CS 202 - Fundamental Structures of Computer Science 2

Homework 1 – Algorithm Efficiency and Sorting

Student Name: Esad İsmail Tök

Student ID: 21801679

Section: 2

Question 1

Part A

- In order to prove that $f(x) = 5n^3 + 4n^2 + 10$ is O(n^4), we need to be able to find two numbers, *c* and n_0, such that $0 \leq 5n^3 + 4n^2 + 10 \leq c \times n^4$ when n ≥ n_0.

- By finding such two integers, *c* and n_0, we will be able to prove that $c \times n^4$ constitutes an upper bound on $f(x) = 5n^3 + 4n^2 + 10$.

- When we choose *c* = 5 and n_0 = 2, the equation that must be satisfied becomes $0 \leq 5n^3 + 4n^2 + 10 \leq 5 \times n^4$ when n ≥ 2. This equation is always true and it can be seen that $5 \times n^4$ is always an upper bound for the equation $f(x) = 5n^3 + 4n^2 + 10$ when we substitute any value for the input size, n, that is greater than or equal to 2.
- Thus we have proved that there is such numbers, *c* and n_0, that holds the equation $0 \leq 5n^3 + 4n^2 + 10 \leq c \times n^4$ when n ≥ n_0, with one of the selections as *c* = 5 and n_0 = 2.
- Therefore the equation $f(x) = 5n^3 + 4n^2 + 10$ is order of n^4, namely O(n^4).

Question 1

Part B

- Tracing the array [24, 8, 51, 28, 20, 29, 21, 17, 38, 27] using the insertion sort algorithm:

24	8	51	28	20	29	21	17	38	27	Copy 8
24	24	51	28	20	29	21	17	38	27	Shift 24
8	24	51	28	20	29	21	17	38	27	Insert 8; copy 51, insert 51 on top of itself
8	24	51	28	20	29	21	17	38	27	Copy 28
8	24	51	51	20	29	21	17	38	27	Shift 51
8	24	28	51	20	29	21	17	38	27	Insert 28; copy 20
8	24	24	28	51	29	21	17	38	27	Shift 51, 28, 24
8	20	24	28	51	29	21	17	38	27	Insert 20; copy 29
8	20	24	28	51	51	21	17	38	27	Shift 51
8	20	24	28	29	51	21	17	38	27	Insert 29; copy 21
8	20	24	24	28	29	51	17	38	27	Shift 51, 29, 28, 24
8	20	21	24	28	29	51	17	38	27	Insert 21; copy 17
8	20	20	21	24	28	29	51	38	27	Shift 51, 29, 28, 24, 21, 20
8	17	20	21	24	28	29	51	38	27	Insert 17; copy 38
8	17	20	21	24	28	29	51	51	27	Shift 51
8	17	20	21	24	28	29	38	51	27	Insert 38; copy 27
8	17	20	21	24	28	28	29	38	51	Shift 51, 38, 29, 28
8	17	20	21	24	27	28	29	38	51	Insert 27

- Tracing the array [24, 8, 51, 28, 20, 29, 21, 17, 38, 27] using the bubble sort algorithm:

24	8	51	28	20	29	21	17	38	27
8	24	51	28	20	29	21	17	38	27
8	24	51	28	20	29	21	17	38	27
8	24	28	51	20	29	21	17	38	27
8	24	28	20	51	29	21	17	38	27
8	24	28	20	29	51	21	17	38	27
8	24	28	20	29	21	51	17	38	27
8	24	28	20	29	21	17	51	38	27
8	24	28	20	29	21	17	38	51	27
8	24	28	20	29	21	17	38	27	51

Pass 1

- In the *Pass 1* figure above, the algorithm compares the array items pairwise and changes the order if the first element is greater than the second. At the end of *Pass 1*, the greatest element is bubbled.

8	24	28	20	29	21	17	38	27	51
8	24	28	20	29	21	17	38	27	51
8	24	28	20	29	21	17	38	27	51
8	24	20	28	29	21	17	38	27	51
8	24	20	28	29	21	17	38	27	51
8	24	20	28	21	29	17	38	27	51
8	24	20	28	21	17	29	38	27	51
8	24	20	28	21	17	29	38	27	51
8	24	20	28	21	17	29	27	38	51

Pass 2

- In the *Pass 2* figure above, the algorithm compares the array items pairwise until the last bubbled item and changes the order if the first element is greater than the second. At the end of *Pass 2*, the second greatest element is bubbled.

8	24	20	28	21	17	29	27	38	51
8	24	20	28	21	17	29	27	38	51
8	20	24	28	21	17	29	27	38	51
8	20	24	28	21	17	29	27	38	51
8	20	24	21	28	17	29	27	38	51
8	20	24	21	17	28	29	27	38	51
8	20	24	21	17	28	29	27	38	51
8	20	24	21	17	28	27	29	38	51

Pass 3

- In the *Pass 3* figure above, the algorithm compares the array items pairwise until the last bubbled item and changes the order if the first element is greater than the second. At the end of *Pass 3*, the third greatest element is bubbled.

8	20	24	21	17	28	27	29	38	51
8	20	24	21	17	28	27	29	38	51
8	20	24	21	17	28	27	29	38	51
8	20	21	24	17	28	27	29	38	51
8	20	21	17	24	28	27	29	38	51
8	20	21	17	24	28	27	29	38	51
8	20	21	17	24	27	28	29	38	51

Pass 4

- In the *Pass 4* figure above, the algorithm compares the array items pairwise until the last bubbled item and changes the order if the first element is greater than the second. At the end of *Pass 4*, the fourth greatest element is bubbled.

8	20	21	17	24	27	28	29	38	51
8	20	21	17	24	27	28	29	38	51
8	20	21	17	24	27	28	29	38	51
8	20	17	21	24	27	28	29	38	51
8	20	17	21	24	27	28	29	38	51
8	20	17	21	24	27	28	29	38	51

Pass 5

- In the *Pass 5* figure above, the algorithm compares the array items pairwise until the last bubbled item and changes the order if the first element is greater than the second. At the end of *Pass 5*, the fifth greatest element is bubbled.

8	20	17	21	24	27	28	29	38	51
8	20	17	21	24	27	28	29	38	51
8	17	20	21	24	27	28	29	38	51
8	17	20	21	24	27	28	29	38	51
8	17	20	21	24	27	28	29	38	51

Pass 6

- In the *Pass 6* figure above, the algorithm compares the array items pairwise until the last bubbled item and changes the order if the first element is greater than the second. At the end of *Pass 6*, the sixth greatest element is bubbled.

8	17	20	21	24	27	28	29	38	51
8	17	20	21	24	27	28	29	38	51
8	17	20	21	24	27	28	29	38	51
8	17	20	21	24	27	28	29	38	51
8	17	20	21	24	27	28	29	38	51

Pass 7

- In the *Pass 7* figure above, the algorithm compares the array items pairwise until the last bubbled array item and there is no data move happens between the pairwise array elements in this pass.

- Therefore bubble sort algorithm decides that the array is sorted using its boolean flag. And since there is no need to go along the other passes and the array becomes sorted and the algorithm ends with the *Pass 7*.

Question 2

Part C

```
---Selection Sort---

The array before selection sort
[12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8]

Number of key comparisons: 120
Number of data moves: 45

The array after selection sort
[3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21]


---Merge Sort---

The array before merge sort
[12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8]

Number of key comparisons: 46
Number of data moves: 128

The array after merge sort
[3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21]


---Quick Sort---

The array before quick sort
[12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8]

Number of key comparisons: 45
Number of data moves: 93

The array after quick sort
[3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21]


---Radix Sort---

The array before radix sort
[12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8]

The array after radix sort
[3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21]
```

- The above figure is the console output of the Question 2 Part C. For the selection sort, merge sort, and quick sort algorithms; the program displays the array before the sorting operation, then displays the number of key comparisons and data moves to sort the array, and lastly displays the sorted form of the array. For the radix sort algorithm, the program displays the array before the sorting operation and the array after the sorting operation.

Question 2

Part D

```
-----Performing Analysis on Randomly Ordered Arrays-----

--------------------------------------------------------
Analysis of Selection Sort
Array Size      Elapsed Time    compCount       moveCount
6000            34      ms      17997000        17997
10000           94      ms      49995000        29997
14000           181     ms      97993000        41997
18000           296     ms      161991000       53997
22000           440     ms      241989000       65997
26000           615     ms      337987000       77997
30000           817     ms      449985000       89997

--------------------------------------------------------
Analysis of Merge Sort
Array Size      Elapsed Time    compCount       moveCount
6000            2       ms      67729           151616
10000           2       ms      120233          267232
14000           3       ms      174990          387232
18000           4       ms      231522          510464
22000           5       ms      289475          638464
26000           5       ms      348075          766464
30000           6       ms      407612          894464

--------------------------------------------------------
Analysis of Quick Sort
Array Size      Elapsed Time    compCount       moveCount
6000            0       ms      225923          86823
10000           2       ms      584190          173112
14000           3       ms      1112980         259110
18000           4       ms      1771131         306582
22000           5       ms      2584627         289272
26000           8       ms      3610581         361692
30000           10      ms      4753363         464052

--------------------------------------------------------
Analysis of Radix Sort
Array Size      Elapsed Time
6000            1       ms
10000           0       ms
14000           0       ms
18000           0       ms
22000           1       ms
26000           1       ms
30000           0       ms
```

- The above figure is the output for the performing analysis of the sorting random order array scenario from Question 2 Part D.
- For selection sort, merge sort, and quick sort; the program displays the elapsed times (ms), key comparison count, and data move count for 7 different input sizes.
- For the radix sort, the program displays the elapsed time (ms) for 7 different input sizes.

Question 2

Part D

```
-----Performing Analysis on Ascendingly Ordered Arrays-----

-----------------------------------------------------------
Analysis of Selection Sort
Array Size      Elapsed Time    compCount       moveCount
6000            34    ms        17997000        17997
10000           102   ms        49995000        29997
14000           180   ms        97993000        41997
18000           324   ms        161991000       53997
22000           452   ms        241989000       65997
26000           616   ms        337987000       77997
30000           817   ms        449985000       89997

-----------------------------------------------------------
Analysis of Merge Sort
Array Size      Elapsed Time    compCount       moveCount
6000            0     ms        42538           151616
10000           1     ms        74599           267232
14000           2     ms        108369          387232
18000           2     ms        142209          510464
22000           3     ms        176166          638464
26000           4     ms        211081          766464
30000           5     ms        250701          894464

-----------------------------------------------------------
Analysis of Quick Sort
Array Size      Elapsed Time    compCount       moveCount
6000            36    ms        17997000        17997
10000           99    ms        49995000        29997
14000           178   ms        97993000        41997
18000           305   ms        161991000       53997
22000           427   ms        241989000       65997
26000           592   ms        337987000       77997
30000           816   ms        449985000       89997

-----------------------------------------------------------
Analysis of Radix Sort
Array Size      Elapsed Time
6000            0     ms
10000           0     ms
14000           0     ms
18000           0     ms
22000           1     ms
26000           1     ms
30000           1     ms
```

- The above figure is the output for the performing analysis of the sorting ascending order array scenario from Question 2 Part D.
- For selection sort, merge sort, and quick sort; the program displays the elapsed times (ms), key comparison count, and data move count for 7 different input sizes.
- For the radix sort, the program displays the elapsed time (ms) for 7 different input sizes.

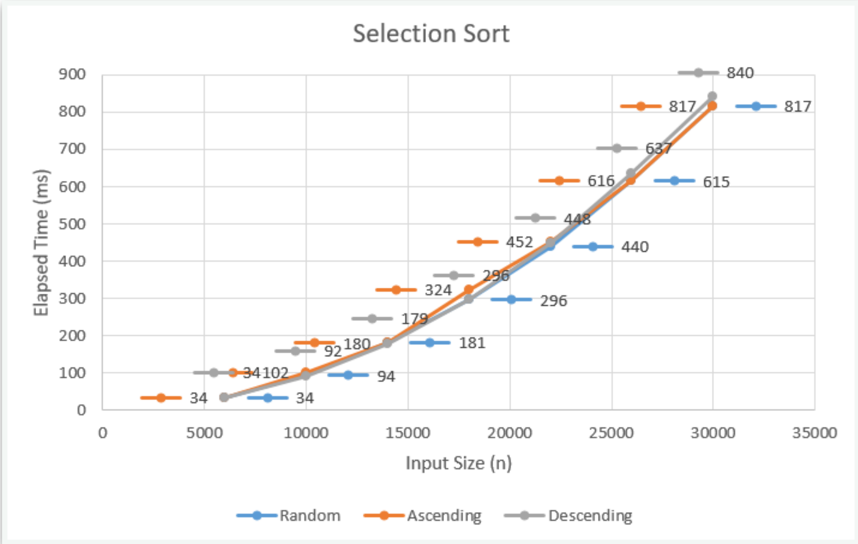
Question 2

Part D

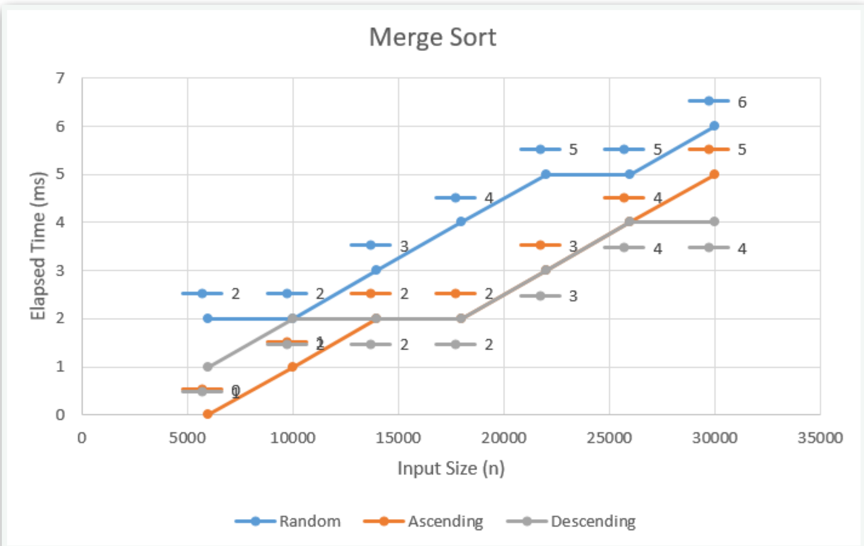
```
-----Performing Analysis on Descendingly Ordered Arrays-----

-------------------------------------------------------
Analysis of Selection Sort
Array Size      Elapsed Time    compCount       moveCount
6000            34      ms      17997000        17997
10000           92      ms      49995000        29997
14000           179     ms      97993000        41997
18000           296     ms      161991000       53997
22000           448     ms      241989000       65997
26000           637     ms      337987000       77997
30000           840     ms      449985000       89997

-------------------------------------------------------
Analysis of Merge Sort
Array Size      Elapsed Time    compCount       moveCount
6000            1       ms      36656           151616
10000           2       ms      64608           267232
14000           2       ms      94256           387232
18000           2       ms      124640          510464
22000           3       ms      154208          638464
26000           4       ms      186160          766464
30000           4       ms      219504          894464

-------------------------------------------------------
Analysis of Quick Sort
Array Size      Elapsed Time    compCount       moveCount
6000            2       ms      764245          898224
10000           3       ms      1460716         1477608
14000           5       ms      2345797         2100615
18000           8       ms      3380398         2714685
22000           10      ms      4564966         3308643
26000           14      ms      5918483         3920526
30000           16      ms      7430242         4521792

-------------------------------------------------------
Analysis of Radix Sort
Array Size      Elapsed Time
6000            0       ms
10000           0       ms
14000           1       ms
18000           1       ms
22000           1       ms
26000           1       ms
30000           1       ms
```

- The above figure is the output for the performing analysis of the sorting descending order array scenario from Question 2 Part D.
- For selection sort, merge sort, and quick sort; the program displays the elapsed times (ms), key comparison count, and data move count for 7 different input sizes.
- For the radix sort, the program displays the elapsed time (ms) for 7 different input sizes.

Question 3



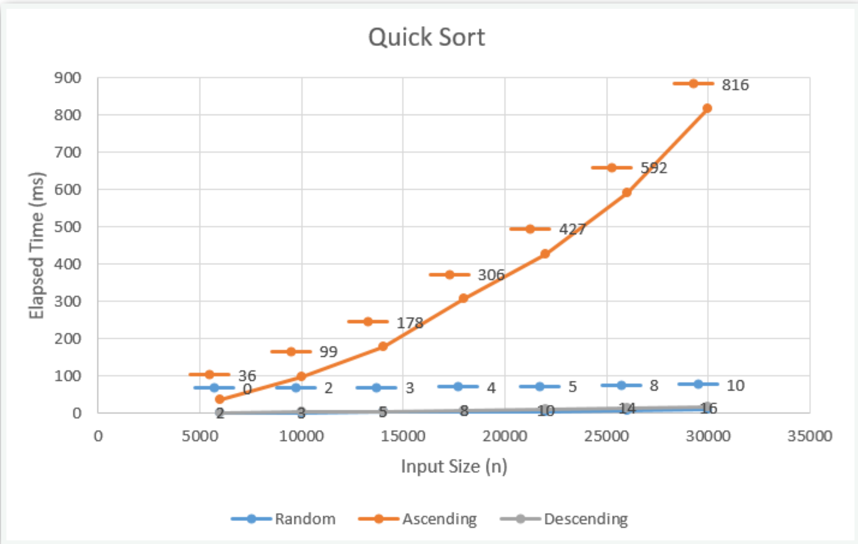
- Selection sort algorithm is O(n^2) for random, ascending, and descending order arrays and its growth rate is independent of the order of the input array since the algorithm needs to find the greatest element in the unsorted sublist in each iteration. As it can be seen in the graph the records are considerably similar for all scenarios.
- When the experimental results are compared with theoretical expectations, it can be seen that the experimental pattern is significantly similar with the theoretical ones. However there occurs some minor elapsed time differences for some executions that can be caused by the amount of CPU usage of the computer at the run time or machine dependent factors if the executions are done in different computers



- Merge sort algorithm is O(n*logn) for random, ascending, and descending order arrays and its growth rate is independent of the order of the input array. The elapsed time records of the merge sort is considerably smaller than the selection sort.

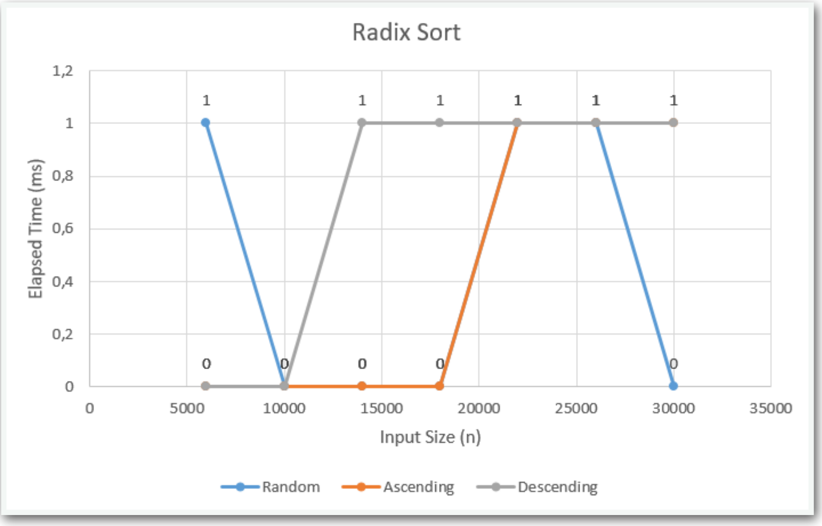
Question 3

- Although the algorithm is always O(n*logn), it can be seen that for randomly ordered array scenario, the elapsed time records are greater than the other scenarios. So there is a contrast between the theoretical expectations.
- It is because even though the number of data moves does not change across the scenarios and decides the growth rate, there is less number of key comparisons in ascending and descending order arrays since one half is always smaller or greater than the other half. Therefore there can happen minor elapsed time differences among different scenarios.



- Quick sort algorithm is O(n*logn) for random order input array but O(n^2) for ascending and descending order input arrays.
- The random input array is the average case of the quick sort algorithm. Both ascending and descending input arrays are the worst cases of quick sort algorithm because in those cases the pivot value, which is the first element of the input array, partitions the array into 2 subarrays with sizes 0 and (n - 1). Therefore bad partition causes the algorithm to grow as fast as selection sort algorithm which is also O(n^2).
- According to the theoretical results, the descending performance of the algorithm also needs to have a similar pattern with ascending scenario. However in the graph, the elapsed times of the ascending array scenario dominates the descending one in contrast to theory. The causes of this situation may be the compiler or any other intermediate step between the compilation step and generating the executable.

Question 3



- Radix sort algorithm is O(n) for random, ascending, and descending order arrays and its growth rate is independent of the order of the input array. It is a significantly faster algorithm compared to other ones.
- The reason why this algorithm is so fast and does not depend on the input array arrangement is, because it does not use key comparisons when sorting the array, so does not waste the time with comparisons.
- At first the patterns of the graph seems not to be fitting the theoretical O(n) pattern of the algorithm. However it is because the elapsed time values are too small and only changes between 0 and 1. Therefore the input sizes up to 30000 are not enough to observe the complete pattern of the radix sort algorithm and larger input sizes are needed to fit the algorithm into theoretical pattern.