University *of* New Haven

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

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 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. The runtime 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 best-case and worst-case analysis of the algorithms. Topologically, the program employs the use of a super loop 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 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. Common time complexities in big-O notation for sorting algorithms are quadratic ($O(n^2)$) and linearithmic ($O(nlogn)$). 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 [7]. 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 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 implementation. 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.

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.

```
/* ============================================================================
algorithm bubbleSort(array[0...n-1])

// Input: array[0...n-1] - an array of integers to be sorted, length n

    for i from 0 to n - 2
        for j from 0 to n - 2 - i
            if array[j] > array[j + 1]
                swap(array[j], array[j + 1])

end bubbleSort
============================================================================ */
```

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 $O(n^2)$. 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.

```
/* ============================================================================
algorithm selectionSort(array[0...n-1])

// Input: array[0...n-1] - an array of integers to be sorted, length n

    for i from 0 to n - 1 do
        minIdx <- i
        for j from i + 1 to n - 1 do
            if array[j] < array[minIdx]
                minIdx <- j
        if minIdx != i
            swap(array[i], array[minIdx])

end selectionSort
============================================================================ */
```

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 $O(n^2)$, 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.

```
/* ============================================================================
algorithm insertionSort(array[0...n-1])

// Input: array[0...n-1] - an array of integers to be sorted, length n

    for i from 1 to n - 1
        key <- array[i]
        j <- i - 1
        while j >= 0 and array[j] > key
            array[j + 1] <- array[j]
            j <- j - 1
        array[j + 1] <- key

end insertionSort
============================================================================ */
```

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 n^2, the average performance of Insertion Sort is $O(n^2)$.

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, taking care to join them while establishing order.

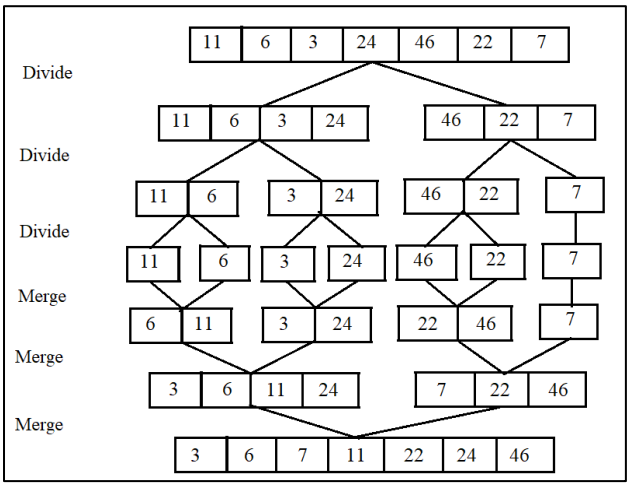

Figure 4. Visualization of the Merge Sort algorithm with an example input array [8]. The final division step, resulting in the subarrays having one element each, represents the point at which the base case is reached. Note that the left half of the array will be separated entirely into its trivial representation 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.

```
/* ============================================================================
algorithm mergeSort(array[0...n-1], leftIdx, rightIdx)

// Input: array[0...n-1] - an array of integers to be sorted, length n
//        leftIdx - the index of the first element in the current subarray
//        rightIdx - the index of the last element in the current subarray

    // Base case
    if leftIdx >= rightIdx
        return

    // Recursive case
    midIdx = (leftIdx + (rightIdx - leftIdx)) / 2
    mergeSort(array, leftIdx, midIdx)
    mergeSort(array, midIdx + 1, rightIdx)
    merge(array, leftIdx, midIdx, rightIdx)

end mergeSort
============================================================================ */
```

Figure 5. Pseudocode representation for the Merge Sort algorithm.

```
/* ============================================================================
algorithm merge(array[0...n-1], p, q, r)

// Inputs: array[0...n-1] - an array of integers to be sorted, length n
//         p - starting index of the first subarray
//         q - ending index of the first subarray
//         r - ending index of the second subarray

    // Determine the size of the two subarrays
    n1 = q - p + 1
    n2 = r - q

    // Create temporary arrays L and M
    create L[n1] containing elements array[p...n1 - 1]
    create M[n2] containing elements array[q + 1...n2 - 1]

    // Initialize pointers for L, M, and array[0...n - 1]
    i <- 0     // Pointer for L
    j <- 0     // Pointer for M
    k <- p     // Pointer for array

    // Merge the two subarrays back into array[0...n - 1]
    while i < n1 and j < n2 do
        if L[i] <= M[j] then
            array[k] = L[i]
            i = i + 1
        else
            array[k] = M[j]
            j = j + 1
        k = k + 1

    // Copy remaining elements from L[] into array[], if any
    while i < n1 do
        arr[k] = L[i]
        i = i + 1
        k = k + 1

    // Copy remaining elements from M[] into array[], if any
    while j < n2 do
        arr[k] = M[j]
        j = j + 1
        k = k + 1

end merge
============================================================================ */
```

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 for 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(nlogn)). 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.

```
/* ============================================================================
algorithm quickSort(array[0...n-1], leftIdx, rightIdx)

// Inputs: array[0...n-1] - an array of integers to be sorted, length n
//         leftIdx - the index of the first element in the current subarray
//         rightIdx - the index of the last element in the current subarray

    // Base case
    if (leftIdx >= rightIdx)
        return

    // Recursive case
    pivotIdx <- partition(array, leftIdx, rightIdx)
    quickSort(array, leftIdx, pivotIdx - 1)
    quickSort(array, pivotIdx + 1, rightIdx)

end quickSort
============================================================================ */
```

Figure 7. Pseudocode representation for Quick Sort algorithm.

```
/* ============================================================================
algorithm partition(array[0...n-1], leftIdx, rightIdx)

// Inputs: array[0...n-1] - an array of integers to be sorted, length n
//         leftIdx - the index of the first element in the current subarray
//         rightIdx - the index of the last element in the current subarray

    pivotVal <- medianOfThree(array, leftIdx, rightIdx)
    storeIdx <- leftIdx - 1
    for i <- leftIdx + 1 to rightIdx
        if array[i] <= pivotVal
            storeIdx <- storeIdx + 1
            swap(array[i], array[storeIdx])
    swap(array[rightIdx], array[storeIdx + 1])
    return storeIdx + 1

end partition
============================================================================ */
```

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

```
/* ============================================================================
algorithm medianOfThree(array[0...n-1], leftIdx, rightIdx)

// Inputs: array[0...n-1] - an array of integers to be sorted, length n
//         leftIdx - the index of the first element in the current subarray
//         rightIdx - the index of the last element in the current subarray

    midIdx = leftIdx + (rightIdx - leftIdx) / 2

    // Ensure array[leftIdx] <= array[midIdx] <= array[rightIdx]
    if array[leftIdx] > array[midIdx]:
        swap(array[leftIdx], array[mid])
    if array[leftIdx] > array[rightIdx]:
        swap(array[leftIdx], array[rightIdx])
    if array[midIdx] > array[rightIdx]:
        swap(array[midIdx], array[rightIdx])

    // Move the median (arr[mid]) to the end for use as the pivot
    swap(array[midIdx], array[rightIdx])

    // Return value of the pivot
    return array[rightIdx]

end medianOfThree
============================================================================ */
```

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) [9]. 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(nlogn)). 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.

```
/* ============================================================================
algorithm shellSort(array[0...n-1])

// Input: array[0...n-1] - an array of integers to be sorted, length n

    m <- 1
    for interval i <- floor(n / 2^m)
        for each interval "i" in array
            sort all the elements at interval "i"
        m <- m + 1

end shellSort
============================================================================ */
```

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 [10]. 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(nlogn)).

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

	Time Complexity			Space Complexity	Implementation	Stable?
Algorithm	**Best Case**	**Worst Case**	**Average Case**			
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	*O(1)*	Unoptomized	Y
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	*O(1)*	-	N
Insertion Sort	*O(n)*	$O(n^2)$	$O(n^2)$	*O(1)*	Iterative	Y
Merge Sort	*O(nlog(n))*	*O(nlog(n))*	*O(nlog(n))*	*O(n)*	Recursive	Y
Quick Sort	*O(nlog(n))*	$O(n^2)$	*O(nlog(n))*	*O(logn)*	Reursive with median of three pivot selection	N
Shell Sort	*O(nlog(n))*	$O(n^2)$	*O(nlog(n))*	*O(1)*	Iterative with Shell's standard sequence	N

Results

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

		Runtime (s)																	
		Unsorted Datasets						Sorted Datasets						Descending Order Sorted Datasets					
Algorithm	n	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Bubble Sort	50,000	3.85036	3.828359	3.909093	3.839268	3.845352	3.854486	0.927942	0.923372	0.926621	0.934939	0.924776	0.92752986	2.12776	2.15787	2.188321	2.209704	2.214287	2.179588
	100,000	16.56135	16.37457	16.37469	16.34533	16.32447	16.39608	3.757387	3.664429	3.643906	3.906801	3.935639	3.78163248	8.523964	8.729843	8.706711	8.770589	8.855833	8.717388
	175,000	48.60074	48.68062	50.59488	49.57839	49.76644	49.44421	11.56001	11.45693	11.41976	11.45762	11.71861	11.5225858	26.12888	26.10015	26.57199	26.6988	26.73292	26.44655
	300,000	152.0877	150.2969	150.5529	151.2256	150.0418	150.841	33.68493	33.6635	34.65956	34.5009	34.38343	34.1784622	80.36963	81.28262	81.87578	81.93025	80.68569	81.22879
	425,000	306.0327	306.9987	304.2535	310.6281	310.4573	307.6741	68.95934	68.85773	69.64143	68.38239	67.68102	68.7043808	164.4812	156.0698	156.6599	159.7566	160.4631	159.4861
	550,000	494.1216	501.7593	486.6197	494.6501	496.2276	494.6756	115.0557	114.6268	117.8628	117.619	116.9145	116.415772	270.1389	259.9609	263.6559	259.3742	260.89	262.804
Selection Sort	50,000	0.890611	0.861947	0.876385	0.878719	0.845987	0.87073	0.870286	0.894227	0.880751	0.871687	0.891891	0.88176838	0.889826	0.866951	0.90057	0.875984	0.887129	0.884092
	100,000	3.468087	3.494095	3.445623	3.440439	3.479148	3.465478	3.472	3.522698	3.51952	3.508506	3.538146	3.51217378	3.584262	3.46247	3.497783	3.55594	3.662364	3.552564
	175,000	10.41495	10.64786	11.10527	11.1216	11.25479	10.9089	11.0205	11.43711	11.23391	11.19731	11.18664	11.2150946	10.95093	11.00062	10.92956	10.91738	10.88143	10.93599
	300,000	31.60908	31.40414	31.48019	31.59794	31.54823	31.52792	32.69337	33.04962	33.19424	32.35979	32.084	32.676206	32.40179	31.92765	32.69167	32.61009	32.73086	32.47241
	425,000	64.80654	67.34658	68.40903	66.92625	66.0271	66.7031	66.03276	67.42167	65.35007	66.77202	64.6197	66.0392442	64.89213	63.72307	63.81994	64.92542	64.6993	64.41197
	550,000	106.8575	108.0881	108.838	112.7331	112.1013	109.7236	108.3689	112.7772	112.5805	111.0176	109.3391	110.816666	110.4812	109.1349	107.7226	113.5605	113.2699	110.8338
Insertion Sort	50,000	0.568815	0.557586	0.559178	0.559914	0.561526	0.561404	7.57E-05	0.000189	0.000075	6.63E-05	8.67E-05	0.00009848	1.087603	1.107942	1.122212	1.118818	1.150187	1.117352
	100,000	2.209086	2.209993	2.209888	2.213052	2.227334	2.213871	0.000146	0.000135	0.000135	0.000153	0.000168	0.00014724	4.262501	4.271028	4.278513	4.414866	4.433451	4.332072
	175,000	6.950642	7.034455	6.79763	6.745538	6.817427	6.869138	0.000252	0.000227	0.000224	0.000246	0.000387	0.00026708	13.04776	13.38227	13.37156	13.4117	13.55983	13.35463
	300,000	20.27747	20.17139	20.14759	19.97253	20.13685	20.14117	0.000712	0.000624	0.000386	0.000464	0.00043	0.00052304	40.86074	41.03057	41.04156	41.31135	41.46262	41.14137
	425,000	40.44298	40.68572	40.11529	40.11976	40.24385	40.32152	0.000659	0.000683	0.000605	0.000652	0.000609	0.00064142	81.62176	83.47128	82.81355	82.31039	81.75415	82.39422
	550,000	66.81084	66.04715	65.77408	65.87078	66.89417	66.2794	0.000726	0.000964	0.000775	0.000771	0.000768	0.00080062	141.3114	139.9031	132.3516	133.5469	132.6221	135.947
Merge Sort	50,000	0.033158	0.024993	0.027163	0.024785	0.026552	0.02733	0.03596	0.032437	0.031761	0.03248	0.029646	0.03245668	0.027336	0.018316	0.020668	0.018178	0.020549	0.021009
	100,000	0.047209	0.052738	0.04811	0.052482	0.047475	0.049603	0.046511	0.043685	0.045584	0.04267	0.04623	0.04493604	0.036843	0.042577	0.036691	0.041959	0.037886	0.039191
	175,000	0.088227	0.079888	0.087163	0.078219	0.087378	0.084175	0.073131	0.080478	0.074568	0.08006	0.074467	0.07654084	0.066186	0.072502	0.064274	0.072146	0.066807	0.068383
	300,000	0.152253	0.138677	0.152937	0.136423	0.151508	0.146359	0.135222	0.12839	0.139768	0.129445	0.142403	0.13504572	0.12178	0.110601	0.124193	0.110119	0.125045	0.118348
	425,000	0.213023	0.19027	0.212371	0.194729	0.218865	0.205852	0.188664	0.1979	0.185359	0.203035	0.189926	0.19297686	0.15835	0.173172	0.154579	0.178288	0.159791	0.164836
	550,000	0.253303	0.27935	0.257995	0.283158	0.25544	0.265849	0.257392	0.244818	0.257179	0.244081	0.256643	0.2520226	0.228493	0.201433	0.228153	0.203097	0.226011	0.217437
Quick Sort	50,000	0.00403	0.003636	0.003726	0.003648	0.003624	0.003733	0.001995	0.001711	0.001173	0.001118	0.000999	0.00139914	0.002543	0.001998	0.002019	0.001959	0.002106	0.002125
	100,000	0.009104	0.007306	0.007563	0.007477	0.008223	0.007935	0.002264	0.002415	0.00226	0.002162	0.002246	0.00226928	0.004313	0.004289	0.004071	0.004013	0.004198	0.004177
	175,000	0.014167	0.014419	0.014129	0.013562	0.013795	0.014015	0.004321	0.006248	0.004055	0.004589	0.004037	0.00464998	0.007575	0.009245	0.007787	0.007149	0.0079	0.007931
	300,000	0.025037	0.02515	0.025242	0.024895	0.025097	0.025084	0.006921	0.006125	0.006274	0.006534	0.006553	0.00648124	0.013653	0.014147	0.013438	0.013759	0.013546	0.013709
	425,000	0.036631	0.03693	0.036297	0.035845	0.036162	0.036373	0.010383	0.009587	0.010347	0.010363	0.009623	0.01006042	0.019734	0.020394	0.019473	0.01998	0.019842	0.019885
	550,000	0.04842	0.049242	0.04702	0.046913	0.047525	0.047824	0.012044	0.013042	0.012166	0.012561	0.012692	0.01250102	0.025601	0.028862	0.025559	0.026895	0.027349	0.026853
Shell Sort	50,000	0.008831	0.007539	0.007282	0.007001	0.007418	0.007614	0.001123	0.001151	0.001001	0.001133	0.001104	0.00110254	0.002027	0.00151	0.001498	0.002145	0.001794	0.001795
	100,000	0.016564	0.016214	0.015792	0.016096	0.016518	0.016237	0.002136	0.002919	0.002122	0.002099	0.002165	0.0022883	0.003282	0.003246	0.003295	0.003087	0.00341	0.003264
	175,000	0.029173	0.029468	0.02902	0.029363	0.030271	0.029459	0.003912	0.004228	0.003959	0.004077	0.004198	0.00407482	0.007436	0.005585	0.005692	0.005961	0.006613	0.006257
	300,000	0.056951	0.055904	0.055635	0.055238	0.057211	0.056188	0.006916	0.007572	0.007172	0.007264	0.007222	0.00722926	0.010884	0.010807	0.010021	0.011214	0.010969	0.010779
	425,000	0.078413	0.08013	0.082782	0.082231	0.080297	0.08077	0.01012	0.009875	0.009889	0.010355	0.010657	0.0101792	0.015784	0.015213	0.014569	0.01501	0.015429	0.015201
	550,000	0.109036	0.107967	0.108912	0.110074	0.108929	0.108984	0.013903	0.014744	0.013628	0.013381	0.013626	0.01385656	0.020063	0.020836	0.020421	0.020691	0.020745	0.020551

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.

	Runtime (s)			
n	Shell	Shell/$\log_2(n)$	Insertion	% Error
50,000	0.0011	0.000071	0.00010	28.28
100,000	0.0023	0.000138	0.00015	6.43
175,000	0.0041	0.000234	0.00027	12.40
300,000	0.0072	0.000397	0.00052	24.03
425,000	0.0102	0.000544	0.00064	15.12
550,000	0.0139	0.000727	0.00080	9.24

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

		Key: if x/y > 1 = *m* , then y is *m* times faster than x, otherwise, x is *1/m* times faster than y.					
		Numerator					
	Algorithm	**Bubble Sort**	**Selection Sort**	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Shell Sort**
Denominator	**Bubble Sort**	-	**Selection/Bubble**	**Insertion/Bubble**	**Merge/Bubble**	**Quick/Bubble**	**Shell/Bubble**
		-	0.23	0.15	0.00709	0.00097	0.00198
			0.21	0.14	0.00303	0.00048	0.00099
			0.22	0.14	0.00170	0.00028	0.00060
			0.21	0.13	0.00097	0.00017	0.00037
			0.22	0.13	0.00067	0.00012	0.00026
			0.22	0.13	0.00054	0.00010	0.00022
	Selection Sort	**Bubble/Selection**	-	**Insertion/Selection**	**Merge/Selection**	**Quick/Selection**	**Shell/Selection**
		4.43	-	0.64	0.0314	0.0043	0.0087
		4.73		0.64	0.0143	0.0023	0.0047
		4.53		0.63	0.0077	0.0013	0.0027
		4.78		0.64	0.0046	0.0008	0.0018
		4.61		0.60	0.0031	0.0005	0.0012
		4.51		0.60	0.0024	0.0004	0.0010
	Insertion Sort	**Bubble/Insertion**	**Selection/Insertion**	-	**Merge/Insertion**	**Quick/Insertion**	**Shell/Insertion**
		6.87	1.55	-	0.0487	0.0066	0.0136
		7.41	1.57		0.0224	0.0036	0.0073
		7.20	1.59		0.0123	0.0020	0.0043
		7.49	1.57		0.0073	0.0012	0.0028
		7.63	1.65		0.0051	0.0009	0.0020
		7.46	1.66		0.0040	0.0007	0.0016
	Merge Sort	**Bubble/Merge**	**Selection/Merge**	**Insertion/Merge**	-	**Quick/Merge**	**Shell/Merge**
		141.03	31.86	20.54	-	0.14	0.28
		330.55	69.86	44.63		0.16	0.33
		587.40	129.60	81.61		0.17	0.35
		1030.62	215.41	137.61		0.17	0.38
		1494.64	324.03	195.88		0.18	0.39
		1860.74	412.73	249.31		0.18	0.41
	Quick Sort	**Bubble/Quick**	**Selection/Quick**	**Insertion/Quick**	**Merge/Quick**	-	**Shell/Quick**
		1032.59	233.26	150.40	7.32	-	2.04
		2066.41	436.76	279.02	6.25		2.05
		3528.06	778.40	490.14	6.01		2.10
		6013.42	1256.89	802.95	5.83		2.24
		8458.85	1833.86	1108.56	5.66		2.22
		10343.67	2294.32	1385.90	5.56		2.28
	Shell Sort	**Bubble/Shell**	**Selection/Shell**	**Insertion/Shell**	**Merge/Shell**	**Quick/Shell**	-
		506.24	114.36	73.73	3.59	0.49	-
		1009.80	213.43	136.35	3.05	0.49	
		1678.41	370.31	233.18	2.86	0.48	
		2684.58	561.12	358.46	2.60	0.45	
		3809.25	825.84	499.21	2.55	0.45	
		4538.99	1006.79	608.16	2.44	0.44	

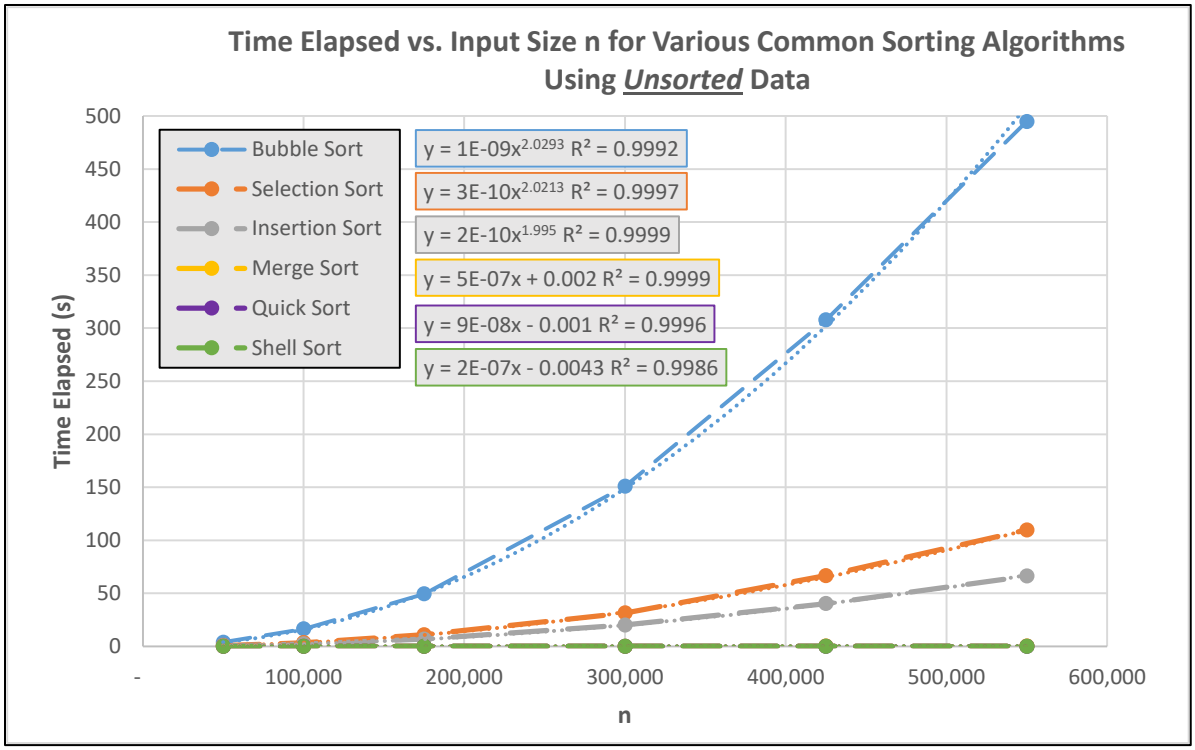

Figure 11. Comparison of runtime durations for randomly sorted data.

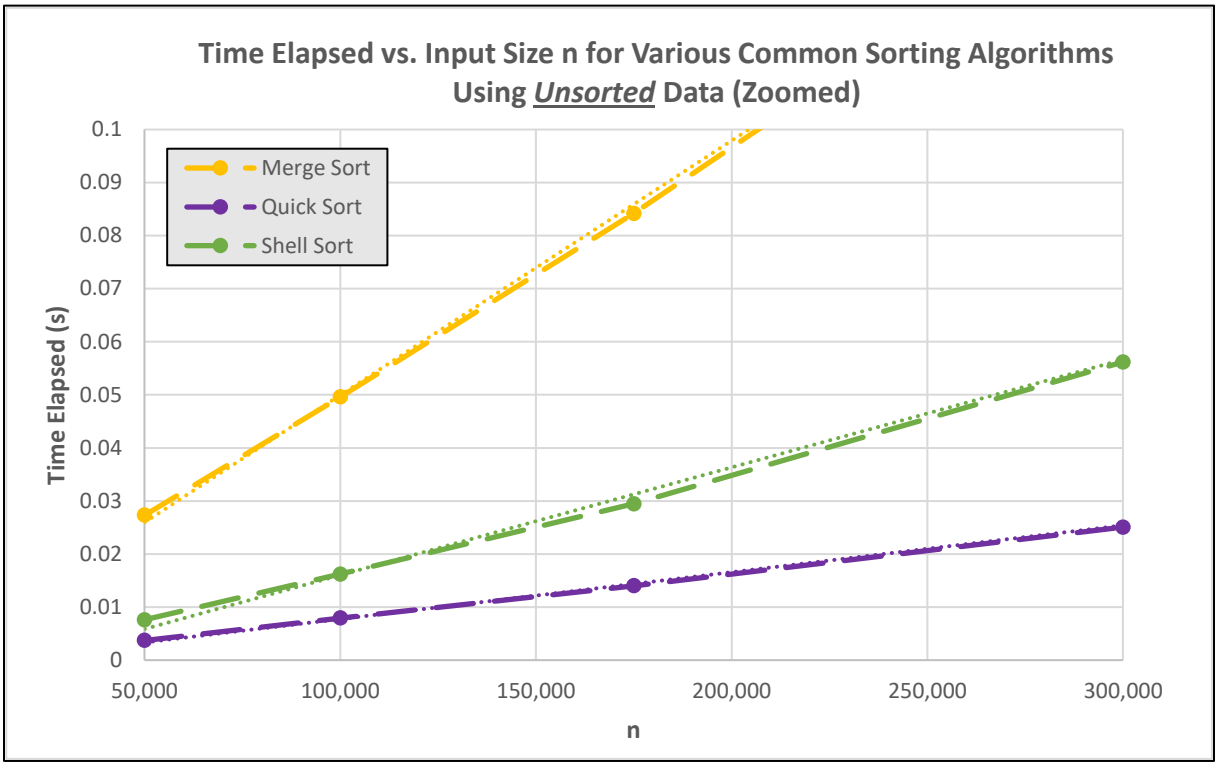

Figure 12. Comparison of runtime durations for randomly sorted data (zoomed).

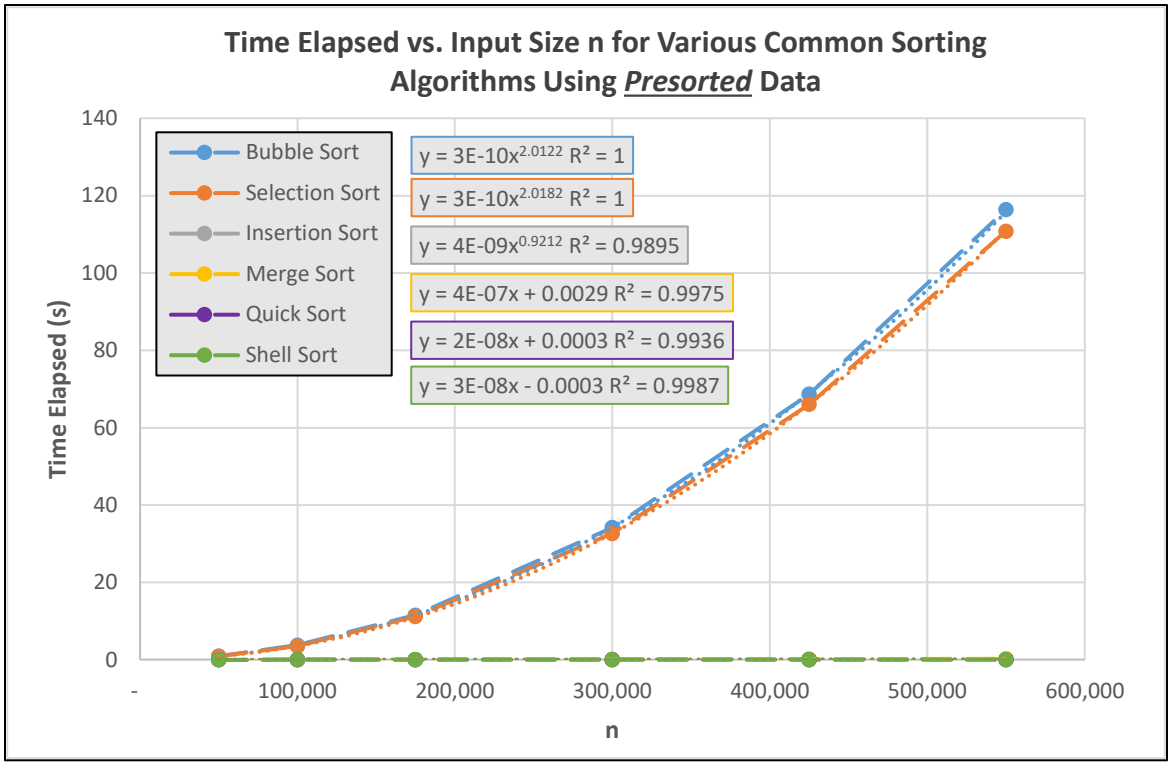

Figure 13. Comparison of runtime durations for increasing order sorted data.

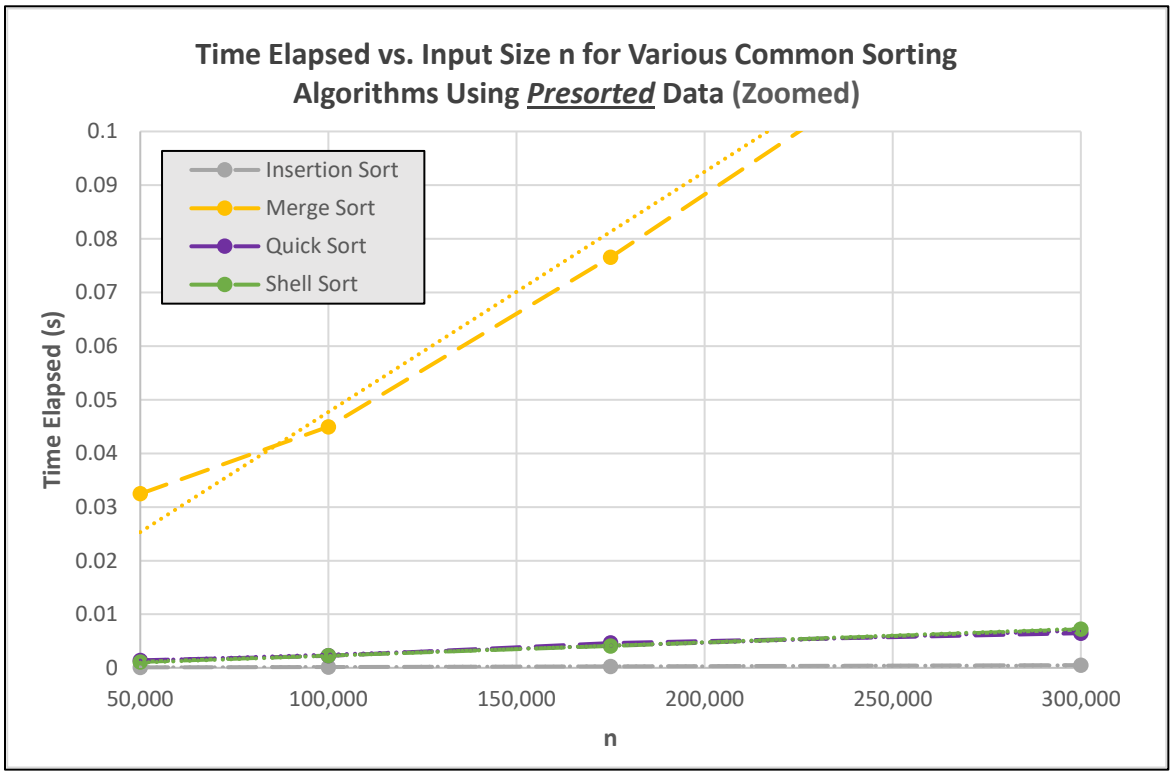

Figure 14. Comparison of runtime durations for increasing order sorted data (zoomed).

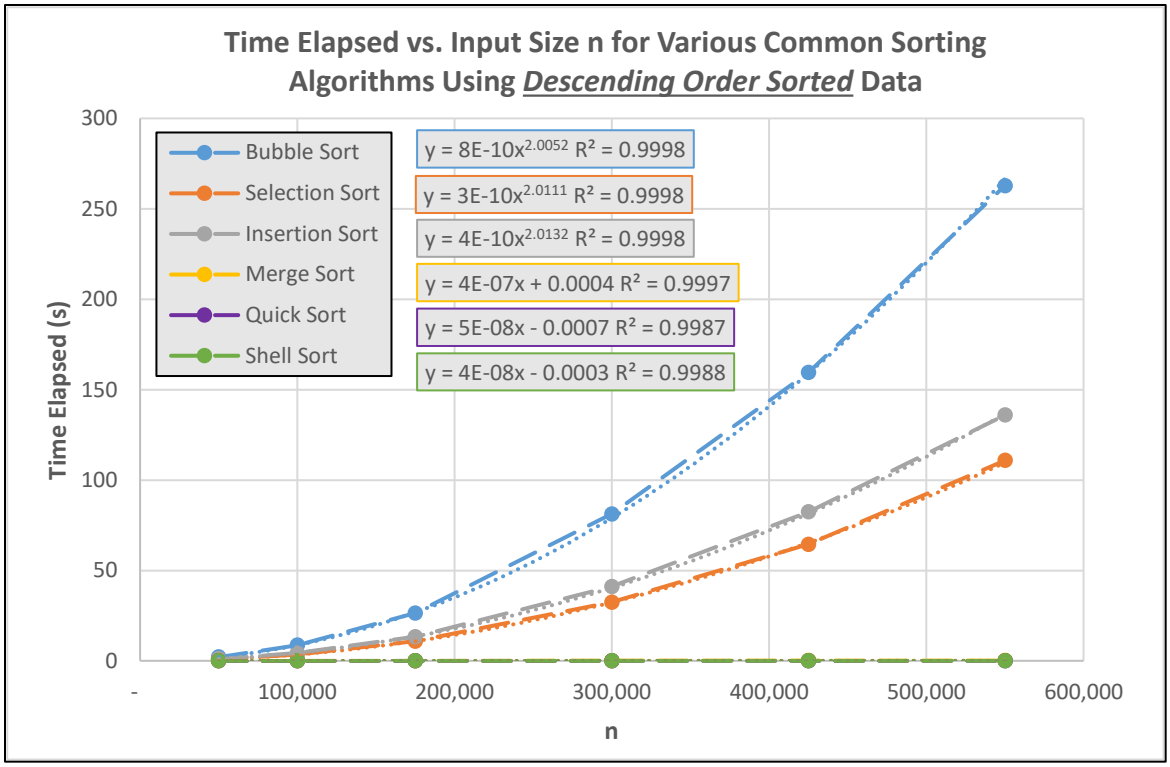

Figure 15. Comparison of runtime durations for descending order sorted data.

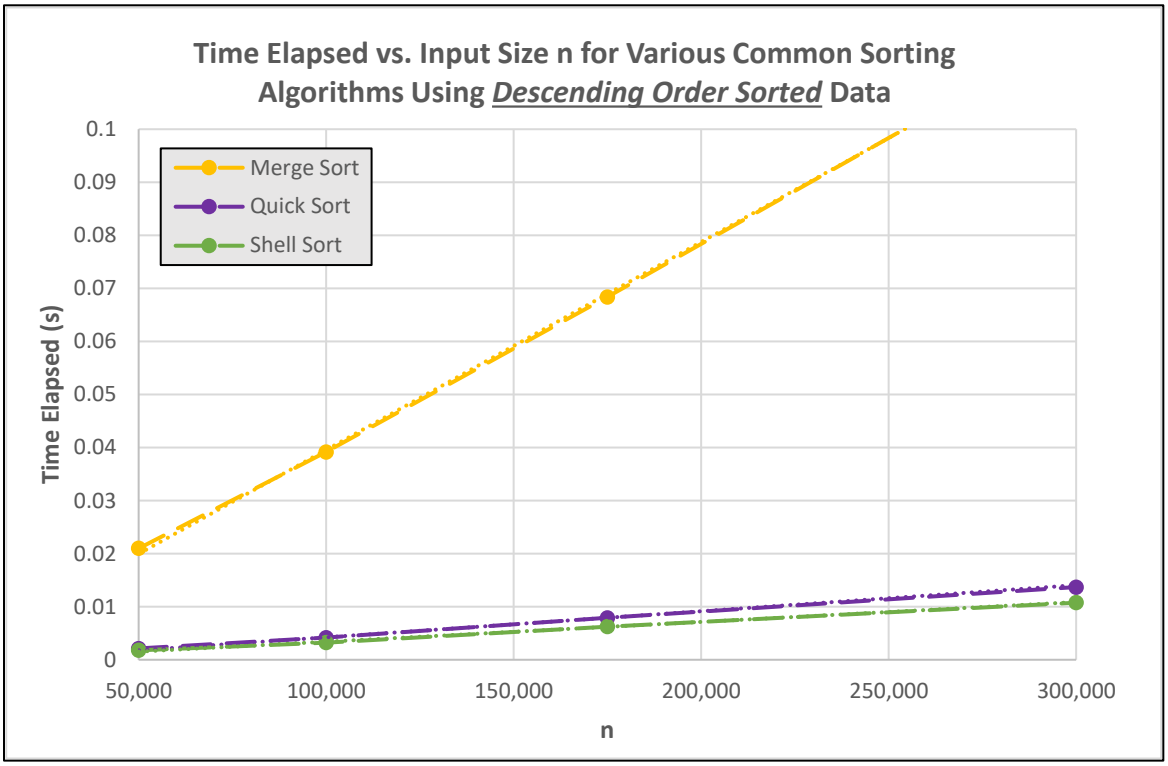

Figure 16. Comparison of runtime durations for descending order sorted data (zoomed).

Table 5. Linearithmic runtime approximation for Merge Sort, Quick Sort, and Shell Sort on presorted data, obtained using Excel's Solver. While the best-fit lines portrayed in figures 11-16 show linear trends for these algorithms, they have linearithmic bounds in theory.

	$T(n) \approx C_{op}*C(n)$; $C(n) = n\log_2 n$		
	Merge Sort	**Quick Sort**	**Shell Sort**
C_{op} (s)	2.43E-08	1.23E-09	1.31E-09
SSR	2.23E-04	1.35E-06	9.83E-08

Item	Value
OS Name	Microsoft Windows 11 Home
Version	10.0.22631 Build 22631
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	KHARP
System Manufacturer	LENOVO
System Model	82WK
System Type	x64-based PC
System SKU	LENOVO_MT_82WK_BU_idea_FM_Legion Pro 5 16IRX8
Processor	13th Gen Intel(R) Core(TM) i7-13700HX, 2100 Mhz, 16 Core(s), 24 Logical Processor(s)
BIOS Version/Date	LENOVO KWCN46WW, 7/4/2024
SMBIOS Version	3.4
Embedded Controller Version	1.46
BIOS Mode	UEFI
BaseBoard Manufacturer	LENOVO
BaseBoard Product	LNVNB161216
BaseBoard Version	SDK0T76461 WIN
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\Windows
System Directory	C:\Windows\system32
Boot Device	\Device\HarddiskVolume1
Locale	United States
Hardware Abstraction Layer	Version = "10.0.22621.2506"
User Name	kharp\kharp
Time Zone	Eastern Daylight Time
Installed Physical Memory (RAM)	16.0 GB

Figure 17. System information for hardware used to analyze algorithm runtime.

Discussion & Analysis

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 (approximately) 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. In general, Bubble Sort, Selection Sort, and Insertion Sort are found to exhibit quadratic time complexities ($O(n^2)$).

Regarding the presorted data (see figures 13 and 14), 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 exhibits quadratic time complexity. This is because Insertion Sort requires fewer insertions on nearly sorted data, and insertions require O(n) time. As mentioned, this is the reason why Shell Sort achieves better 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 preference.

When the data is sorted in descending order (see figures 15 and 16), 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, Merge Sort, and Shell Sort are found to perform better when the data is sorted in reverse order, despite requiring a greater number of swaps for array elements. At the machine level, factors like cache utilization, branch prediction, pipelining, and the mechanics of comparisons and swaps come into play. These low-level considerations might explain why algorithms can operate more efficiently on reverse-sorted data than randomly sorted data despite the apparent increase in swaps.

For input data in various forms, Merge Sort, Quick Sort, and Shell Sort performance is relatively consistent. This indicates that the worst-case, best-case, and average-case time complexities for these sorting algorithms have a greater dependence on the implementation details rather than the form of the input data.

Considering unordered input data, Selection Sort performs roughly 5 times as fast as Bubble Sort for all dataset sizes (see table 4). Quick Sort performs about 2 times faster than Shell Sort. As dataset size increases, Quick Sort approaches a performance roughly 5 times that of Merge Sort, and 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 a 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, Merge Sort is roughly 140 times faster than Bubble Sort for this dataset size.

Algorithms with O(nlogn) complexity (like Merge Sort and Quick Sort) are generally preferred for larger datasets, while $O(n^2)$ algorithms (like Bubble, Selection, and Insertion Sort) can be useful for smaller datasets or specific cases where their properties provide benefits. In general, the divide-and-conquer algorithms perform faster than some other approaches due to several key

characteristics inherent in their design. Divide-and-conquer algorithms break a large problem into smaller, more manageable sub-problems. Each sub-problem is of the same type as the original, allowing the same algorithm to be applied recursively. Since sub-problems are often independent, they can be processed in parallel on multi-core processors. This parallel execution can significantly reduce overall computation time. By working on smaller chunks of data, divide-and-conquer algorithms can reduce the number of operations needed to find solutions.

Conclusion

Quick Sort and Shell Sort are found to be the top performing algorithms. However, as input data size becomes smaller, the difference between runtimes becomes negligible. When choosing an algorithm per specific application, it is important to consider implementation complexity, time and space efficiencies, the form of input data, expected input size, and whether stability is necessary. For example, a divide-and-conquer algorithm might not be an appropriate choice in a resource constrained environment due to the memory overhead they require.

Possible future enhancements to the program include a graphical user-interface, general optimizations for application specific features, animation for better visualization, and inclusion of additional algorithms, like Heap Sort and non-comparative approaches. Furthermore, it might be instructive to include alternative implementations of the algorithms, such as a different pivot selection scheme for Quick Sort, or an alternative gap sequence for Shell Sort.

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] “9: Sorting Algorithms.” *Engineering LibreTexts*, Libretexts, 10 Mar. 2021.

[8] “Merge Sort Algorithm: Working and Example of Merge Sort Algorithm.” *EDUCBA*, 5 Apr. 2023.

[9] Singh, Shlok. “Different Pivot Selection in Quick Sort.” *OpenGenus IQ: Learn Algorithms, DL, System Design*, OpenGenus IQ: Learn Algorithms, DL, System Design, 4 May 2022.

[10] “Shell Sort Algorithm Implementation in C: Knuth’s and Hibbard’s Intervals - a High-Speed Sorting Procedure.” *MYCPLUS*, 21 Feb. 2024.

[11] Harper, Kevin. “CSCI2226 - Sorting Algorithm Project Demo.” YouTube, 6 Oct. 2024, www.youtube.com/watch?v=jgh3wizjoh4.