Berk Çakar — 22003021 — CS202-3

Homework 1 – Algorithm Efficiency and Sorting

Question 1

a) By the definition of Big-O, $T(n) = O(f(n))$ if there are positive constants c and n_0 such that $T(n) \leq c * f(n)$, where $n \geq n_0$

In order to prove $f(n) = 8n^4 + 5n^3 + 7$ is $O(n^5)$, we need to find two positive constants c and n_0 which satisfy,

$$0 \leq 8n^4 + 5n^3 + 7 \leq c * n^5; \text{ for all } n \geq n_0$$

Let $c = 8$ and $n_0 = 2$

$$8n^4 + 5n^3 + 7 \leq 8 * n^5 \text{ for all } n \geq 2$$

b)

Tracing of Bubble Sort:

Unsorted Sorted

Swaps performed during each pass, respectively:

Before sorting:

22	8	49	25	18	30	20	15	35	27

At the end of pass 1:

8	22	25	18	30	20	15	35	27	49

arr[0] <-> arr[1], arr[2] <-> arr[3], arr[3] <-> arr[4], arr[4] <-> arr[5], arr[5] <-> arr[6], arr[6] <-> arr[7], arr[7] <-> arr[8], arr[8] <-> arr[9]

At the end of pass 2:

8	22	18	25	20	15	30	27	35	49

arr[2] <-> arr[3], arr[4] <-> arr[5], arr[5] <-> arr[6], arr[7] <-> arr[8],

At the end of pass 3:

8	18	22	20	15	25	27	30	35	49

arr[1] <-> arr[2], arr[3] <-> arr[4], arr[4] <-> arr[5], arr[6] <-> arr[7],

At the end of pass 4:

8	18	20	15	22	25	27	30	35	49

arr[2] <-> arr[3], arr[3] <-> arr[4]

At the end of pass 5:

8	18	15	20	22	25	27	30	35	49

arr[2] <-> arr[3]

At the end of pass 6:

8	15	18	20	22	25	27	30	35	49

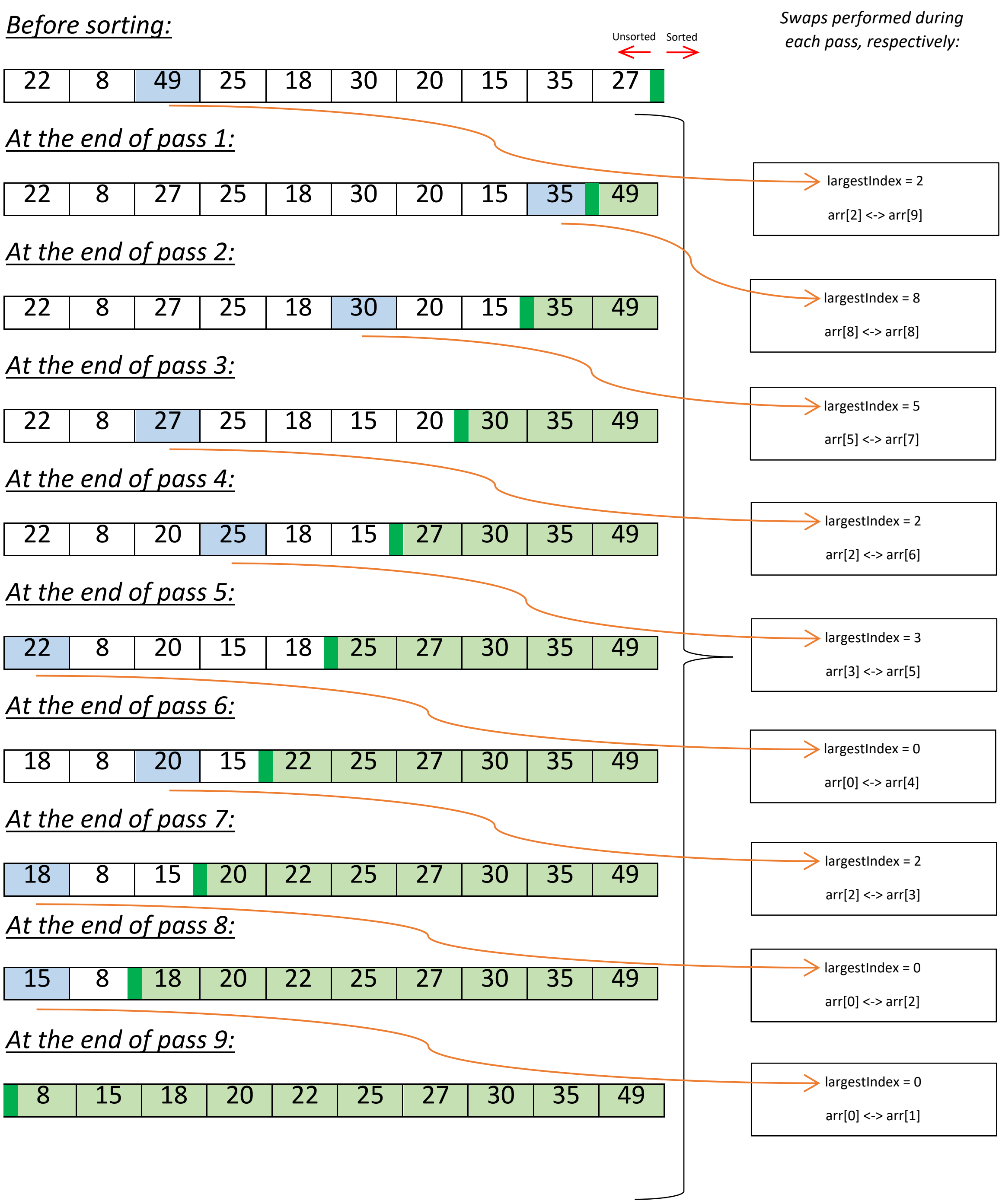
arr[1] <-> arr[2]

At the end of pass 7:

8	15	18	20	22	25	27	30	35	49

no swaps -> sorted = true;

Tracing of Selection Sort:



Indices marked as blue indicate the location of the largest element which will be used in the swap during the $n + 1^{\text{th}}$ pass.

Question 2

c)

```
[berk.cakar@dijkstra hw1]$ ls
main.cpp  Makefile  sorting.cpp  sorting.h
[berk.cakar@dijkstra hw1]$ make
g++ -Wall -Wextra -std=c++03 sorting.cpp main.cpp -o hw1
[berk.cakar@dijkstra hw1]$ ./hw1
QUESTION 2 - C

Array before sorting: 9 6 7 16 18 5 2 12 20 1 16 17 4 11 13 8

Sorting the array using insertion sort
Array after insertion sort: 1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20
Number of key comparisons: 69
Number of data moves: 88

Sorting the array using bubble sort
Array after bubble sort: 1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20
Number of key comparisons: 110
Number of data moves: 174

Sorting the array using merge sort
Array after merge sort: 1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20
Number of key comparisons: 47
Number of data moves: 128

Sorting the array using quick sort
Array after quick sort: 1 2 4 5 6 7 8 9 11 12 13 16 16 17 18 20
Number of key comparisons: 50
Number of data moves: 125
```

d)

Using arrays filled with randomly generated integers:

Analysis of Insertion Sort

Array Size	Elapsed Time	compCount	moveCount
5000	17.26 ms	6194943	6199947
10000	71.964 ms	24992842	25002847
15000	158.435 ms	56217361	56232366
20000	277.774 ms	99720056	99740062
25000	439.241 ms	155720107	155745112
30000	630.069 ms	225551438	225581446
35000	848.139 ms	306800328	306835336
40000	1132.55 ms	402085900	402125909

Analysis of Bubble Sort

Array Size	Elapsed Time	compCount	moveCount
5000	47.115 ms	12490479	18569847
10000	208.739 ms	49963875	74948547
15000	541.806 ms	112447949	168607104
20000	1032.85 ms	199986514	299100192
25000	1687.72 ms	312481284	467085342
30000	2550.47 ms	449958435	676564344
35000	3514.39 ms	612467965	920296014
40000	4617.22 ms	799947869	1206137733

Analysis of Merge Sort

Array Size	Elapsed Time	compCount	moveCount
5000	0.794 ms	55257	123616
10000	1.533 ms	120466	267232
15000	2.338 ms	189227	417232
20000	3.211 ms	260927	574464
25000	4.081 ms	334067	734464
30000	5.182 ms	408545	894464
35000	5.804 ms	484631	1058928
40000	6.887 ms	561935	1228928

Analysis of Quick Sort

Array Size	Elapsed Time	compCount	moveCount
5000	0.509 ms	73972	118160
10000	1.087 ms	153315	239732
15000	1.813 ms	252247	424167
20000	2.33 ms	337950	562115
25000	2.995 ms	415837	658906
30000	3.674 ms	529573	889948
35000	4.689 ms	626427	1023174
40000	5.595 ms	761954	1331178

Using almost sorted arrays:

Analysis of Insertion Sort

Array Size	Elapsed Time	compCount	moveCount
5000	2.117 ms	739795	744794
10000	9.082 ms	3098755	3108754
15000	20.818 ms	7021849	7036848
20000	36.16 ms	12405355	12425354
25000	58.195 ms	19958061	19983060
30000	82.458 ms	28178167	28208166
35000	107.477 ms	38088931	38123930
40000	143.39 ms	50956061	50996060

Analysis of Bubble Sort

Array Size	Elapsed Time	compCount	moveCount
5000	30.311 ms	12409929	2204388
10000	124.708 ms	49975890	9266268
15000	283.042 ms	112299990	21020550
20000	507.679 ms	199665585	37156068
25000	797.883 ms	312400764	59799186
30000	1028.75 ms	449791247	84444504
35000	1461.05 ms	612340722	114161796
40000	1972.04 ms	799788110	152748186

Analysis of Merge Sort

Array Size	Elapsed Time	compCount	moveCount
5000	0.508 ms	50573	123616
10000	1.046 ms	111750	267232
15000	1.583 ms	175068	417232
20000	2.199 ms	242945	574464
25000	2.709 ms	311440	734464
30000	3.511 ms	380102	894464
35000	4.023 ms	451294	1058928
40000	4.489 ms	527979	1228928

Analysis of Quick Sort

Array Size	Elapsed Time	compCount	moveCount
5000	0.723 ms	231083	184375
10000	1.771 ms	590779	447266
15000	2.502 ms	717449	922007
20000	3.5 ms	899801	1236897
25000	4.482 ms	1255950	1486732
30000	5.093 ms	1310738	2033400
35000	6.609 ms	2413149	1825273
40000	8.878 ms	3182065	2295726

Using almost unsorted arrays:

Analysis of Insertion Sort

Array Size	Elapsed Time	compCount	moveCount
5000	33.251 ms	11732227	11737240
10000	132.536 ms	46879727	46889746
15000	296.871 ms	105467401	105482424
20000	528.136 ms	187826491	187846512
25000	777.721 ms	293770871	293795918
30000	1132.54 ms	422394649	422424656
35000	1509.55 ms	573787009	573822048
40000	1955.17 ms	750489391	750529416

Analysis of Bubble Sort

Array Size	Elapsed Time	compCount	moveCount
5000	63.695 ms	12497500	35181726
10000	252.199 ms	49995000	140609244
15000	556.338 ms	112492500	316357278
20000	1011.13 ms	199989999	563419542
25000	1528.06 ms	312487500	881237760
30000	2196.93 ms	449985000	1267093974
35000	3019.99 ms	612482500	1721256150
40000	4063.81 ms	799980000	2147483647

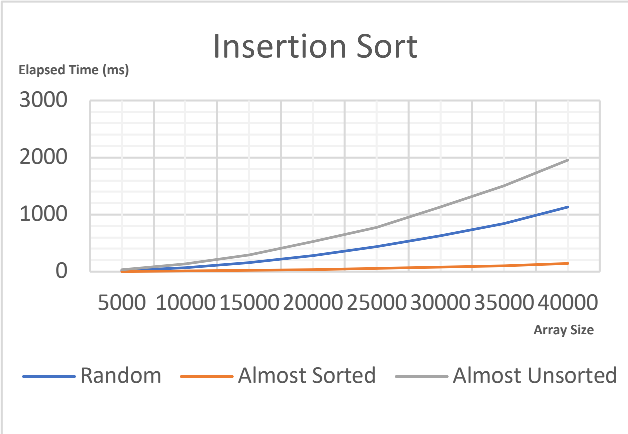
Analysis of Merge Sort

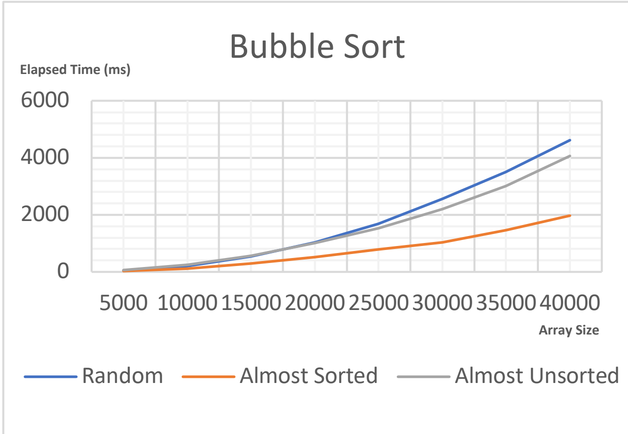
Array Size	Elapsed Time	compCount	moveCount
5000	0.525 ms	48901	123616
10000	1.031 ms	108157	267232
15000	1.495 ms	171862	417232
20000	2.189 ms	237249	574464
25000	2.664 ms	303132	734464
30000	3.251 ms	373055	894464
35000	3.735 ms	444481	1058928
40000	4.533 ms	511280	1228928

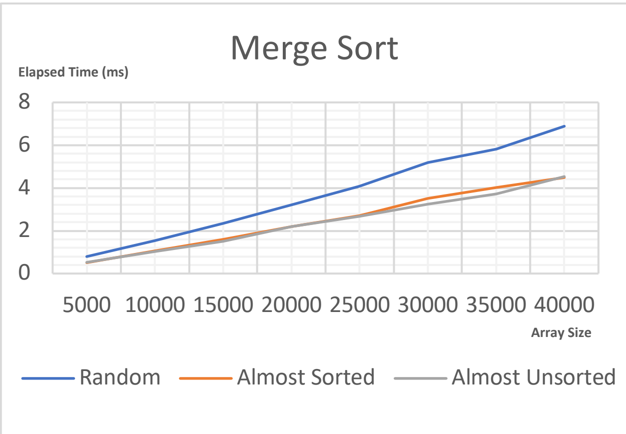
Analysis of Quick Sort

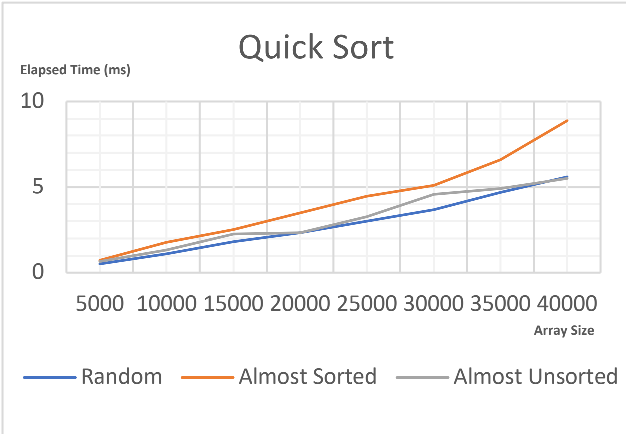
Array Size	Elapsed Time	compCount	moveCount
5000	0.653 ms	148815	247390
10000	1.316 ms	318102	513923
15000	2.244 ms	541510	865974
20000	2.317 ms	530751	860857
25000	3.284 ms	819008	1331230
30000	4.572 ms	1135730	1816011
35000	4.899 ms	1299751	1957746
40000	5.505 ms	1279108	2214196

Question 3









For insertion sort, it is known that the algorithm's complexity for worst and average cases is $O(n^2)$, whereas in the best case, the time complexity is $O(n)$. The worst case happens when the input array is in reverse order, and if the given array is already sorted in ascending order, then the best case happens. By looking at the insertion sort graph created with the observed values, it can be seen that the results obtained are consistent with the theoretical assumptions. Because the almost sorted case is nearly a best case for insertion sort, and it produces a line proportional to n as expected. Similarly, since the almost unsorted case is close to the worst case scenario, its time complexity is proportional to $O(n^2)$ and takes the most time. The case where the elements of the arrays are randomly generated can be considered as the average case, and the time taken by this case is between almost unsorted and almost sorted cases, and its time complexity is also proportional to $O(n^2)$.

In theory, bubble sort has the time complexity of $O(n^2)$ in the worst and average cases and $O(n)$ in the best case. If the array to be sorted is in reverse order, then it is the worst case for bubble sort, and conversely, if the array is already sorted, time complexity results in the best case. As can be seen, the general behavior of the algorithm in terms of time complexity is similar to insertion sort. Confirming the assumptions, the almost sorted case spent the least amount of time on the graph obtained as a result of the measurements and formed a line close to $O(n)$. On the other hand, interestingly, almost unsorted arrays are sorted slightly faster than random arrays. Theoretically, almost unsorted arrays should be the worst case of the algorithm. However, either almost unsorted or random, it was observed that the time consumed to sort the array is proportional to $O(n^2)$. Although the bubble sort algorithm shares the same time complexities with insertion sort, bubble sort takes more time in milliseconds since it involves more key comparisons and move operations.

Merge sort has the time complexity of $O(n \log n)$ in the best, worst and average cases. The best case happens when all the elements in the first half of the array are smaller (or larger) than all the elements in the second half of the array. For the worst case, merge sort requires a specific permutation of numbers in the sorted array. Therefore, as shown in the obtained graph, almost sorted and almost unsorted cases spend the least time sorting since they fit in the description of the best case of the merge sort algorithm. At this point, sorting random arrays takes the most time, and this case can be evaluated as the average case of the merge sort or even the worst case. However, either random, sorted, or unsorted; according to the obtained graph, the elapsed time is almost linear (close to $O(n \log n)$), which validates the theoretical assumptions.

Since the array's first element is taken as the pivot for this experience, in theory, the quick sort will result in the worst case if the given array is already sorted with the time complexity of $O(n^2)$. For the rest of the cases, this version of the quick sort has the time complexity of $O(n \log n)$. In this regard, almost sorted case is the main candidate for the worst case of this experiment for quick sort. Likely, in the obtained graph, elapsed time for almost sorted arrays is nearly order of n^2, and the remaining two cases are almost linear ($O(n \log n)$).