HiLCoE School of Computer Science and Technology

Batch: DRBSE2103

Data Structure and Algorithm Analysis (CS321)

Group Assignment I (Algorithm Profiler for Simple Sorting Algorithms)

Name	**ID Number**
Assier Anteneh	WQ0129
Yostena Tadesse	OL9245
Kunuz Mohammed	GX3230
Yabsira Kassaye	YC3901

Submission Date: 14th of August 2023
Submitted To: Tariku Worku

Table of Contents

1. INTRODUCTION I

1.1. ALGORITHM ANALYSIS I
1.2. SORTING I
1.3. ALGORITHM PROFILER II

2. BUBBLE SORT IV

2.1. BEST CASE V
2.2. WORST CASE VI
2.3. AVERAGE CASE VII

3. SELECTION SORT VIII

3.1. BEST CASE VIII
3.2. WORST CASE X
3.3. AVERAGE CASE XI

4. INSERTION SORT XII

4.1. BEST CASE XIII
4.2. WORST CASE XIV
4.3. AVERAGE CASE XV

5. CONCLUSION XVI

1. Introduction

This assignment is basically about building an algorithm profiler for the 3 simple sorting algorithms: Bubble Sort, Selection Sort and Insertion Sort. Before getting into the details of the assignment first let’s start by explaining the simple terms like what algorithm analysis is, what sorting is, what an algorithm profiler is.

1.1. Algorithm Analysis

Algorithm analysis is the process of studying and evaluating the efficiency and performance characteristics of algorithms. It involves assessing how an algorithm behaves in terms of time and space complexity as the size of the input data increases. The primary goal of algorithm analysis is to understand how well an algorithm can solve a problem and to compare different algorithms to determine which one is more suitable for a particular task.

Complexity refers to the analysis of the resources (such as time and space) required by an algorithm to solve a problem as a function of the size of the input data. The primary resources considered in algorithm complexity analysis are:

1. **Time Complexity**: Time complexity measures the amount of time or the number of basic operations an algorithm takes to execute as the size of the input data increases. It helps in understanding how the running time of an algorithm grows with the input size. Time complexity is often expressed using Big O notation (O()), which provides an upper bound on the growth rate of the algorithm's execution time.
2. **Space Complexity**: Space complexity measures the amount of memory or space an algorithm requires to execute as the size of the input data increases. It helps in understanding how the memory usage of an algorithm grows with the input size. Space complexity is also expressed using Big O notation.

1.2. Sorting

Sorting in the context of algorithm analysis refers to the process of arranging elements in a specific order, typically in ascending or descending order. Sorting algorithms are fundamental algorithms in computer science and play a crucial role in various applications, including data processing, database management, searching, and more.

The performance of a sorting algorithm is evaluated based on its time complexity and space complexity, as explained earlier.
Sorting algorithms are usually categorized based on their time complexity.

- Best Case: The time complexity when the input data is already sorted or nearly sorted.
- Average Case: The time complexity when the input data is randomly distributed.
- Worst Case: The time complexity when the input data is in reverse order or in a specific pattern that maximizes the algorithm's work.

In this assignment we'll only focus on simple sorting algorithms. Simple sorting algorithms are basic sorting techniques that are relatively easy to understand and implement. While they may not be the most efficient algorithms for large datasets, they serve as educational tools and building blocks for more advanced sorting algorithms. Here are some examples of simple sorting algorithms which will all be explained later on in this documentation:

- Bubble Sort
- Selection Sort
- Insertion Sort

These simple sorting algorithms are useful for educational purposes and small datasets. However, for larger datasets or performance-critical applications, more efficient sorting algorithms like Merge Sort, Quick Sort, and Heap Sort are typically preferred due to their better average-case time complexities (O(n log n)) and better practical performance.

1.3. Algorithm Profiler

An algorithm profiler, also known as a performance profiler or code profiler, is a software tool used by developers and software engineers to measure and analyze the performance of an algorithm or a piece of code. Its primary purpose is to identify performance bottlenecks and areas where the code can be optimized to run faster and use fewer resources.

Here's how an algorithm profiler typically works:

1. **Data Collection**: The profiler monitors the execution of the code and collects data on various aspects of its performance, such as the time taken to execute each function or method, the number of times each function is called, and memory usage.
2. **Profiling Modes**: Profilers can operate in different modes, such as CPU profiling (measuring CPU usage), memory profiling (identifying memory leaks or excessive memory usage), and other specific performance metrics.
3. **Visualization**: The collected data is often presented in a graphical or tabular format, making it easier for developers to analyze and interpret the performance characteristics of their code.
4. **Identifying Hotspots**: The profiler highlights the "hotspots" in the code, which are the parts of the code that consume the most time or resources. These hotspots are the areas that need the most attention for optimization.
5. **Optimization Insights**: Based on the data collected, the profiler can provide insights into areas that can be optimized, such as optimizing loops, reducing unnecessary function calls, or using more efficient data structures.
6. **Real-World Performance**: Profilers give a better understanding of how the code performs in real-world scenarios, which may be different from what developers assume based on the theoretical complexity of their algorithms.
7. **Debugging**: In addition to performance analysis, profilers can also help identify bugs or unexpected behavior in the code by providing detailed information about its execution.

Using an algorithm profiler is a crucial step in the software development process, especially when dealing with complex algorithms or performance-critical applications. It allows developers to identify and address performance issues, ultimately leading to faster and more efficient software. There are various profiling tools available for different programming languages and platforms, each with its own set of features and capabilities.

2. Bubble Sort

Bubble Sort is a simple and elementary sorting algorithm used to arrange elements in a list or array in ascending or descending order. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process gets its name from the way smaller elements "bubble" to the top of the list while larger elements "sink" to the bottom. i.e. after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For these reason Bubble Sort is considered to be a stable and an in-place sorting algorithm. And to complete the sorting If we have total n elements, then we need to repeat this process for n-1 times.

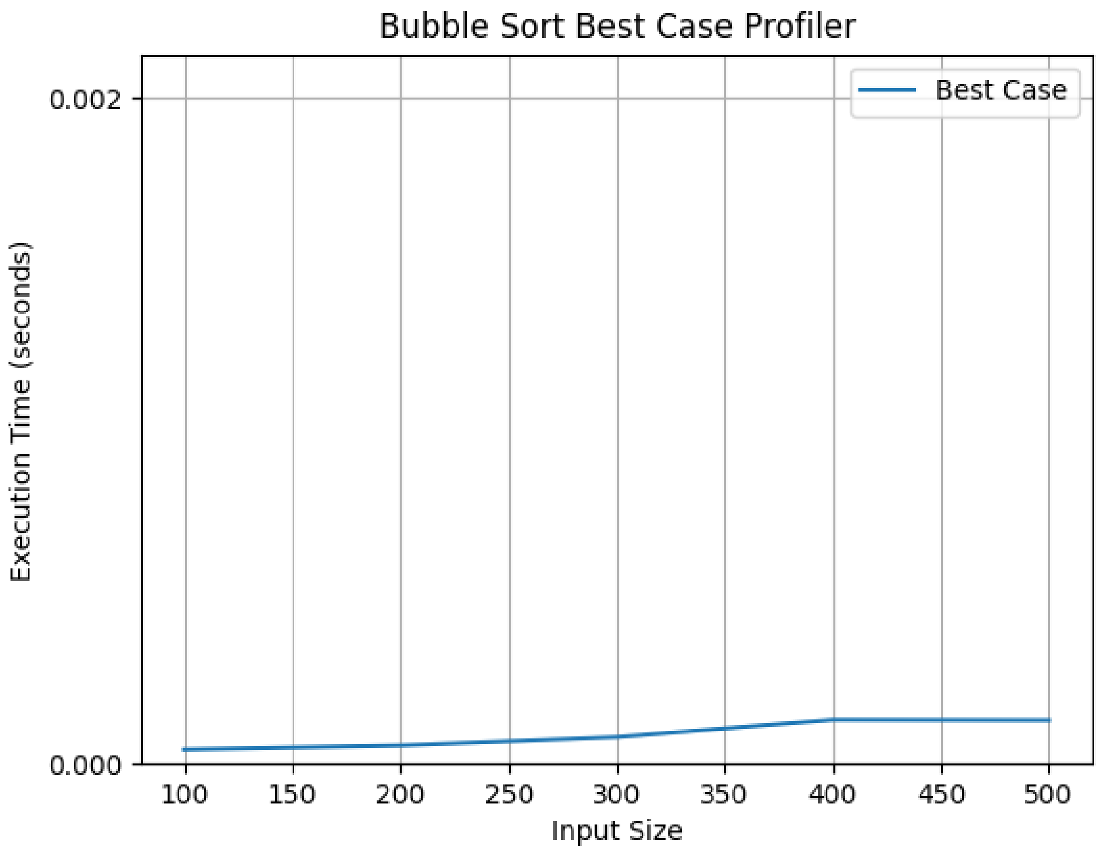
The algorithm works as follows:

1. Start at the beginning of the list (index 0).
2. Compare the first two elements. If the element on the left is greater than the element on the right (in ascending order), swap them.
3. Move to the next pair of elements (index 1 and index 2) and repeat the comparison and swapping process if needed.
4. Continue this process until you reach the end of the list. The largest element will now be at the last position.
5. Repeat steps 1-4, but this time ignore the last sorted element, as it is already in its correct position.
6. Keep repeating steps 1-5 until the entire list is sorted.

2.1. **Best Case**

The best case scenario for bubble sort occurs when the input list is already sorted. In this situation the algorithm will still go through all the iterations but will not perform any swaps because the elements are already in their correct order. As a result, the algorithm terminates after a single pass without making any swaps.

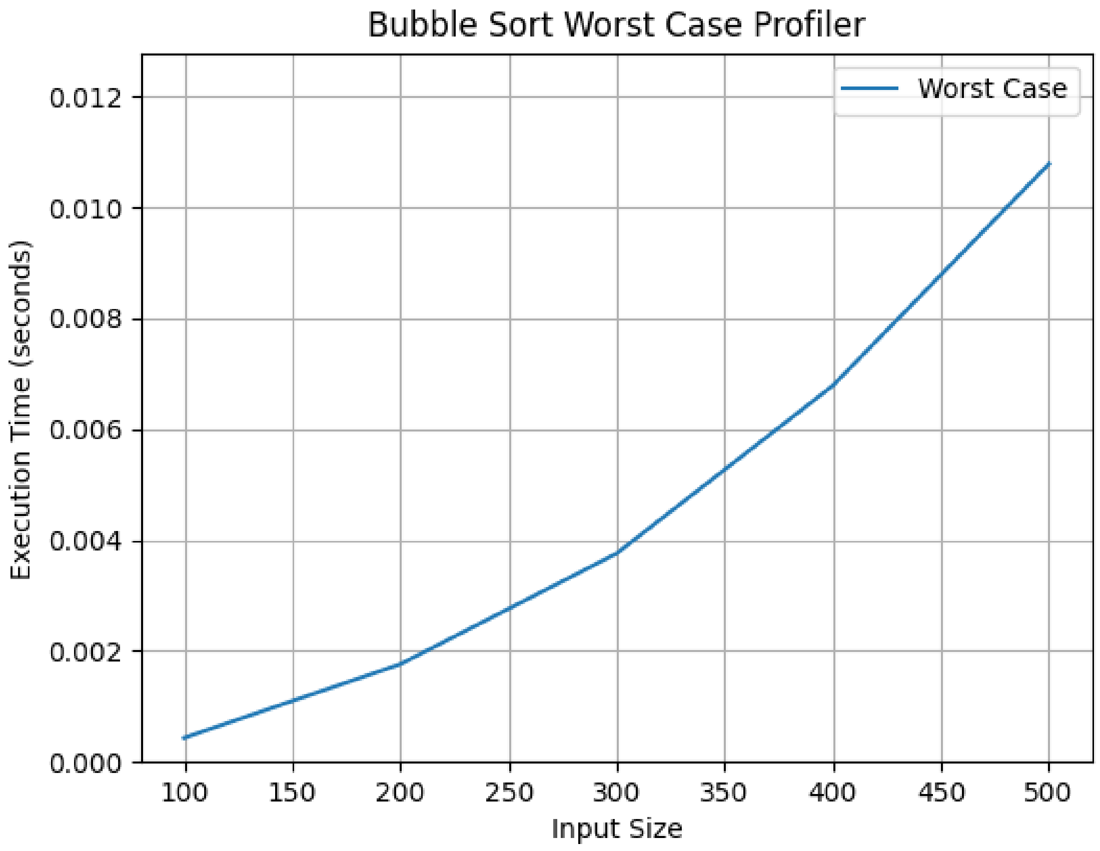
In the best-case scenario, the time complexity of bubble sort is O(n) because it requires only one pass through the list to determine that it is sorted. Here, n represents the number of elements in the list.



2.2. Worst Case

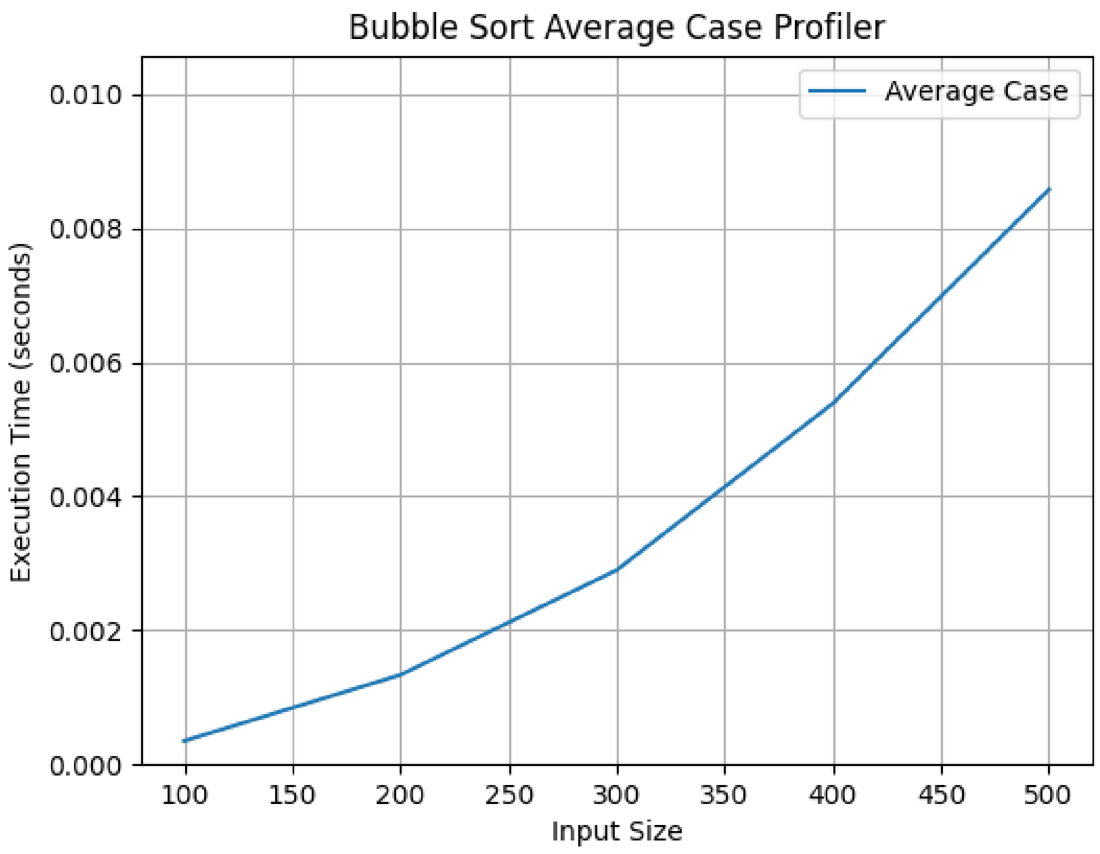
The worst case scenario for bubble sort occurs when the input list is in reverse order or sorted in the opposite direction to the desired order. In this situation the algorithm has to compare and swap every adjacent element in each pass and it will require the maximum number of iterations to sort the list.

In the worst-case scenario bubble sort has a time complexity of O(n^2), where n is the number of elements in the list this means that the time taken to sort the list grows quadratically with the number of elements.



2.3. Average Case

The average case time complexity of bubble sort is also O(n^2), where n is the number of elements in the input list. This means that on average bubble sort will require a number of comparisons and swaps proportional to the square of the number of elements in the list.
The reason for the O(n^2) average case complexity is that bubble sort s behavior largely depends on the initial order of the elements in the list. If the list is randomly ordered the algorithm will have to perform on average about (n * (n - 1)) / 4 comparisons and swaps to sort the list this can be derived from the fact that each element has an equal probability of being in any position and on average it will need to travel about halfway through the list to reach its correct position.



3. Selection Sort

Selection Sort is a simple sorting algorithm that repeatedly selects the smallest (or largest, depending on the sorting order) element from an unsorted part of the list and moves it to the front. It is an in-place comparison-based sorting algorithm and works well for small datasets but becomes inefficient for larger datasets due to its time complexity. And since it swaps non-adjacent elements it is not a stable sorting algorithm.

Here's how the Selection Sort algorithm works:

1. Start with the first element of the list as the minimum (or maximum) value.
2. Iterate through the rest of the list to find the smallest (or largest) element.
3. Swap the found smallest (or largest) element with the first element of the unsorted part of the list.
4. Move the boundary of the unsorted part one element to the right (i.e., exclude the already sorted elements).
5. Repeat steps 1 to 4 until the entire list is sorted.

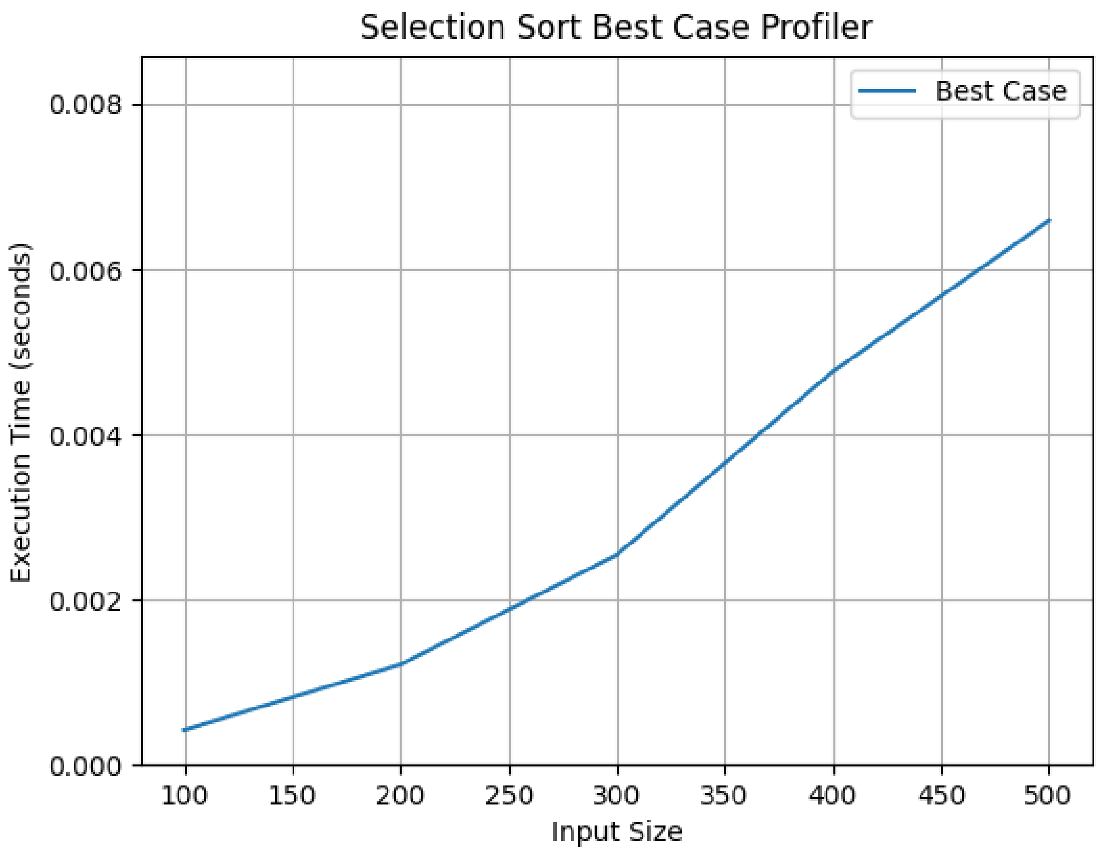
The process continues until all elements are in their correct positions, and the list is fully sorted. Selection Sort has a time complexity of O(n^2), where n is the number of elements in the list. It performs n swaps in the worst and average case scenarios and makes n - 1 passes through the list, making it inefficient for large datasets compared to more advanced sorting algorithms like Merge Sort or Quick Sort. However, Selection Sort has the advantage of being easy to understand and implement, and it is often used as an educational example to illustrate the concept of sorting algorithms.

3.1. Best Case

In the best-case scenario for Selection Sort, the input list is already sorted. When the list is already sorted, the algorithm's behavior is slightly optimized, but it still requires a full pass through the list to determine that the list is sorted. Therefore, the best-case time complexity of Selection Sort remains the same as the average and worst-case scenarios.

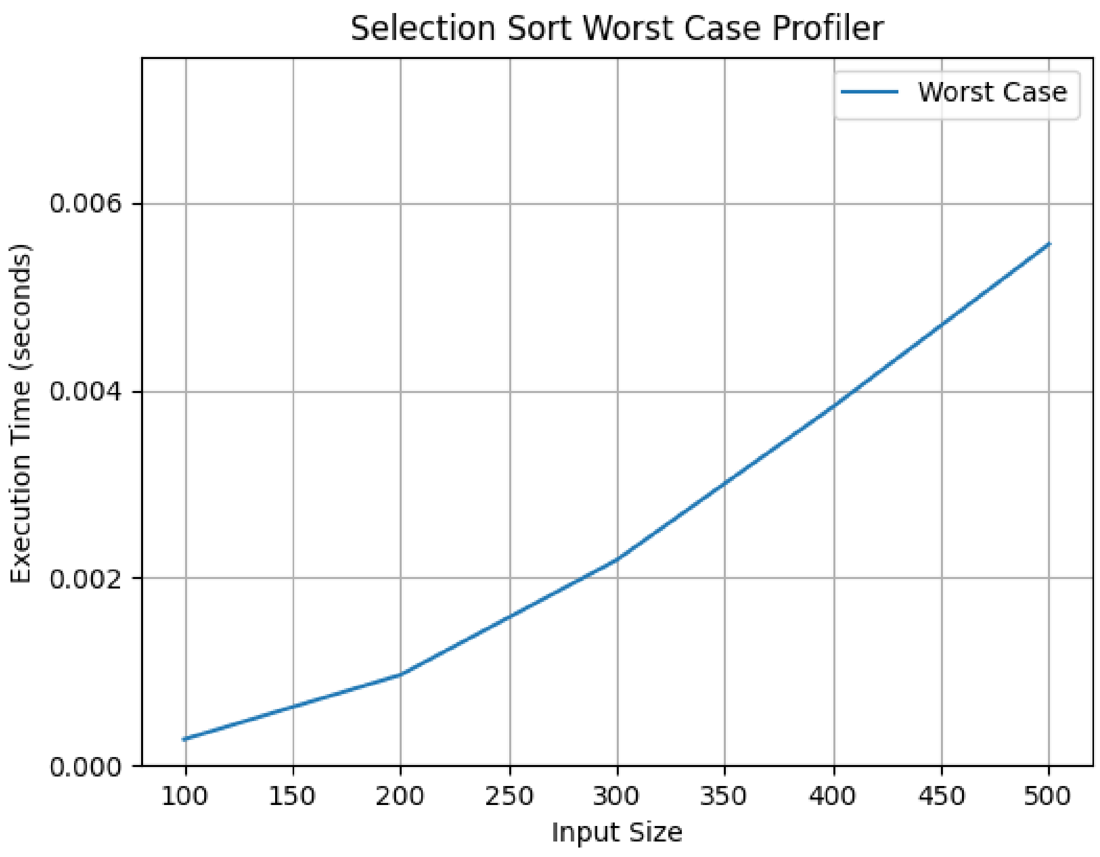
The best-case time complexity of Selection Sort is O(n^2), where n is the number of elements in the list. This is because the algorithm still needs to make n - 1 passes through the list, even if the

list is already sorted. During each pass, it performs comparisons to find the smallest element and swaps elements to place the smallest element at the beginning of the unsorted part of the list.
In the best case scenario for selection sort the input list is already sorted when the list is already sorted the algorithm's behavior is slightly optimized but it still requires a full pass through the list to determine that the list is sorted. Therefore, the best-case time complexity of selection sort remains the same as the average and worst-case scenarios.
The best-case time complexity of selection sort is O(n^2), where n is the number of elements in the list this is because the algorithm still needs to make n - 1 passes through the list even if the list is already sorted during each pass, it performs comparisons to find the smallest element and swaps elements to place the smallest element at the beginning of the unsorted part of the list.



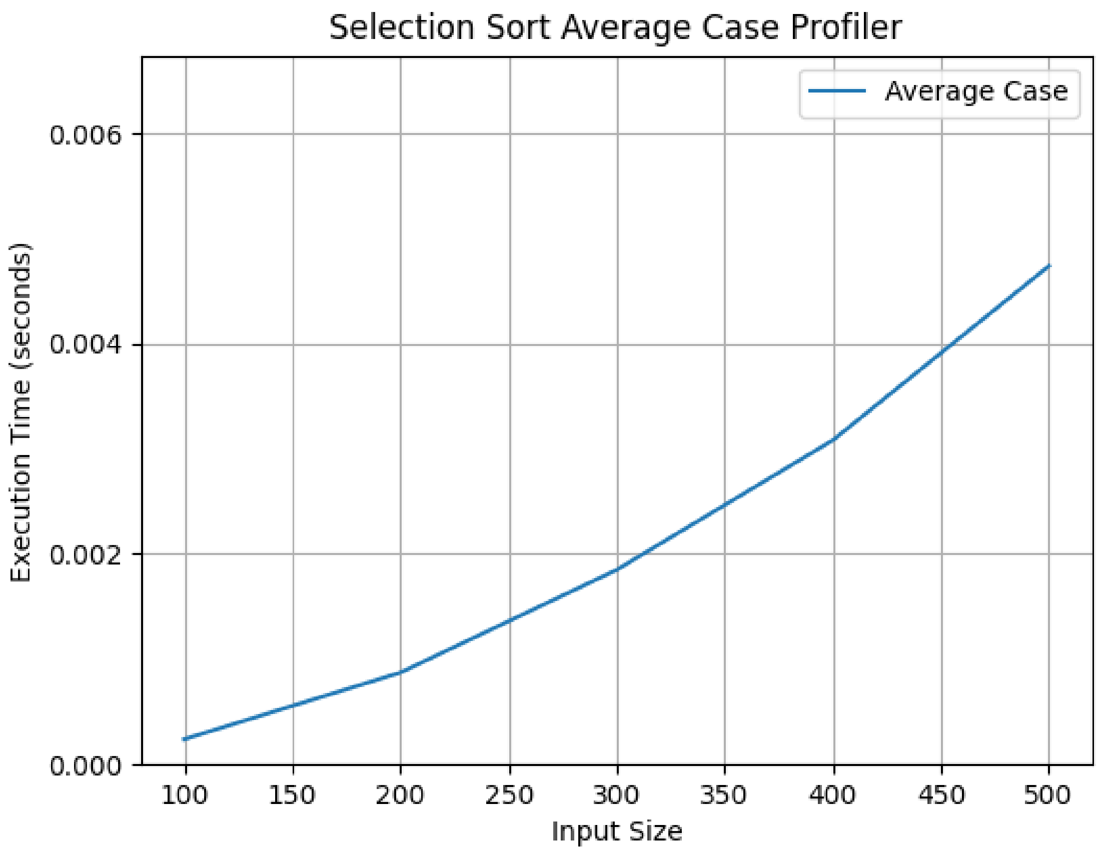
3.2. <u>Worst Case</u>

In the worst case scenario for selection sort the input list is in reverse order or sorted in descending order when the list is in reverse order the algorithm's performance is at its worst because it needs to make the maximum number of comparisons and swaps to sort the list. In the worst case scenario the time complexity of selection sort is still O(n^2), where n is the number of elements in the list this is because the algorithm needs to make n - 1 passes through the list and during each pass it performs comparisons to find the smallest element and swaps elements to place the smallest element at the beginning of the unsorted part of the list the number of comparisons and swaps in the worst case is given by the sum of the first n - 1 positive integers which is (n - 1) * n / 2. This leads to an overall time complexity of O(n^2), for the worst-case scenario the worst-case time complexity of selection sort makes it inefficient for large datasets as the time required to sort the list grows quadratically with the number of elements as a result selection sort is not recommended for use on large datasets and other sorting algorithms with better time complexities are preferred in practical applications.



3.3. <u>Average Case</u>

In the average case scenario for selection sort the input list is a random permutation of elements in other words the elements in the list are in no particular order and each possible ordering is equally likely the average case time complexity of selection sort is still O(n^2), where n is the number of elements in the list on average the algorithm needs to make n - 1 passes through the list and during each pass it performs comparisons to find the smallest element and swaps elements to place the smallest element at the beginning of the unsorted part of the list the number of comparisons and swaps in the average case is also proportional to (n - 1) * n / 2, which results in an overall time complexity of o n 2 for the average case scenario.



4. Insertion Sort

Insertion Sort is another simple sorting algorithm that works by building a sorted list one element at a time. It is an in-place, comparison-based sorting algorithm and is particularly efficient for small datasets or nearly sorted lists. The main idea behind Insertion Sort is to divide the list into a sorted and an unsorted part and then iteratively insert elements from the unsorted part into the correct position in the sorted part.

Here's how the Insertion Sort algorithm works:

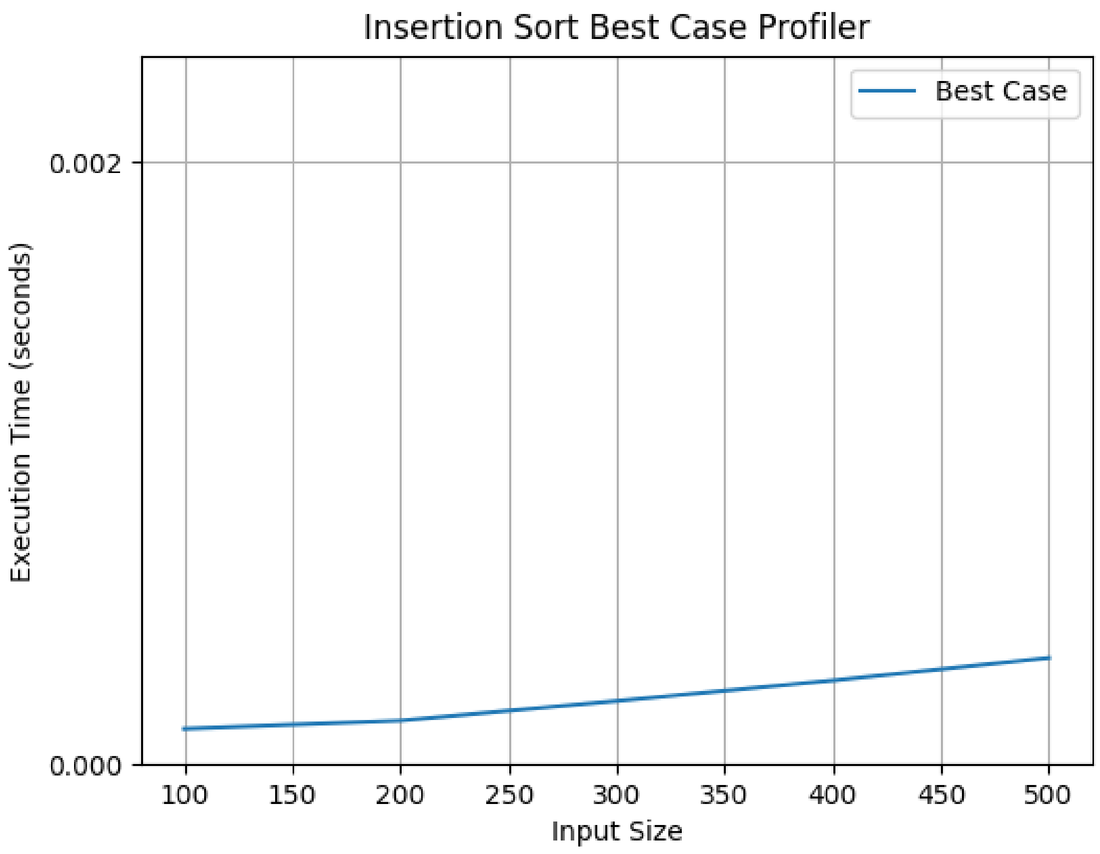
1. Start with the second element (index 1) of the list and assume it is the first element of the sorted part.
2. Compare this element with the element before it (in the sorted part).
3. If the current element is smaller (for ascending order) or larger (for descending order) than the element before it, swap them and continue comparing and swapping until the current element is in its correct position in the sorted part.
4. Move to the next element in the unsorted part and repeat steps 2 and 3 until all elements are in their correct positions.

The process continues until the entire list is sorted, and the sorted part keeps growing as new elements from the unsorted part are inserted into it.

Insertion Sort has a time complexity of O(n^2), where n is the number of elements in the list. In the best-case scenario (when the list is already sorted), it has a time complexity of O(n), making it more efficient than Selection Sort or Bubble Sort for nearly sorted data. However, for large datasets, Insertion Sort is less efficient compared to more advanced sorting algorithms like Merge Sort, Quick Sort, or Heap Sort, which have better average and worst-case time complexities. Nonetheless, Insertion Sort is simple to implement and has good performance on small datasets or partially sorted data.

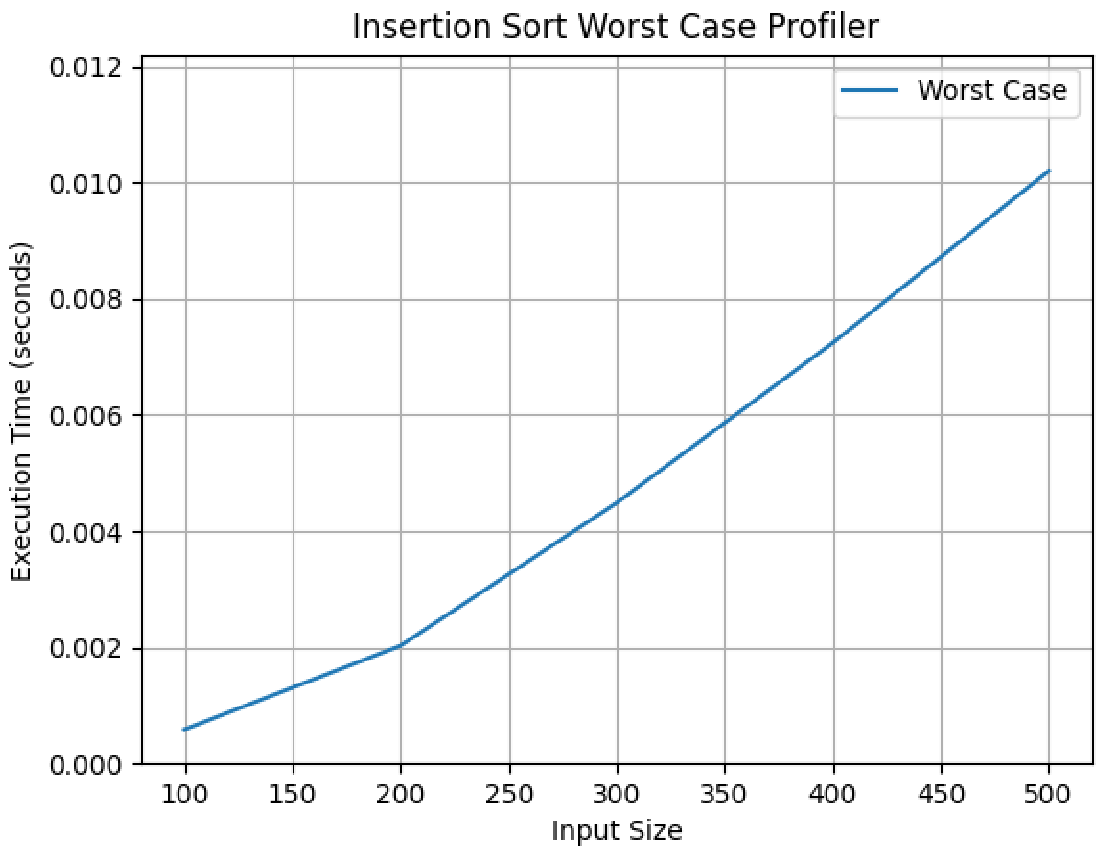
4.1. Best Case

The best case scenario for insertion sort occurs when the input list is already sorted in ascending order in this case the algorithm s behavior is optimized and it performs fewer comparisons and swaps compared to other scenarios when the list is already sorted insertion sort still needs to make n - 1 passes through the list but during each pass it only needs to compare each element with its previous element to determine if any swaps are necessary since the list is already sorted no swaps are needed and the algorithm exits early without making any changes to the list the best case time complexity of insertion sort is O(n), where n is the number of elements in the list this is because it only needs to make n - 1 comparisons to determine that the list is already sorted and no swaps are required



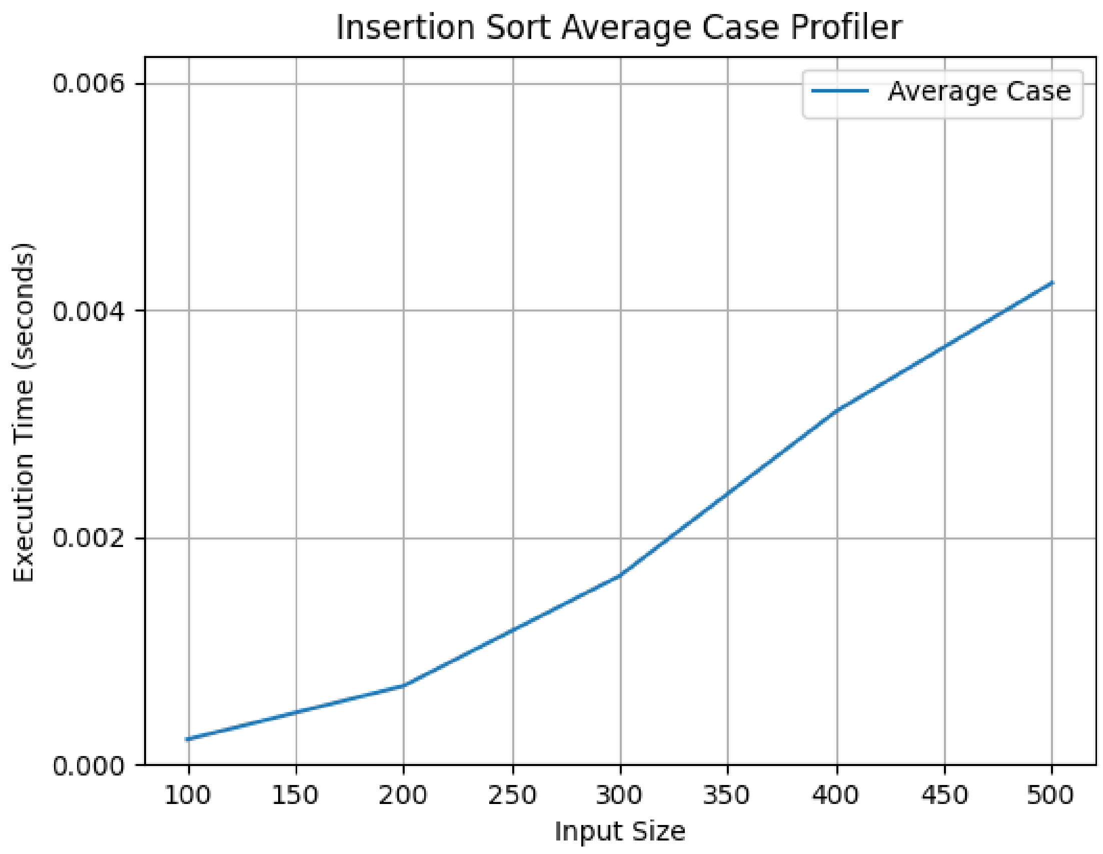
4.2. <u>Worst Case</u>

The worst case scenario for insertion sort occurs when the input list is sorted in descending order (i.e., in reverse order) in this case insertion sort performs the maximum number of comparisons and swaps to sort the list when the list is sorted in descending order insertion sort needs to compare each element in the unsorted part with every element in the sorted part to find its correct position for each element in the unsorted part it may need to swap with elements in the sorted part until it reaches its proper place in the sorted part the worst case time complexity of insertion sort is O(n^2), where n is the number of elements in the list this is because in the worst case for each element in the unsorted part insertion sort needs to perform approximately n 2 comparisons on average and may perform n 2 swaps resulting in a total of roughly n 2 operations per element therefore the overall number of operations in the worst case is approximately (n/2) * n = (n^2)/2, which gives a time complexity of O(n^2).



4.3. <u>Average Case</u>

The average case scenario for insertion sort occurs when the input list is a random permutation of elements in other words the elements in the list are in no particular order and each possible ordering is equally likely in the average case insertion sort needs to make approximately n^2/4 comparisons and n^2/4 swaps to sort the list where n is the number of elements in the list the actual number of comparisons and swaps may vary depending on the specific order of the elements in the input list the average case time complexity of insertion sort is still o n 2 where n is the number of elements in the list this is because on average for each element in the unsorted part insertion sort needs to perform about n 2 comparisons and swaps this leads to a total of (n/2) * n = (n^2)/2 comparisons and swaps resulting in a time complexity of O(n^2),



5. **Conclusion**

Sorting algorithms play a fundamental role in computer science and data processing, enabling the arrangement of data in a particular order for efficient retrieval and analysis. Among the basic sorting algorithms are Bubble Sort, Selection Sort, and Insertion Sort. These simple and intuitive algorithms have been widely used for educational purposes and as building blocks for more complex sorting techniques. However, they differ in their performance characteristics and suitability for various types of data. In this comparison, we analyze the strengths and weaknesses of Bubble Sort, Selection Sort, and Insertion Sort, shedding light on their efficiency, time complexity, adaptivity, stability, and potential use cases. We explore how these algorithms fare in real-world scenarios and discuss their relevance in modern data processing applications.
For the conclusion let's compare and analyze these three sorting algorithms in terms of:

- Efficiency and Time Complexity: Bubble Sort is simple to implement but inefficient for larger datasets. All its scenarios have time complexity of O(n^2). As a result, it performs a large number of comparisons and swaps even on partially sorted data, making it less practical for real-world applications on large datasets. Selection sort also have the same time complexity. Even though it minimizes swaps, it's still impractical for large datasets. On the other hand, Insertion Sort has time complexity of O(n) for its best-case scenarios, making it suitable for nearly sorted or small datasets. However, its average and worst-case complexities remain O(n^2), limiting its scalability for larger datasets.
- Adaptivity: Bubble Sort is adaptive, showing improved performance on partially sorted data, but its overall time complexity remains O(n^2), limiting its suitability for large datasets. In contrast, Selection Sort is not adaptive, consistently making the same number of comparisons and swaps regardless of the input data's initial order, making it inefficient for all cases. Insertion Sort is also adaptive and efficiently handles nearly sorted data, achieving its best-case time complexity of O(n) under such conditions, making it a valuable choice for dealing with partially ordered data.
- Stability: A sorting algorithm is considered stable if it maintains the relative order of equal elements with respect to their original positions in the input. Having this in mind, Bubble Sort and Insertion Sort are stable sorting algorithms, preserving the relative order of equal elements in the sorted output. They do not change the positions of elements with equal keys during the sorting process. However, Selection Sort is not stable and may

interchange the positions of elements with equal keys, potentially altering their relative order in the sorted output.

- Use Case: While Bubble Sort and Selection Sort are rarely used in practical applications due to their inefficiency for larger datasets. However, they can be useful for educational purposes or situations where simplicity of implementation is more important than performance, and the dataset size is small. But on the other hand, Insertion Sort can be beneficial in scenarios where the data is partially sorted or when the dataset size is small. It performs well on nearly sorted data, making it a viable option in certain cases.

In conclusion, while Bubble Sort, Selection Sort, and Insertion Sort are simple and easy to understand, they suffer from quadratic time complexity, limiting their practicality for large datasets. For most real-world applications, more efficient sorting algorithms like Merge Sort, Quick Sort, or Heap Sort are preferred. However, in specific situations where the data is already partially sorted or when simplicity is a priority for small datasets, Insertion Sort can be a reasonable choice.