Alec Kain

Dr. Shah, Dr. Cost, Dr. Almes

EN.605.202.81.FA23

December 4, 2023

Lab 4 Analysis: Comparing Quick Sort and Merge Sort

Description of Sorting Algorithms:

Quick Sort and Merge Sort, two divide-and-conquer sorting algorithms, implemented recursively, showcase distinct strategies in sorting arrays.

A Quick Sort typically works by first partitioning a collection of values in a list-like data structure (in my implementation, I used an array for simplicity) into multiple, slightly more sorted collections of data, and then recursively partition each smaller partition until each partition is sorted, and therefore the original data structure has been sorted. The partition method essentially swaps values around the pivot: to the left of the pivot, if a value larger than the pivot is encountered, and to the right of the pivot, if a value less than the pivot is encountered, the two swap until the pivot is reached or the left-to-right traversal index becomes greater than or equal to the right-to-left traversal index, and a new low and high index are parametrized for the next recursive partition call; the next pivots are determined either methodically or arbitrarily by the sort user.

A Merge Sort works by dividing a list-like data structure (an array in my implementation) into smaller sizes until each sub-list structure contains only one element (base case using recursion), thus having a length of 1. Then, these unitary sub-list are merged with each other, with the smallest sub-list value being placed at the front of the newly formed merged list, recursively merging multiple merged lists and placing the smallest value among these to-be merged lists at the front in the newly merged list. The Natural Merge Sort accommodates dynamically larger inputs through a linked implementation in which pre-existing sorted portions are identified in the list and the algorithm merges these together, altogether circumventing the need for comparisons and swaps, before performing the traditional Merge Sort on unsorted elements, rather than dividing each value of the list initially as the regular Merge Sort does.

Justification of Implementation Approach:

The recursive implementation of both sorting algorithms aligns with the lab's instructions, ensuring consistent performance comparisons and efficient sorting of dynamically increasing input sizes. Recursive strategies uphold fairness in assessing efficiency across both Quick Sort and Merge Sort, neutralizing disparities between algorithm executions and their associated operations such as swaps, comparisons, and memory/space overhead. A Natural Merge Sort was implemented to effectively handle dynamically larger datasets such as those with 10,000 values, as opposed to the regular Merge Sort, in which the former typically expedites the Merge Sort procedure for the rationale mentioned in the prior section.

Appropriateness to the Application:

Quick Sort’s adaptability to different partition sizes and pivot selection, and Merge Sort's adeptness in segmenting arrays of dynamical, increasingly larger sizes make them excellent choices for handling diverse dataset volumes.

Design Decisions and Justification:

The multiple variations of Quick Sort based on pivot selection and partition sizes facilitate a multifaceted performance evaluation. Designing different variations of the Quick Sort experimentally demonstrates the merits and pitfalls of each sort, depending on arbitrarily chosen partition sizes and pivots. Merge Sort's simplicity in segmenting arrays for sorting provides a stable baseline for comparative analysis. The decision to implement each algorithm recursively facilitates a consistent basis for comparison between each sort in terms of operations such as swaps, comparisons, and overall efficiency and efficacy. I chose this paradigm over the iterative one due to the nature of the problem these sorting methods are employed to solve. It is natural to use a method in which one can use the same algorithm to solve smaller and smaller bits of the problem which results in the problem solved in the simplest way, rather than explicitly calling functions on smaller and smaller bits of the problem and focusing on minutiae. In addition, the fact that the algorithms' iterative counterparts must use the stack explicitly which adds overhead, whereas the overhead memory added in the recursive style is implicit.

Efficiency, Comparison, and Performance Analysis:

Detailed comparisons of Quick Sort and Merge Sort encompassing comparisons, swaps, and time complexities reveal insights into their comparative and individual efficiencies. Observations consider data orders, sizes, pivot selections, and partition sizes, capturing relative efficiencies and trade-offs. In general, the efficiency/runtime of a Quick Sort tends to be in the best-case O(nlogn) for smaller input sizes, largely ranging between O(nlogn) and O(n^2) depending on pivot position choice, and approaches an O(n^2) runtime for much larger input sizes. It is favorable to use this approach when sorting data in this case over the Merge Sort. In contrast, the efficiency/runtime of Merge Sort is always O(nlogn), regardless of input size. For the Natural Merge Sort, however, the best-case runtime is O(n) when the data is already sorted. The Quick Sort performs best on randomized data, worse on reverse-ordered data, and worst on sorted data. The Merge Sort and Natural Merge Sort perform best on ordered data, slightly worse on reverse-ordered data, and worst on random-ordered data. The Merge Sort and its Natural variant are more efficient for very large input sizes than Quick Sort, yet less efficient for smaller and moderate input sizes. The Merge/Natural Merge Sort has a space complexity of O(n), whereas the QuickSort, in its recursive form, has a space complexity of O(logn) making it more space-efficient and less overhead-intensive in regards to memory than Merge Sort and its variants.

However, in this lab, runtimes and efficiencies vary depending on data order, sizes, pivot selections, and partition sizes, as well as the variation of the algorithm implemented. My summarized findings are below:

QuickSort (First Pivot)

Comparisons and Swaps:

Ordered: Showed a consistent increase in comparisons and swaps with larger arrays.

Reverse Ordered: Displayed similar trends as ordered arrays.

Random Ordered: Demonstrated variable counts of comparisons and swaps.

QuickSort (Insertion Sort for Small Partitions)

Comparisons and Swaps:

Ordered, Reverse Ordered, and Random Ordered: Maintained relatively lower counts compared to QuickSort with the first pivot. The number of comparisons and swaps increased gradually with larger arrays, but the counts were consistently lower than the first pivot QuickSort.

QuickSort (Median-of-Three Pivot)

Comparisons and Swaps:

Showed reduced comparisons compared to the first pivot QuickSort in all three types of arrays. Swaps varied but generally followed a similar pattern to comparisons. Showed better performance in terms of comparisons compared to the first pivot QuickSort and insertion sort for small partitions.

Natural MergeSort

Comparisons and Swaps:

Exhibited lower counts of comparisons and lower counts of swaps reported among most of the instances across array types and sizes compared to QuickSort variations. Higher counts of swaps were present in larger reverse-ordered arrays and random-ordered arrays; however, comparison counts were lower.

Efficiency Observations:

QuickSort with First Pivot: Shows higher comparisons and swaps, becoming more pronounced with larger arrays and varying orders.

QuickSort with Insertion Sort for Small Partitions: Offers improved performance compared to the first pivot QuickSort, showcasing lower comparisons and swaps for the same array orders and sizes.

QuickSort with Median-of-Three: Demonstrates better efficiency in comparison count compared to the first pivot QuickSort. Swaps also tend to be lower, especially for larger arrays.

Natural MergeSort: Consistently displays lower counts of comparisons swaps across most array types and sizes, indicating better efficiency compared to QuickSort variations.

Overall Findings:

QuickSort: Shows sensitivity to the order of elements, performing differently for ordered, reversed ordered, and randomized ordered arrays. Efficiency depends on the pivot selection and handling of small partitions.

Natural MergeSort: Shows consistent performance across different array orders and sizes, with notably lower comparisons and swaps, demonstrating a more stable and efficient behavior for many sizes and initial orderings of elements.

Key Learnings:

The analysis highlights the impact of pivot selection, partition sizes, and recursive strategies on sorting performance. It underscores the importance of considering different factors influencing sorting efficiency. The findings indicate that while QuickSort variations are efficient, they demonstrate variability in performance based on the order of elements and pivot selection, whereas Natural MergeSort maintains a more consistent efficiency across different scenarios.

Effectiveness of Sorting Algorithms:

In evaluating the sorting algorithms, variations in Quick Sort exhibit distinct efficiency trends influenced by pivot selection, particularly noticeable in different partition sizes. Merge Sort consistently handles various data orders and sizes, showcasing steady performance.

Enhancements:

Current enhancements include thorough error handling and checking, incrementing counts and swaps at each instance of the respective operation and the output of these values at the end of each run, and recursive implementation of each sorting method as opposed to their less-efficient, iterative variation. Potential future enhancements include investigating alternative pivot selection strategies, partitioning techniques, or hybrid sorting algorithms to optimize efficiency further.

Project Alignment with Requirements:

The project demonstrates successful implementation, error handling, and analysis of Quick Sort and Merge Sort performances across varied datasets. The analysis adheres to the outlined requirements, showcasing diverse scenarios and comprehensive evaluations.

Conclusion and Future Considerations:

The lab effectively explores Quick Sort and Merge Sort efficiency in diverse scenarios with alternative implementations for each algorithm. Future considerations may involve further optimizations and exploring additional sorting strategies to improve overall efficiency. Further considerations include enhancements mentioned in the “Enhancements” section.

TABLE OF FINDINGS:

*ord = ordered, rev_ord = reverse-ordered, rand_ord = random-ordered

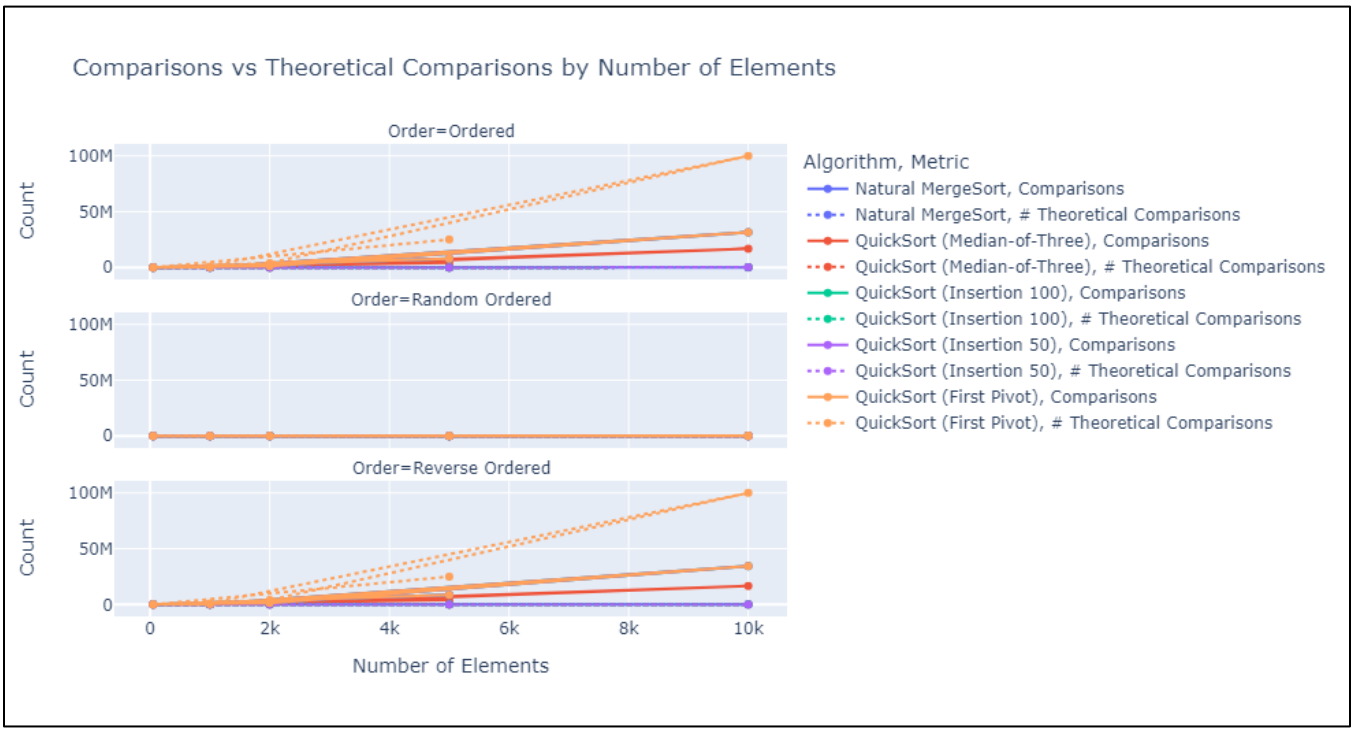
*# = number of array elements

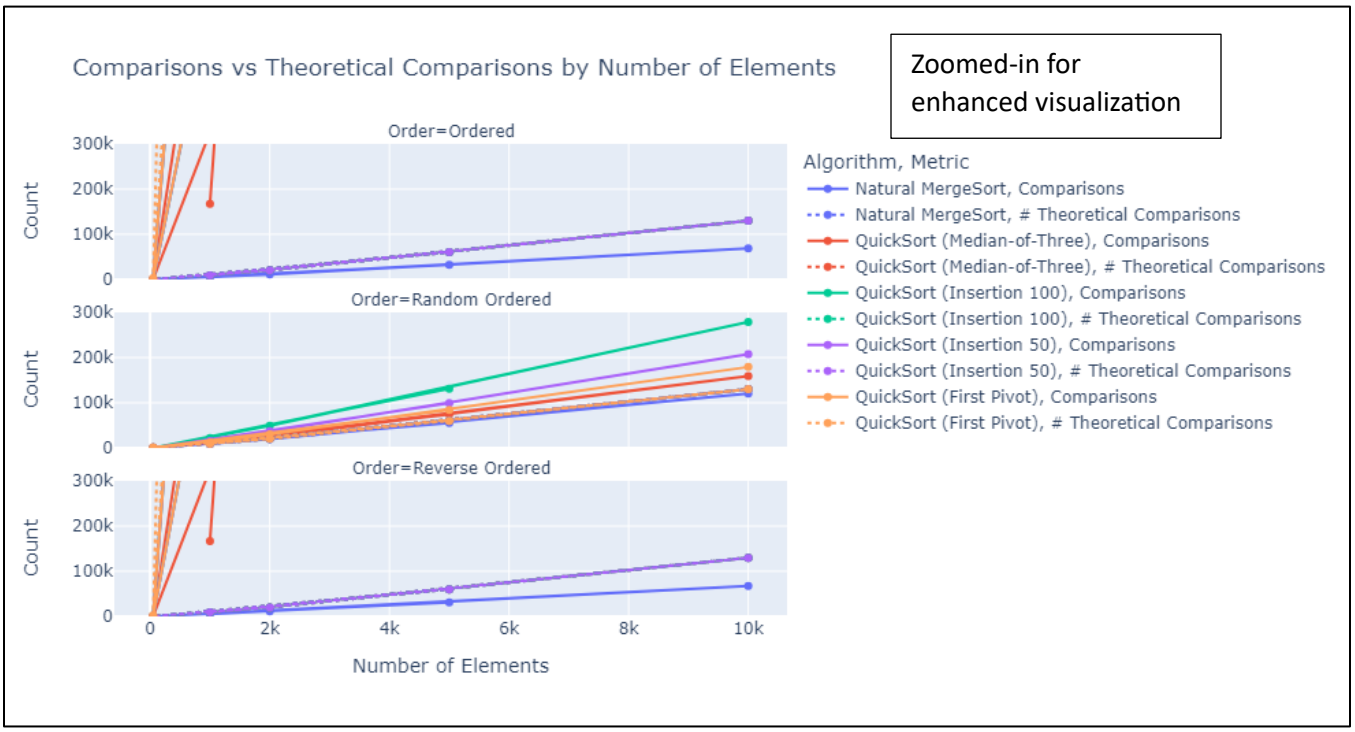
Array Type	Algorithm	Comparisons	Swaps	Theoretical Comparisons	Theoretical Swaps	# Theoretical Comparisons	# Theoretical Swaps
ord_1000	Natural MergeSort	5044	0	O(n log n)	O(log n)	9000	9
ord_1000	QuickSort (Median-of-Three)	167202	991	O(n log n)	O(log n)	9000	9
ord_1000	QuickSort (Insertion 100)	313868	571	O(n log n)	O(log n)	9000	9
ord_1000	QuickSort (Insertion 50)	316674	608	O(n log n)	O(log n)	9000	9
ord_1000	QuickSort (First Pivot)	317420	725	O(n^2)	O(n)	1000000	1000
ord_10000	Natural MergeSort	69008	0	O(n log n)	O(log n)	130000	13
ord_10000	QuickSort (Median-of-Three)	16680187	9987	O(n log n)	O(log n)	130000	13
ord_10000	QuickSort (Insertion 100)	31536510	6245	O(n log n)	O(log n)	130000	13
ord_10000	QuickSort (Insertion 50)	31538765	6276	O(n log n)	O(log n)	130000	13
ord_10000	QuickSort (First Pivot)	31540889	7365	O(n^2)	O(n)	100000000	10000
ord_2000	Natural MergeSort	11088	0	O(n log n)	O(log n)	20000	10
ord_2000	QuickSort (Median-of-Three)	667904	1990	O(n log n)	O(log n)	20000	10
ord_2000	QuickSort (Insertion 100)	1248940	1208	O(n log n)	O(log n)	20000	10
ord_2000	QuickSort (Insertion 50)	1251630	1244	O(n log n)	O(log n)	20000	10

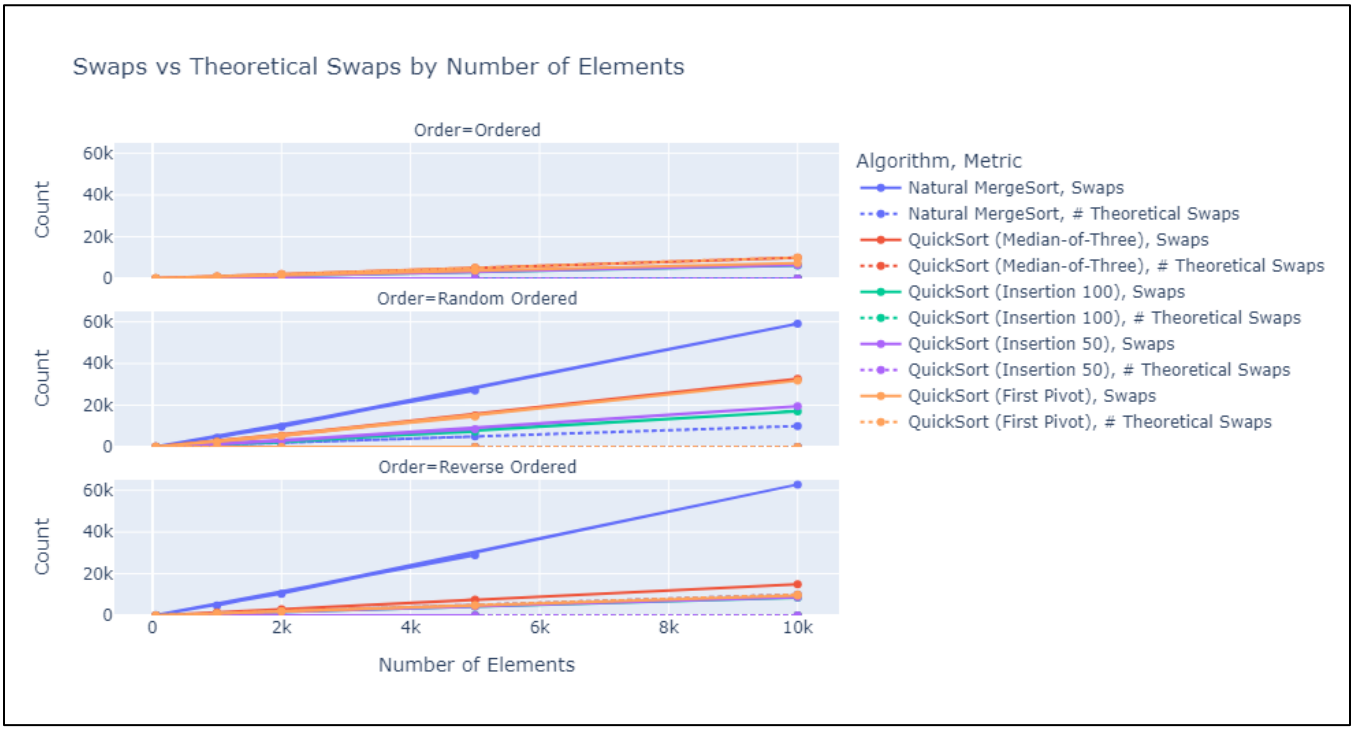
ord_2000	QuickSort (First Pivot)	1252641	1471	O(n^2)	O(n)	4000000	2000
ord_50	QuickSort (Insertion 100)	0	0	O(n log n)	O(log n)	250	5
ord_50	QuickSort (Insertion 50)	0	0	O(n log n)	O(log n)	250	5
ord_50	Natural MergeSort	153	0	O(n log n)	O(log n)	250	5
ord_50	QuickSort (Median-of-Three)	444	44	O(n log n)	O(log n)	250	5
ord_50	QuickSort (First Pivot)	1225	49	O(n^2)	O(n)	2500	50
ord_5000	Natural MergeSort	32004	0	O(n log n)	O(log n)	60000	12
ord_5000	QuickSort (Median-of-Three)	4169392	4989	O(n log n)	O(log n)	60000	12
ord_5000	QuickSort (Insertion 100)	7905752	3102	O(n log n)	O(log n)	60000	12
ord_5000	QuickSort (Insertion 50)	7907763	3129	O(n log n)	O(log n)	60000	12
ord_5000	QuickSort (First Pivot)	7909146	3663	O(n^2)	O(n)	25000000	5000
rand_ord_1000	Natural MergeSort	8717	4325	O(n log n)	O(n)	9000	1000
rand_ord_1000	QuickSort (Median-of-Three)	11470	2424	O(n log n)	O(log n)	9000	9
rand_ord_1000	QuickSort (First Pivot)	11978	2377	O(n log n)	O(log n)	9000	9
rand_ord_1000	QuickSort (Insertion 50)	14964	1172	O(n log n)	O(log n)	9000	9
rand_ord_1000	QuickSort (Insertion 100)	22159	917	O(n log n)	O(log n)	9000	9
rand_ord_10000	Natural MergeSort	120439	59160	O(n log n)	O(n)	130000	10000
rand_ord_10000	QuickSort (Median-of-Three)	158986	32660	O(n log n)	O(log n)	130000	13
rand_ord_10000	QuickSort (First Pivot)	179203	31765	O(n log n)	O(log n)	130000	13
rand_ord_10000	QuickSort (Insertion 50)	207758	19460	O(n log n)	O(log n)	130000	13
rand_ord_10000	QuickSort (Insertion 100)	278803	17008	O(n log n)	O(log n)	130000	13
rand_ord_2000	Natural MergeSort	19364	9635	O(n log n)	O(n)	20000	2000
rand_ord_2000	QuickSort (Median-of-Three)	24800	5463	O(n log n)	O(log n)	20000	10
rand_ord_2000	QuickSort (First Pivot)	30305	5172	O(n log n)	O(log n)	20000	10
rand_ord_2000	QuickSort (Insertion 50)	36650	2701	O(n log n)	O(log n)	20000	10
rand_ord_2000	QuickSort (Insertion 100)	48780	2317	O(n log n)	O(log n)	20000	10
rand_ord_50	QuickSort (Insertion 100)	0	0	O(n log n)	O(log n)	250	5
rand_ord_50	QuickSort (Insertion 50)	0	0	O(n log n)	O(log n)	250	5
rand_ord_50	Natural MergeSort	225	105	O(n log n)	O(n)	250	50
rand_ord_50	QuickSort (Median-of-Three)	277	74	O(n log n)	O(log n)	250	5
rand_ord_50	QuickSort (First Pivot)	321	67	O(n log n)	O(log n)	250	5
rand_ord_5000	Natural MergeSort	55217	27114	O(n log n)	O(n)	60000	5000
rand_ord_5000	QuickSort (Median-of-Three)	74041	15095	O(n log n)	O(log n)	60000	12
rand_ord_5000	QuickSort (First Pivot)	84157	14644	O(n log n)	O(log n)	60000	12
rand_ord_5000	QuickSort (Insertion 50)	98817	8538	O(n log n)	O(log n)	60000	12

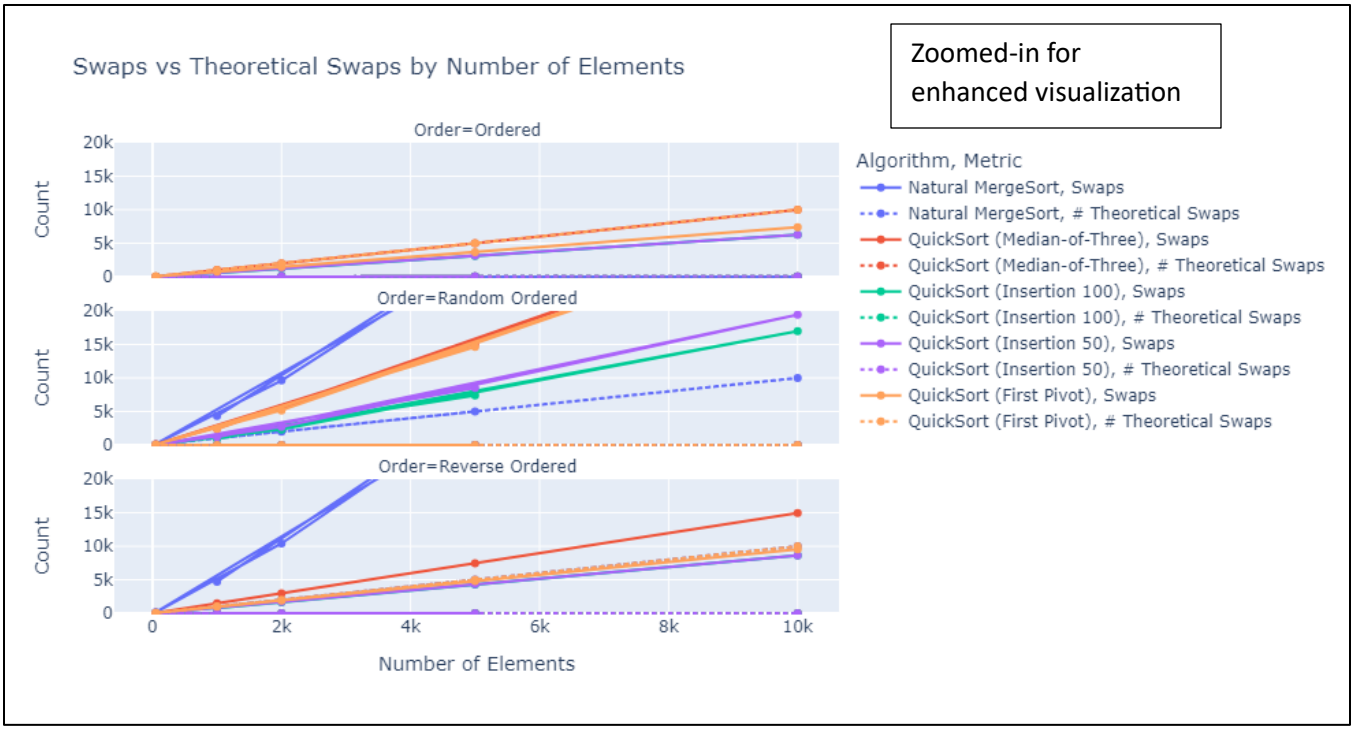
rand_ord_5000	QuickSort (Insertion 100)	131071	7377	O(n log n)	O(log n)	60000	12
rev_ord_1000	Natural MergeSort	5146	4721	O(n log n)	O(n)	9000	1000
rev_ord_1000	QuickSort (Median-of-Three)	166772	1487	O(n log n)	O(log n)	9000	9
rev_ord_1000	QuickSort (First Pivot)	336429	968	O(n^2)	O(n)	1000000	1000
rev_ord_1000	QuickSort (Insertion 50)	336482	818	O(n log n)	O(log n)	9000	9
rev_ord_1000	QuickSort (Insertion 100)	336970	774	O(n log n)	O(log n)	9000	9
rev_ord_10000	Natural MergeSort	67234	62836	O(n log n)	O(n)	130000	10000
rev_ord_10000	QuickSort (Median-of-Three)	16679127	14985	O(n log n)	O(log n)	130000	13
rev_ord_10000	QuickSort (Insertion 50)	34416420	8655	O(n log n)	O(log n)	130000	13
rev_ord_10000	QuickSort (First Pivot)	34417179	9538	O(n^2)	O(n)	100000000	10000
rev_ord_10000	QuickSort (Insertion 100)	34417752	8614	O(n log n)	O(log n)	130000	13
rev_ord_2000	Natural MergeSort	11305	10427	O(n log n)	O(n)	20000	2000
rev_ord_2000	QuickSort (Median-of-Three)	668093	2986	O(n log n)	O(log n)	20000	10
rev_ord_2000	QuickSort (Insertion 50)	1350414	1680	O(n log n)	O(log n)	20000	10
rev_ord_2000	QuickSort (First Pivot)	1350440	1916	O(n^2)	O(n)	4000000	2000
rev_ord_2000	QuickSort (Insertion 100)	1352293	1641	O(n log n)	O(log n)	20000	10
rev_ord_50	Natural MergeSort	133	133	O(n log n)	O(n)	250	50
rev_ord_50	QuickSort (Median-of-Three)	454	67	O(n log n)	O(log n)	250	5
rev_ord_50	QuickSort (First Pivot)	1225	49	O(n^2)	O(n)	2500	50
rev_ord_50	QuickSort (Insertion 100)	1225	0	O(n log n)	O(log n)	250	5
rev_ord_50	QuickSort (Insertion 50)	1225	0	O(n log n)	O(log n)	250	5
rev_ord_5000	Natural MergeSort	31101	28876	O(n log n)	O(n)	60000	5000
rev_ord_5000	QuickSort (Median-of-Three)	4169103	7484	O(n log n)	O(log n)	60000	12
rev_ord_5000	QuickSort (Insertion 50)	8557123	4321	O(n log n)	O(log n)	60000	12
rev_ord_5000	QuickSort (First Pivot)	8557379	4785	O(n^2)	O(n)	25000000	5000
rev_ord_5000	QuickSort (Insertion 100)	8557761	4276	O(n log n)	O(log n)	60000	12

COMPARISON GRAPHS:






Swaps vs Theoretical Swaps by Number of Elements
Order=Ordered
Order=Random Ordered
Order=Reverse Ordered
Count
60k
40k
20k
0
0
2k
4k
6k
8k
10k
Number of Elements
Algorithm, Metric
Natural MergeSort, Swaps
Natural MergeSort, # Theoretical Swaps
QuickSort (Median-of-Three), Swaps
QuickSort (Median-of-Three), # Theoretical Swaps
QuickSort (Insertion 100), Swaps
QuickSort (Insertion 100), # Theoretical Swaps
QuickSort (Insertion 50), Swaps
QuickSort (Insertion 50), # Theoretical Swaps
QuickSort (First Pivot), Swaps
QuickSort (First Pivot), # Theoretical Swaps


Swaps vs Theoretical Swaps by Number of Elements
Zoomed-in for enhanced visualization
Order=Ordered
Order=Random Ordered
Order=Reverse Ordered
Count
20k
15k
10k
5k
0
0
2k
4k
6k
8k
10k
Number of Elements
Algorithm, Metric
Natural MergeSort, Swaps
Natural MergeSort, # Theoretical Swaps
QuickSort (Median-of-Three), Swaps
QuickSort (Median-of-Three), # Theoretical Swaps
QuickSort (Insertion 100), Swaps
QuickSort (Insertion 100), # Theoretical Swaps
QuickSort (Insertion 50), Swaps
QuickSort (Insertion 50), # Theoretical Swaps
QuickSort (First Pivot), Swaps
QuickSort (First Pivot), # Theoretical Swaps