CS-201 Homework 2: Algorithm Efficiency and Sorting

Simay Uygur 22203328 Section 1 Date: 08.04.2024

Task 1:

In this problem, changing the tree's position requires more time than comparing a pair of trees. Therefore, the amount of money the worker will be paid increases based on the number of swapping operations in the sorting functions. The number of operations that locate trees in different places must be minimized to reduce the labor work. Taking this situation into account, the sorting algorithm should be chosen based on the minimum total duration. In the quick sort implementation, I have selected the pivot as the first element of the given array without any changes.

For selection sort, a method that finds the largest element's index compares every array component with the remaining unsorted part's elements. So, the number of comparisons won't change according to the element's values but can change according to the number of elements in the array. Bubble sort and insertion sort swaps have too many elements for this case. In my quick sort algorithm, I have chosen the pivot as the first element to see all cases.
The estimated times that each sorting will require are given in the table below.

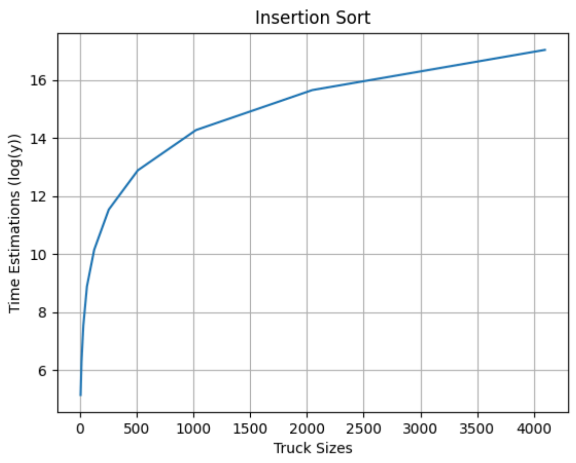
The estimated times that each sorting will require is given in the below table.

n	Insertion Sort (seconds)	Selection Sort (seconds)	Bubble Sort (seconds)	Quick Sort (seconds)
2^3	170.8	133	290.4	201.8
2^4	531.8	345	1136.2	616.6
2^5	1859.2	961	4415.2	1402
2^6	7227.2	2961	17794.4	3332.2

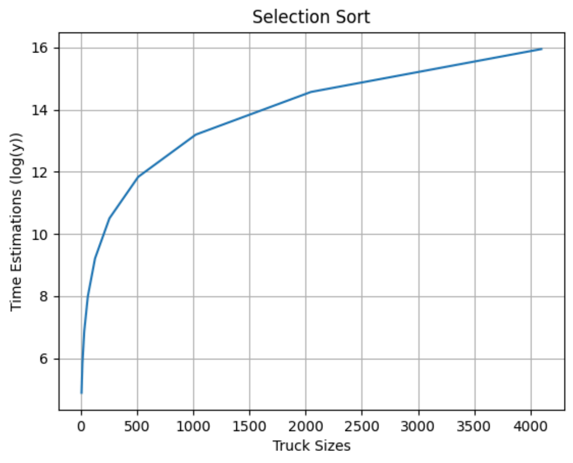
2^7	25609.6	10033	71018.2	7991.4
2^8	102263.2	36465	280957.6	17790.6
2^9	396199.8	138481	1105001.6	46125.8
2^{10}	1588010	539121	4426515.4	107037.2
2^{11}	6306166.6	2126833	17457266.4	233257
2^{12}	25223610.2	8447985	70822308.2	533730.4

According to these values, I plotted the graphs in Python using plotting libraries. The first four graphs are plotted only taking the logarithm of estimated values, but the graph containing all sortings is plotted taking the log of x and y values.

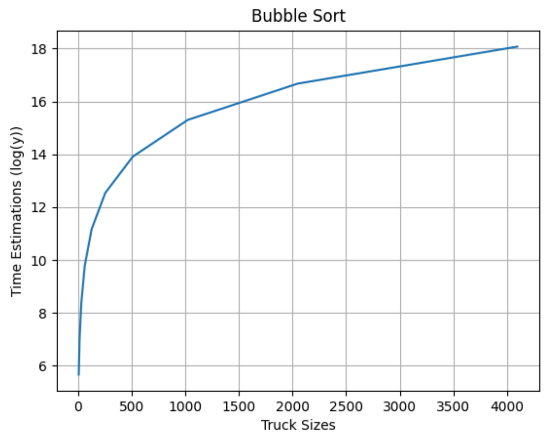
Graph1: Insertion Sort (plotted by taking only the log of y values)



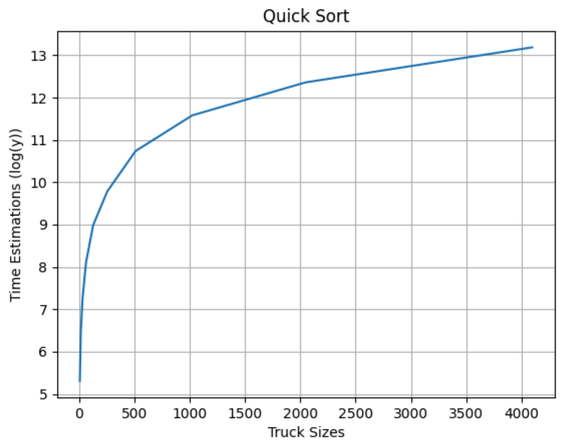
Graph 2: Selection Sort (plotted by taking only the log of y values)



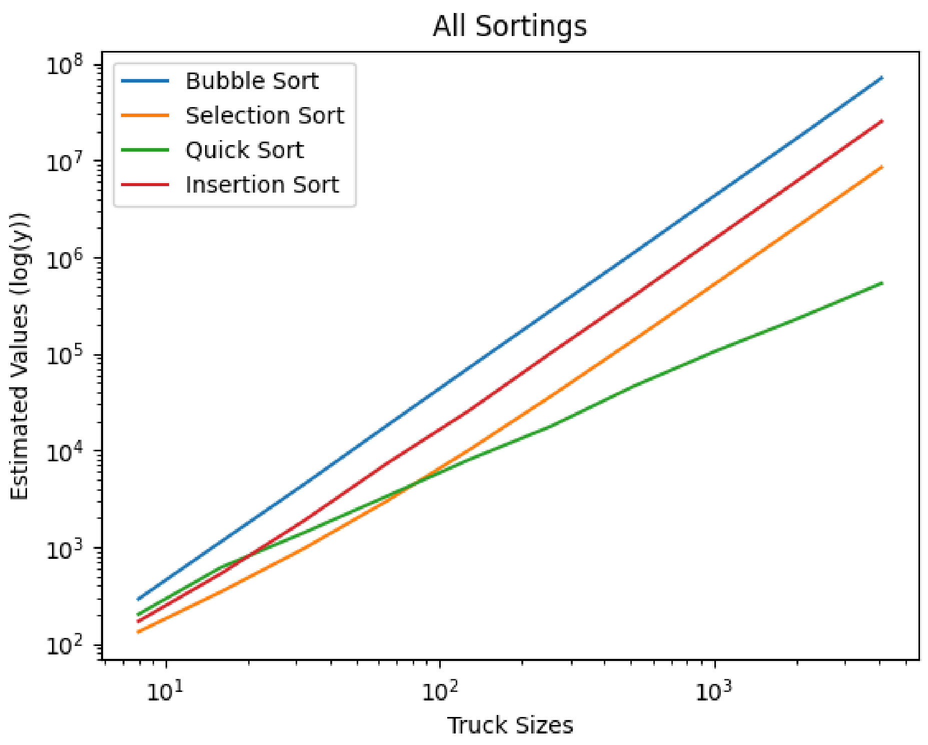
Graph 3: Bubble Sort (plotted by taking only the log of y values)



Graph 4: Quick Sort (plotted by taking only the log of y values)



Graph 5: All Sorting Algorithms with estimated values (x-axis and y-axis log(n))



Results Discussion:

When the truck sizes increase from 23 to 212, quick sort results in the least estimated value. For instance, If we analyze the biggest truck size and calculate values for that size, the quick sort algorithm is near 13, but others are near 16. In fact, Bubble Sort gives the worst estimation for the biggest size. For this reason, as the array's capacity enlarges, the quick sort algorithm's values grow slower than the others.

If the plots are compared, it can be seen that Quick Sort becomes advantageous when the truck size becomes 27 from 26 because its growth rate starts to drop because of its dividing array into two parts of logic. At each Sort, it divides the array that contains the elements smaller than the pivot located on the left of the pivot and greater than the pivot into an array. However, these subarrays are not moved to another array but kept in the same original array. In small arrays, this might seem extra work, as it can be easily seen from 23 to 26. However, as the array contains more elements, other sorting algorithms start to grow smaller because their average case time complexity is O(n2). But Quick Sort's algorithm is O(nlog(n)). Therefore, the most efficient Sort is Quick Sort for these conditions. Comparing the remaining ones, it can be observed that at the end of the graph, the insertion sort becomes more than 16 (log of time estimation), but the selection sort is near 16, and the bubble sort is near 18. Therefore, insertion sort is slightly better than selection sort, and the worst is bubble sort according to these values.

Analyzing the fifth graph, it seems that for small size arrays the orange one (selection sort) is more beneficial, however when the size gets bigger, the green line is the undermost of the graph, which means it is the best fit for this problem.

Task 2:

In my quick sort algorithm, I have chosen the pivot as the first element to see all cases. In the almost sorted array, the algorithm that will be selected must make its swap operations reasonable. For instance, this algorithm should compare the array's elements to change positions according to the need. To expand, the critical point in sorting this type of array is not sorting the whole array but changing the locations of the trees in the wrong location. Selection sort is eliminated here because it always finds the largest element and puts it in the last index, so every time it searches for the largest and puts it at the beginning of the sorted part without checking other conditions (if it is in that index, etc.). Bubble sort is also eliminated because it makes a vast amount of swap and compare operations even if most of the elements are in the correct position. For example, if one element is not in the correct position, that element should be swapped one by one with its adjacent. That is not necessary for this case. Finally, quick Sort is not a good choice for an almost sorted array if the pivot selected is the largest or smallest element. That will make the whole array sorted as if it is randomly generated.

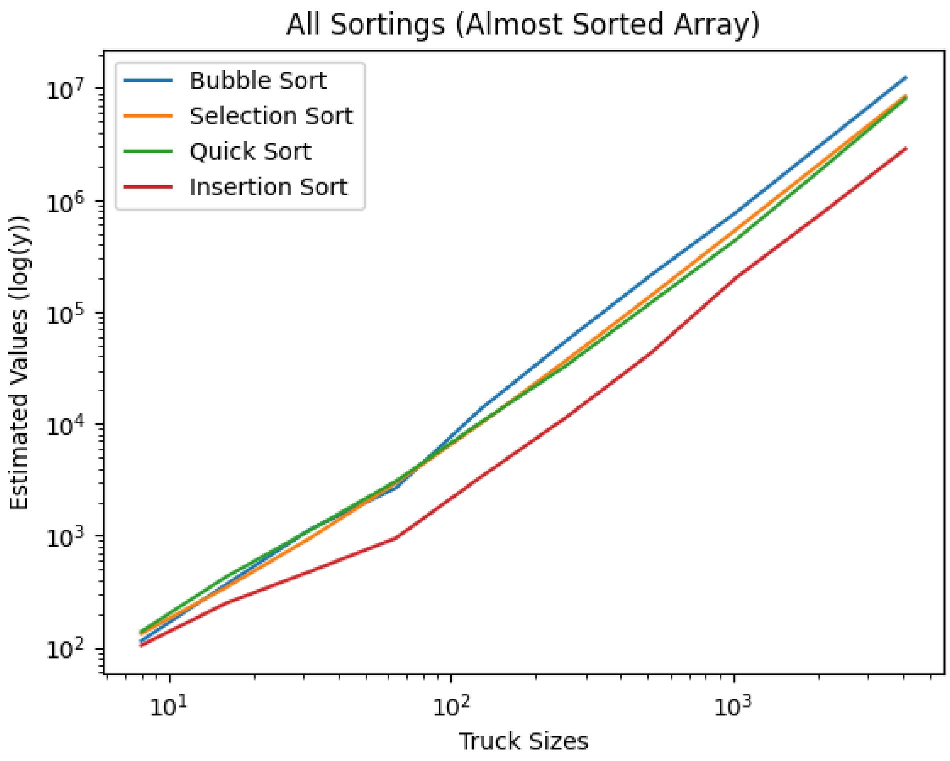
However, for almost sorted array cases, it is logical to use Insertion sort because it doesn't change if the element is smaller than the following ones, but if it is not in the correct position, this algorithm tries to open up the space for that element. Therefore, this algorithm chooses when to change its element's position more beneficially for this case. For this reason, I offer the **insertion sort algorithm.**

The table of estimated values for sorting almost sorted arrays

n	Insertion Sort (seconds)	Selection Sort (seconds)	Bubble Sort (seconds)	Quick Sort (seconds)
2^3	104.6	133	114.6	139.0

2^4	249.2	345	366.6	429.0
2^5	482.4	961	1149.0	1131.4
2^6	950.0	2961	2673.0	3067.6
2^7	3343.0	10033	13559.4	10326.0
2^8	11329.0	36465	54677.8	32860.6
2^9	42601.6	138481	211812.4	120114.8
2^{10}	199717.8	539121	767163.4	438200.6
2^{11}	743427.8	2126833	3078767.2	1849081.8
2^{12}	2851412.0	8447985	12357937.8	8027024.0

Graph 6: All Sorting Algorithms (plotted taking log of x and y values)



Results Discussion:

In this table, it can be perceived that insertion sort dominates in having the least estimated values in all sizes. Besides, bubble sort is the least efficient one according to its data. The insertion sort algorithm works better if the array is nearly sorted.
In addition, from the graph, it is observable that the sorting with the least estimated values is the insertion sort. If the remaining ones are compared, in some truck sizes, they change the domination and become the worst one. However, in the end, the worst one seems to Bubble Sort. Considering its swapping with the position after comparing until where the element should be inserted logic, Insertion Sort becomes the most advantageous in this problem.

Task 3:

In this problem, the number of workers is doubled, and an algorithm that is more suitable for this condition is expected. This new adjustment is similar to what divide and conquer algorithms do: dividing the array into two parts and laboring these subarrays, then combining these subarrays. The homework document indicates that the recursion tree is a divide-and-conquer approach. The efficiency of divide and conquer algorithms can be seen from task 1. It can be seen that quick Sort surpasses the efficiency of other algorithms because of its divide-and-conquer logic. By dividing the array into half, smaller than the pivot and greater than the pivot, every time, it divides the array according to the pivot.

In addition, enlarging the number of workers for this algorithm makes it more powerful because when it is divided by 2 (checking from the table), its estimation time gets half. This stems from the logic that workers work at the same time; therefore, the total duration decreases compared to that of one worker. Therefore, I suggest using a parallel quick sort algorithm for this problem. The elements less than the pivot (that is at the left side of the array) are assigned to one worker, and the elements greater than the pivot are assigned to the other worker. I am choosing the pivot randomly for this problem to have better pivots.

Time Estimations:

In this algorithm, the worst case is distributing the whole array to one worker, which means there is too much unfair work on one. In that case, the time complexity becomes O(n^2). For instance, if a large-sized array is mostly assigned to one worker and the worker always selects the worst pivot, that is the least or the greatest element. The (n-1) size array for one worker compares $\frac{n^2}{2} - \frac{n}{2}$ times and swaps $\frac{n^2}{2} + \frac{n}{2} - 1$ times. Therefore, the maximum duration for one worker is $1 \times (\frac{n^2}{2} - \frac{n}{2}) + 15 \times (\frac{n^2}{2} + \frac{n}{2} - 1) = 8n^2 + 7n - 15$ for the n size array (multiplying with the problem's given durations for comparing and swapping operations for each worker). For one worker, the function for the worst case is T(n)=T(n−1)+c×n, T(n) becomes O(n2), where cn represents the partitioning function complexity.

However, the best case is dividing the array into half exactly and going along with the same process (ignoring the fact that the worker has no tree to sort). For the best case, the pivot separates the array into half T(n/2)= T(n/4)+T(n/4) +c×n, for one worker. Thus, the complexity is O($\frac{n}{2} log(\frac{n}{2}$) for the best case in terms of one worker's duty period. To expand, from the recursion tree of normal Quick Sort, it can be seen that our solution partitions the array first, and its complexity is O(n), and the right and the left sides of the tree must be assigned to each worker. Therefore, one worker will have the ($\frac{n}{2}$O(1))s at the end of the recursion tree. The length of the recursion tree for each worker is log($\frac{n}{2}$). Combining these, the complexity becomes O($\frac{n}{2} log(\frac{n}{2}$)) .

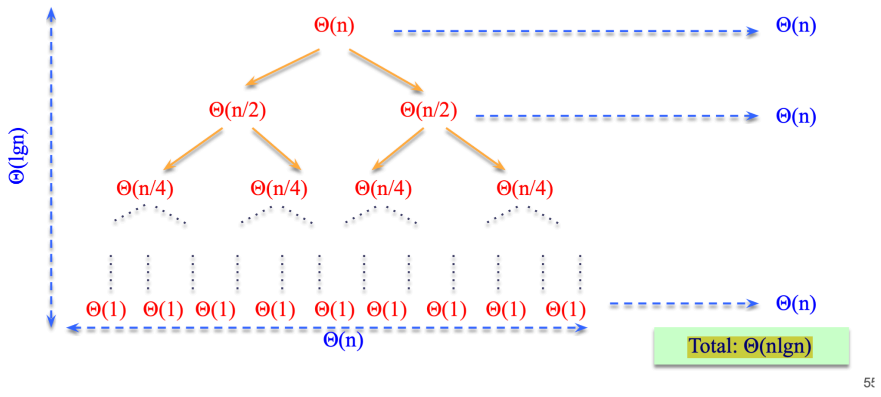


Image 1: Recursion Tree of Quick Sort (from the course slides)

Results Discussion:

The best algorithm for this problem is parallel Quick Sort, and it's logical to employ two workers considering the demand for less time. Instead of waiting for the sorting operation for the first subarray and then processing the second array, the workers work at the same time to process the subarrays in this solution. That affects the duration in a favorable way. Also, the total cost won't change because the complete amount of work didn't change in the cases of both one worker and two workers. Only the total work is distributed so that the span of the job is lessened. In the worst case, it is similar to other sorting algorithms in terms of its complexity, which is O(n^2). But the best scenario is O($\frac{n}{2} log(\frac{n}{2})$). Therefore, by improving the best algorithm from the first task, Quick Sort becomes more powerful and is the best solution for this problem.