Analysis of Lab 4

Sean Dickson
4/25/25

Table of Results:

Type	Size	Count	Quicksort v1	Quicksort v2	Quicksort v3	Quicksort v4	Natural Merge Sort	Merge Sort (normal)
Asc	**50**	**Comparisons**	1321	50	50	392	50	286
		Exchanges	0	0	0	0	0	0
	1000	**Comparisons**	500737	496590	500415	11555	1000	9976
		Exchanges	5	5	5	6	0	0
	2000	**Comparisons**	2003007	1999860	2003686	25103	2000	21952
		Exchanges	11	10	11	11	0	0
	5000	**Comparisons**	12507524	12507377	12511202	71592	5000	61808
		Exchanges	28	27	27	28	0	0
	10000	**Comparisons**	50015048	50019901	50023726	153190	10000	133616
		Exchanges	52	51	51	52	0	0
Ran	**50**	**Comparisons**	372	594	594	468	640	286
		Exchanges	62	544	544	106	364	105
	1000	**Comparisons**	15118	22659	17142	14799	245925	9976
		Exchanges	2368	16565	9529	3243	162510	4269
	2000	**Comparisons**	31204	47501	34237	32825	990060	21952
		Exchanges	5184	34589	18420	7038	676446	9613
	5000	**Comparisons**	91726	137190	102902	87799	6225434	61808
		Exchanges	14446	91831	51243	19154	4133075	27171
	10000	**Comparisons**	215788	302437	234070	197311	25007429	133616
		Exchanges	30554	182331	100197	40387	16753290	59136
Rev	**50**	**Comparisons**	1345	1275	1275	396	1324	286
		Exchanges	25	1225	1225	28	1225	133
	1000	**Comparisons**	499615	500367	500492	11559	498895	9976
		Exchanges	504	5404	1704	509	496896	4930
	2000	**Comparisons**	1998782	2000623	2000656	25107	1991566	21952
		Exchanges	1005	5902	2205	1014	1987567	10858
	5000	**Comparisons**	12467041	12471854	12471905	71596	12434871	61808
		Exchanges	2512	7411	3712	2531	12424872	29790
	10000	**Comparisons**	49897492	49907220	49907345	153194	49759495	133616
		Exchanges	5025	9925	6225	5055	49739496	64587

(See charts based off this data at the bottom of this file!)

Comparison of Quicksort Algorithm Variations:

Among the quicksort variants, all (with the exception pf variant 4) performed best with randomized data rather than sorted data. For variants 1 through 3, working with sorted data resulted in a performance curve resembling O(n^2), while random data yielded a curve closer to O(n*log(n)). This suggests that the order and size of the data have a more significant impact on performance than the choice of stop-case (whether using insertion sort to finish or stopping at 1-2-sized partitions).

Quicksort variant 4, which uses a median-of-three pivot selection, was an exception to this trend. While the other variants exhibited O(n^2) performance with sorted data, variant 4 seemed to maintain roughly O(n*log(n)) performance across all data types. This indicates that the median-of-three pivot selection method played a critical role in enhancing the algorithm's performance, regardless of data size or order.

Comparison of Quicksort vs Natural Merge Algorithms:

Natural Merge Sort (NMS) consistently outperformed all quicksort algorithms when the data was sorted in ascending order, which was the expected result given that NMS is optimized for "nearly sorted" data.

However, as the data transitioned from ascending to random or descending order, the efficiency of NMS significantly declined, giving way to the quicker performance of the quicksort algorithms. While NMS maintained an O(n*log(n)) time complexity for sorted data, its performance deteriorated to O(n^2) as the data became less ordered. This implies that, for NMS, the initial order of the data is the most important factor in determining its efficiency.

Comparison of Merge Sort vs Natural Merge Sort Algorithms:

When comparing Natural Merge Sort (NMS) to the standard Merge Sort algorithm, it becomes clear that the advantages of NMS diminish as the data becomes less sorted. As previously mentioned, NMS performs with an O(n*log(n)) complexity for sorted or “nearly sorted” data, slightly outperforming normal Merge Sort in this scenario.

However, as the data becomes less organized, Merge Sort maintains its O(n*log(n)) time complexity, while NMS's performance degrades, approaching an O(n^2) curve. This suggests that normal Merge Sort is more efficient for handling unsorted data, whereas NMS is more beneficial when working with “nearly sorted” datasets.

Justification for Recursion as Opposed to Iteration:

Recursion appears to be the more intuitive approach for implementing these algorithms. In the case of quicksort, the process involves "partitioning" a list into progressively smaller sub-lists. Similarly, both Natural Merge Sort and standard Merge Sort break the list down recursively into individual items, which are then "merged" back together in sorted order. Given the structure of these algorithms, recursion seemed like the natural choice over iteration.

However, recursion is not without its drawbacks. The memory overhead required to execute these functions is considerable, and at one point, I had to manually increase Python’s default recursion limit by 20x. While this was a straightforward fix, it highlights a key limitation of recursion – something that iteration could avoid more effectively.

Description and Justification of Module Design:

The Lab4 Python Package is broken into the following files and folders:

__init__.py

This file allows Python to recognize this folder as a Python package and exposes the process_files() function found in lab4.py.

__main__.py

This file is run when the package is called as a standalone program. The file uses the native "argparse" library to parse command line arguments from the user. The file then passes these arguments to the process_files() function, executing the lab4.py program.

lab4.py

This file contains the main functions for the Lab4 project, including all sort algorithm functions and the 'process_files()' function for file I/O.

README.md

This file contains instructions on how to run the package as a standalone program and includes relevant details on language and IDE used to develop the module.

lab4/input_files/

This directory contains a series of my own input files (using Python's "random" package), containing txt files representing lists of integers of size 50, 1000, 2000, 5000, and 10000, each in 3 different orders (random, ascending, reverse).

lab4/output_files/

This directory contains all output files – one for each corresponding input file. Each output file contained the output from all sorting methods enacted on the input.

Note: Output files include data from all 6 sorting methods. This was an intentional choice, to reduce the number of total output files from 90 down to 15.

I designed each file with readability and compartmentalization in mind. Each file is no longer than a page and has its own unique purpose, making readability and debugging relatively easy.

What I Learned and What I Might do Differently Next Time:

When I began the project, recursion immediately seemed like the more intuitive choice for implementing these functions, as opposed to iteration. While I still stand by my decision to use recursion for these sorting algorithms, it did introduce significant overhead challenges. As part of the lab, I had to manually increase my recursion limit well beyond the default of 1000. Had I anticipated this issue from the start, I might have opted for an iterative approach instead.

In terms of subject matter, the lab provided valuable insights into which sorting algorithms work best in different scenarios. Completing it has deepened my understanding of these various sorting techniques and their use-cases.

<u>Discussion of Enhancements:</u>

1. **Merge Sort Function (Normal) and Corresponding Output:**

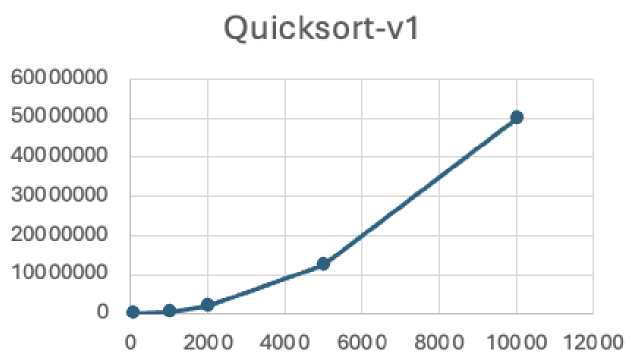
To make the comparison between Natural Merge Sort and the normal Merge Sort algorithms easier to quantify, I have included the normal Merge Sort function as part of this lab. In the output files, I've tracked the number of comparisons and exchanges for both algorithms in the same format for consistency.

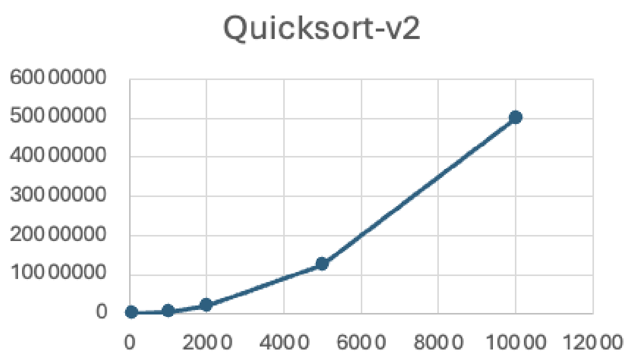
2. **Added headers to output file for clarity**

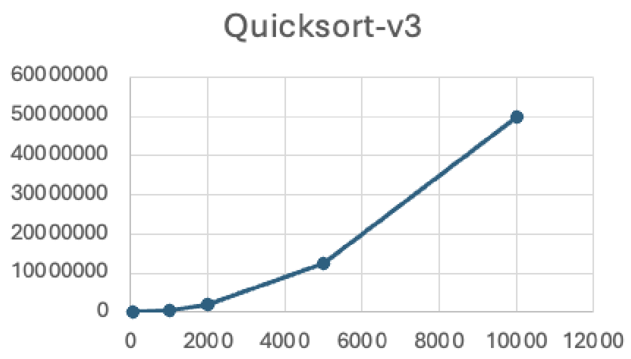
In the output files, I made several cosmetic choices for ease of analysis. Each sorting method is clearly identified with its own header for easy differentiation.

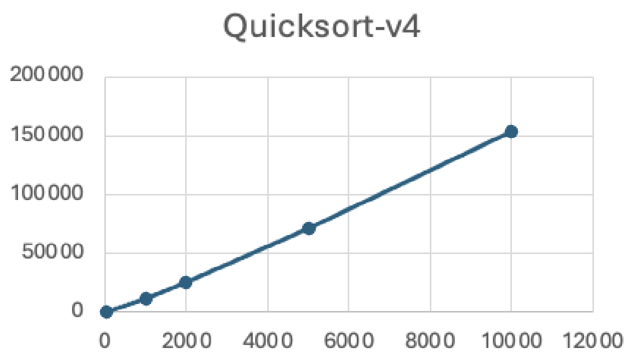
<u>Graphs Comparing Observed vs Theoretical Cost:</u>

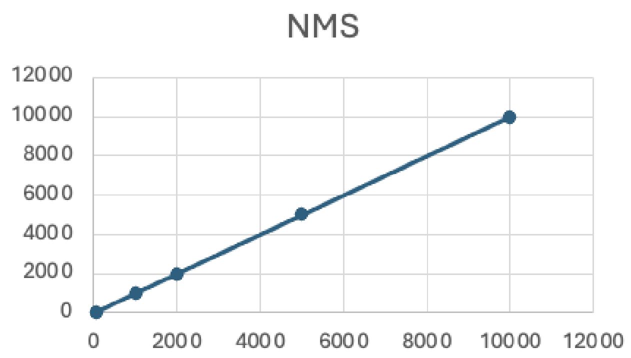
Ascending-Order (Sorted) Files

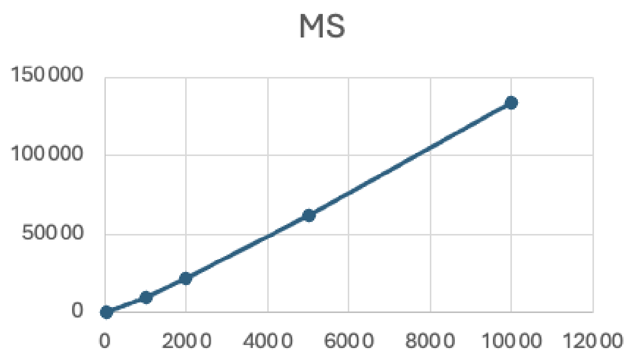




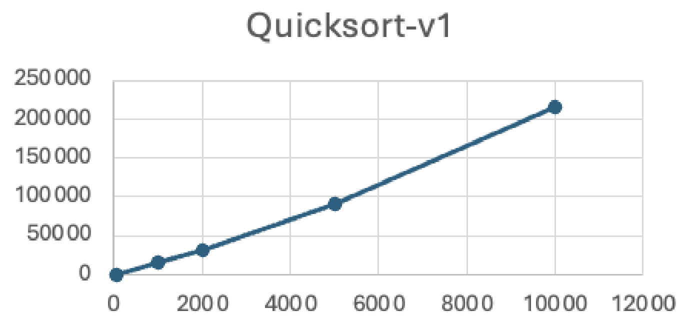


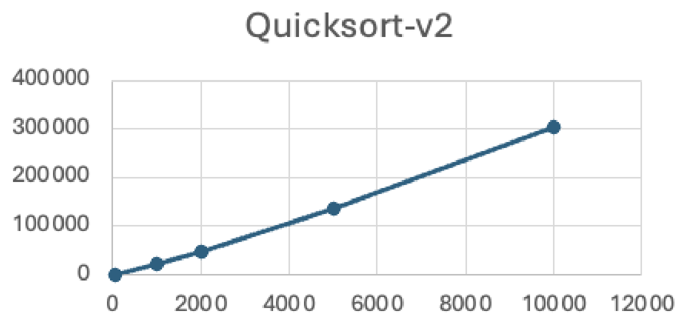


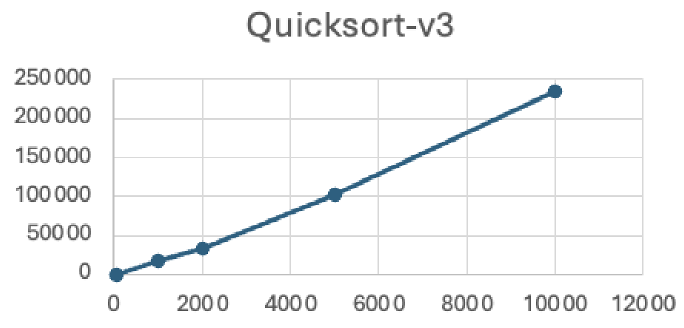


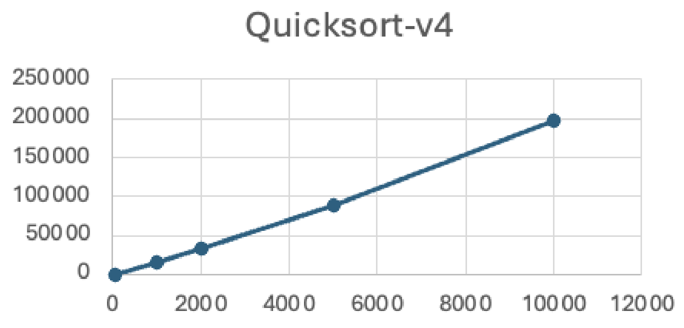


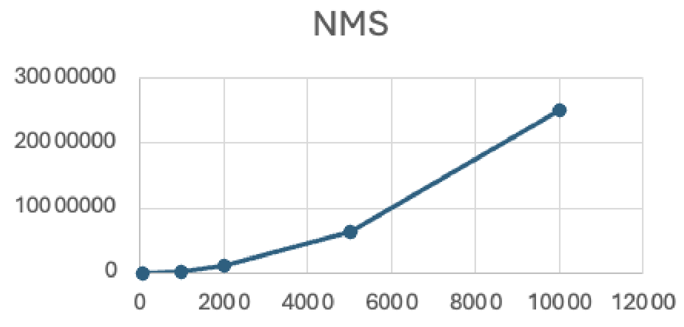
Random-Order Files

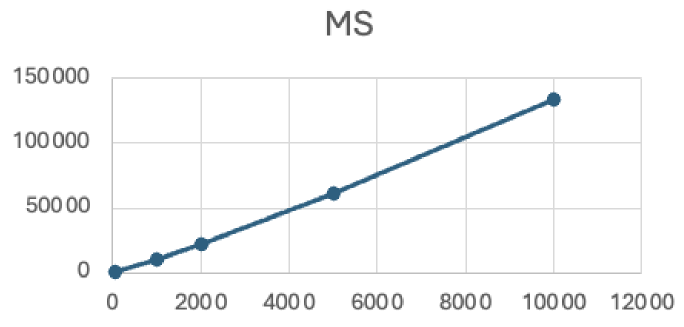




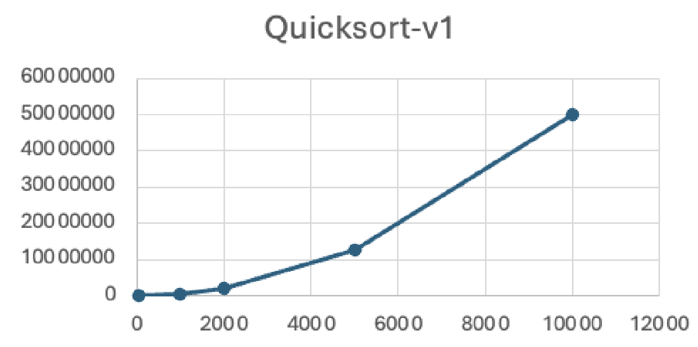


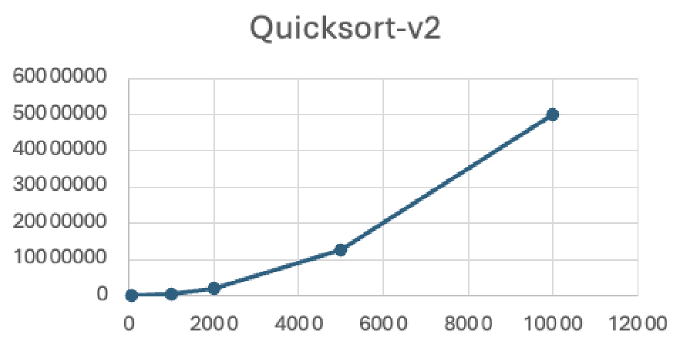


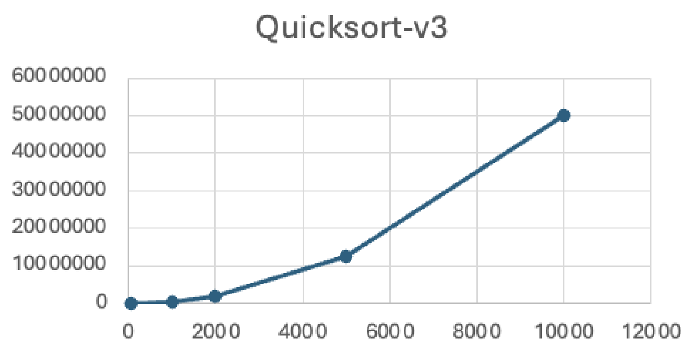


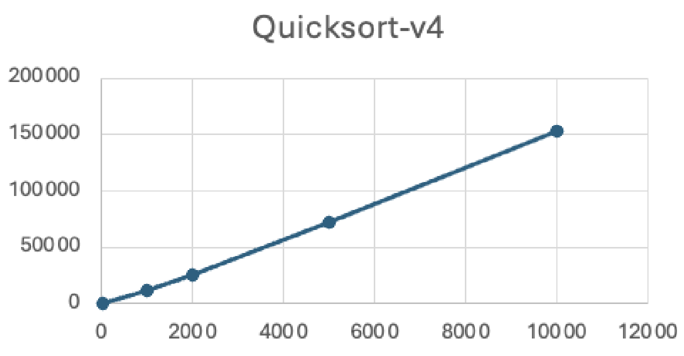


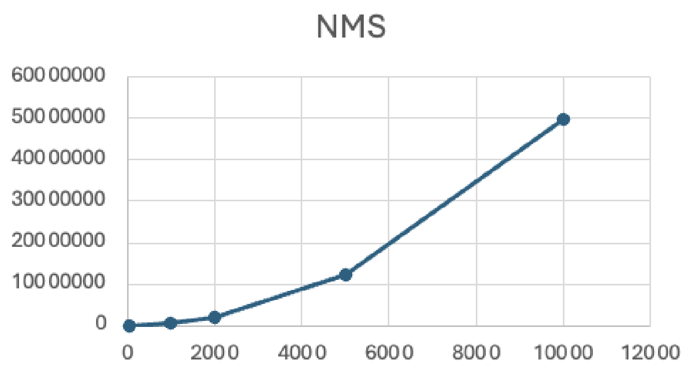
Reverse-Order Files

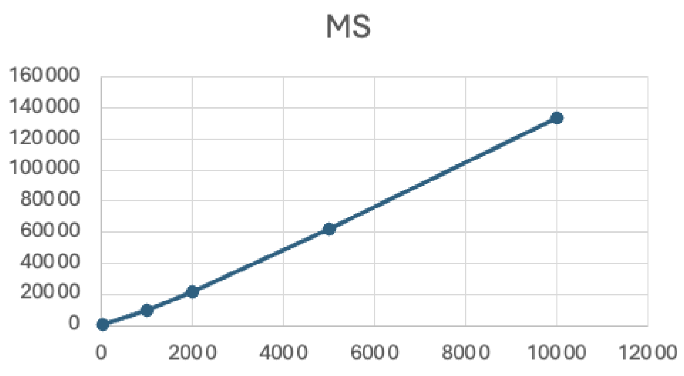




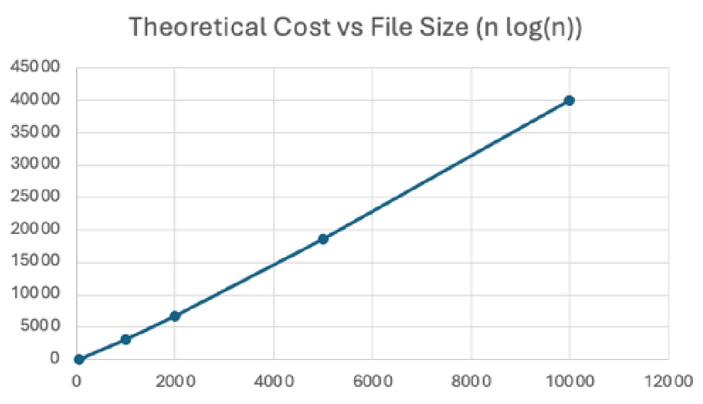








Theoretical Costs | n log(n)


Theoretical Cost vs File Size (n log(n))