CS202 Fall 2017 Homework 1

Section 1
Selim Firat Yilmaz
21502736
firat.yilmaz@ug.bilkent.edu.tr

October 16, 2017

Question 1

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

$O(10^{\pi}) < O(\log n) < O(n^{0.0001}) < O(\sqrt{n}) < O(n) < O(n * \log n) = O(\log n!) < O(2^n) < O(n!) < O(n^n)$

(b) In loop A, inner loop executes i times for each i and outer loop executes n times. Thus, it is $\Theta(n * (n-1)/2) = \Theta(n^2)$.

$$T_A(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$$

In loop B, while executes $logn$ times and for executes i times for each i. Since inner loop depends on i, it executes when $i = 1, 2, 4, ..., 2^n$. Thus, the running time complexity is $\Theta(2^{logn}) = \Theta(n)$.

$$T_B(n) = \sum_{i=0}^{log_2 n} 2^i = 2^{log_2 n+1} - 1 = 2n - 1 = \Theta(n)$$

In loop C, first while executes $log_2 n$ times, and second (inner) while executes when $i * i = 1, 4, 9, 16, .., n^2$. Thus, second for executes $logn(logn - 1)(2logn - 1)/6$ times. The most inner loop executes n times independent of any other variable. Thus the complexity is $\Theta(nlogn(logn + 1)(2logn + 1)/6) = \Theta(nlog^3 n)$.

$$T_c(n) = \frac{1}{6} nlogn(logn + 1)(2logn + 1) = \Theta(nlog^3 n)$$

(c) **Merge Sort Worst Case Recurrence Relation**

(Assume $n = 2^k$. If $x \neq 2^k$, problem can be reduced to the solution below by computing T(n - 1) since $n - 1 = 2^k$)

$$T(n) = 2T(\frac{n}{2}) + c_1 n + c_2$$

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

According to the Master Theorem (case 2),

$$\texttt{If } T(n) = aT(n/b) + \Theta(n^{log_b a} log^k n) \quad \texttt{with } k \geq 0\texttt{, then } T(n) = \Theta(n^{log_b a} log^{k+1} n).$$

$$\texttt{We have } a = 2,\ b = 2,\ \texttt{and } k = 0.$$

$$\texttt{Thus, } T(n) = \Theta(nlogn) = O(nlogn)$$

Quick Sort Worst Case Recurrence Relation

$$T(0) = T(1) = 0$$

$$T(n) = n + T(n-1)$$

$$\begin{aligned} T(n) - T(n-1) &= n \\ T(n-1) - T(n-2) &= n-1 \\ T(2) - T(n-3) &= n-2 \\ &\ldots \\ T(2) - T(1) &= 2 \\ T(1) &= 0 \end{aligned}$$

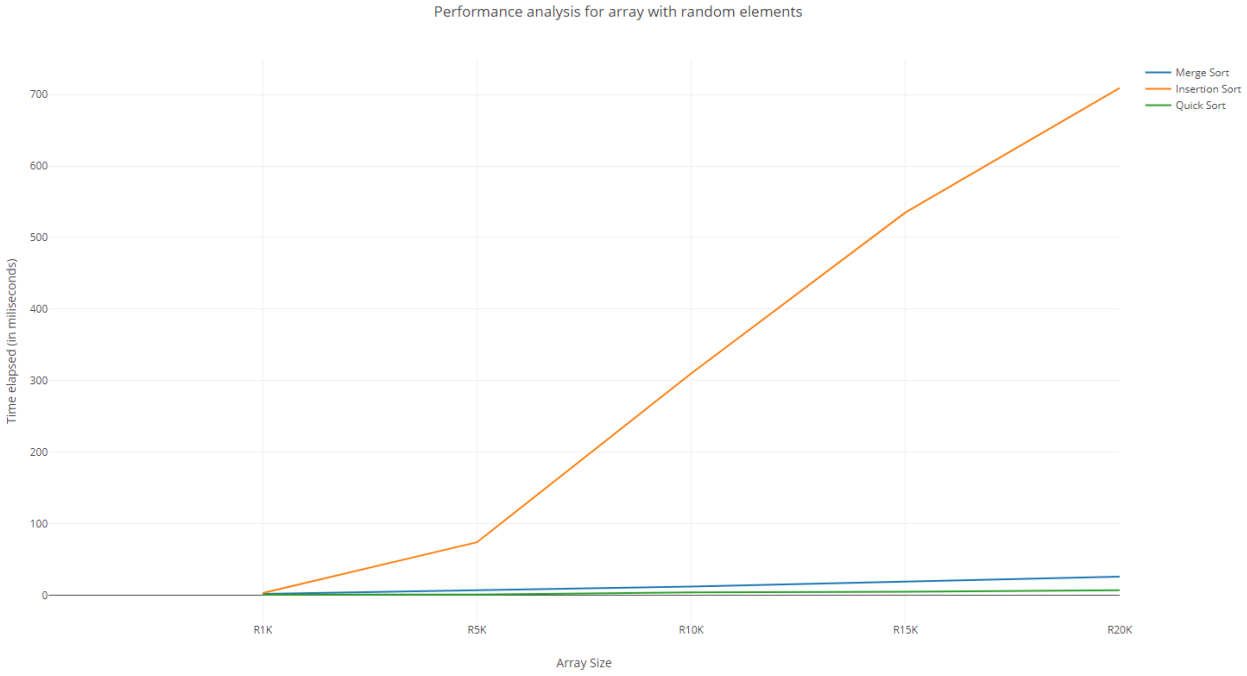
Hence,
$T(n) = n + (n-1) + ... + 3 + 2 = \frac{(n-1)(n+1)}{2} = O(n^2)$

Question 3

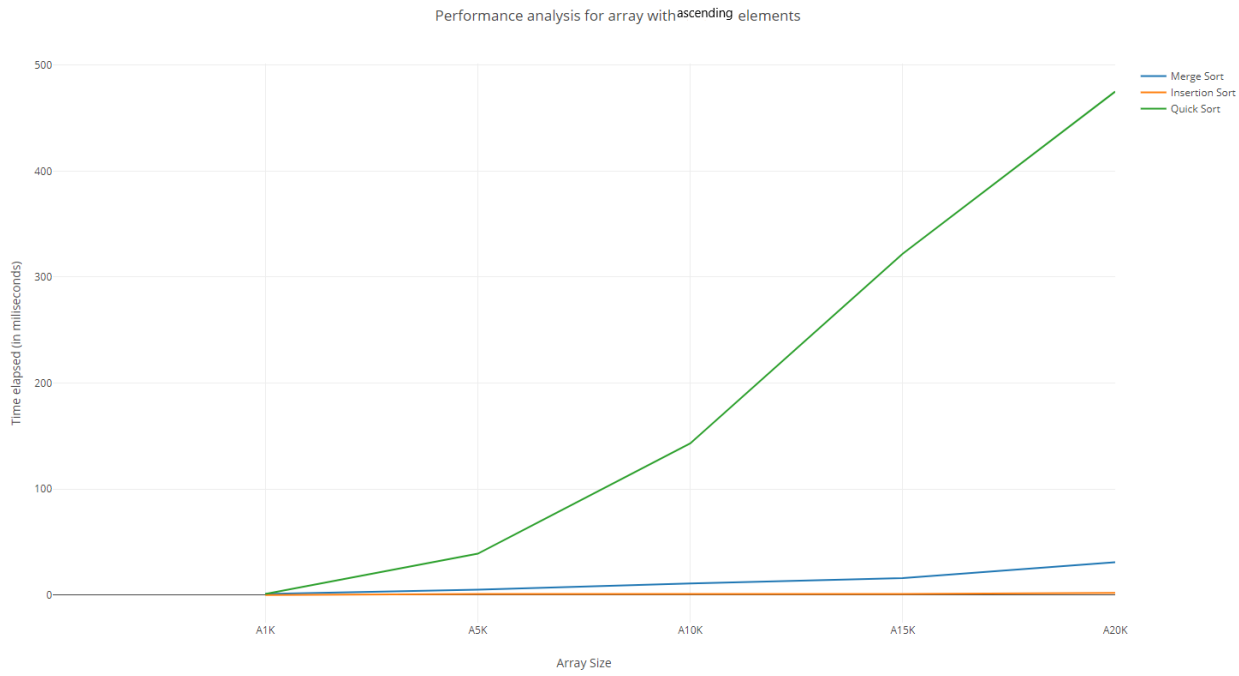
(a) Performance Analysis Table

Array	Elapsed Time (in milliseconds)			Number of Comparisons			Number of Data Moves		
	Insertion Sort	Merge Sort	Quick Sort	Insertion Sort	Merge Sort	Quick Sort	Insertion Sort	Merge Sort	Quick Sort
R1K	2	2	1	249143	8729	12617	250142	19952	21051
R10K	533	15	4	44576318	120489	161160	44586317	267232	244125
R20K	797	24	5	100583692	260855	356235	100603691	574464	526755
A1K	1	1	1	999	5044	499500	1998	19952	2997
A10K	1	8	91	9999	69008	49995000	19998	267232	29997
A20K	2	16	334	19999	148016	199990000	39998	574464	59997
D1K	3	1	2	499500	4932	499500	500499	19952	752997
D10K	238	7	116	49995000	64608	49995000	50004999	267232	75029997
D20K	1026	13	648	199990000	139216	199990000	200009999	574464	300059997

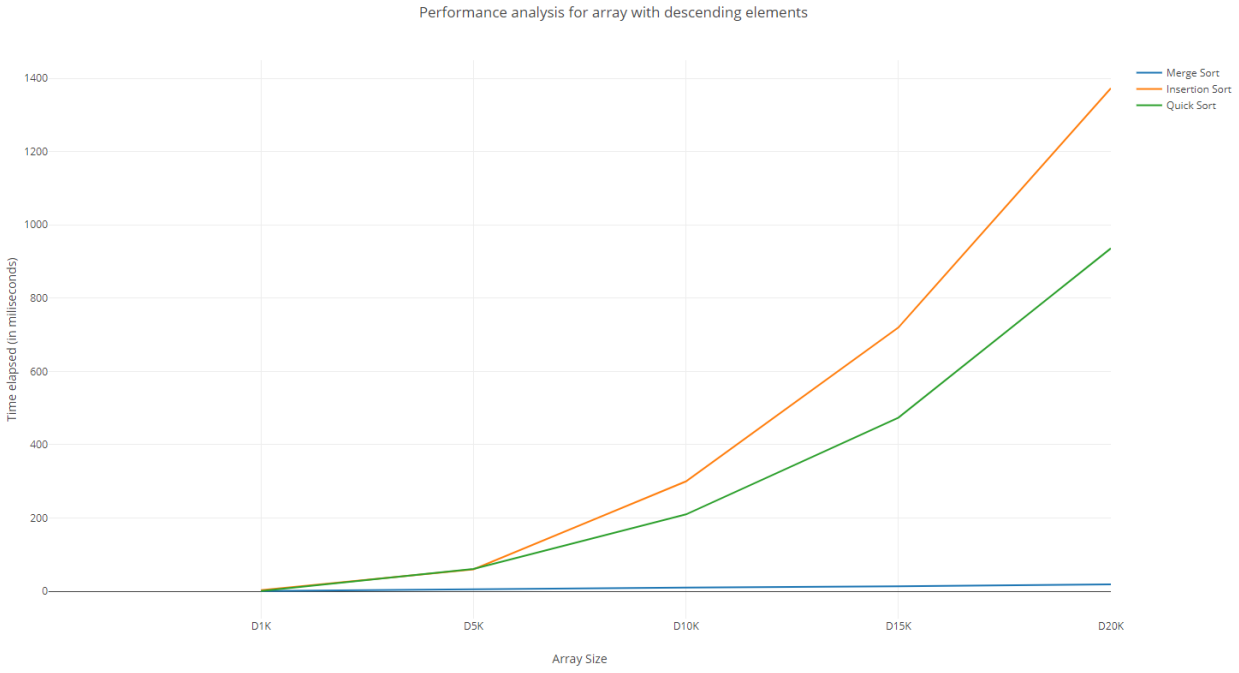
Performance Analysis Chart of Randomly Ordered Elements



Performance Analysis Chart of Ascendingly Ordered Elements



Performance Analysis Chart of Descendingly Ordered Elements



Insertion Sort has $O(n^2)$ worst/average case running time complexity. As expected, in arrays of descendingly and randomly sorted elements, it took the most time to run and its graph seems like n^2's graph. In an array of ascendingly sorted elements, it

took the least time since this case is Insertion Sort's best case and has O(n) complexity.

Merge Sort has $O(nlogn)$ worst/average/best case running time complexity. As expected it worked best in descending, and worked just a bit worse than the best one in others in terms of running time.

Since Quick Sort has $O(n^2)$ worst time running time complexity, it took lots of time to sort an array of ascendingly sorted elements which is Quick Sort's worst case. Nevertheless, Quick Sort performed as best one in terms of running time in sorting an array of randomly sorted elements as expected since it has $O(nlogn)$ average running time complexity.

When should insertion sort algorithm be preferred over merge sort and quick sort algorithms?
It is easier to write since it has the least lines of code. Thus, it can be used when quick implementation is needed. However, since it has $O(n^2)$ average time complexity, it should not be used to sort large array or the time allowed for an array to be sorted is need to be big enough so that the timing is ignored. Another aspect is that insertion sort has O(1) space complexity whereas quick sort has $O(logn)$ space complexity and merge sort has $O(n)$ space complexity. Thus, it can also be preferred over quick sort and merge sort when space is so limited.

When should merge sort algorithm be preferred over quick sort algorithm?
A requirement to do this is space because merge sort has $O(n)$ space complexity whereas quick sort has $O(logn)$ space complexity. If this requirement is satisfied, and if there is high probability that an array may be sorted or almost-sorted, then merge sort can be preferred over quick sort because merge sort has guaranteed $O(nlogn)$ running time complexity whereas quick sort can have $O(n^2)$ in its worst case. Since merge sort uses less comparisons, it can be used when comparison has high costs such as comparing database rows. (It is assumed that databases has high comparison cost because to compare, you need to access these data but to move data, you may just move indexes.)