CS 202

Homework 1

Mehmet Feyyaz Küçük

22003550

Section-2

Question-1)

(a) $f(n) = 8n^4 + 5n^3 + 7 <= c*n^5$ for $n > n_0$. For $c = 8$ and $n_0 = 2$, this inequality holds. Therefore $f(n) = 8n^4 + 5n^3 + 7$ is $O(n^5)$.

(b) 1. Selection Sort

[22, 8, 49, 25, 18, 30, 20, 15, 35, 27]; find min in arr[0-9] and put it into arr[0].
=> [**8**, 22, 49, 25, 18, 30, 20, 15, 35, 27]; find min in arr[1-9] and put it into arr[1].
=> [8, **15**, 22, 49, 25, 18, 30, 20, 35, 27]; in this manner find min in arr[k-9] and put it into arr[k]
.
.
=> [8, 15, 18, 20, 22, 25, 27, **30**, 49, 35]
=> [8, 15, 18, 20, 22, 25, 27, 30, **35**, 49]; the array is sorted!

2. Bubble Sort

[22, 8, 49, 25, 18, 30, 20, 15, 35, 27]; swap arr[0:1], arr[1:2], … arr[k-1:k] for k < 9.
=> [**22**, **8**, 49, 25, 18, 30, 20, 15, 35, 27]; 8 < 22, so swap 22 and 8.
=> [8, **22**, **49**, 25, 18, 30, 20, 15, 35, 27]; 22 < 49, so keep the order.
=> [8, 22, **49**, **25**, 18, 30, 20, 15, 35, 27]; 49 > 25, so swap 49 and 25.
=> [8, 22, 25, **49**, **18**, 30, 20, 15, 35, 27]; 49 > 18, so swap 49 and 18.
=> [8, 22, 25, 18, **49**, **30**, 20, 15, 35, 27]; 49 > 30, so swap 49 and 30.
=> [8, 22, 25, 18, 30, **49**, **20**, 15, 35, 27]; 49 > 20, so swap 49 and 20.
=> [8, 22, 25, 18, 30, 20, **49**, **15**, 35, 27]; 49 > 15, so swap 49 and 15.
=> [8, 22, 25, 18, 30, 20, 15, **49**, **35**, 27]; 49 > 35, so swap 49 and 35.
=> [8, 22, 25, 18, 30, 20, 15, 35, **49**, **27**]; 49 > 27, so swap 49 and 27.
=> [8, 22, 25, 18, 30, 20, 15, 35, 27, 49]; continue swapping until the array is sorted.
.
.

=> [8, 15, 18, 20, 22, 25, 27, 30, 35, 49]; algorithm does a final iteration when the array is sorted. If no swapping is done, it knows that the array is sorted.

Question-2)

```
Insertion Sort
Comp Count: 69
Move Count: 88
[ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 ]

Bubble Sort
Comp Count: 165
Move Count: 174
[ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 ]

Merge Sort
Comp Count: 47
Move Count: 128
[ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 ]

Quick Sort
Comp Count: 50
Move Count: 125
[ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 ]
```

Performance Analysis with random arrays

Analysis of Insertion Sort

Array Size	Elapsed Time	compCount	moveCount
5000	19 ms	6329247	6334256
10000	83 ms	25155604	25165614
15000	185 ms	56107700	56122713
20000	340 ms	100088037	100108043
25000	542 ms	157631925	157656930
30000	776 ms	225188838	225218843
35000	1050 ms	305384710	305419720
40000	1404 ms	398644862	398684872

Analysis of Bubble Sort

Array Size	Elapsed Time	compCount	moveCount
5000	90 ms	24550089	18403539
10000	376 ms	97810218	73844898
15000	923 ms	222780147	168036474
20000	1673 ms	395280235	298236549
25000	2672 ms	621900123	470549526
30000	3900 ms	895740141	676932486
35000	5399 ms	1217685208	908941335
40000	7231 ms	1588240293	1198762983

Analysis of Merge Sort

Array Size	Elapsed Time	compCount	moveCount
5000	1 ms	55284	123616
10000	3 ms	120546	267232
15000	4 ms	189286	417232
20000	6 ms	260745	574464
25000	7 ms	334086	734464
30000	8 ms	408541	894464
35000	11 ms	484585	1058928
40000	11 ms	561932	1228928

Analysis of Quick Sort

Array Size	Elapsed Time	compCount	moveCount
5000	1 ms	67926	115803
10000	1 ms	150449	251971
15000	2 ms	271347	376908
20000	2 ms	339759	522185
25000	3 ms	463172	801873
30000	4 ms	511597	807675
35000	4 ms	636352	952541
40000	5 ms	725044	1215031

Performance Analysis with almost sorted arrays

Analysis of Insertion Sort

Array Size	Elapsed Time	compCount	moveCount
5000	2 ms	769119	774118
10000	10 ms	2914297	2924296
15000	26 ms	7122331	7137330
20000	40 ms	11650341	11670340
25000	70 ms	19439915	19464914
30000	108 ms	29272677	29302676
35000	131 ms	35534565	35569566
40000	141 ms	39378855	39418854

Analysis of Bubble Sort

Array Size	Elapsed Time	compCount	moveCount
5000	62 ms	24010197	2212554
10000	256 ms	94420557	9496866
15000	620 ms	223590093	21369054
20000	1069 ms	383420828	34339224
25000	1730 ms	611175552	57791976
30000	2499 ms	879420685	88489386
35000	3139 ms	1105618410	107613072
40000	3756 ms	1289127771	120053058

Analysis of Merge Sort

Array Size	Elapsed Time	compCount	moveCount
5000	2 ms	50903	123616
10000	3 ms	110670	267232
15000	4 ms	173436	417232
20000	5 ms	234992	574464
25000	6 ms	307409	734464
30000	8 ms	377150	894464
35000	9 ms	435271	1058928
40000	9 ms	483589	1228928

Analysis of Quick Sort

Array Size	Elapsed Time	compCount	moveCount
5000	0 ms	200792	194333
10000	2 ms	626460	470396
15000	2 ms	803264	880593
20000	4 ms	1056359	1085643
25000	5 ms	1482778	1515985
30000	5 ms	1623986	1718177
35000	11 ms	4671384	1625211
40000	61 ms	28560523	2075440

Performance Analysis with almost unsorted arrays

Analysis of Insertion Sort

Array Size	Elapsed Time	compCount	moveCount
5000	36 ms	11726986	11732026
10000	157 ms	46918566	46928618
15000	348 ms	105361804	105376816
20000	618 ms	188559415	188579436
25000	944 ms	293392837	293417860
30000	1390 ms	420145923	420175947
35000	1896 ms	575574780	575612032
40000	2505 ms	759578934	759626176

Analysis of Bubble Sort

Array Size	Elapsed Time	compCount	moveCount
5000	88 ms	24995000	35282382
10000	366 ms	99990000	140363286
15000	780 ms	224985000	314897766
20000	1386 ms	399980000	566505318
25000	2226 ms	624975000	879524676
30000	3280 ms	899970000	1263187674
35000	4346 ms	1224965000	1730682180
40000	5718 ms	1599960000	-2015408452

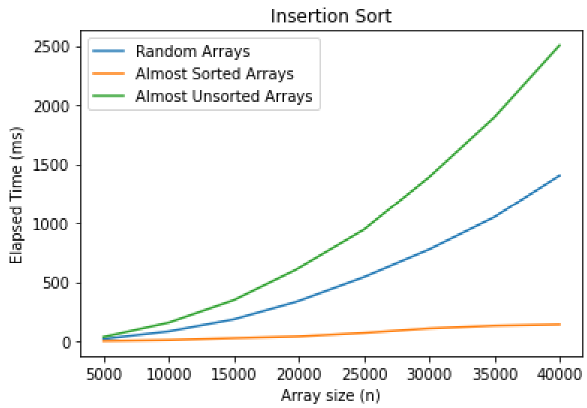
Analysis of Merge Sort

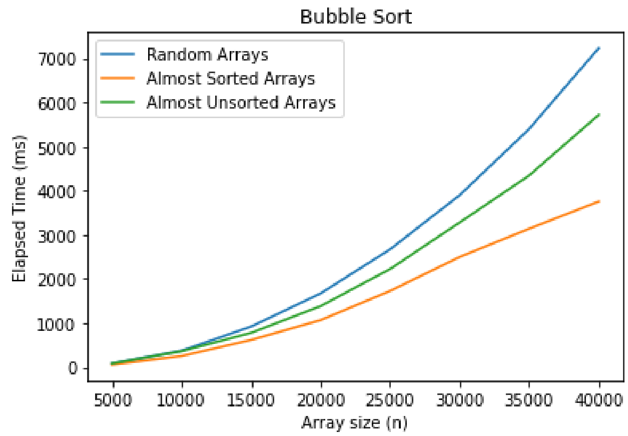
Array Size	Elapsed Time	compCount	moveCount
5000	1 ms	49028	123616
10000	2 ms	108775	267232
15000	4 ms	172884	417232
20000	4 ms	237661	574464
25000	5 ms	306043	734464
30000	7 ms	376237	894464
35000	8 ms	436897	1058928
40000	9 ms	484975	1228928

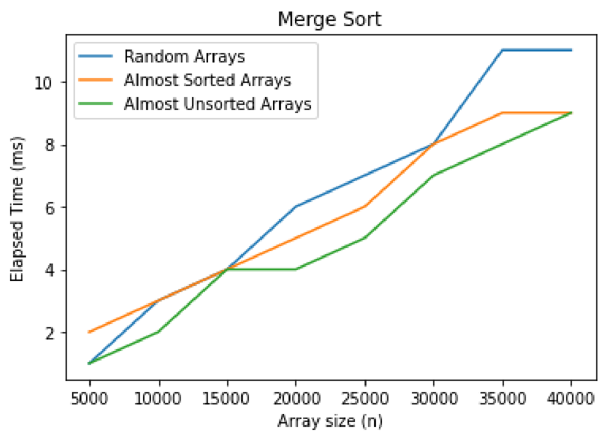
Analysis of Quick Sort

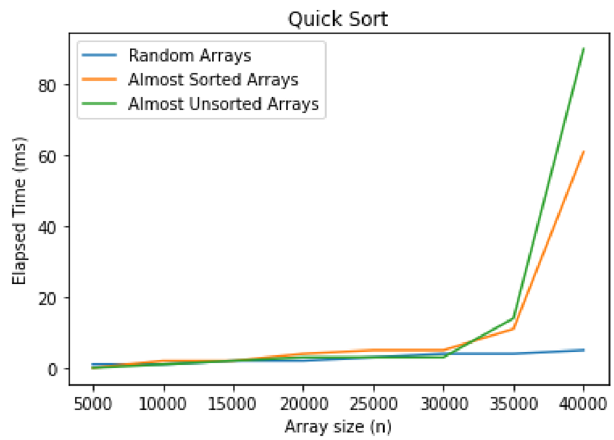
Array Size	Elapsed Time	compCount	moveCount
5000	0 ms	128095	227091
10000	1 ms	253253	429402
15000	2 ms	469595	761713
20000	3 ms	751208	1180216
25000	3 ms	889118	1446624
30000	3 ms	939007	1526865
35000	14 ms	4233430	6397736
40000	90 ms	27455050	41297686

Question-3)


Insertion Sort
Random Arrays
Almost Sorted Arrays
Almost Unsorted Arrays
Elapsed Time (ms)
0
500
1000
1500
2000
2500
5000
10000
15000
20000
25000
30000
35000
40000
Array size (n)


Bubble Sort
Random Arrays
Almost Sorted Arrays
Almost Unsorted Arrays
Elapsed Time (ms)
0
1000
2000
3000
4000
5000
6000
7000
5000
10000
15000
20000
25000
30000
35000
40000
Array size (n)


Merge Sort
Random Arrays
Almost Sorted Arrays
Almost Unsorted Arrays
Elapsed Time (ms)
2
4
6
8
10
5000
10000
15000
20000
25000
30000
35000
40000
Array size (n)


Quick Sort
Random Arrays
Almost Sorted Arrays
Almost Unsorted Arrays
Elapsed Time (ms)
0
20
40
60
80
5000
10000
15000
20000
25000
30000
35000
40000
Array size (n)

Discussion

Insertion sort, as seen from the graph runs in $O(n^2)$ time at its worst and average case. But it runs at O(n) time at its best case. Because since every element is in almost in order it does not need to carry elements to the beginning. Hence the traverse occurs more than one time but way less than n times.

Bubble Sort, as seen from the graph runs in $O(n^2)$ time at its worst and average case like insertion sort. And just like insertion sort it runs in O(n) time at its best case. Because in an almost sorted array, minimum numbers of swaps are needed that the traverse happens a few times.

For merge sort, the graphs do not give us an idea about its time complexity behavior since it runs very fast and we have small values. But the graphs tell us that this algorithm is faster than insertion and bubble sort and also its worst, average, and best cases have similar behaviors. This is indeed in correlation with the theoretical results of merge sort. It runs in O(n*log(n)) time for all three cases which is faster than insertion and bubble sort. Also, all cases have the same time complexities as predicted.

For quick sort, first we can see that it is much faster than insertion and bubble sort just like merge sort. Almost sorted and almost unsorted array cases seem slower compared to the random array case. The reason for this is the worst case for quick sort is picking a bad pivot number. For an almost sorted array we would always choose the smaller numbers as a pivot which is worse case for quick sort. Similarly, for an almost unsorted array we would always choose the greater numbers as a pivot which is also bad for quick sort. On the other hand, random arrays are sorted much faster with quick sort because picking a bad pivot in a randomly created array is less likely. The results from the graph are supported by theoretical results. Since, quick sort runs in $O(n^2)$ time at its worst case and in O(n*log(n)) time at its average case. We can also observe that average case of quick sort is very similar to all cases of merge sort.