Comparison of Sorting Algorithms

Alex Merk Nicole Kauer

Module: Simulation of Sorting Techniques
Course: TCSS 598 Masters Seminar

Lecturer: Mohamed Ali, Ph.D.

October 24, 2018

Abstract

Five sorting techniques were implemented and compared for run time and memory use against data size and degree of sortedness. The five sorting techniques were Insertion Sort, Bubble Sort, Merge Sort, Selection Sort, and Quicksort. Four data sets were used to test each sorting algorithm, two simulated with mathematical equations and two real, which were derived from US Census Data and Climate Data Online. The experiments suggest that Mergesort and Quicksort are the fastest sorting algorithms for large datasets over all others. However, both of these algorithms take a significant amount of memory, with Mergesort using the most. Insertion Sort, Bubble Sort, and Selection Sort use a negligible amount of memory compared to the Mergesort and Quicksort, with Insertion Sort being the fastest and Selection Sort being the slowest of the three algorithms. The degree of sortedness did not have a drastic effect on the runtimes, nor the memory use with real data. Simulated data showed faster runtimes for Insertion Sort and Bubble Sort with data that was 60% sorted or higher.

Contents

1 Introduction

While all sorting algorithms accomplish the same goal, choosing the best algorithm for a given sorting task can be challenging. Some sorting algorithms are known to be slow for large datasets, while other sorting algorithms are known to take a significant amount of memory. In this experiment, we tested five well-known sorting algorithms to find which ones performed the best in memory use and speed. As the data size, sortedness, and type can be critical to the decision making process, two types of data and four total datasets were chosen. Additionally, the sortedness was simulated for each dataset to get an idea of how it affects the sorting algorithm behavior.

2 Experiment

Five algorithms were tested with four sets of data, two simulated and two real. The five algorithms tested were Insertion Sort, Bubble Sort, Selection Sort, Mergesort, and Quicksort. An overview of each algorithm is given in Section 2.2 on page 4. An overview of each dataset can be found in Section 2.1 below. The tests consisted of measuring the runtime and memory use for each algorithm while varying the degree of sortedness and the data size. Each experiment was run five times and the average runtime and memory use was calculated. The experiment was coded in Java 8.

2.1 Data

Four sets of data were tested, two real and two simulated, as shown in Table 1. A description of where the dataset came from and how it was derived can be found in the respective section. Each dataset was tested with five different sizes and degrees of sortedness, shown in Table 2. The degree of sortedness used is discussed in section 2.3 on page 6

Table 1: Data Overview

Category	**Name**	**Source**
Simulated	Reverse	Reverse sorted data obtained from a function
	Periodic	Periodic data from a simple sine function
Real	Census	Wages for a subset of WA state residents in 2010
	Weather	Geopotential heights from 2018 weather balloons

Table 2: Size and Degree of Sortedness Tested

Size	20,000	40,000	60,000	80,000	100,000
Degree of Sortedness	20%	40%	60%	80%	100%

2.1.1 Simulated Data

The simulated data was created using simple algorithm. The Reverse set simply creates a dataset sorted in reverse order. The Periodic set was created with the function $\lfloor x^3 sin(x) \rfloor + \lceil x^2 cos(x) \rceil$.

2.1.2 Real Data

The two subsets of real data, Census and Climate, came from the US Census Bureau and the National Oceanic and Atmospheric Administration, respectively. We created the Census dataset by taking 100,000 of the wage values from the Washington State wages subset. (US Census Bureau 2011) The Weather dataset was created from 100,000 of the geopotential heights in the Weather Balloon dataset for the current year. (National Centers for Environmental Information 2018)

2.2 Sorting Algorithms

Five different sorting techniques were used for each experiment. Table 3 shows each sorting algorithm and the average and worst known runtimes in terms of Big-Oh. (Rowell 2018)

Table 3: Sorting Algorithm Time and Space Complexity

Algorithm	**Average** $O(f(x))$	**Worst** $O(f(x))$	$O(f(x))$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Mergesort	$O(n \log_2 n)$	$O(n^2)$	$O(n)$
Quicksort	$O(n \log_2 n)$	$O(n^2)$	$O(1)$

2.2.1 Insertion Sort

Insertion Sort treats an array of values as if they exist in two separate arrays, one that is sorted, A, and one that is unsorted, B. Initially, the first element in the original array is chosen to be the first element A and is considered sorted given that there is only a single element in A. The algorithm then takes the first element in B, finds the location where the element belongs in A and inserts the element there, growing the sorted array, A, one element at time. Even though the algorithm treats the data as if it were in two separate arrays, Insertion Sort is an in-place sort and the delineation of the sorted and unsorted arrays is handled by index tracking.

Insertion Sort has both a worst case scenario and a best case scenario for runtime. In the worst case, the original data is sorted in the reverse order. In this instance, each element in the unsorted array would need to be checked against every element in the sorted array, resulting in a $O(n^2)$. However, the best case scenario is when the original data is already sorted. In this case, each element in the unsorted array is only checked against the first element in the sorted array, resulting in a $O(n)$. Given that that the data would not be expected to be sorted most of the time, the average case would be $O(n^2)$. Additionally, Insertion Sort only uses $O(1)$ extra space.

2.2.2 Selection Sort

Selection Sort treats an array of values as if they exist in two separate arrays, one that is sorted, A, and one that is unsorted, B. Initially, A is empty and B holds all of the initial values. Selection Sort takes one element from B, either the minimum or maximum value in B depending on the sorting order desired, and adds the element to the end of array A. The sort is done once B is empty. While the algorithm treats the data as if it were in two arrays, this is an in-place sort and the delineation between the sorted and unsorted array is taken care of by index tracking.

Due to the need to search the unsorted array for the minimum or maximum value on each iteration, Selection Sort runs in $O(n^2)$. There is no best case scenario for Selection Sort, making this an inefficient algorithm with an average case of $O(n^2)$ Given that Selection Sort is an in-place sorting algorithm, this algorithm uses only $O(1)$ extra space.

2.2.3 Mergesort

Mergesort uses a divide-and-conquer recursive approach to sorting an array of data, splitting the array in half and sending each half recursively back through Mergesort until both arrays have only a single element each. After the arrays are split down to a single element, Mergesort sends those elements recursively back through a Merge function. At each Merge, arrays from the Mergesort recursion are compared and sorted into the correct order.

The runtime for Mergesort is based on two factors, the dividing of the array and the merging the smaller arrays back into a single array. Due to using recursion to divide the original array in half until the array is separated into its single elements, the initial recursion takes $O(\log_2 n)$. Merging the elements back into a single array takes $O(n)$. Together, the runtime for Mergesort is $O(n \log_2 n)$ for all cases. Since Mergesort is not an in-place sorting algorithm, it takes $O(n)$ extra space.

2.2.4 Bubble Sort

Bubble Sort is an in-place sorting algorithm that iterates over the original array, comparing adjacent elements and swapping if they are out of order. In each pass through the array, the elements "bubble" up or down towards their correct location in the array. The algorithm continues iterating over the array until it gets through the entire array once without a single swap.

Bubble Sort's worst case scenario would be when the elements are in reverse order and the best case would be when the array was already sorted. However, most of the time, the array would not be expected to be sorted, making the worst and average runtime be $O(n^2)$. Bubble Sort only uses $O(1)$ extra space given that it is an in-place sorting algorithm.

2.2.5 Quicksort

Quicksort is similar to Mergesort in that the algorithm attempts to divide the original array into two halves and recursively sort each half. This is done by choosing a pivot element. Each element smaller than the pivot point is placed on the left side of the pivot and elements larger than the pivot are kept on the right side. Each half of the array is then recursively sorted via this same Quicksort algorithm until the array to be sorted consists of a single element. Quicksort is an in-place sorting algorithm, with the recursive calls sending indices that indicate the first element in the left array, the pivot index, and the last element in the right array. There are many ways of choosing the pivot element in the array; some algorithms choose a random pivot, some choose the first or last element, some choose the middle value of three elements, and so on.

In Quicksort, the best case scenario where the pivot is the middle element in every recursive call results in $O(n \log_2 n)$, but the worst case scenario where the pivot is the highest or lowest element in each array results in $O(n^2)$. The best case scenario can become the average case scenario, however, if the pivot is chosen in such a way that the likelihood of it being the worst pivot choice is low. Given that Quicksort is an in-place sort, it only uses $O(1)$ extra space.

2.3 Degree of Sortedness

The degree of sortedness is an indication of how sorted an array is, and can be calculated in a variety of ways, including counting the inversions and the length of the longest sorted section of the array. (Gopalan 2007) For this experiment, we chose that degree of sortedness would be the percentage of the array that is sorted, from the start of the array. For the degree of sortedness experiments, Selection Sort was used to achieve a specific percentage of sorted elements starting at the beginning of each array. Memory and time data was not collected during the initial presorting process.

3 Results

All of the result data can be found in Appendix A on page 14. Graphs created from the results can be found in the respective sections for data type and source. Each data source has four graphs, two that map data size against time and memory use, and two that map degree of sortedness against time and memory use.

3.1 Simulated Data

3.1.1 Reverse Sorted Data

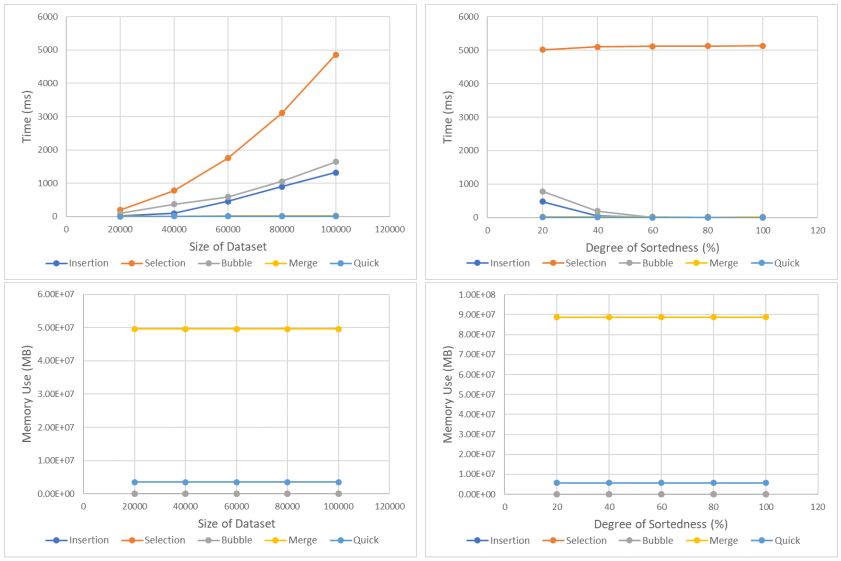


Figure 1: Time and space complexity graphs for Reverse Sorted Data. The left graphs are data size versus time and memory use and the right graphs are degree of sortedness versus time and memory use.

The graphs shown in Figure 1 show the results for each sorting algorithm used with the US Census data. Overall, the memory use was constant for all agorithms regardless of degree of sortedness or size, with Mergesort and Quicksort using the highest and second highest memory, respectively. However, Mergesort used significantly more memory than any other algorithm. When the sortedness of the dataset was held constant and the number of elements varied, Mergesort and Quicksort performed the best. Insertion Sort and Bubble Sort had similar runtimes, but still performed progressively worse as the data size increased. Selection Sort performed the worst overall. When the degree of sortedness was varied, Selection Sort performed the same regardless of sortedness, while the other algorithms performed progressively better with higher sortedness. Overall, all algorithms except for Selection Sort had relatively low runtimes even when the degree of sortedness was low.

3.1.2 Periodic Data

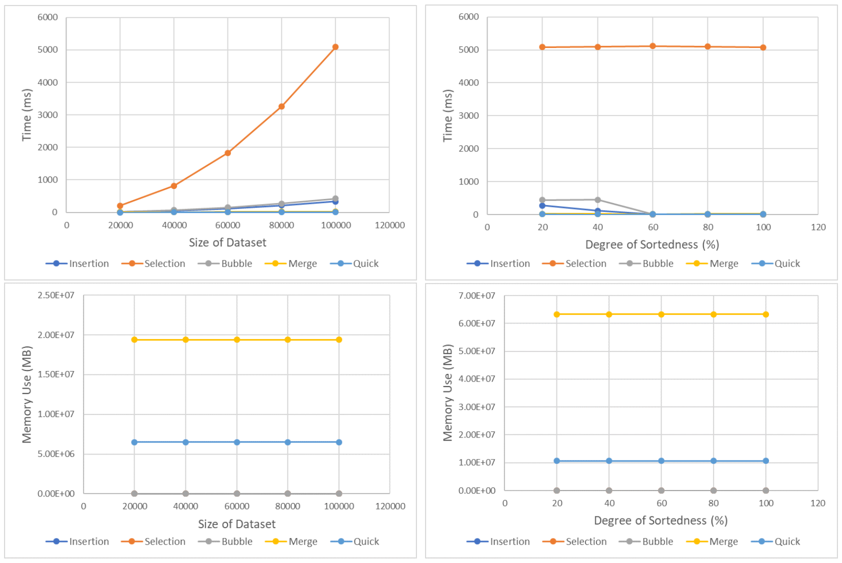


Figure 2: Time and space complexity graphs for Periodic Data. The left graphs are data size versus time and memory use and the right graphs are degree of sortedness versus time and memory use.

The graphs shown in Figure 2 show the results for each sorting algorithm used with the US Census data. Overall, the memory use was constant for all agorithms regardless of degree of sortedness or size, with Mergesort and Quicksort using the highest and second highest memory, respectively. However, Mergesort used significantly more memory than any other algorithm. When the sortedness of the dataset was held constant and the number of elements varied, all algorithms except Selection Sort performed well and similar to each other. Selection Sort performed the worst overall. When the degree of sortedness was varied, Selection Sort performed the same regardless of sortedness, while the other algorithms performed progressively better with higher sortedness. Overall, all algorithms except for Selection Sort had relatively low runtimes even when the degree of sortedness was low.

3.2 Real Data

3.2.1 US Census Data

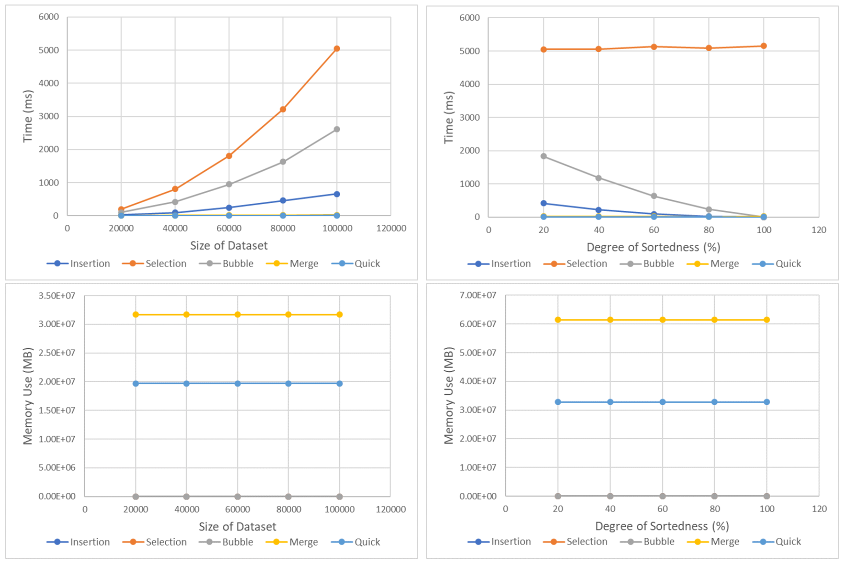


Figure 3: Time and space complexity graphs for Census Wage Data. The left graphs are data size versus time and memory use and the right graphs are degree of sortedness versus time and memory use.

The graphs shown in Figure 3 show the results for each sorting algorithm used with the US Census data. Overall, the memory use was constant for all agorithms regardless of degree of sortedness or size, with Mergesort and Quicksort using the highest and second highest memory, respectively. When the sortedness of the dataset was held constant and the number of elements varied, Mergesort and Quicksort performed the best. Insertion Sort, Bubble Sort, and Selection Sort perform progressively worse as the data size increased. When the degree of sortedness was varied, Selection Sort performed the same regardless of sortedness, while the other algorithms performed progressively better with higher sortedness.

3.2.2 Weather Balloon Data

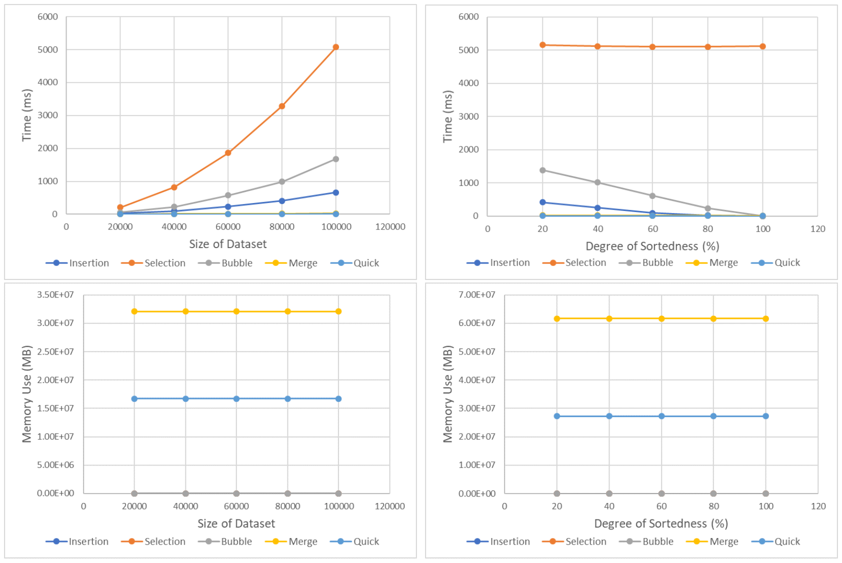


Figure 4: Time and space complexity graphs for Weather Balloon Data. The left graphs are data size versus time and memory use and the right graphs are degree of sortedness versus time and memory use.

The graphs shown in Figure 4 show the results for each sorting algorithm used with the US Census data. Overall, the memory use was constant for all agorithms regardless of degree of sortedness or size, with Mergesort and Quicksort using the highest and second highest memory, respectively. When the sortedness of the dataset was held constant and the number of elements varied, Mergesort and Quicksort performed the best. Insertion Sort, Bubble Sort, and Selection Sort perform progressively worse as the data size increased. When the degree of sortedness was varied, Selection Sort performed the same regardless of sortedness, while the other algorithms performed progressively better with higher sortedness.

4 Analysis

4.1 Degree of Sortedness

4.1.1 Simulated Data

The data's degree of sortedness played a small role in runtimes and memory use for simulated data. As seen in the graphs in the Results section, Selection Sort took approximately the same time regardless of whether the data was sorted 20% or 100%. While Bubble Sort and Insertion Sort took longer to sort unsorted data than Mergesort and Quicksort, all four algorithms had similar results once the data was 60% sorted or higher. Memory use was nearly constant across the board for all algorithms, with Mergesort using the most memory, by far, as expected. If given a large, simulated dataset with a high degree of sortedness, any algorithm tested other than Selection Sort would be useful. However, given a constraint of memory use, Mergesort would be the worst choice.

4.1.2 Real Data

Real data performed better with the sorting algorithms overall, with regards to expected outcomes for runtime and memory use. As seen in the Results section, Mergesort and Quicksort perform consistently fast and Selection Sort performed consistently slow, regardless of sortedness. Insertion Sort and Bubble Sort showed a more realistic trend with real data as opposed to simulated data, being slower with less sorted data and faster with more sorted data. The more realistic trend is most likely due to the inherent randomness in real data that was not achieved with simulated data. Memory use for Mergesort was still higher than Quicksort, but by a smaller margin than with simulated data. All three other sorting algorithms used nearly no extra memory in comparison to Mergesort and Quicksort. Given a large real dataset with a low degree of sortedness, Mergesort and Quicksort would be the best options with Quicksort being the better choice for low memory use, if not by much. With a higher degree of sortedness, Insertion Sort becomes an efficient candidate with fast runtimes and negligible memory use compared to Mergesort and Quicksort.

4.2 Data Size

4.2.1 Simulated Data

The data size largely affected the runtime of each sorting algorithm on the simulated data, while not affecting memory use, as seen in the Results section graphs. Mergesort and Quicksort performed consistently fast regardless of data size, as expected. Bubble Sort and Insertion Sort had similar runtimes that grew as the data size grew. Selection Sort showed a clear exponential relationship between data size and runtime. Oddly, while the memory use for any data given data set tested shows a similar pattern, the Quicksort algorithm used significantly less memory than Mergesort for reverse sorted data. Additionally, as mentioned in the Experiment section, Mergesort's memory use is known to grow with the size of the data. However, our results show a constant relationship; this may be due to the relatively small size of the dataset using a negligible amount of memory compared to the stack needed to recursively run the Mergesort algorithm. As expected, the three other algorithms, Insertion Sort, Bubble Sort, and Selection Sort used nearly no extra memory in comparison to Mergesort and Quicksort. Given a large simulated dataset, Mergesort and Quicksort are the best algorithms to use, with Quicksort being slightly better due to lower memory use. In addition, if the dataset is also periodic in nature, Insertion Sort and Bubble Sort are potentially efficient sorting solutions with low overhead memory needed.

4.2.2 Real Data

As with simulated data, the data size largely affected the runtime of each sorting algorithm on real data, if not memory use. Real world data also brought out the more realistic trend that was expected for runtime of each algorithm. As with all other datasets, Selection Sort performed the worst overall, growing exponentially slower with increasing data size. Mergesort and Quicksort outperformed all other algorithms, being consistently fast even with large data sizes. Insertion Sort and Bubble Sort came in third and fourth place, respectively, with Bubble Sort following a significantly steeper curve than Insertion Sort. Memory use for all algorithms on real data was consistent with all other experiments. As discussed in the previous section, Mergesort should have shown a linear growth rate in memory use, but this trend does not appear in the data and may be attributed to the large stack size compared to the relatively small data size. Given a real, unsorted dataset, Mergesort and Quicksort are the clear winners, with Quicksort pulling ahead slightly due to lower memory use.

5 Conclusion

Overall, Mergesort and Quicksort were consistently faster than all other algorithms in nearly every experiment, with Insertion Sort being a potential candidate for sorting given data with a high degree of sortedness or small datasize. Selection Sort was shown to be the worst sorting algorithm, followed by Bubble Sort. Real data was a better indicator of the predicted trends in runtime across the five algorithms, most likely due to the more random nature of the data. Memory use for all five algorithms showed the same trends regardless of experiment with Mergesort taking the most memory, Quicksort taking slightly less, and the rest taking nearly no extra memory.

References

Gopalan 2007
GOPALAN, et a. Parikshit: Estimating the sortedness of data stream. In: *Proceeding SODA '07 Proceedings of the eigtheenth annual ACM-SIAM symposium on Discrete algorithms* (2007), pages 318–327

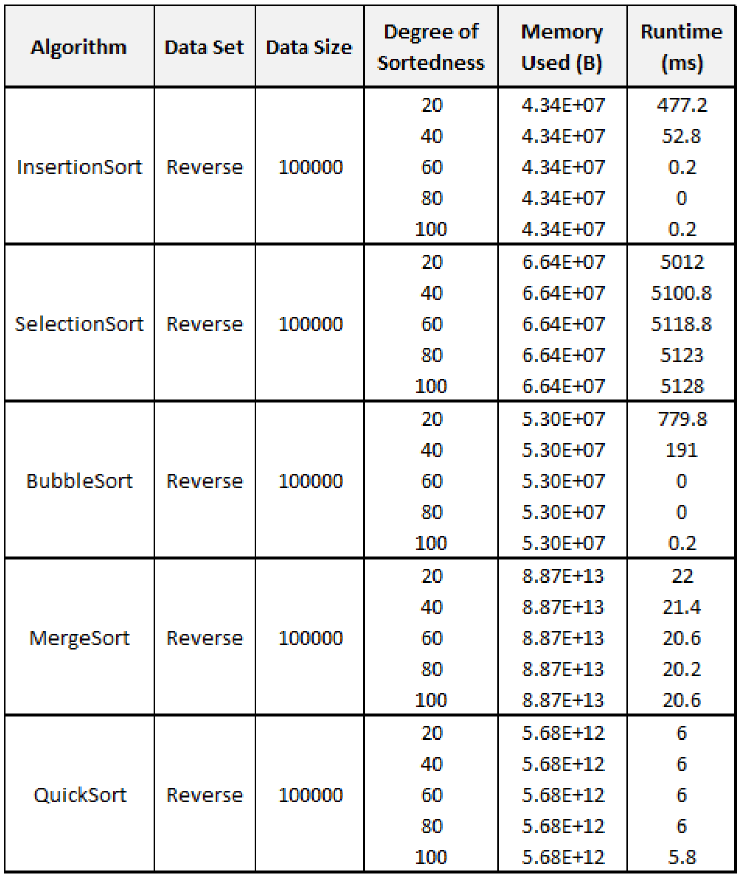
National Centers for Environmental Information 2018
NATIONAL CENTERS FOR ENVIRONMENTAL INFORMATION: *Integrated Global Radiosonde Archive*. `https://www.ncdc.noaa.gov/data-access/weather-balloon/integrated-global-radiosonde-archive`. Version: 2018. – Last accessed 18 October 2018

Rowell 2018
ROWELL, et a. E: *Big-O Algorithm Complexity Cheat Sheet*. `http://bigocheatsheet.com/`. Version: 2018. – Last accessed 19 October 2018

US Census Bureau 2011
US CENSUS BUREAU: *Demographic Profile for Washington State*. `https://www2.census.gov/census_2010/03-Demographic_Profile/Washington/`. Version: 2011. – Last accessed 18 October 2018

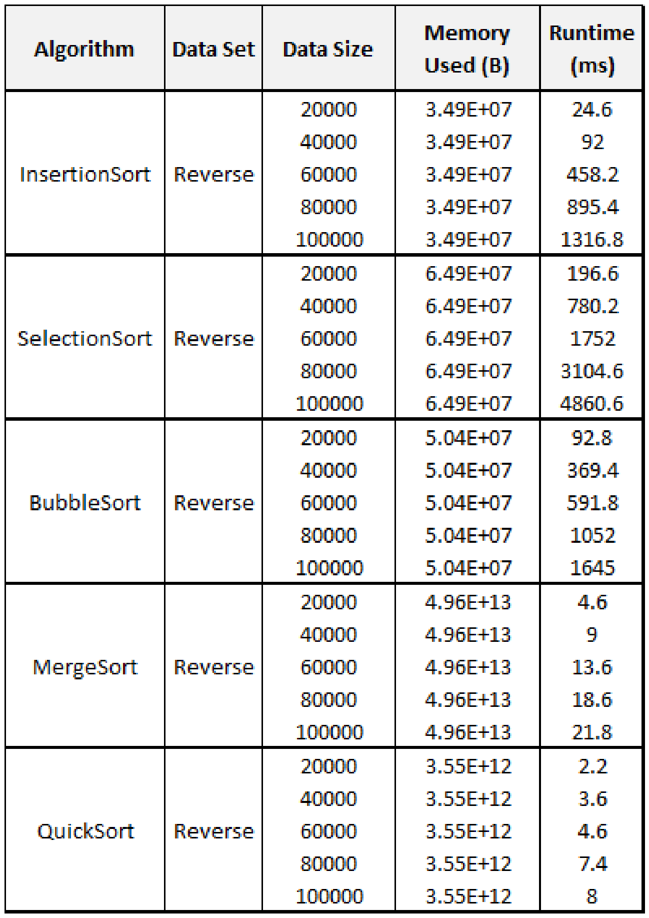
Appendices

A Results Data



Algorithm	Data Set	Data Size	Degree of Sortedness	Memory Used (B)	Runtime (ms)
InsertionSort	Reverse	100000	20	4.34E+07	477.2
			40	4.34E+07	52.8
			60	4.34E+07	0.2
			80	4.34E+07	0
			100	4.34E+07	0.2
SelectionSort	Reverse	100000	20	6.64E+07	5012
			40	6.64E+07	5100.8
			60	6.64E+07	5118.8
			80	6.64E+07	5123
			100	6.64E+07	5128
BubbleSort	Reverse	100000	20	5.30E+07	779.8
			40	5.30E+07	191
			60	5.30E+07	0
			80	5.30E+07	0
			100	5.30E+07	0.2
MergeSort	Reverse	100000	20	8.87E+13	22
			40	8.87E+13	21.4
			60	8.87E+13	20.6
			80	8.87E+13	20.2
			100	8.87E+13	20.6
QuickSort	Reverse	100000	20	5.68E+12	6
			40	5.68E+12	6
			60	5.68E+12	6
			80	5.68E+12	6
			100	5.68E+12	5.8

Figure 5: Degree of Sortedness experiment results with reverse sorted data.

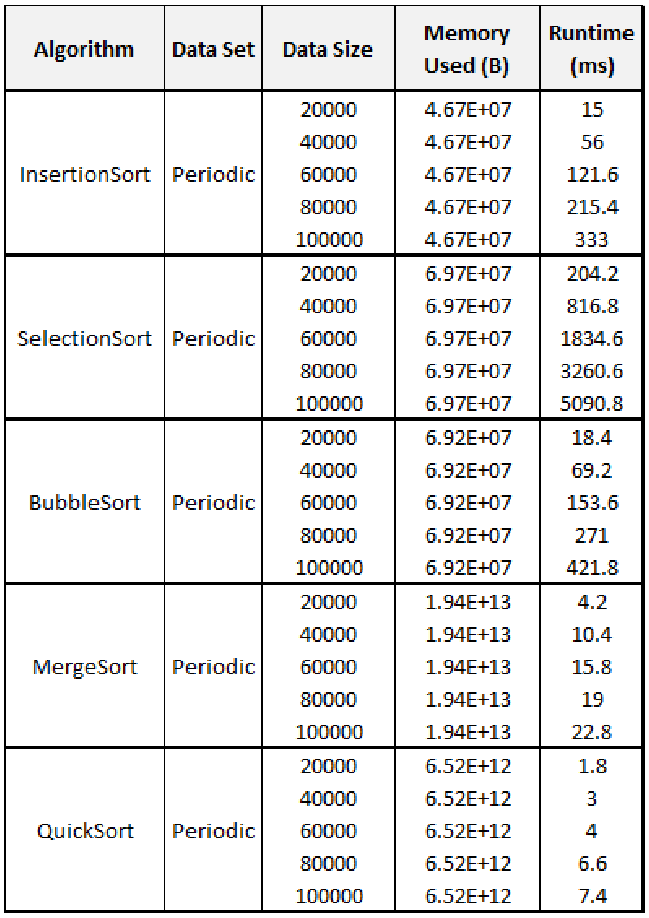


Algorithm	Data Set	Data Size	Memory Used (B)	Runtime (ms)
InsertionSort	Reverse	20000	3.49E+07	24.6
		40000	3.49E+07	92
		60000	3.49E+07	458.2
		80000	3.49E+07	895.4
		100000	3.49E+07	1316.8
SelectionSort	Reverse	20000	6.49E+07	196.6
		40000	6.49E+07	780.2
		60000	6.49E+07	1752
		80000	6.49E+07	3104.6
		100000	6.49E+07	4860.6
BubbleSort	Reverse	20000	5.04E+07	92.8
		40000	5.04E+07	369.4
		60000	5.04E+07	591.8
		80000	5.04E+07	1052
		100000	5.04E+07	1645
MergeSort	Reverse	20000	4.96E+13	4.6
		40000	4.96E+13	9
		60000	4.96E+13	13.6
		80000	4.96E+13	18.6
		100000	4.96E+13	21.8
QuickSort	Reverse	20000	3.55E+12	2.2
		40000	3.55E+12	3.6
		60000	3.55E+12	4.6
		80000	3.55E+12	7.4
		100000	3.55E+12	8

Figure 6: Data size experiment results with reverse sorted data.

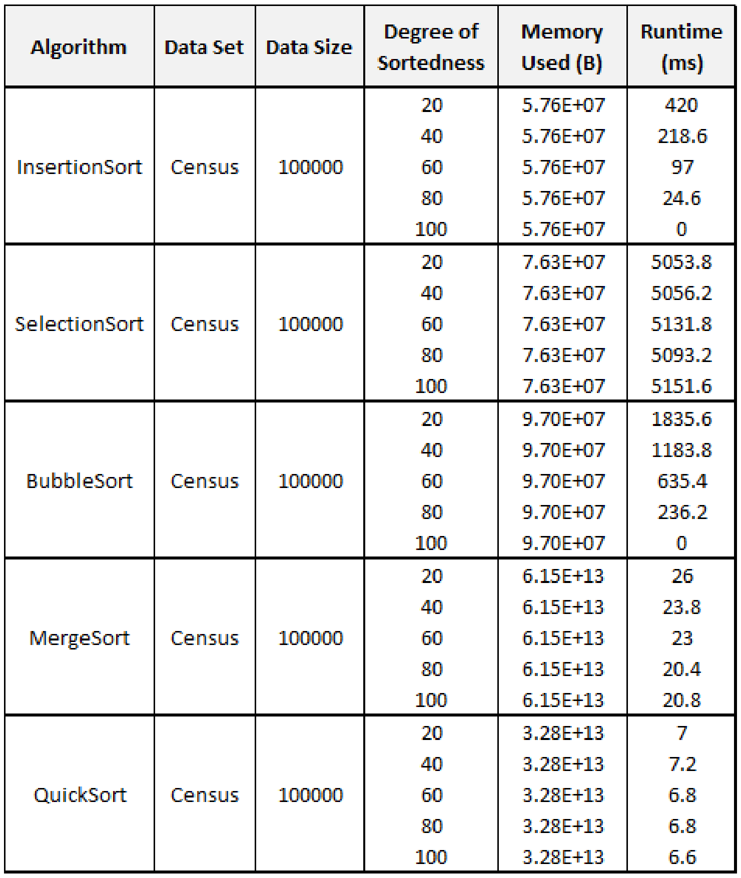
Algorithm	Data Set	Data Size	Degree of Sortedness	Memory Used (B)	Runtime (ms)
InsertionSort	Periodic	100000	20	4.67E+07	275
			40	4.67E+07	116.6
			60	4.67E+07	0.2
			80	4.67E+07	0
			100	4.67E+07	0
SelectionSort	Periodic	100000	20	6.99E+07	5088.4
			40	6.99E+07	5093.6
			60	6.99E+07	5118.6
			80	6.99E+07	5098.2
			100	6.99E+07	5080.2
BubbleSort	Periodic	100000	20	6.92E+07	438.4
			40	6.92E+07	447.8
			60	6.92E+07	0
			80	6.92E+07	0
			100	6.92E+07	0
MergeSort	Periodic	100000	20	6.33E+13	21.8
			40	6.33E+13	22.4
			60	6.33E+13	21
			80	6.33E+13	21.4
			100	6.33E+13	21.2
QuickSort	Periodic	100000	20	1.07E+13	7.8
			40	1.07E+13	7.4
			60	1.07E+13	7.8
			80	1.07E+13	7.8
			100	1.07E+13	7.6

Figure 7: Degree of Sortedness experiment results with periodic data.



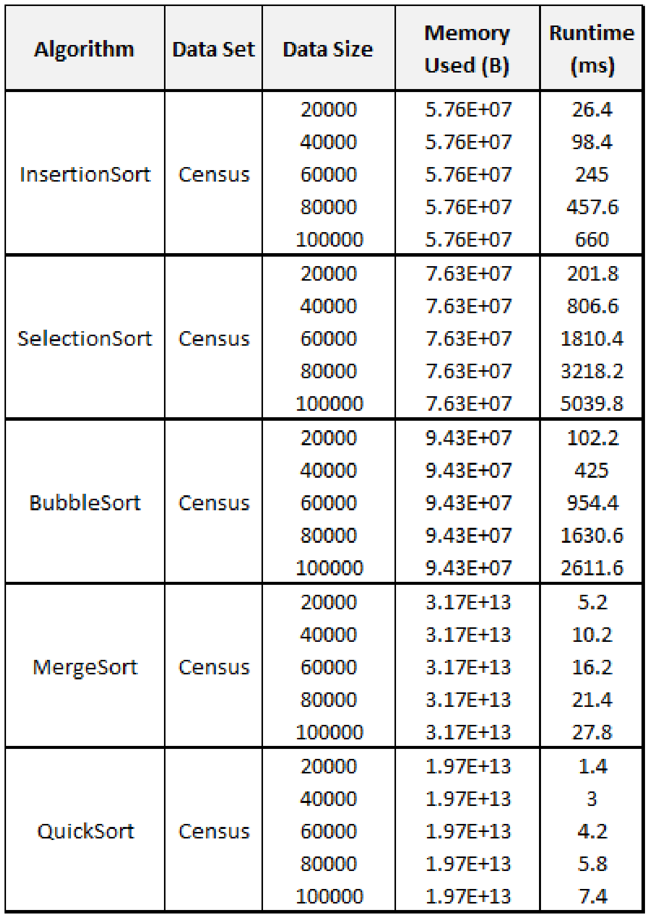
Algorithm	Data Set	Data Size	Memory Used (B)	Runtime (ms)
		20000	4.67E+07	15
		40000	4.67E+07	56
InsertionSort	Periodic	60000	4.67E+07	121.6
		80000	4.67E+07	215.4
		100000	4.67E+07	333
		20000	6.97E+07	204.2
		40000	6.97E+07	816.8
SelectionSort	Periodic	60000	6.97E+07	1834.6
		80000	6.97E+07	3260.6
		100000	6.97E+07	5090.8
		20000	6.92E+07	18.4
		40000	6.92E+07	69.2
BubbleSort	Periodic	60000	6.92E+07	153.6
		80000	6.92E+07	271
		100000	6.92E+07	421.8
		20000	1.94E+13	4.2
		40000	1.94E+13	10.4
MergeSort	Periodic	60000	1.94E+13	15.8
		80000	1.94E+13	19
		100000	1.94E+13	22.8
		20000	6.52E+12	1.8
		40000	6.52E+12	3
QuickSort	Periodic	60000	6.52E+12	4
		80000	6.52E+12	6.6
		100000	6.52E+12	7.4

Figure 8: Data size experiment results with periodic data.



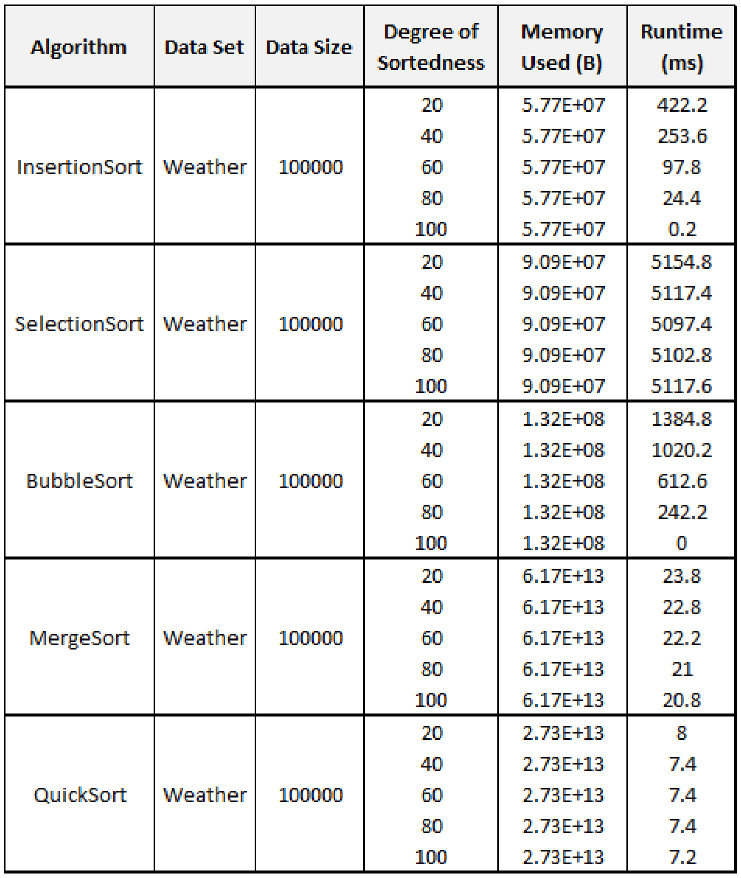
Algorithm	Data Set	Data Size	Degree of Sortedness	Memory Used (B)	Runtime (ms)
InsertionSort	Census	100000	20	5.76E+07	420
			40	5.76E+07	218.6
			60	5.76E+07	97
			80	5.76E+07	24.6
			100	5.76E+07	0
SelectionSort	Census	100000	20	7.63E+07	5053.8
			40	7.63E+07	5056.2
			60	7.63E+07	5131.8
			80	7.63E+07	5093.2
			100	7.63E+07	5151.6
BubbleSort	Census	100000	20	9.70E+07	1835.6
			40	9.70E+07	1183.8
			60	9.70E+07	635.4
			80	9.70E+07	236.2
			100	9.70E+07	0
MergeSort	Census	100000	20	6.15E+13	26
			40	6.15E+13	23.8
			60	6.15E+13	23
			80	6.15E+13	20.4
			100	6.15E+13	20.8
QuickSort	Census	100000	20	3.28E+13	7
			40	3.28E+13	7.2
			60	3.28E+13	6.8
			80	3.28E+13	6.8
			100	3.28E+13	6.6

Figure 9: Degree of Sortedness experiment results with census data.



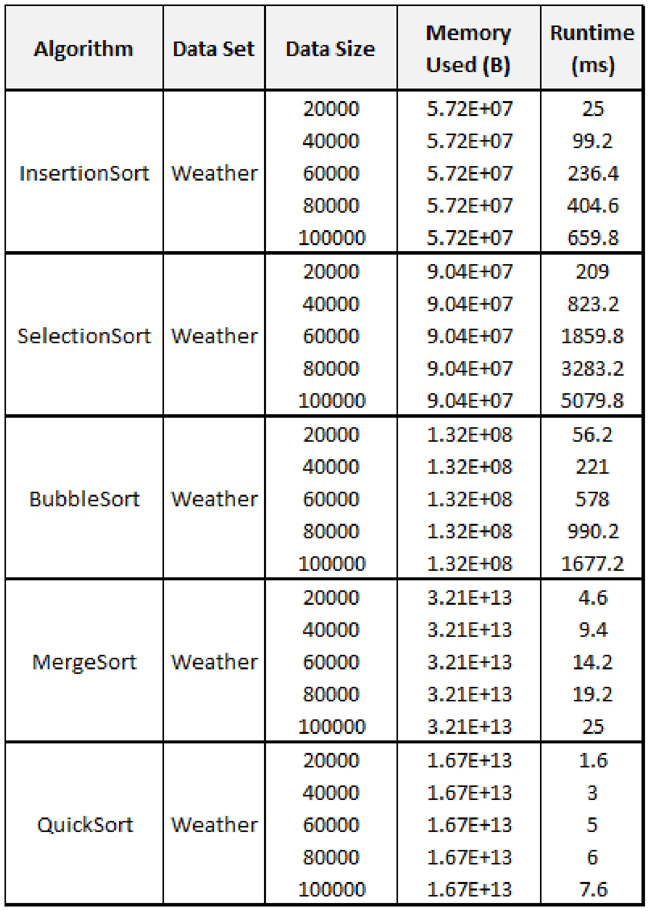
Algorithm	Data Set	Data Size	Memory Used (B)	Runtime (ms)
InsertionSort	Census	20000	5.76E+07	26.4
		40000	5.76E+07	98.4
		60000	5.76E+07	245
		80000	5.76E+07	457.6
		100000	5.76E+07	660
SelectionSort	Census	20000	7.63E+07	201.8
		40000	7.63E+07	806.6
		60000	7.63E+07	1810.4
		80000	7.63E+07	3218.2
		100000	7.63E+07	5039.8
BubbleSort	Census	20000	9.43E+07	102.2
		40000	9.43E+07	425
		60000	9.43E+07	954.4
		80000	9.43E+07	1630.6
		100000	9.43E+07	2611.6
MergeSort	Census	20000	3.17E+13	5.2
		40000	3.17E+13	10.2
		60000	3.17E+13	16.2
		80000	3.17E+13	21.4
		100000	3.17E+13	27.8
QuickSort	Census	20000	1.97E+13	1.4
		40000	1.97E+13	3
		60000	1.97E+13	4.2
		80000	1.97E+13	5.8
		100000	1.97E+13	7.4

Figure 10: Data size experiment results with census data.



Algorithm	Data Set	Data Size	Degree of Sortedness	Memory Used (B)	Runtime (ms)
InsertionSort	Weather	100000	20	5.77E+07	422.2
			40	5.77E+07	253.6
			60	5.77E+07	97.8
			80	5.77E+07	24.4
			100	5.77E+07	0.2
SelectionSort	Weather	100000	20	9.09E+07	5154.8
			40	9.09E+07	5117.4
			60	9.09E+07	5097.4
			80	9.09E+07	5102.8
			100	9.09E+07	5117.6
BubbleSort	Weather	100000	20	1.32E+08	1384.8
			40	1.32E+08	1020.2
			60	1.32E+08	612.6
			80	1.32E+08	242.2
			100	1.32E+08	0
MergeSort	Weather	100000	20	6.17E+13	23.8
			40	6.17E+13	22.8
			60	6.17E+13	22.2
			80	6.17E+13	21
			100	6.17E+13	20.8
QuickSort	Weather	100000	20	2.73E+13	8
			40	2.73E+13	7.4
			60	2.73E+13	7.4
			80	2.73E+13	7.4
			100	2.73E+13	7.2

Figure 11: Degree of Sortedness experiment results with weather balloon data.



Algorithm	Data Set	Data Size	Memory Used (B)	Runtime (ms)
InsertionSort	Weather	20000	5.72E+07	25
		40000	5.72E+07	99.2
		60000	5.72E+07	236.4
		80000	5.72E+07	404.6
		100000	5.72E+07	659.8
SelectionSort	Weather	20000	9.04E+07	209
		40000	9.04E+07	823.2
		60000	9.04E+07	1859.8
		80000	9.04E+07	3283.2
		100000	9.04E+07	5079.8
BubbleSort	Weather	20000	1.32E+08	56.2
		40000	1.32E+08	221
		60000	1.32E+08	578
		80000	1.32E+08	990.2
		100000	1.32E+08	1677.2
MergeSort	Weather	20000	3.21E+13	4.6
		40000	3.21E+13	9.4
		60000	3.21E+13	14.2
		80000	3.21E+13	19.2
		100000	3.21E+13	25
QuickSort	Weather	20000	1.67E+13	1.6
		40000	1.67E+13	3
		60000	1.67E+13	5
		80000	1.67E+13	6
		100000	1.67E+13	7.6

Figure 12: Data size experiment results with weather balloon data.