Theoretical vs. Actual Sort Performance

Joshua Burkett
CoSc 320, Data Structures
Pepperdine University

November 20, 2021

Abstract

Insertion, Selection, Heap, Merge, and Quick are widely know sort methods. They serve as perfect candidates to put theory to the test to see if the real-world data supports the theory. To accomplish this, performance data on each sort is gathered. Using Clion, RStudio, and Dr. J. Stanley Warford's dp4dsDistribution makes it possible to analyze that data. This data analysis concludes that the best sort out of the five is the merge sort.

1 Introduction

Sorting algorithms are fundamental to computer science; but choosing the most effective sort for the job is less fundamentally obvious. The purpose of this paper is to delve into answering that very question. This project analyzes the performance of five common sorting algorithms, the insertion sort, selection sort, heap sort, merge sort, and quick sort. As such, the raw performance data of each sort is gathered as a function of the number of values being sorted. Next, a least squares curve fit is done with the data based on an n^2 and an $n \lg n$ model. This data also serves to confirm whether the theories are correct and agree with the analysis. In other words, it will confirm or deny whether the theoretical Θ increase in execution time as a function of the number of values sorted is correct for these five sorts. Lastly all of this data helps to draw conclusions about which sort is best and any intricacies involved in that answer.

This paper is broken up into four sections, the remaining three are as follows: Section 2 describes the method of data collection used along and compares the characteristics of the sort algorithms. Section 3 describes the analysis and data for each sort. Section 4 wraps up with concluding thoughts.

2 Method

Computer Science theory is fantastic, and presents us with many ways to solve recurrence problems. There are methods such as backwards substitution, the recursion-tree method, guess and verify, and the master method. All of these methods can give us a theoretical Θ increase in execution time for each sort. However, theory is useless if it fails to agree with real-world data.

2.1 Sort algorithms

A characterization of the sort algorithms, as well as a general taxonomy are detailed below. The sort visualizations are from *Design Patterns for Data Structures*. [2] The explanations were created from information from *GeeksforGeeks* and Design Patterns for Data Structures. [1][2]

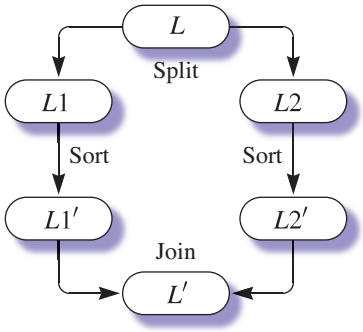


Figure 1: The general Merritt sort taxonomy algorithm. [2]

Although different, all five of the sorts share an overarching characteristic; they all split, sort, then join. The general Merritt sort taxonomy algorithm, displayed in Figure 1, helps to visualize how these sorts work. The various bubbles represent arrays with data stored in them.

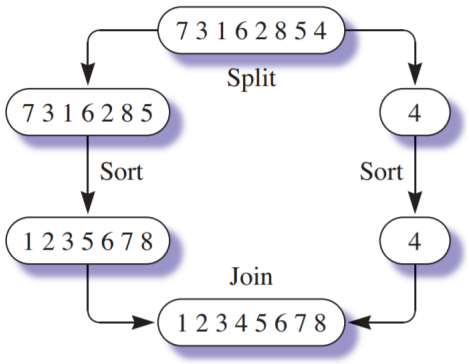


Figure 2: The insertion sort algorithm. [2]

The insertion sort works by splitting off the last value from the array, then sorting the half displayed on the left side of Figure 2. That sort step works by creating a sorted and unsorted section of the array. The first value in the unsorted section is compared to the value of the item in the sorted section. Based on if its greater or lesser, that value will get moved into the proper order inside the sorted section. This is repeated until the entire array is sorted. In the join step, one last loop cycles through to find where that originally split-off value belongs. Theoretically, it has a time complexity of $\Theta(n^2)$.

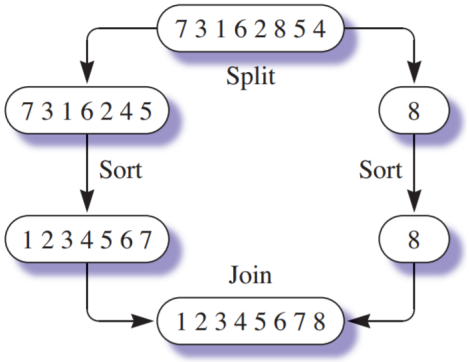


Figure 3: The selection sort algorithm. [2]

The selection sort works by splitting off the largest value from the array, then sorting the half displayed on the left side of Figure 3. That sort step works by creating a sorted and unsorted section of the array, similar to the insertion sort. Dissimilar, however, is that the first value in the unsorted section is compared to the other values in the unsorted section until a lowest value is found. Based on if its greater or lesser, that value will get moved front of the array. This is repeated with each new lowest being placed behind

the lowest from the last loop. This is done until the entire array is sorted. In the join step, one last loop cycles through to find where that originally split-off value belongs. Theoretically, it has a time complexity of $\Theta(n^2)$.

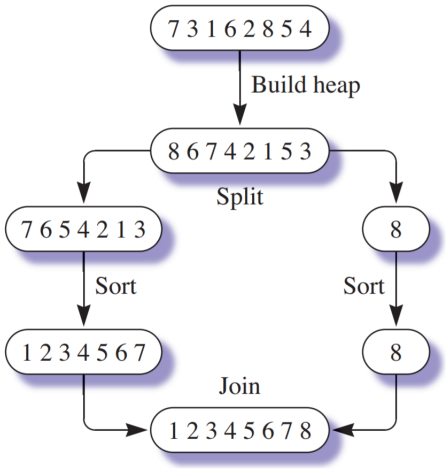


Figure 4: The heap sort algorithm. [2]

The heap sort in Figure 4 is unique compared to the others in the fact that it cannot accept just any ordinary array. The array must be pre-sorted into a binary tree that satisfies both the max heap order, and the max heap shape. The largest value is split off, and the last value of the array is put in its place at the root. Because its a smaller value, it will be in the wrong place, and will need to be sifted down to re-satisfy the max heap shape. The process is repeated until the array is sorted. Theoretically, it has a time complexity of $\Theta(n \lg n)$.

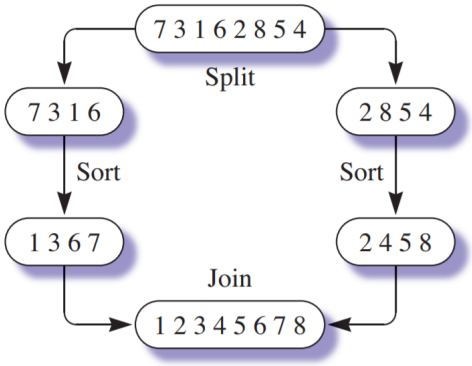


Figure 5: The merge sort algorithm. [2]

The merge sort in Figure 5 is relatively simple. It begins by splitting the array into two halves. The two individual halves are each recursively sorted, then joined together. This join step simply compares each value from left to right of each list; whichever value is lower is placed into the sorted array. Theoretically, it has a time complexity of $\Theta(n \lg n)$.

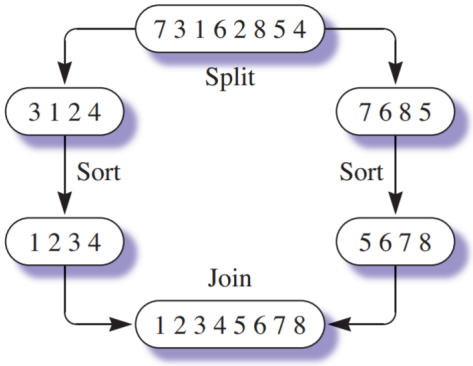


Figure 6: The quick sort algorithm. [2]

The quick sort in Figure 6 has similarities to the merge sort, but the initial split is done based on a rough calculation of the median of the array. Anything above that median value will get placed to the right, and everything below that value will get placed to the left. These new lists are recursively sorted, then easily joined with no additional calculation required. Theoretically, it has a time complexity of $\Theta(n \lg n)$ or $\Theta(n^2)$ based on if its a average case or worst case respectively.

2.2 Data collection

The method needed to gather this data is more complicated than it may at first seem. This project requires gathering the counts of comparisons and assignments that the code of each sort makes during run time. Since manually counting these would be ridiculous, Dr. J. Stanley Warford created a solution for this very problem in his project called dp4dsDistribution.

This project was made possible by the C++ integrated development environment, Clion, developed by JetBrains Inc., and the dp4dsDistribution project. The dp4dsDistribution was created to include SortCompAsgn.cpp, which uses the sort algorithmns created by the user and outputs the raw count data of the comparisons and assignments made when running with the test files. These test files are .txt files filled with random integers. Each test file includes 500 more numbers than the last, which provides a range of quantities

to test by selecting each different test file.

2.3 Analysis

Residual standard error (RSE) is crucially important for determining whether the data gathered represents a $n \lg n$ or n^2 model. A lower RSE means our raw data more closely matches the current model being tested ($n \lg n$ or n^2). This will allow us to label the sorts, but also to determine which is statistically best.

The following is the formula to calculate the RSE:

$$RSE = \sqrt{\frac{\sum (y_i - \hat{y}_i)^2}{d.f.}}$$

This formula takes the sum over all the data points. y_i is the y value of an individual data point, $\hat{y}_i$ is the y value of the point on the curve whose x value is the same as the x value of y_i, and $d.f.$ is the degrees of freedom.

RSE is extremely useful, but not without a model to compare against for our $n \lg n$ and n^2 theoretical execution time increases. Here is the quadratic curve fit equation.

$$y = An^2 + Bn + C$$

Here is the $n \lg n$ curve fit equation.

$$y = An \lg n + Bn + C$$

These forumla's are used inside the R integrated development environment called RStudio.

3 Results

This section presents the raw data and analysis.

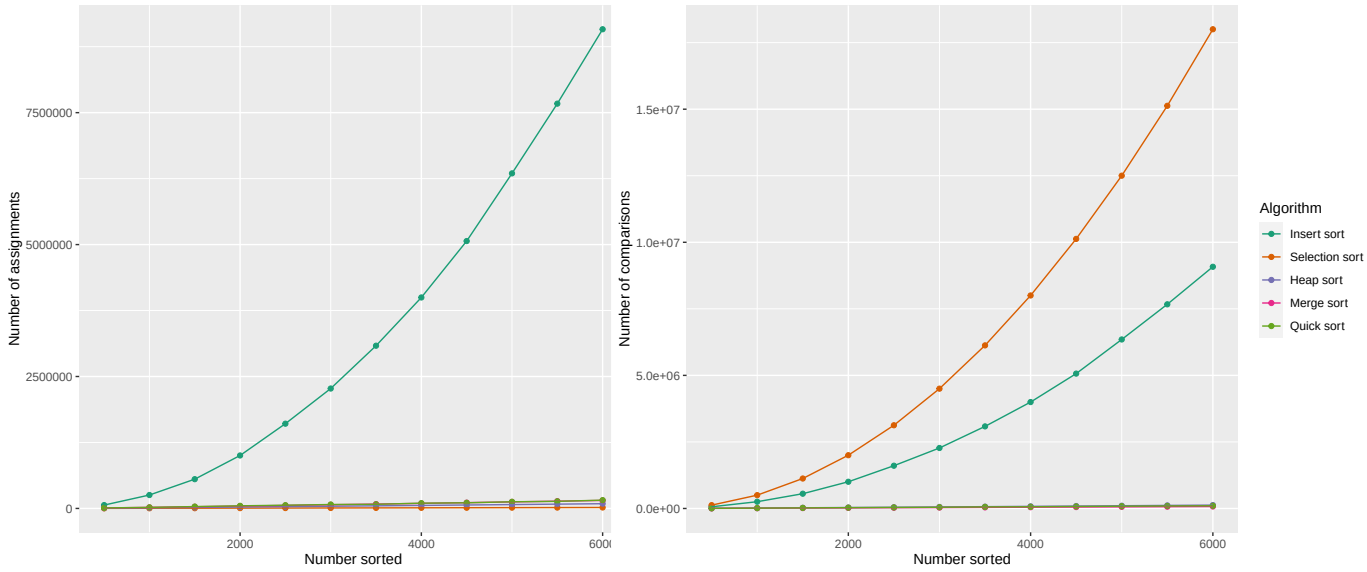
3.1 Raw data

Number of data points	Algorithm				
	Insert	Select	Heap	Merge	Quick
500	63597	125249	7000	4344	6275
1000	254362	500499	15967	9700	13618
1500	554589	1125749	25813	15467	22049
2000	1003667	2000999	35962	21400	30942
2500	1606133	3126249	46662	27595	39413
3000	2272433	4501499	57608	33950	46249
3500	3083553	6126749	68686	40320	59176
4000	3997696	8001999	79945	46820	64164
4500	5066599	10127249	91547	53511	75776
5000	6349081	12502499	103334	60253	85019
5500	7670586	15127749	115147	66972	94234
6000	9080159	18002999	127042	73910	101023

Figure 7: Number of comparisons.

Number of data points	Algorithm				
	Insert	Select	Heap	Merge	Quick
500	63604	1497	5667	8976	9829
1000	254367	2997	12335	19952	20130
1500	554593	4497	19415	31904	35252
2000	1003673	5997	26684	43904	49612
2500	1606137	7497	34207	56808	61185
3000	2272438	8997	41845	69808	74336
3500	3083561	10497	49556	82808	78058
4000	3997707	11997	57388	95808	99816
4500	5066605	13497	65354	109616	107027
5000	6349094	14997	73410	123616	126268
5500	7670593	16497	81442	137616	134445
6000	9080166	17997	89631	151616	156893

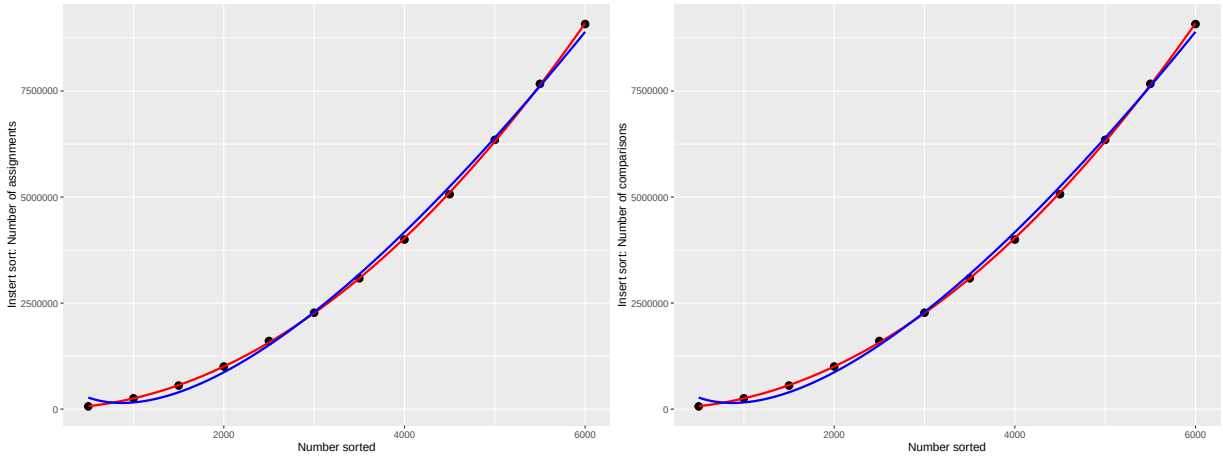
Figure 8: Number of assignments.



(a) Plot of number of assignments. (b) Plot of number of comparisons.

Figure 9: Sort algorithm assignments and comparisons.

3.2 Insertion sort



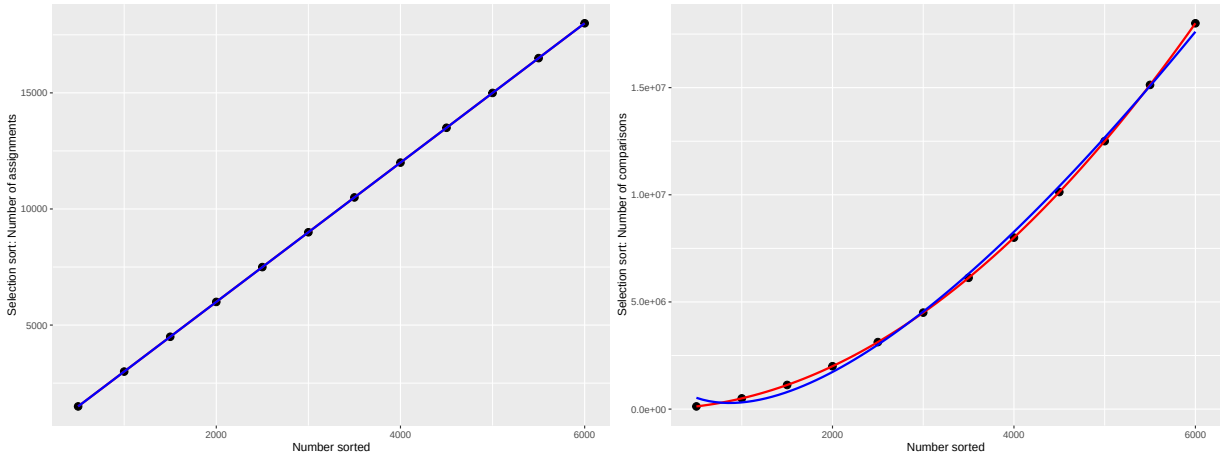
(a) Plot of number of assignments. (b) Plot of number of comparisons.

Figure 10: Insert sort algorithm assignments and comparisons.

Figure 10 displays the number of assignments (a), and number of comparisons (b) for the insertion sort data. Starting with number of assignments, the n^2 curve fit yields an RSE of 28130 on 9 degrees of freedom. This is compared to the $n \lg n$ curve fit RSE of 155300 on 9 degrees of freedom. Assessing the number of comparisons, the n^2 curve fit yields an RSE of 28130 on 9 degrees of freedom and a $n \lg n$ curve fit RSE of 155300 on 9 degrees of freedom. For both assignments and comparisons, the RSE

of the n^2 curve fit is lowest. This confirms the theory that the asymptotic bounds are $\Theta(n^2)$.

3.3 Selection sort

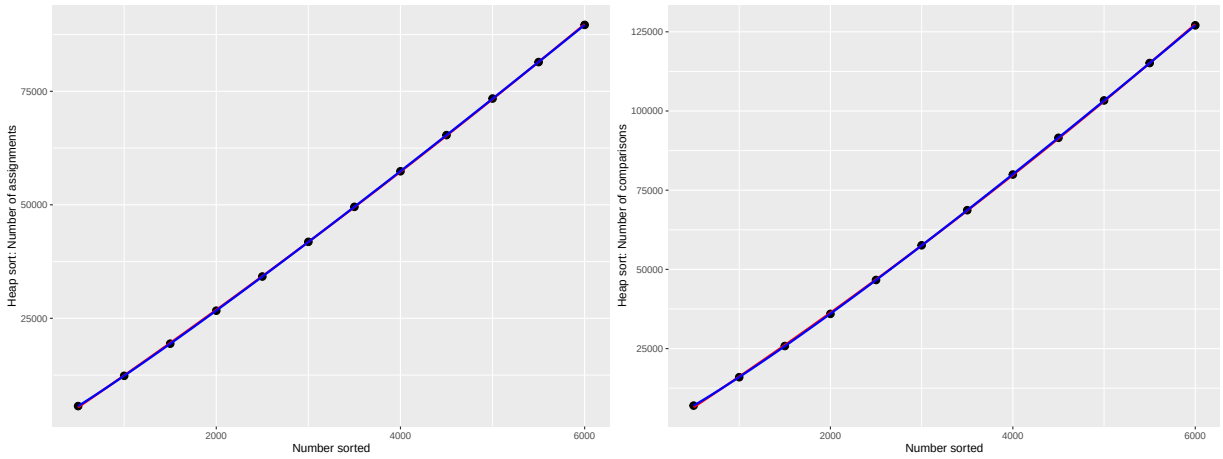


(a) Plot of number of assignments. (b) Plot of number of comparisons.

Figure 11: Selection sort algorithm assignments and comparisons.

Figure 11 displays the number of assignments (a), and number of comparisons (b) for the selection sort data. Starting with number of assignments, the n^2 curve fit yields an RSE of 3.832e-12 on 9 degrees of freedom. This is compared to the $n \lg n$ curve fit RSE of 2.256e-12 on 9 degrees of freedom. Assessing the number of comparisons, the n^2 curve fit yields an RSE of 4.585e-09 on 9 degrees of freedom and a $n \lg n$ curve fit RSE of 291500 on 9 degrees of freedom. There is actually no single answer for this sort. Seeing as a sort involves both the assignments and the comparisons, this confirms the $\Theta(n^2)$ asymptotic bounds posed by the theory.

3.4 Heap sort

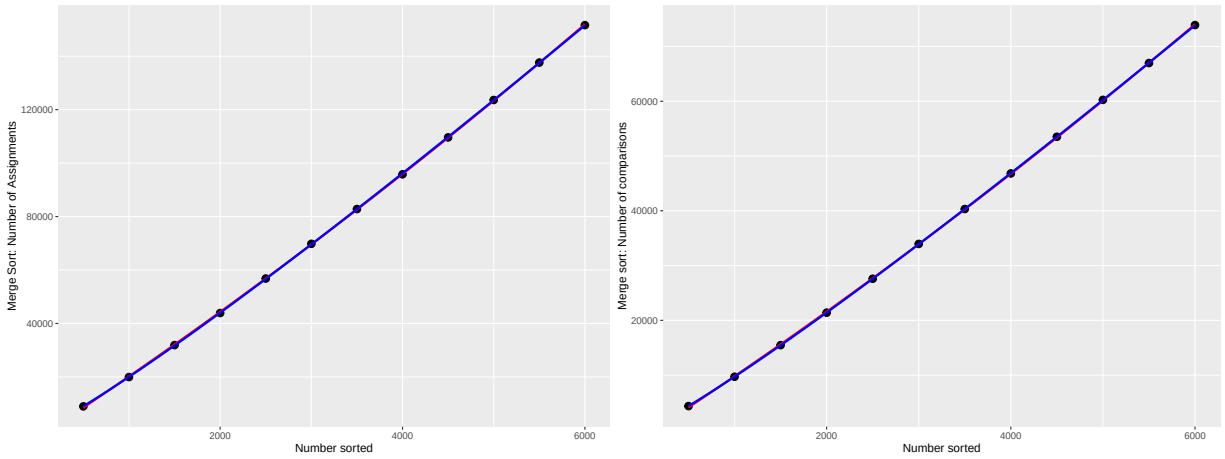


(a) Plot of number of assignments. (b) Plot of number of comparisons.

Figure 12: Heap sort algorithm assignments and comparisons.

Figure 12 displays the number of assignments (a), and number of comparisons (b) for the heap sort data. Starting with number of assignments, the n^2 curve fit yields an RSE of 155.4 on 9 degrees of freedom. This is compared to the $n \lg n$ curve fit RSE of 22.41 on 9 degrees of freedom. Assessing the number of comparisons, the n^2 curve fit yields an RSE of 312.9 on 9 degrees of freedom and a $n \lg n$ curve fit RSE of 56.79 on 9 degrees of freedom. For both assignments and comparisons, the RSE of the $n \lg n$ curve fit is lowest. This confirms the theory that the asymptotic bounds are $\Theta(n \lg n)$.

3.5 Merge sort

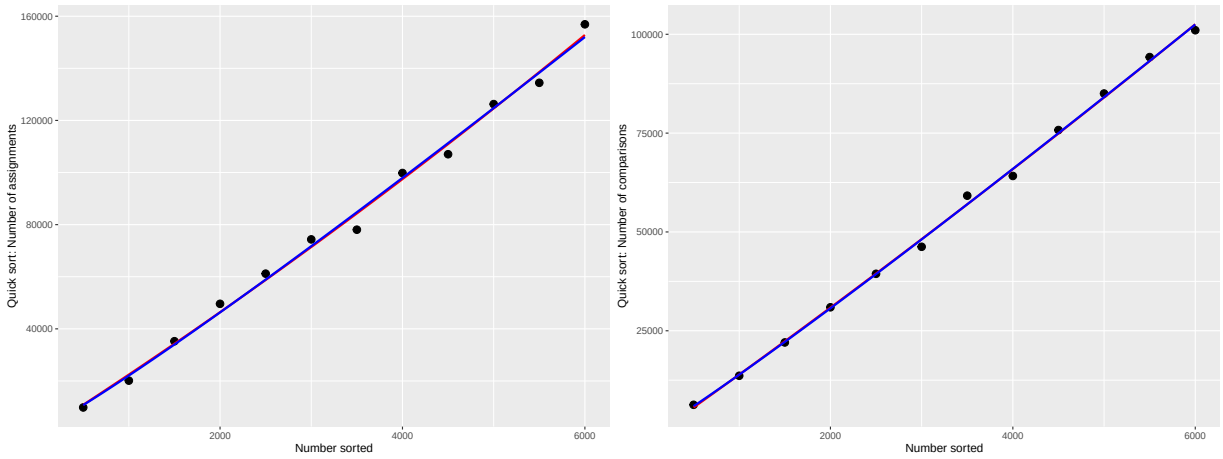


(a) Plot of number of assignments. (b) Plot of number of comparisons.

Figure 13: Selection sort algorithm assignments and comparisons.

Figure 13 displays the number of assignments (a), and number of comparisons (b) for the merge sort data. Starting with number of assignments, the n^2 curve fit yields an RSE of 291.8 on 9 degrees of freedom. This is compared to the $n \lg n$ curve fit RSE of 190.1 on 9 degrees of freedom. Assessing the number of comparisons, the n^2 curve fit yields an RSE of 147.8 on 9 degrees of freedom and a $n \lg n$ curve fit RSE of 36.19 on 9 degrees of freedom. For both assignments and comparisons, the RSE of the $n \lg n$ curve fit is lowest. This confirms the theory that the asymptotic bounds are $\Theta(n \lg n)$.

3.6 Quick sort



(a) Plot of number of assignments. (b) Plot of number of comparisons.

Figure 14: Selection sort algorithm assignments and comparisons.

Figure 14 displays the number of assignments (a), and number of comparisons (b) for the quick sort data. Starting with number of assignments, the n^2 curve fit yields an RSE of 3776 on 9 degrees of freedom. This is compared to the $n \lg n$ curve fit RSE of 3900 on 9 degrees of freedom. Assessing the number of comparisons, the n^2 curve fit yields an RSE of 1382 on 9 degrees of freedom and a $n \lg n$ curve fit RSE of 1346 on 9 degrees of freedom. For both assignments and comparisons, the RSE between the $n \lg n$ and n^2 curve fits were very close. Technically though, the assignments curve fit best to n^2, while the comparisons curve fit best to $n \lg n$. This lines up perfectly with the theory being $\Theta(n^2)$ for the worst case, and $\Theta(n \lg n)$ for an average case.

3.7 Sort comparisons

The merge sort is by far the best of the five sorts analyzed.While the quick sort is a very close contender, under most tests, it had equivalent or less assignments/comparisons. The fact that it scales is really important too, making it great for applications that may need large quantities sorted. An insert or selection sort might be fine with small amounts of data, but as soon as the sorts need to handle large quantities, those two crumble under their inefficiencies. The overall dominance of the merge sort is visualized as the lowest (and therefore most efficient) pink line on Figure 9 (a) and (b).

4 Conclusions

This project has checked all the intended boxes, and most importantly, proved that the theoretical asymptotic bounds are correct when compared to real world data. This experiment discovered that the merge sort is the best of the five sorts tested. That said, using a quick sort or heap sort still has far better performance than the selection and insertion sorts. So, the most important finding is this: for any project where large data arrays currently do, or may need to be sorted, insertion and selection sorts should be avoided at all costs.

References

[1] Sorting algorithms. https://www.geeksforgeeks.org/sorting-algorithms, 2008. Online; accessed 20-November-2021.

[2] Dung X. Nguyen and J. Stanley Warford. *Design Patterns for Data Structures*. Pepperdine, prepublication manuscript edition, 2021.