Name Yang Zhang

Lab 4 Analysis

With the specifications of Lab 4, we are requested to examine the results of Quick Sort and Natural Merge sorting algorithms to gain an understanding of the factors that should be taken into account when choosing an acceptable sort.

There are four versions of Quick Sort, General Quick Sort(QS) with the first item of the partition as the pivot, Quick Sort & Insertion50(QSI50) with a partition of size 50 or less for Insertion Sort, Quick Sort & Insertion1000(QSI00) with a partition of size 100 or less for Insertion Sort,
Quick Sort & MedianOf3(QSIM3) with the median-of-three as the pivot, and Natural Merge Sort(NMS).

The natural merge sort is achieved by the linked implementation so that the space complexity can be reduced to O(1). Both Quick Sort and Natural Merge Sort have the same time complexity of O(NlogN), and Quick Sort' s space complexity is O(N). Therefore, Natural Merge Sort's space complexity of O(1) can be a factor that is taken into account when evaluating how it might affect the algorithm's effectiveness.

All five sorting algorithms are implemented using iteration since recursion, despite being more intuitive and straightforward to utilize in code, runs the risk of a stack overflow when the amount of data is excessive. Also, Natural Merge Sort identifies pre-sorted areas ("runs") in the input data and merges them. This prevents the unnecessary further dividing and merging of presorted subsequences. Since it is challenging to determine whether the pivot is already in order, which partitions the pivot belongs to, and the maximum length of each ordered sub-list, iterating the list at the beginning to find the sub-lists that are already in order appears to be much simpler than recursively splitting the list into two parts. Those are the main reasons to help justify choosing iteration instead of recursion.

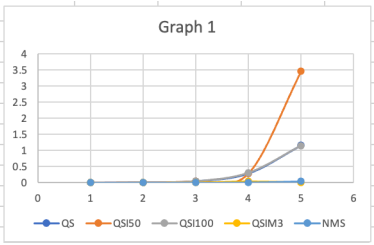
As requested, two separate folders are formed, one with non-duplicate values and the other with duplicate values that occupy 15-20%.

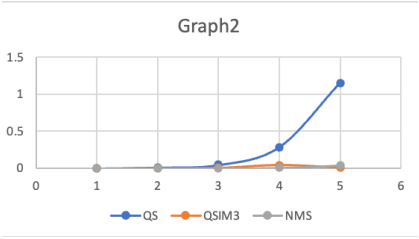
1	**rev50.txt**	**QS**	**1225**	**147**	**0.0001**
2	**rev50.txt**	QSI50	1226	1131	0.000101
3	**rev50.txt**	QSI100	1226	1131	9.6E-05
4	**rev50.txt**	QSIM3	351	74	5.9E-05
5	**rev50.txt**	NMS	332	133	9.2E-05
6	**asc1K.txt**	QS	499500	2997	0.017491
7	**asc1K.txt**	QSI50	498276	2847	0.017503
8	**asc1K.txt**	QSI100	494551	2697	0.016953
9	**asc1K.txt**	QSIM3	9009	1022	0.000582
10	**asc1K.txt**	NMS	999	0	6.3E-05
11	**ran50.txt**	QS	245	178	5E-05
12	**ran50.txt**	QSI50	420	349	5.1E-05
13	**ran50.txt**	QSI100	420	349	5E-05
14	**ran50.txt**	QSIM3	231	101	4.9E-05
15	**ran50.txt**	NMS	398	194	6.7E-05
16	**ran1K.txt**	QS	11449	5396	0.000773
17	**ran1K.txt**	QSI50	15859	10047	0.000913
18	**ran1K.txt**	QSI100	22525	17282	0.001328
19	**ran1K.txt**	QSIM3	9426	3061	0.000808
20	**ran1K.txt**	NMS	16214	8214	0.001438

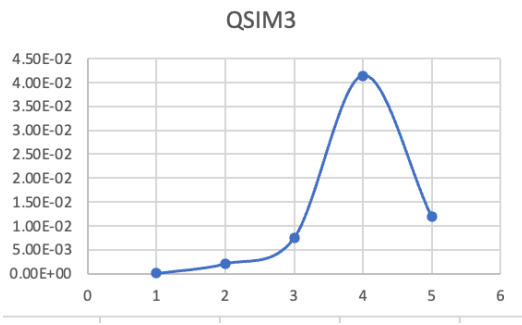
20	**ran1K.txt**	NMS	16214	8214	0.001438
21	**rev1K.txt**	QS	499500	2997	0.017165
22	**rev1K.txt**	QSI50	499501	4072	0.01691
23	**rev1K.txt**	QSI100	499501	7647	0.016952
24	**rev1K.txt**	QSIM3	125751	1499	0.004797
25	**rev1K.txt**	NMS	12932	4932	0.001574
26	**asc50.txt**	QS	1225	147	7.7E-05
27	**asc50.txt**	QSI50	98	3	3.1E-05
28	**asc50.txt**	QSI100	98	3	3.1E-05
29	**asc50.txt**	QSIM3	255	62	5E-05
30	**asc50.txt**	NMS	49	0	2.3E-05
31	**rev10K.txt**	QS	49995000	29997	1.719034
32	**rev10K.txt**	QSI50	49995001	31072	1.722158
33	**rev10K.txt**	QSI100	49995001	34647	1.721779
34	**rev10K.txt**	QSIM3	12507501	14999	0.462484
35	**rev10K.txt**	NMS	184604	64608	0.019398
36	**asc5K.txt**	QS	12497500	14997	0.431142
37	**asc5K.txt**	QSI50	12496276	14847	0.42624
38	**asc5K.txt**	QSI100	12492551	14697	0.427556
39	**asc5K.txt**	QSIM3	57726	5904	0.003176
40	**asc5K.txt**	NMS	4999	0	0.000252

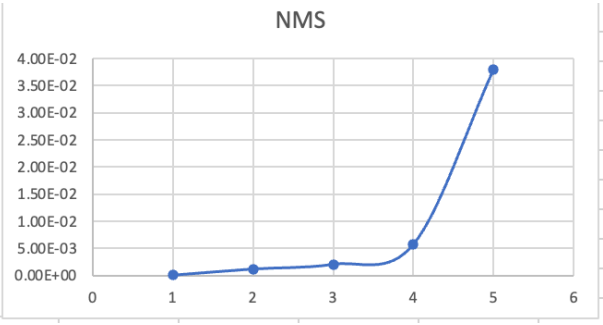
The part of outcome from sorting is displayed above. The number of companions and exchanges are shown in the third and fourth columns, respectively, along with the sorting algorithms in the second column. The duration of each algorithm's execution is shown in the final column. Based on the programs, it is already verified that each method sorts the list correctly. From the graphs, the number of comparisons and time period rise roughly in proportion to the size of the file. Additionally, for every dataset in ordinarily ascending order, the number of exchanges in the graphs for NMS is always zero, which would be another proof that the coding is sound.

Size/Sort	50	1k	2k	5k	10k
QS	0.000075666	0.01181	0.046134	0.287126	1.154931
QSI50	0.000061	0.01177533	0.04682267	0.284854	3.45317
QSI100	5.90E-05	0.011744	0.046361	0.30665067	1.149077
QSIM3	5.27E-05	0.002062	0.007474	0.04141933	0.011905
NMS	6.10E-05	0.001198	0.002066	0.005756	0.038057









When our data is relatively large, Insertion Sort is not appropriate because of its O(N^2) time complexity. This behavior can also be observed by comparing QSI50 and QSI100 with QSIM3 and NM3 sorting algorithms shown in Graph 1.

The time complexity of Insertion Sort is O(N^2). In graph1, QSI50 and QSI100 both substantially increase when the size of the file is large, specially at point 4(5k). This is reasonable because other sorting algorithms have the time complexity of O(NlogN). N^2 is greater then NlogN, and it makes more prominence as the file size increases. Starting at the dataset in size of 5k, Quick Sort plus Insertion sort for a partition of size 100 or less seems better based on the time span compared with QSI50, which would be that having too many partitions may consume much more extra time.

It can be shown in Graph1 that the efficiency of GQS and QSI00 is comparable when the volume of data is less than 10k. However, It is believed that GQS will perform better than QS100 as the volume of data grows on this basis.

Because QSIM3 and NMS still have a very short running time for large-size files, they perform significantly better than the others. No matter the case is the best or worst, the time complexity of running Merge Sort is O(NlogN). In the worst case, when the pivot is chosen poorly, QS can take

O(n^2) time complexity. Median-of-three quicksort addresses this issue by selecting the pivot as the median of three elements: the first, the middle, and the last element of the array. QSIM3 ensures that the pivot is closer to the actual median of the array, reducing the likelihood of choosing a poor pivot and improving the overall efficiency of the algorithm. Therefore, QSIM3 becomes able to compete with NMS.

It should be also noticed that QSIM3 reaches its highest position at point 4 before declining. If there is no errors happened, it is an interesting part that could be explored further next time. Is that because choosing the appropriate pivot in a higher volume of data could save a lot of time? Does it take too long to split data into multiple partitions and merge them?

Non-duplicate					
Comarisons					
Size/sort	50	1k	2k	5k	10k
QS	4166323.3333	336816.33333	1341370	8355574.3333	33379030
QSI50	581.33333333	337878.66666	1344149.6666	8363039.6666	33394303
QSI100	581.33333333	338859	1347343.6666	8994701	33417889.333
QSIM3	279	48062	181679.66666	1083218	4256353.6666
NMS	259.66666666	10048.333333	22433	66370.333333	144519.33333
Exchanges					
Size/sort	50	1k	2k	5k	10k
QS	157.33333333	3796.6666666	7920.6666666	20826.333333	43321.666666
QSI50	494.33333333	5655.3333333	11549.333333	29374.666666	59849.666666
QSI100	494.33333333	9208.6666666	17616	44051	88445.333333
QSIM3	79	1860.6666666	3802.6666666	10337	21548
NMS	109	4382	9773.6666666	28038.666666	61158.333333

Duplicate					
Comarisons					
Size/sort	50	1k	2k	5k	10k
QS	4166335.3333333	337431	1340669.6666666	8354779.6666666	33380746.33333
QSI50	632	338585	1343404.6666666	8361590.3333333	33395504
QSI100	632	339923.3333333	1347224.6666666	8994696	33417992
QSIM3	290.66666666666	45387	168263.6666666	998153	3983441.6666666
NMS	265.66666666666	10099.666666666	22517.666666666	67020	145243
Exchanges					
Size/sort	50	1k	2k	5k	10k
QS	154.33333333333	3768.6666666666	7948	20862.666666666	43266.666666666
QSI50	539.66666666666	5688.6666666666	11541.666666666	28708.333333333	59319.333333333
QSI100	539.66666666666	9678.6666666666	18347.666666666	43916.666666666	86826.666666666
QSIM3	81.333333333333	1921	4018.6666666666	10597.333333333	21960
NMS	109.66666666666	4315	9613.3333333333	28084.666666666	60943.666666666

The comparison & exchange tables above can be also an evident of proving QSIM3 and NMS are much better, because they spend much less number of both comparisons and exchanges no matter duplicate values are contained or not. In more detail, it seems that QSIM3 is better than NMS, because QSIM3 has the least number of both comparisons and exchanges. However, as the size of the data increases, NM3 runs more slowly and performs fewer comparisons. That is possible because a very large amount of data that is in reversed or ascending order has a significant impact on the running time.

ran50.txt	QS	245	178	5E-05
asc50.txt	QS	1225	147	7.7E-05
rev50.txt	QS	1225	147	0.0001
ran1K.txt	QS	11449	5396	0.000773
rev1K.txt	QS	499500	2997	0.017165
asc1K.txt	QS	499500	2997	0.017491

ran2K.txt	QS	26110	11768	0.001629
rev2K.txt	QS	1999000	5997	0.067926
asc2K.txt	QS	1999000	5997	0.068847
ran5K.txt	QS	71723	32485	0.004435
rev5K.txt	QS	12497500	14997	0.425802
asc5K.txt	QS	12497500	14997	0.431142

ran10K.txt	QS	147090	69971	0.008825
rev10K.txt	QS	49995000	29997	1.719034
asc10K.txt	QS	49995000	29997	1.736935

When the input is sorted or reversed, Quick Sort is slow but otherwise works well. The tables above showing 5 sizes of files in random, reverse, and ascending order can prove it. As a result, the initial data that has been sorted has a significant impact on the efficiency of Quick Sort. In addition, it makes little difference whether the original data is ascending or descending.

The space complexity of QS, QSI50, QSI100, QSIM3 should be O(N), because there is need to request an additional space to store temporary value for later sub-sorting. The space of complexity of NMS is

O(1), because a linked node list is created at the beginning for storing raw data so that exchanges can be achieved by doing node link or unlink without any extra space to store values.

Quick Sort is a comparison-based sorting algorithm that is known for its simplicity and efficiency. It has an average-case time complexity of O(n log n), but in the worst case, it can take O(n^2) time complexity. The algorithm selects a pivot element and partitions the array into two sub-arrays: one with elements less than the pivot and another with elements greater than the pivot. This process is repeated recursively until the entire array is sorted. The choice of pivot element can significantly affect the performance of Quick Sort. In the best case, the pivot element is the median of the array, and Quick Sort has an optimal time complexity of O(n log n). However, if the pivot element is poorly chosen, the algorithm can take O(n^2) time complexity.

On the other hand, Natural Merge Sort is a stable, comparison-based sorting algorithm that is known for its adaptability. It has an average-case time complexity of O(n log n) same with its worst-case time complexity. The algorithm works by identifying and merging already sorted sub-arrays in the input array. It does not require a pivot element like Quick Sort and thus avoids the worst-case scenario where the pivot element is poorly chosen. Natural Merge Sort divides the input array into runs, which are either increasing or decreasing sequences of elements. The algorithm then merges these runs until only one run remains, which is the sorted array.

When comparing the number of comparisons and swaps performed by Quick Sort and Natural Merge Sort, it is important to consider their best-case, average-case, and worst-case scenarios. In the best-case scenario, both algorithms perform roughly the same number of comparisons and swaps. However, in the worst-case scenario, Quick Sort performs more comparisons and swaps than Natural Merge Sort.

The worst-case scenario for Quick Sort occurs when the pivot element is either the smallest or largest element in the array. In this case, the partitioning process only reduces the size of the array by one element, resulting in n recursive calls with only one element each. This leads to O(n^2) time complexity and a large number of comparisons and swaps. In contrast, Natural Merge Sort's worst-case scenario occurs when the input array is in reverse order, leading to a large number of runs and merges. However, this worst-case scenario still has a time complexity of O(n log n), which is much better than Quick Sort's O(n^2) worst-case time complexity.

In the average-case scenario, both Quick Sort and Natural Merge Sort perform roughly the same number of comparisons and swaps. However, the number of swaps performed by Quick Sort can be slightly higher than that of Natural Merge Sort, especially when the input array contains many duplicates. This is because Quick Sort partitions the array based on the pivot element, which can result in many swaps of elements with the same value.

In conclusion, both Quick Sort and Natural Merge Sort are efficient and widely used sorting algorithms. Quick Sort is faster than Natural Merge Sort in the average case but can be slower in the worst case. Natural Merge Sort is more adaptable and has a guaranteed worst-case time complexity of O(n log n).

Because Quick Sort experienced its worst-case scenario, QSIM3 was developed and is capable of performing on par with Natural Merge Sort. QSIM3 is a preferable option if the data is small and in random order. When the size of the data increases significantly, Natural Merge Sort usually proves to be more effective.While the number of comparisons and swaps performed by these algorithms may differ in different scenarios, they all provide good performance for sorting large datasets.