Assignment 2 – Sorting Algorithms

Alikhan Makhamadaliev
CSC 4180
February 6, 2026

Performance Comparison on Random Data

How the experiments were run:

For this experiment, I tested six different sorting algorithms: Bubble sort, Insertion sort, Selection sort, Shell sort, Quick sort, and Merge sort.

Each algorithm was tested on arrays with the following sizes:

100, 500, 1000, 2000, 5000, 10000, 20000, 75000, and 150000 elements.

The arrays were filled with random integers. For each array size, the sorting was run 20 times, and the average runtime was recorded. Timing was done using System.nanoTime() and only included the time spent sorting the array, not generating it. All times are reported in milliseconds.

Results on Random Data

N	Bubble (ms)	Insertion (ms)	Selection (ms)	Shell (ms)	Quick (ms)	Merge (ms)
100	0.162	0.134	0.064	0.022	0.067	0.019
500	0.257	0.159	0.207	0.108	0.032	0.038
1000	0.606	0.502	0.227	0.085	0.057	0.081
2000	1.833	0.260	0.824	0.131	0.081	0.324

N	Bubble (ms)	Insertion (ms)	Selection (ms)	Shell (ms)	Quick (ms)	Merge (ms)
5000	11.358	1.143	4.823	0.348	0.207	0.304
10000	57.110	4.411	18.908	0.774	0.406	0.635
20000	315.463	17.665	72.273	1.685	0.869	1.347
75000	6047.350	252.032	1014.186	7.459	3.724	5.870
150000	23992.005	1011.375	4030.296	16.013	7.882	12.201

Observations from Random Data:

The first thing I noticed is that bubble sort is extremely slow for large input sizes. At 150,000 elements, it took almost 24 seconds, which is much worse than the other algorithms. This makes sense because bubble sort has a time complexity of $O(n^2)$, so its runtime grows very quickly as n increases.

Selection sort also performed poorly. Even though it doesn't do many swaps, it still checks the whole unsorted part of the array every time, so it also ends up being $O(n^2)$. This is why it took over 4 seconds at 150,000 elements.

Insertion sort did better than bubble and selection sort, but it still became slow for large inputs. At 150,000 elements, it took about 1 second, which is still much slower than the faster algorithms. This matches the fact that insertion sort is also $O(n^2)$ in the average case.

The fastest algorithms were quicksort, merge sort, and shell sort. All three of these finished in under 20 milliseconds at 150,000 elements. Among them, quick sort was the fastest on random data, followed by merge sort and shell sort. This matches what we expect from algorithms with around O(n log n) performance.

Overall, the results on random data closely matched what we learned from asymptotic analysis in class.

Issue When Making the Graph (Random Data)

When trying to graph all six algorithms on the same chart using a normal y-axis, the times for bubble sort and selection sort were so large that the faster algorithms looked almost flat at the bottom of the graph.

This happens because the runtimes are on very different scales. Bubble sort takes thousands of milliseconds, while quick and merge sort take only a few milliseconds.

A good way to fix this would be to use a logarithmic y-axis or to make separate graphs for the slow and fast algorithms.

Performance on 10-Sorted (k-Sorted) Data

How k-sorted data was tested:

In this experiment, the same tests were repeated using 10-sorted arrays, where each element is at most 10 positions away from where it would be in a fully sorted array. The same array sizes and number of iterations were used.

Results on 10-Sorted Data

N	Bubble (ms)	Insertion (ms)	Selection (ms)	Shell (ms)	Quick (ms)	Merge (ms)
100	0.076	0.008	0.050	0.014	0.019	0.026
500	0.112	0.032	0.141	0.064	0.024	0.028
1000	0.130	0.036	0.204	0.045	0.037	0.058
2000	0.467	0.035	0.780	0.038	0.049	0.462

N	Bubble (ms)	Insertion (ms)	Selection (ms)	Shell (ms)	Quick (ms)	Merge (ms)
5000	2.827	0.042	4.588	0.103	0.116	0.297
10000	10.719	0.074	18.313	0.218	0.285	0.303
20000	41.065	0.147	74.350	0.457	0.541	0.651
75000	571.323	0.569	1044.976	1.927	5.207	3.502
150000	2257.751	1.145	4146.024	4.041	17.924	4.606

Observations from 10-Sorted Data

The biggest change was with insertion sort. On 10-sorted data, insertion sort became extremely fast. At 150,000 elements, it only took about 1.15 ms, compared to over 1000 ms on random data. This happens because insertion sort works very well when elements are already close to their correct positions.

Bubble sort also improved a lot compared to random data, but it was still much slower than insertion sort and the faster algorithms. Even with nearly sorted input, bubble sort still does many comparisons.

Selection sort barely improved at all. This makes sense because selection sort always searches through the unsorted part of the array, even if the array is almost sorted.

Shell sort benefited from k-sorted data and became even faster than before. This is because shell sort helps move elements closer to their correct positions early.

Quick sort became slower on the largest k-sorted input compared to random data. This can happen depending on how the pivot divides the array, especially when the input has a lot of structure.

Merge sort stayed very consistent and slightly improved. Since merge sort always does the same divide-and-merge process, the input order doesn’t affect it as much.

Graphing Issue (10-Sorted Data):

The same graphing issue appeared again with k-sorted data. Bubble and selection sort runtimes were still much larger than the others, which makes the faster algorithms hard to see on a normal graph.

Using a logarithmic y-axis would again be the best solution.

Final Summary

From these experiments, algorithms with O(n log n) performance are much better for large inputs than $O(n^2)$ algorithms. However, input order also matters a lot. Nearly sorted data makes insertion sort extremely fast, even though its worst-case complexity is still $O(n^2)$.

Overall, the experimental results matched what we expected from class, but they also showed how real performance can change depending on the type of input.