Title: Algorithm Efficiency and Sorting
Author: EFE ACER
ID: 21602217
Section: 3
Assignment: 1
Description: The answers of questions 1 and 3 are given in this pdf.

Question 1:

(a)

$$f_4(n) < f_9(n) < f_8(n) < f_2(n) = f_{10}(n) < f_5(n) < f_3(n) < f_1(n) = f_6(n) < f_7(n) < f_{11}(n)$$

(b-1)

$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n^2$, $T(1) = 1$ where n is an exact power of 3

using the repeated substitution method:

$$T(n) = 9 \cdot \left(9 \cdot T\left(\frac{n}{3^2}\right) + \left(\frac{n}{3}\right)^2\right) + n^2$$

$$= 9 \cdot \left(9 \cdot \left(9 \cdot T\left(\frac{n}{3^3}\right) + \left(\frac{n}{3^2}\right)^2\right) + \left(\frac{n}{3}\right)^2\right) + n^2$$

$$= 9 \cdot 9 \cdot 9 \cdot T\left(\frac{n}{3^3}\right) + n^2 + 9 \cdot \left(\frac{n}{3}\right)^2 + 9 \cdot 9 \cdot \left(\frac{n}{3^2}\right)^2$$

$$= T\left(\frac{n}{3^3}\right) \cdot \prod_{i=1}^{3} 9 + \sum_{i=1}^{3} 9^{i-1} \cdot \left(\frac{n}{3^{i-1}}\right)^2$$

$$= T\left(\frac{n}{3^3}\right) \cdot 9^3 + \sum_{i=1}^{3} n^2$$

$\vdots$ after $\log_3 n$ substitutions in total

$$= T\left(\frac{n}{3^{\log_3 n}}\right) \cdot 9^{\log_3 n} + \sum_{i=1}^{\log_3 n} n^2 = T(1) \cdot 9^{\log_3 n} + \sum_{i=1}^{\log_3 n} n^2$$

$$= 9^{\log_3 n} + n^2 \cdot \log_3 n = 3^{2 \cdot \log_3 n} + n^2 \cdot \log_3 n = 3^{\log_3 n^2} + n^2 \cdot \log_3 n$$

$$= n^2 \cdot (1 + \log_3 n) = n^2 \cdot (\log_3 3 + \log_3 n) = n^2 \cdot (\log_3 3n)$$

$$= n^2 \cdot \left(\frac{\log_2 3n}{\log_2 3}\right) = n^2 \cdot \left(\frac{\log_2 n + \log_2 3}{\log_2 3}\right) = n^2 \cdot (\log_2 n) \cdot \frac{1}{\log_2 3} + n^2$$

We can ignore the low − order term and the multiplicative constant of the highest − order term:

Hence, $T(n) = \Theta\left(n^2 (\log n)\right)$

(b-2)

$T(n) = T\left(\frac{n}{2}\right) + 2, T(1) = 1$ where n is an exact power of 2

using the repeated substitution method:

$$T(n) = \left(T\left(\frac{n}{2^2}\right) + 2\right) + 2$$

$$= \left(\left(T\left(\frac{n}{2^3}\right) + 2\right) + 2\right) + 2$$

$$= T\left(\frac{n}{2^3}\right) + \sum_{i=1}^{3} 2 = T\left(\frac{n}{2^3}\right) + 2 \cdot 3$$

$\vdots$ after $\log_2 n$ substitutions in total

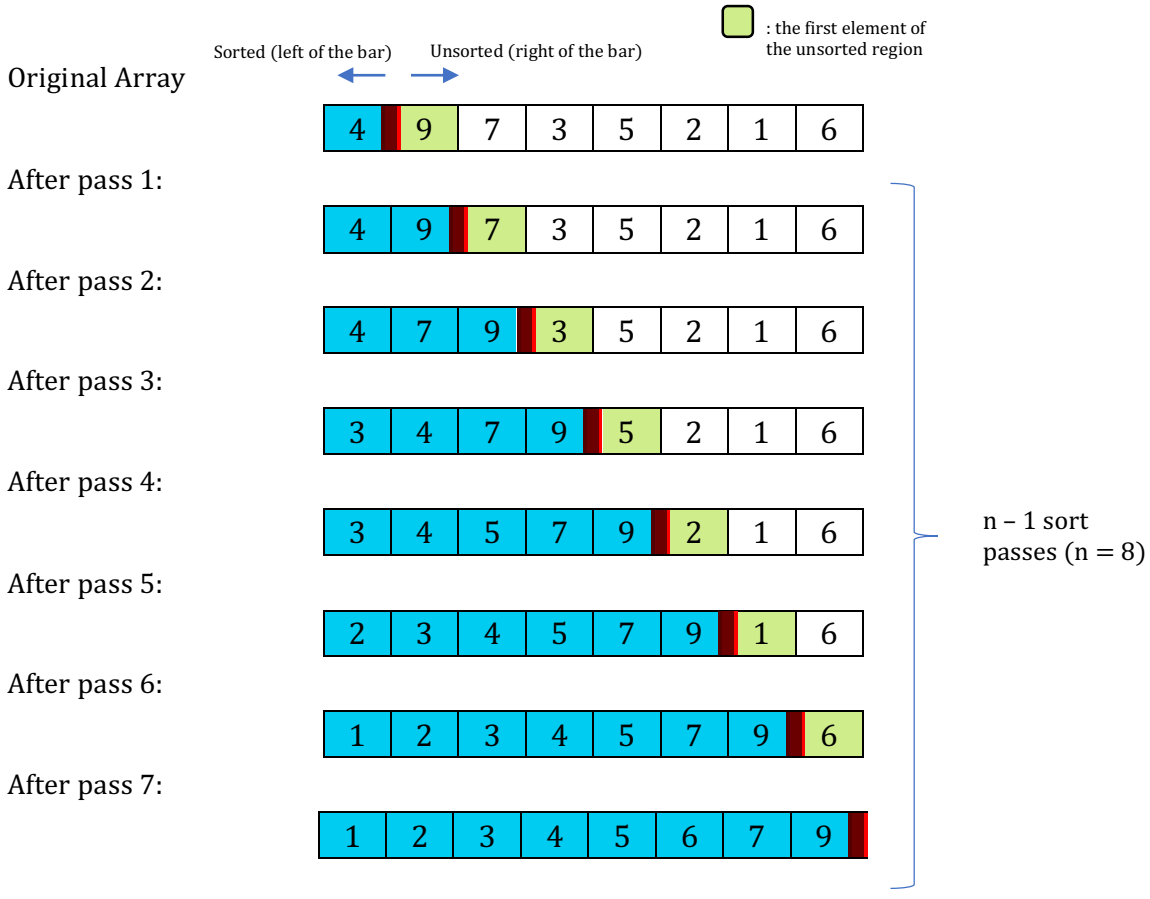
$$= T\left(\frac{n}{2^{\log_2 n}}\right) + 2 \cdot \log_2 n = T(1) + 2 \cdot \log_2 n = 1 + 2 \cdot \log_2 n$$

We can ignore the low − order term and the multiplicative constant of the highest − order term:

Hence, $T(n) = \Theta(\log n)$

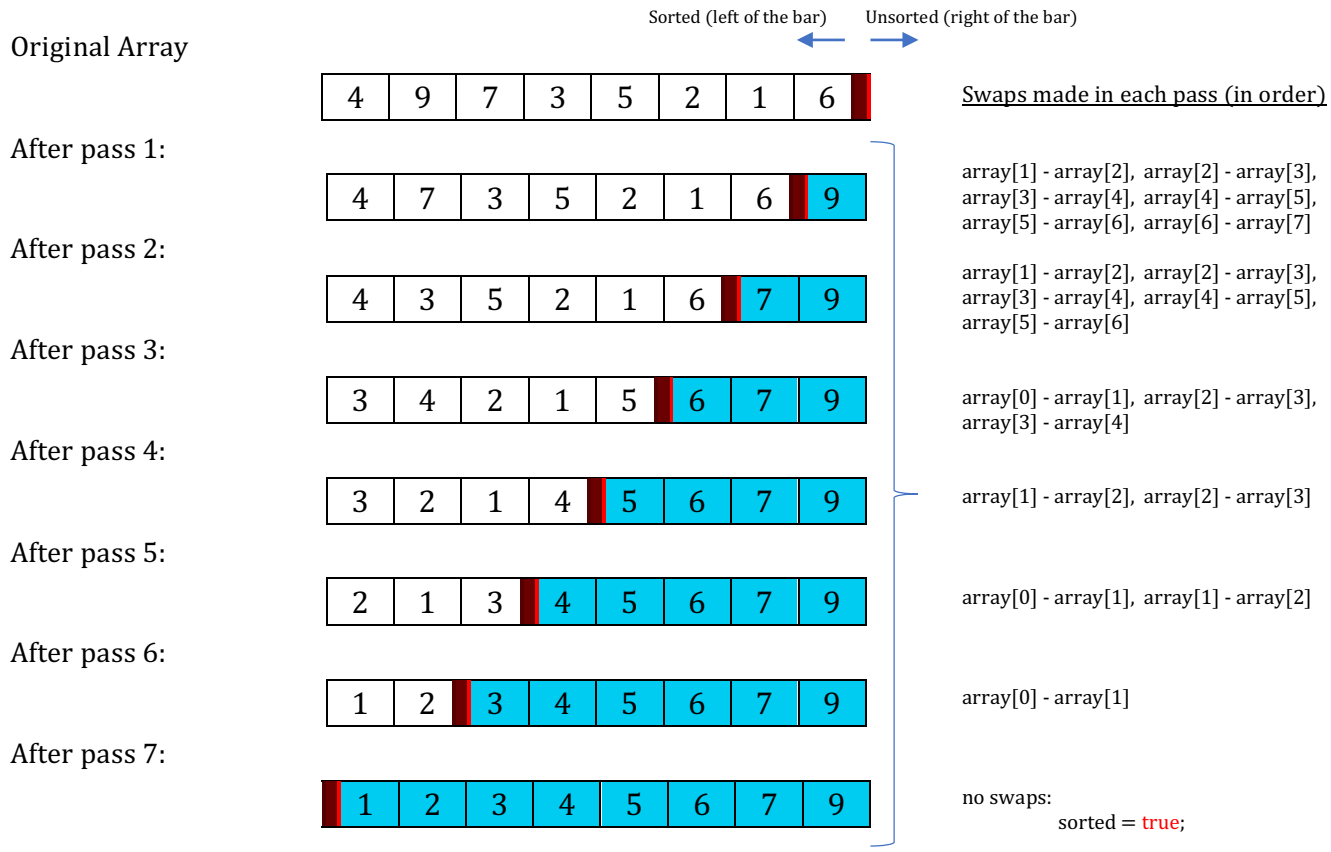
(c-1)

Trace of Insertion Sort:



(c-2)

Trace of Bubble Sort:



Question 3:

Selection Sort, Merge Sort and Quick Sort algorithms are implemented in question 2 and they are tested for various array types and array sizes. The details in their implementation reflected on the total run time, key comparison count and item move count for each of the array types. Certain amount of data is obtained in terms of run time, key comparison count and item move count. Theoretical predictions are tested through the obtained data and their validity became evident. The obtained data is presented in the tables and graphs that are displayed in the following pages.

In the tables:

RxK indicates an array of size x000 with random entries.

AxK indicates an array of size x000 with ascending entries.

DxK indicates an array of size x000 with descending entries.

MxK indicates an array of size x000 with ascending entries up to half of its size and descending entries up to the end.

Tables:

	Elapsed Time (msec)		
Array Type	Selection Sort	Merge Sort	Quick Sort
R1K	1.489	0.169	0.138
A1K	1.302	0.115	2.505
D1K	1.389	0.115	1.374
M1K	1.395	0.108	1.902
R6K	47.005	1.048	0.92
A6K	45.864	0.611	83.325
D6K	42.712	0.608	40.107
M6K	41.884	0.637	64.854
R12K	173.192	2.066	1.945
A12K	172.865	1.227	341.14
D12K	178.023	1.187	156.885
M12K	172.709	1.296	259.868
R18K	383.591	3.114	2.682
A18K	394.701	1.814	798.35
D18K	404.509	1.776	351.925
M18K	381.061	1.916	587.776

Table 1:
Elapsed Time
vs
Array Types

	Key Comparison Count		
Array Type	Selection Sort	Merge Sort	Quick Sort
R1K	499500	8725	11081
A1K	499500	4932	499500
D1K	499500	5044	499500
M1K	499500	5487	250499
R6K	17997000	67776	85564
A6K	17997000	36656	17997000
D6K	17997000	39152	17997000
M6K	17997000	40903	9002999
R12K	71994000	147635	194711
A12K	71994000	79312	71994000
D12K	71994000	84303	71994000
M12K	71994000	87807	36005999
R18K	161991000	232016	296960
A18K	161991000	124640	161991000
D18K	161991000	130592	161991000
M18K	161991000	136615	81008999

Table 2:
Key Comparison
Count
vs
Array Types

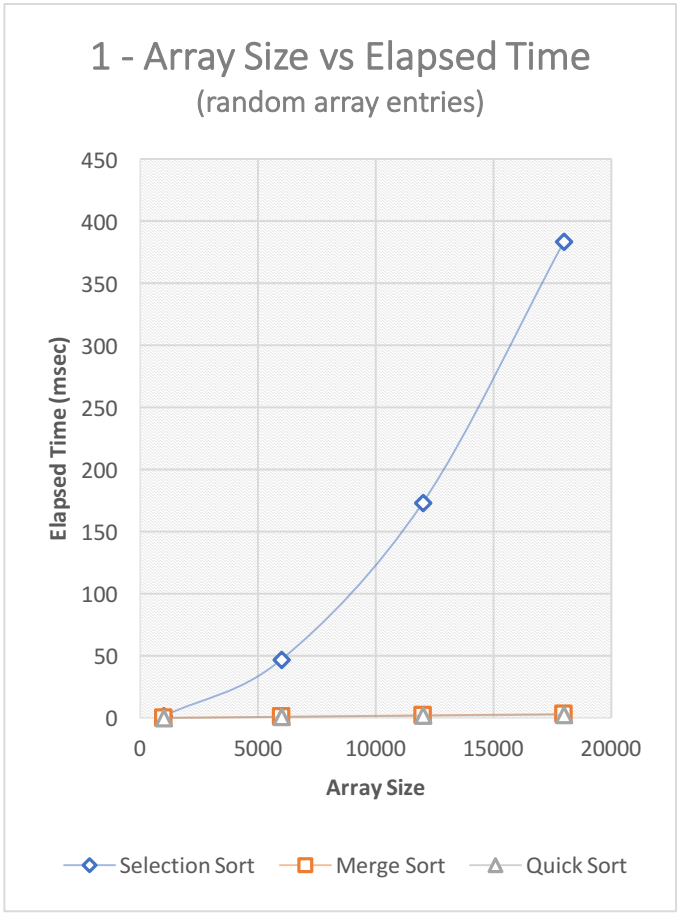
	Item Move Count		
Array Type	Selection Sort	Merge Sort	Quick Sort
R1K	2997	19952	20096
A1K	2997	19952	753996
D1K	2997	19952	3996
M1K	2997	19952	752000
R6K	17997	151616	143106
A6K	17997	151616	27023996
D6K	17997	151616	23996
M6K	17997	151616	27012000
R12K	35997	327232	334524
A12K	35997	327232	108047996
D12K	35997	327232	47996
M12K	35997	327232	108024000
R18K	53997	510464	489649
A18K	53997	510464	243071996
D18K	53997	510464	71996
M18K	53997	510464	246000303

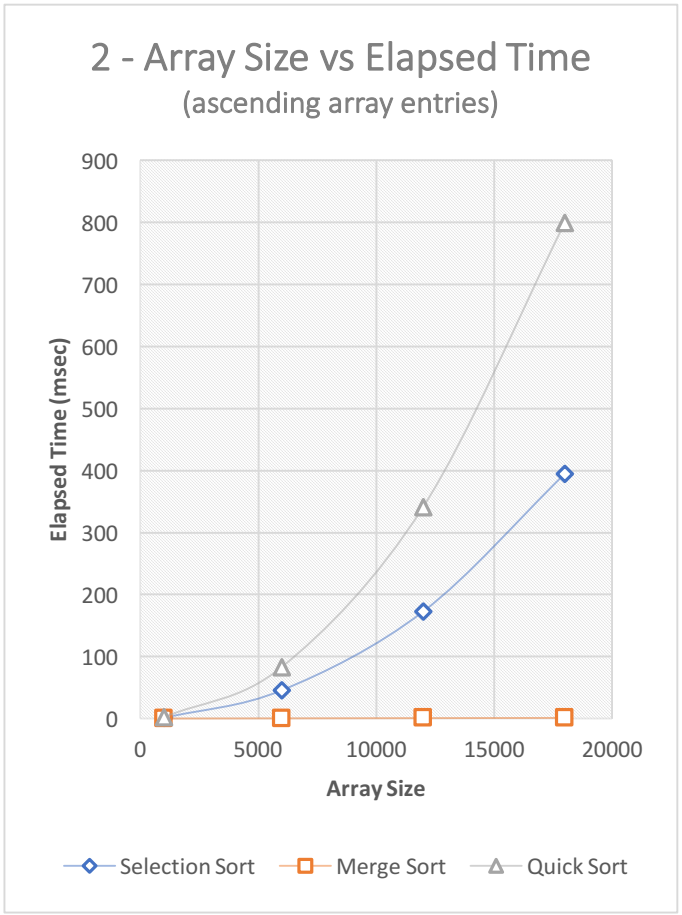
Table 3:
Item Move Count
vs
Array Types

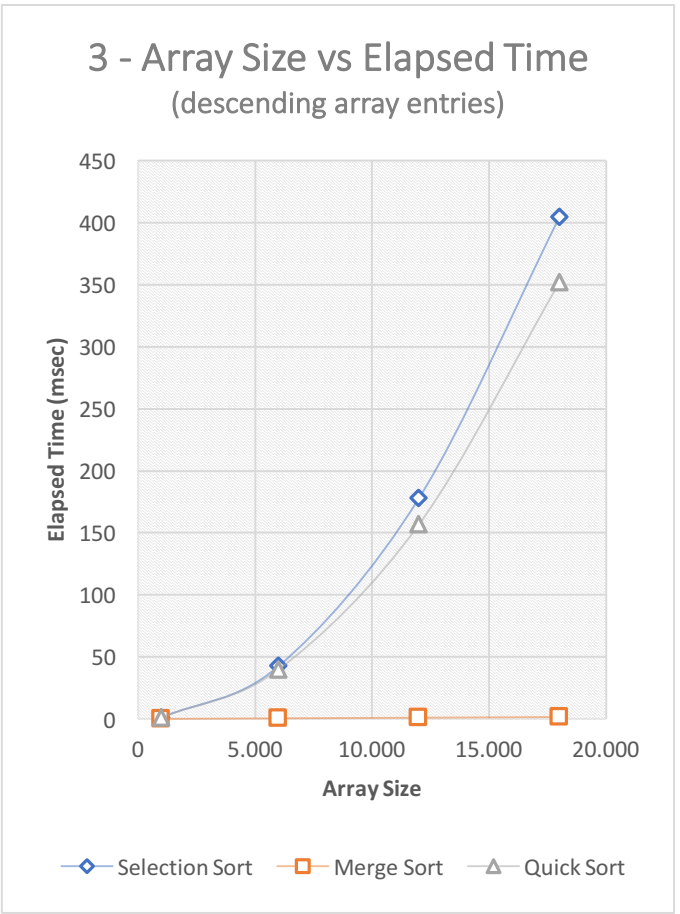
Key Comparison Count and Item Move Count Analysis - Comments (n: array size):

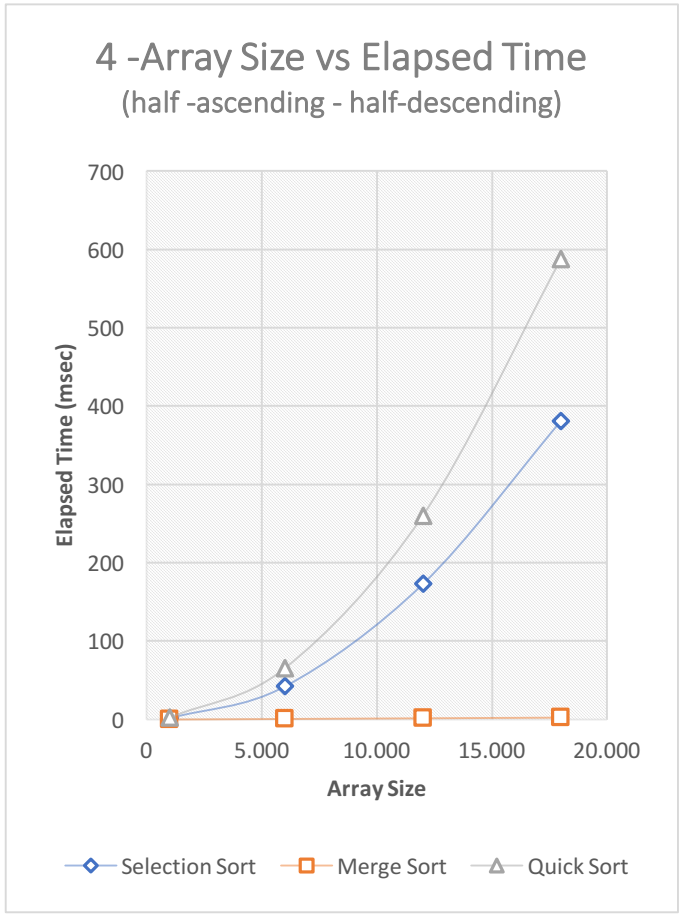
- ➔ For Selection Sort, all key comparisons are made when the method tries to find the index of the largest/smallest element. This process requires $n.(n - 1)/2$ key comparisons in total. Also, there are $3(n - 1)$ item moves due to $(n - 1)$ swap operations. Since these numbers are exact, they are not affected by the distribution of the data.
- ➔ For Merge Sort, merging two arrays of size n requires 2n moves to form the auxilary array and 2n more to copy it back. Hence, number of moves in a single merge operation is 4n. This number is exact so the data distribution does not affect the item move count as it can be seen from the table. However, number of comparisons in the merge operation varies between n and 2n - 1 depending on the magnitude of first array's items relative to the second array's items. Thus, number of key comparisons is not constant for different distributions.
- ➔ For Quick Sort, there are no exact values for key comparison count and item move count because of the random nature of the algorithm. Yet, number of key comparisons can at most be equal to the Selection Sort's in the worst case.

Elapsed Time Analysis using Graphs:









Comments (n is used as array size):

- ➔ Looking at Graph 1, we can say that Quick Sort and Merge Sort outperforms Selection Sort. If we look closer, we can also say that Quick Sort is indeed a little bit faster than Merge Sort. The reason for this is that Merge Sort allocates an auxiliary array in memory for each merge operation, resulting in a cumulative increase in elapsed time. This situation is valid only if we are sorting an array of random entries. However, real-life problems often contain randomly distributed data, thus Quick Sort is naturally a more appealing selection than the other two.
- ➔ Looking at Graph 2, we can say that the behavior of Quick Sort changed dramatically. The change actually occurred because of our pivot selection. Since we are trying to sort the list in descending order and the pivot is always chosen as the very first item; an ascending list becomes our worst case. In other words, each partition moves (n - 1) items that are on the right of the pivot to the left of the pivot resulting in two lists of sizes (n - 1) and 0. Since we failed to divide the list in to two lists of its half size and did a lot more item moves than usual, Quick Sort lost the logarithmic behavior and became very slow for this case.
- ➔ Looking at Graph 3, we can say that ascending lists are no good for Quick Sort too. The pivot selection resulted in the decrease in speed again. In this case, partition does not move any item but it divides the list to two sub-lists of (n - 1) items and 0 items, which again violates our aim of having sub-problems of half the problem's size.
- ➔ Even before looking at Graph 4, we can say that the behavior of Quick Sort will be worse than its behavior in Graph 3 but better than the behavior in Graph 2. The reason being that we divide the lists in to two lists of size (n - 2) and 1 and do less key comparisons but do much more item moves since the list involves ascending sub-lists.
- ➔ Considering all four graphs, we can say that the time complexities of Selection Sort and Merge Sort are independent of the distribution of array's entries. These two algorithms assure a consistent behavior.
- ➔ No difference occurred between the empirical results and the theoretical ones. Selection Sort performed an $\Theta(n^2)$ and Merge Sort performed an $\Theta\big(n(\log n)\big)$ complexity in all cases. On the other hand, Quick Sort run in $\Theta\big(n(\log n)\big)$ for the best and average cases but $\Theta(n^2)$ for the worst case, as expected.