Sorting Algorithm Writeup

Dan Haub
CPSC 350-01

I. Introduction

In the realm of computer science, sorting algorithms are used constantly for many reasons such as analyzing data, increasing search time, and simply keeping a list of data organized [1]. Because of this, the efficiency of the chosen algorithm is extreemly important; it needs to be fast and light on memory while, sometimes while being simple enough to implement in limited time. Because of this, each algorithm has its benifits and disadvantages that need to be taken into account.

II. Testing Methods and Data

Four different sorting algorithms were tested: bubble sort, insertion sort, tree sort, and quick sort. Each algorithm was tested in the same way with the same randomly generated dataset for varying element counts. Each count was tested four times for each algorithm and the average time in microseconds (μs) among those four trials was recorded.

TABLE I
Algorithm Times

Number of Elements	Algorithm Times (μs)			
	Bubble	***Insertion***	***Tree***	***Quick***
10	1	0	1	1
50	22	6	6	5
100	63	19	12	10
500	1558	449	89	65
1000	6057	1796	176	185
5000	162218	45688	1549	974
10000	677184	179538	3231	1929
50000	17350403	4447510	26673	11825
100000	69677364	17856790	52319	26356

Fig. 1. Here the runtimes of each algorithm are shown. This shows the stark difference between $O(n^2)$ algorithms and O(n log(n)) algorithms as well as the differences between algorithms with the same worst case time complexity.

Note: This form of analysis, empirical analysis, although accurate, is impractical in examining the efficiency of algorithms because of its variability. Had these calculations been run on a more powerful computer or even at a different time, the trends in the data would have been similar but the exact times would have been different. Additionally, setting up the analysis took a significant amount of time that might not be available in practice.

III. Algorithm Analysis

A. Bubble Sort

Bubble sort is a very simple algorithm that has a time complexity of $O(n^2)$. It functions by checking each pair of adjacent elements in order and swapping them if they are in the wrong order relative to each other. This is repeated until the entirety of the list has been sorted. This algorithm is very slow as it will never make use of a partially sorted list. This caused the algorithm to take close to 70 seconds at only 100,000 elements which, compared to the other algorithms was incredibly slow.

B. Insertion Sort

Insertion sort is also very simple and, like bubble, has a worst case time complexity of $O(n^2)$, however it is able to make use of a sorted list, having a best case of $\Omega(n)$. It is able to do this because it works in a manor similar to sorting playing cards: each element is sequentially ”inserted” into a sorted partition at the front of the list. If the next element to insert happens to belong at the end of the sorted partiton it will stay in place. This allows insertion sort to run significantly faster than bubble sort, taking just over 17 seconds on average at 100,000 elements, 24% the time of bubble.

C. Tree Sort

Tree sort is simple in concept but is significantly more dificult to implement from scratch than the previous two algorithms, however that complexity allows for a durastic increase in efficiency over the two algorithms before it: an average time complexity of O(n log(n)). This algorithm works by inserting all of the list's elements into a binary search tree and performing an in order traversal of the tree to repopulate the initial list. The two main drawbacks of this algorithm are its worst case time complexity of $O(n^2)$ that only occurs for a sorted or nearly sorted list (and can be fixed using a self balancing tree) and its space complexity of O(n). Because of its good time complexity in the average case, 100,000 elements was no problem for it only taking 0.5 seconds.

D. Quick Sort

Of the algorithms tested, quick sort was the most efficient by far but also the most complex. It involves choosing a pivot element such after some element swaps, it will be in its place and everything on either side of the pivot will belong on that side. This process is then repeated for the two sub lists on either side of the pivot. This gives the algorithm a very rare worst case of $O(n^2)$ in the case that every single pivot chosen is either the largest or smallest element in its list. However, the average case, like tree sort, is O(n log(n) and in practice runs faster than tree sort and has a better space complexity of O(1). The 100,000 benchmark for this algorithm was completed in just 0.26 seconds.

IV. Conclusion

For significantly large numbers of elements, time complexity of a sorting algorithm is extreemly valuable and all but required in today's world with extreemly large datasets being commonplace. A time complexity of O(n log(n)) is durastically more efficient than $O(n^2)$; in this test, the former about 0.3% of the time of the later at 100,000 elements. However, there are sacrifices in choosing a faster, yet more complex algorithm as they might use a larger amount of memory that might not be available on smaller systems, or the cost of the greater implementation time can't be paid. Quick sort was much more difficult and time consuming to implement than bubble sort and insertion sort, and tree sort uses double the memory of the other three algorithms. The fastest algorithm might not always be the best one for the job.

References

[1] . Kaushal R. Why Sorting is so Important in Data Structures. International Journal for Scientific Research and Development. 2017.