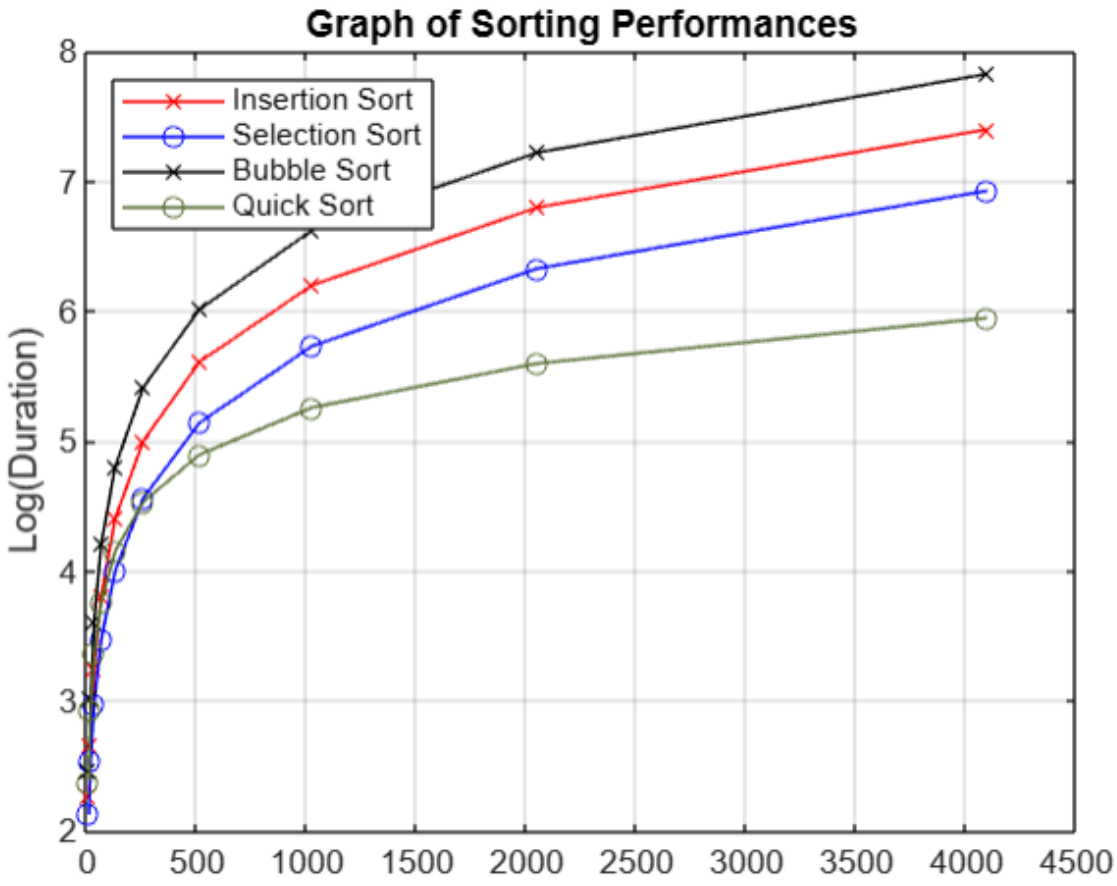
TASK-1

n	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort
2^3	184	133	291,2	237,6
2^4	456	345	1033,6	860,6
2^5	1786	961	4108,8	2294,4
2^6	6409,8	2961	16102,4	5720,4
2^7	25441,4	10033	63862,3	14264,2
2^8	100693,8	36465	259766,4	34066,4
2^9	407778	138481	1033987	78665,8
2^10	1583673	539121	4192637	181214,4
2^11	6281091	2126833	16615353,6	395525
2^12	25069587	8447985	66993369,6	885983

Average of Time Requirements for Different Sorting Algorithms with Respect to Truck Sizes



Insertion sort has outside loop that iterates through second element to the last. In each iteration, it keeps an element as a key and in inner loop, it compares prior elements with current key. Inner loop changes positions as shifting right. Inner loop will be executed whenever prior elements is greater than key or index of prior elements is greater or equal to than 0. After inner loop terminates, key will be replaced at specific index. **Discussion:** In best-case, array is already sorted and inner loop will not be executed. Only repositioning occurs when assigning key tree to current tree and placing it to specific index. Timer will be incremented 2*5 for n-1. For best case, complexity is O(n). In worst-case inner loop will be executed for 1 to n-1 times. Each execution causes 1 repositioning. Outside loop generates 2*(n-1) moves. Total moves complexity is O(n^2). Timer will be incremented by 5*O(n^2). In inner loop, pair of trees are compared that timer will be incremented by 1. Insertion sort performs efficient for low sized trucks. However, since average case is O(n^2), **Insertion sort** is not efficient for large sized trucks.

Selection sort has outside loop that controls the position of imaginary wall between sorted and unsorted array parts. Inner loop determines the maximum tree height of unsorted part and swaps that tree with last element of unsorted

part. After swapping, index of wall decrements and unsorted size lessens by 1 whereas sorted size increments by 1. Progress continues until array size = sorted part size. **Discussion:** Outside loop executes for n-1 times. Inner loop executes for 1,2,3, … ,n-1 times. Inner loop determines the max by comparing pair of trees that increments timer by O(n^2)*1. Swap occur in outside loop that increments timer by 5*O(n). For all cases, complexity is O(n^2). Since **Selection sort** requires O(n) moves, it is good choice if moving trees is costly but comparing trees are not costly since comparison requires O(n^2). Allover, not preferrable for large sized truck to sort trees.

Bubble sort has outside loop that executes n-1 for worst case, or less for almost sorted array. Inner loop compares current indexed element with next elements. Whenever current indexed tree height is greater than the subsequent, they are swapped until index reaches length-(number of successful replaced heights). **Discussion:** If array is already sorted, outside loop executed for once, inner loop executes for n-1. Total complexity will be O(n). For worst-case, array is in reverse order. Outer loop executes n-1. Inner loop executes n-1-forwarded index that increments timer by 1 and 5 for swapping. Since complexity is O(n^2), **Bubble sort** is not suitable for large sized trucks.

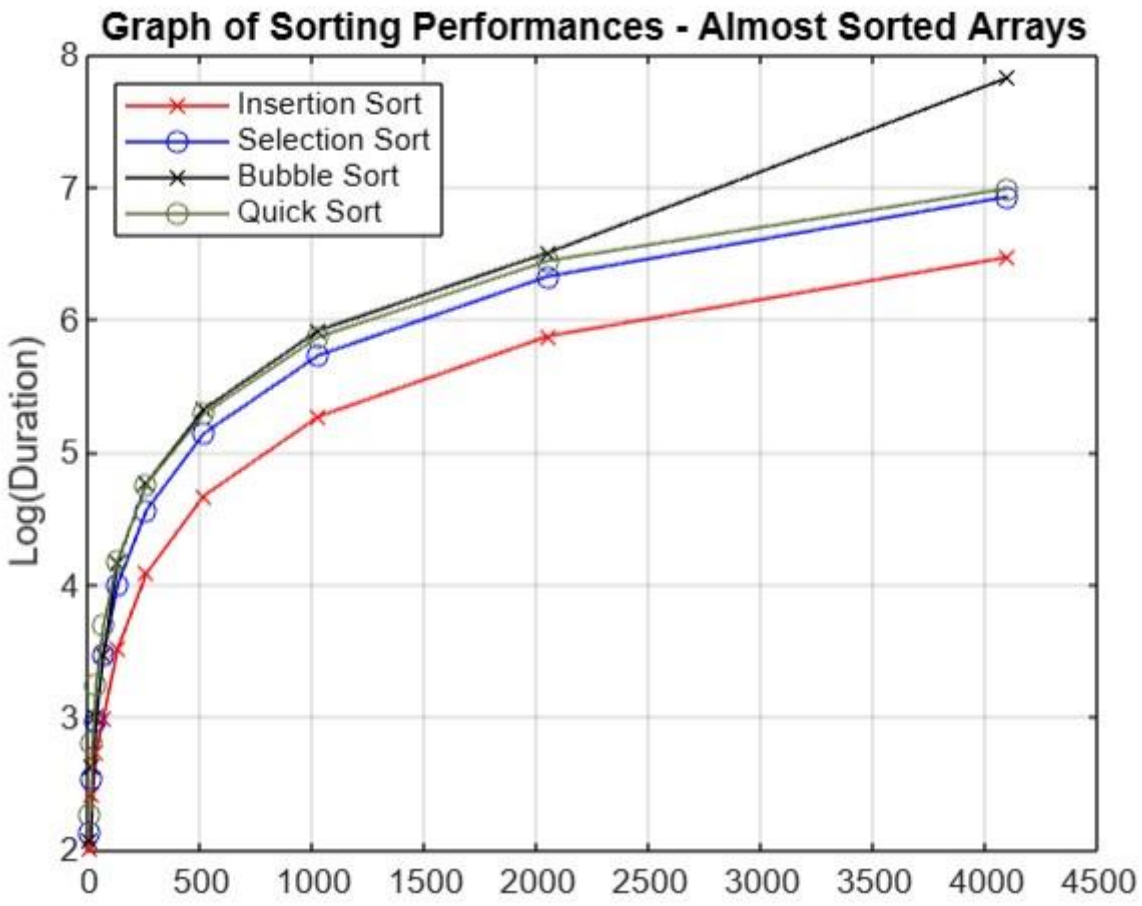
Quick sort requires a pivot. During iterating through unknown part, if elements are less than pivot, they swapped from their index to last index of S1(imaginary part of array that contains elements less than pivot). If element is greater than pivot, nothing happens, stored in S2. After partitioning, pivot replaced between S1 and S2. This progress occurs recursively for S1 and S2 with the base case that is first < last. **Discussion: Quick Sort** time complexity mainly depends on pivot chose and status of array. If pivot is about to median value, and array is not sorted, recursion tree will be balanced for both S1 and S2 and T(n) = 2*T(n/2) + O(n). Each partition has one swap outside of the loop and inside of the loop, swap and height comparison occurs that increments timer by 15 and 1 respectively, through first to last. For small size trucks, quick sort is a bit slower since recursive calls adding up timer compared to selection and insertion sort but for large size trucks, it is useful at all. Overall complexity for average case is O(n*log(n)). **Quick Sort is good fit for this problem as overall complexity is O(n*log(n)).**

TASK-2

If an array is almost sorted, my recommendation to sort the rest of the array will be **Insertion Sort**. Because, inner loop will be executed less if most of the prior elements are less than the current indexed element. Number of moves will be 2*(execution of outside loop = n-1) which means that timer will be incremented by 2*5*O(n). Inner loop execution will be significantly reduced since %94 of array is already sorted. Overall complexity yields O(n) with slightly higher.

n	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort
2^3	101	133	117,2	187,6
2^4	259,8	345	421	650,4
2^5	551	961	999	1772
2^6	994,2	2961	2952	4940,6
2^7	3297,8	10033	14903,8	15442,6
2^8	12149,4	36465	58492,6	57403
2^9	46829	138481	210883,4	195922,8
2^10	185857,8	539121	831022,4	741687,6
2^11	751761,8	2126833	3206782,6	2785281,4
2^12	2958024,6	8447985	66993369,6	9752839,2

*Sorting Algorithms Performance for **Almost Sorted** Arrays*



*Average of 5 Trials of Different Sorting Algorithms Performing on **Almost Sorted Arrays***

Insertion sort performs advantageously compared to others. It requires few comparisons and swaps when %94 of array elements in correct position.

Selection sort does not take advantage of the initial state of elements. Therefore, its duration values with respect to size are same with the previous chart values.

Bubble sort performed both well and bad inconsistently. That is, if unsorted elements are at the beginning, and after swapping them to correct locations, outer loop is early terminated since sorted signal exchanged “True”. However, in average case, bubble sort required large duration proportional to O(n^2). I noted that, bubble sort performs better at nearly sorted arrays compared to non-sorted arrays. But overall duration is the highest among other sorting algorithms.

Quick sort performed not very well for almost sorted arrays. This is because of (I chose pivot as first element of an array), pivot chosen is less than most of array items. I noted that if arrays is in reverse order and pivot is first element, in that

case in partition function if statement executed, adding +15 to timer by swap. In the case array is in ascending order, this causes dividing array into a full or nearly full part and empty or nearly empty part, making T(n) ~ T(n-1) + T(0) + O(n). As n increases, **Quick sort** yields O(n^2), performing worst case.

TASK-3

I suggest **Quick Sort** for this problem. I added 2 different timers measuring each worker's working duration. Then I will compare the total cost(TL) of 2 worker leveraged work vs best duration algorithm's cost, that is quick sort. For choosing pivot as middle element of an unsorted array, these are the empirical results:

n	Quick Sort with 2 workers	Quick Sort with 1 worker
2^3	T1 = 49 Cost = 4 T2 = 95	T = 237,6 Cost = 6,6
2^4	T1 = 181 Cost = 11,3 T2 = 226	T = 860,6 Cost = 23,9
2^5	T1 = 496 Cost = 30,8 T2 = 616	T = 2294,4 Cost = 63,7
2^6	T1 = 1369 Cost = 79,2 T2 = 1483	T = 5720,4 Cost = 158,9
2^7	T1 = 2748 Cost = 165,3 T2 = 3205	T = 14264,2 Cost = 396,2
2^8	T1 = 7679 Cost = 434 T2 = 7947	T = 34066,4 Cost = 946,2
2^9	T1=18274 Cost= 1000 T2 = 17727	T = 78665,8 Cost = 2185,1
2^10	T1=45906 Cost =2939,4 T2 = 59913	T=181214,4 Cost = 5033,7
2^11	T1 = 107025 Cost=6253,4 T2 = 118100	T = 395525 Cost = 10986,8
2^12	T1=239904 Cost= 13270,5 T2 = 237837	T = 885983 Cost = 24610,6

With 2 workers, I assured that each worker worked on smaller parts of the array from the beginning resulted in decrease in total time and cost. I note that pivot choose is critical since if the pivot is max value, then one worker will do the all job on his/her own but another does nothing. Time complexity will be T(n) = T(n-1) + O(n). It will be O(n^2)

Allover, in average, complexity is O(n*logn). The reason why I did not prefer merge sort is that merge sort requires same sized temporary array, and swapping is essential in that case. In quick sort, I can avoid choosing pivot as minimum or maximum value with O(n) operation. Also, 2 workers can finish the job earlier and will less cost without requiring additional swaps, dividing workload from the beginning.