Data Structures

EN.605.202.81.SP24

Lab 4: Sorting

Logan Griffin

4/30/2024

Analysis

Structures

For the quick sort algorithms, a standard array of integer was used. This was advantageous because swapping, comparing, and indexing required very little work. A single integer array also allowed the sorting to be done in place.

The natural merge sort used an array of linked lists to run. Although the linked list implementation was a requirement for the lab, they served as an efficient data structure. The linked list structure allowed for items to easily be inserted and deleted from each list. Additionally, there was no searching in the algorithm, so the linked lists did not need to be iterated through at all. The program only cared about the first item in the list. Pointers to the head node were updated every time an item was taken from a list to go into the merged list. While the array of linked lists was the same array for the entire sort, there was a new merged linked list created for every two linked lists that needed to be merged. Instead of linked lists, queues could have been used to achieve the same outcome since the algorithm needed a first in, first out data structure.

Storing the linked lists in an array aided the recursive nature of the function. Once two linked lists were merged, the array made it easy to insert the merged list into a specific index. Checking the array for empty values also made it easy to identify the base case for the sort of their only being one list left and nothing to merge with.

Implementation and Design

The purpose of this project was to observe the efficiencies of different sorting algorithms in terms of the amount of comparisons and swaps required for the sort.

Types:

- Quick sort variation 1: Select the first item of the partition as the pivot. Treat partitions of size one and two as stopping cases.
- Quick sort variation 2: Select the first item of the partition as the pivot. For a partition of size 100 or less, use an insertion sort to finish.
- Quick sort variation 3: Select the first item of the partition as the pivot. For a partition of size 50 or less, use an insertion sort to finish.
- Quick sort variation 4: Select the median-of-three as the pivot. Treat partitions of size one and two as stopping cases.
- Natural Merge

The program works by first taking the file paths for a folder containing input files, an output file to see the sorting process, and a separate output file to see how many comparisons and swaps were done by each algorithm. To read in the data, the program determines the number of data items by looking at the

name of the file. Therefore, it requires the use of a very specific naming convention for the input files. If this program were to actually be used in practice, a different method to determine the file size would need to be implemented.

Once the file size is determined, the file is read and an array is created for all of the quick sorts. The natural merge sort reads the file in separately, creating an array of linked lists that represent the natural runs in the data set.

The program loops through the folder, performing each of the sort types on every file in the folder. For the files that are size 50, the program writes the process to an output file using a buffered writer. There are also class variables to keep track of the number of comparisons and swaps performed during each sort. Once a specific sort variation is completed on a file, those metrics are written to a stats file, and then reset for the next variation.

The sorting algorithms were adapted from the class ZyBook. There were changes that had to be made to meet the lab requirements, but the foundation of the code came from the textbook section on the quick sort and merge sort. One of the design decisions that had to be made was how to store the linked lists for the natural merge. At first, I tried using an array of nodes that stored the head node of each linked list. I ran into referencing issues, so I opted for an array of linked lists rather than an array of single nodes.

What I Learned & What I would Do Differently

By completing this lab, I learned that there is always an optimal sorting algorithm for a data set. The more that is known about the data and its ordering can help determine which sorting algorithm should be used. When sorting was introduced in the class, I didn't realize that the type of sort could have such a large impact on performance and cost.

If the data is relatively ordered, a natural merge sort should be used over a quick sort. If a quick sort has to be used, a quick sort with an appropriate pivot selection can make a huge difference over choosing the first item in the partition. Adding insertion sort at the end of the quick sort algorithm has an effect on the cost as well. Once the partitions are small enough, it is more efficient to finish with an insertion sort compared to finishing with the quick sort.

If I were to do this lab again, I would try to make the code more modular. I ended up writing the sorting methods for the quick sort and merge sort in my main class, but they could have been implemented as separate classes. In practice, this would be the right way to go about writing the code, and it would make the overall main file much easier to read and follow.

Iterative vs. Recursive

I chose to implement the sorts using their recursive variations. The code for the recursive methods is more concise, and it logically makes more sense due to quick sort and merge sort being recursive in nature.

If the iterative versions were used, the sorts would likely take up less memory during the run by eliminating the need to store the state of the recursive calls. However, this difference is not apparent in run time or the number of comparisons and swaps.

Hypothesis on Sort Performance

Types:

- Quick sort variation 1: Select the first item of the partition as the pivot. Treat partitions of size one and two as stopping cases.

- Quick sort variation 2: Select the first item of the partition as the pivot. For a partition of size 100 or less, use an insertion sort to finish.
- Quick sort variation 3: Select the first item of the partition as the pivot. For a partition of size 50 or less, use an insertion sort to finish.
- Quick sort variation 4: Select the median-of-three as the pivot. Treat partitions of size one and two as stopping cases.
- Natural Merge

The first variation of quick sort should be the least efficient out of all 4 methods, especially for the ordered and reverse ordered data because they represent the worst case for that sort. This is because of the large partition sizes that will be created. For random data, it should be slightly better, but still not any better than the other variations.

Variations two and three of the quick sort should be more efficient than the first variation because of the insertion sort. Although insertion sort is a costly process for large data files, if it is only used when the partitions get down to size 100 and size 50, it should be more efficient than the quick sort at that set size.

Variation four of the quick sort should be more efficient than the other three. Optimizing a quick sort is about choosing an appropriate pivot. The median-of-three approach is a more effective pivot selection method than selecting the first item in the partition as the pivot every time. The median insures that a value to swap with will be found. The efficiency gains of the pivot selection method will be most visible in the ordered data sets, because the partitioning will be relatively even.

The natural merge sort should be the most efficient sorting method overall. For an ordered data set, it will require very little work, only making a single linked list. Sorting reverse ordered data will be more expensive, but should still require less work than the quick sorts. Sorting random data should fall somewhere in between ordered data and reverse ordered data in terms of cost.

Actual Sort Performance

The actual sort performances mostly aligned with the hypotheses. However, the variations of quick sort that implemented the insertion sorts, turned out to be more expensive in terms of comparisons and swaps than the first variation of the quick sort. This could be due to starting the insertion sort when the partitions are still too large. Variation 3, with the insertion sort starting at size 50 partitions, was more efficient than the insertion sort starting at partitions of size 100. Since the insertion sort is an overall more expensive algorithm, starting it too early can be counter-effective for efficiency. There may be an optimal partition size to start the insertion sort at. This could be tested by writing other variations, perhaps starting the insertion sort at partitions of size 20, 30, 40, 200, 300, 400, etc.

The natural merge sort out performed all of the quick sort variations in every aspect except for the number of comparisons for the randomized data sets, where it required more comparisons than the median-of-three quick sort method. Due to the way it was implemented, the natural merge sort also never required any swaps.

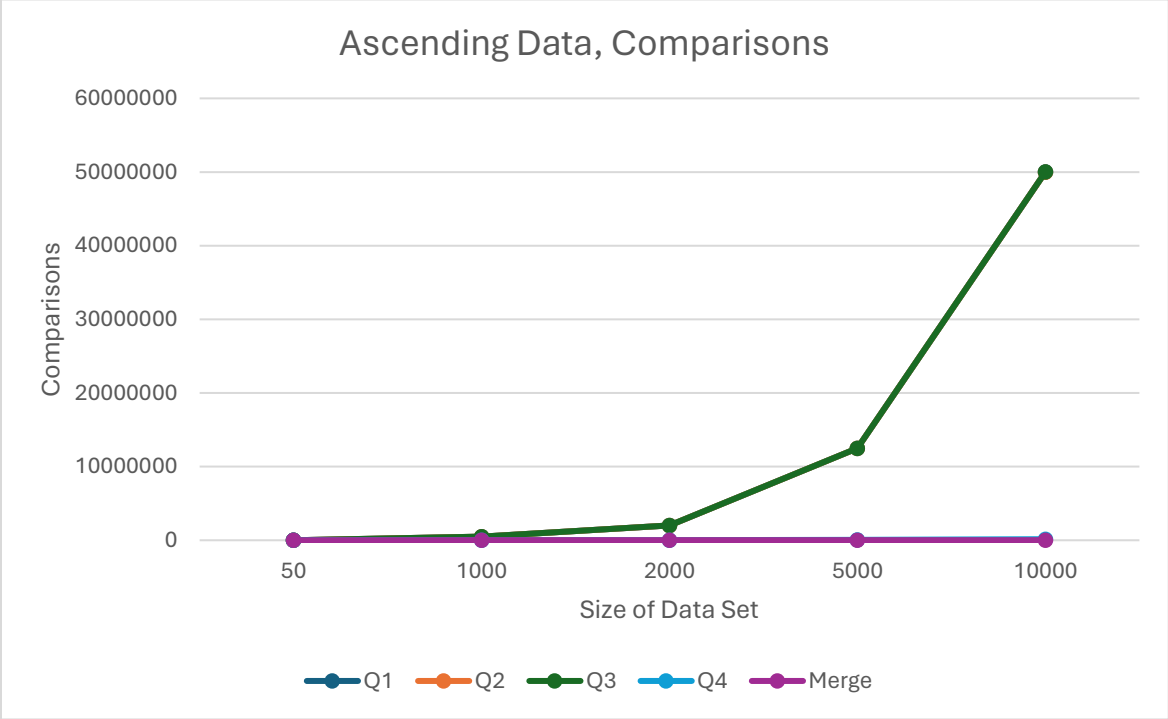
The effects of specific aspects on performance are discussed throughout the rest of the analysis.

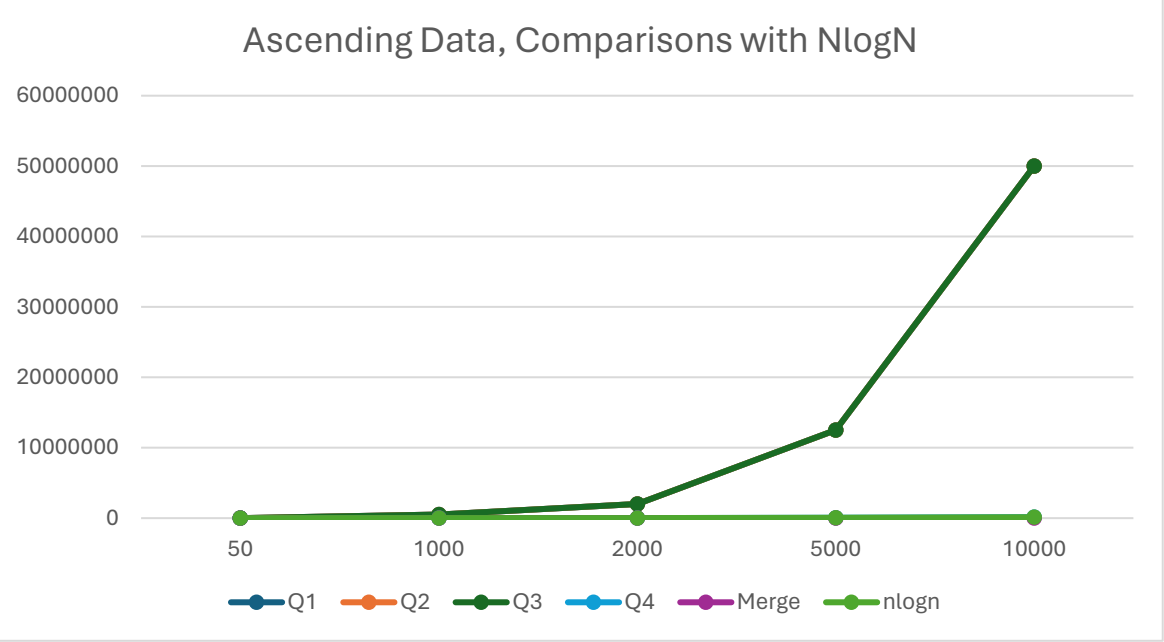
Observed Comparisons and Swaps

File Sorted	Type of Sort	Size	Comparisons	Swaps
asc1000.dat	Quicksort Variant 1	1000	500500	0
asc1000.dat	Quicksort Variant 2	1000	495551	0
asc1000.dat	Quicksort Variant 3	1000	499276	0
asc1000.dat	Quicksort Variant 4	1000	10975	0
asc1000.dat	MergeSort	1000	1000	0
asc10000.dat	Quicksort Variant 1	10000	50005000	0
asc10000.dat	Quicksort Variant 2	10000	50000051	0
asc10000.dat	Quicksort Variant 3	10000	50003776	0
asc10000.dat	Quicksort Variant 4	10000	143615	0
asc10000.dat	MergeSort	10000	10000	0
asc2000.dat	Quicksort Variant 1	2000	2001000	0
asc2000.dat	Quicksort Variant 2	2000	1996051	0
asc2000.dat	Quicksort Variant 3	2000	1999776	0
asc2000.dat	Quicksort Variant 4	2000	23951	0
asc2000.dat	MergeSort	2000	2000	0
asc50.dat	Quicksort Variant 1	50	1275	0
asc50.dat	Quicksort Variant 2	50	100	0
asc50.dat	Quicksort Variant 3	50	100	0
asc50.dat	Quicksort Variant 4	50	335	0
asc50.dat	MergeSort	50	50	0
asc5000.dat	Quicksort Variant 1	5000	12502500	0
asc5000.dat	Quicksort Variant 2	5000	12497551	0
asc5000.dat	Quicksort Variant 3	5000	12501276	0
asc5000.dat	Quicksort Variant 4	5000	66807	0
asc5000.dat	MergeSort	5000	5000	0
ran1000.dat	Quicksort Variant 1	1000	9835	2335
ran1000.dat	Quicksort Variant 2	1000	21108	16667
ran1000.dat	Quicksort Variant 3	1000	15430	10179
ran1000.dat	Quicksort Variant 4	1000	8774	2390
ran1000.dat	MergeSort	1000	9211	0
ran10000.dat	Quicksort Variant 1	10000	130865	31210
ran10000.dat	Quicksort Variant 2	10000	257531	191238
ran10000.dat	Quicksort Variant 3	10000	178635	103254
ran10000.dat	Quicksort Variant 4	10000	106741	31922
ran10000.dat	MergeSort	10000	128866	0
ran2000.dat	Quicksort Variant 1	2000	21115	5178
ran2000.dat	Quicksort Variant 2	2000	47011	37402
ran2000.dat	Quicksort Variant 3	2000	30010	18776
ran2000.dat	Quicksort Variant 4	2000	18589	5264
ran2000.dat	MergeSort	2000	20456	0
ran50.dat	Quicksort Variant 1	50	248	70
ran50.dat	Quicksort Variant 2	50	421	356

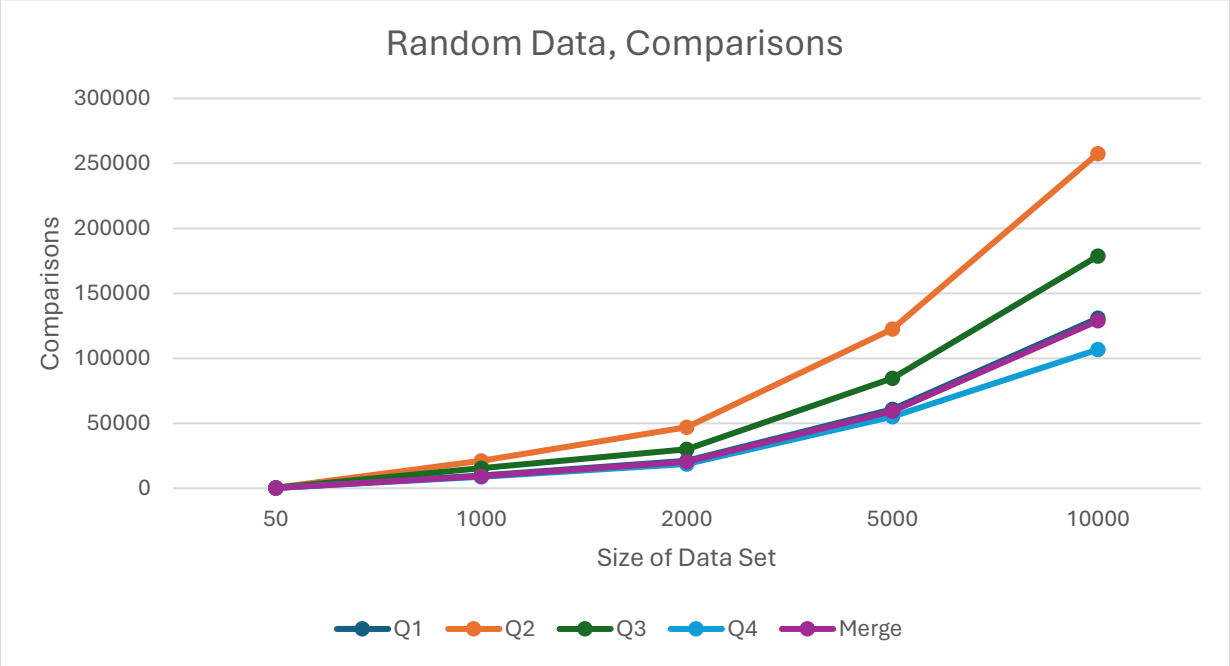
ran50.dat	Quicksort Variant 3	50	421	356
ran50.dat	Quicksort Variant 4	50	261	64
ran50.dat	MergeSort	50	244	0
ran5000.dat	Quicksort Variant 1	5000	60898	14712
ran5000.dat	Quicksort Variant 2	5000	122670	91044
ran5000.dat	Quicksort Variant 3	5000	84727	48698
ran5000.dat	Quicksort Variant 4	5000	55230	14796
ran5000.dat	MergeSort	5000	59282	0
rev1000.dat	Quicksort Variant 1	1000	500000	500
rev1000.dat	Quicksort Variant 2	1000	500400	5400
rev1000.dat	Quicksort Variant 3	1000	500450	1700
rev1000.dat	Quicksort Variant 4	1000	9976	500
rev1000.dat	MergeSort	1000	5931	0
rev10000.dat	Quicksort Variant 1	10000	50000000	5000
rev10000.dat	Quicksort Variant 2	10000	50004900	9900
rev10000.dat	Quicksort Variant 3	10000	50004950	6200
rev10000.dat	Quicksort Variant 4	10000	133616	5000
rev10000.dat	MergeSort	10000	74607	0
rev2000.dat	Quicksort Variant 1	2000	2000000	1000
rev2000.dat	Quicksort Variant 2	2000	2000900	5900
rev2000.dat	Quicksort Variant 3	2000	2000950	2200
rev2000.dat	Quicksort Variant 4	2000	21952	1000
rev2000.dat	MergeSort	2000	12863	0
rev50.dat	Quicksort Variant 1	50	1250	25
rev50.dat	Quicksort Variant 2	50	1227	1129
rev50.dat	Quicksort Variant 3	50	1227	1129
rev50.dat	Quicksort Variant 4	50	286	25
rev50.dat	MergeSort	50	182	0
rev5000.dat	Quicksort Variant 1	5000	12500000	2500
rev5000.dat	Quicksort Variant 2	5000	12502400	7400
rev5000.dat	Quicksort Variant 3	5000	12502450	3700
rev5000.dat	Quicksort Variant 4	5000	61808	2500
rev5000.dat	MergeSort	5000	34803	0

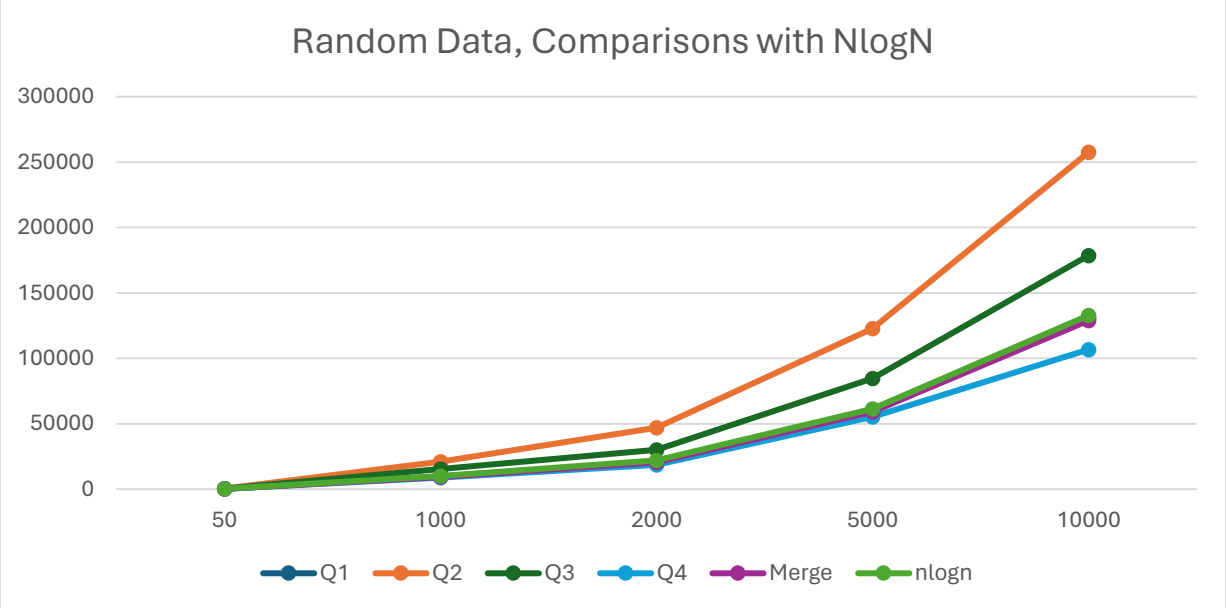
Ascending Data, Comparisons					
	50	1000	2000	5000	10000
Q1	1275	500500	2001000	12502500	50005000
Q2	100	495551	1996051	12497551	50000051
Q3	100	499276	1999776	12501276	50003776
Q4	335	10975	23951	66807	143615
Merge	50	1000	2000	5000	10000
nlogn	282	9966	21931	61349	132877



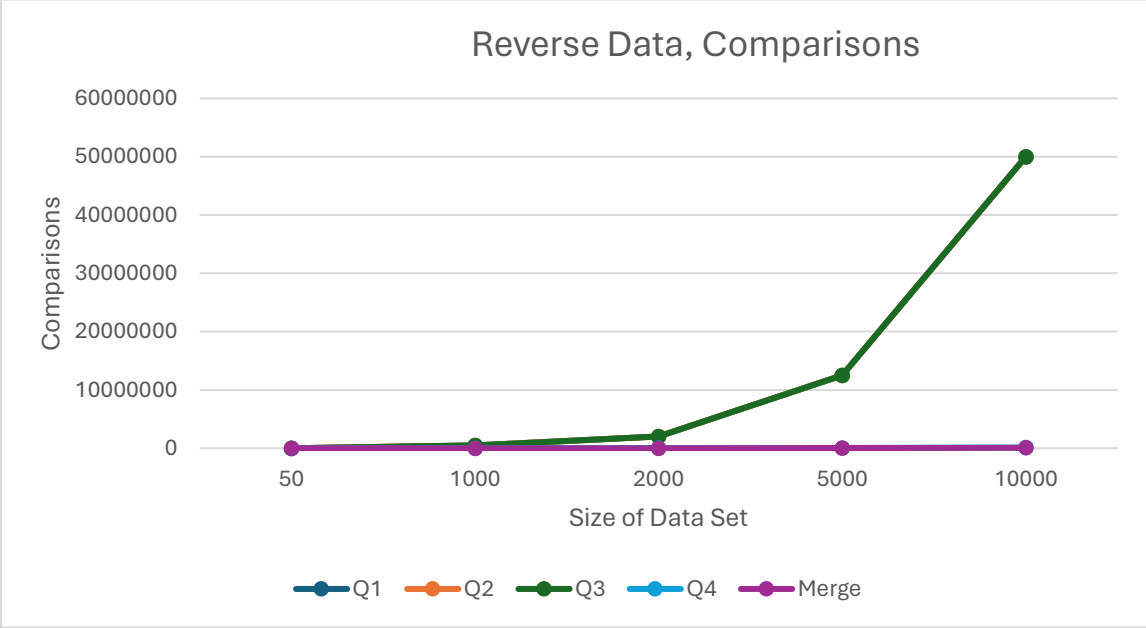


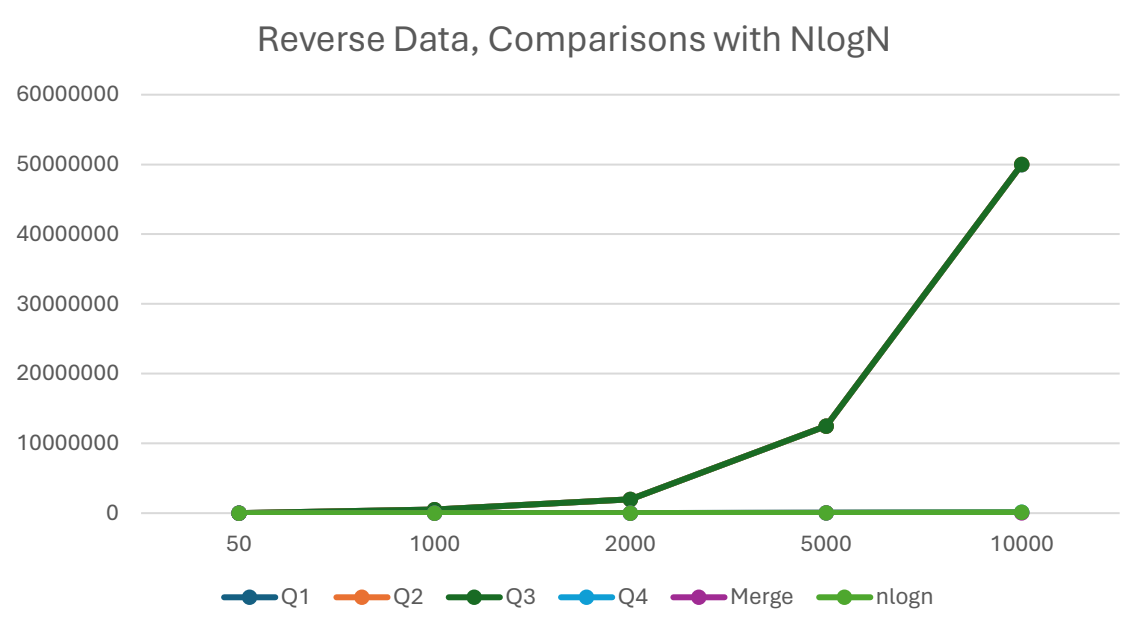
Random Data, Comparisons					
	50	1000	2000	5000	10000
Q1	248	9835	21115	60898	130865
Q2	421	21108	47011	122670	257531
Q3	421	15430	30010	84727	178635
Q4	261	8774	18589	55230	106741
Merge	244	9211	20456	59282	128866
nlogn	282	9966	21931	61349	132877



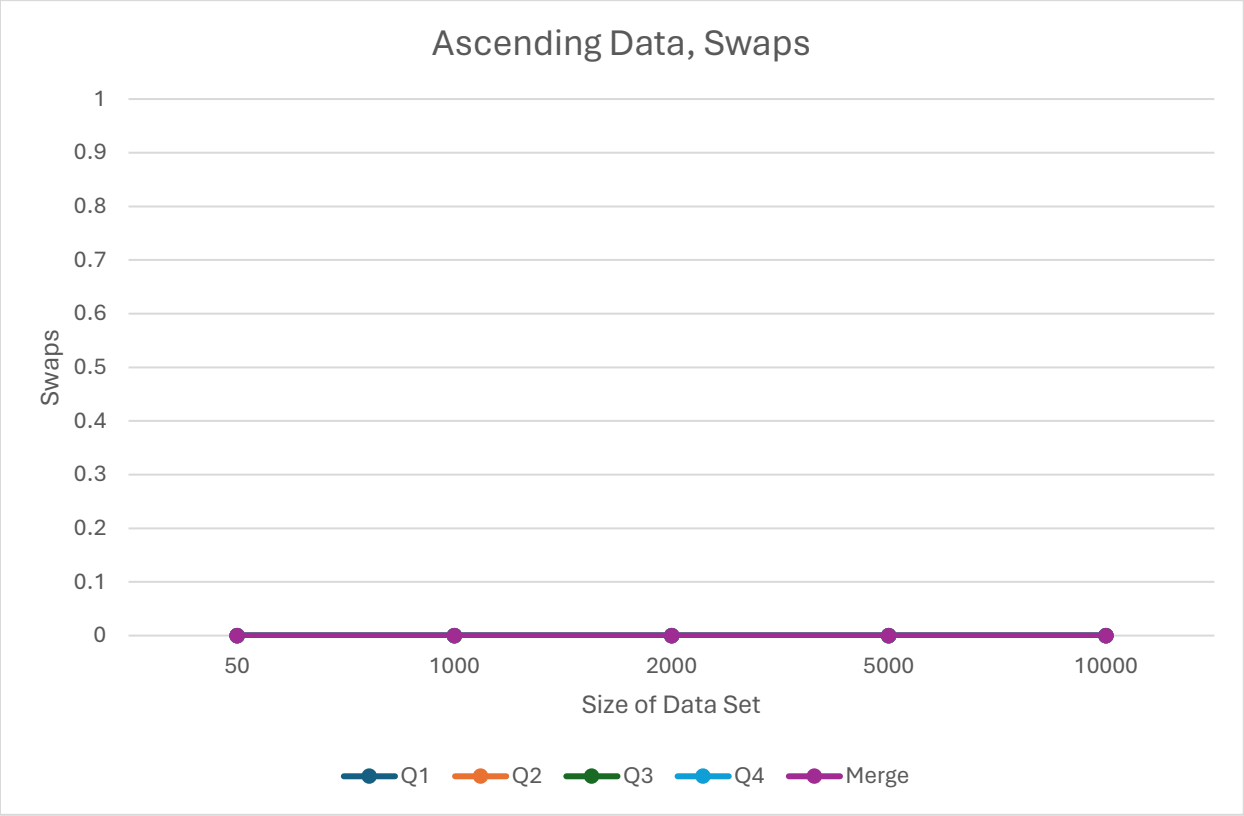


Reverse Data, Comparisons					
	50	1000	2000	5000	10000
Q1	1250	500000	2000000	12500000	50000000
Q2	1227	500400	2000900	12502400	50004900
Q3	1227	500450	2000950	12502450	50004950
Q4	286	9976	21952	61808	133616
Merge	182	5931	12863	34803	74607
nlogn	282	9966	21931	61349	132877

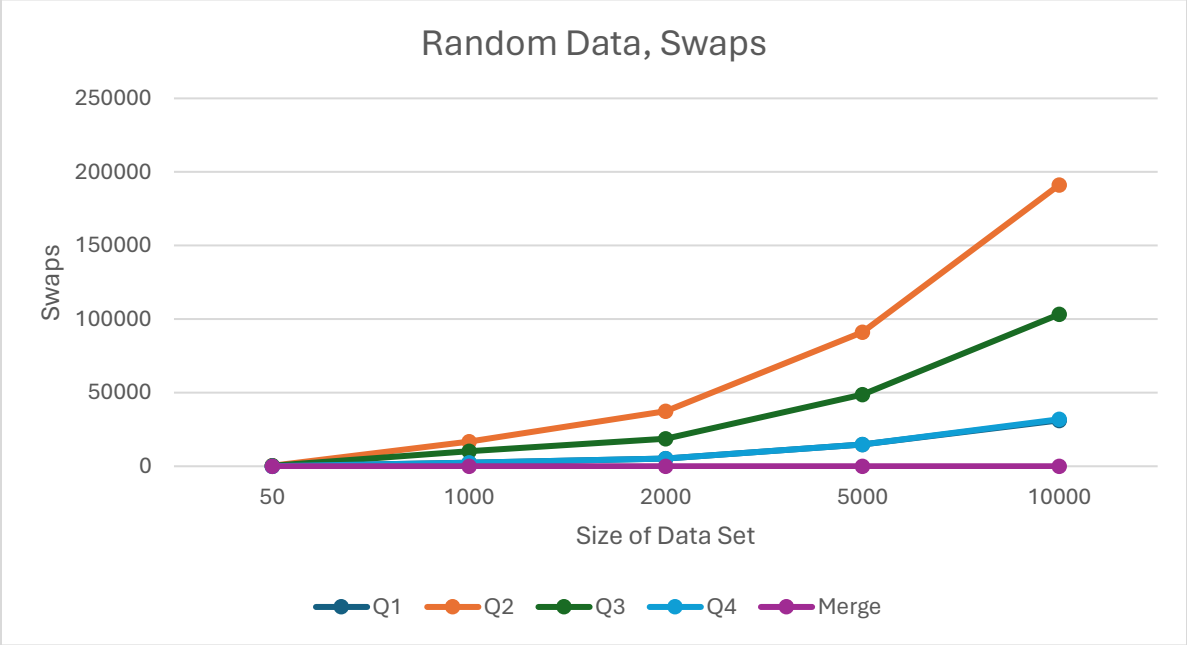


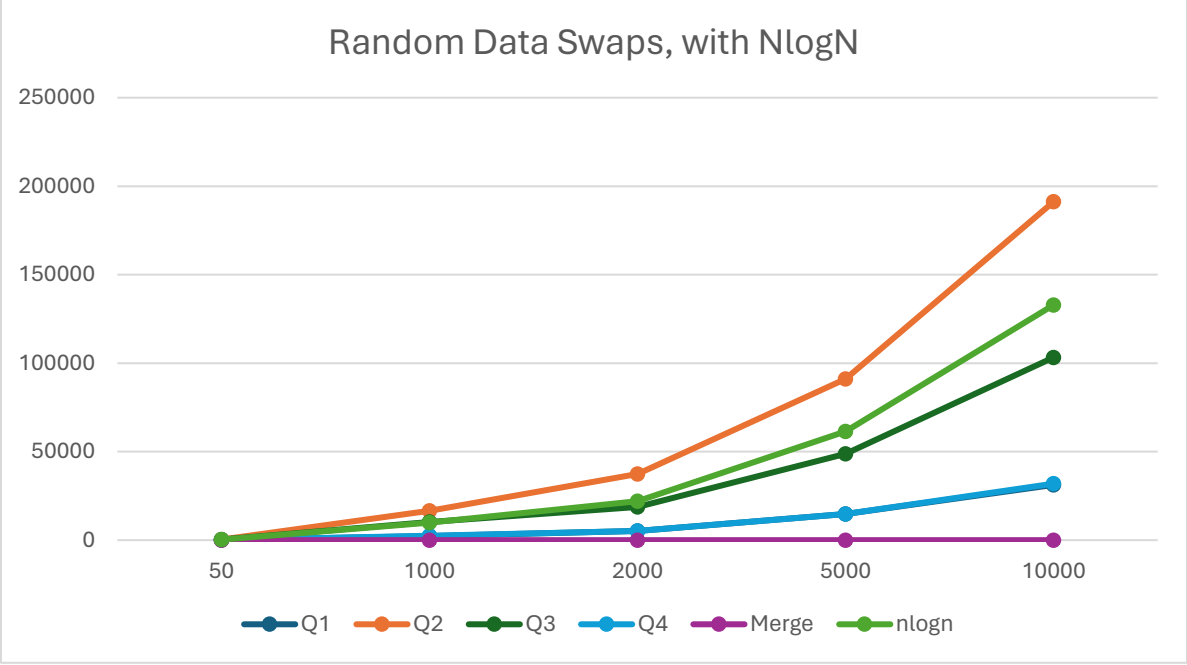


Ascending Data, Swaps					
	50	1000	2000	5000	10000
Q1	0	0	0	0	0
Q2	0	0	0	0	0
Q3	0	0	0	0	0
Q4	0	0	0	0	0
Merge	0	0	0	0	0

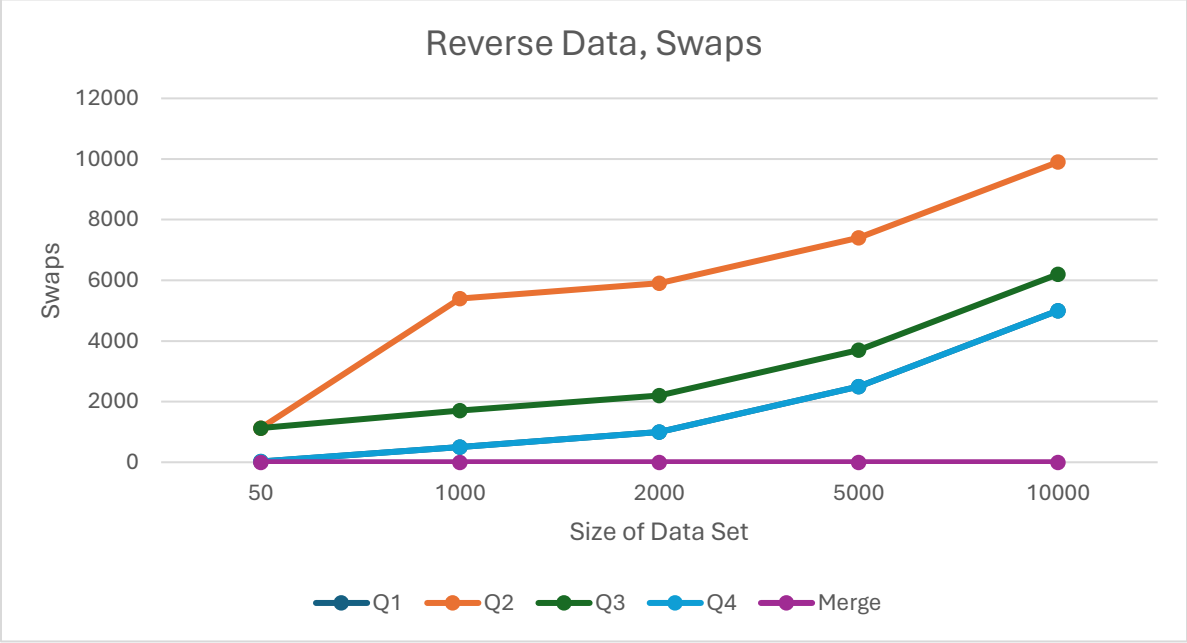


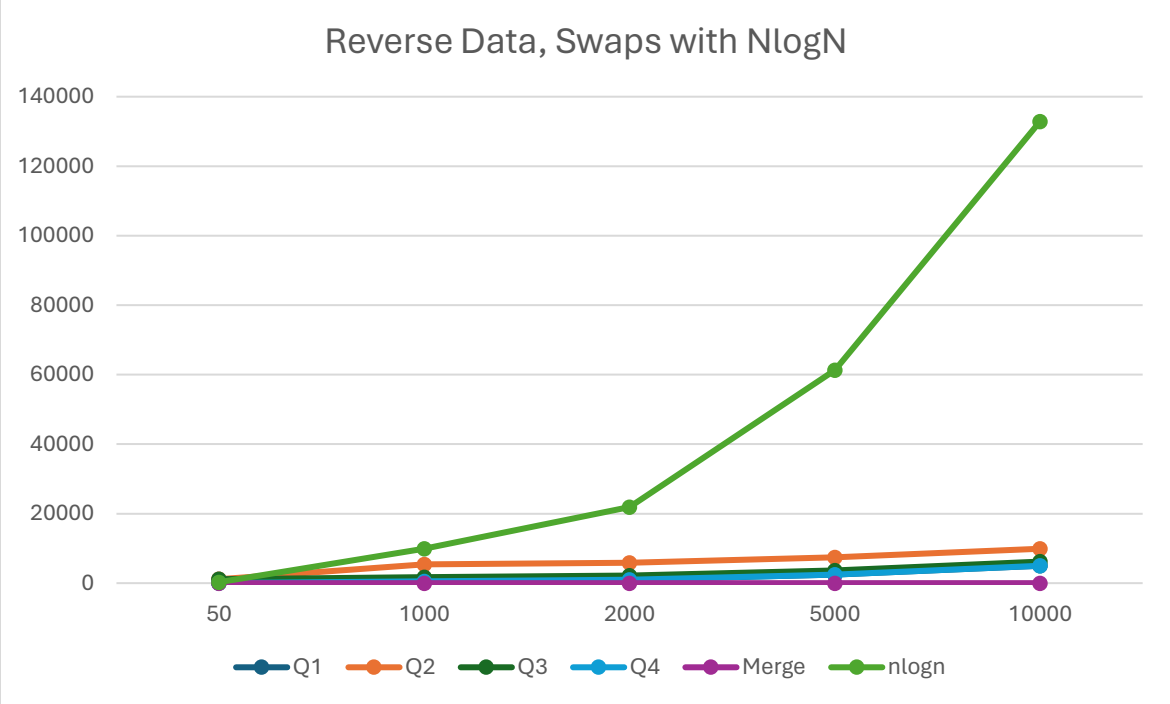
Random Data, Swaps					
	50	1000	2000	5000	10000
Q1	70	2335	5178	14712	31210
Q2	356	16667	37402	91044	191238
Q3	356	10179	18776	48698	103254
Q4	64	2390	5264	14796	31922
Merge	0	0	0	0	0
nlogn	282	9966	21931	61349	132877





Reverse Data, Swaps					
	50	1000	2000	5000	10000
Q1	25	500	1000	2500	5000
Q2	1129	5400	5900	7400	9900
Q3	1129	1700	2200	3700	6200
Q4	25	500	1000	2500	5000
Merge	0	0	0	0	0
nlogn	282	9966	21931	61349	132877





Effects of Order

The quick sort variants that used the first item in the partition fared the worst in regards to comparisons on both in order and reverse ordered data. This is because at every recursive partition call, the entirety of the partition had to be compared, creating a compounding effect. On the other hand, the median-of-3 pivot quick sort eliminated many of the comparisons by reducing the number of recursive calls. For the merge sort, only one round of comparisons had to be done to know that the data was already in order. This makes the natural merge sort the best choice for trying to sort data that is already in order or is relatively ordered.

When the order of the data was random, all of the sorting algorithms were closer in performance.. The quick sort variations using the insertion sort had the worse performance for both comparisons and swaps with the random data. This is due to insertion sort having a high complexity (O(n^2)). Even for the large file sizes, the use of insertion sort made a substantial difference in the performance over the other quick sort variants. The first quick sort variant, although only choosing the first item in the partition as the pivot, was comparable to the natural merge sort for random data. This was surprising, as I thought the merge sort would have greatly outperformed it. After thinking about it, random data is obviously not the worst case, but it will still create a much larger number of lists to merge than relatively ordered data would. For comparisons, the best performing algorithm with random data was the median-of-3 pivot quick sort. For swaps, the natural merge sort was the best performing, requiring no swaps. However, the creation of new linked lists was not measured, but would factor into the overall performance of the algorithm.

Effect of File Size

Logically, an increase in file size will weaken the performance of the algorithms . The effect of file size was most clearly shown in the randomly ordered files. The quick sort algorithms using the insertion sort were increased the number of comparisons and swaps at the highest exponential rate, while the other algorithms increased at a much more gradual rate. Considering the highest input file size was only 10000, any files substantially larger would show a massive gap in performance for the insertion sort algorithms.

However, there were a few important observations about reverse ordered data and the number of swaps required. As the file size of reverse ordered data grew, the number of swaps required by each algorithm, other than the natural merge sort, was linear.

Effect of Partition Sizes & Pivot Selection

Partition size and pivot selection played a major role in the performance of the sorting algorithms. In the quick sorts, the pivot selection directly impacted the partition size. Larger partitions led to an increase in the total number of comparisons and swaps. Since the algorithm is recursive and calls itself again on smaller partitions each time based on the pivot, the low end pivot selection requires many more runs. The median-of-3 pivot selection needed fewer runs to complete the sort. If this was thought about like a tree, with each new leaf representing a partition, the low end pivot selection creates a very unbalanced tree with a depth close to n. On the other hand, a median selection will, on average, split the partition more evenly. If the low or high value was used as the pivot in the median-of-3 sort, it would also take less comparisons, again, on average, to find a value that is suitable for swapping. Overall, a more even partition split and pivot selection led to better performance.

Natural Merge & Effects

The natural merge sorting algorithm was the outlier in the algorithms tested. The natural merge sort created a linked list for every natural “run” in the data set. For example, in a data set of [1, 4, 3, 5, 9],

there would be three separate linked lists formed: (1, 4), (3, 5), and (9). After all of the linked lists have been formed, the lists are merged in pairs of two using a standard merge sort algorithm. Looking at the same example, lists (1, 4) and (3, 5) would be merged one integer at a time to create a single list (1, 3, 4, 5). (9) would be merged with an empty list and remain the same. The natural merge is then called recursively and merges the two remaining lists to create a final list (1, 3, 4, 5, 9). The algorithm does require a new linked list to be made every time, but this eliminates the need for swapping and storing temp nodes or values.

Due to its process, the natural merge shines when the data set is already somewhat ordered, or is known to have a significant number of natural runs. Such a data set would result in a fewer number of linked lists created in the beginning, leading to fewer recursive calls. The data gathered in the lab supported this. The ordered data sets represented the best case for the natural merge, where it required only n comparisons and no swaps to complete the sort.

With random data, the merge sort performed averagely. Reverse ordered data is the worst case run for a natural merge sort because it has to create a linked list for every element in the data set. This leads to the maximum number of recursive calls, and the smallest incremental gain when merging two lists together. Even so, the natural merge outperformed the other quick sort algorithms with the reversed data.

Compared to a regular merge sort, the natural sort is more efficient. While the natural merge keeps the "runs" together when reading in the data, a regular merge sort would ignore that they are already in order and split them up to be sorted anyways. This makes a huge difference for data that is already somewhat in sorted order. The regular merge sort algorithm usually uses two auxiliary arrays and merges into a third array that is then inserted back into the data set. The natural merge uses the already existing linked lists and only creates one new linked list, plus, it deletes the old linked lists to free up space.

Most Crucial Factor

Within the quick sorting algorithms tested, the use of insertion sort was the most crucial factor in performance, followed by the pivot selection as discussed earlier. The insertion sorting algorithm requires a much higher number of comparisons. Both the variants that used the insertion sort performed poorly across all of the input files. With random and reverse ordered data, the insertion sort starting at partitions of size 50 outperformed the sort switching to insertion sort at partitions of size 100. Since that was the only difference in the process for those two sorts, starting the insertion sort earlier was the only reason the performance was worse. The algorithm without the insertion sort fared much better performance wise across the board.