Section 1 – Best Sort?

Data generation

I use random_data(size, seed) to generate a array of a given size with a given seed for the random numbers. Inside the function I use malloc to dynamically create arrays of given sizes and use srand() to set the random seed.

After generation, I copy the random array into four arrays for each algorithm to sort respectively, making sure they are sorting the same random array. Then the random array is freed to avoid memory leakage.

I also checked each array to see if they are sorted correctly to make sure the data is valid.

To maximize automation, I wrote a function called get_steps(size, seed), which prints out steps taken to sort the random array for each sorting algorithm generated by the given seed, and created two corresponding arrays for size and seed to call get_steps() iteratively.

Choice of test cases

To know the difference better at smaller sizes and find the crossing point between mergesort and insertion sort, I incremented the size by 5 each time until 50, then I just try to increase by 10 times and tested for the middle (5 times) in case it goes up too fast. After some simple testing I just found that insertion sort gets pretty slow at size of 500,000. Since getting data for 5 data sets already took hours and the difference is very obvious at the point, I just stopped there. To accommodate the big numbers of steps taken by insertion sort, I changed the type of number_steps in logger to long int.

Results

I made what I got from the program into a table:

Data Set 1

Array Size	Mergesort	Hybrid Sort	Quicksort	Insertionsort
1	3	3	1	2
5	151	45	165	26
10	374	199	320	108
15	614	346	516	406
20	876	567	662	744
25	1151	751	849	1186
50	2598	1769	2083	5716
75	4183	2872	3187	11006
100	5814	4193	4332	20056
500	35806	28495	25548	497216
1000	77644	63105	54368	2014504
5000	460482	383937	315958	49523384
10000	980560	828737	669988	197560744
50000	5599002	4812727	3798048	5004532800
100000	11797508	10227713	7935755	1.9959E+10
500000	65773414	58278867	43970685	4.9987E+11

Data Set 2

Array Size	Mergesort	Hybrid Sort	Quicksort	Insertionsort
1	3	3	1	2
5	149	45	140	26
10	374	167	263	68
15	618	338	507	278
20	878	527	735	672
25	1147	757	922	1282
50	2612	1777	1962	5452
75	4187	2814	3098	10790
100	5822	4243	4270	21096
500	35860	28161	25660	500904
1000	77708	62395	54306	1956632
5000	460460	384429	316493	49790256
10000	980768	829173	673653	199905304
50000	5598266	4813187	3775097	5005310088

100000	11796622	10226275	7929349	1.9948E+10
500000	65774866	58276635	44230748	4.9942E+11

Data Set 3

Array Size	Mergesort	Hybrid Sort	Quicksort	Insertionsort
1	3	3	1	2
5	153	37	124	18
10	374	181	336	164
15	604	384	523	502
20	874	573	687	952
25	1141	733	872	1394
50	2618	1723	1994	4324
75	4193	2876	3262	10422
100	5846	4001	4412	17408
500	35890	28209	25320	501080
1000	77698	62581	55011	1993464
5000	460436	383867	317275	51028656
10000	980808	828503	673270	198674216
50000	5598854	4812235	3779076	5000252288
100000	11797722	10227815	7966322	1.9964E+10
500000	65774194	58278323	43868744	4.999E+11

Data Set 4

Array Size	Mergesort	Hybrid Sort	Quicksort	Insertionsort
1	3	3	1	2
5	147	69	140	50
10	372	199	295	100
15	614	370	500	350
20	876	567	703	720
25	1145	801	922	1138
50	2602	1921	1912	5292
75	4169	2956	3130	11134
100	5792	4367	4236	21112
500	35784	28727	25074	499760
1000	77622	63037	55015	1984304
5000	460544	384801	316075	50411280
10000	980968	828565	671373	202161648
50000	5598084	4814195	3780104	5010634528
100000	11796572	10225891	7997186	1.9979E+10
500000	65776062	58267999	44121336	4.9909E+11

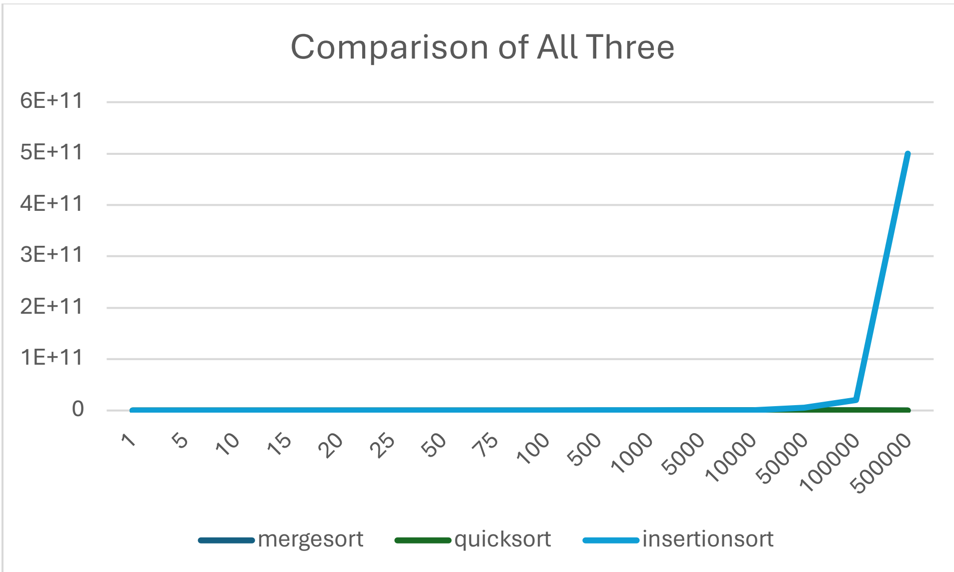
Data Set 5

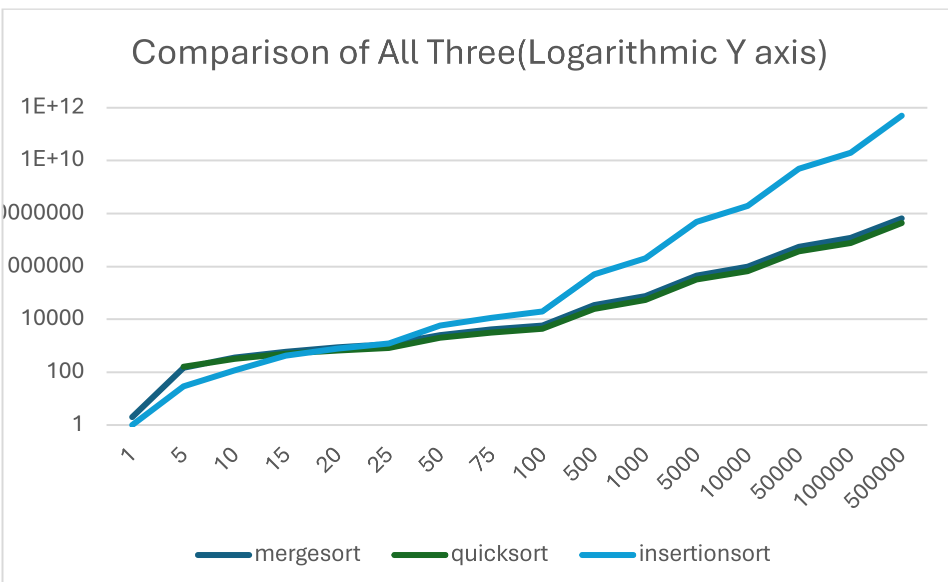
Array Size	Mergesort	Hybrid Sort	Quicksort	Insertionsort
1	3	3	1	2
5	149	61	165	42
10	368	211	336	188
15	606	366	475	406
20	868	603	719	648
25	1137	737	881	1282
50	2586	1851	2035	5436
75	4169	3016	3205	11446
100	5796	4323	4275	19968
500	35788	28485	25678	487104
1000	77618	63035	55340	2033280
5000	460486	382371	318281	50515800
10000	980532	828029	671141	201527776
50000	5599002	4811807	3771560	4992409384
100000	11797328	10229715	7975659	1.9971E+10
500000	65773086	58278673	44209748	5.0012E+11

AVERAGE

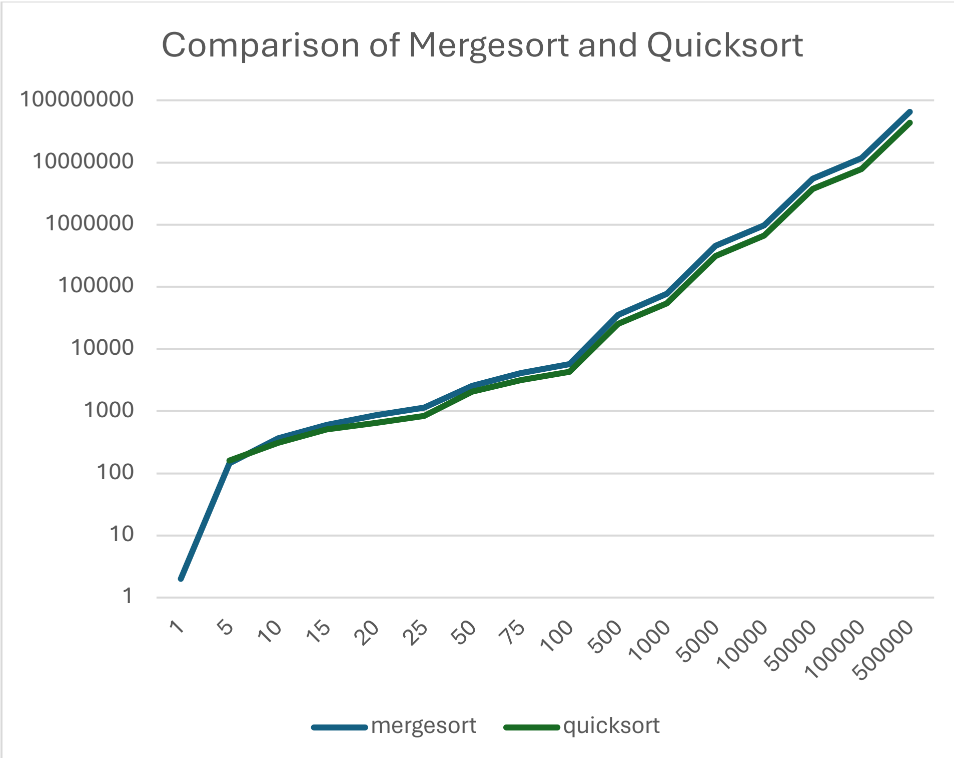
Mergesort	Hybrid Sort	Quicksort	Insertionsort
3	3	1	2
149.8	51.4	146.8	32.4
372.4	191.4	310	125.6
611.2	360.8	504.2	388.4
874.4	567.4	701.2	747.2
1144.2	755.8	889.2	1256.4
2603.2	1808.2	1997.2	5244
4180.2	2906.8	3176.4	10959.6
5814	4225.4	4305	19928
35825.6	28415.4	25456	497212.8
77658	62830.6	54808	1996436.8
460481.6	383881	316816.4	50253875.2
980727.2	828601.4	671885	199965938
5598641.6	4812830.2	3780777	5002627818
11797150.4	10227481.8	7960854.2	1.9964E+10
65774324.4	58276099.4	44080252.2	4.9968E+11

From the table I made a diagram of average steps to compare all three, and the proportion got weird because the large size takes way more steps, so I went on and made another diagram with a logarithmic Y axis





From the second diagram we can clearly see that the insertion sort gets significantly slower when it comes to large sizes, and the difference will become larger if the size continues to grow, though it is actually faster when the array is smaller than size of about 15. Since the merge sort and quicksort still look close, I made another diagram just for these two:



We can see that while merge sort and quick sort are following the same pattern of increase, the quick sort always takes less steps, I think the most significant reason for this is that quicksort doesn't need an extra array and can be done within the original array, which saves a lot of steps of copying one array into another, especially when the array gets large. Also, this got me thinking, merge sort might have an edge if we are sorting a linked list instead of an array.

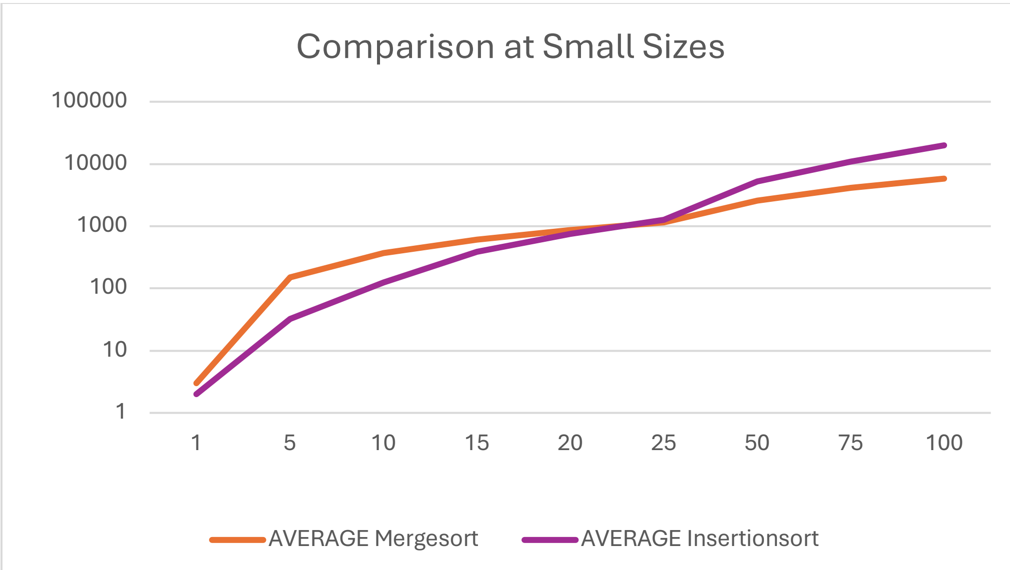
Conclusion

Amid the 3 algorithms, insertion sort is obviously the overall worst, taking significantly more time when the array gets large, while having an edge when the size is less than about 15. When sorting an array as we are doing now, quicksort is faster than merge sort as it doesn't need to make and copy from a temporary array. <u>So, when sorting a **LARGER** sized **ARRAY** quicksort is overall the best in my opinion.</u>

However, I can assume from above that when sorting a linked list merge sort could have an edge because it's easier to copy a list and merge lists (because they go one by one) and it's harder to partition (traversing in the list back and forth and swapping elements).

Section 2 – Hybrid Sort

To zoom in on the difference when arrays are small, I made a diagram for size 1-100:



As the diagram shows, the crossing point is approximately between 20-25, and the difference peaks between 5-10.

My thought is that, if the algorithm switches to insertion sort right at the crossing point, we won't get much benefit as they both take similar steps, only the reminders of sizes smaller than the crossing point will save significant steps. Consequently, I think we should switch at the point where the difference is the most to maximize the benefit. According to the graph the turning point should be set between size 5 and 10.

So I set the turning point at 10, and got the table as follows:

AVERAGE		
Array Size	Mergesort	Hybrid Sort
1	3	3
5	149.8	51.4
10	372.4	191.4
15	611.2	360.8
20	874.4	567.4
25	1144.2	755.8
50	2603.2	1808.2
75	4180.2	2906.8
100	5814	4225.4
500	35825.6	28415.4

1000	77658	62830.6
5000	460481.6	383881
10000	980727.2	828601.4
50000	5598641.6	4812830.2
100000	11797150.4	10227481.8
500000	65774324.4	58276099.4

I also tested for some other turning points such as 15, 20, etc. which as expected don't work as fast as 5 and 10 on average. This can count as some empirical proof for my choice of turning point.