605.202: Introduction to Data Structures

Claire Harelson

JHU ID: charels1

Lab 4 Analysis

Due Date: May 2, 2023

Dated Turned In: May 2, 2023

Lab 4 Analysis

For this lab I utilized various data structures to carry out various types of sorts. I used a linked list implementation for my natural merge sort by creating a linked list class then initializing a list. This was useful to keep track of pointers to data, which is beneficial in that the program reassigns the pointers rather than moving data around. The linked list implementation works to reduce the space complexity of a natural merge. My merge sort function goes through my list and calls to a secondary function to determine the next stopping point of the data. If the sequential data values are larger than the previous one, they remain in place. Once a smaller data value is hit, the values will be swapped, and the list header is updated. The reverse function takes advantage of the perks of natural merge, capitalizing on the sections of data that are already ordered. Instead of moving data points around, the head and tail pointers of the linked list are simply updated. The time complexity of a natural merge sort is O(nlogn) for best, average, and worst case. The space complexity is O(n) for the linked list implementation. Merge sort performed better on data that was ordered either ascending or reverse and was less efficient on randomly ordered data. In terms of input file size, merge sort was more efficient on smaller sizes than quick sort.

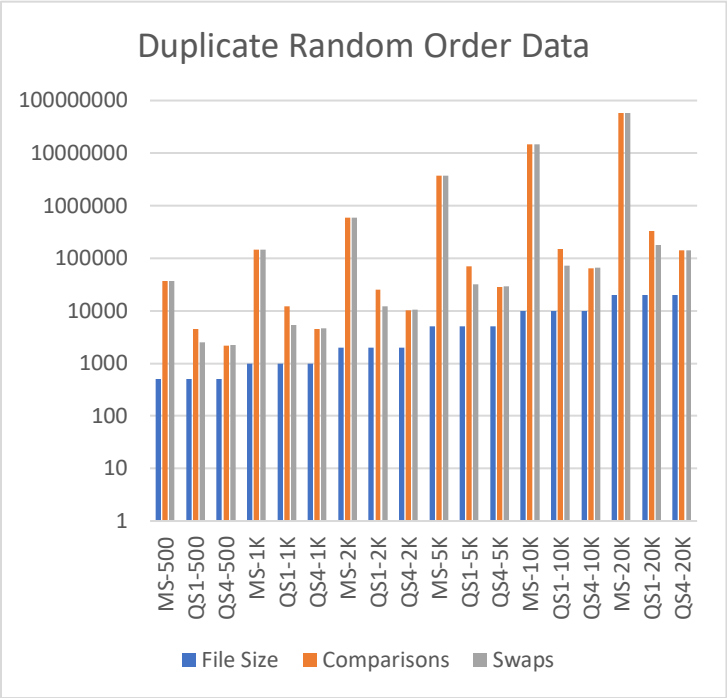
For quicksort I used a stack data structure to manipulate my data, which was beneficial for implementation of a pivot. Three of my quick sorts picked the first value as the pivot and carried out partitioning and sorting from there. My fourth quick sort utilized the median of three approach to pick the pivot, which proved to be more efficient for larger data sets and randomly ordered data sets. The other pivot selection approach worked more efficiently in all other cases. Two of my quick sorts were meant to utilize insertion sort for partition sizes of 50 and 100 or smaller. Unfortunately, I couldn't properly figure out how to determine partition size continuously throughout the sort in order to do this. Although not ideal, I instead told my algorithm to apply insertion sort if the file size was 50 or 100. Therefore, my code isn't completely correct in that any file larger than 50 or 100 do not properly call to insertion sort, resulting in the same statistics for all three sorts that use the same pivot selection. Because of this, I decided to only graph quicksort type 1, type 4, and merge sort for the larger files. I did graph all five sort types for files of size 50 and from this was able to conclude that insertion sort is more efficient when the data is in ascending order. Quicksort has a best-case and average-case time complexity of O(nlogn) and worst case of O(n^2). The space complexity is O(n).

For enhancements I included a runtime metrics function to analyze the time it took for each sorting algorithm to execute. Unfortunately, I ran out of time to sufficiently analyze the runtime data, but this would be an interesting factor to look at and compare as well. I also included a small error handling addition in terms of file input from the user. The addition of more extensive error checking and handling approaches would have been beneficial to my code. I decided to take an iterative approach rather than recursive simply due to personal preference as I have a tough time thinking recursively. I believe that a recursive approach would likely be more efficient in terms of both space and time complexity.

Something I could have done much better in this lab would be to improve the modularity. I have a LOT of redundancy of code which isn't beneficial in terms of space usage. Ideally my quicksort function would be able to call to separate pivot and insertion functions rather than having multiple copies of a quicksort function to do this. Additionally, I wasn't able to figure out how to properly carry out the call to the insertion function depending on partition size, so this is something that needs to be improved as well. Overall, I learned a good amount about algorithm efficiency and in which cases certain algorithms work better depending on input.

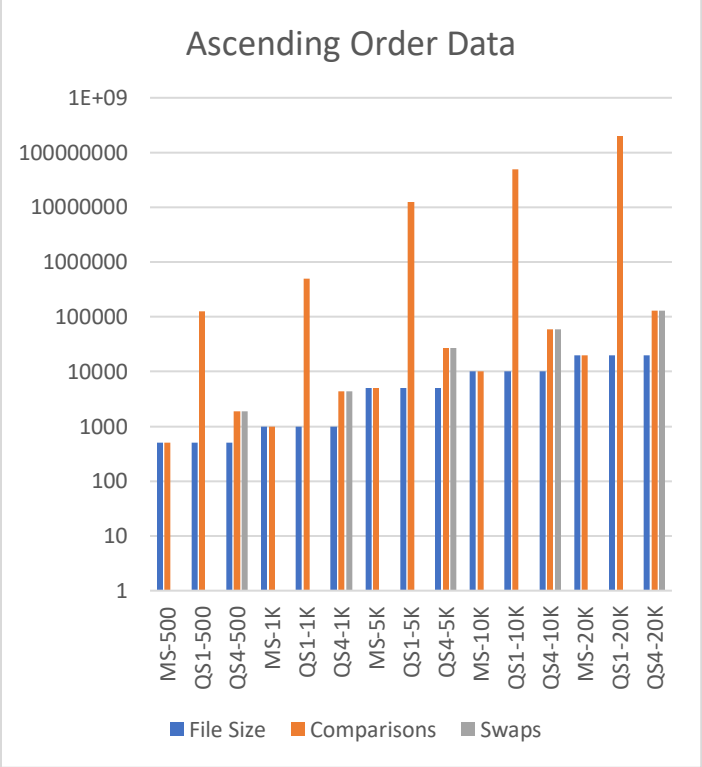
Graphs and Tables

All graphs are on a logarithmic scale of base 10 for ease of visualizing data. Key: MS = merge sort, QS1 = type 1 quicksort, QS2 = type 2 quicksort, QS3 = type 3 quicksort, QS4 = type 4 quicksort



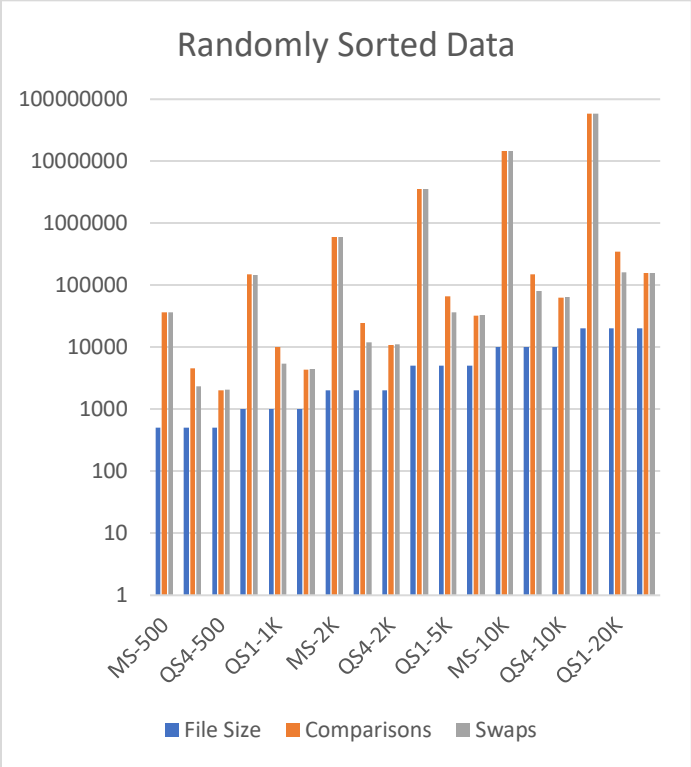
Duplicate Randomly Ordered Data

Sort Type	File Size	Comparisons	Swaps
MS-500	500	36998	36835
QS1-500	500	4526	2533
QS4-500	500	2168	2225
MS-1K	1000	146281	145985
QS1-1K	1000	12294	5431
QS4-1K	1000	4545	4644
MS-2K	2000	589307	588695
QS1-2K	2000	25188	12223
QS4-2K	2000	10348	10549
MS-5K	5000	3690389	3688890
QS1-5K	5000	70426	32300
QS4-5K	5000	28625	29134
MS-10K	10000	14711459	14708441
QS1-10K	10000	150148	72482
QS4-10K	10000	64573	65608
MS-20K	20000	58511951	58505989
QS1-20K	20000	331603	180375
QS4-20K	20000	140791	142902



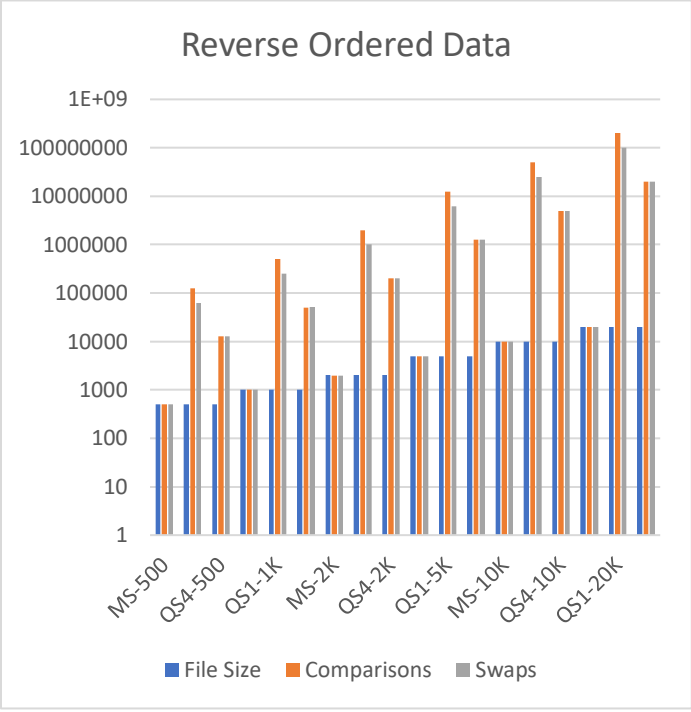
Ascending Ordered Data

Sort Type	File Size	Comparisons	Swaps
MS-50	50	49	0
QS1-50	50	1225	0
QS2-50	50	49	0
QS3-50	50	49	0
QS4-50	50	110	110
MS-500	500	499	0
QS1-500	500	124750	0
QS4-500	500	1906	1906
MS-1K	1000	999	0
QS1-1K	1000	499500	0
QS4-1K	1000	4308	4308
MS-5K	5000	4999	0
QS1-5K	5000	12497500	0
QS4-5K	5000	27210	27210
MS-10K	10000	9999	0
QS1-10K	10000	49995000	0
QS4-10K	10000	59414	59414
MS-20K	20000	19999	0
QS1-20K	20000	199990000	0
QS4-20K	20000	128823	128823



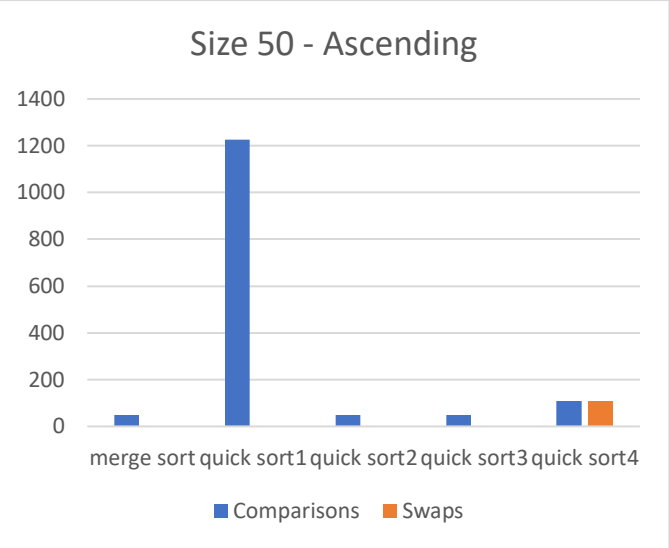
Randomly Ordered Data

Sort Type	File Size	Comparisons	Swaps
MS-500	500	36701	36559
QS1-500	500	4522	2360
QS4-500	500	1996	2054
MS-1K	1000	148316	148015
QS1-1K	1000	10094	5423
QS4-1K	1000	4379	4479
MS-2K	2000	594461	593905
QS1-2K	2000	24549	11961
QS4-2K	2000	10961	11205
MS-5K	5000	3596487	3595082
QS1-5K	5000	66092	36467
QS4-5K	5000	32241	32786
MS-10K	10000	14674072	14671053
QS1-10K	10000	151458	80899
QS4-10K	10000	63450	64574
MS-20K	20000	58740921	58735028
QS1-20K	20000	344444	160916
QS4-20K	20000	156144	158408



Reverse Ordered Data

Sort Type	File Size	Comparisons	Swaps
MS-500	500	498	498
QS1-500	500	124750	62500
QS4-500	500	12850	12949
MS-1K	1000	998	998
QS1-1K	1000	499500	250000
QS4-1K	1000	50700	50899
MS-2K	2000	1998	1998
QS1-2K	2000	1999000	1000000
QS4-2K	2000	201400	201799
MS-5K	5000	4998	4998
QS1-5K	5000	12497500	6250000
QS4-5K	5000	1253500	1254499
MS-10K	10000	9998	9998
QS1-10K	10000	49995000	25000000
QS4-10K	10000	5007000	5008999
MS-20K	20000	19998	19998
QS1-20K	20000	199990000	100000000
QS4-20K	20000	20014000	20017999

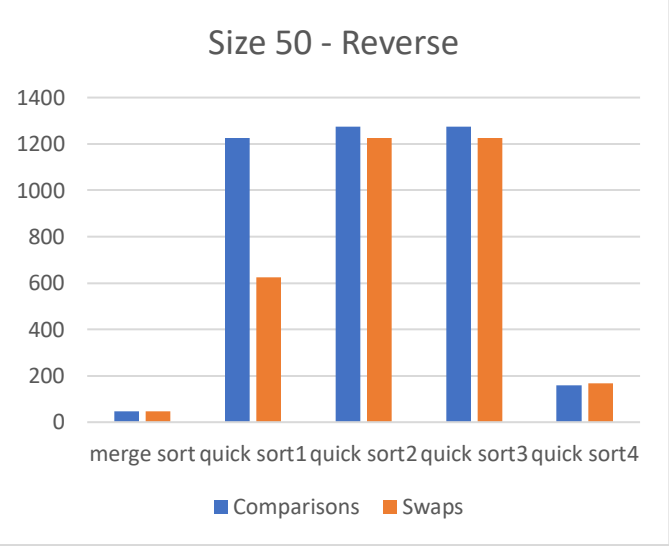


Ascending Ordered Data
File Size 50

Sort Type	Comparisons	Swaps
merge sort	49	0
quick sort1	1225	0
quick sort2	49	0
quick sort3	49	0
quick sort4	110	110

Randomly Ordered Data
File Size 50

Sort Type	Comparisons	Swaps
merge sort	424	405
quick sort1	214	99
quick sort2	681	632
quick sort3	681	632
quick sort4	104	109



Reverse Ordered Data
File Size 50

Sort Type	Comparisons	Swaps
merge sort	48	48
quick sort1	1225	625
quick sort2	1274	1225
quick sort3	1274	1225
quick sort4	160	169