	20	15	25	27	30	35	49

8	18	22	20	15	25	27	30	35	49

8	18	22	20	15	25	27	30	35	49

8	18	20	22	15	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

Pass 5

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

Pass 6

8	18	15	20	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

Pass 7

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

Note: Normally, bubble sort ends here as the “boolean isSorted” return true. However, for clarity and analysis, following steps are also showed.

Pass 8

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

Pass 9

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

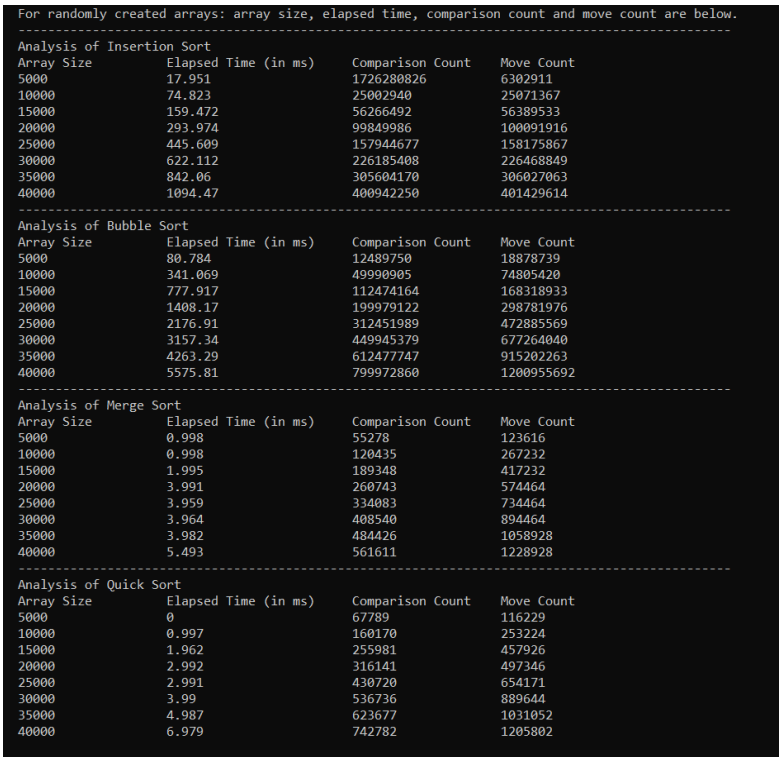
Question 2.)

c.) In part c of the question 2, the implemented sorting methods are called for the given array {9, 6, 7, 16, 18, 5, 2, 12, 20, 1, 16, 17, 4, 11, 13, 8}. The same array was used for every sorting algorithm and the resulting comparison counts and key movements are recorded. Naturally, the result of each sort in terms of the place of the elements became same while they differed in number of movements and comparisons. Below, there is the output of the program.

```
Insertion Sort:
Number of key comparisons: 69
Number of moves: 88
Result of sort: The arrary is: {1,2,4,5,6,7,8,9,11,12,13,16,16,17,18,20}

Bubble Sort:
Number of key comparisons: 110
Number of moves: 174
Result of sort: The arrary is: {1,2,4,5,6,7,8,9,11,12,13,16,16,17,18,20}

Merge Sort:
Number of key comparisons: 47
Number of moves: 128
Result of sort: The arrary is: {1,2,4,5,6,7,8,9,11,12,13,16,16,17,18,20}

Quick Sort:
Number of key comparisons: 50
Number of moves: 125
Result of sort: The arrary is: {1,2,4,5,6,7,8,9,11,12,13,16,16,17,18,20}
```

Figure 1.

d.) In part d, the results of each sort were taken for randomly created, almost sorted and almost unsorted arrays. Although the general results were got as expected, there were some little deviations, too. Below, figure 2 shows the results of the data for randomly created arrays, figure 3 shows the data for almost sorted arrays and the figure 4 shows the data for almost unsorted arrays. Note that because the bubble sort requires too many movements, there is a overflow for bubble sort in figure 4 for the array size of 40000.



For randomly created arrays: array size, elapsed time, comparison count and move count are below.

Analysis of Insertion Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	17.951	1726280826	6302911
10000	74.823	25002940	25071367
15000	159.472	56266492	56389533
20000	293.974	99849986	100091916
25000	445.609	157944677	158175867
30000	622.112	226185408	226468849
35000	842.06	305604170	306027063
40000	1094.47	400942250	401429614

Analysis of Bubble Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	80.784	12489750	18878739
10000	341.069	49990905	74805420
15000	777.917	112474164	168318933
20000	1408.17	199979122	298781976
25000	2176.91	312451989	472885569
30000	3157.34	449945379	677264040
35000	4263.29	612477747	915202263
40000	5575.81	799972860	1200955692

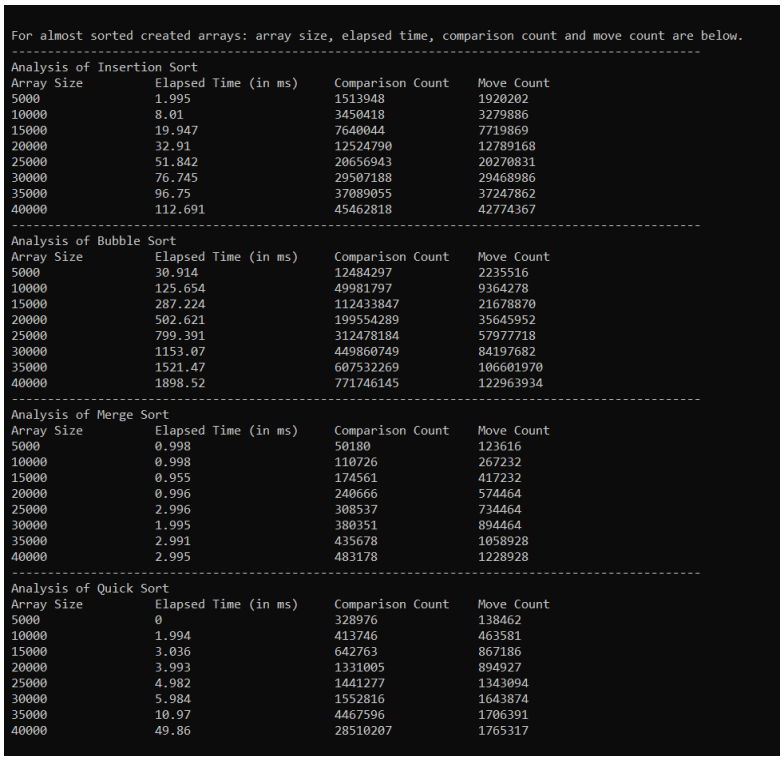
Analysis of Merge Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	0.998	55278	123616
10000	0.998	120435	267232
15000	1.995	189348	417232
20000	3.991	260743	574464
25000	3.959	334083	734464
30000	3.964	408540	894464
35000	3.982	484426	1058928
40000	5.493	561611	1228928

Analysis of Quick Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	0	67789	116229
10000	0.997	160170	253224
15000	1.962	255981	457926
20000	2.992	316141	497346
25000	2.991	430720	654171
30000	3.99	536736	889644
35000	4.987	623677	1031052
40000	6.979	742782	1205802

Figure 2.



For almost sorted created arrays: array size, elapsed time, comparison count and move count are below.

Analysis of Insertion Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	1.995	1513948	1920202
10000	8.01	3450418	3279886
15000	19.947	7640044	7719869
20000	32.91	12524790	12789168
25000	51.842	20656943	20270831
30000	76.745	29507188	29468986
35000	96.75	37089055	37247862
40000	112.691	45462818	42774367

Analysis of Bubble Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	30.914	12484297	2235516
10000	125.654	49981797	9364278
15000	287.224	112433847	21678870
20000	502.621	199554289	35645952
25000	799.391	312478184	57977718
30000	1153.07	449860749	84197682
35000	1521.47	607532269	106601970
40000	1898.52	771746145	122963934

Analysis of Merge Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	0.998	50180	123616
10000	0.998	110726	267232
15000	0.955	174561	417232
20000	0.996	240666	574464
25000	2.996	308537	734464
30000	1.995	380351	894464
35000	2.991	435678	1058928
40000	2.995	483178	1228928

Analysis of Quick Sort

Array Size	Elapsed Time (in ms)	Comparison Count	Move Count
5000	0	328976	138462
10000	1.994	413746	463581
15000	3.036	642763	867186
20000	3.993	1331005	894927
25000	4.982	1441277	1343094
30000	5.984	1552816	1643874
35000	10.97	4467596	1706391
40000	49.86	28510207	1765317

Figure 3.

```
For almost unsorted created arrays: array size, elapsed time, comparison count and move count are below.
-------------------------------------------------------------------------------------------------
Analysis of Insertion Sort
Array Size          Elapsed Time (in ms)     Comparison Count    Move Count
5000                31.913                   40232477            13497583
10000               126.642                  46981600            47068835
15000               285.232                  105440196           105588105
20000               515.587                  189198544           189471206
25000               800.77                   294222490           294637391
30000               1146.58                  422741508           423344871
35000               1584.72                  578278205           578967414
40000               2075.78                  764171697           766260325
-------------------------------------------------------------------------------------------------
Analysis of Bubble Sort
Array Size          Elapsed Time (in ms)     Comparison Count    Move Count
5000                89.798                   12497500            35166804
10000               360.01                   49995000            140606970
15000               814.8                    112492500           315672162
20000               1441.43                  199990000           566442864
25000               2254.13                  312487500           880733148
30000               3246.78                  449985000           1265733474
35000               4401.37                  612482500           1731914202
40000               5776.25                  799980000           -2013954490
-------------------------------------------------------------------------------------------------
Analysis of Merge Sort
Array Size          Elapsed Time (in ms)     Comparison Count    Move Count
5000                0                        49193               123616
10000               0                        108088              267232
15000               0.998                    172782              417232
20000               1.995                    236765              574464
25000               1.995                    307248              734464
30000               2.992                    376200              894464
35000               2.993                    434403              1058928
40000               2.992                    484291              1228928
-------------------------------------------------------------------------------------------------
Analysis of Quick Sort
Array Size          Elapsed Time (in ms)     Comparison Count    Move Count
5000                0.997                    112608              179847
10000               1.994                    216142              334053
15000               1.994                    384256              616920
20000               2.993                    644774              1009677
25000               3.989                    830350              1373715
30000               4.986                    973471              1592682
35000               16.953                   3834095             5842725
40000               124.658                  28672404            43098807
```

Figure 4.

Question 3.)

Below are the graphs for each sort type with randomly, almost sorted and almost unsorted arrays.

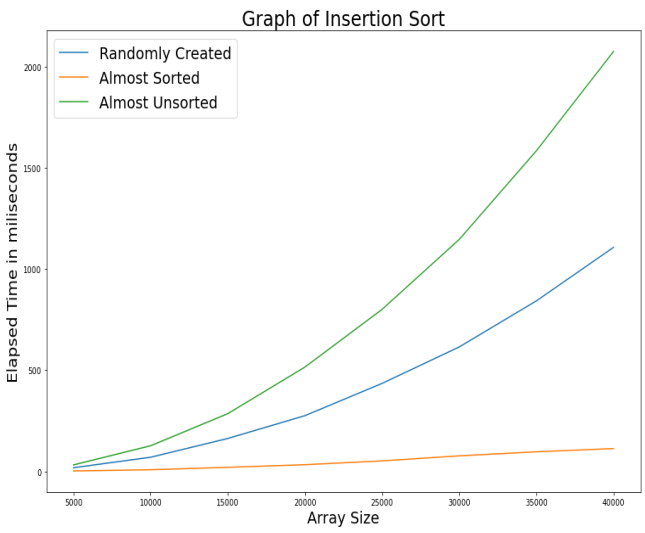


Figure 5.

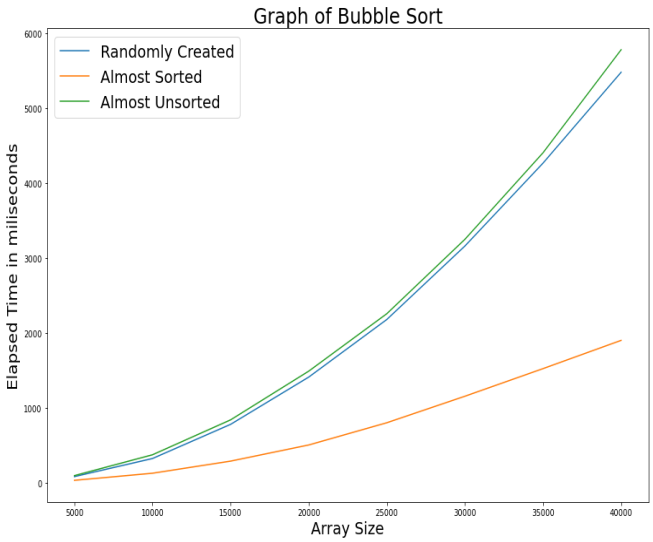


Figure 6.

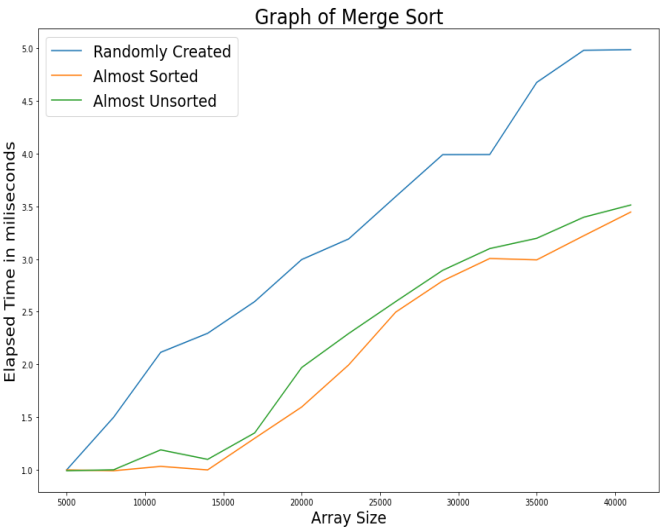


Figure 7.

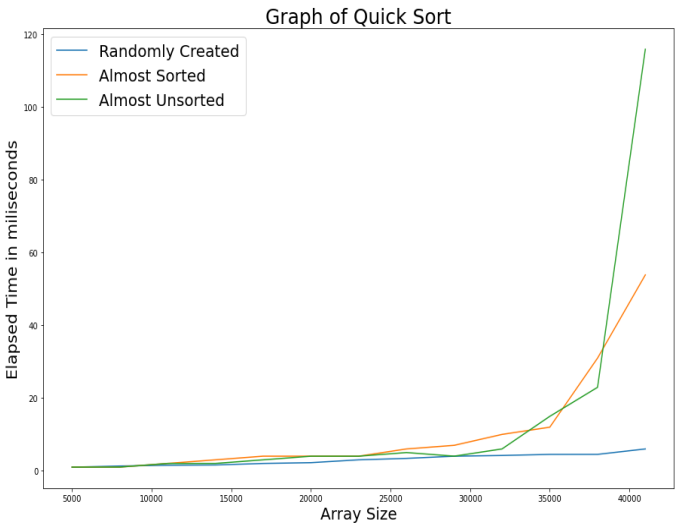


Figure 8.

Analysis of Data

Insertion Sort: For the insertion sort, the empirical results and the theoretical ones were very similar. The best case and the worst case of insertion sort happens when the array is sorted and in reverse order respectively. For sorted arrays, time complexity becomes O(n) and it becomes $O(n^2)$ for the arrays in reverse order. These results correlate with figure 5 as almost unsorted array shows a quadratic growth while almost sorted array shows a growth rate between O(n) and $O(n^2)$, being closer to linear growth. As expected, randomly created arrays also show a growth rate of $O(n^2)$ which is the average case of insertion sort. The reason why almost unsorted arrays function in $O(n^2)$ is that the array needs to be shifted at each pass as the larger values go to the end and this takes much more time.

Bubble Sort: Similar to insertion sort, best case happens when the array is almost sorted and worst case happens when the array is in reverse order. Although the theoretical growth rate of almost sorted arrays matches with the experimental results with being between O(n) and $O(n^2)$, there is a deviation from theoretical values for randomly created arrays because they are nearly same with the almost unsorted ones. This might be due to the fact that randomly created arrays also show a similar pattern with the almost unsorted arrays that explains the deviation. Still, both of them show a growth rate of $O(n^2)$ as expected. In bubble sort, best case is O(n) as the inner "for loop" is skipped when the array is sorted in order which makes the function dependent only on "n".

Merge Sort: In merge sort, time complexity for worst case, average case and best case is O(nlogn). Following this complexity, experimental values actually match with the theoretical ones as figure 7 shows that all three types of arrays show an almost linear growth rate since logn is too small to contribute significantly. Although their general time complexities are the same, sorting randomly created arrays takes more time because they have more key comparisons. On the other hand, both almost sorted and almost unsorted arrays are considered to be in the best case as they require fewer key comparisons and less time. Figure

7 proves this because the elapsed times of almost unsorted and sorted arrays are quite similar, with being less than the randomly created array.

Quick Sort: Unlike merge sort, the time complexity of quick sort changes in the worst case. It is O(nlogn) for best and average cases whereas it becomes $O(n^2)$ for the worst case. The worst case happens when the pivot is the largest or the smallest element. As a result, almost sorted and unsorted arrays show the growth rate of $O(n^2)$ as being the worst cases. The reason is that partitioning the array takes a lot of time and movement for the worst case as the algorithm has to go through the whole array. Still, the peak points in figure 8 is a proof of the quadratic growth rates of almost sorted and unsorted arrays. Randomly created arrays, on the other hand, show a growth rate of O(nlogn) which is the correct average case. However, the theoretical results and the experimental results do not corelate much for figure 8 as the graphs could have been more quadratic for the worst cases. This is because for small values of the size, partitioning takes very little time and quick sort cannot behave correctly. As the size approaches to 30.000, more reliable results emerge.