Sophie Samuels

Z23748744

COP 3410

Assignment 2

April 1, 2024

Introduction:

Sorting is a process we encounter daily. Some common uses of sorted datasets include ability to view items by price or rating on e-commerce websites, student course registration slots determined by credits earned, and 5k race results listed by alphabetic order and/or finish time. Sorting is a problem in computer science that has been heavily studied resulting in the development and analysis of a wide variety of sorting algorithms.[5]

Efficient sorting algorithms are necessary for a range of applications and industries including "data organization, searching, and data analysis". Each sorting algorithm that is developed carries its own features and performance trade-offs. It is often challenging to select the most appropriate sorting algorithm for a given task and the decision can depend on multiple factors including but not limited to the size of the dataset, data distribution, and computational resources available.[6] This action of sorting datasets is crucial to maximizing the runtime efficiency for locating values within datasets. Three simple implementations of sorting algorithms typically used for small data sets are known as bubble sort, selection sort, and insertion sort.

The bubble sort is the simplest of the sorting algorithms, though it is primarily used for educational purposes due to its notoriously slow sorting rate.[2] It is comprised of two nested loops and a nested if statement.[6] The bubble sort algorithm is aptly named as it "bubbles out" the smallest or largest element from an array dependent on whether the array needs to be "sorted in descending or ascending order respectively". It compares each element in an array with the element immediately next to it, then swaps the elements if they are not in the desired order. This described operation will perform 'n-1' times or until it passes through the entire array without any swaps, indicating a fully sorted array.[4]

The algorithm makes "about $\frac{N^2}{2}$ comparisons…there are fewer swaps than comparisons because two elements are swapped only if they need to be." With random data, swapping elements is necessary about half of all comparisons, "so there would be about $\frac{N^2}{4}$ swaps…both swaps and comparisons are proportional to N^2".[2] The actual number of swaps is dependent upon the initial order of elements, with best-case zero swaps for perfectly sorted data. This algorithm is easy to apply and for an array that is already sorted, the best-case time complexity is O(n). Overall, it is known to be inefficient and unsuitable for large data sets with a typical complexity of $O(n^2)$.[4]

The selection sort algorithm selects either the smallest or largest element of a list dependent on whether an array is sorted "in ascending or descending order respectively". It swaps the selected element with the element in the first position of the array, then the second ranked element is swapped with the element in the second position of the array and so on until all elements are in the desired order.[4] Similar to the bubble sort, this algorithm requires two nested loops, however, the outer loop iterates over an entire array and divides the array into a sorted and an unsorted unit.[2]

The algorithm performs about $\frac{N^2}{2}$ comparisons to sort an array and the number of swaps is much lower than the number of comparisons made. Regardless of the data's original order, the number of swaps will always be N-1 and the number of comparisons made will remain proportional to N^2, marking a major difference in efficiency between this and the bubble sort algorithm.[1] Therefore, the selection sort algorithm does not serve any advantage from a perfectly ordered data set. The best and worst-case time complexity of this algorithm is $O(n^2)$. Although the best-case run time is worse than that of bubble sort, it is still more efficient for sorting arrays because there are O(n) swaps instead of $O(n^2)$ swaps made.[2]

The insertion sort algorithm starts at the second element of an array, considering the first element as already have been sorted. The current element iterated over is compared with the element(s) in the sorted unit of the array, then placed at the correct position in the sorted unit, which can require shifting elements of the sorted unit. This algorithm requires two loops, one 'for' loop which iterates over each element of an array and one 'while' loop which inserts each element in the sorted array unit.[4] For most data sets, insertion sort performs about twice as fast as the sort algorithm and to some degree faster than the selection sort algorithm.[2]

For random data, the insertion sort algorithm has a time complexity of $O(n^2)$ and for data that is approximately or fully sorted, the algorithm has a time complexity of closer to O(n).[1] This occurs as data that is already ordered never results in a True condition from the while loop, so the outer loop primarily iterates and executes N-1 times.[2] Two important disadvantages to note is that the algorithm's efficiency deteriorates at a high rate as data size increases and the time complexity is quadratic in worst-case when an array is originally sorted in reverse order.[4]

Method Analysis:

To further analyze the time complexity of the bubble sort, selection sort, and insertion sort algorithms, it is necessary to generate both random and pre-sorted data sets with a running time test to allow quantification of the running time on a specific computing system for varied sizes and arrangements of input data. In this analysis, the data sets used to test each algorithm included a list of random integers between the closed interval [0, 50010] ranging in length from 10 to 50,000. The random integers were generated using the import random module in python, which "implements pseudo-random number generators for various distributions".[6] Due to the limitations of the hardware used to test the algorithms, the maximum list size was set at 50,000 elements. The execution or running time was quantified for each data set and algorithm using python's time module. The code for this module calculates the execution time (in seconds as a floating-point number) as the absolute value of the difference between the time recorded directly before and the time recorded directly after the execution of the sorting algorithm. Scatter plots representing each set of results of the individual algorithms for both random and sorted data were created using Matplotlib comparing list size and time taken (in seconds) for completion.

Data:

Time Analyses for Random Data Sets

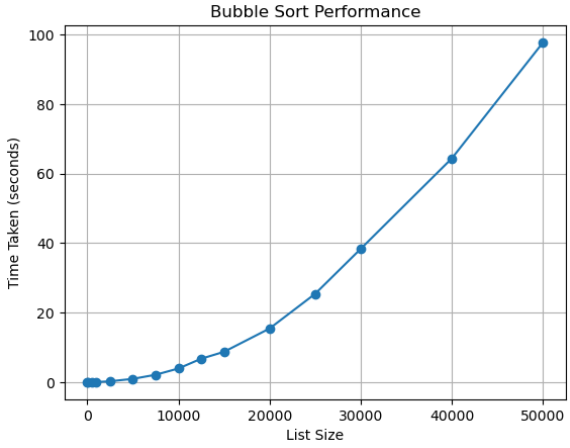

Figure 1: The above scatterplot shows the running time in seconds for a list of size n using the bubble sort algorithm. The input size is varied from 0 to 50,000. The datasets are generated using a pseudorandom method.

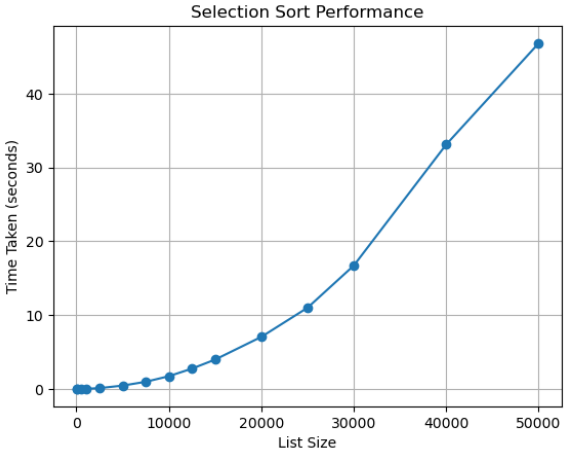

Figure 2: The above scatterplot shows the running time in seconds for a list of size n using the selection sort algorithm. The input size is varied from 0 to 50,000. The datasets are generated using a pseudorandom method.

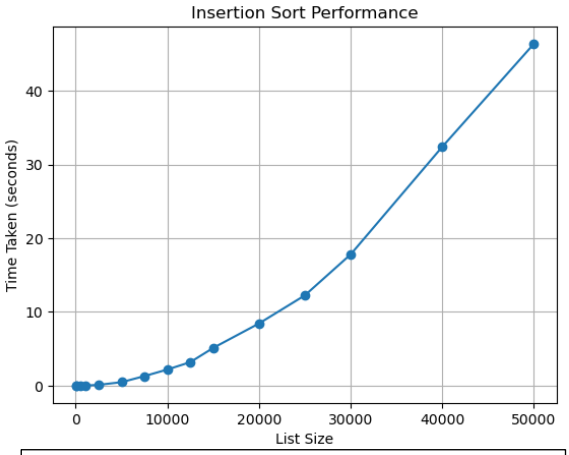

Figure 3: The above scatterplot shows the running time in seconds for a list of size n using the insertion sort algorithm.The input size is varied from 0 to 50,000. The datasets are generated using a pseudorandom method.

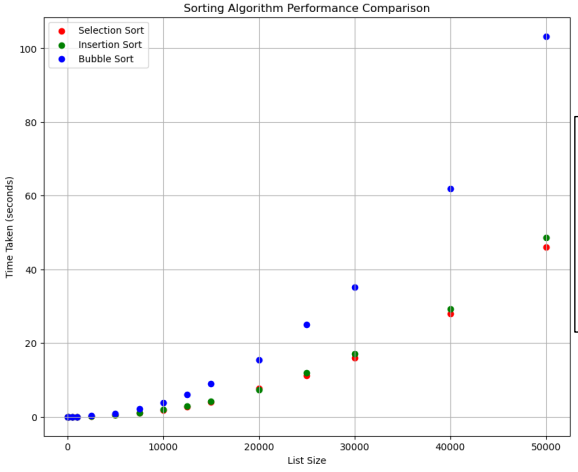

Figure 4: The scatterplot to the left shows the running time in seconds for a list of size n comparing bubble, selection, and insertion sort algorithms. The input size is varied from 0 to 50,000. The datasets are generated using a pseudorandom method.

Time Analyses for Sorted Data Sets

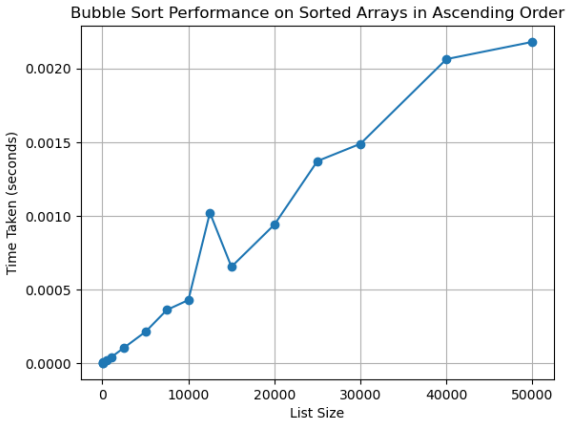

Figure 5: The above scatterplot shows the running time in seconds for a list of size n using the bubble sort algorithm. The input size is varied from 0 to 50,000. The datasets are generated using a sorted method.

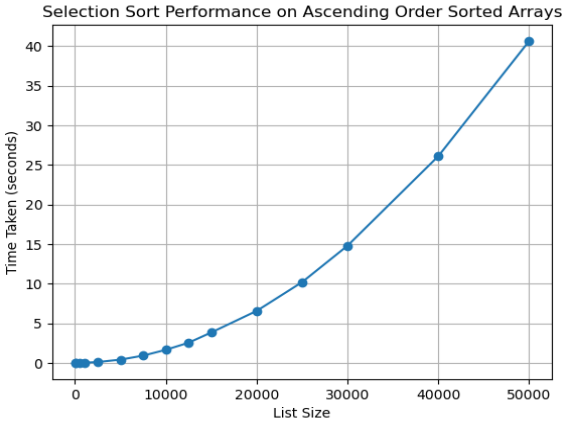

Figure 6: The above scatterplot shows the running time in seconds for a list of size n using the selection sort algorithm. The input size is varied from 0 to 50,000. The datasets are generated using a sorted method.

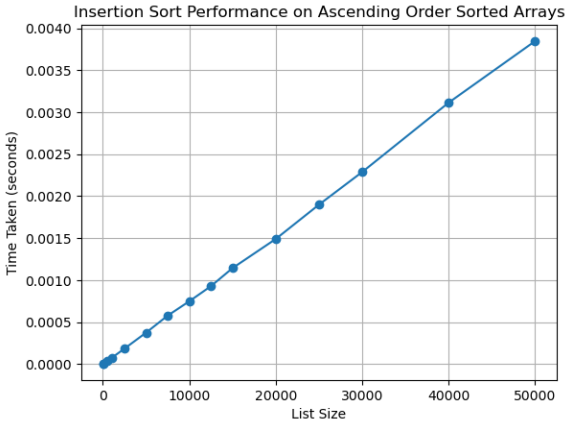

Figure 7: The above scatterplot shows the running time in seconds for a list of size n using the insertion sort algorithm. The input size is varied from 0 to 50,000. The datasets are generated using a sorted method.

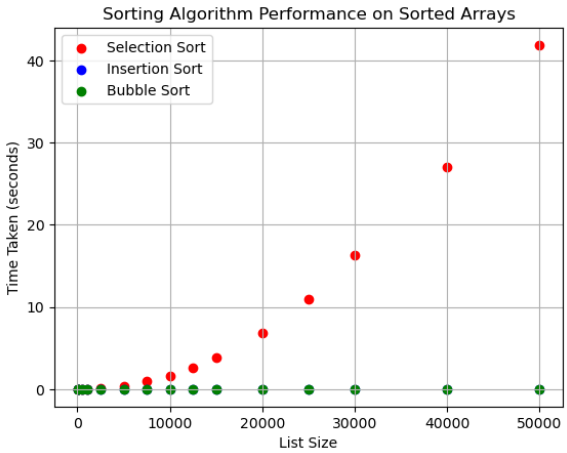

Figure 8: The above scatterplot shows the running time in seconds for a list of size n comparing bubble, selection, and insertion sort algorithms. The input size is varied from 0 to 50,000. The datasets are generated using a sorted method. The insertion sort is not visible due to its comparatively efficient runtimes compared to the other algorithms.

Analysis & Conclusions:

As the data in *Figure 1, Figure 2,* and *Figure 3* show, the bubble sort, selection sort, and insertion sort algorithms each perform in $O(n^2)$ time complexity for unsorted datasets. Although all three algorithms have similar asymptotic growth rates, a closer look at the 'time taken' scale of the graphs demonstrates clear distinctions between the performance of each for sorting random data. The selection sort and insertion sort algorithms display similar execution times for both small and large inputs with selection sort having slightly faster performance with (smaller) inputs from size 10 through 30,000 at which point insertion sort overtakes selection sort in performance (for larger inputs). In contrast, the bubble sort algorithm in *Figure 3* takes approximately twice the length of time to perform the same ascending order sorting tasks as both the selection and insertion sort algorithms.

When comparing the results of the pre-sorted data sets, the selection sort data in *Figure 6* displays the selection sort algorithm as having the poorest performance of all three algorithms for sorting ordered arrays with a time complexity of $O(n^2)$. The bubble sort algorithm data in *Figure 5* also indicates a time complexity of $O(n^2)$ for ordered arrays, but with significantly less execution times than seen with the selection sort algorithm. The insertion sort algorithm data in *Figure 7* reveals a more efficient time complexity of $O(n)$ for ordered arrays with suitability for small and large data sets. When considering the dynamic complexities of real-world sorting algorithm applications, it is important to consider all factors including data input size, the ordered and unordered nature of data sets, and performance of each algorithm under varied conditions. This data demonstrates that the insertion sort algorithm is optimal given its efficiency with both unsorted and sorted data sets as well its comparatively acceptable handling of both small and large data sets. Given this handling of varied data sets and its ability to differentiate ordered and unordered data sets, this algorithm is more suitable for a variety of real-world applications. Comprehensive analysis of algorithm performance with varied data sets including the evaluation of best and worst-case scenarios is necessary to identify the advantages and disadvantages of the algorithms.

References:

1. Bible, P.W. and Moser, L. (2023) in *An Open Guide to Data Structures and Algorithms*. Indianapolis, Indiana: Private Academic Library Network of Indiana, Inc., pp. 62–87.

2. Canning, J., Lafore, R. and Broder, A. (2022) in *Data Structures & Algorithms in Python*. 1st edn. Pearson Education, pp. 77–87.

3. GfG. (2024, March 14). *Python program for Bubble Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/python-program-for-bubble-sort/

4. Kaur Gill, S. *et al.* (2018b) 'A Comparative Study of Various Sorting Algorithms', *International Journal of Advanced Studies Scientific Research (IJASSR)*, (Special Issue based on proceedings of 4th International Conference on Cyber Security (ICCS)), pp. 367–372.

5. Nasar, A. (2019) 'A Mathematical Analysis of Student-Generated Sorting Algorithms', *The Mathematics Enthusiast*, 16(1), pp. 315–330.

6. Real Python. (2023, June 9). *Generating random data in python (guide)*. https://realpython.com/python-random/

7. Sabah, A.S. *et al.* (2023) 'Comparative Analysis of the Performance of Popular Sorting Algorithms on Datasets of Different Sizes and Characteristics', *International Journal of Academic Engineering Research (IJAER)*, 7(6), pp. 76–84.