Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab1 Report on Algorithms and Complexity

"COMP 314"

Submitted by:

Bijay Sapkota

Roll no. 43

Submitted to:

Rajani Chulyadyo(Ph.D)

DOCSE

1. Introduction

Sorting is a fundamental concept in computer science that involves arranging a collection of items in a specific order. This order can be ascending (from smallest to largest) or descending (from largest to smallest).

There are numerous sorting algorithms, each with its own advantages and disadvantages. The choice of algorithm depends on factors like the size of the data set, the desired order (ascending or descending), and whether the data is partially sorted.

Insertion sort is a sorting algorithm that works similarly to how one might organize a hand of playing cards. It iterates through the list, taking an unsorted element and inserting it into its correct position within a sorted sub-list.

Insertion sort has a time complexity of O(n^2) in the worst case, meaning the sorting time grows quadratically with the number of elements (n). However, it has a best-case complexity of O(n) when the data is already sorted or nearly sorted. This makes it a good choice for small lists or data sets that might be partially ordered.

Selection sort is another relatively simple sorting algorithm. It works by repeatedly finding the minimum (or maximum) element in the unsorted portion of the list and swapping it with the first element of that unsorted portion.

2. Implementation

Pre-sorted arrays were generated using the **np.arange** function to represent the best-case scenario for the sorting algorithms (already sorted data). For the worst-case scenario (descending order), a separate function named **invert_array** was used to create reversed sorted arrays. Finally, to represent typical unsorted data, arrays of random integers were generated using **np.random.normal**(to generate normally distributed values and randomize them) with a normal distribution around a specific value and then converted to integers using **astype(int)**.

2.1 Testing and Analysis

(a) Test Cases:

Unit tests were conducted to ensure the correctness of both sorting algorithms for various input scenarios, including empty lists, sorted lists, and reverse-sorted lists.

(b) Performance Measurement:

Random data sets of varying sizes from 0 to 1000 were generated. The execution time of both Insertion Sort and Selection Sort was measured for each data set size. The measurements were repeated multiple times to account for potential variations.

(c) Observations:

The results were visualized in an input-size vs execution-time graph. As expected, the execution time generally increased with input size for both algorithms.
While having same upper bound insertion sort was observed to be efficient over selection sort for the best case (already sorted data).

The average case too performed better in terms of time taken for insertion sort but the bound was defined by O(n^2) as the curve was observed to be of quadratic nature.

*Note:*However, there were likely spikes in the graph due to factors like context switching in the operating system.

4. Discussion

According to time complexity theory, Insertion Sort and Selection Sort have a worst-case time complexity of O(n^2). This indicates that the number of operations (comparisons and swaps) grows quadratically with the input size. This implies that these algorithms become increasingly inefficient for large datasets.

5. Selection and Insertion Sort in Practice

Despite the O(n^2) time complexity, Selection Sort might still be used in some scenarios:

- **Simplicity:** Both algorithms are very easy to understand and implement. This makes them ideal for educational purposes or for situations where a quick and dirty sorting solution is needed. Their simplicity also translates to less code to maintain and debug.
- **Small Data Sets:** For small data sets, the constant factor hidden in the O(n^2) notation can make them surprisingly fast in practice. The overhead of more complex sorting algorithms might outweigh the benefits for small lists.

- **Partially Sorted Data:** Insertion sort, in particular, can be very efficient for data sets that are already partially sorted. This is because it only needs to shift elements a few positions on average, as opposed to completely re-arranging them.
- **Space Complexity:** Both Insertion Sort and Selection Sort have a space complexity of O(1), which is considered constant space complexity. This means they use a fixed amount of additional space regardless of the size of the input data set.

However, it's important to be aware of their limitations:

- **Large Data Sets:** As data sizes grow, the O(n^2) complexity becomes a significant bottleneck. For large data sets, it's recommended to use more efficient sorting algorithms like merge sort (O(n log n)) or quicksort (average O(n log n)).

6. Recommendations

For future exploration, it would be beneficial to:

- Compare the performance of these algorithms with other sorting techniques like Merge Sort or Quick Sort, which have a better average-case time complexity of O(n log n).
- Analyze the impact of pre-sorted or partially sorted data on the performance of these algorithms.

This report provides a basic analysis of the implemented sorting algorithms. Further investigation can delve deeper into the performance characteristics and explore more complex sorting techniques.

7.Appendix

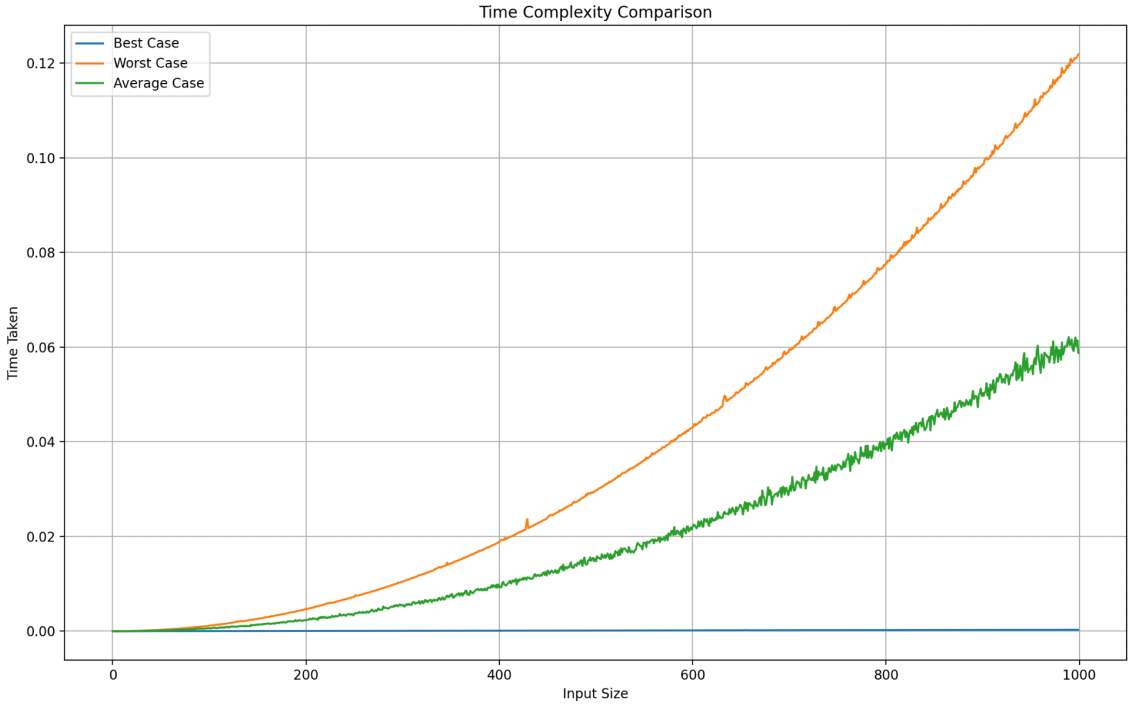

Fig: Time Complexity Comparison in Insertion Sort

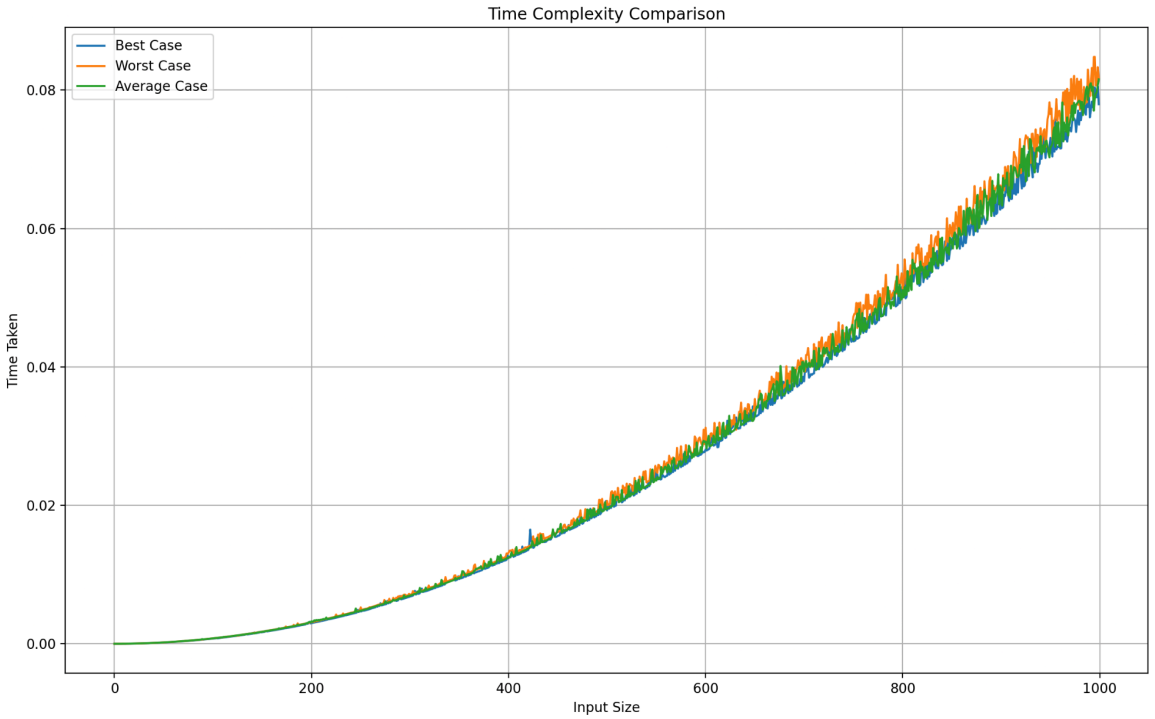

Fig: Time Complexity Comparison in Insertion Sort