A logo with text on it

Description automatically generated

**Empirical Analysis of Common Sorting Algorithms**

**Data structures and Algorithms (CSCI2226)**

**Kevin Harper (00816532)**

**Electrical and Computer Engineering Department**

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

**September 18, 2024**

**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 input 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]. Unique integer values are used as some sorting algorithms are unstable. Dataset generation is performed by selecting the appropriate number of integers from this set according to a uniform distribution of integers in this range and thereafter stored in a dedicated file for recurring execution. Each sorting algorithm uses identical datasets for sorting for a comparison of run-times between sorting algorithms on identical data in terms of content and relative order. The run-time duration of each algorithm is sampled five times per dataset size and averaged accordingly for further analysis. 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 (unsorted, sorted in increasing order, sorted in decreasing order), displaying the configuration, and performing numerous consecutive sorts without requiring any additional user input. Upon performing any sort, if no file exists for the sorted datasets, the user will be prompted to store the sorted data in increasing order, decreasing order, or both for later analysis, offering true best-case and worst-case time complexity 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 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.

**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 memory is needed beyond the input data being sorted, 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. Finally, some algorithms, such as quick sort and shell sort, have various implementations which offer slightly different performances.

*Bubble sort*

Bubble sort is a simple and 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 up until the last unsorted element. 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’s source.

*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 of 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 indicated by the 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 implementation is comparison. For arrays that are nearly sorted, observe that the inner loop will terminate faster, significantly increasing the time efficiency. Again, the average performance of insertion sort is O(n2) as indicated by the 2 nested loops.

*Merge sort*

The merge sort algorithm uses the divide and conquer approach to recursively sort an array. The base case checks if the array length is 1 or less. Otherwise, the array is recursively divided into two halves until the base case is reached. Finally, each of the sub halves are individually merged and simultaneously sorted, taking care to join them while retaining order.

A computer screen shot of a computer code

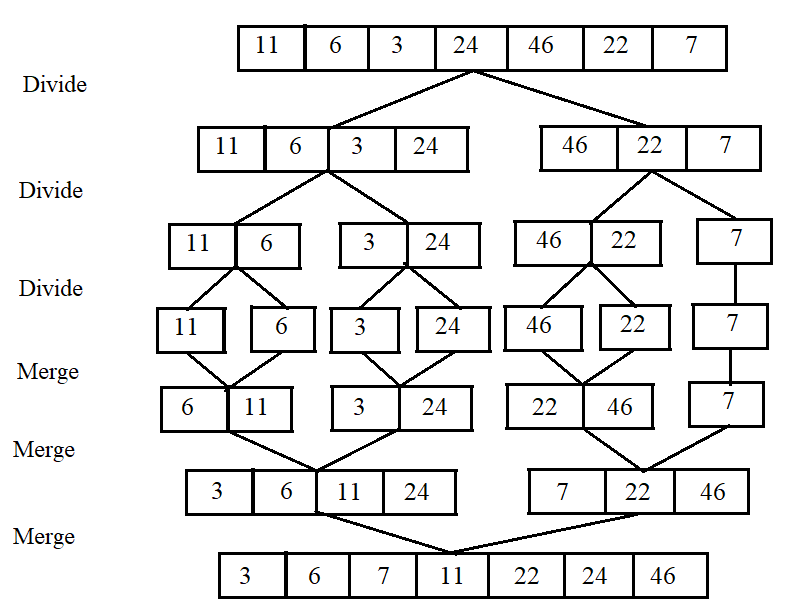
Description automatically generated

**Figure 4.** Pseudocode representation for the merge sort algorithm.

**A screenshot of a computer program

Description automatically generated**

**Figure 5.** Pseudocode representation for the merge operation in the merge sort algorithm.



**Figure 6.** https://www.educba.com/merge-sort-algorithm/

Include pseudocode for merge() sub algorithm. The actual merging of trivially sorted (length 1) arrays is where the sorting takes place. Elements in each half are placed in the correct position.

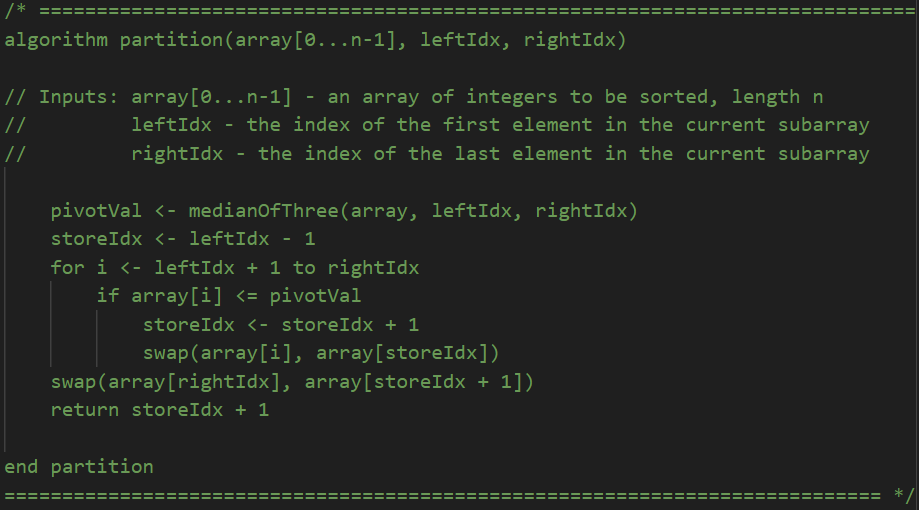
*Quick sort*

The quick sort algorithm, like merge sort, is based on a divide and conquer approach. The algorithm partitions an array into numerous subarrays according to a chosen pivot selection scheme. A 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, which is trivially sorted. Finally, the individual subarrays are recombined to constitute a sorted version of the input array.

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.

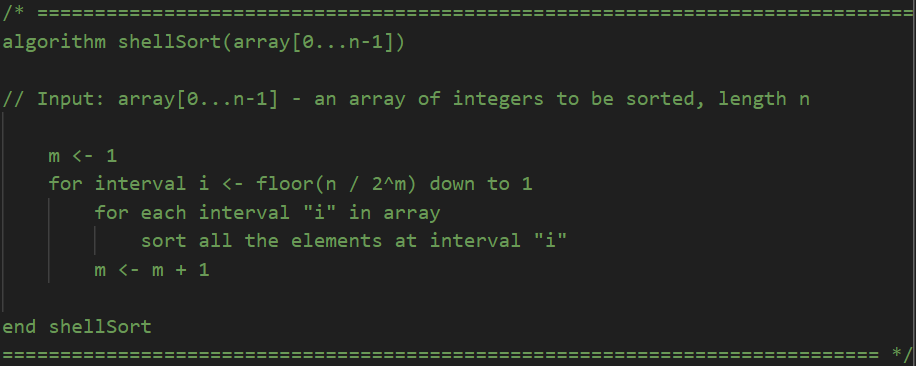
A screenshot of a computer program

Description automatically generated

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

*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 insertion sort.



**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.

To do?

* Clean up code and add comments.
* Make user options in code correspond to a custom class

 **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.** Sorting algorithm speeds relative to one another.

******Figure 11.** Always remember to wipe from front to back.

**Figure 12.** Always remember to wipe from front to back.



**Figure 13.** Always remember to wipe from front to back.



**Figure 14.** Always remember to wipe from front to back.



**Figure 15.** Always remember to wipe from front to back.



**Figure 16.** Always remember to wipe from front to back.

**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. 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.

**Conclusion**

**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