Assignment 5 Report
CSCI-2720
Nathan Jacobi
Taylor Wetterhan

Assignment 5: Nathan Jacobi and Taylor Wetterhan

Introduction

The main goal of this assignment is to use different sorting algorithms to sort an array of integers, then compare their efficiency by counting the number of comparisons in each sorting session. This was mainly done through two exercises, as outlined in the instructions of the assignment. The first exercise was to sort text documents which were provided, and find how many comparisons each algorithm used to finish sorting. The second exercise is to create random strings of integers at different sizes and assess the relationship between input size and number of comparisons. After implementing the algorithms, assessing them theoretically, and completing these exercises, the algorithms used were analysed using the data gathered.

Results

The different algorithms were first used to sort the list of 10,000 integers given in the three different text files: ordered.txt, reverse.txt, and random.txt. Between each of the files, the efficiency of most of the algorithms based on the number of comparisons stayed fairly consistent, even more so in the case of selection sort, which was always the same number. In general, it was found that the efficiency of each algorithm from least to most is as follows: selection sort, heap sort, quicksort (random pivot), and merge sort. Quicksort (first pivot), is very inefficient with lists that are already mostly sorted. However, with the random list, it was nearly as efficient as merge sort and quicksort (random pivot). The table shown in Figure 1 below shows the number of comparisons from each algorithm, along with some comments on its time complexity and number of comparisons.

Figure 1. Table showing each algorithm and its number of comparisons based on the file.

Algorithm	Input Type	# of Comparisons	Comments
Selection Sort	Ordered	49,995,000	$O(N^2)$, expected 8-9 digits of comparisons
Merge Sort	Ordered	69,008	O(NlogN), expected 5-6 digits of comparisons
Heap Sort	Ordered	244,460	O(NlogN), expected 5-6 digits of comparisons
Quicksort First Pivot	Ordered	49,995,000	$O(N^2)$ - worst case, expected 8-9 digits of comparisons
Quicksort Random Pivot	Ordered	147,857	O(NlogN), expected 5-6 digits of comparisons [Average of 5]

Assignment 5: Nathan Jacobi and Taylor Wetterhan

Algorithm	Input Type	# of Comparisons	Comments
Selection Sort	Reverse	49,995,000	$O(N^2)$, expected 8-9 digits of comparisons
Merge Sort	Reverse	64,608	O(NlogN), expected 5-6 digits of comparisons
Heap Sort	Reverse	226,682	O(NlogN), expected 5-6 digits of comparisons
Quicksort First Pivot	Reverse	49,995,000	$O(N^2)$ - worst case, expected 8-9 digits of comparisons
Quicksort Random Pivot	Reverse	155,673	O(NlogN), expected 5-6 digits of comparisons [Average of 5]
Selection Sort	Random	49,995,000	$O(N^2)$, expected 8-9 digits of comparisons
Merge Sort	Random	120,414	O(NlogN), expected 5-6 digits of comparisons
Heap Sort	Random	235,430	O(NlogN), expected 5-6 digits of comparisons
Quicksort First Pivot	Random	162,640	O(NlogN), expected 5-6 digits of comparisons
Quicksort Random Pivot	Random	151,457	O(NlogN), expected 5-6 digits of comparisons [Average of 5]

A vector was used when reading in the values from the text files for its dynamic attributes, namely that vectors grow as needed when adding values. Once the vector was created, the length of the list was found using the .size() function, which is in the CPP standard library. Afterwards, the vector was copied into an array. This is accomplished in main.cpp, under the makeList function.

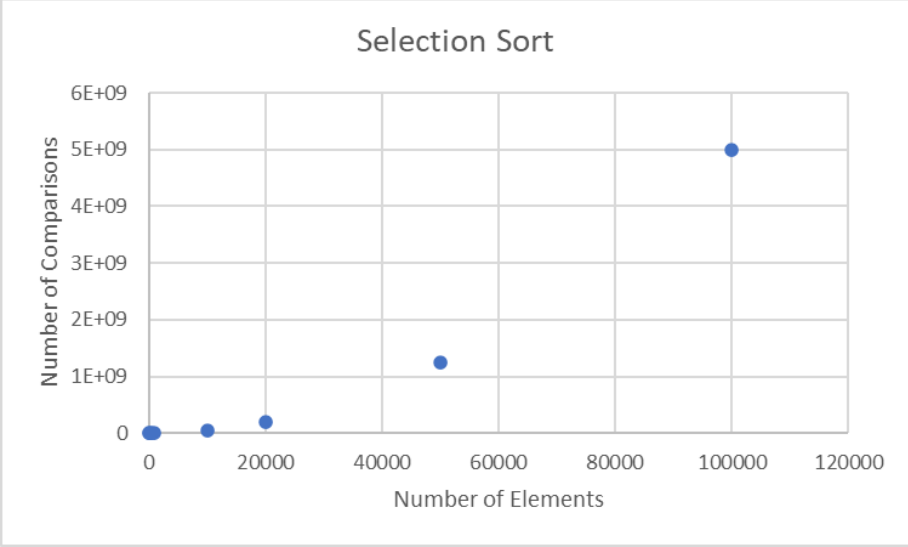
Based on the first experiment with the three given files, merge sort is always more efficient than the other algorithms. However, especially in an unsorted list, quicksort, both with random and first pivot, has a similarly minimized number of comparisons. Thus, it could be expected that quicksort may be more efficient than merge sort given a specific condition. On the other hand, it could often be worse due to the fact that it can reach $O(N^2)$ complexity.

Then, in the second experiment, each algorithm was assessed using lists of random integers of different lengths. Through this experiment, it becomes more clear which algorithms are more efficient and when since their functions grow differently. The conclusion that merge sort is the most efficient from the first experiment is supported for most list sizes. Figures 2 and 3 show the number of comparisons for each algorithm at different list sizes. The plots in Figure 3 clearly display the change in efficiency of the algorithms, which also bear resemblance to their time complexity functions.

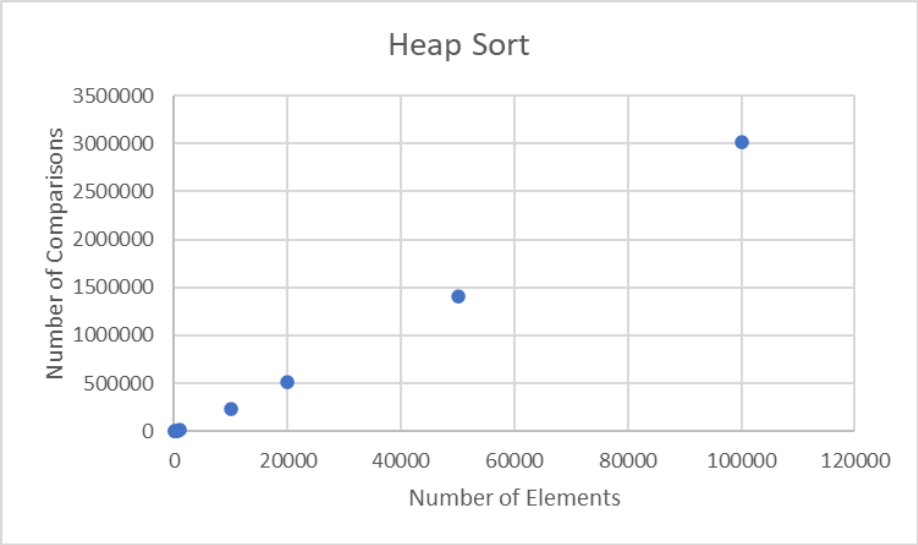
Figure 2. Table showing each algorithm and its number of comparisons based on the number of random inputs.

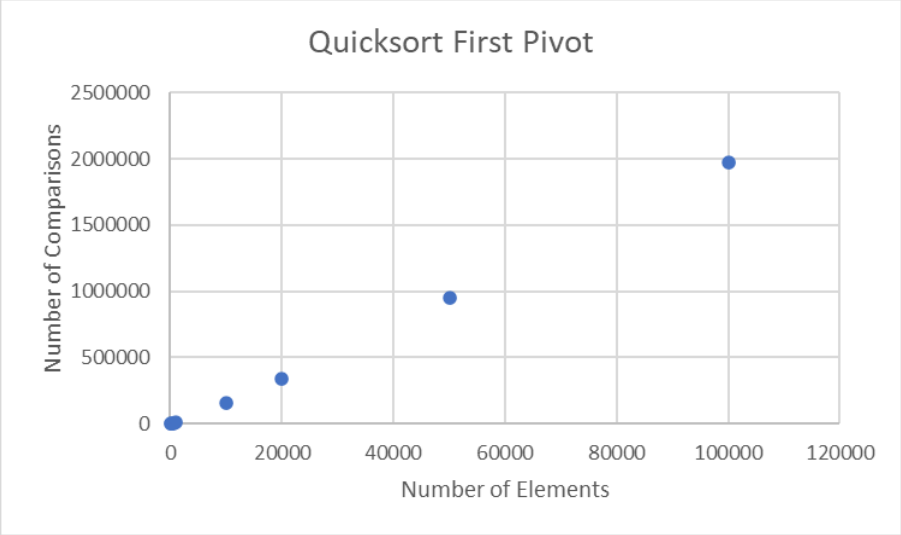
Algorithm	**10**	**100**	**500**	**1000**	**10000**	**20000**	**50000**	**100000**
Selection Sort	45	4,950	124,750	499,500	49,995,000	199,990,000	1,249,975,000	4,999,950,000
Merge Sort	23	540	3843	8,683	120,529	260,894	718,275	1,536,494
Heap Sort	38	1,030	7,420	16,862	235,275	510,708	1,409,888	3,019,610
Quicksort First Pivot	22	589	4,600	12,531	156,358	341,441	946,377	1,971,889
Quicksort Random Pivot	27	620	4,745	11,487	161,704	342,977	943,308	2,049,700

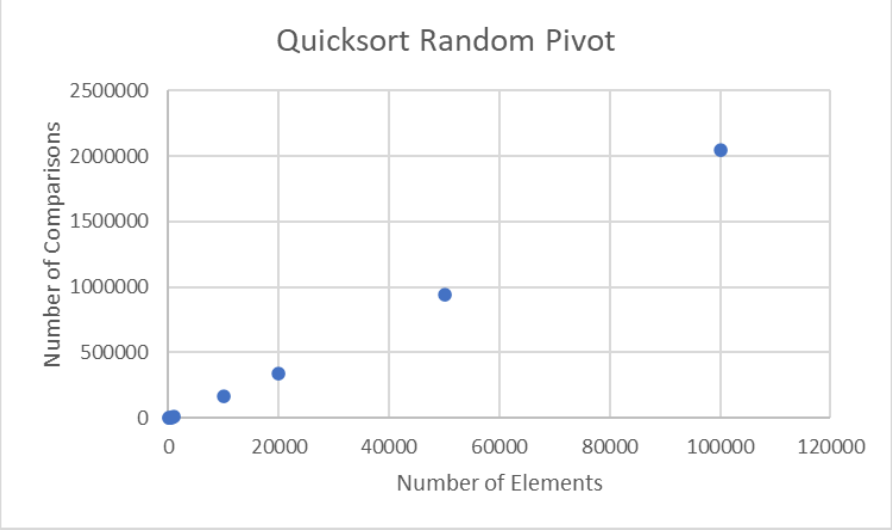
Figure 3. Plots of the number of comparisons of each algorithm based on the number of elements in the list..



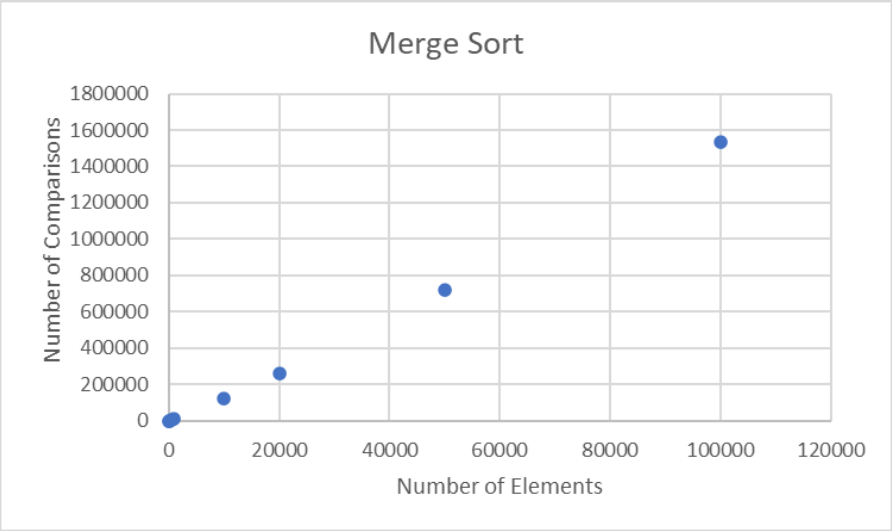
Assignment 5: Nathan Jacobi and Taylor Wetterhan







Assignment 5: Nathan Jacobi and Taylor Wetterhan



Our theoretical and experimental results for each algorithm coincided with what was expected. Each algorithm reached a similar experimental number of comparisons to the 10000 value theoretical comparisons provided in the files. None of them were more than a few thousand apart, and it could be expected that with more experiments, the averages could shift even closer between the theoretical and experimental values. Each of the experimental values also coincided with the theoretical Big-O complexities, confirmed by the number of digits. Selection sort is $O(N^2)$, and across all experiments, it reaches the number of comparisons expected. For the other algorithms, they are all supposed to be complexity O(Nlog(N)). The plots mostly follow the theoretical expected values, but there is some variance. These variances are likely due to a low sample size in experiments, and if more values were considered into the average, they would likely be more accurate.

Conclusion

Through the experiments performed and analysis of the various algorithms, it can be determined that in most circumstances merge sort is the most efficient algorithm. However, to confirm this, future experiments should be performed with various types of unsorted arrays and other sorting algorithms. For example, near sorted arrays, arrays with excessive repeating values, etc. Other algorithms that could be included could be bubble sort, binary tree sorting, and bogo sort. These further experiments could confirm the findings here.