Alec Braynen

Project Report – Asymptotic Analysis

Algorithms

SelectionSort

InsertionSort

QuickSort (Median-Of-Three)

Table of Results

n	SS	SC	SR	IS	IC	IR	QS	QC	QR
10,000	12,031,800	12,999,486	38,913,953	442,049	352,817	10,707,998	1,319,228	15,986,414	2,333,850
20,000	41,693,628	40,613,080	138,290,884	600,890	548,296	39,598,788	2,093,417	44,099,847	3,251,282
30,000	86,024,854	89,883,921	326,753,119	770,063	739,059	76,747,556	4,070,535	88,125,409	4,934,125
40,000	159,944,288	163,009,976	610,368,976	1,296,421	1,080,854	152,190,015	4,172,547	144,507,126	7,646,035
50,000	232,269,192	189,592,810	875,267,093	1,158,780	1,230,482	181,979,644	7,653,047	274,798,748	7,644,771
60,000	260,935,369	289,577,655	1,234,335,213	1,543,114	1,569,379	372,185,927	5,795,165	298,088,783	6,283,801
70,000	342,789,128	467,855,677	1,771,468,869	1,994,998	1,591,870	405,828,411	6,514,783	425,533,448	8,465,264
80,000	588,033,091	458,391,745	2,131,607,524	1,612,358	1,634,749	476,395,013	10,074,942	597,987,927	9,561,688
90,000	652,958,200	608,626,894	2,762,717,483	1,751,160	1,743,022	779,553,597	8,485,138	738,577,109	18,004,883
100,000	788,829,019	778,215,293	3,441,758,886	2,305,411	2,067,298	753,742,201	9,749,057	805,776,802	11,948,632
500,000	23,534,044,262	22,656,891,090	84,443,269,553	3,083,676	5,491,995	19,903,664,776	16,584,676	22,464,468,626	53,720,963
1,000,000	92,974,419,027	77,863,586,301	339,069,768,490	4,191,179	6,128,612	72,873,461,361	25,604,135	117,143,215,177	80,272,113
10,000,000	11,837,829,395,962	12,677,105,774,797	35,979,879,862,256	22,297,410	12,898,920	10,150,245,951,629	203,349,539	12,583,715,704,150	812,806,901
100,000,000	N/A	N/A	N/A	112,497,095	83,643,119	N/A	1,970,242,283	N/A	8,834,612,560
1,000,000,000	OutOfMemoryError	OutOfMemoryError	OutOfMemoryError	OutOfMemoryError	OutOfMemoryError	OutOfMemoryError	OutOfMemoryError	OutOfMemoryError	OutOfMemoryError

Alec Braynen

Project Report

The goal of the project was to implement 3 sorting algorithms: selection sort, insertion sort and quick sort. For this project, the algorithms were implemented in java (jdk-11).

Implementing and testing the algorithms led to these results:

- For data that had 1,000,000 or less elements, InsertionSort was the best algorithm, best being defined as the fastest, for sorting those 1,000,000 or less elements
- For data between 1,000,000 to 1,000,000,000, InsertionSort was faster on sorted and constant data arrangements, however QuickSort was much faster on the random data arrangement. Random data is arguably much more important than, sorted or constant data for a sorting algorithm, so quicksort is ultimately the most useful algorithm in this range.
- Every iteration of the algorithms and data arrangements was sorted correctly.

Additionally, and unfortunately, I was unable to run any of the algorithms on the billion element arrays. The machine and the java environment where the code was tested, didn't allow for array sizes as big as 1,000,000,000 elements and instead returned out of memory errors. These errors occurred because the amount of contiguous memory the program needed in those scenarios was more than what was physically available.

However, more positively, the project's results mostly lined up with what I expected and what we learned in class. Quicksort was much better than InsertionSort and SelectionSort on randomly arranged data, and the disparity and performance continued to grow as the data size grew. This lines up with the time complexities of these algorithms which are, $\theta(n^2)$, $\theta(n^2)$, $\theta(nlgn)$ for, SelectionSort, InsertionSort and QuickSort respectively (average case).

InsertionSort outperformed QuickSort on constant and sorted data. Constant and sorted data allowed InsertionSort to operate more closely to its best-case time complexity of $\Omega(n)$, whereas QuickSort's best case time complexity is $\Omega(nlgn)$ which lines up with the disparity in performance. SelectionSort's time complexity is $O(n^2)$ in the best worst, and average case, which also lines up with the results. SelectionSort was either the slowest or second slowest algorithm in every comparable case.

Surprisingly, QuickSort's performance on constant data was much worse than I thought it would be. However, after further analyzing the code and gaining a better understanding of quicksort, it made

sense that its performance deteriorated so greatly with this data. Constant data creates a scenario where the pivot element can't divide the array. This leads to the algorithm iterating through every element in its arbitrary split which increases the time complexity greatly. I ended up having to modify my original QuickSort implementation to help the JVM compiler complete the code execution. The original implementation was filling the stack and crashing the program due to QuickSort's behavior with constant data. The modification was to implement a check that made the recursive call execute the smaller split in the data first. This allowed some of the recursive calls to reach the bottom of their stack trace faster and ease pressure on the overall stack of the program.

Finally, the results were a bit better than the Justification of Big-Oh slide we were shown in class:

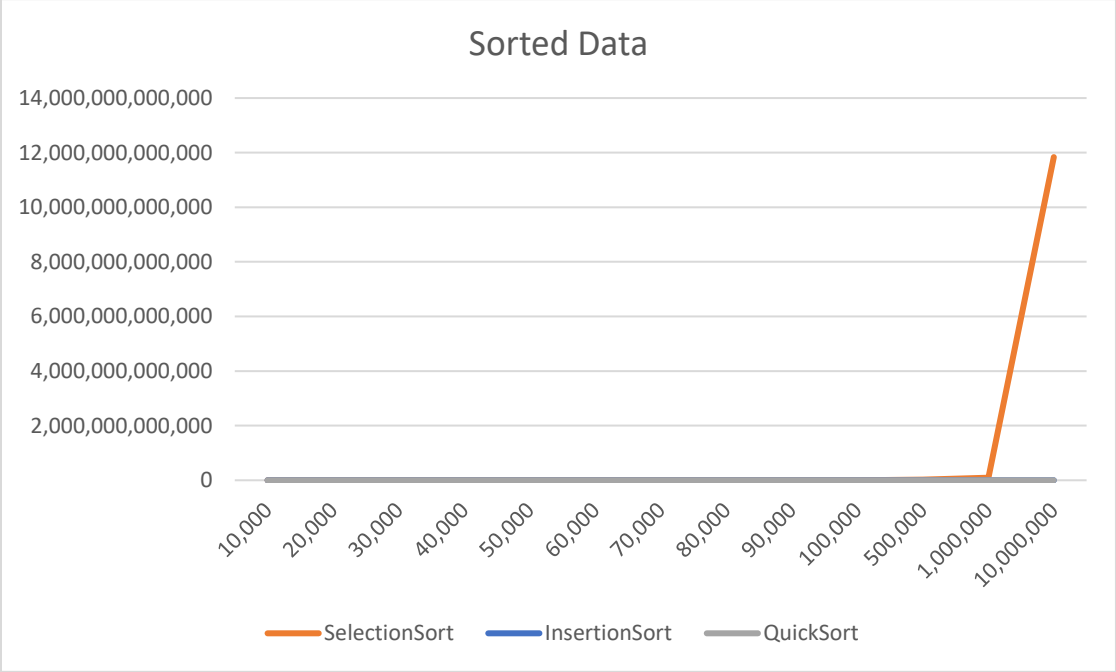
n=	lg(n)	n	$n$lg(n)	n^2	n^3	2^n	$n!$
10	3 ns	10 ns	33 ns	100 ns	1 µs	1 µs	3.6 ms
20	4 ns	20 ns	86 ns	400 ns	8 µs	1 ms	77 yrs
30	5 ns	30 ns	147 ns	900 ns	27 µs	1 s	
40	5 ns	40 ns	213 ns	1.6 µs	64 µs	18.3 min	
50	6 ns	50 ns	282 ns	2.5 µs	125 µs	13 days	
100	7 ns	100 ns	644 ns	10 µs	1 ms		
1,000	10 ns	1 µs	9.97 µs	1 ms	1 s		
1,000,000	20 ns	1 ms	19.9 ms	16.7 min	31.7 yrs		
1,000,000,000	30 ns	1 s	29.9 s	31.7 yrs			

n^2 algorithms took approximately 1.5 – 2 minutes at one million elements in this implementation of the algorithms. This is probably due to advancements in CPU technology, architecture, and design. At ten million elements, SelectionSort on random data elements and QuickSort on constant data elements took long enough that they were infeasible to wait to complete. Logically, at one hundred million elements SelectionSort became completely infeasible to wait for, while InsertionSort and QuickSort became infeasible on random and constant data respectively. Those data types cause those algorithms to run tending to their worst case.

Alec Braynen

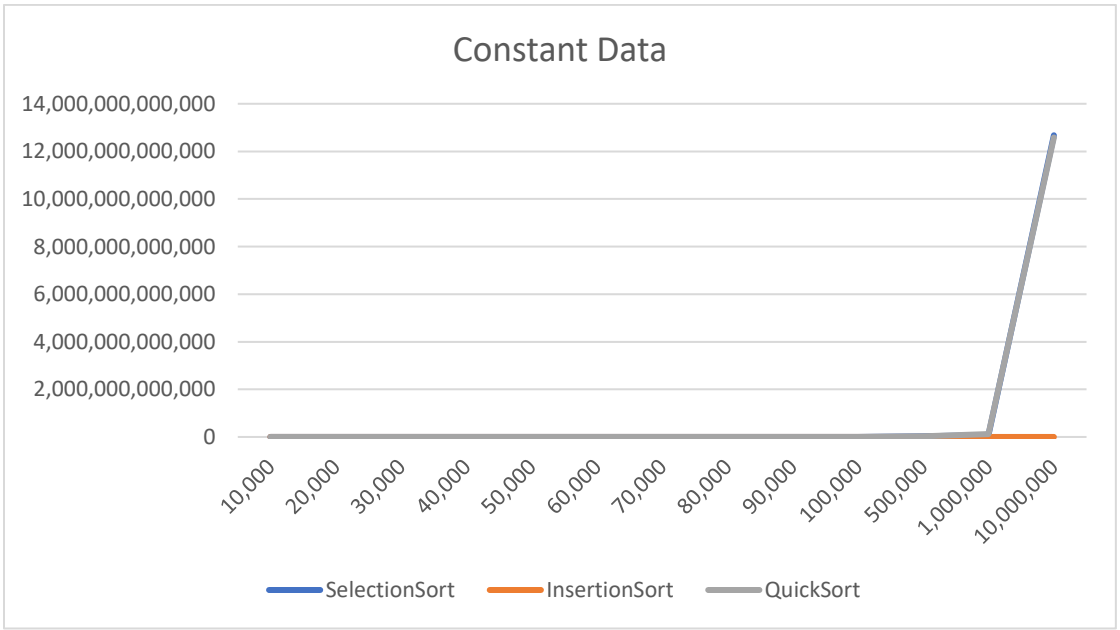
Charts

Here are charts of the results:



[***Time Taken*** in nanoseconds (y-axis) vs ***Number of Elements*** (x-axis)]

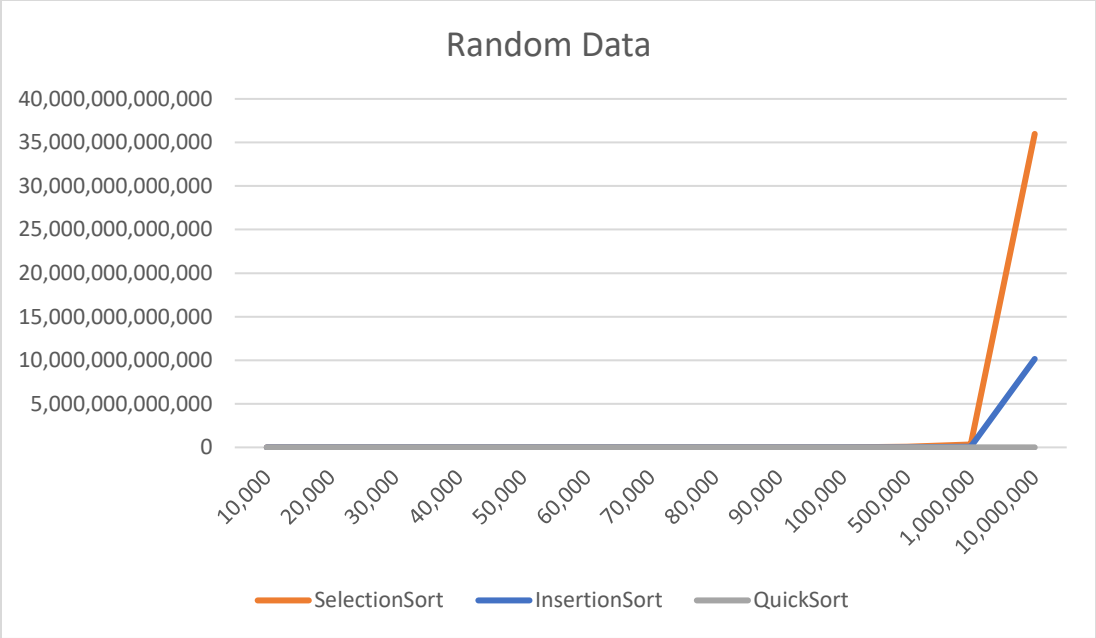
For sorted data, SelectionSort's execution time grows much faster than QuickSort's or InsertionSort's



[***Time Taken*** in nanoseconds (y-axis) vs ***Number of Elements*** (x-axis)]

For constant data, SelectionSort's and QuickSort's execution time grows much faster than InsertionSort's (Both SelectionSort and QuickSort take about 4 hours at 10 million elements here)

Alec Braynen



[***Time Taken*** in nanoseconds (y-axis) vs ***Number of Elements*** (x-axis)]
For random data, SelectionSort's and InsertionSort's execution time grows much faster than QuickSort's

Conclusion

SelectionSort's, InsertionSort's and QuickSort's average time complexities are: $\theta(n^2)$, $\theta(n^2)$, $\theta(nlgn)$ respectively. This lines up with the results demonstrated by this project. Particularly, with random data these time complexities match accurately.

SelectionSort's, InsertionSort's and QuickSort's best case time complexities are: $\Omega(n^2)$, $\Omega(n)$,$\Omega(nlgn)$ respectively. This also lines up with the results demonstrated by this project. SelectionSort's performance stayed poor no matter the data arrangement, while InsertionSort's execution time improved greatly on constant and sorted data, and QuickSort's on sorted data.

In the worst case, particularly for QuickSort, its time complexity became $O(n^2)$. This shows a big reduction in performance for QuickSort on constant data, where the algorithm had to iterate through every element in the array n*n times.