A logo with text on it

Description automatically generated

**Project Report - Empirical Analysis of Common Sorting Algorithms**

**Data Structures and Algorithms (CSCI2226) with Professor Adrian Rusu**

**Kevin Harper**

**Electrical and Computer Engineering Department**

**University of New Haven Tagliatela College of Engineering**

**September 18, 2024**

**Table of Contents**

**Introduction2**

**Theory3**

Bubble Sort4

Selection Sort5

Insertion Sort6

Merge Sort7

Quick Sort9

Shell Sort11

Summary of Implementation Details12

**Results13**

Raw Data13

Speed Comparisons Between Algorithms14

Graphs for Unsorted Data16

Graphs for Presorted Data17

Graphs for Descending Order Sorted Data18

**Discussion & Analysis18**

**Conclusion19**

**Introduction**

This project entails the C++ implementation and empirical time complexity analysis of six array sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Shell Sort. The data to facilitate analysis of the time efficiencies are standard (C-style) arrays which vary in size from 50,000 to 550,000. The array members include non-duplicate random integers in the range [1, 4,000,000]. Dataset generation is performed by selecting several thousand integers in the range, with each value having the probability of being selected according to a uniform distribution. Thereafter, the sets of integers are stored in a dedicated file for recurring analysis. Unique integer values are used since some of the sorting algorithms are unstable. Each sorting algorithm uses identical datasets for a fair comparison of run-times between sorting algorithms. The run-time duration for each algorithm is sampled five times per each of the datasets and averaged. A program is developed which interacts with the user via a console interface, allowing options for dataset generation, sorting algorithm selection, dataset size selection, dataset type selection (unsorted, sorted in increasing order, sorted in decreasing order), displaying the configuration, and performing numerous consecutive sorts without requiring any additional user input. When selecting a file for sorting or attempting to perform a sort, the user will be prompted to generate the appropriate dataset given it does not yet exist. Upon performing any sort, if no file exists for the sorted datasets, the user will again be prompted to store the sorted data in increasing order, decreasing order, or both for later analysis, offering true best-case and worst-case analysis of the algorithms. Topologically, the program employs the use of a superloop with an embedded switch case for option selection via a console interface. The user may quit the program at any time from the main menu by configuring a “quit” character and rebuilding the executable, or by using the default option (‘x’). The user may also cancel their selection returning to the main menu from any other submenu using this same character. Throughout the program’s execution, useful information is printed to the console to help keep track of which sorting algorithm is in use as well as which dataset is being sorted.

**Theory**

In general, a sorting algorithm’s performance and applicability is quantified via its time and space complexities. These measures of performance scale proportional to the data input size. Efficiencies are typically measured according to big-O notation, which asymptotically classifies the performance of an algorithm as its input size grows, providing an upper bound for what to expect in terms of performance. Common time complexities in big-O notation for sorting algorithms are linear (O(n)), quadratic (O(n2)), and linearithmic (O(nlog(n))). For example, for an algorithm exhibiting linear time complexity, the run-time would scale linearly relative to the growth of the input size. In terms of spatial efficiency, some algorithms sort “in place” such that little or no memory is needed for the algorithm’s execution, while others require additional memory to perform the sort.

Beyond average time or space efficiency measures, there are other considerations which are important in selecting the appropriate sorting algorithm for a given application. One such consideration is the algorithm’s stability. A sorting algorithm is said to be stable if it maintains the relative order of data members with equal values after sorting. Additionally, the expected form of the input data (e.g. nearly sorted) could influence the choice of a particular algorithm as the efficiency measures can be extended to consider the best, worst, and average cases. Therefore, a programmer’s knowledge of the input data should influence the appropriate choice of sorting algorithm. Finally, some algorithms, such as Quick Sort and Shell Sort, have various implementations which offer slightly different performances. These might be in the form of a routine fundamental to the algorithm’s operation, such as choosing how to subdivide an array into smaller arrays, or it might be an iterative variation of a recursive algorithm. Nonetheless, it’s important to understand that there are numerous implementation methods for the same algorithmic description.

This project explores comparison-based sorting algorithms. One comparison-based algorithm which was not explored is Heap Sort. Additionally, non-comparative approaches to sorting exist. For example, Counting Sort, Radix Sort, and Bucket Sort are algorithms which do not employ the use of comparison to sort the input data.

A description of each of the implemented sorting algorithms follows.

*Bubble Sort*

Bubble Sort is an easily understandable array sorting algorithm which uses a pair of nested loops. The two loops work together to compare values at adjacent indices in a pairwise fashion. The outer loop selects a value at a smaller index, and the inner loop swaps the elements so long as the element in the smaller index is larger in value than the element at the larger index. After each iteration, the largest encountered value will shift indices until the end of the array, and a pointer indicating the end of the array is decremented to reflect the increase in size of the sorted portion of the array. In each iteration, comparison takes place until the last unsorted element is encountered. Pseudocode describing the Bubble Sort algorithm is shown below in figure 1.

A screen shot of a computer program

Description automatically generated

**Figure 1.** Pseudocode representation of the Bubble Sort algorithm.

As indicated in figure 1 above, the common operation in Bubble Sort is comparison. Since the algorithm employs nested loops requiring n(n-1)/2 comparisons, its theoretical average time complexity is O(n2). However, the algorithm can be optimized through an early termination of the outer loop given that no swapping takes place. This optimization results in O(n) time complexity in the best-case scenario; however, the optimization is not implemented in this project.

*Selection Sort*

The Selection Sort algorithm, like the Bubble Sort algorithm, also uses nested loops to sort an array. In the chosen implementation, the outermost loop initially considers the first array index to be the index containing the minimum value in the array. The inner loop then iterates through every other array index, comparing values to the initial minimum and updating the pointer to a new minimum value as necessary. When the inner loop terminates, given that the pointer to the minimum value has changed, the value at the minimum pointer location in the array is swapped with the first position in the unsorted portion of the array. As the outer loop iterates, the sorted portion of the array grows from index 0 onwards, and the inner loop iterates for fewer array indices as the beginning portion of the array is sorted.

A screen shot of a computer program

Description automatically generated

**Figure 2.** Pseudocode representation of the Selection Sort algorithm.

Figure 2 shows that the most frequent operation performed in the Selection Sort algorithm is comparison. While swapping values has the potential to occur frequently, comparisons are guaranteed to occur most often. As was the case for Bubble Sort, the number of comparisons made is n(n-1)/2. Therefore, the performance of Selection Sort is O(n2), which is also expected given the presence of 2 nested loops.

*Insertion Sort*

Insertion Sort uses nested loops in a similar manner to the Bubble Sort and Selection Sort algorithms. In the outer loop, each element i (for which i > 0) of the array is used as a key which is to eventually be inserted at the correct position of the array. The inner loop iterates backwards through the sorted portion of the array (indices [0:i]) making comparisons with the key value until the correct insertion position is found, simultaneously making space for the key value to be inserted by pushing sorted values to the right. When the insertion position is found via iterative comparison, the inner loop terminates, at which point the loop variable has been decremented to the index where the key value should be placed. Finally, the key value is placed at the correct position in the array.

A screen shot of a computer program

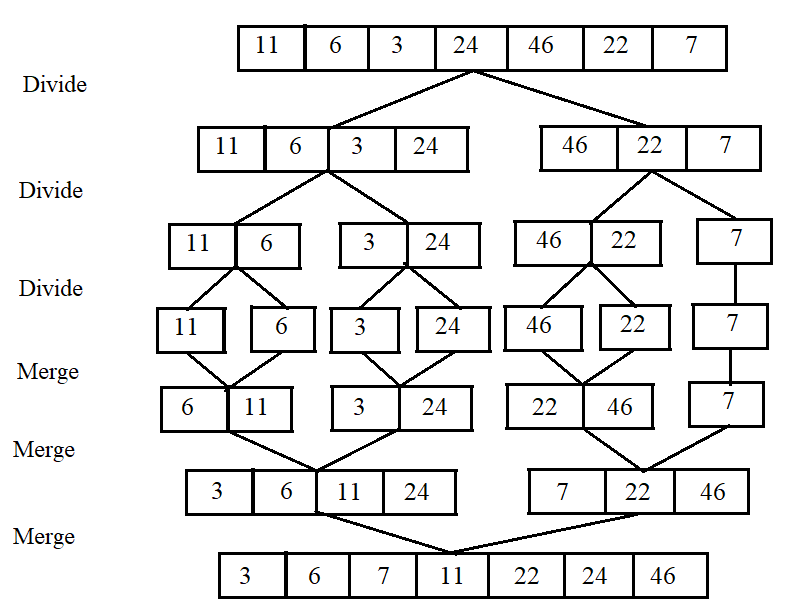
Description automatically generated

**Figure 3.** Pseudocode representation for the Insertion Sort algorithm.

Figure 3 shows that the most frequent operation performed in the Insertion Sort algorithm is comparison. For arrays that are nearly sorted, observe that the inner loop will terminate more rapidly, significantly increasing time efficiency. Again, since the number of comparisons made in the average case is proportional to n2, the average performance of Insertion Sort is O(n2).

*Merge Sort*

The Merge Sort algorithm uses a divide and conquer approach to recursively sort an array. The base case checks if the array length is 1. Otherwise, the array is recursively divided into two halves, starting from the center of the array, until the base case is ultimately reached. Finally, each of the sub halves are individually merged together, taking care to join them while establishing order.



**Figure 4.** Visualization of the Merge Sort algorithm with an example input array. The final division step, resulting in the subarrays having one element each, represents the point at which the base case is reached. Note that the left half of the array will be separated entirely into its trivial representation prior to the right half beginning to be subdivided. This is because there are two sequential recursive calls in the algorithm (see figure 5). The second recursive call (on the right half) is not executed until recursion associated with the left half of the array terminates.

A computer screen shot of a computer code

Description automatically generated

**Figure 5.** Pseudocode representation for the Merge Sort algorithm.

**A screenshot of a computer program

Description automatically generated**

**Figure 6.** Pseudocode representation for the merge operation in the Merge Sort algorithm.

As indicated in figures 4-6, element sorting occurs during the merge process, after the arrays have been divided into n trivially sorted units, where n is the input size to the algorithm. The merging process inherently involves many comparisons to determine the necessary order of the original elements. Hence, the basic operation in the Merge Sort algorithm is comparison. The number of comparisons made during execution of Merge Sort can be found by solving a recurrence relation and employing the use of the master theorem. It can be shown that the theoretical average time complexity for Merge Sort is linearithmic (O(nlog(n))). Note that unlike the previously discussed algorithms, Merge Sort does not sort-in-place. It requires extra memory to perform its work. The spatial complexity of Merge Sort is linear (O(n)).

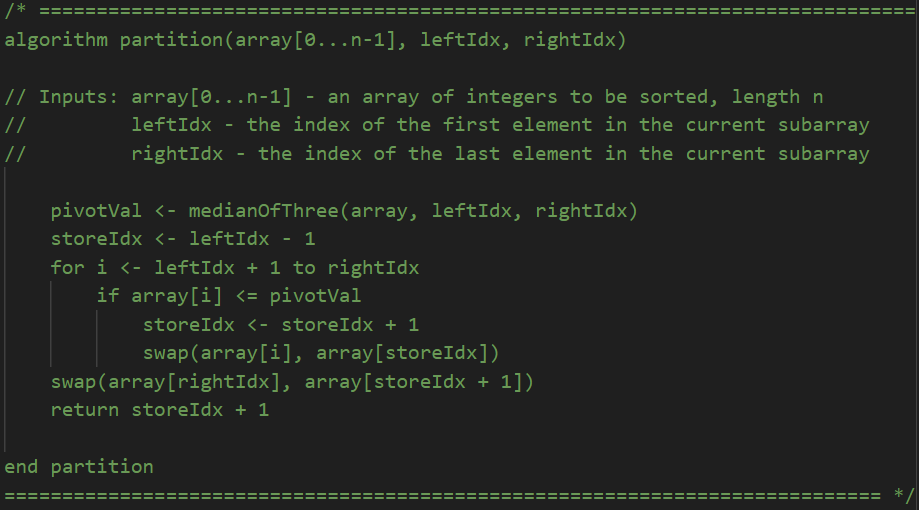
*Quick Sort*

The Quick Sort algorithm, like Merge Sort, is based on a divide and conquer approach. The procedure is conceptually similar to Merge Sort. However, the subdivisions of the array are obtained differently. The algorithm partitions an array into numerous subarrays according to a chosen pivot selection scheme. This partitioning is done in such a way that a pivot element is placed in its final sorted position after the operation. The pivot index is selected such that elements greater than the pivot element are in larger indices (to the right of the pivot), whereas elements less than the pivot element are in smaller indices (to the left of the pivot). Each of the left and right halves are recursively divided according to the same pivot selection scheme until each subarray contains a single element (the recursive base case), which is trivially sorted. Unlike Merge Sort, Quick Sort does not require a separate recombination step, as the elements are already arranged in their correct relative order after partitioning.

A computer screen shot of a computer code

Description automatically generated

**Figure 7.** Pseudocode representation for Quick Sort algorithm.



**Figure 8.** Pseudocode representation for partition operation in Quick Sort algorithm.

A screenshot of a computer program

Description automatically generated

**Figure 9.** Pseudocode representation for median of three operation in partition operation.

The pseudocode in figures 7-9 indicate that the partitioning mechanism in the chosen implementation uses a “median of three” pivot selection. This means that in each subdivision of the array, the element values are compared at the beginning, middle, and end of the subarray, and the median value of the three elements is used as the pivot. This method was chosen since segmentation faults arose with a simpler pivot selection scheme. Some other pivot selection methods include first, middle, or last element selection, random selection, median-of-medians, and dual-pivot (used in Java’s .sort() method). The time complexity of the Quick Sort algorithm is heavily dependent on how well the pivot element segments the data. In this case a recursive relation can be solved using the master theorem to see that the average time complexity is linearithmic (O(nlog(n))). Additionally, the algorithm is more efficient in terms of spatial complexity than Merge Sort since it works in-place without requiring additional arrays for merging. The spatial complexity of Quick Sort is O(logn).

*Shell Sort*

The Shell Sort algorithm is a generalized form of the Insertion Sort algorithm which first sorts elements that are far apart. As the algorithm executes, the sorting interval decreases in size until a standard Insertion Sort is performed in the final iteration (interval size of 1). However, at this point, the array is nearly sorted, which greatly increases the efficiency of the single execution of the traditional Insertion Sort.

A screenshot of a computer program

Description automatically generated

**Figure 10.** Pseudocode representation for the Shell Sort algorithm.

In figure 6 above, the pseudocode representation of the Shell Sort algorithm indicates that the outer loop progressively decreases the interval size, initially targeting array elements in the inner loop that are farther apart and gradually approaching a standard Insertion Sort. The chosen implementation uses Shell’s original sequence, which halves the interval size for each iteration. Some other sequences include Hibbard’s, Tokuda’s, and Sedgewick’s intervals, which are more complex, but may offer more optimal performance in the worst-case. The exact time complexity for Shell Sort is highly dependent on the gap sequence used. In the case of Shell’s original sequence, the average time complexity is linearithmic (O(nlog(n))).

 **Table 1.** Theoretical time and space complexities for each sorting algorithm, implementation details, and stability classification.

**Results**

**Table 2.** Raw and average data obtained from numerous executions of each sorting algorithm for each dataset size.

**Table 3.** Comparison of Shell Sort with Insertion Sort on presorted data, omitting the logarithmic factor inherent to Shell Sort’s time complexity which arises from the presort operation at intervals in increasing factors of 2.



**Table 4.** Sorting algorithm speeds relative to one another for unsorted input data.



**Figure 11.** Comparison of empirical time complexities on randomly sorted data.

****

**Figure 12.** Comparison of empirical time complexities on randomly sorted data (zoomed).



**Figure 13.** Comparison of empirical time complexities on increasing order sorted data.



**Figure 14.** Comparison of empirical time complexities on increasing order sorted data (zoomed).



**Figure 15.** Comparison of empirical time complexities on descending order sorted data.



**Figure 16.** Comparison of empirical time complexities on descending order sorted data (zoomed).

**Table 5.** Linearithmic runtime approximation for Merge Sort, Quick Sort, and Shell Sort on presorted data, obtained using Excel’s Solver.

****

**Discussion & Analysis**

* Analyze information in results section and explain why some algorithms perform better than others.
* Talk about results for best case and worst case and how they relate to random case.
* which algorithm performs five times as fast as the slowest algorithm and for what data set size? how about twenty times, a hundred times?

The results show that Merge Sort, Shell Sort, and Quick Sort are the three best performing algorithms amongst the six examined. As indicated with the equations of best-fit lines in each of the figures, these algorithms have a consistent linear time complexity (O(n)) regardless of the form of the input data (sorted, unsorted, reverse sorted). Quick Sort and Shell Sort are very closely related in terms of performance, whereas Merge Sort grows at a slightly faster rate in all situations. Interestingly, both Quick Sort and Shell Sort perform better when the data is sorted in reverse order than when the data is unsorted. In general, Bubble Sort, Selection Sort, and Insertion Sort exhibit quadratic time complexities (O(n2)).

Regarding the presorted data (figures N:N), Insertion Sort is the best performing algorithm, showing approximately linear growth. In the worst (descending order sorted) and average cases (unsorted), the Insertion Sort algorithm instead has quadratic time complexity. This is because Insertion Sort performs well with nearly sorted data, requiring fewer insertions. As mentioned, this is the reason why Shell Sort achieves consistent performance: it is an optimized version of the Insertion Sort algorithm. When the data is presorted, both Bubble Sort and Selection Sort achieve similar performance, indicating that the choice between the two in situations where data is nearly sorted should be made according to user preferences and space complexities.

When the data is sorted in descending order, the performance of Selection Sort exceeds that of Insertion Sort, whereas in all other cases, Insertion Sort was found to outperform Selection Sort. Unintuitively, Bubble Sort, Quick Sort, and Shell Sort are actually found to perform better when the data is sorted in reverse order, despite requiring a greater number of swaps for array elements.

For input data in various forms, Merge Sort and Quick Sort performance is relatively consistent. This indicates that the worst-case, best-case, and average-case time complexities for these sorting algorithms has a greater dependence on the implementation rather than the form of the input data. For these algorithms, there are numerous ways to achieve the divide and conquer approach, and some perform better than others.

For all dataset sizes, considered unordered input data, Selection Sort performs roughly 5 times as fast as Bubble Sort. As dataset size increases, Quick Sort approaches a performance roughly 5 times that of Merge Sort. For all dataset sizes, Quick Sort performs about 2 times faster than Shell Sort. As dataset size increases, the performance of Shell Sort approaches to be 2 times as fast as Merge Sort. For a dataset size of 550,000 Quick Sort performs 10,000 times faster than Bubble Sort. For an unsorted dataset size of 50,000, Merge Sort performs around 20 times faster than Insertion Sort. For a dataset size of 50,000, Shell Sort performs roughly 100 times faster than Selection Sort (114x). Additionally, for a dataset size of 50,000, Merge Sort is roughly 140 times faster than Bubble Sort.

Interestingly, Bubble Sort was found to perform better on data that was formed in descending order, despite requiring a far greater number of swaps to be performed. A similar statement can be made for Merge Sort, Quick Sort, and Shell Sort; they all perform better with descending order sorted data than for unsorted data. The performance of Selection Sort is relatively impartial to the form of the input data.

**Conclusion**

* should contain any thoughts you may have about your current findings, and suggestions for possible future enhancements and modifications.

**Citations**

[1] “Bubble Sort Algorithm.” *Programiz*, www.programiz.com/dsa/bubble-sort.

[2] “Selection Sort Algorithm.” *Programiz*, www.programiz.com/dsa/selection-sort.

[3] “Insertion Sort Algorithm.” *Programiz*, www.programiz.com/dsa/insertion-sort.

[4] “Merge Sort Algorithm.” *Programiz*, www.programiz.com/dsa/merge-sort.

[5] “Quicksort Algorithm.” *Programiz*, www.programiz.com/dsa/quick-sort.

[6] “Shell Sort Algorithm.” *Programiz*, www.programiz.com/dsa/shell-sort.

[7] “Merge Sort Algorithm: Working and Example of Merge Sort Algorithm.” *EDUCBA*, 5 Apr. 2023, www.educba.com/merge-sort-algorithm/.

mergesort recursion figure

[8] “9: Sorting Algorithms.” *Engineering LibreTexts*, Libretexts, 10 Mar. 2021, eng.libretexts.org/Courses/Folsom\_Lake\_College/CISP\_430%3A\_Data\_Structures\_(Aljuboori)/09%3A\_Sorting\_Algorithms.

General algorithm analysis

[9] Shell sequences https://www.mycplus.com/featured-articles/shell-sort-algorithm/