CS 202
Fundamental Structures of Computer Science II

FALL 2021-2022

HOMEWORK 1

NAME: Ferhat

SURNAME: Korkmaz

ID: 21901940

SECTION: 01

Question 1-)

- $T(n) = 3T(n/3) + n$ where $T(1) = 1$ and n is an exact power of 3.

Solution:

$T(n) = 3(3T(n/9) + n/3) + n$

$T(n) = 3(3(3T(n/27) + n/9) + n/3) + n$ *This will keep going until…*

$T(n) = 3^{\log_3(n)} * T(1) + n + n + n \ldots + n$ *Yellow highlighted part is equal to n and green highlighted part is equal to $nlog_3{}^{(n)}$.*

$T(n) = n + nlog_3{}^{(n)}$

Big-O notation is $O(nlog(n))$

- $3T(n/2) + 1$ where $T(1) = 1$ and n is an exact power of 2.

Solution:

$T(n) = 3(3T(n/4) + 1) + 1$

$T(n) = 3(3(3T(n/8) + 1) + 1) + 1$ *This is like what I solved above. This will keep going…*

$T(n) = 3^{\log_2(n)} * T(1) + 1 + 3 + 9\ldots + 3^{\log_2(n)}$

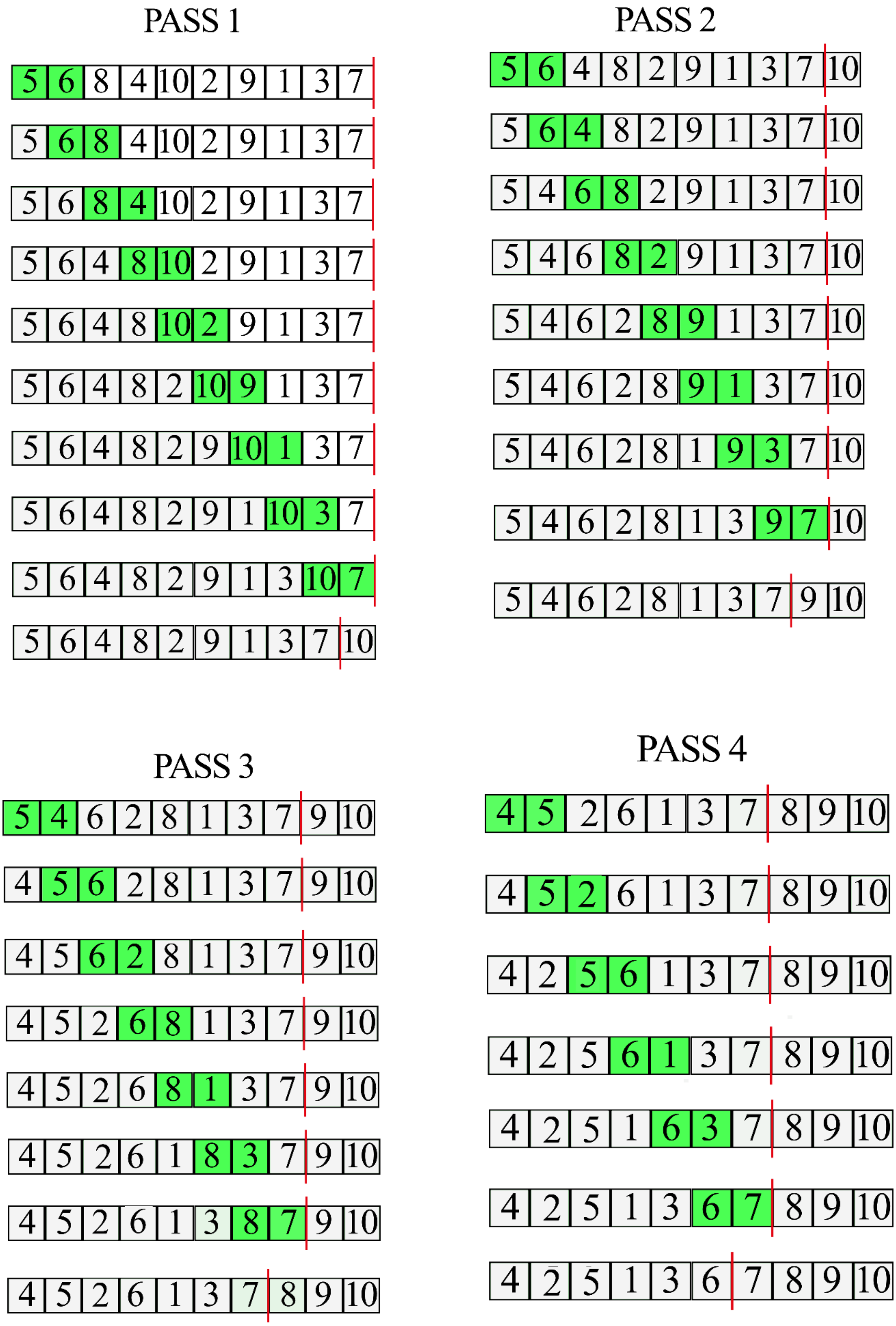
$T(n) = 2 * 3^{\log_2(n)}$

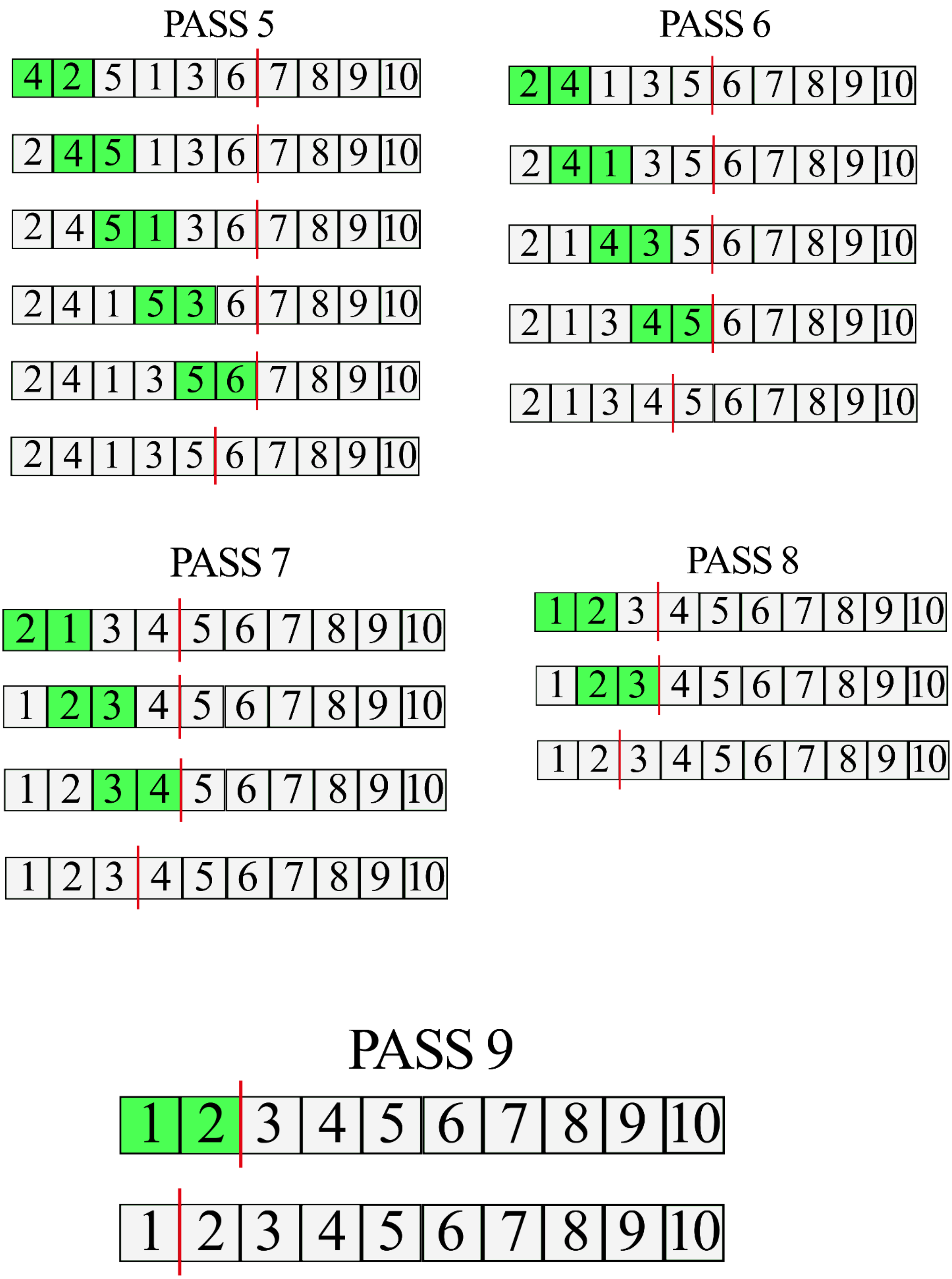
$T(n) = 2 * n^{\log_2(3)}$

Big-O notation is $O(n^{log_2(3)})$

Question 1-) Tracing

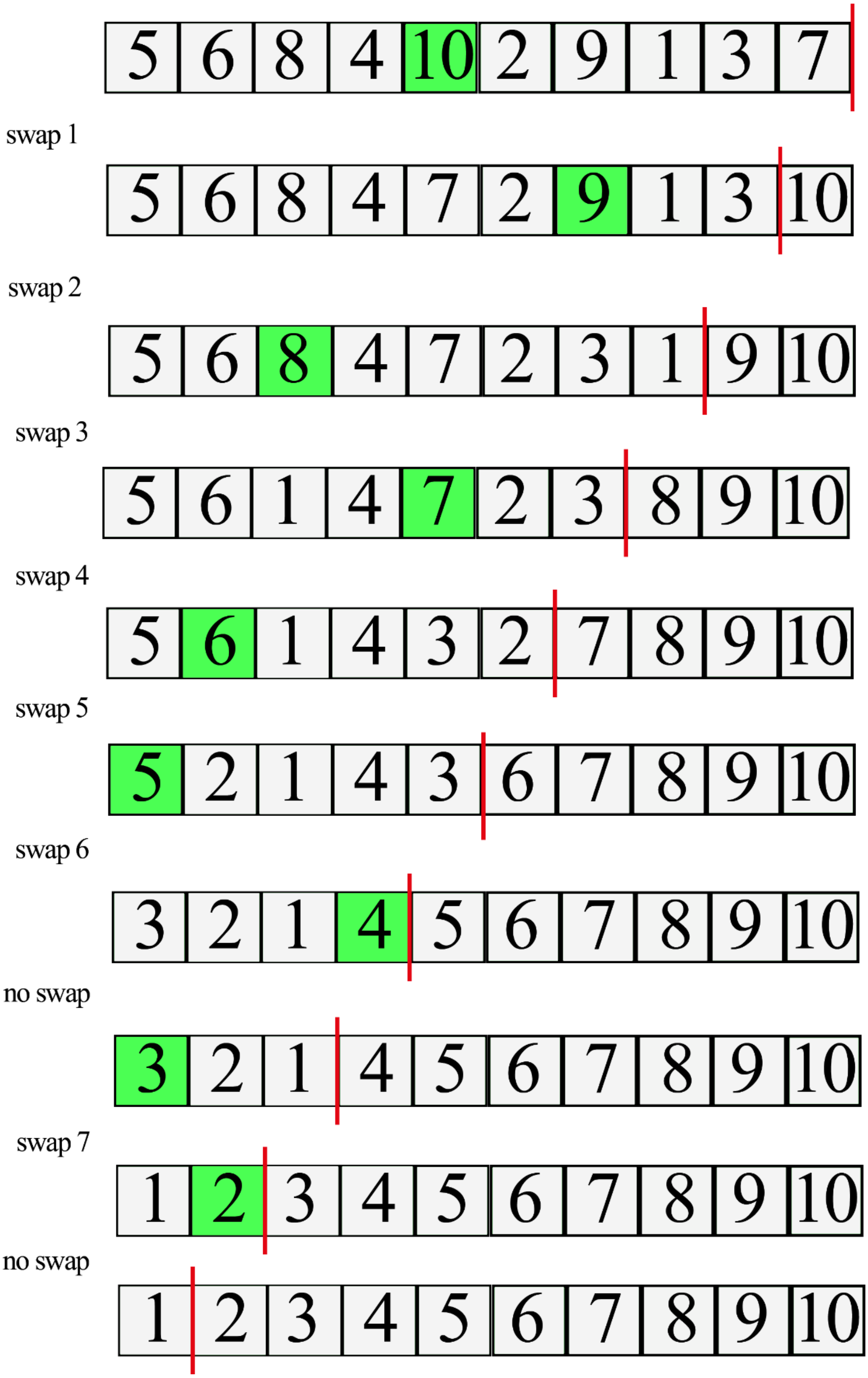
- **Bubble Sort**





Array is sorted.

- **Selection Sort**



Question 1-) Worst case of Quick Sort

Worst case of the quick sort occurrs when the pivot is chosen the largest or the smalles element in the array. In this case, one of the lists always will be empty causing that quick sort will be working only one sublist during sorting. Following recurrence relation explains the algorithm’s worst case.

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

T(n) = T(n - 2) + n -2 + n -1 *This will keep going..*

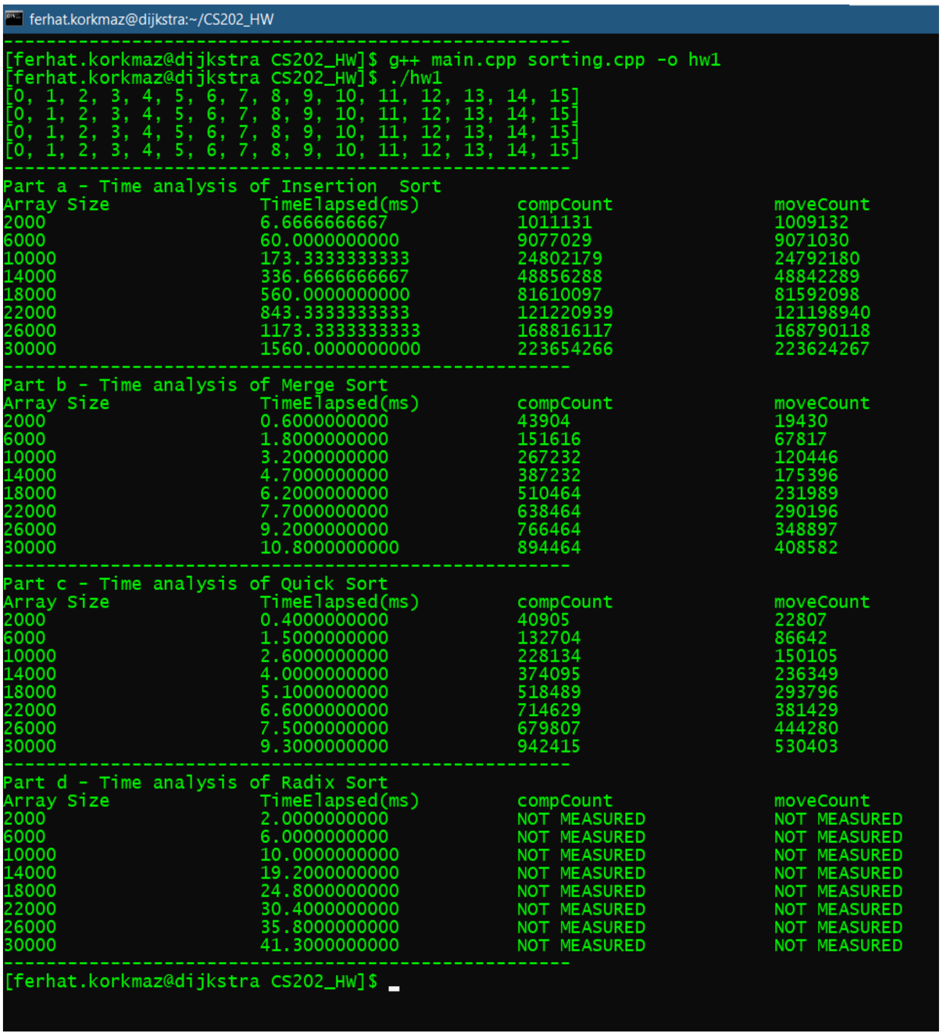
…

T(n) = T(1) + $\sum_{i=0}^{i=n-1} n - i$

T(n) = n(n-1) / 2

Big-O notation is $O(n^2)$

Question 2-)



```
ferhat.korkmaz@dijkstra:~/CS202_HW
-----------------------------------------------------
[ferhat.korkmaz@dijkstra CS202_HW]$ g++ main.cpp sorting.cpp -o hw1
[ferhat.korkmaz@dijkstra CS202_HW]$ ./hw1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
-----------------------------------------------------
Part a - Time analysis of Insertion  Sort
Array Size      TimeElapsed(ms)     compCount       moveCount
2000            6.6666666667        1011131         1009132
6000            60.0000000000       9077029         9071030
10000           173.3333333333      24802179        24792180
14000           336.6666666667      48856288        48842289
18000           560.0000000000      81610097        81592098
22000           843.3333333333      121220939       121198940
26000           1173.3333333333     168816117       168790118
30000           1560.0000000000     223654266       223624267
-----------------------------------------------------
Part b - Time analysis of Merge Sort
Array Size      TimeElapsed(ms)     compCount       moveCount
2000            0.6000000000        43904           19430
6000            1.8000000000        151616          67817
10000           3.2000000000        267232          120446
14000           4.7000000000        387232          175396
18000           6.2000000000        510464          231989
22000           7.7000000000        638464          290196
26000           9.2000000000        766464          348897
30000           10.8000000000       894464          408582
-----------------------------------------------------
Part c - Time analysis of Quick Sort
Array Size      TimeElapsed(ms)     compCount       moveCount
2000            0.4000000000        40905           22807
6000            1.5000000000        132704          86642
10000           2.6000000000        228134          150105
14000           4.0000000000        374095          236349
18000           5.1000000000        518489          293796
22000           6.6000000000        714629          381429
26000           7.5000000000        679807          444280
30000           9.3000000000        942415          530403
-----------------------------------------------------
Part d - Time analysis of Radix Sort
Array Size      TimeElapsed(ms)     compCount       moveCount
2000            2.0000000000        NOT MEASURED    NOT MEASURED
6000            6.0000000000        NOT MEASURED    NOT MEASURED
10000           10.0000000000       NOT MEASURED    NOT MEASURED
14000           19.2000000000       NOT MEASURED    NOT MEASURED
18000           24.8000000000       NOT MEASURED    NOT MEASURED
22000           30.4000000000       NOT MEASURED    NOT MEASURED
26000           35.8000000000       NOT MEASURED    NOT MEASURED
30000           41.3000000000       NOT MEASURED    NOT MEASURED
-----------------------------------------------------
[ferhat.korkmaz@dijkstra CS202_HW]$ _
```

Figure 1: Result from BCC Linux machine.

Question 3-)

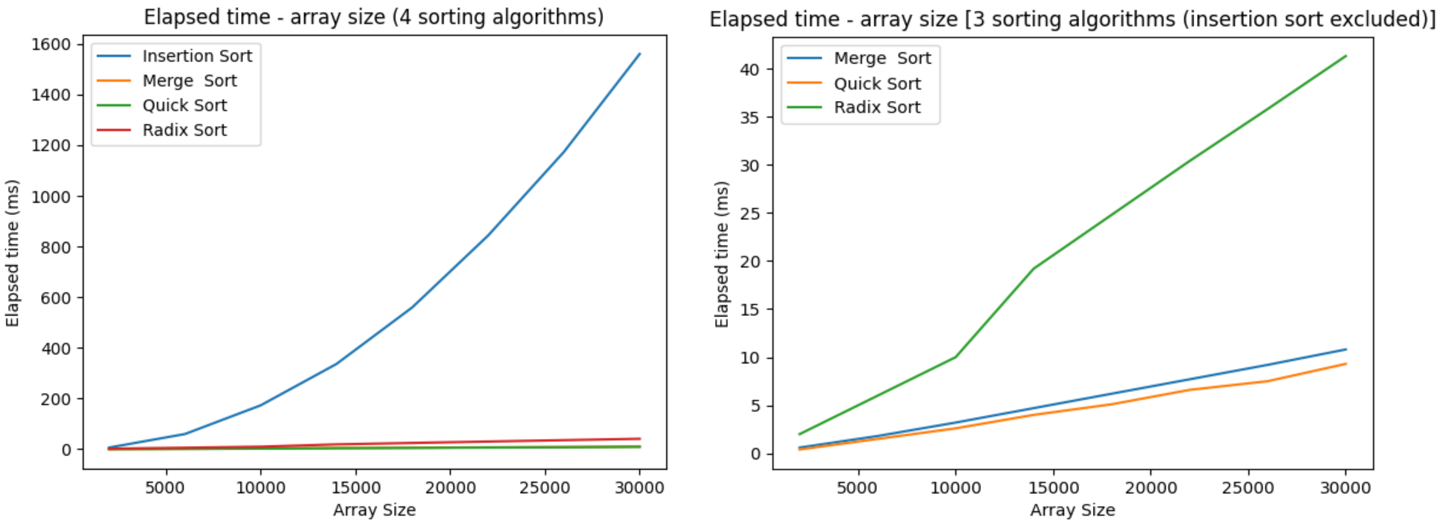


Figure 2: Plotted graphs of performance analysis of the algorithms (Python matplotlib library is used).

- I have two plots here due to insertion sort invading so much space causing preventing us to see merge sort, quick sort, and radix sort algorithms' performances properly.
- Quick sort always slightly better than merge sort and so much faster than radix sort and, of course, insertion sort when the arrays are randomly created.
- We know from the theoretical result that merge sort always shows *O(nlog(n))* for best, average, and worst cases. However, in the average-case, quick sort is performing *O(nlog(n))* and in the worst-case, it is $O(n^2)$.
- Nevertheless, quick sort algorithm outperformed the merge sort algorithm since merge sort requires extra memory for the *tempArray* that we can see in the slides. Creating and deallocating that tempArray everytime we merge took time. I think, that is why it is slightly slower than quick sort algorithm.
- Radix sort is an algorithm whose Big-O notation is *O(nk)*, where n is equal to size of the data list, and k is equal to number of digits. Radix sort algorithm outperformed the insertion sort algorithm in my experimental result. That is, of course, expected since insertion sort performs $O(n^2)$. In order for insertion sort to outperform radix sort, we

need a k such that k > n. In my testings k is between 0 and array size *x* 10. So, the result is what it is expected.

- If the arrays were inserted such that every element is increasing, that means array is already sorted. Quick sort will be outperformed by merge sort since quick sort would live its worst case, which is $O(n^2)$, yet, nothing would change for merge sort since it would keep performing $O(n\log(n))$. On the other hand, even the radix sort, would be faster than quick sort since it would perform $O(nk)$- unless k > n-. Also, insertion sort will outperform not only quick sort but also other sorting algorithms as well since it will have the time complexity $O(n)$ because this is its best case.