Analysis of Sorting Algorithms

Augusto Alvarez

CoSc 320, Data Structures

Pepperdine University

October 26, 2016

Abstract

The purpose of this paper is to compare theoretical efficiencies of various sorting algorithms with their actual efficiencies. By sorting data ranging from 200 to 6400 values, the number of assignments and comparisons were calculated for each algorithm. The results were plotted and the residual standard error was computed to see how well each algorithm matched their respective theoretical values. For the most part, the actual values corroborated the theoretical efficiencies.

1 Introduction

The five sorting algorithms and their respective theoretical efficiencies are as follows: Insertion Sort, $\Theta(n^2)$, Selection Sort, $\Theta(n^2)$, Heap Sort, $\Theta(n \lg n)$, Merge Sort, $\Theta(n \lg n)$, Quick Sort, best case $\Theta(n \lg n)$, worst case $\Theta(n^2)$. The efficiencies are presented as asymptotic tight bounds. This simply means that each algorithm is bounded both above and below by a certain function, giving a precise prediction for how long it will take to sort a certain number of values. In practice, however, computing the actual time requires a rigorous analysis of each line in the algorithm. Instead, in running each sort, the number of comparisons and assignments are counted. This is a good approximation because much of the time elapsed when a program is run is due to the number of comparisons or assignments made. Actual unsorted data sets of quantities 200, 400, 800, 1600, 2400, 3200, 4000, 4800, 5600, and 6400 were provided. Each set was sorted with each of the algorithms above. Using a program to count the number of comparisons and the number of assignments made during the execution of each algorithm, the data was tabulated and plotted. The goal is to statistically analyze the data and find out whether or not the actual efficiencies match the theoretical efficiencies above.

The method section describes the details of the experimental process. This includes a brief description of each sorting algorithm and their characteristics, the object oriented design pattern used for collecting the data, and an analysis of the statistical procedure used to corroborate or falsify the theoretical efficiencies of each algorithm. The results section contains the raw tabulated data, curve fit plots, and an analysis of each algorithm. The conclusion section gives a summary of the specific results.

2 Method

2.1 Sort Algorithms

All of the algorithms stated in the introduction adhere to the general Merritt Sort Taxonomy pattern. This pattern is seen directly in the implementation of the abstract sort method, which first splits the data, recursively calls the sort method for the two sublists, then joins the data. Although seemingly counterintuitive, the same sort method is used for each algorithm. This is because all the work of actually sorting the data is done in either the split or join methods, which are implemented differently for each algorithm. Each sort can be categorized as having either an easy split and a hard join or a hard split and an easy join. Selection Sort, Quick Sort, and Heap Sort perform all the work in the split, while the join is trivial. Insertion Sort and Merge Sort perform all the work in the join, while the split is trivial.

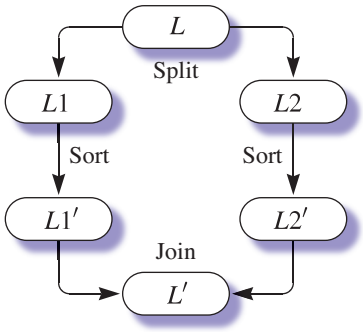


Figure 1: The general Merritt sort taxonomy algorithm.

Selection Sort finds the largest element in the list and swaps it with the last element. The recursive call performs this split with all but the last element, so by the time the split calls are done, the list is already sorted. Quick sort is unique in the sense that the split is done by computing the median of three randomly selected values in the list and setting that to be the key. Any value less than the key is placed to the left and any value

greater than the key is placed to the right. By the time the split is done, all the values are sorted and the join is easy. Heap Sort relies on formulating the original data set into a binary tree that adheres to max heap properties. Because of the max heap order, it can split off the first value, which will always be the largest and place it at the end of the list. This breaks the max heap order and sift down is called to reorder the elements. This is done recursively until the list is sorted, making the join very easy. Insertion Sort trivially splits off the elements one by one, while the join method does all the work by inserting each element in the correct place in the sorted sublist. The merge sort easily splits the data set in half each time, while the join method does the work of comparing the values in the sublists and joining them in the correct order.

2.2 Data Collection

Provided with ten sets of data ranging from 200 to 6400 unsorted values, the program SortCompAsgn counts both the number of pairwise comparisons and number of assignments done for each sorting algorithm. It does this with function overloading in the CAMetrics class. These functions, however, are only for elements of type T. This means that only these elements of type T will be used in the comparison and assignment counts. The program first asks for an array capacity and a text file containing the unsorted values. It then executes each sorting algorithm on the provided data set and returns the number of comparisons and assignments for each algorithm. Running the program for all ten sets, the raw data is produced.

2.3 Analysis

This data is then imported into RStudio, a software used for statistical analysis. Within RStudio, the data for each sorting algorithm was plotted with number of data points sorted on the x-axis and either number of comparisons or number of assignments on the y-axis. Because each algorithm has a theoretical efficiency of either $\Theta(n^2)$ or $\Theta(n \lg n)$, curve fits for both functions were done on each plot. The quadratic and logarithmic functions are shown below, respectively.

$$y = Ax^2 + Bx + C$$

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

In RStudio, the curve fit automatically modifies the coefficients to better fit the experimental values. A residual standard error (RSE) of both curve fits is then computed

for each algorithm.

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

The RSE finds the difference between the y-value on the curve fit and the corresponding actual value in the experimental values for each data set (x-value). Because the RSE is a measure of how much the actual values deviate from the theoretical values given by the curve fit function, a smaller RSE corresponds to a better fit. This statistical procedure gives a concrete way of verifying the theoretical efficiencies of each algorithm.

3 Results

	Number of data points									
Algorithm	200	400	800	1600	2400	3200	4000	4800	5600	6400
Insert	10124	40020	166846	638426	1448480	2548171	4024038	5760162	7789322	10231211
Select	20099	80199	320399	1280799	2881199	5121599	8001999	11522399	15682799	20483199
Heap	2854	6781	15769	35829	57542	80525	103787	127935	152958	178174
Merge	1470	3366	7511	16671	26381	36477	46787	57523	68411	79401
Quick	2143	5137	10892	23792	38492	51488	65118	81712	92705	108077

Figure 2: Number of comparisons.

	Number of data points									
Algorithm	200	400	800	1600	2400	3200	4000	4800	5600	6400
Insert	10130	40023	166851	638438	1448491	2548181	4024049	5760169	7789330	10231222
Select	597	1197	2397	4797	7197	9597	11997	14397	16797	19197
Heap	2019	4406	9639	20834	32647	44845	57294	70040	83004	96150
Merge	3088	6976	15552	34304	54208	75008	95808	118016	140416	162816
Quick	3391	8361	16000	37245	61532	83539	104799	124169	144979	168954

Figure 3: Number of Assignments.

Shown above is the raw data for both the number of comparisons and the number of assignments counted for each of the sorting algorithms. The table below shows the RSE for a logarithmic and quadratic fit for all the sorting algorithms.

Algorithm	$\Theta(n \lg n)$	$\Theta(n^2)$
Insert	233300	18920
Select	467400	3.56e-09
Heap	125.4	685.1
Merge	44.79	246.9
Quick	979.1	1038

(a) Number of comparisons.

Algorithm	$\Theta(n \lg n)$	$\Theta(n^2)$
Insert	233300	18920
Select	1.295e-12	2.333e-12
Heap	30.09	246
Merge	181.9	499.3
Quick	1643	1667

(b) Number of assignments.

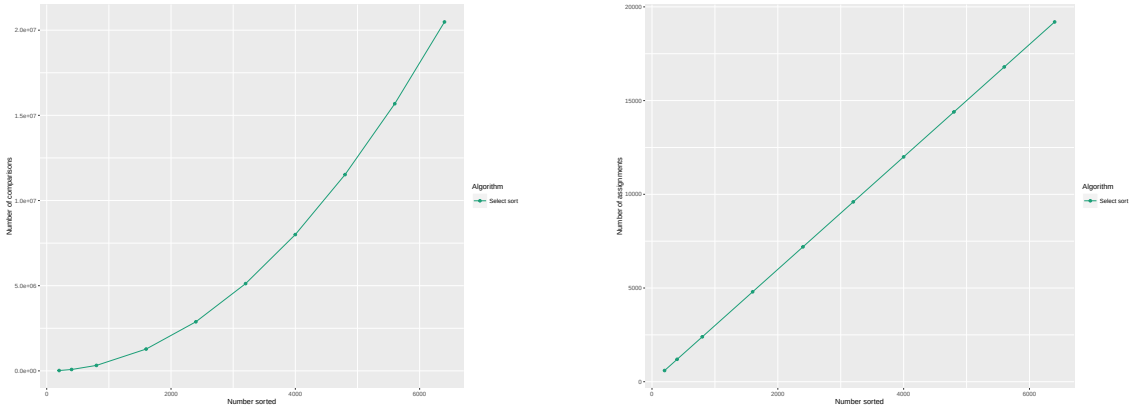
Figure 4: RSE data for logarithmic and quadratic fit.

For Insertion Sort, the RSE for the $\Theta(n^2)$ fit is smaller than the RSE for the $\Theta(n \lg n)$ fit. This is the case for both number of comparisons and number of assignments. Therefore, it is safe to conclude that the experimental efficiency matches the theoretical efficiency. A quick look at the raw data in figures 2 and 3 also reveals some valuable information. The number of assignments and number of comparisons for Insertion Sort are much larger than any of the other sorts, with the exception of the number of comparisons done for Selection Sort.

The analysis of Selection Sort gives rise to an interesting phenomenon. Looking at the number of comparisons, the RSE for the $\Theta(n^2)$ fit is smaller than the RSE of the $\Theta(n \lg n)$ fit. This is in accordance to the theoretical efficiency for Selection Sort. Looking at the number of assignments, the RSE is practically zero for both fits, meaning that the experimental data fits both the logarithmic and quadratic fit nearly perfectly. This result is, of course, erroneous. Taking a look at the raw data from figures 2 and 3 again reveals that there is a huge discrepancy between the number of comparisons and the number of assignments made. This discrepancy is explained by the way the algorithm is written. Selection Sort works by finding the largest element in the list, placing it at the end, then repeating the same process recursively for $n-1$ elements. The algorithm loops through the entire list making comparisons to find the largest element, but only makes one assignment per recursive call. Shown below are the two plots for Selection Sort.

Although it may seem like the algorithm is extremely efficient because the number of assignments made is related linearly to the number of data points sorted, this is an

erroneous conclusion to make. An algorithm is only as fast as its slowest process. Even though the number of assignments made implies a theoretical efficiency of $\Theta(n)$, this does not change the fact that it still has to make a very large amount of comparisons. This idea can be applied to all algorithms. Whenever there is a large discrepancy between the number of comparisons and the number of assignments, the larger of the two must be used for analysis. Therefore, the experimental data for Selection Sort does match the theoretical efficiency of $\Theta(n^2)$.

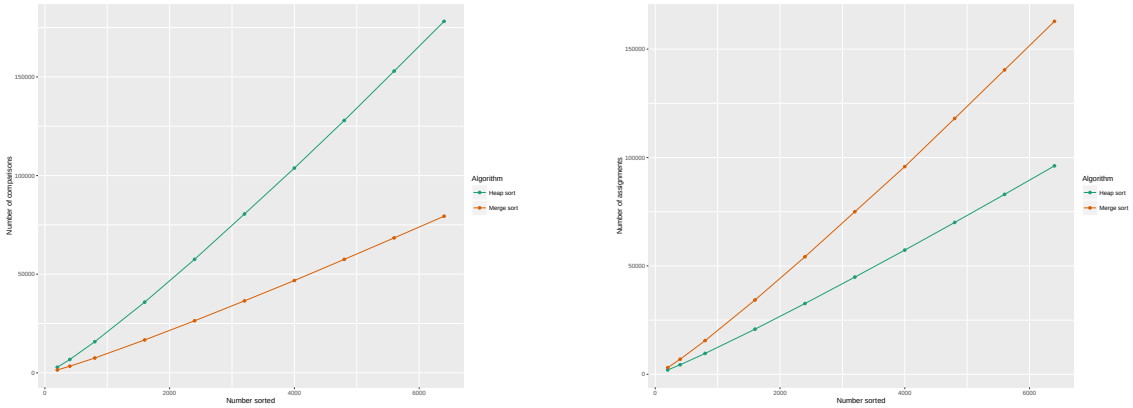


(a) Number of comparisons. (b) Number of assignments.

Figure 5: The discrepancy in Select sort's comparisons and assignments.

For Heap Sort, we see that for both the number of comparison and the number of assignments, the RSE is smaller for the logarithmic fit than for the quadratic fit. This corroborates the theoretical efficiency of $\Theta(n \lg n)$. For Merge Sort, this is also the case. Theoretically, both Heap Sort and Merge Sort are equally as efficient, but this does not indicate which one would be better in practice. This can be answered by plotting the raw data of the two sorts. Something interesting arises when analyzing the two plots. For the number of comparisons, Merge Sort is more efficient, but for the number of assignments, Heap sort is more efficient. So which of the two is better? One way to approximate an answer is to assume that the number of comparisons and the number of assignments contribute both equally to the total time it takes for an algorithm to sort data. This way, we can simply add the number of comparisons and assignments together for both algorithms and see which total is less. Since in the analysis of algorithms it is almost always more informative to see how the algorithm trends with larger values, the addition is performed with the results collected from the 6400 data set. For Heap Sort we get $178174 + 96150 = 274324$, and for Merge Sort we get $79401 + 162816 = 242217$. Therefore the more efficient of the two is Merge Sort. The difference here is practically

negligible but with extremely large data sets this could make a big difference.



(a) Number of comparisons. (b) Number of assignments.

Figure 6: A comparison of Heap Sort and Merge Sort.

The analysis of Quick Sort also yields some interesting results. We see that for both the number of comparisons and the number of assignments, the RSE is smaller for the logarithmic fit. This is in accordance with the best case theoretical efficiency of $\Theta(n \lg n)$. Taking a closer look at figure 4, it is clear that for both the number of comparisons and number of assignments, the RSE values for the logarithmic fit and the quadratic fit are extremely close. This means that the experimental data actually corroborates both curve fits, it simply depends on the case which one it fits better. Due to the inherently random nature of the split method in this algorithm, there will be cases in which we are unlucky and the RSE will be smaller for the quadratic fit. In this particular execution of the code, we got slightly lucky in the random selection of the key variables for each call of the split method, which led to RSE values that are slightly lower for the logarithmic fit. Because the RSE values are very close together, this particular execution is a good representation of the best and worst case theoretical efficiencies of $\Theta(n \lg n)$ and $\Theta(n^2)$, respectively.

4 Conclusion

All of the data corroborated the theoretical efficiencies. Insertion Sort and Selection Sort have an efficiency of $\Theta(n^2)$, Heap Sort and Merge Sort have an efficiency of $\Theta(n \lg n)$, and Quick Sort has a best case efficiency of $\Theta(n \lg n)$ and a worst case efficiency of $\Theta(n^2)$. From the analysis of Selection Sort, it was shown that if either the

number of comparisons or assignments greatly surpasses the other, to use the largest of the two to calculate the RSE and draw conclusions. From the analysis of Quick Sort, it is shown that the similar RSE values for number of comparisons and assignments are a direct result of the random nature of the algorithm's split method.

References

[1] Dung X. Nguyen and J. Stanley Warford. *Design Patterns for Data Structures*. Pepperdine, Prepublication manuscript, 2016.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[3] David Gries and Fred B. Schneider. *Equational propositional logic*. Information Processing Letters, 53rd Volume, 1995.

[4] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.