1 Theoretical Analysis

The theoretical analysis of all the sorting algorithms were covered in class. For reference, we have the table of the relevant sorting algorithms and their efficiency for the best, worst, and average cases.

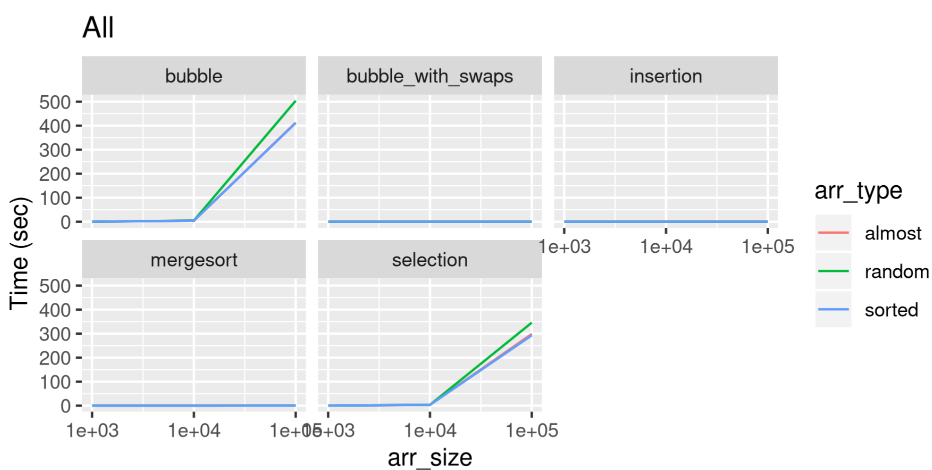
Algorithm	Best	Worst	Average
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble (with swaps)	$\Theta(1)$	$\Theta(n^2)$	$\Theta(n^2)$
Quick	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
Merge	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

2 Empirical Analysis

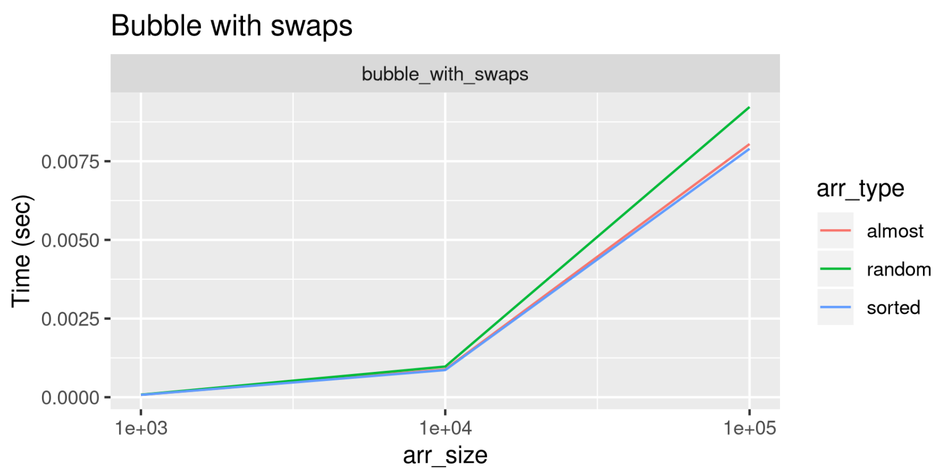
The following section covers the empirical analysis of the above sorting algorithms. Due to memory issues (for more details see section 2.2), quicksort gets its own section. All the algorithms were implemented in Python, and the source code can be found on GitHub. The tables with the raw data for the analysis can be found in appendix A.

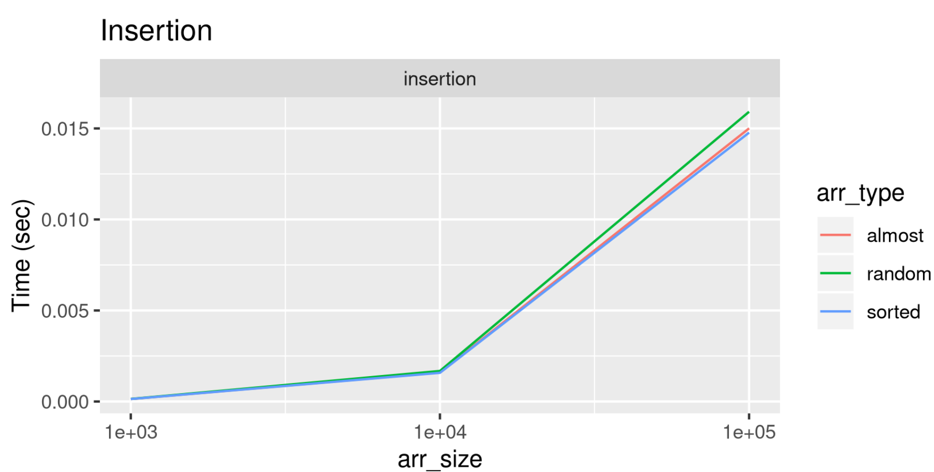
2.1 Graphs and Discussion

We have the graphs from all the algorithms below.

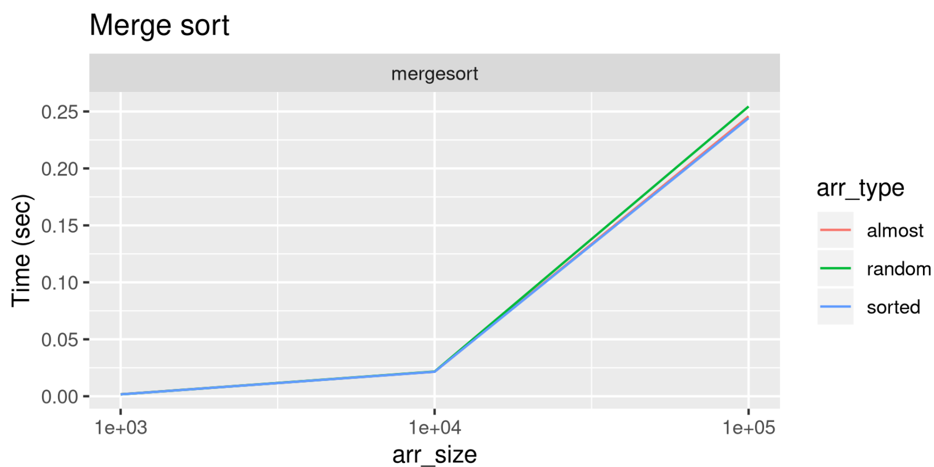


From the graphs we can see that bubble and selection sort grow much quicker than the other sorting algorithms. So much so, that the scale of the vertical axis makes it to where the other algorithms are basically a flat line. To see the growth more clearly of the other sorting algorithms, we have separate graphs.





It is hard to tell with only three data points, but we can see that all the sorting algorithms have similar shapes, albeit with bubble and selection having much steeper curves. This makes sense for the most part since most of the algorithms (except for mergesort), have efficiency $\Theta(n^2)$. From a purely theoretical view, we would expect to see that mergesort has a less steep



curve than the other algorithms since it is non-adaptive and has efficiency $\Theta(n \log n)$. However, the curve is the same and, in fact, the absolute time is much higher than either the improved bubble sort or the insertion sort. On the other hand, our theoretical analysis is supported by the data for mergesort being non-adaptive because the curves for all the array types (sorted, almost sorted, and random) are essentially the same. This does not hold for all the algorithms that are adaptive.

In tables 1 and 2, we can see that the improved bubble sort performs the best for both almost sorted and random arrays, with insertion sort being a close second. This hold true for sorted lists as well, which confirms our theoretical analysis with the improved bubble sort having best case efficiency $\Theta(1)$. Of course, the question of which sorting algorithm performs best on an already sorted list is a bit moot.

Selection and regular bubble sort perform much worse than the others. This shows that even if two algorithms have the same order of growth, it does not necessarily mean they are as fast as one another.

Algorithm	Average time (sec)
Bubble with swaps	0.00300
Insertion	0.00558
Mergesort	0.0897
Selection	101
Bubble	139

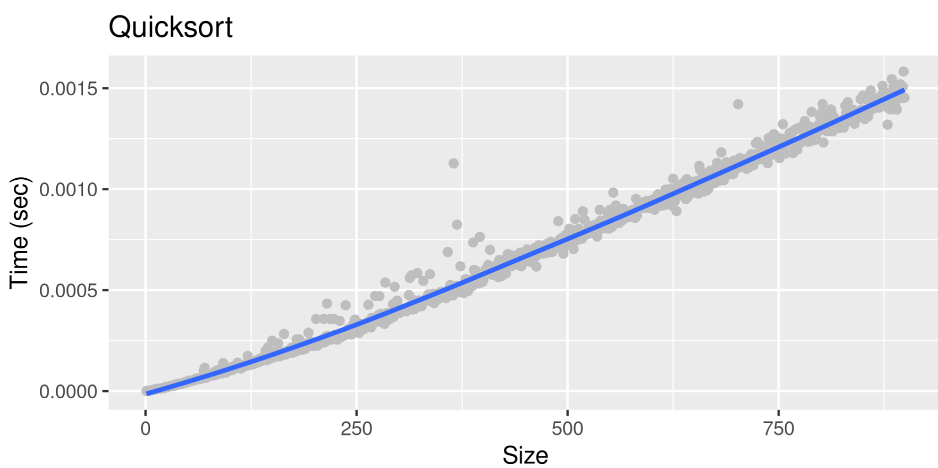
Table 1: Almost sorted

Algorithm	Average time (sec)
Bubble with swaps	0.00300
Insertion	0.00558
Mergesort	0.0897
Selection	101
Bubble	139

Table 2: Random

2.2 Quicksort

The space efficiency of quicksort is $\Theta(n)$. Therefore, in our code, when we try to apply quicksort to an already sorted array (the worst case scenario), we quickly hit Python's recursion stack limit of 1000. This is true even for random arrays bigger than about 1000. Thus, quicksort was not included with the other algorithms. We did, however, run quicksort on arrays of random elements to confirm our theoretical analysis. From plot 2.2, we can see that our empirical analysis supports our theoretical analysis that quicksort is of efficiency $\Theta(n \log n)$.



A Tables of Raw Data

Main data:

Algorithm	Array size	Array type	Time
bubble	1000	random	0.038051
bubble	1000	sorted	0.036030
bubble	1000	almost	0.036322
bubble_with_swaps	1000	random	0.000080
bubble_with_swaps	1000	sorted	0.000074
bubble_with_swaps	1000	almost	0.000077
insertion	1000	random	0.000141
insertion	1000	sorted	0.000136
insertion	1000	almost	0.000138
selection	1000	random	0.029308
selection	1000	sorted	0.029346
selection	1000	almost	0.029343
mergesort	1000	random	0.001656
mergesort	1000	sorted	0.001664
mergesort	1000	almost	0.001725
bubble	10000	random	4.894341
bubble	10000	sorted	4.316173
bubble	10000	almost	4.411719
bubble_with_swaps	10000	random	0.000974
bubble_with_swaps	10000	sorted	0.000863
bubble_with_swaps	10000	almost	0.000884
insertion	10000	random	0.001678
insertion	10000	sorted	0.001576
insertion	10000	almost	0.001597
selection	10000	random	3.433090
selection	10000	sorted	3.214696
selection	10000	almost	3.250444
mergesort	10000	random	0.021676
mergesort	10000	sorted	0.021493
mergesort	10000	almost	0.021575
bubble	100000	random	504.072366
bubble	100000	sorted	411.598890
bubble	100000	almost	412.980531
bubble_with_swaps	100000	random	0.009224
bubble_with_swaps	100000	sorted	0.007896

Algorithm	Array size	Array type	Time
bubble_with_swaps	100000	almost	0.008047
insertion	100000	random	0.015917
insertion	100000	sorted	0.014770
insertion	100000	almost	0.015004
selection	100000	random	345.659118
selection	100000	sorted	292.783123
selection	100000	almost	298.523657
mergesort	100000	random	0.254392
mergesort	100000	sorted	0.244164
mergesort	100000	almost	0.245749

Quicksort (first 50 rows):

Size	Time
1	2.860033e-07
2	9.200012e-07
3	1.471999e-06
4	2.091001e-06
5	2.850000e-06
6	3.626999e-06
7	4.440997e-06
8	5.275004e-06
9	6.234004e-06
10	6.784001e-06
11	8.042007e-06
12	9.005002e-06
13	9.743002e-06
14	1.082400e-05
15	1.161200e-05
16	1.182300e-05
17	1.120100e-05
18	1.332000e-05
19	1.425501e-05
20	1.428901e-05
21	1.416600e-05
22	1.551500e-05
23	1.858900e-05
24	2.174100e-05
25	1.893500e-05

Size	Time
26	1.985000e-05
27	2.217100e-05
28	2.274000e-05
29	2.392000e-05
30	2.470800e-05
31	2.475700e-05
32	2.661700e-05
33	2.646400e-05
34	2.634100e-05
35	3.218700e-05
36	3.039000e-05
37	3.291200e-05
38	3.145500e-05
39	3.752200e-05
40	3.581899e-05
41	3.496500e-05
42	3.750700e-05
43	3.590100e-05
44	4.008000e-05
45	3.819700e-05
46	4.240200e-05
47	4.525700e-05
48	4.359600e-05
49	4.237900e-05
50	4.867100e-05