Program 4
Sorter

B CSS 342: Winter 2024

by

Hoang Long Bui **- Student ID 2373329**

Program Due Date: _February 29, 2024_

1. Algorithm Analysis

1.1 Objective

This program provides hands on experience with various types of sorting algorithms introduced in class. Secondly, the aim of this program is to explicitly demonstrate the cost of $O(n^2)$ and $O(n * log_n)$. Depending on the dataset, the behavior of the specific algorithm varies.

1.2 Discussion of Measurements

Figure 1 demonstrates the runtime of each sorting algorithm with specified set of elements (varies from a set of 10 elements to a set of 1 million elements).

	Bubble Sort	**Insertion Sort**	**Quick Sort**	**Shell Sort**	**Recursive Merge Sort**	**Iterative Merge Sort**
# elements	**Time in milliseconds (ms)**					
10	3	3	2	3	3	3
100	58	25	12	19	21	13
1000	5862	1575	103	715	763	284
2500	37454	9541	289	2126	731	603
5000	130430	43457	2283	1713	1382	992
10,000	538633	152709	1339	3058	5297	2867
25,000	3459900	961412	3677	9999	6810	6018
50,000	13892620	3852531	7740	20274	13913	12705
100,000	56010596	15399168	16591	46107	29717	26960
250,000	35066750	96531229	45180	128304	79298	73098
500,000	1403358554	387459072	95661	286785	167960	153626

Figure 1 – Efficiency of sorting algorithms

Based on observations, it is noticeable that with small dataset (10 to 100 elements), the performance of listed algorithms is consistent with runtime. However, $Bubble\ Sort$ takes slightly more time compared to others at 100 elements (but not significantly).

As the dataset increases in elements, the difference in runtime becomes more pronounced. $Bubble\ Sort$ and $Insertion\ Sort$ exhibit poor scalability with their associated runtimes increases tremendously as the elements grow.

$Quick\ Sort, Shell\ Sort, Recursive\ Merge\ Sort$ and $Iterative\ Merge\ Sort$ demonstrate a better scalability compared to $Bubble\ Sort$ and $Insertion\ Sort$, especially evident in larger datasets (10,000 elements and above). These algorithms show significantly lower runtimes even for larger datasets.

To provide a better visualization of runtime, we will plot a graph to demonstrate the behavior of Figure 1.

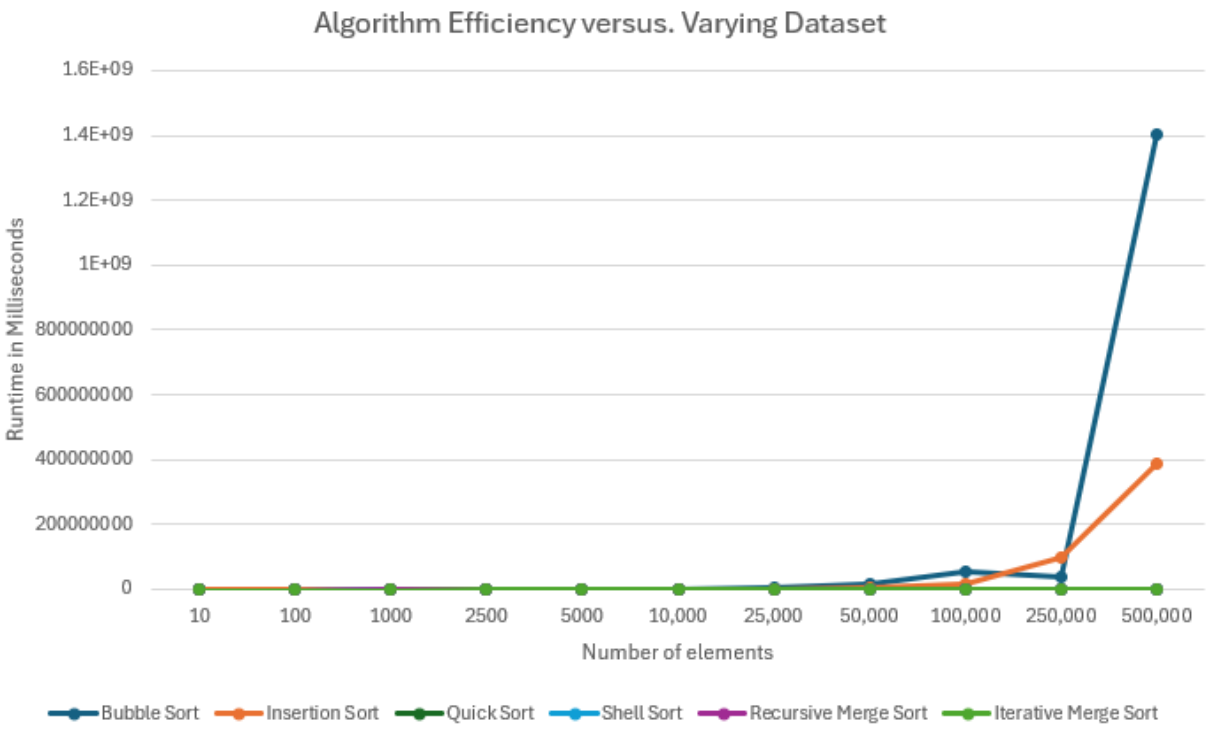

Figure 2 – Efficiency of algorithms versus datasets

Figure 2 visualizes the runtime behavior of varying sorting algorithms versus datasets. The growth of *Bubble Sort* is too large that leads to the significant peak in the y – axis. Observing the graph, we can clearly see the "poor" performance of *Bubble Sort* and *Insertion Sort* as the datasets grow over time. The others' performances are not so called "steady", but indeed they perform in a good "stable" state.

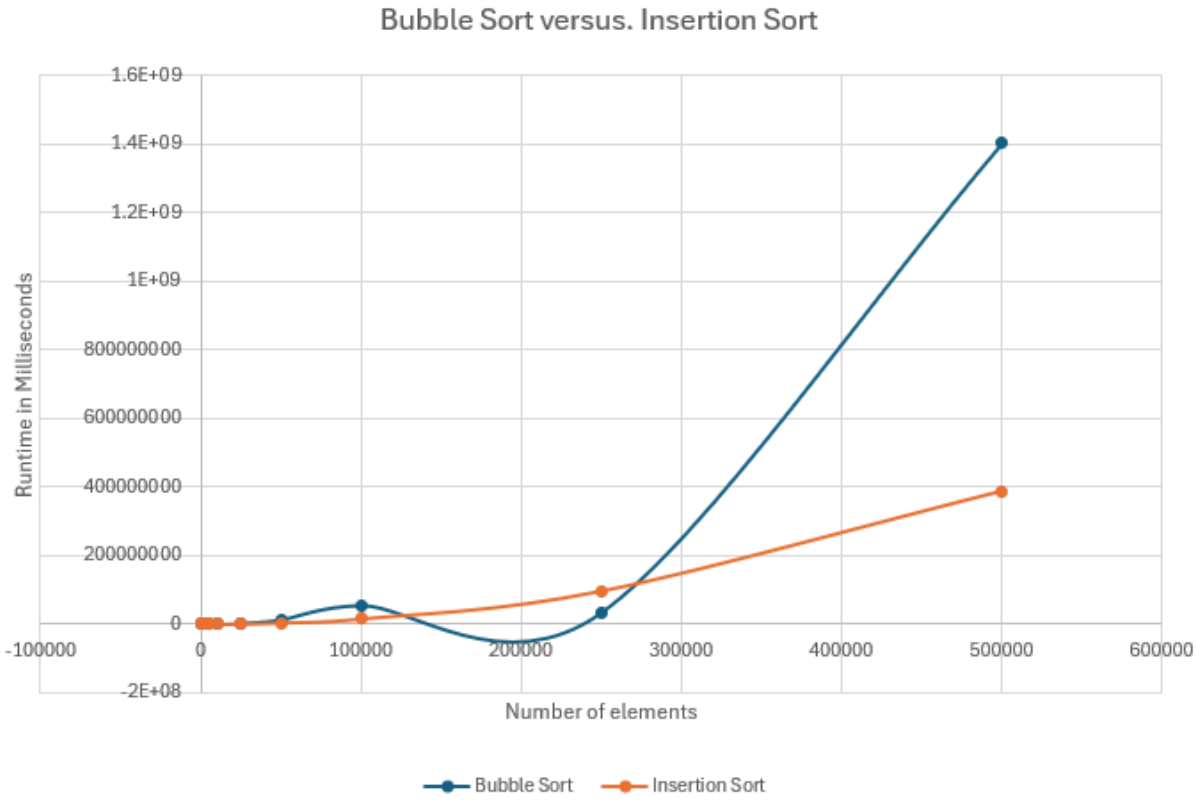

Figure 3 – Efficiency Bubble Sort and Insertion Sort versus datasets

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares the adjacent elements, and swaps them if they do not satisfy the conditions. The process is heavily repeated until the list is sorted. As shown in the dataset above, the runtime of *Bubble Sort* increases dramatically as the number of elements in the dataset increases. The reason accommodates for this behavior is that it must run through two $for\ loops$, leading to a time complexity of $O(n^2)$ for average cases. *Bubble Sort* is often considered impractical for real world applications due to its poor runtime. However, it is a "memorable" algorithm, which everyone can implement, if the dataset contains less than 100 ~ 200 elements.

Insertion Sort is another simple sorting algorithm that builds the final sorted array (or list) one item at a time. It iterates through the input elements, removing one element from the input data, finding the location it belongs to in the sorted list, and inserting it there. Based on the graph, while the *Insertion Sort* is more efficient than *Bubble Sort* in most of the datasets. However, for a partially sorted datasets, it might suffer a "poorer" performance compared to *Bubble Sort*. For instance, at the dataset of 250,000 elements, the runtime of *Insertion Sort* bumps higher, which might be the case that the generated dataset is partially sorted. Other than that, the behavior of the two sorting algorithms meets the expectations.

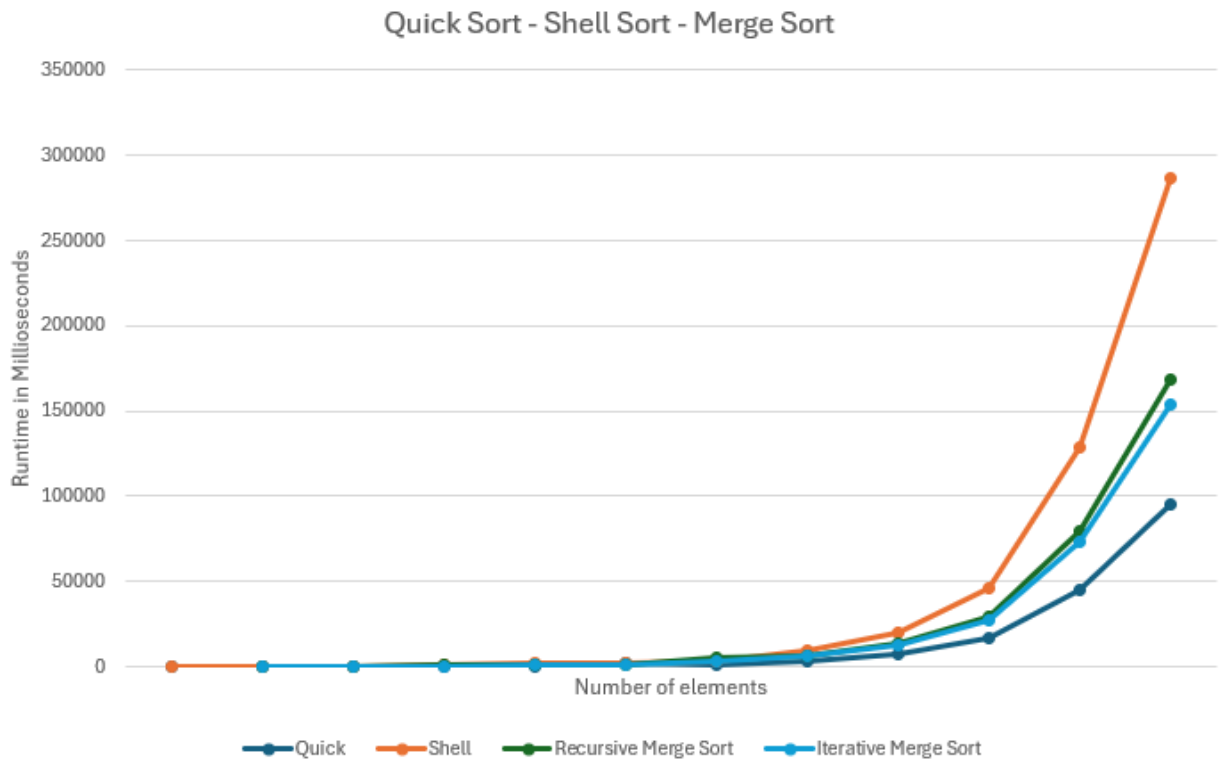

Figure 4 – Efficiency Quick Sort, Shell Short and Merge Sort

The four sorting algorithms demonstrated an appropriate $O(n * log_n)$ behavior. By investigating the relationship between the number of elements and runtime. It clearly shows that $Shell\ Sort$ runs the slowest, then $Recursive\ Merge\ Sort, Iterative\ Merge\ Sort$ and $Quick\ Sort$. To answers for this, $Shell\ Sort$ requires moving the elements around multiple times as the sub-array decreases, the data moving costs more resources, especially the gap that we pick plays a major role in the performance of $Shell\ Sort$ (can lead to $O\ (n^2)$ if the gap is not wisely considered). However, for a partially sorted dataset, $Shell\ Sort$ can benefit from the existed order of elements and outperform the other algorithms.

Throughout 11 test points, $Quick\ Sort$ is leading in the fastest sorting algorithms. Even on small or large dataset, the result is consistent with time. One step prior to partitioning, I used the technique "Median of three" to efficiently pick the "right" pivot point. As it is the soul of $Quick\ Sort$, just like the right gap in $Shell\ Sort$. A bad pivot point can lead to a $O(n^2)$ time complexity, we need to think about the correlation between the dataset and its "story" to pick the right pivot.

The discussion of $Recursive\ Merge\ Sort$ and $Iterative\ Merge\ Sort$ is discussed in the next part.

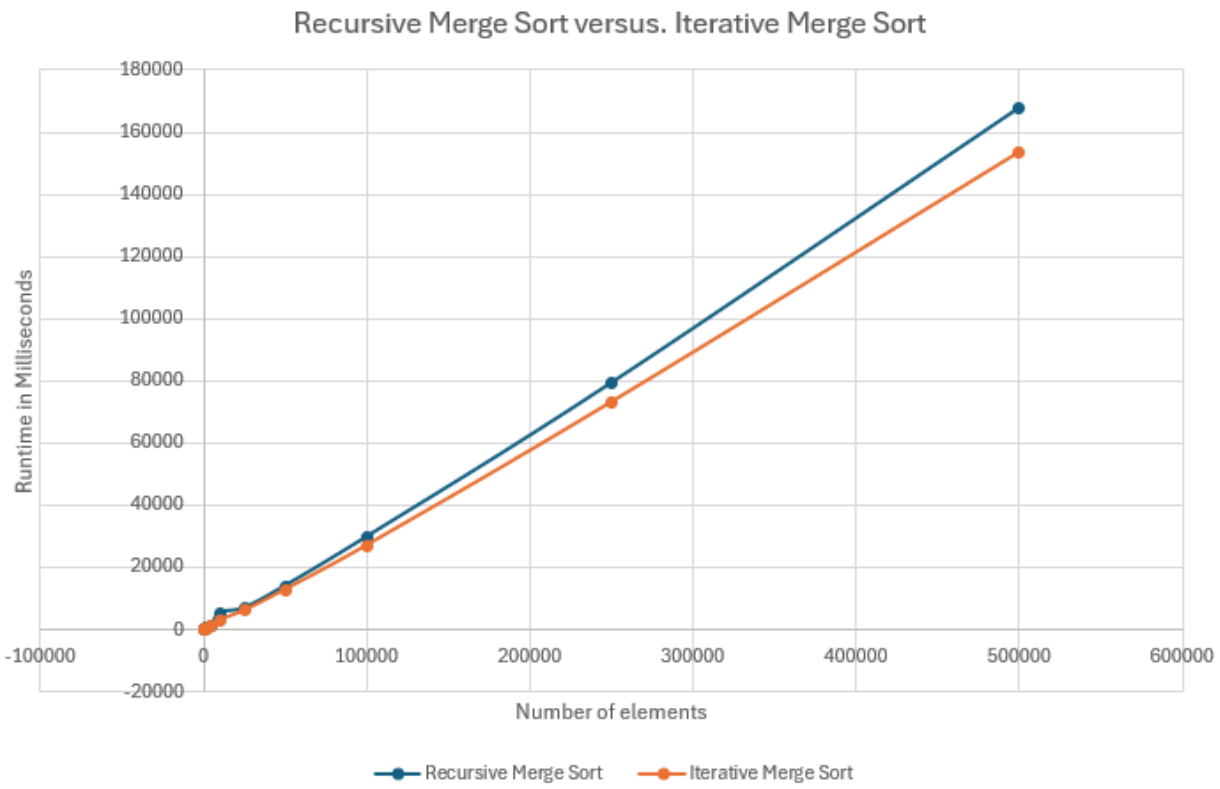

Figure 5 – Efficiency of Recursive Merge Sort and Iterative Merge Sort

Merge Sort can be successfully achieved by Bottom-Up or Top-Down approaches (so called Iterative and Recursive, respectively). In the Recursive approach, it continuously splits the dataset into halves until it cannot be further divided (base case: an array with one number, and it is always sorted). Then the sorted sub-array is merged into one sorted array. By mentioning recursion, *Recursive Merge Sort* allocates a temporary vector at each recursive call and that can be very expensive when it comes to large dataset. On the other hand, *Iterative Merge Sort* only requires one extra auxiliary vector as a placeholder. We alternatively copy back and forth between the original and the auxiliary vector, this approach does not build up calls on the stack. The result speaks for itself, the runtime of *Iterative Merge Sort* drops significantly compared to *Recursive Merge Sort*.

Figures 6 shows the runtime complexity of discussed sorting algorithms.

	Average	Best	Worst
Bubble Sort	$O\ (n^2)$	$O\ (n^2)$	$O\ (n^2)$
Insertion Sort	$O\ (n^2)$	$O(n)$	$O\ (n^2)$
Recursive Merge Sort	$O(n * log_n)$	$O(n * log_n)$	$O(n * log_n)$
Iterative Merge Sort	$O(n * log_n)$	$O(n)$	$O(n * log_n)$
Quick Sort	$O(n * log_n)$	$O(n * log_n)$	$O\ (n^2)$
Shell Sort	$O(n * log_n)$	$O(n)$	$O(n^2)$

Figure 6 – Time Complexity for different sorting algorithms

1.3 Summary and Conclusions

In this program, we implemented and analyzed the performance of several common sorting algorithms including $Bubble\ Sort,\ Insertion\ Sort,\ Quick\ Sort,\ Shell\ Sort,$ $Recursive\ Merge\ Sort\ and\ Iterative\ Merge\ Sort$. The algorithms were tested on datasets ranging from 10 to 500,000 elements.

The results clearly demonstrated the $O(n^2)$ runtime complexity of simple sorting approaches like $Bubble\ Sort, Insertion\ Sort$. As the input size increased, the runtime of these algorithms increased dramatically, making them impractical for large datasets.

In contrast, more advanced algorithms: $Quick\ Sort, Shell\ Sort, Recursive\ Merge\ Sort\ and\ Iterative\ Merge\ Sort$ demonstrated $O(n * log_n)$ complexity. Their running times increased much more slowly with input size due to their divide-and-conquer approaches. $Quick\ Sort$ was the fastest overall, while $Iterative\ Merge\ Sort$ had better performance than $Recursive\ Merge\ Sort$ due to reduced memory overhead.

$Shell\ Sort$ performance was hampered by its reliance on sequential data movement. With proper choices for the gap, it can match the other advanced algorithms.

In summary, $O(n^2)$ sorts are only practical for tiny datasets, while $O(n * log_n)$ algorithms like $Quick\ Sort$ should be preferred for most applications. The program provided hands-on experience with algorithm analysis and reinforced theoretical concepts of time complexity and scalability in programming.