/*
 * Title : Algorithm Efficiency and Sorting
 * Author : Bartu Atabek
 * ID : 21602229
 * Section : 02
 * Assignment : 1
 * Description : This file contains the answers to the questions 1 and 3.
 */

Question 1

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

(b)

—> T(n) = 9T(n/3) + n^2, T(1) = 1 where n is an exact power of 3

$$T(n) = 9T(n/3) + O(n^2)$$

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

$$T(n/9) = 9T(n/27) + O\left(\frac{n}{9}\right)^2$$

$$T(n) = 9\left[9T\left(\frac{n}{9}\right) + O\left(\frac{n^2}{9}\right)\right] + O(n^2)$$

$$= 81T\left(\frac{n}{9}\right) + 9O\left(\frac{n^2}{9}\right) + O(n^2)$$

$$= 81\left[9T\left(\frac{n}{27}\right) + O\left(\frac{n^2}{81}\right)\right] + 9O\left(\frac{n^2}{9}\right) + O(n^2)$$

$$= 729T\left(\frac{n}{27}\right) + 81O\left(\frac{n^2}{81}\right) + 9O\left(\frac{n^2}{9}\right) + O(n^2)$$

$$\Rightarrow T(n) = 9^m T\left(\frac{n}{3^m}\right) + 9^{m-1}O\left(\frac{n^2}{9^{m-1}}\right) + \ldots + O\left(\frac{n^2}{9^0}\right)$$

$$T(1) = 1 \Rightarrow m = \log(n) \Rightarrow 9^m = n^2$$

$$\Rightarrow T(n) = n^2 + O(n^2) + O(n^2) + \ldots + O(n^2)$$

$$T(n) = O(n^2 \log n)$$

—> T(n) = T(n/2) + 2, T(1) = 1 where n is an exact power of 2

$$T(n) = T(n/2) + O(2)$$
$$T(n/2) = T(n/4) + O(2)$$
$$T(n/2) = T(n/4) + O(2)$$

$$\begin{aligned} T(n) &= T(n/4) + 2O(2) \\ &= T(n/8) + 3O(2) \\ &= T(n/2^m) + mO(2) \end{aligned} \qquad \begin{aligned} \frac{n}{2^m} &= 1 = O(1) \\ n &= 2^m \\ m &= \log_2 n \end{aligned}$$

$$T(n) = O(1) + \log_2 n O(2)$$
$$T(n) = O(\log n)$$

(c) Below are the tracing of the two sorting algorithms for the given array. The value that the arrow points out for the insertion sort is the key value and for the bubble sort it's the max value.

Insertion Sort:

4	9	7	3	5	2	1	6

Sorted

4	9	7	3	5	2	1	6

Sorted

4	7	9	3	5	2	1	6

Sorted

3	4	7	9	5	2	1	6

Sorted

3	4	5	7	9	2	1	6

Sorted

2	3	4	5	7	9	1	6

Sorted

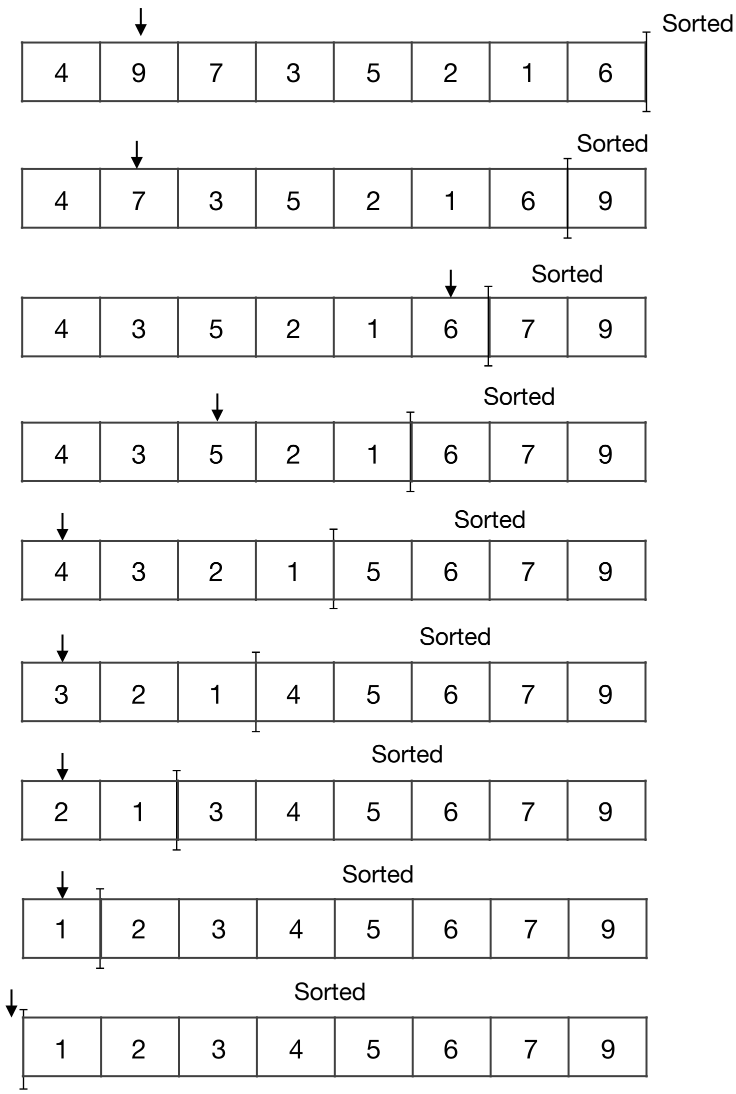
1	2	3	4	5	7	9	6

Sorted

1	2	3	4	5	6	7	9

Sorted

Bubble Sort:


Sorted
4 9 7 3 5 2 1 6
Sorted
4 7 3 5 2 1 6 9
Sorted
4 3 5 2 1 6 7 9
Sorted
4 3 5 2 1 6 7 9
Sorted
4 3 2 1 5 6 7 9
Sorted
3 2 1 4 5 6 7 9
Sorted
2 1 3 4 5 6 7 9
Sorted
1 2 3 4 5 6 7 9
Sorted
1 2 3 4 5 6 7 9

Question 3

Table 1: Experimental results for 1k array

Input Size (n)	Selection Sort			Merge Sort			Quick Sort		
	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count
R1K	1.325	499500	2997	0.144	8678	19952	0.106	10715	17277
A1K	1.327	499500	2997	0.087	4941	19952	2.681	497506	749497
D1K	1.366	499500	2997	0.089	4941	19952	2.63	499500	752496
M1K	1.519	499500	2997	0.108	4932	19952	2.694	499500	753996

Table 2: Experimental results for 6k array

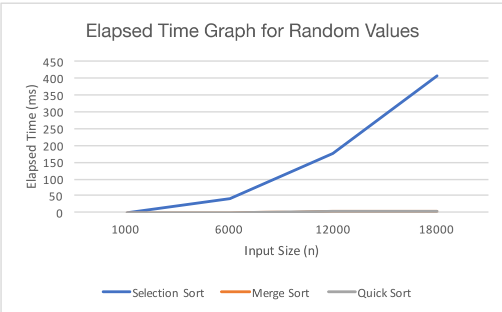
Input Size (n)	Selection Sort			Merge Sort			Quick Sort		
	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count
R6K	43.086	17997000	17997	1.03	67785	151616	0.807	88901	149743
A6K	44.228	17997000	17997	0.872	36668	151616	102.879	17997000	27014996
D6K	42.542	17997000	17997	0.573	36668	151616	95.151	17997000	27014996
M6K	49.892	17997000	17997	0.660	36668	151616	98.495	17997000	27014996

Table 3: Experimental results for 12k array

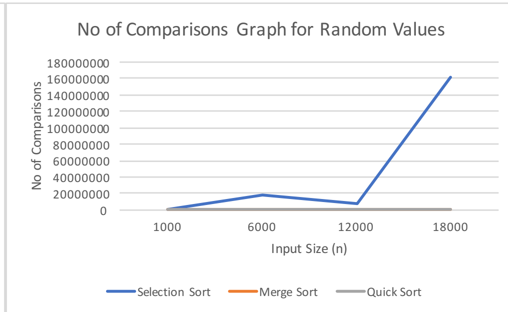
Input Size (n)	Selection Sort			Merge Sort			Quick Sort		
	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count
R12K	176.833	7199400	35997	2.111	147729	327232	1.667	192000	303750
A12K	186.366	7199400	35997	1.618	79325	327232	420.665	71970006	107993997
D12K	181.855	7199400	35997	1.295	79325	327232	401.927	71994000	108029996
M12K	192.943	7199400	35997	1.347	79325	327232	401.702	71994000	108047996

Table 4: Experimental results for 18k array

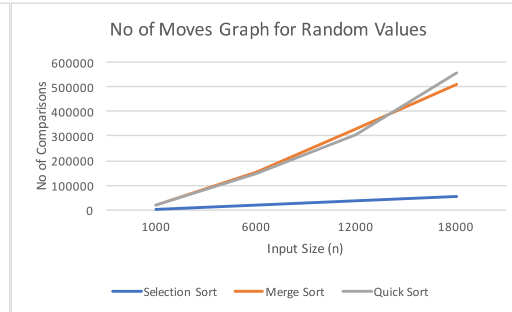
Input Size (n)	Selection Sort			Merge Sort			Quick Sort		
	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count	Elapsed Time	Comp Count	Move Count
R18K	408.688	161991000	53997	3.331	231918	510464	2.808	308963	558083
A18K	431.956	161991000	53997	2.056	124654	510464	895.742	161991000	243044996
D18K	429.688	161991000	53997	1.909	124654	510464	927.593	161991000	243044996
M18K	466.108	161991000	53997	2.123	124640	510464	852.951	161991000	243071996



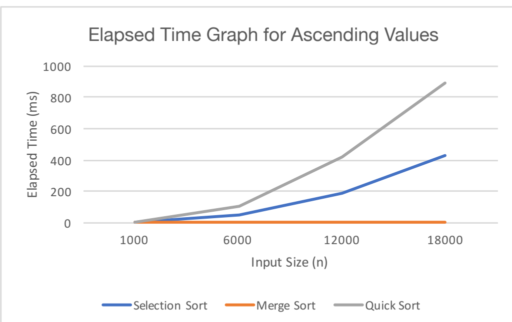
Elapsed Time Graph for Random Values
Elapsed Time (ms)
450
400
350
300
250
200
150
100
50
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



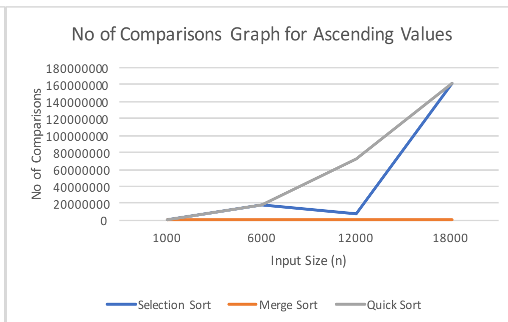
No of Comparisons Graph for Random Values
No of Comparisons
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



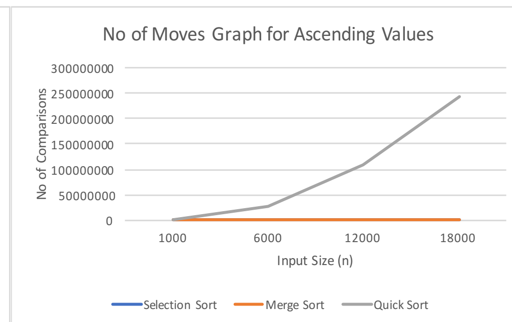
No of Moves Graph for Random Values
No of Comparisons
600000
500000
400000
300000
200000
100000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



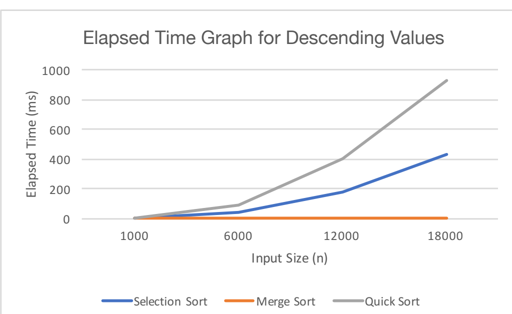
Elapsed Time Graph for Ascending Values
Elapsed Time (ms)
1000
800
600
400
200
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



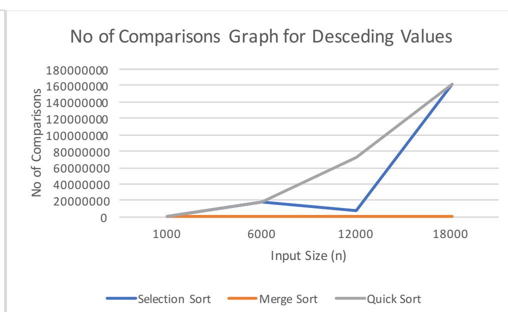
No of Comparisons Graph for Ascending Values
No of Comparisons
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



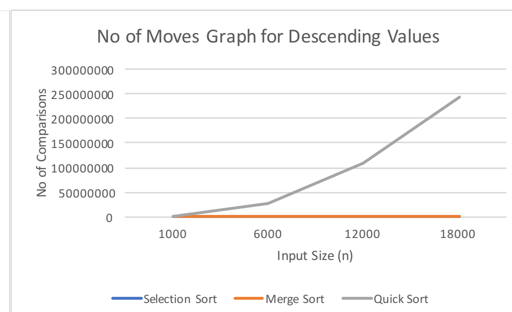
No of Moves Graph for Ascending Values
No of Comparisons
300000000
250000000
200000000
150000000
100000000
50000000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



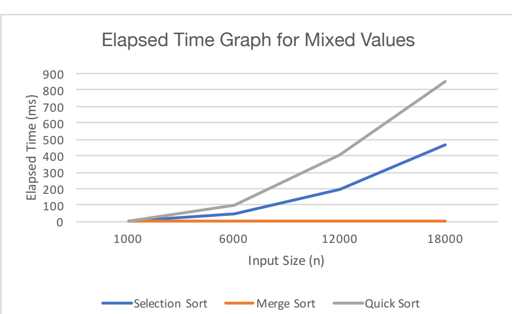
Elapsed Time Graph for Descending Values
Elapsed Time (ms)
1000
800
600
400
200
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



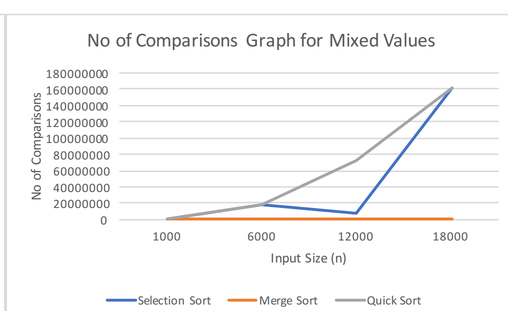
No of Comparisons Graph for Desceding Values
No of Comparisons
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



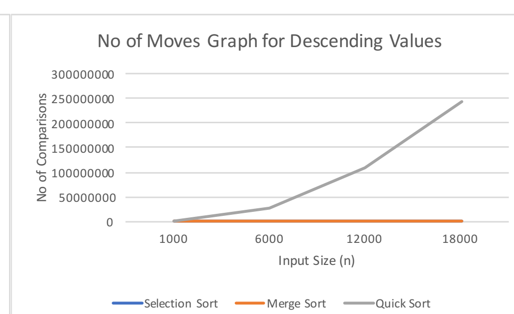
No of Moves Graph for Descending Values
No of Comparisons
300000000
250000000
200000000
150000000
100000000
50000000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



Elapsed Time Graph for Mixed Values
Elapsed Time (ms)
900
800
700
600
500
400
300
200
100
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



No of Comparisons Graph for Mixed Values
No of Comparisons
180000000
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort



No of Moves Graph for Descending Values
No of Comparisons
300000000
250000000
200000000
150000000
100000000
50000000
0
1000
6000
12000
18000
Input Size (n)
Selection Sort
Merge Sort
Quick Sort

For the “Selection Sort” algorithm the theoretical time complexity is given as $O(n^2)$ in every (best-average-worst) case. As it can be seen from the tables and graphs elapsed time for selection sort increases on a polynomial rate close to $O(n^2)$ which is much higher in contrast to the other sort algorithms. In addition to this, for the case of 1K values the different type of inputs such as random, ascending, descending and mixed does not affect the elapsed time and it can be seen from the first column of graphs that the line for the elapsed time of the algorithm is identical to each other. Also number of comparisons and number of moves can be interpreted from the following theoretical values of $O(n^2)$ and $O(n)$ for all cases and from the second row of graphs although theres a derivation in the number of comparisons between the interval 6000-12000 the overall curve is again reminiscent of the theoretical values and for the number of moves it can be clearly seen from the graphs that it is much lesser that the number of comparisons and therefore consistent with the theoretical formula.

For the “Merge Sort” algorithm theoretical time complexity is given as O(n*logn) in every (best-average-worst) case which is expected to be much lower than the selection sort algorithm. The difference in the running times of these algorithms can be observed from every type of data from the tables and from the graphs. Merge sort apart from the two other has the most stability in terms of running time, number of comparisons and moves. As it can be seen from the graphs the algorithm follows a path nearly close to linear O(n) in terms of execution time and does not differ when the type or the amount of data changes which shows that the theoretical results are similar to the experimental ones. Since the algorithm uses recursion and works by dividing the array into half in each call it uses an extra memory which is something the other two algorithms do not use and therefore it’s not an in-place algorithm.

For the “Quick Sort” algorithm theoretical time complexity is given as O(n*logn) for the best case scenario however, for the worst case scenario the algorithm acts as $O(n^2)$ which is equal to the time complexity of selection sort. As it can be seen from the tables and the graphs the algorithm acts differently when the type of input changes from random to ascending values. Since quick sort uses the pivot value as the element in the first index when the given array has ascending values the algorithm act like $O(n^2)$ because it has to reverse the array and the distribution of the values are not ideal. As a consequence the number of comparisons and moves also increases tremendously which can be seen from the graphs that in some cases the lines of selection sort and quick sort are nearly identical which is the caused by reason explained above. When compared to the theoretical results it can be said that it’s parallel with the results obtained from the experimental values.

In general the experimental results did not cause any misconception. In order to be sure about the differences in execution times the program is tested in both Xcode C++ compiler and GNU C++ compiler on the Dijkstra server. Even though there were numerical differences the overall behaviors of the algorithms did not differ from the results obtained previously.