LAB 4 ANALYSIS REPORT

by Chih Yuan Chan ID: 28A8CA

Date: 05/04/2023

1. PROBLEM DEFINITION

1.1 General approach

This project compares the algorithmic efficiency of the Quick Sort (hereby referred to as QS) algorithm Vs Natural Merge Sort (NMS) for a series of input numbers organized as follows:

- Random ordered data
- Reverse ordered data
- Ascending ordered data
- Random ordered data with 20% duplication

The assignment calls for the analysis of 5 different sizes of integer input: 50, 1000, 2000, 5000 and 10000.

However, due to the recursive implementation of my solution for both QS and NMS and the limitations of the available RAM on my Laptop computer, I was unable to run the 5000 and 10000 sample sizes.

To address the loss of the 5000 and 100000 sample sizes I have replaced these with alternative smaller sample sizes of 500 and 3000.

To aid the conciseness of the descriptions of each algorithmic run, a shorter acronym form to represent each run is made, using the following convention:

Nomenclature:

- The first three characters of each run represent the format of the data (per above descriptions) being processed:
  - "**ran**" represents random ordered data
  - "**rev**" represents reverse ordered data
  - "**asc**" represents ascending ordered data
  - "**dup**" represents random ordered data with duplication (max 20%)
- The remainder characters represent the format of the sort implementation:
  - "**QS**" represents Quick Sort
    - **QS1_0**: Quick Sort with Type 1 (First index position) pivot, no Insertion Sort
    - **QS1_50**: Quick Sort with Type 1 pivot, Insertion Sort activated on partition sizes of 50 and smaller
    - **QS1_100**: Ditto above, for partition sizes 100 and smaller
    - **QS2_0**: Quick Sort with Type 2 (median of three) pivot, no Insertion Sort
    - **QS3_0**: Quick Sort with Type 3 (middle index) pivot, no Insertion Sort
  - "**NMS**" represents Natural Merge Sort

Some examples of usage of the above convention:

- "**asc2k_QS1_0**" represents Data of size 2000 arranged in ascending order, sorted using Quick Sort with 1st index pivot and no Insertion Sort activation
- "**dup50_NMS**" represents Data of size 50 ordered randomly with potential for up to 20% duplication.

1.2 Project Enhancements

The following enhancements were incorporated:

- Comprehensive output reporting for size 50 files including highly detailed "play by play" reporting on NMS formation of sequence runs, merging and partitioning, Quicksort partitioning staging and activation of Insertion sort.
- Analysis for additional random ordered number sets with 20% duplicates
- Additional alternative Quicksort pivot using middle index
- The above additions increased the number of output runs to 120
- Modules to produce automated production of Pandas Dataframes for summary reports or quick plotting of graphs
- Standalone number generator script to generate any size of ordered and un-ordered data

1.3 Justification of data structures:

1. <u>Quick Sort</u>

A dynamic array (Python List) was used to implement Quicksort recursively.

Justification:

The dynamic array permits the in-place sorting of the Quick Sort algorithm without incurring extra overhead for external sorting (**O(1)** space complexity). The array allows for **O(1)** Time Performance random accessing to retrieve and compare the values of integers using indexing, as was used for all Comparison and Swapping operations.

An array was chosen for the following reasons:

- Simple implementation: The array implementation of Quick Sort is straightforward to understand and implement using the python list.
- Cache-friendly: Arrays have good cache locality, which can lead to faster sorting performance on the CPU.
- In-place sorting: The array implementation of Quick Sort sorts the elements in place, which means it requires no additional memory other than the array being sorted.

The disadvantage of the dynamic array is for insertion operations at intermediate locations or the start of the array, which would occur when repositioning the pivot in the first position from another location (such as for median-of-three or middle indexing). In such an instance, there will be re-arranging of the data values within the array and is close to **O(N)** in the worst case, if a pivot is chosen towards the end of the array and then relocated to the first index for convenience of partitioning. Insertion at the end of the array would still be **O(1)** but this is less relevant for the algorithm, since it does not involve insertion at the end.

As noted previously, the primary operation is swapping, which is performed in place so the array does not require re-arrangement during swapping. However, the penalty incurred is the use of an extra temp variable to store one of the swapping elements, which increases the work and space requirements per swap.

Considering both the advantages and disadvantages, I have chosen the array implementation to take advantage of its speed using random access indexing since the primary operation in the sort is swapping.

There would also be potentially some wastage of dynamically allocated space depending on the number of stored elements.

2. Insertion Sort

A dynamic array (Python List) was used to implement Insertion Sort recursively.

Justification:

As for Quick Sort, the swapping operations and their speed is crucial so the array with random access indexing is ideal, per the considerations discussed above.

3. Natural Merge Sort

A Linked List implementation was chosen to implement Natural Merge Sort.

Justification:

This was determined to be the best option because it permits a reduction in the external storage overhead of conventional Merge Sort (**2N**) to just **N**, because the only storage is for the nodes of the linked-list, and merging does not require additional storage because the pointers are just re-arranged to permit in-place re-organization.

The linked list is interconnected node to node so this also permits the ease of the comparisons taking place to determine sequential runs as the first step of NMS.

A Queue ADT was chosen to implement a linked list of linked lists. The Queue ADT is highly suited to implement the balanced binary tree approach which ensures the merging of linked lists is performed efficiently (in pairs). After the sequential runs are determined, these are fed into the Queue and then dequeued in pairs (popped), merged together and then enqueued back into the Queue. As can be seen the FIFO nature of the Queue is well suited for this operation.

1.3.1 Recursion Vs Iteration

I chose the recursive approach for the coding because both QS and NMS are highly recursive in nature, particularly the partitioning aspects of QS (base case is arrived at after only 1 single unit remains) and the merging aspects of NMS, hence the recursive approach was intuitive and resulted in much simpler, elegant code.

In particular, the recursive implementation highly suits the linked lists and tree implementation of enqueueing and de-queueing using the Queue ADT when merging.

Notwithstanding, I underestimated how deep the recursive stack would be for the larger sample sizes and would have re-implemented the alternative iterative approaches if time had permitted.

In principle, the iterative implementation would likely also be faster for large inputs since they avoid the overhead of function calls and stack manipulation.

Tail recursion is also not available in Python but several aspects of the algorithms such as the determination of sequential runs for NMS are last-entry function calls, which means that they can be easily replaced with an iterative approach.

In summary, the recursive implementation made the code and algorithm more understandable but this requires additional memory overhead to maintain the recursive stack calls. In my case this resulted in stack overflow issues.

2. RESULTS ANALYSIS

The results will be discussed in 3 sections.

In the first section, theoretical expectations will be discussed to predict the expected output from this exercise.

Next, the total Sorting Steps (instead of division into separate counts of comparisons and swaps) will be compared against the theoretical expectations..

Lastly, a review will be made of the separate comparison and swap counts on each of the runs.

2.1 Theoretical Expectations

2.1.1 Insertion Sort

Insertion Sort (IS) algorithm will be the most effective for smaller sizes of data that have ordered or partially ordered sequences. This is because it partial sequences permit IS to make one comparison per pass through without executing any swapping or movement of the elements. The higher the level of order, the better is the performance of Insertion Sort.

Time Complexity Considerations:

- Best case time performance: **O(N)** – this occurs when the list is already fully sorted.
- Worst case time performance: **$O(N^2)$** – this occurs when the list is sorted in reverse order, in which case a comparison at each location also requires multiple swap / shifting of elements to take place. The number of shifts and comparisons for a reverse ordered array is the same ($N^2/2$), so the total work is N^2, hence **$O(N^2)$**.
- Average performance: **$N^2/2$ –** average performance occurs between fully sorted and sorted in reverse, hence approximately $N^2/2$. This will occur most of the time in a randomly ordered set of numbers since at least some partial ordering is expected. This is still $O(N^2)$ but the half coefficient is significant in practice to give Insertion Sort reasonable performance for smaller element sets.

Space Complexity Considerations:

- Insertion Sort sorts elements in place without any need for external space, it's space complexity is therefore **O(1)**.

2.1.2 Quick Sort

Time Complexity Considerations:

- Best case performance: **O(N log N)** – This occurs when the pivot divides the partitions evenly into equal halves since this produces the effect of a balanced tree. Halving occurs at each "level" of the "tree" and a partition is performed on all the subarrays which add up to N, hence N log N.
- Worst case performance: $\mathbf{N^2/2}$ – This occurs when the partition is to one side of the pivot, in which case there are $N^2/2$ steps giving rise to $O(N^2)$.
- Average case: O**(N log N)** – The pivot divides the partitions somewhere between half and to one side, but is still bounded by O(N log N).

Space Complexity Considerations:

- Quick Sort sorts elements in place without any need for external space, it's space complexity is therefore **O(1)**.

2.1.3 Quick Sort with Insertion Sort

When Quick Sort utilizes Insertion Sort, the only advantage is when there is a higher level of sequential order to the set since best-case performance is O(N) vs O(N log N) from Quick Sort. However, for random ordered data sets, we expect Quick Sort to be faster since it is O(N log N) on average vs $O(N^2)$ from Insertion Sort.

If the data set is very small, the effect of the Insertion Sort will be larger. If the set is in ascending order, we will expect a very large performance boost. Conversely, if the data set is randomly ordered or reverse ordered, we expect Insertion Sort to slow down the Quicksort algorithm, the degree depending on the proportion of the partition size sorted by IS vs the overall size of the data set.

2.1.4 Natural Merge Sort

Time Complexity Considerations:

- Best case performance: **O(N) –** The array is full sorted - only a single pass is made to get one sequential run for the whole list and no merging is done.
- Worst case performance: O**(N log N)** – The array is sorted in reverse order. In which case this becomes like regular merge sort, without conferring any advantage.
- Average case: O**(N log N)** – The pivot divides the partitions somewhere between half and to one side, but there is still some order to be exploited hence is still bounded by O(N log N) on average.

Space Complexity Considerations:

The recursive tree has a height of $\log_2 N$ and assuming we require a dummy head node at each level to store the sorted list, then the space complexity would be $O(\log_2 N)$

2.1.5 Natural Merge Sort vs Conventional Merge Sort

As discussed above, the advantage of Natural Merge Sort is only when the array is substantially ordered, in which case the performance would be O(N) (linear time) for a fully ordered file, otherwise the amount of work done is about the same as conventional merge sort.

Natural Merge Sort implemented using linked lists can have a reduction in the space overhead O(log N) (per above) vs O(N) for conventional merge sort.

2.1.6 Quicksort “Swaps” vs Natural Merge Sort “Exchanges”

An advantage that the Natural Merge Sort has over Quicksort in terms of work done in repositioning data elements is that NMS does this by rearranging pointers, not the data position itself. Hence, NMS exchanges do not incur extra space beyond the need for the pointers. With QS a temp variable is required to “hold” the value of one element while swapping with the other so this is a space penalty.

2.2 Total Step Counts – Results and Observations

2.2.1 Case 1: Random Ordered Data – No duplicates

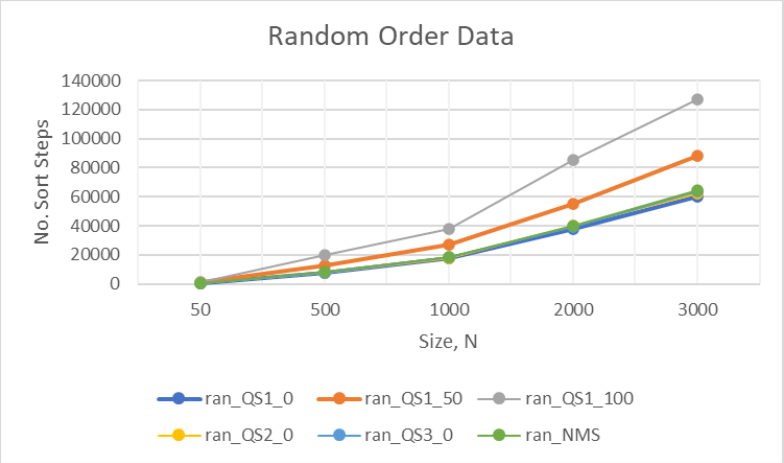

Fig 1. Random ordered data (asymptotic curves omitted)

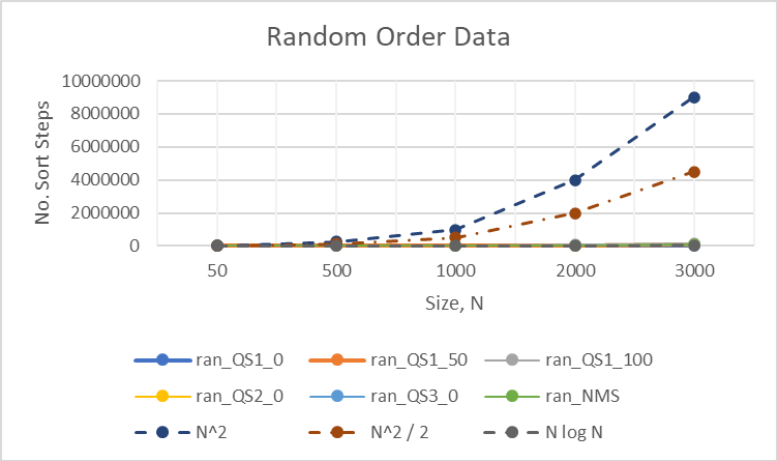

Fig 2. Random ordered data (asymptotic curves superimposed)

	SIZE, N				
SORT/ FILE TYPES	**50**	**500**	**1000**	**2000**	**3000**
ran_QS1_0	465	7598	17763	37958	60310
ran_QS1_50	1286	12494	26707	55449	88497
ran_QS1_100	1286	19956	38045	85392	126826
ran_QS2_0	487	7969	17852	39674	62247
ran_QS3_0	507	7816	18120	39578	63863
ran_NMS	478	8052	18272	40364	64423
N^2	2500	250000	1000000	4000000	9000000
N^2 / 2	1250	125000	500000	2000000	4500000
N log N	282	4483	9966	21932	34652

Fig 3. Tabled combined comparison and swap counts

Observations – Random Ordered Data:

- QS1_50 and QS1_100 results (Insertion Sort with minimum 50 and 100 size limits, respectively) are identical because the file size is 50 only, so both files are entirely sorted using Insertion Sort instead of Quick Sort. The number of sort steps is almost perfectly correlating with $N^2/2$ since the data is randomly ordered i.e. closer to average case.
- QS1_50 performs better than QS1_100 with increasing file size because the slower performance of Insertion Sort becomes less dominant given it kicks in at a lower threshold limit compared to QS1_100 (100 limit).
- QS1_0 (Quick Sort only – no Insertion Sort) performs significantly better to QS1_50 and QS1_100 because QS has N log N performance for average case vs $N^2/2$ for IS. N log n for size 50 is circa 300, so result of 465 is reasonably close to the theoretical prediction.
- The use of the First Pivot for QS is not that much of a disadvantage from the Median of Three or Middle Index since this is a randomly ordered file so the first position can still result in the partitions being reasonably evenly divided. We see that across each size file set, the results using each of the three different pivots is similar and is close to N log N theoretical QS average performance.
- Natural Merge Sort is performing at N log N so the performance is similar to QS.

2.2.2 Case 2: Reverse Ordered Data

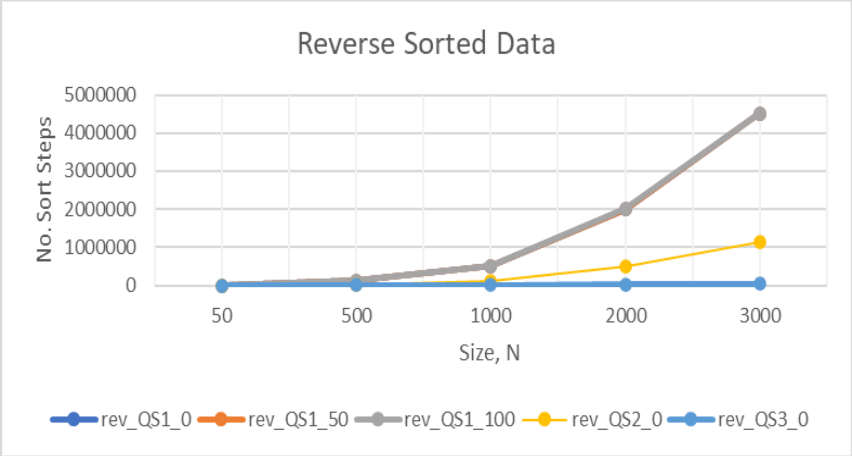

Fig 4. Reverse ordered data (asymptotic curves omitted)

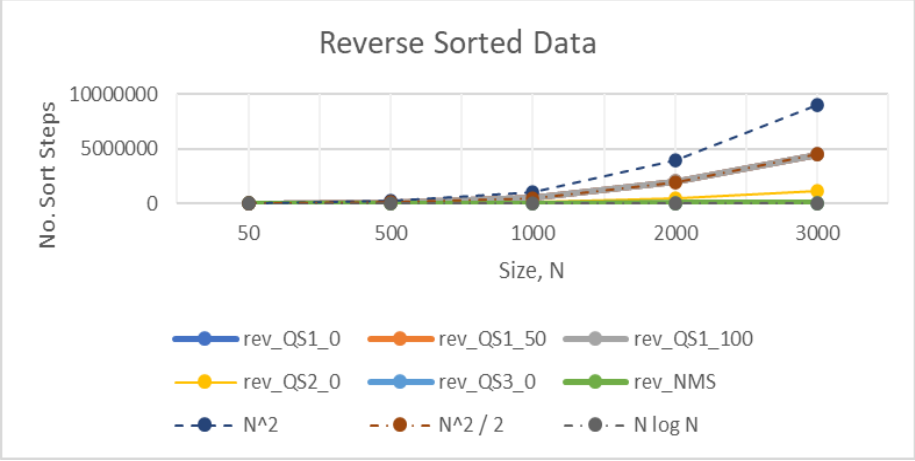

Fig 5. Reverse ordered data (asymptotic curves superimposed)

	SIZE, N				
SORT/ FILE TYPES	**50**	**500**	**1000**	**2000**	**3000**
rev_QS1_0	1323	125748	501498	2002998	4504498
rev_QS1_50	2450	126875	502625	2004125	4505625
rev_QS1_100	2450	130500	506250	2007750	4509250
rev_QS2_0	592	34117	130742	511492	1142242
rev_QS3_0	390	5469	11939	25876	40994
rev_NMS	504	7303	16107	35215	55275
N^2	2500	250000	1000000	4000000	9000000
N^2 / 2	1250	125000	500000	2000000	4500000
N log N	282	4483	9966	21932	34652

Fig 6. Reverse ordered data

Observations – Reverse ordered data:

- QS1_50 and QS1_100 perform poorest because Insertion Sort is activated and has $O(N^2)$ performance on a reverse ordered set. QS1_0 is better than QS1_50 and QS1_100 because Insertion Sort is deactivated so it is fully Quicksort and the time complexity is $N^2/2$ vs N^2 for Insertion Sort. At increasing data sizes, the effects of the small partition sizes activating IS are relatively insignificant, hence we expect $N^2/2$ performance as the data sets increase, which is demonstrated in the results.

- QS2_0 (median of three pivot) performs similar to QS3_0 (middle index) at smaller data sizes but starts to lose out as the data size increases. This is because the middle index will always pick the value that splits the data set evenly into half for a sorted array (ascending or reversed). The middle of three pivot will try to achieve this but the small differences in the balance of partition sizes will amplify as the data increases.

- Natural Merge Sort (rev_NMS) is performing close to N log N throughout so the performance is consistently good for Reverse ordered data. However, it is slightly slower than QS3_0 (QS with middle index pivot) because it is performing excellent balancing of partition sizes with its precise pivot value.

2.2.3 Case 3: Ascending Ordered Data

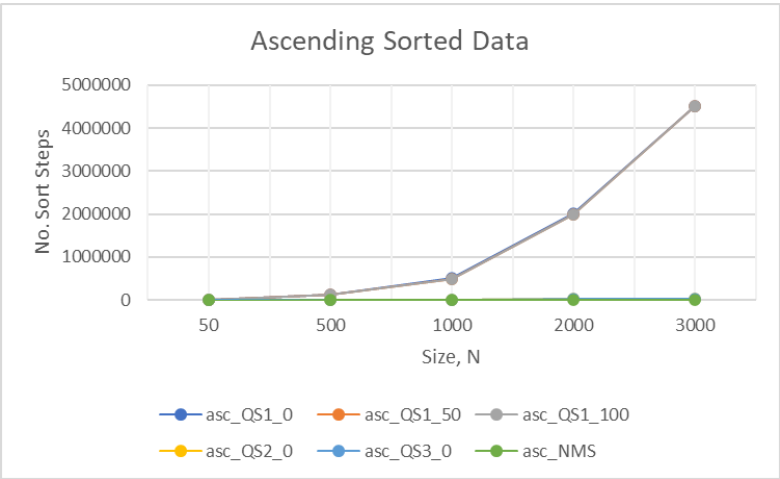

Fig 8. Ascending ordered data (asymptotic curves omitted)

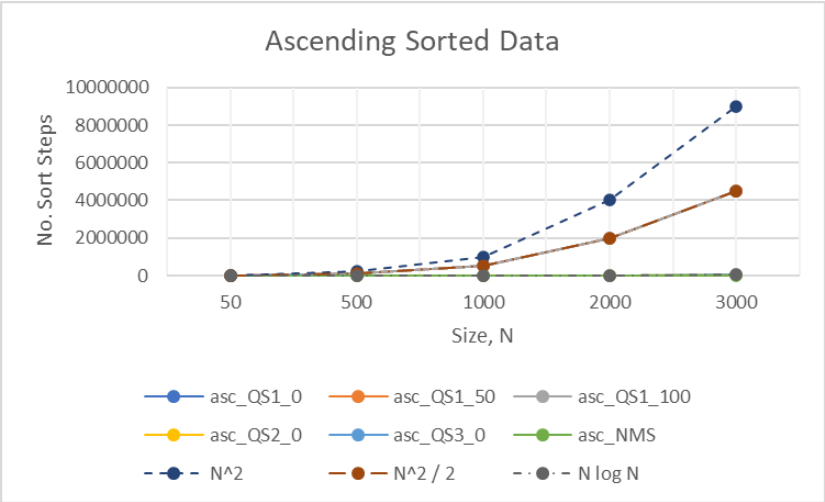

Fig 9. Ascending ordered data (asymptotic curves omitted)

	SIZE, N				
SORT/ FILE TYPES	**50**	**500**	**1000**	**2000**	**3000**
asc_QS1_0	1323	125748	501498	2002998	4504498
asc_QS1_50	49	124474	500224	2001724	4503224
asc_QS1_100	49	120699	496449	1997949	4499449
asc_QS2_0	317	4518	10031	22056	36821
asc_QS3_0	286	4263	9520	21033	34845
asc_NMS	99	999	1999	3999	5999
N^2	2500	250000	1000000	4000000	9000000
N^2 / 2	1250	125000	500000	2000000	4500000
N log N	282	4483	9966	21932	34652

Fig 10. Ascending ordered data

Observations – Ascending ordered data:

- QS1_0 is the worst performer at both the smaller and larger range of data sets. This is because the pivot is at the first position and ends up having the partition consistently skewed to one side (maximum comparisons and swaps).

- QS1_50 and QS1_100 are the best performers (linear time) at size 50 because Insertion Sort is done for the entire set and there are zero swaps, only N comparisons, when the list is sorted. As the size of the data increases, the 1st position pivot from QS starts to dominate with it's poor performance.

- Natural Merge Sort (asc_NMS) is performing close to N log N throughout and is easily the best performer from size 500 onwards. This is to be expected because for a fully sorted list, NMS obtains just one single run (linear) and no merging is performed. However, in the case for this report, pushing (enqueueing) the single linked list into the queue and popping it from the queue is considered equivalent to a merge operation since the definition of what a merge is here is uncertain, hence the breakdown in Section 2.3 below shows the merging as equal to the number of comparisons.

2.2.4 Case 4: Random Ordered Data – 20% duplicates

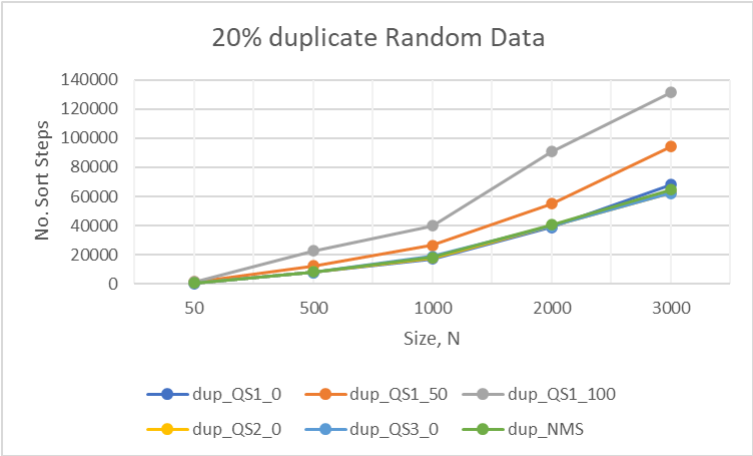

Fig 11. Random ordered data with duplicates – no asymptotic curves

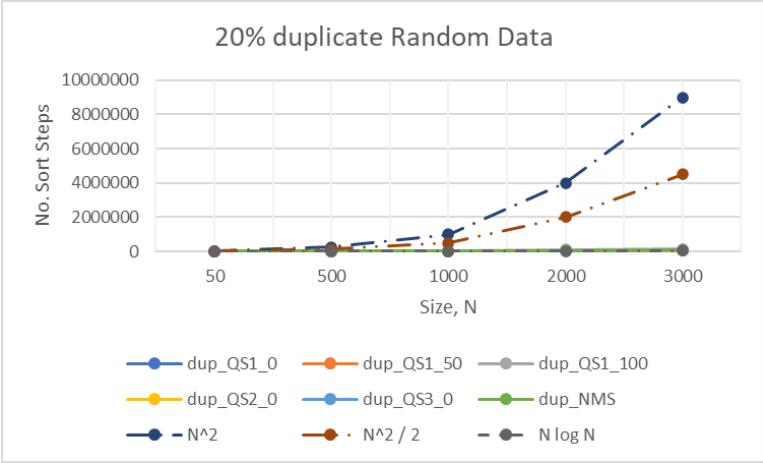

Fig 12. Random ordered data with duplicates – asymptotic curves superimposed

	SIZE, N				
SORT/ FILE TYPES	**50**	**500**	**1000**	**2000**	**3000**
dup_QS1_0	414	7716	17194	38712	67906
dup_QS1_50	1341	12275	26435	55123	94134
dup_QS1_100	1341	22741	39904	90801	131451
dup_QS2_0	507	8182	17910	39681	62458
dup_QS3_0	532	7728	18688	39576	62208
dup_NMS	500	8043	18208	40400	64570
N^2	2500	250000	1000000	4000000	9000000
N^2 / 2	1250	125000	500000	2000000	4500000
N log N	282	4483	9966	21932	34652

Fig 12. Random ordered data with duplicates

Observations – Ascending ordered data:

- QS1_0, QS2_0, QS3_0 and NMS all perform well and at N log N performance. This is not unexpected since both Quicksort and Natural Merge Sort have N log N average case performance.
- Insertion Sort is the worst performer so the QS + IS runs are dragged down by it's $N^2/2$ speed. More data is run by QS1_100 vs QS1_50 so it's performance is the poorest as the data grows in size. It is apparent that 20% duplicates does not confer any advantage to the any pre-ordering of data by Insertion Sort since the duplicates may not occur in sequence frequently enough to confer some effect of sequential ordering.

2.3 Comparison Vs Swaps

Comparisons and swaps on size 50 files:

	Compare	Swaps
ran50_QS1_0	406	59
ran50_QS1_50	643	643
ran50_QS1_100	643	643
ran50_QS2_0	391	96
ran50_QS3_0	409	98
ran50_NMS	249	229
rev50_QS1_0	1298	25
rev50_QS1_50	1225	1225
rev50_QS1_100	1225	1225
rev50_QS2_0	518	74
rev50_QS3_0	307	83
rev50_NMS	218	286
asc50_QS1_0	1323	0
asc50_QS1_50	49	0
asc50_QS1_100	49	0
asc50_QS2_0	268	49
asc50_QS3_0	237	49
asc50_NMS	49	50
dup50_QS1_0	412	58
dup50_QS1_50	486	483
dup50_QS1_100	486	483
dup50_QS2_0	391	91
dup50_QS3_0	389	92
dup50_NMS	253	237

Comparisons and swaps on size 500 files:

	Compare	Swaps
ran500_QS1_0	6586	1012
ran500_QS1_50	7814	4680
ran500_QS1_100	11144	8812
ran500_QS2_0	6617	1352
ran500_QS3_0	6496	1320
ran500_NMS	4061	3991
rev500_QS1_0	125498	250
rev500_QS1_50	125425	1450
rev500_QS1_100	125350	5150
rev500_QS2_0	33368	749
rev500_QS3_0	4621	848
rev500_NMS	2815	4488
asc500_QS1_0	125748	0
asc500_QS1_50	124474	0
asc500_QS1_100	120699	0
asc500_QS2_0	4019	499
asc500_QS3_0	3764	499
asc500_NMS	499	500
dup500_QS1_0	6463	981
dup500_QS1_50	7905	5132
dup500_QS1_100	10643	8412
dup500_QS2_0	7061	1287
dup500_QS3_0	6440	1321
dup500_NMS	4074	3982

Fig. 13 Size 50 and size 500 files

Comparisons and swaps on size 1000 files:

	Compare	Swaps
ran1k_QS1_0	15564	2199
ran1k_QS1_50	17498	9209
ran1k_QS1_100	22476	15569
ran1k_QS2_0	14935	2917
ran1k_QS3_0	15197	2923
ran1k_NMS	9254	9018
rev1k_QS1_0	500998	500
rev1k_QS1_50	500925	1700
rev1k_QS1_100	500850	5400
rev1k_QS2_0	129243	1499
rev1k_QS3_0	10241	1698
rev1k_NMS	6131	9976
asc1k_QS1_0	501498	0
asc1k_QS1_50	500224	0
asc1k_QS1_100	496449	0
asc1k_QS2_0	9032	999
asc1k_QS3_0	8521	999
asc1k_NMS	999	1000
dup1k_QS1_0	14623	2180
dup1k_QS1_50	16146	8676
dup1k_QS1_100	22662	17033
dup1k_QS2_0	15153	2854
dup1k_QS3_0	15980	2747
dup1k_NMS	9218	8969

Fig. 14 Size 1000 files

Observations:

- In the worst case of reversed ordered data, the number of comparisons to shifts (both $N^2/2$) is the same for Insertion Sort, so the smaller data sets with Insertion Sort kicking in have roughly equal comparisons to swaps. The differences start to diverge as the data size increases since QS becomes dominant.
- Swaps are always zero for Insertion Sort and Quicksort with the data in perfect ascending order.
- The number of comparisons and swaps is roughly equal for Natural Merge Sort, regardless of the data size or configuration.

3. LESSONS LEARNT

- There is no single one size fits all Sort. All sorts have situations where they have advantages and where they are simply inappropriate.
- Size of data set and ordering of the data are all important and must be understood well before determining a sort algorithm to use.
- In real life purely sorted data is unlikely and is more close to the random cases. In this situation, Big Theta (average case) performance can be more important than Big O performance, particularly if the worst case scenario is remote.
- Sorts like Insertion Sort with “poorer” algorithmic efficiency may in some cases still be preferred to better performing algorithms if the data set is small and implementation must be fast and single since they are very low in complexity. For smaller data sizes,

Insertion Sort can perform reasonably well provided the proportion of data has some partial sequencing.

- A very good general all-purpose high performing sort is Merge Sort since it gives consistent N log N performance across worst, best and average cases. The only downside is the penalty of extra space to hold the merged elements.
- A linked list is implementation helpful to reduce space overhead for the Merge Sort versus the array since only a dummy positioning node is required at all levels of the recursive tree.
- Recursive functions called at the end should be converted to Iterative approaches whenever possible or using Tail recursion.

4. FUTURE CONSIDERATIONS

- If I had more time I would try to change the algorithms into an iterative one and compare the performance to the recursive one.
- Next time I will make sure to perform some sample runs if the algorithm is recursive to check memory performance.
- I would like to time the difference in performance between the different Quick Sort and Natural Merge Sort implementations.