EN.605.202 Introduction to Data Structures (Spring 2022)

Mukul Sherekar

Lab-4 Analysis

Due Date: 05/03/2022

Submission Date: 05/03/2022

- **NOTE:** Quick Sorts are in recursion and Natural Merge by Linked List implementation is by iteration so there will be a difference in overhead: *Unfortunately, I couldn't find a code that used recursion to implement natural merge sort using linked lists. I am not well versed in programming to write a sort by myself in a week so used a iteration version of natural version of natural merge sort using linked lists.*

- **Pros & Cons of Natural Merge Sort over Traditional Merge Sort:**
  - Traditional Merge Sort requires double space: It creates copies of the list when it calls itself recursively. It also creates a new list to sort the sub-lists and return the sub-lists. As a result, it uses much more memory than bubble and insertion sort which sort the data in place.
  - This is especially true for small lists as time cost of recursion takes it toll and insertion & bubble sort out-pace the Merge Sort. Due to these two limitations, merge sort is not a preferable option to sort large lists when memory is limited or constrained.
  - Run time complexity of merge sort is O(nlogn) which is great for large sized lists. And it is easy to parallelize it uses divide and conquer strategy which results in smaller sub-lists that can distributed and processed in parallel
  - It can access data in sequence hence necessity of random ordered data is low
  - It's a stable sort with same complexity i.e O(nlogn) for best, worst and average case scenario
  - The main reason for using Natural Merge Sort with linked list implementation is space. Space complexity of Natural Merge Sort with linked list implementation using iteration is O(nlogn) while using recursion is O(logn)
  - The main advantage of linked list implementation over list implementation is linked lists allow random access to nodes/data while lists don't allow such easy access.

- **Effect of duplication (15%) in quick sorts:** Not included in analysis

- **Use of larger sizes of input to demonstrate asymptotic cost:** Please refer to following graphs to see effect of input sizes on the cost

- **Relative number of comparisons & swaps**

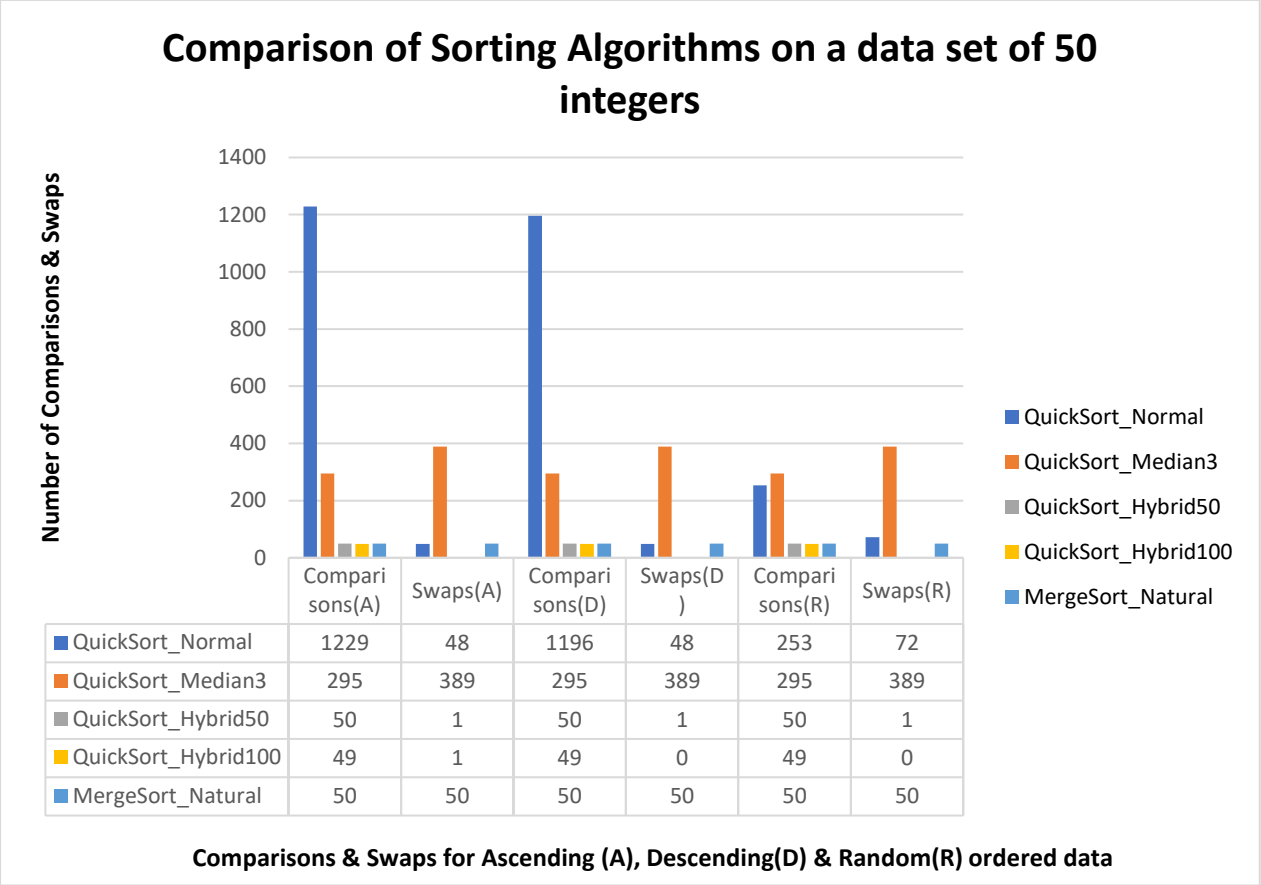
  Relative number of comparisons and swaps in 50 data points (I am going to just compare comparisons) (x = best case scenario):

  For quick sort, best case scenario is 282 O(nlogn)) comparisons and worst case is 2500 (O(n*2)comparisons. My sorting algorithm gave out 4x times comparisons for ascending & descending order data and 1x comparisons for random ordered data. For quick sort with median of three as pivot, 1x comparisons for all three order types of data which I am not sure is accurate. For both hybrid sorts and natural merge sort, my code counted only 50 (i.e n) comparison which is ~17% of best case scenario.

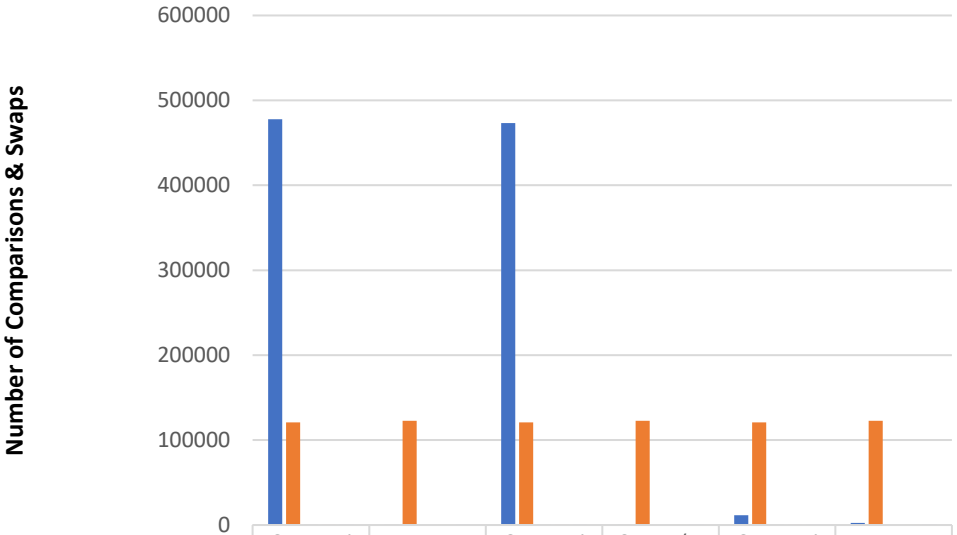
  Random ordered data fared the best in terms of comparisons and swaps than ascending and descending ordered data for quick sort(normal) and quick sort(median of three). While, comparison and swaps numbers were the same for both hybrid sorts. Again, which I think is inaccurate. Performance of these these two hybrid sorts seems to be 5.6 times better than best case scenario of quick sort.

  As far as merge sort is concerned, natural merge sort was 6 times better than best case of traditional merge sort i.e 282 i.e O(nlogn). Not sure if this is accurate.

  Similar, trends for 1k, 2k, 5K and 10k data points as expected.


Comparison of Sorting Algorithms on a data set of 50 integers
Number of Comparisons & Swaps
1400
1200
1000
800
600
400
200
0
QuickSort_Normal
QuickSort_Median3
QuickSort_Hybrid50
QuickSort_Hybrid100
MergeSort_Natural
Comparisons(A) Swaps(A) Comparisons(D) Swaps(D) Comparisons(R) Swaps(R)
QuickSort_Normal 1229 48 1196 48 253 72
QuickSort_Median3 295 389 295 389 295 389
QuickSort_Hybrid50 50 1 50 1 50 1
QuickSort_Hybrid100 49 1 49 0 49 0
MergeSort_Natural 50 50 50 50 50 50
Comparisons & Swaps for Ascending (A), Descending(D) & Random(R) ordered data

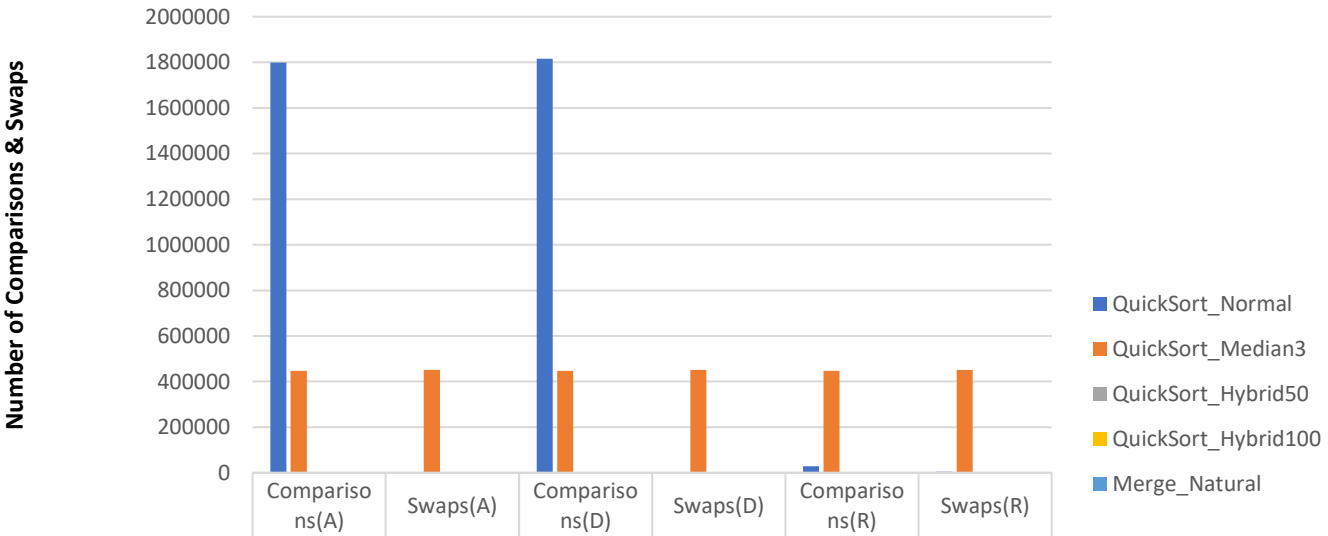
Comparison of Sorting Algorithms on a data set of 1000 integers



	Comparisons(A)	Swaps(A)	Comparisons(D)	Swaps(D)	Comparisons(R)	Swaps(R)
QuickSort_Normal	477856	961	473239	980	11506	2379
QuickSort_Median3	120761	122715	120761	122715	120761	122715
QuickSort_Hybrid50	1000	1	1000	1	1000	1
QuickSort_Hybrid100	1000	1	1000	1	1000	1
Merge_Natural	1000	1000	1000	1000	1000	1000

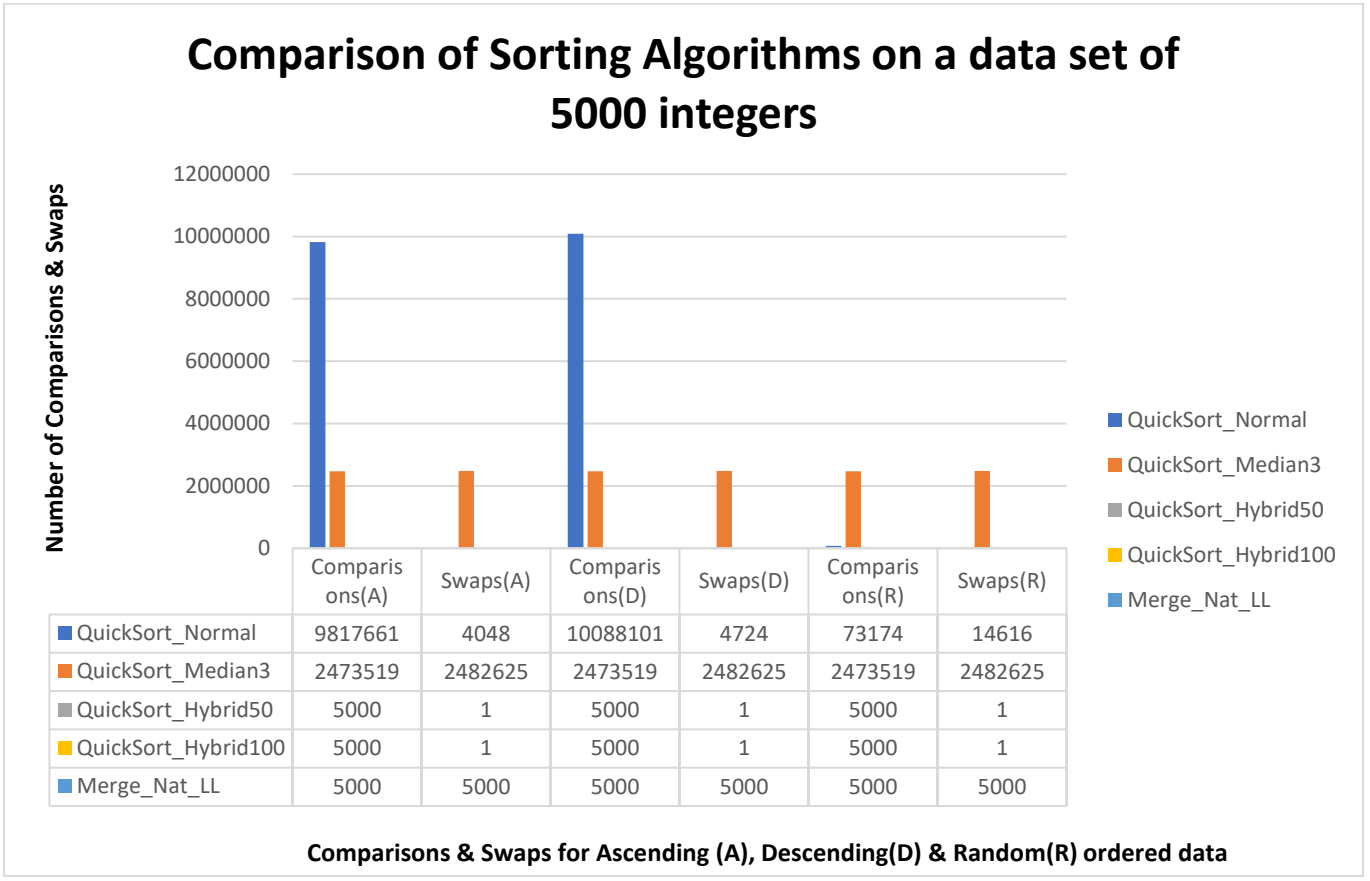
Comparisons & Swaps for Ascending (A), Descending(D) & Random(R) ordered data

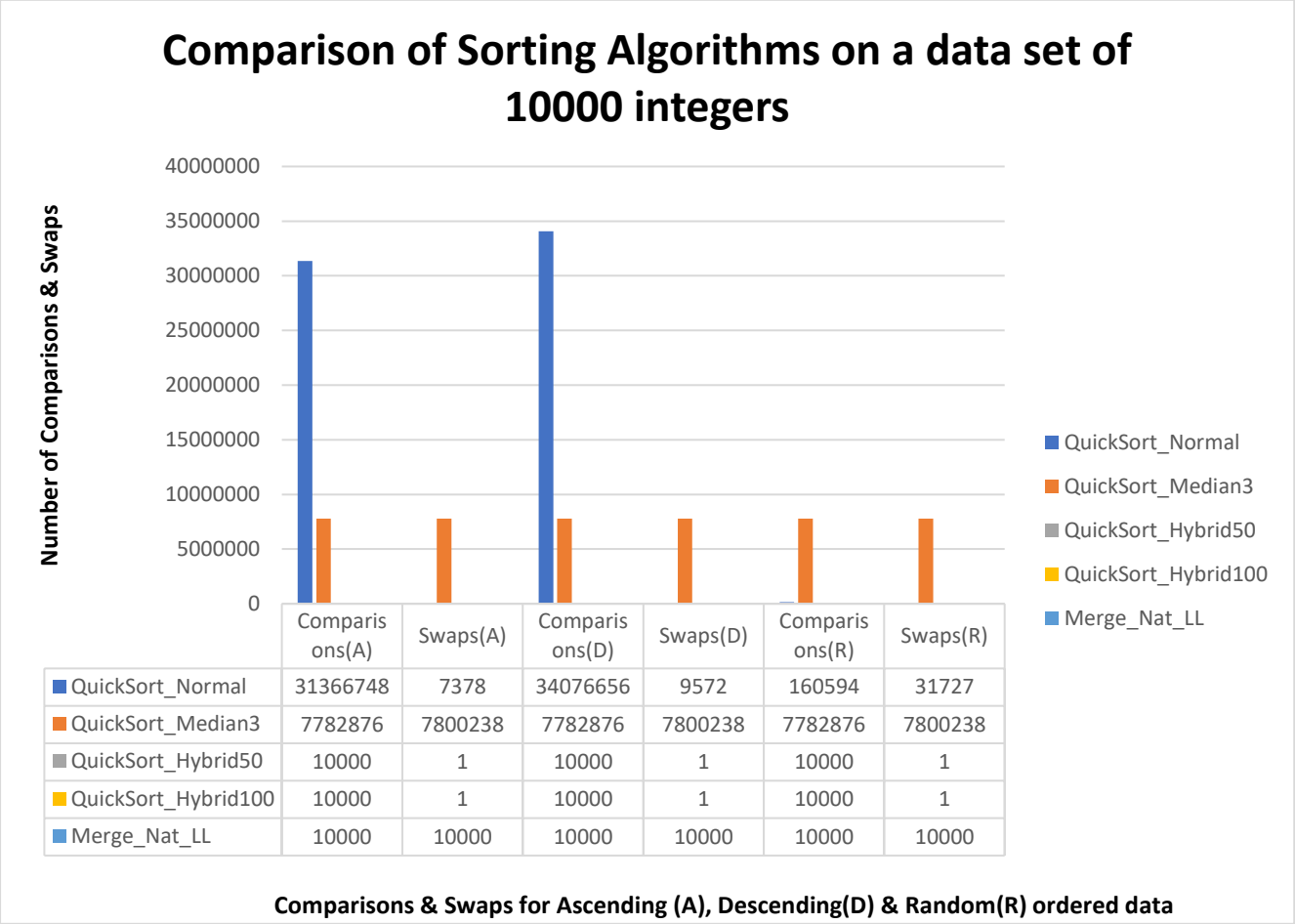
Comparison of Sorting Algorithms on a data set of 2000 integers



	Comparisons(A)	Swaps(A)	Comparisons(D)	Swaps(D)	Comparisons(R)	Swaps(R)
QuickSort_Normal	1799229	1803	1815239	1911	28846	5222
QuickSort_Median3	447080	450878	447080	450878	447080	450878
QuickSort_Hybrid50	2000	1	2000	1	2000	1
QuickSort_Hybrid100	2000	1	2000	1	2000	1
Merge_Natural	2000	2000	2000	2000	2000	2000

Comparisons & Swaps for Ascending (A), Descending(D) & Random(R) ordered data


Comparison of Sorting Algorithms on a data set of 5000 integers
Number of Comparisons & Swaps
12000000
10000000
8000000
6000000
4000000
2000000
0
QuickSort_Normal
QuickSort_Median3
QuickSort_Hybrid50
QuickSort_Hybrid100
Merge_Nat_LL
Comparisons(A)
Swaps(A)
Comparisons(D)
Swaps(D)
Comparisons(R)
Swaps(R)
QuickSort_Normal 9817661 4048 10088101 4724 73174 14616
QuickSort_Median3 2473519 2482625 2473519 2482625 2473519 2482625
QuickSort_Hybrid50 5000 1 5000 1 5000 1
QuickSort_Hybrid100 5000 1 5000 1 5000 1
Merge_Nat_LL 5000 5000 5000 5000 5000 5000
Comparisons & Swaps for Ascending (A), Descending(D) & Random(R) ordered data

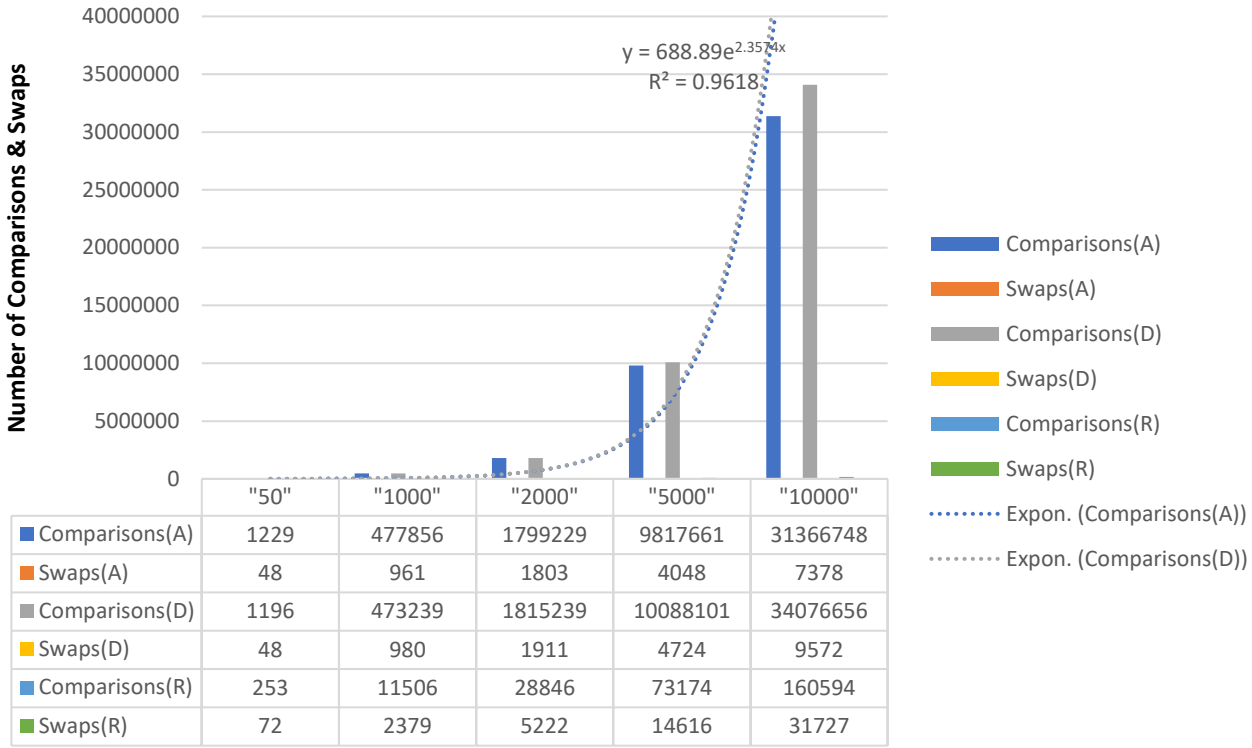


- **Effect of the order of data on comparisons & swaps**

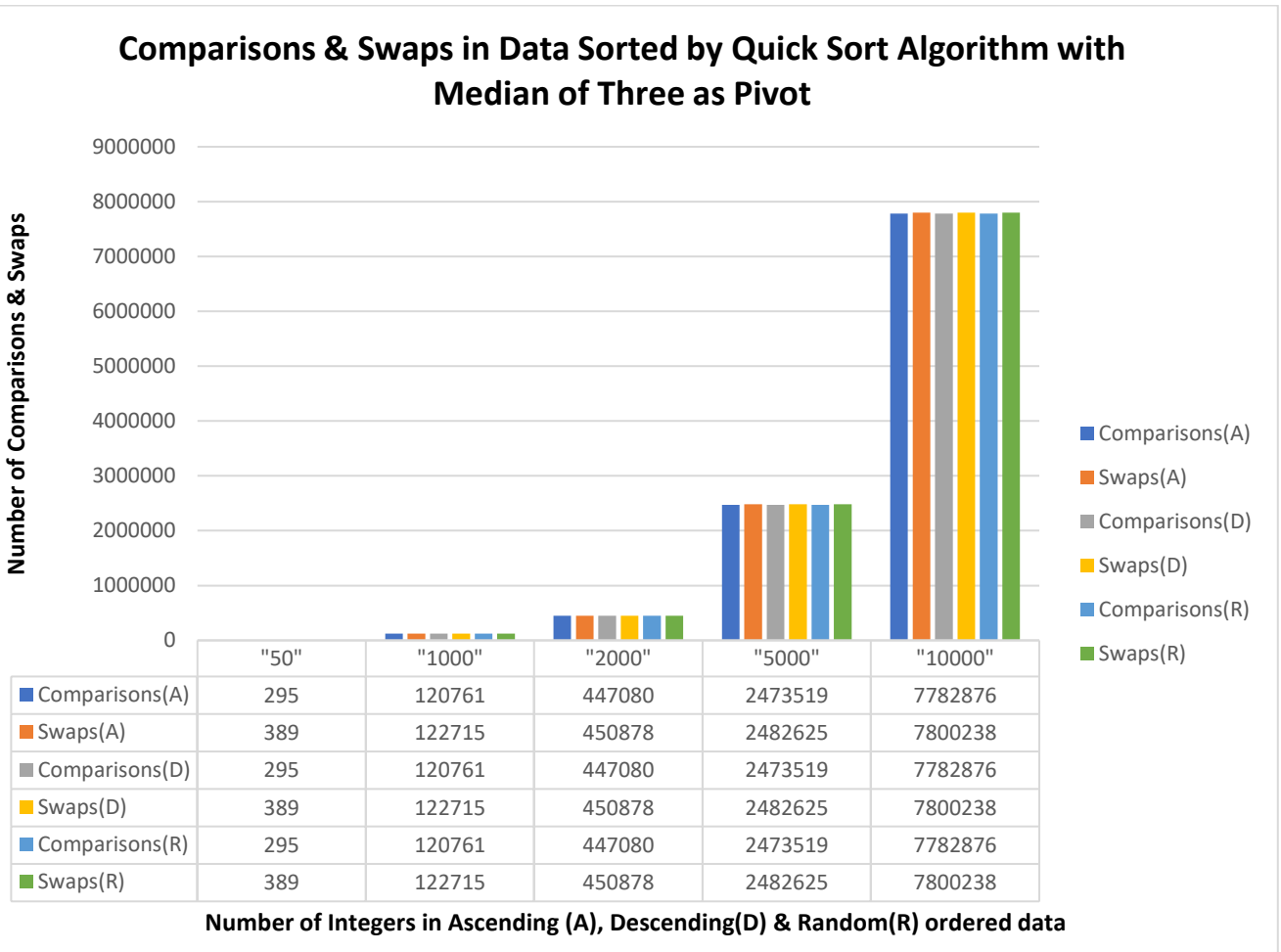
Random ordered data had the best performance among the three in terms of comparisons but not so in terms of swaps. This is true for normal quick sort and quick sort with pivot as median but for other three sorts number of comparisons were exactly the same i.e 50 comparisons and 1 swap. I think I made some mistake in the code in calculations.

50 data points:

- Ascending and descending ordered data had less number of swaps (1.5 times less) than random ordered data in normal quick sort but more number of comparisons (5 times more). Equal number of swaps and comparisons for all four other sort types irrespective data order which I think is inaccurate. Similar trends for higher number of data with number of swaps in random ordered data being 2.5-3 times more than ordered data while difference in comparisons being 40-200 times more in ordered data
- Effect of difference size files
- Number of swaps and comparisons in two hybrids sorts and natural merge sort increased linearly which I think is inaccurate but I don't have time to trouble shoot my code. As far as normal quick sort and quick sort with median as pivot are concerned, number of swaps and comparisons increase exponentially with size for all data order types. Please refer to the following two graphs.



	"50"	"1000"	"2000"	"5000"	"10000"
Comparisons(A)	1229	477856	1799229	9817661	31366748
Swaps(A)	48	961	1803	4048	7378
Comparisons(D)	1196	473239	1815239	10088101	34076656
Swaps(D)	48	980	1911	4724	9572
Comparisons(R)	253	11506	28846	73174	160594
Swaps(R)	72	2379	5222	14616	31727



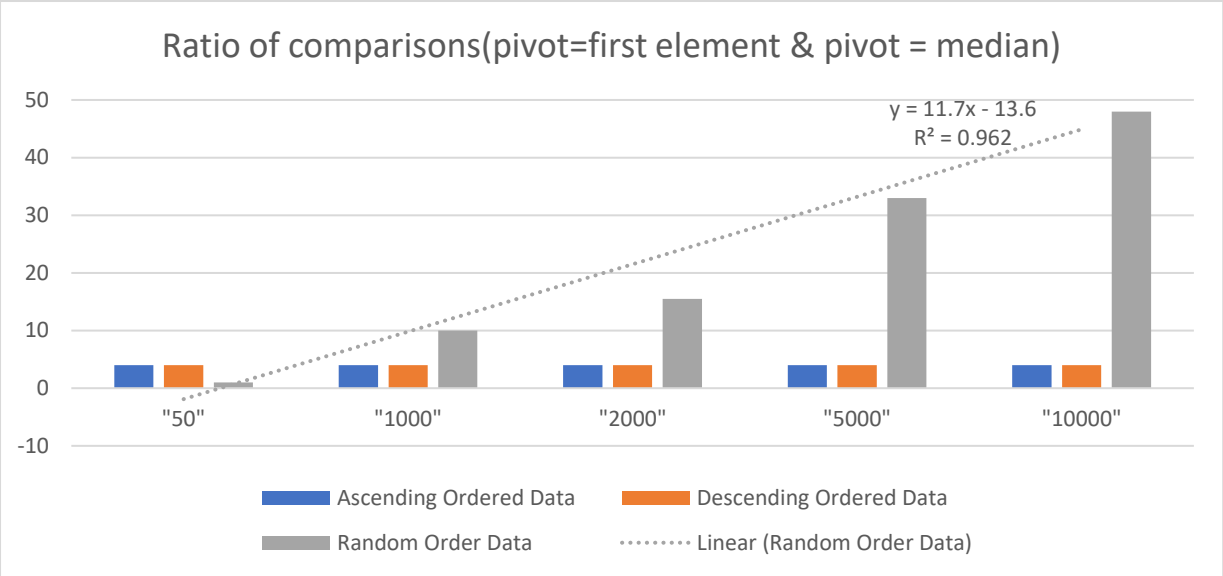
	"50"	"1000"	"2000"	"5000"	"10000"
Comparisons(A)	295	120761	447080	2473519	7782876
Swaps(A)	389	122715	450878	2482625	7800238
Comparisons(D)	295	120761	447080	2473519	7782876
Swaps(D)	389	122715	450878	2482625	7800238
Comparisons(R)	295	120761	447080	2473519	7782876
Swaps(R)	389	122715	450878	2482625	7800238

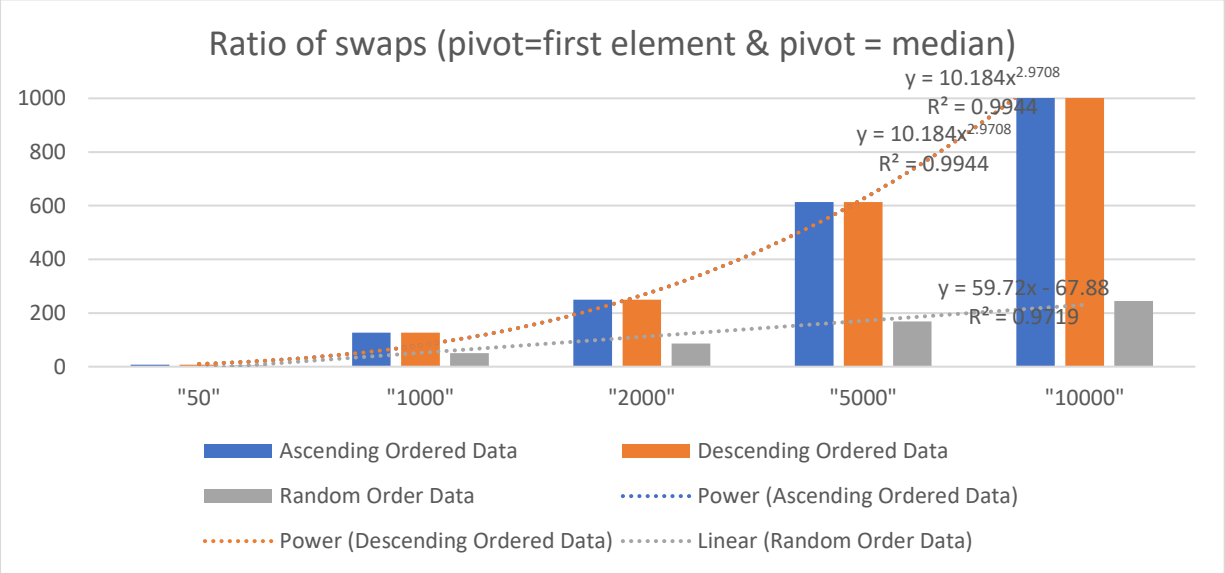
- **Effect of different partition sizes:**

Looks like my code has bugs and partition size (two hybrid sorts) give out same number of swaps and comparisons for three data order types.

- **Effect of pivot selection methods**

Yes, dramatic changes in performance in all three orders of data (ascending, descending and random. Number of comparisons are 4 times more when pivot is first element than when it is median; this is true for ascending and descending ordered data irrespective of number of data points. For random ordered data, comparisons are more when pivot is median; the ratio of comparisons in median pivot to first element as pivot increases with number of data points. Hence, performance in terms of comparisons is better in ascending and descendingordered data when the pivot is first element. It is 4 times better irrespective of number of data points. Performance is better in random ordered data when pivot is a median. The performance-betterment ratio increases linearly with number of data points.





- **Performance in terms of number of swaps varies in a different manner with choice of pivot.** Performance is better throughout when pivot is first element i.e number of swaps are more when pivot is median. The ratio of swaps in QuickSort with median pivot to Quick Sort with first element

as pivot increases at the same rate for ascending and descending ordered data. This rate is more than rate of increase for random ordered data.

- **Effect of Natural Merge Sort:**

My data shows that number of swaps and comparisons increase linearly with number of data points and are equal to number of data points to sort. Not sure if this is correct. As mentioned earlier, natural merge sort reduces space complexity by half

- **Factor with maximum effect on efficiency**

As per my numbers, median as pivot in quick sort and jus the use of merge sort with linked list makes a huge difference. I don't trust my hybrid sort numbers so can't comment on them.

- **Time & Space efficiency**

	Best	Worst	Avg	Space(worst)
Quick Sort	O(nlog(n))	O(n*2)	O(nlog(n))	O(nlog(n))
Merge Sort	O(nlog(n))	O(nlog(n))	O(nlog(n))	O(2n)
Linked List				O(n)

Data(n)	QuickSort(Worst)	QuickSort(Best)	MergeSort(Worst&Best)
50	2500	282.2	282.2
1000	1000000	9966	9966
2000	4000000	21932	21932
5000	25000000	61440	61440
10000	100000000	132880	132880

- **Justification of data structures**
  1. We were asked to use linked list for natural merge to save space especially for large datafiles. Traditional merge sort's space efficiency is O(2n) and use of linked lists results in just O(n).
  2. The worst and average time efficiency for linked lists for access, search, insertion and deletion is O(n), O(n), O(1), O(1) respectively.
  3. For the quick sort and its variations, I used lists to import data from the text files and passed lists as arguments into functions.
  4. The worst and average time efficiency for lists for access, search, insertion and deletion is O(1), O(n), O(n), O(n) respectively.
  5. Since lots of comparisons and swaps are involved in quick sort, O(1) access efficiency is of supreme importance.

- **Justification of choice – iteration vs recursion**
  1. I didn't quite think a lot about iteration and recursion. Recursions are easy to write but may cost more space while iterations are difficult to write while cost less space. I used the codes from internet and most of them were recursion based so modified them as per my understanding.

- **What did I learn**
  1. Better use of args i.e gave a location of folder, choose files, process files and output multiple files in the same folder although would have liked to choose a different folder
  2. How to analyse and modify other persons code
  3. Know more about sorting algorithms – basics, implementations, complexities, how they are affected by data order and number of data points
  4. Understand situations when and which sorts can be used

- **What would I do differently:**
  - Use duplicates to understand offect of duplicates on sorting algorithms
  - Measure time taken by algorithms
  - Use iteration also to see how sorting performance changes
  - Use better logic to count swaps and comparisons

References:

1. www.realpyton.com
2. https://stackabuse.com/sorting-algorithms-in-python/
3. https://www.geeksforgeeks.org/merge-sort-for-linked-list/