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**

**Abstract**

Words go here.

**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]. 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 the very same data and in the same 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. 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.

**Methods**

* Working explanation of each algorithm: what are theoretical best and worst cases? How do the algorithms work? Include algorithm pseudocode

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 black screen with green text

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 time complexity is O(n2). \*mention optimized bubble sort\*

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) as 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, the first element of the array is initially considered to be sorted. For the remaining elements, the chosen implementation compares each of the remaining array elements to the present minimum value, starting at the end of the array. Iterating backwards through the array, if a value at an index beyond the current minimum

A screenshot of a computer

Description automatically generated

**Figure 3.** Pseudocode representation for the insertion sort algorithm.

Merge sort

Quick sort

Shell sort

**Results**

**Discussion & Analysis**

**Conclusion**

**Citations**

[1] <https://www.programiz.com/dsa/bubble-sort>

[2] <https://www.programiz.com/dsa/selection-sort>

[3] <https://www.programiz.com/dsa/insertion-sort>

[4] <https://www.programiz.com/dsa/merge-sort>

[5] <https://www.programiz.com/dsa/quick-sort>

[6] <https://www.programiz.com/dsa/shell-sort>