ECE345
Assignment 1 Writeup
Ryan Do (1001254117), Munef Agag (1000973707)

6.4.

(a) Experimentally identify the runtime complexity and identify any constants that determine the performance of your algorithms. You will need to run your experiments for various input sizes to obtain good approximations for the growth functions that characterize your algorithms' runtimes, but also to obtain a good estimate for the constants.

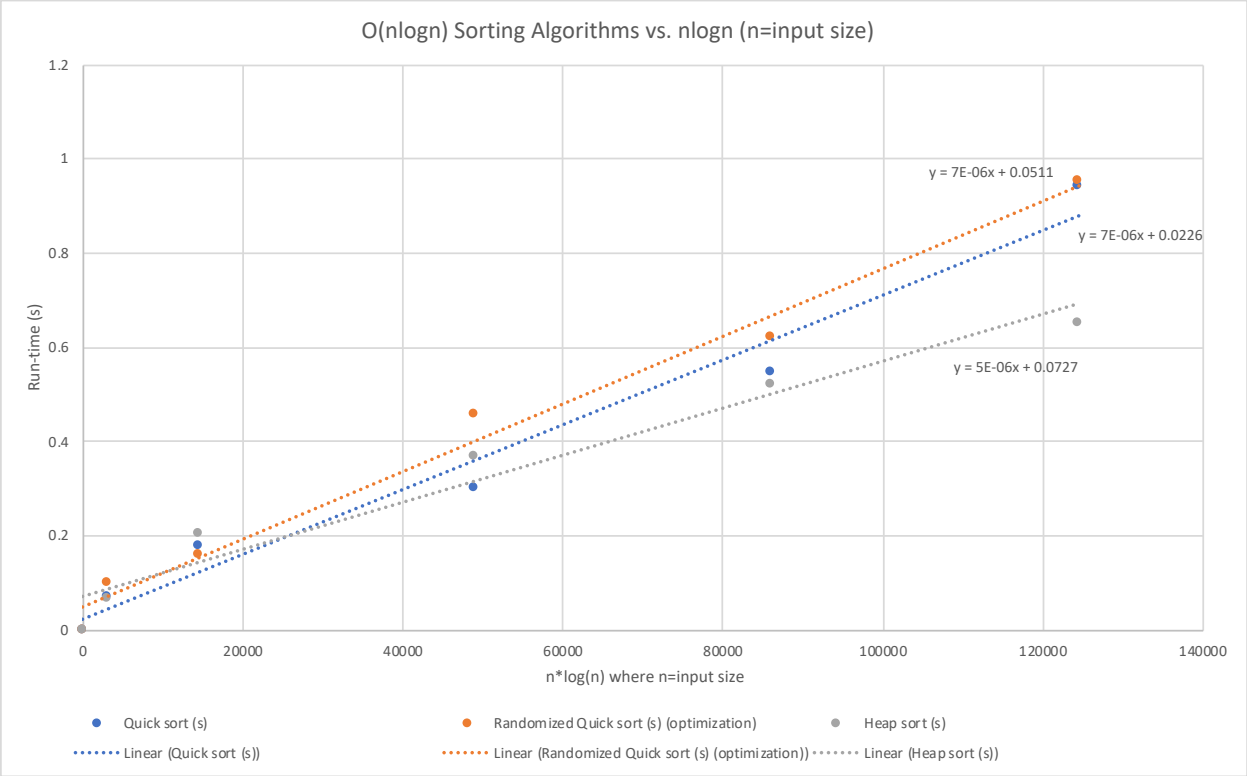
Input size (n)	Quick sort (s)	Randomized Quick sort (s) (optimization)	Heap sort (s)	Bubble sort (s)
0	0	0	0	0
1000	0.07192	0.1012	0.06806	0.1463
4000	0.1790	0.1586	0.2053	1.6300
12000	0.3005	0.4583	0.3697	10.0193
20000	0.5484	0.6222	0.5223	40.0157
28000	0.9402	0.9533	0.6528	73.3262

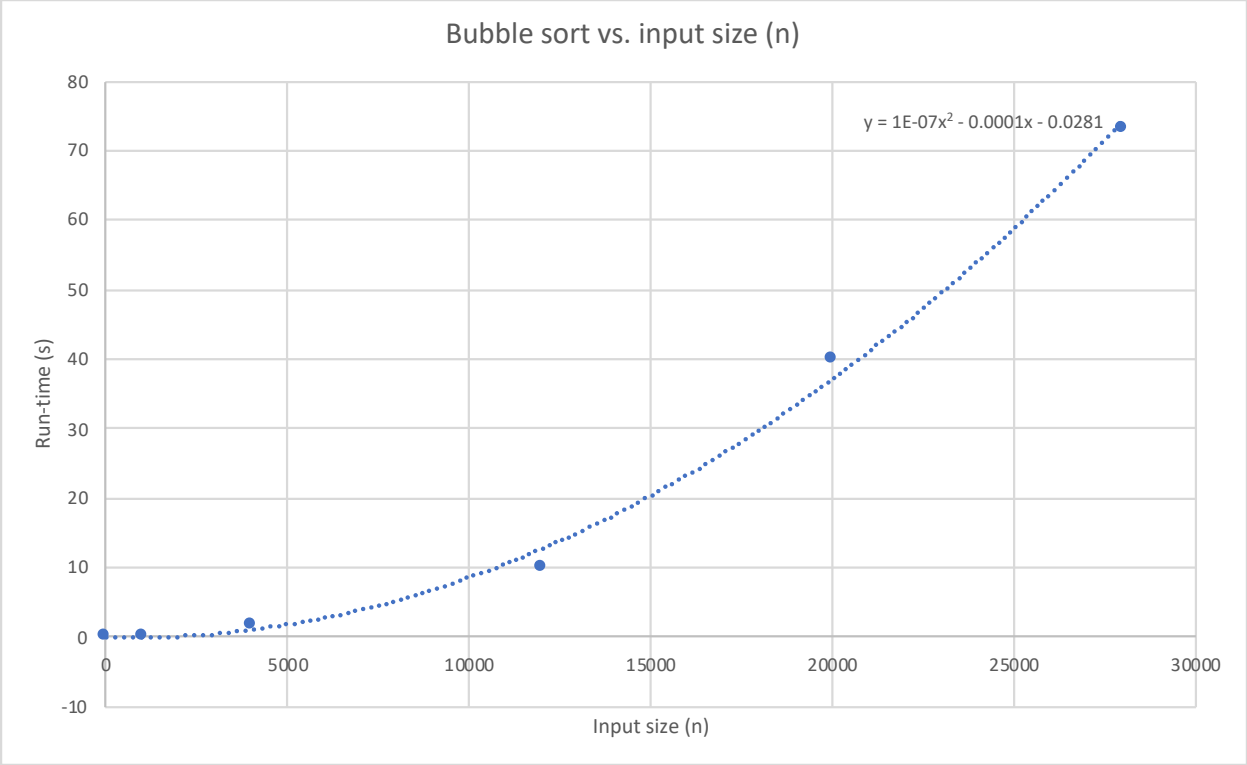
We know that quick sort in the worst case (a near sorted list) has a runtime complexity of **O(n^2**) but is O(nlgn) in the average case. The average case can be expected by adding a small optimization that randomizes the index of the pivot with every iteration. Heap sort has a runtime complexity of **O(nlgn)**. Bubble sort has a runtime complexity of **O(n^2)**.

Experimentally, with the runtimes for multiple input size scenarios shown above, the complexities can be characterized. For bubble sort for example, an input size increase from 4000 to 12000 (3x) leads to a ~9x increase in runtime. Quick sort and heap sort roughly reflect O(nlogn) behaviour. Nlogn is faster is linear but not quite quadratic.

(b) Provide one graph that demonstrates how you determined the constants for each of your implementations.

Input size (n)	nlog(n)	n^2	Quick sort (s)	Randomized Quick sort (s) (optimization)	Heap sort (s)	Bubble sort (s)
0	0	0	0	0	0	0
1000	3000	1000000	0.07192	0.1012	0.06806	0.1463
4000	14408.24	16000000	0.179	0.1586	0.2053	1.63
12000	48950.175	144000000	0.3005	0.4583	0.3697	10.0193
20000	86020.5999	400000000	0.5484	0.6222	0.5223	40.0157
28000	124520.425	784000000	0.9402	0.9533	0.6528	73.3262





Two graphs were produced since the O(nlogn) plots required a nlogn valued x-axis while for O(n^2), a n valued x-axis was sufficient. The constants were determined by performing a regression on the run-time vs input size data points using each given growth function, and taking the coefficient of the largest growth term in the resulting function. Note that for $n > 0$, the functions are monotonically increasing thus the n0 constant is 0.

(c) Provide a table that lists row-wise the algorithm implemented, its growth function, and its constant.

	Growth Function	**Constant (to convert to s)**
Quick sort	O(nlog(n))	7e-06
Randomized Quick sort	O(nlog(n))	7e-06
Heap Sort	O(nlog(n))	5e-06
Bubble Sort	O(n^2)	1e-07

(d) Pick any one of your algorithms and realize an optimization of your choice; re-run the experiments and report the outcome in your graph and table as well.

Implemented optimization for quick sort: randomized quick sort.

(e) What do you conclude about the performance of the four implementations (three different algorithms, one with additional optimization)?

At larger input sizes, the run-time discrepancy is far more noticeable, especially between O(n^2) bubble sort and quick sort/ heap sort. Randomized quick sort actually did not offer a performance increase over regular quick sort using this particular dataset. Perhaps the ordering of the sort keys were already in such a way that quick sort performs better than average with it.
As expected, quick sort performed better than heap sort in general. This is due to the fact that heap sort performs many redundant swaps due to repeated calls to heapify to extract the root out of the heap, which bubbles down small sort keys. Also as expected, bubble sort performed abysmally with a guaranteed runtime of O(n^2). The repeated swaps also give this a high multiplier since 3 writing actions are required for each swap.

(f) Diligently list all sources that you used and discuss the optimization you applied.

Sources

Jacobsen, Hans-Arno. 2019. *ECE345 Lecture Slides.*

The optimization I applied was randomized quick sort, which removes bias of values towards a particular index in a list. Taking the last index in the input data causes quick sort to perform at O(n^2) for near sorted data since the partitioning merely slices off one element each time. By adding randomization and swapping the last index element with a random element in the data set, the average pivot value is the median of the set. As a result, the partitioning recursively divides each subsequent sub-problem into two equally sized subsets (on average), avoiding the worst-case O(n^2) time complexity for near sorted lists.