CS 201 Spring 2024

HOMEWORK 2

Name: Emine İrem

Surname: Esendemir

ID: 22202103

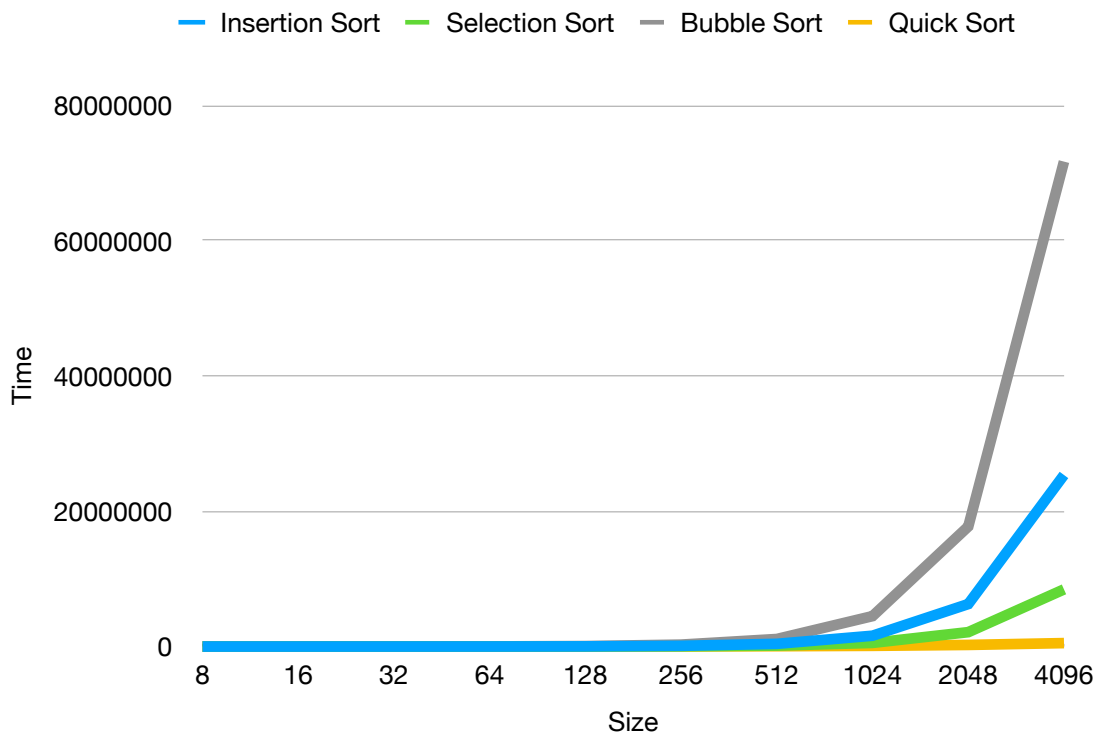
Section: 2

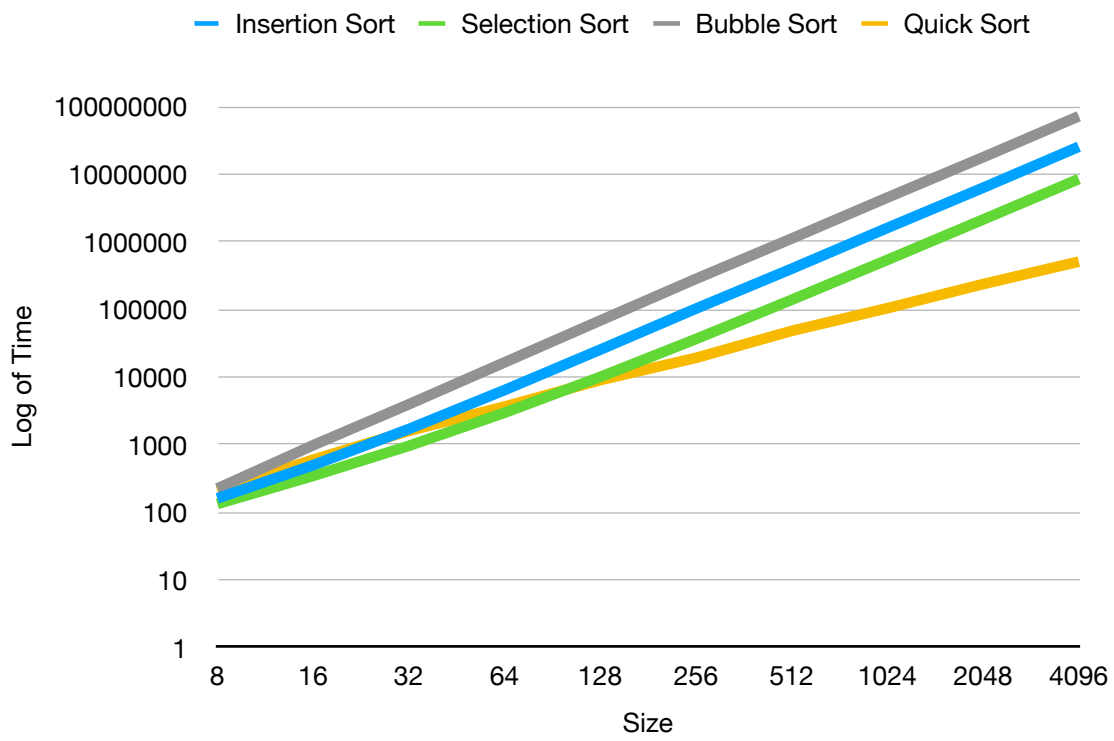
Task 1: Which Algorithm?

Time Estimation Table:

n	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort
2^3	161	133	255	208
2^4	498	345	980	600
2^5	1703	961	3951	1600
2^6	6415	2961	16442	3656
2^7	25382	10033	68386	8993
2^8	102579	36465	282432	19199
2^9	398627	138481	1114031	48073
2^{10}	1599536	539121	4495895	104034
2^{11}	6278830	2126833	17740513	236434
2^{12}	25361223	8447985	71684597	507923

Plot of change of time according to the size:





Result Discussion:

These tables and graphs show the results of tests with random arrays. Here is the examination of these 4 algorithm:

As it can be also seen from the graphs and table, the worst one is the Bubble Sort. Bubble Sort has time complexity $O(n^2)$ in the average case and worst case and O(n) time complexity as the best case, which is when the array is already sorted. In the task 1, random arrays are created. Therefore, the time complexity for Bubble Sort is $O(n^2)$. As the worker spend more time doing swaps compared to making comparisons and Bubble Sort is mostly based on swapping the consecutive elements, Bubble Sort gives worse result compared to the other algorithms with $O(n^2)$, which are Insertion and Selection Sort.

Selection Sort has always the same time spent on the array independent from the array's elements' order. It has nested two for loops, so it has time complexity is $O(n^2)$. Insertion sort has best time complexity as O(n) when the array is already sorted and has average and worst time complexity as $O(n^2)$, as the arrays are created randomly, the time complexity is $O(n^2)$.

Quick Sort has best and average time complexity as O(n*logn) and worst time complexity as $O(n^2)$. The worst case happens when the array is already sorted. As the arrays are random here, the time complexity is O(n*logn).

To conclude, when the truck size is small, the chosen algorithm does not create a massive difference. However, as the truck size increases, bubble sort, insertion sort, and selection sort becomes worse as their time complexity is $O(n^2)$. The chosen algorithm should be Quick Sort as its time complexity is O(n*logn) and we do not have almost sorted trucks.

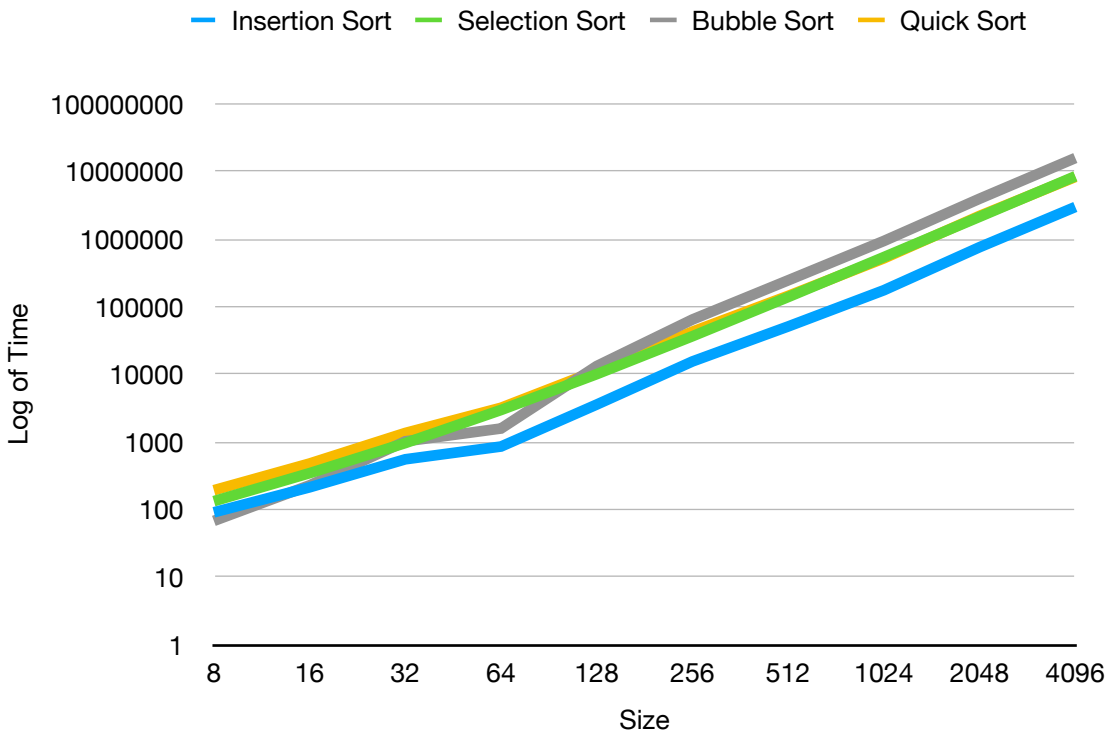
- **Task 2: An Assumption!**

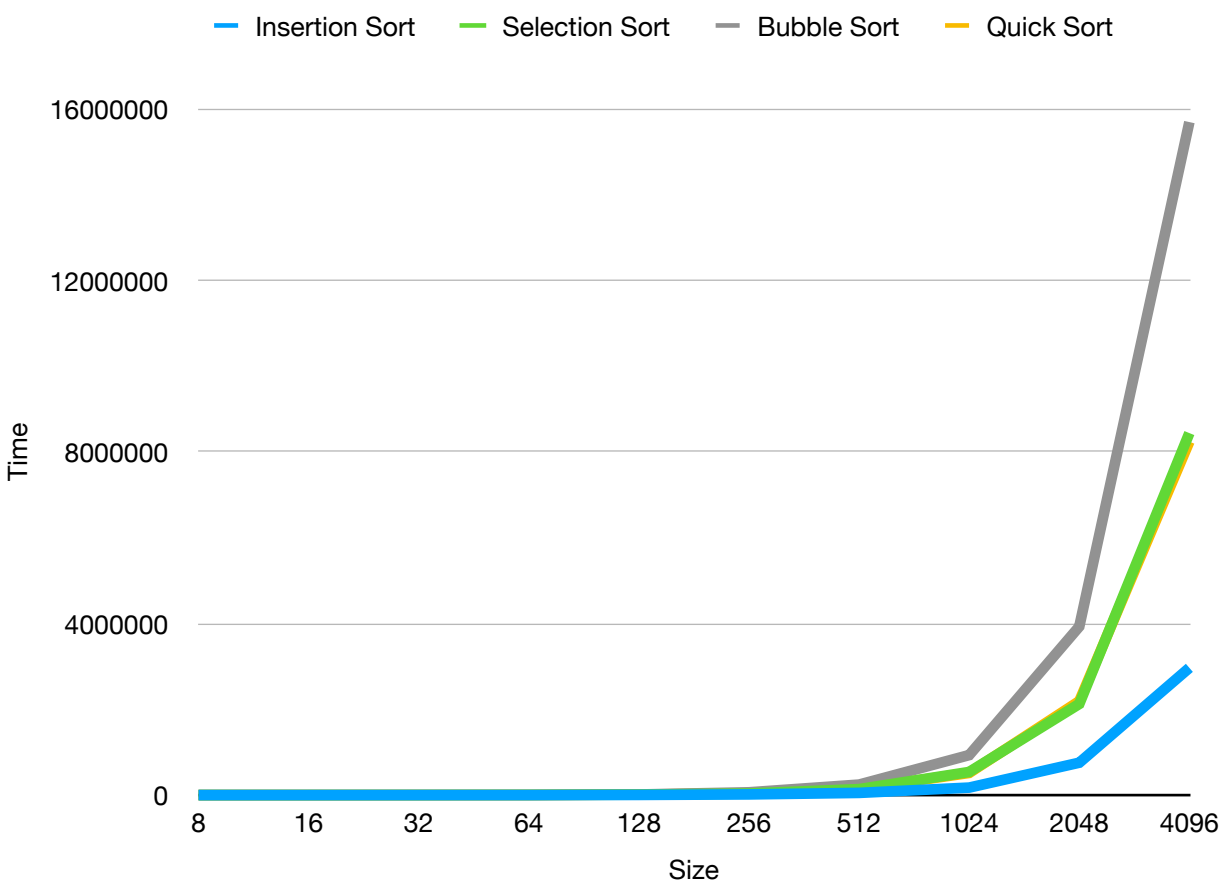
Recommended algorithm and the reason behind it: The recommended algorithm is the insertion sort. The quick sort algorithm takes $O(n^2)$ when the array is already sorted, therefore it is not logical to use it in an almost sorted array. Selection sort always takes same amount time, it does not have any advantage in here. Bubble sort would not create a difference when the array is almost sorted, even though it takes O(n) times when the array is sorted. The only option we have is the Insertion Sort. It works by iteratively selecting an element from the unsorted section of the array and inserting it into its correct position within the sorted section. This process often involves minimal operations when the array is already sorted, making insertion sort the logical choice for sorting almost sorted arrays.

Time Estimation Table:

n	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort
2^3	91	133	68	193
2^4	213	345	228	479
2^5	553	961	1019	1355
2^6	854	2961	1577	3195
2^7	3566	10033	13207	11299
2^8	15460	36465	64449	42646
2^9	50851	138481	244520	143944
2^{10}	174514	539121	933244	516939
2^{11}	755531	2126833	3932607	2101696
2^{12}	2969288	8447985	15705908	8249406

Plot of change of time according to the size:





Result Discussion:

These tables and graphs show the results of tests I did with also almost sorted arrays. An array is considered almost sorted if 6% of its items are far from their correct positions. With this new change, the most efficient is not quick sort anymore. It is known that quick sort does not perform well when the array is sorted and almost sorted. As its partitioning approach may not be optimal for almost sorted arrays, potentially resulting in unnecessary comparisons and swaps. It's time complexity becomes more closer to the $O(n^2)$ under this condition. Insertion sort becomes a lot better in the situation where we have almost sorted arrays as it iteratively takes an element from the unsorted section of the array and inserts it into its correct position in the sorted section, often with minimal operations. It makes this algorithm excel in the sorted and almost sorted arrays. As selection sort has same number of swap and comparison, its time complexity remains consistent regardless of the array's order. As it can be also seen from the graph and the table, Insertion Sort has the biggest advantage for the time when we have almost sorted arrays especially in the larger datasets.

Task 3: Increasing the number of workers

Based on the hint provided and considering the use of recursion in the algorithm, I chose to use the Quick Sort algorithm for this part of the homework. The chosen pivot for partitioning the array is the middle element. The first worker spends time for the first partitioning. Each worker is then assigned a portion of the array. Although it's not guaranteed that the array will be evenly divided, resulting in each worker having an equal workload, the overall workload for each worker tends to decrease. While it's expected that the workers won't have identical workloads, the maximum time required by any worker typically reduces compared to when only one worker is used. Also as the total time does not change drastically, it does not create an efficiency in terms of money. Therefore, Quick Sort with this approach tends to improve efficiency, particularly in terms of reducing the maximum workload for individual workers while it does not serves for the efficiency of the money, as the total time does not change much.

Here are the tables and plots of Quick Sort with One Worker and Two Worker:

Table to compare time efficiency:

One Worker	One Worker	Two Worker(1^{st})	Two Worker(2^{nd})	Two Worker(Maximum) Time Efficiency
2^3	208	146	74	151
2^4	600	374	276	399
2^5	1600	658	708	818
2^6	3656	1519	2149	2283
2^7	8993	3783	5521	5521
2^8	19199	8809	11391	11435
2^9	48073	18918	25399	25399
2^{10}	104034	39696	64417	64417
2^{11}	236434	92179	140266	140266
2^{12}	507923	193924	304521	304521

Plot to compare time efficiency

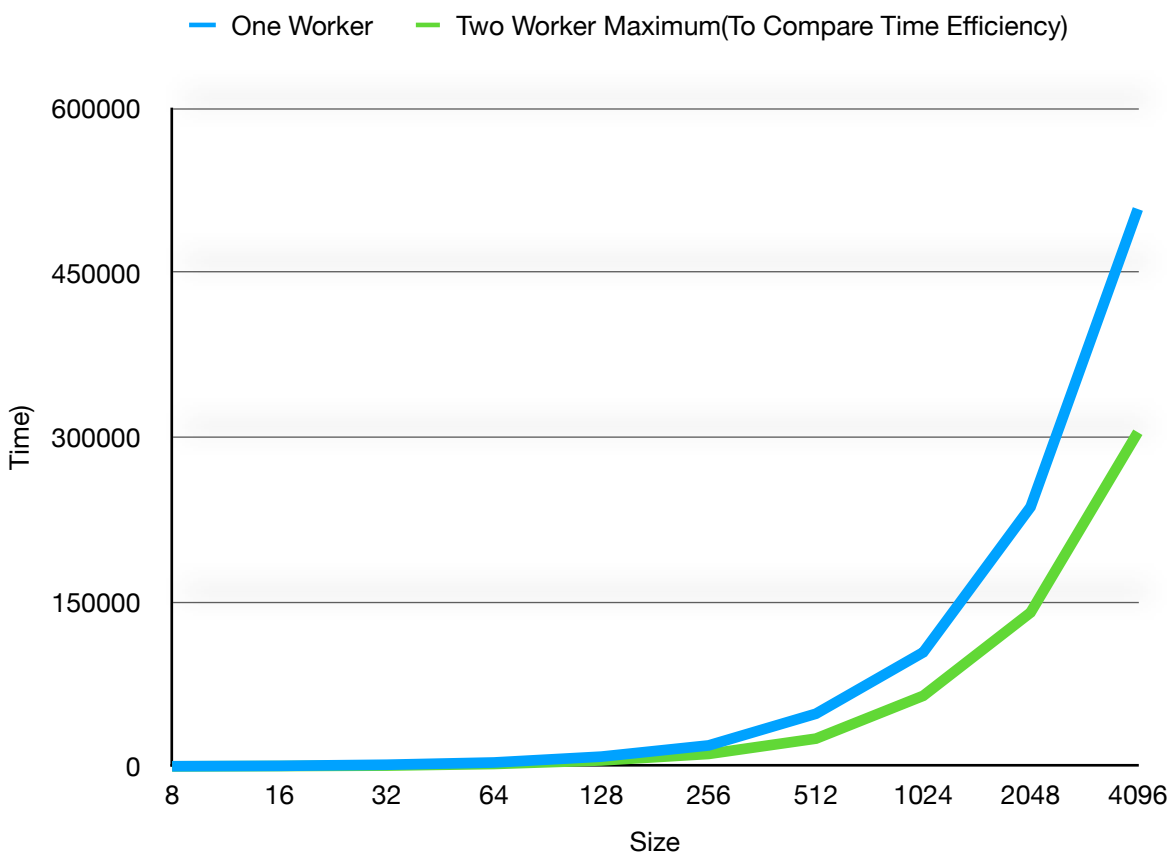
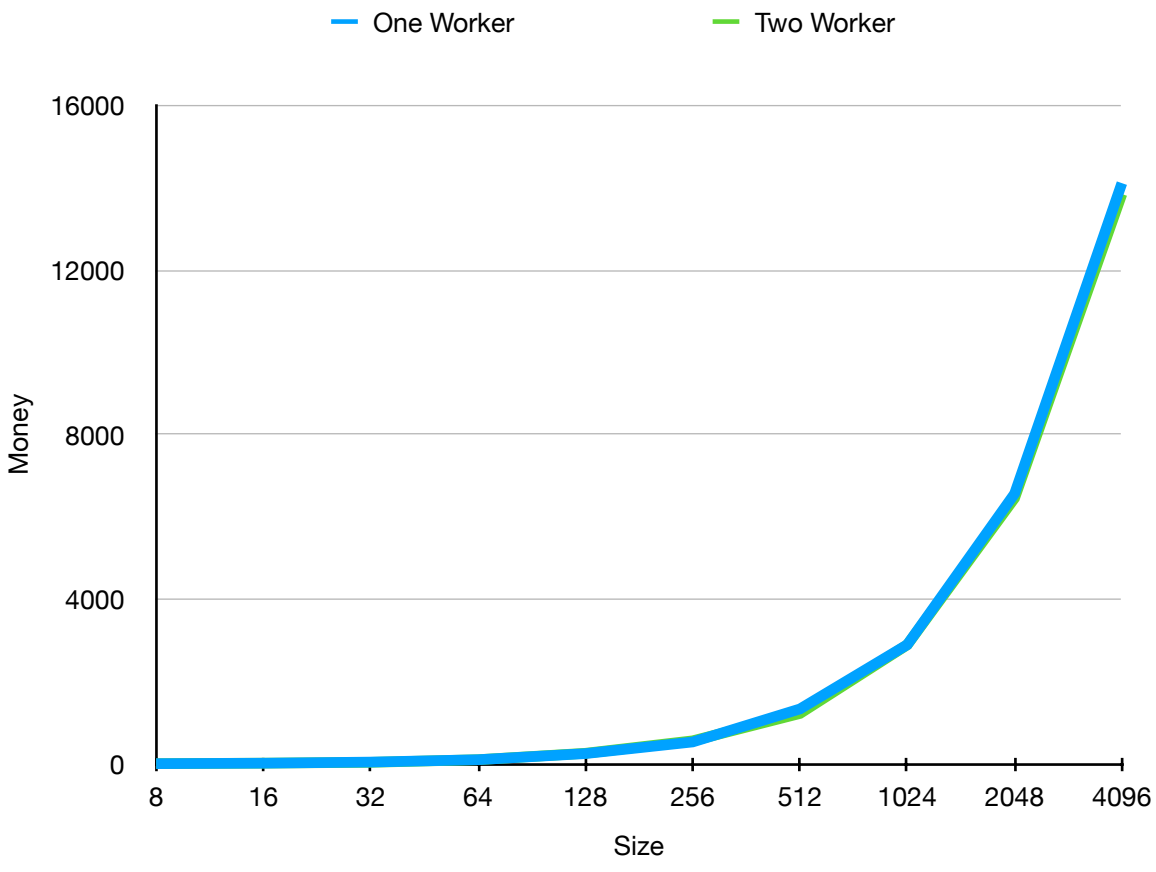


Table to compare Money Efficiency:

n	One Worker	Two Worker(1^{st})	Two Worker(2^{nd})	Two Worker(Total) Money Efficiency
2^3	6	4	2	6
2^4	17	10	8	18
2^5	44	18	20	38
2^6	102	42	60	102
2^7	250	105	153	258
2^8	533	245	316	561
2^9	1335	525	706	1225
2^{10}	2896	1102	1789	2893
2^{11}	6563	2561	3896	6457
2^{12}	14110	5381	8464	13846

Plot to compare money efficiency:



As it is assumed, when the administration employs two workers instead of one, the time required to complete the task approximately halves, while the monetary expenses remain approximately same. Therefore, if the administration wants to halve the time with approximately same money, they can employ two workers instead of one.