/*
* Title : Algorithm Efficiency and Sorting
* Author : Deniz Yüksel
* ID : 21600880
* Section : 1
* Assignment : 1
* Description : This is the document that contains solutions for question
* 1 and 3.
*/

CS 202 HOMEWORK 1

Question 1)

a) Show that *f(n) =* $4n^5 + 3n^2 + 1$ *is* $O(n^5)$ by specifying appropriate *c* and *n0* values in Big-O definition.

Solution: T(n) = O(f(n)) such that T(n) <= c * f(n) when n >= n_0 where c and n_0 are positive constants.

$4n^5 + 3n^2 + 1 <= c * n^5$

$= 3n^2 + 1 <= n^5(c – 4)$

$= 3/n^3 + 1/n^5 <= c – 4$

$= 3/n^3 + 1/n^5 + 4 <= c.$

Select n as 1 => 3 + 1 + 4 <= c. Therefore c can be 8.

The answer for c and n_0 values can be 8 and 1 respectively.

b) Find the asymptotic running times (in Θ notation, tight bound) of the following recurrence equations by using the repeated substitution method. Show your steps in detail.

1.T(n) = T(n - 1) + n^2, T(1) = 1
2.T(n) = 2 T(n/2) + n/2, T(1) = 1

Solution for 1:

$T(n-1) = T(n-2) + (n-1)^2$

$T(n-2) = T(n-3) + (n-2)^2$.

$T(n) = T(n-2) + (n-1)^2 + n^2$

…

$T(n) = T(n-k) + (n-(k+1))^2 + (n-(k+2))^2 + … + n^2$

$T(n) = T(1) + 2^2 + 3^2 + 4^2 + \ldots + n^2$,   for $n - k = 1$.

$T(n) = 1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2$

$T(n) = n * (n + 1) * (2n + 1) / 6 = n^3/3 + n^2/2 + n/6 \Rightarrow \Theta(n^3)$.

Solution for 2:

$T(n) = 2 * T(n/2) + n/2$

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

$T(n/8) = 2 * T(n/16) + n/16$.

$T(n) = 2 * [2 * T(n/4) + n/4] + n/2$

$T(n) = 2^2 * T(n/4) + n/2 + n/2$

$T(n) = 2^3 * T(n/8) + n/2 + n/2 + n/2$

…

$T(n) = 2^m * T(n/ 2^m) + n/2 + n/2 + \ldots + n/2$, where $n/2^m = 1$.

$n = 2^m \Rightarrow m = \log_2 n$.

$T(n) = n + (m*n) / 2 = n + (\log_2 n * n)/2 \Rightarrow \Theta(n * \log n)$.

c) Sorting [8, 4, 5, 1, 9, 6, 2, 3] in ascending order with

Selection sort: (unsorted | sorted)

[8, 4, 5, 1, 9,6, 2, 3 |] -> [8, 4, 5, 1, 3,6, 2, | 9] -> [2, 4, 5, 1, 3,6, | 8, 9] ->

[2, 4, 5, 1, 3, | 6, 8, 9] -> [2, 4, 3, 1, | 5, 6, 8, 9] -> [2, 1, 3, | 4, 5, 6, 8, 9] ->

[2, 1, | 3, 4, 5, 6, 8, 9] -> [1, | 2, 3, 4, 5, 6, 8, 9]. Array is sorted.

Bubble sort:

[8, 4, 5, 1, 9, 6, 2, 3] -> [4, 5, 1, 8, 6, 2, 3, 9] (After the first pass).

[4, 5, 1, 8, 6, 2, 3, 9] -> [4, 1, 5, 6, 2, 3, 8, 9] (After the second pass).

[4, 1, 5, 6, 2, 3, 8, 9] -> [1, 4, 5, 2, 3, 6, 8, 9] (After the third pass).

[1, 4, 5, 2, 3, 6, 8, 9] -> [1, 2, 4, 3, 5, 6, 8, 9] (After the fourth pass).

[1, 2, 4, 3, 5, 6, 8, 9] -> [1, 2, 3, 4, 5, 6, 8, 9] (After the fifth pass).

Question 3)

Table (1) for Elapsed Time per Algorithm and Arrays with Variety of Sizes

	Elapsed time (miliseconds)			
Arrays	Insertion Sort	Merge Sort	Quick Sort	Hybrid Sort
R1K	0	15	0	0
R7K	46	47	1	16
R14K	171	0	3	63
R21K	328	0	6	125
A1K	0	0	0	0
A7K	0	47	47	46
A14K	0	0	198	203
A21K	0	16	454	447
D1K	0	22	4	3
D7K	93	31	172	172
D14K	312	0	688	717
D21K	672	0	1641	1622

Table (2) for Key Comparison Count per Algorithm and Arrays with Variety of Sizes

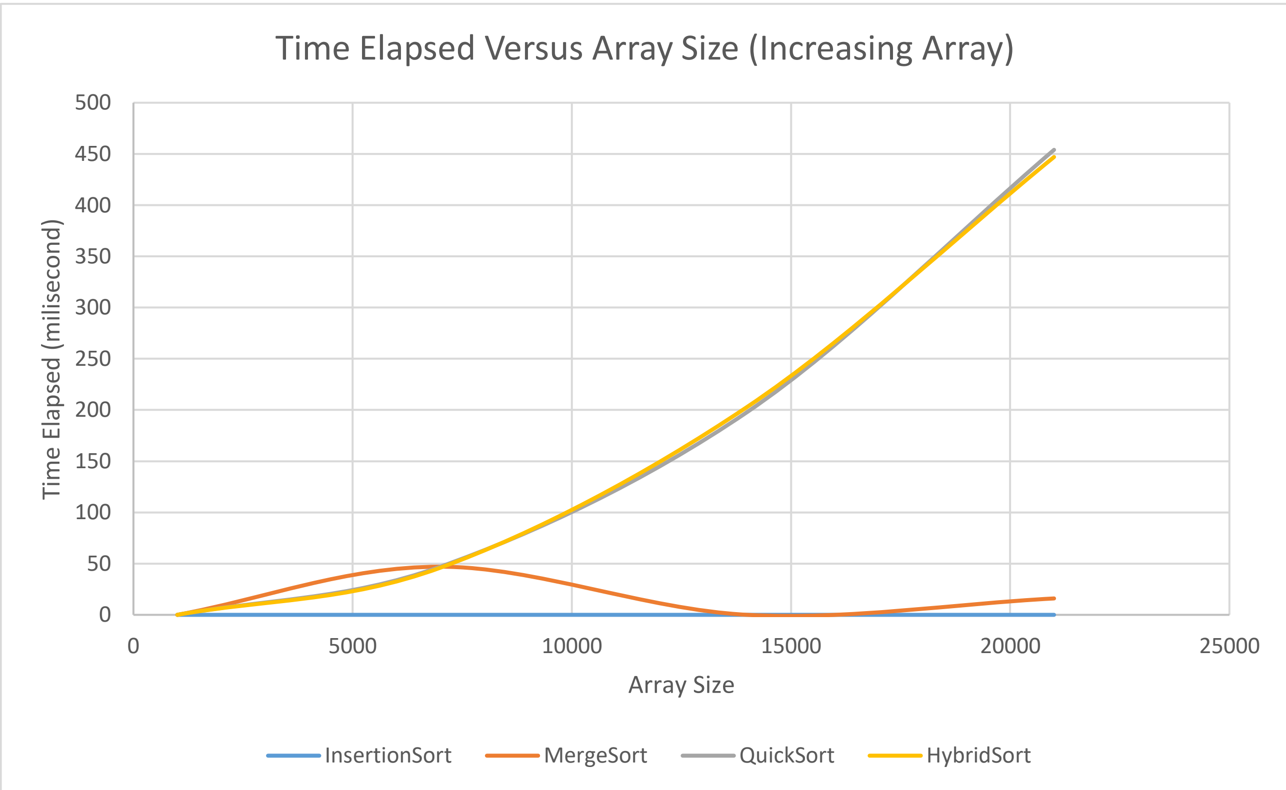
	Key Comparison Count			
Arrays	Insertion Sort	Merge Sort	Quick Sort	Hybrid Sort
R1K	254378	8715	10660	79162
R7K	12263173	80744	101893	3438521
R14K	48862665	175204	230412	13641412
R21K	110083299	275475	387526	30778464
A1K	999	5044	499500	24499464
A7K	6999	46180	24496500	24496464
A14K	13999	99360	97993000	97992964
A21K	20999	156508	220489500	220489464
D1K	501498	19952	251998	252513
D7K	24496500	43628	24496500	24500004
D14K	97993000	94256	97993000	98000004
D21K	220489500	146724	220489500	220500004

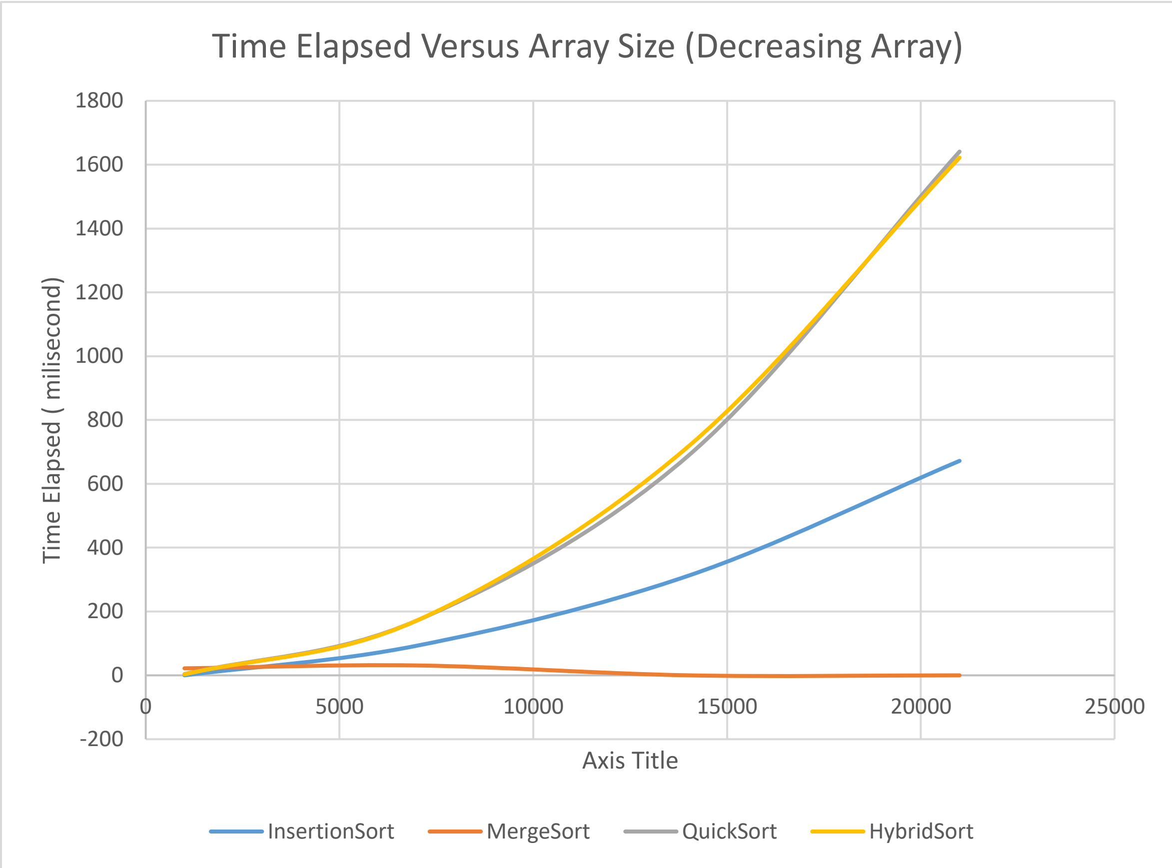
Table (3) for Data Move Count per Algorithm and Arrays with Variety of Sizes

	Data Move Count			
Arrays	Insertion Sort	Merge Sort	Quick Sort	Hybrid Sort
R1K	255376	19952	7309	76124
R7K	12270165	179616	52965	3392269
R14K	48876656	387232	139855	13556152
R21K	110104286	606464	229372	30628024
A1K	999	19952	1998	1989
A7K	6999	179616	13998	13989
A14K	13999	387232	27998	27989
A21K	20999	606464	41998	41989
D1K	501498	19952	251998	252513
D7K	24510498	179616	12263998	12267513
D14K	98020998	387232	49027998	49035013
D21K	220531498	606464	110291998	110302513

Graphs


Time Elapsed Versus Array Size (Random Elements)
350
300
250
200
150
100
50
0
-50
0
5000
10000
15000
20000
25000
Time Elapsed (milisecond)
Array Size
InsertionSort
MergeSort
QuickSort
HybridSort


Time Elapsed Versus Array Size (Increasing Array)
500
450
400
350
300
250
200
150
100
50
0
0
5000
10000
15000
20000
25000
Time Elapsed (milisecond)
Array Size
InsertionSort
MergeSort
QuickSort
HybridSort



When we look at graph 1 for random elements inside an array, we see that insertion sort takes the longest time to complete. This occasion correlates with the theoretical case, where insertion sort takes $O(n^2)$.

On the other hand, hybrid sort and quick sort are expected to have the same amount of time to complete. However, I believe the graph 1 can have experimental errors. It is because hybrid sort takes less time than quick sort theoretically and insertion sort is more efficient for sorting low number of inputs.

As expected, merge sort is slower than insertion sort when the array size is low. However as array size gets larger, merge sort creates a considerable amount of difference between itself and insertion sort. It also gets faster than quick sort where in the graph 1, quick sort depicts a trail almost like a constant operation.

When we inspect graph 2 which shows the same information but with an array of ascending elements, we see that hybrid sort and quicksort algorithms almost perform the same. However, for the sake of displaying all information, we cannot really see the difference of hybrid sort for small sizes. Predictibly, hybrid sort works better for small array sizes, than that of quick sort. Because we are taking the pivot as the last point, the performance of quick and hybrid sort are the worst case which is $O(n^2)$. Also, merge sort works with a predicted performance which is O(nlogn) in both worst and average case.

In graph 3, where the array elements decrease continuously, it is the same case with the previous graph for quick sort and hybrid sort. Both algorithms perform their worst case again. Independently, merge sort performs with O(nlogn) once again.

Generally, and as concluded from the graphs, insertion sort is prefferable to quick sort and merge sort when the input size is low. We should also choose merge sort rather than quick sort when the array's elements are random. It is because quick sort performs its worst case for arrays that have increasing or decreasing elements. Obviously, the case is our quick sort where the pivot is last element.

Also explained before in this report, hybrid sort has advantages to quick sort in the case of small inputs.