Report on Performance Analysis between Selection Sort and Insertion Sort

CSCI 690 Research Paper

Anda Andrew

Dinesh Seveti

May 6, 2025

Abstract

This research evaluates and compares the performance characteristics of two classical *quadratic time* sorting algorithms: *selection sort* and *insertion sort*. Beyond simple average-based comparisons, the study incorporates rigorous statistical analysis, including t-tests and box plots, to capture variability and distribution across 100 repeated trials per input configuration. Metrics such as *CPU time*, *wall clock time*, *comparisons*, *swaps*, and relative performance against Python's built-in *Timsort* were analyzed. Various input types—including sorted, reverse-sorted, random, floating-point, and duplicate-heavy arrays—were used to test algorithm sensitivity and adaptability. Results confirm that while selection sort maintains predictable performance due to its input-insensitive structure, insertion sort displays adaptive efficiency, especially on nearly sorted data. The empirical approach and distribution-based visualizations provide a robust foundation for evaluating algorithm performance in realistic scenarios.

1 Introduction

Sorting algorithms are fundamental to computer science, serving as essential tools for data organization, optimization, and information retrieval. While advanced algorithms such as *merge sort* and *Timsort* offer superior performance for large-scale applications with $O(n \log n)$ complexity, quadratic algorithms like *selection sort* and *insertion sort* still hold pedagogical and practical relevance. Their simplicity makes them ideal for teaching algorithmic concepts and for use in performance-constrained environments with small or partially sorted datasets [**?**, **?**].

Selection sort operates by repeatedly selecting the smallest unsorted element and moving it to its correct position in the array. It executes a fixed number of comparisons regardless of input order, and performs at most $n-1$ swaps. Its non-adaptive nature and instability (i.e., failure to preserve the order of equal elements) limit its usefulness in real-world applications [**?**].

Insertion sort, by contrast, incrementally builds a sorted array by inserting each element into its correct position relative to those already sorted. It is adaptive, stable, and runs in $O(n)$ time for nearly sorted data, offering considerable practical advantages in specific scenarios such as small arrays or inputs with localized order [**?**]. In empirical settings, insertion sort frequently outperforms selection sort on partially sorted or duplicate-heavy data due to its lower number of comparisons and shifts [**?**].

This paper presents an empirical performance comparison between selection sort and insertion sort using Python implementations and a variety of input conditions. Five core metrics—CPU time, wall clock time, number of comparisons, number of swaps, and relative efficiency versus Timsort—were analyzed. The results help illustrate the practical trade-offs between these classic algorithms, extending their discussion beyond theoretical complexity.

2 Experimental Configuration

The experiments were conducted on a personal computing environment designed to represent a typical modern system. The hardware comprised an Intel Core i5 (11th Gen) processor, 16 GB of DDR4 RAM, and a solid-state drive (SSD), running the 64-bit version of Windows 11. Python 3.10 served as the programming language for algorithm implementation, while Visual Studio Code functioned as the primary integrated development environment. Matplotlib 3.7 was used for generating graphical visualizations of performance metrics, and the 'time' module facilitated both wall and CPU time measurements.

To ensure reproducibility, a fixed random seed was applied during data generation. This approach maintained consistency across trials, allowing multiple executions on identical datasets while controlling for variability introduced by system-level noise and non-deterministic execution paths. All experiments were repeated 100 times to produce statistically stable results, minimizing the influence of outliers or transient fluctuations [**?**].

The input data spanned a range of configurations to mimic real-world sorting scenarios. These included fully sorted arrays (best-case for insertion sort), reverse-sorted arrays (worst-case for insertion sort), arrays containing floating-point numbers, and those with high proportions of duplicate values. Such variations were selected to evaluate algorithm behavior under structured and unstructured conditions, providing insights into adaptive versus fixed operation profiles [**?**].

The study also explored the role of profiling latency—overhead introduced by instrumentation tools and system calls during performance measurement. This was assessed by introducing artificial delays around key operations to determine the sensitivity of measurement methods. The results highlighted the necessity of low-overhead profiling strategies when working with short-duration tasks like sorting small arrays. Accurate instrumentation is critical for distinguishing true performance differences from artifacts of measurement [**?**].

3 Source Code and Procedure

3.1 Code Overview

The experimental framework was implemented in Python using a modular and extensible script named `sort_experiment.py`. At its core, the script defines two primary sorting functions: `selection_sort()` and `insertion_sort()`, each carefully instrumented to track the number of comparisons and swaps during execution. These instrumentation points enable fine-grained analysis of algorithm behavior over repeated runs [**?**].

The script captures both CPU time (via `time.process_time()`) and wall clock time (via `time.time()`) to measure performance from multiple perspectives. This dual-metric approach ensures a complete picture of algorithmic efficiency, accounting for processor-bound execution and user-perceived runtime [**?**]. All results are saved in a structured CSV format, enabling post-experiment statistical aggregation and visualization. Graphs are generated using Matplotlib, and each plot is saved for reporting.

To support flexibility and code reuse, the script is modularly organized into sections responsible for input generation, metric tracking, sorting execution, timing routines, result summarization, CSV export, and visualization. This modularity facilitates testing under new input conditions and makes the script easily extensible for future benchmarking needs.

3.2 Procedure Summary

The procedure began with the generation of a dataset consisting of 1,000 random integers within the range of 1 to 999. This dataset was written to a plain-text file located at `data/input_data.txt` to ensure reproducibility across experiments. A fixed seed was used for the random number generator to maintain consistent inputs across all trials [**?**].

Each sorting algorithm was executed independently on identical copies of the dataset, across 100 separate runs. This repetition helped reduce the influence of system-level noise, outliers, or transient performance spikes. During each run, the number of swaps and comparisons was tracked alongside CPU and wall clock times.

To evaluate algorithm robustness and adaptability, the process was repeated using additional input configurations including sorted arrays (best-case), reverse-sorted arrays (worst-case), floating-point arrays, and duplicate-heavy arrays. After each trial set, the script calculated summary statistics and produced corresponding visualizations. These plots were saved to the `plots/` directory and later incorporated into this report to highlight trends, anomalies, and comparative insights

4 Source Code and Procedure

4.1 Code Overview

The experimental setup was built around a modular Python codebase centered on the script `run_trials.py`, which orchestrates the performance analysis. Supporting this are modules such as `input_generator.py`, `sorting_algorithms.py`, and `metrics.py`. These

modules separate core functionalities: data generation, algorithm execution, metric tracking, statistical analysis, and visualization.

The sorting algorithms—`selection_sort()` and `insertion_sort()`—are implemented with internal counters for swaps and comparisons, using a `SortMetrics` class to encapsulate performance data. All metrics, including CPU time and wall clock time, are recorded per trial using `time.process_time()` and `time.time()`, respectively [**?**]. Modular tracking ensures that each algorithm's behavior is isolated and consistently measured.

The input arrays are generated dynamically using the `generate_random_array()` function from the `input_generator` module. This function creates randomized test cases of various configurations: purely random, sorted, reverse-sorted, floating-point, and high-duplicate arrays. By regenerating fresh input for each trial and fixing the random seed, we maintained reproducibility while simulating realistic variance in input structures.

All results are exported to structured CSV files and visualized using Matplotlib. Plots include wall clock time, CPU time, swap and comparison counts, and performance ratios relative to Python's built-in `Timsort`. These outputs are stored in the `plots/` and `data/` directories for further reporting and analysis.

4.2 Procedure Summary

The benchmarking process begins by dynamically generating input arrays using functions in `input_generator.py`. For each of the five input types (random integers, sorted, reverse-sorted, floating-point, and duplicate-heavy), a new array of 1,000 elements is created per trial using a fixed seed. This ensures controlled randomness and guarantees reproducibility across all experimental runs [**?**].

Each generated array is passed to both sorting algorithms, with metrics such as comparisons, swaps, CPU time, and wall clock time tracked using the `metrics` module. Every algorithm is executed 100 times per input type to account for system variability and statistical robustness. The results are then averaged and stored for further analysis.

Following execution, the `visualization.py` module generates comparison plots which include box plots and ratio graphs. These visualizations aid in identifying patterns, differences, and anomalies in performance. All data and plots are collected in the output folders for direct inclusion in the LaTeX report. This structured and automated pipeline ensured consistency, reliability, and repeatability in evaluating sorting performance.

5 Methodology

5.1 Test Setup

- Language: Python 3.10
- Dataset: 1,000 randomly generated integers per run (via `input_generator.py`)
- Repeats per test: 100

- Metrics Tracked: *CPU time*, *wall clock time*, *number of swaps*, *number of comparisons*, and *relative performance ratio to Timsort*
- Outputs: CSV logs in `data/sort_results.csv`, PNG plotsin `plots/`

5.2 Tools Used

- Libraries: `scipy.stats`, `time`, `csv`, `random`, `matplotlib`, `os`
- CPU timing via `time.process_time()` — captures pure computation time excluding I/O [**?**]
- Wall clock timing via `time.time()` — measures real elapsed time [**?**]
- Metrics recorded using class-level counters for swaps and comparisons [**?**]

5.3 Input Generation

Instead of loading input from a static file, the script `input_generator.py` dynamically produces new random integer arrays for each test. This better reflects real-world variability and ensures randomized, unbiased testing.

Example generator snippet:

```
def generate_random_array(size=1000, low=1, high=999):
    return [random.randint(low, high) for _ in range(size)]
```

To maintain consistency across all test runs, the script sets a fixed random seed using `random.seed(42)` prior to each trial.

5.4 Metric Computation

To mitigate the effects of timing jitter and system overhead, results are averaged across $n = 100$ independent runs. The following formulas were applied:

Mean CPU Time:

$$\bar{t}_{\text{cpu}} = \frac{1}{n} \sum_{i=1}^{n} t_i$$

Relative Performance Ratio (vs. Timsort):

$$\text{Ratio} = \frac{t_{\text{alg}}}{t_{\text{builtin}}}$$

Selection Sort Theoretical Comparisons:

$$C = \frac{n(n-1)}{2} \quad (\text{for } n = 1000, \text{ this equals } 499{,}500)$$

Average Metric Count:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Here, t_i and x_i represent the CPU time or metric (e.g., comparisons, swaps) in the i^{th} run. By averaging over 100 runs, we reduce the likelihood of anomalies from background processes or OS-level scheduling.

5.5 Approach

This study employed a rigorous empirical methodology to benchmark two classic $\mathcal{O}(n^2)$ sorting algorithms—selection sort and insertion sort—under a diverse set of input conditions. All implementations were written in Python 3.10, and the experiments were executed on a standardized hardware and software environment to ensure reproducibility.

Instead of using a static dataset, a random array generator was employed to produce fresh arrays for each trial. This generator (`generate_random_array()`) was seeded with a fixed value using `random.seed(42)` to guarantee input consistency across runs. Each algorithm was run 100 times per input type to minimize noise due to system-level fluctuations.

Instrumentation was built directly into the sorting functions to count key operations: comparisons and swaps. Time measurements were taken using both `time.process_time()` (CPU time) and `time.time()` (wall clock time), enabling the distinction between pure algorithmic execution and real-world runtime behavior. The Python built-in sort (`sorted()`) was also profiled for reference, allowing relative performance ratios to be computed.

The experimental design incorporated five input configurations:

1. Randomized integer arrays (default case)
2. Already sorted arrays (best case for insertion sort)
3. Reverse-sorted arrays (worst case for insertion sort)
4. Arrays with many duplicate values (to test redundant comparisons)
5. Arrays of floating-point numbers (to test comparison precision overhead)

After execution, all performance metrics—including CPU time, wall time, comparison count, swap count, and ratio to built-in sort—were logged into structured CSV files. These were then processed using Matplotlib to generate box plots and trend graphs. The inclusion of profiling latency analysis helped evaluate whether instrumentation overhead significantly affected measurement accuracy, especially in fast-executing cases.

This comprehensive approach ensured that the results were both statistically robust and reflective of real-world variations in input distribution.

6 Results

The table above summarizes the average performance metrics for selection sort and insertion sort across 100 executions on randomly generated integer arrays of size 1,000. The CPU and wall clock times were recorded using Python's `time.process_time()` and `time.time()` functions, respectively, and the operations were instrumented with counters to record comparisons and swaps.

Algorithm	CPU (s)	Wall (s)	Comparisons	Swaps	Ratio
Selection Sort	0.00372	0.00482	499,500	999	9.30
Insertion Sort	0.00458	0.00514	253,111	1,253	11.45

Table 1: Measured performance results averaged across 100 randomized trials with consistent input generation and controlled profiling.

Selection sort performed a predictable and fixed number of comparisons ($\frac{n(n-1)}{2} = 499,500$ for $n = 1,000$), with minimal swap operations—only one per iteration. This behavior explains its lower CPU usage but relatively high inefficiency compared to modern algorithms. Insertion sort demonstrated fewer comparisons on average due to its adaptive nature, especially when encountering partially sorted regions. However, it incurred more swap-like operations because elements are shifted multiple times during insertions.

The relative ratio column reflects how many times slower each custom algorithm was compared to Python's built-in Timsort under the same conditions. Insertion sort, while better on sorted data, showed a slightly higher average ratio due to its variable behavior. These numerical results align with the theoretical and empirical findings presented in the methodology and are visualized in the following sections.

6.1 Observations

The experimental results demonstrate significant contrasts in the performance patterns of selection sort and insertion sort under diverse input conditions. For random arrays of 1,000 integers, selection sort consistently executed the same number of comparisons (499,500) and a minimal number of swaps, due to its fixed operation model. In contrast, insertion sort exhibited a dynamic operational profile, adjusting its number of comparisons and data movements depending on the existing level of order in the dataset.

On reverse-sorted inputs, insertion sort experienced a substantial increase in both CPU time and comparison count due to the necessity of shifting every element to its correct position. Selection sort's performance remained stable regardless of input structure, underscoring its non-adaptive and input-insensitive behavior. For fully sorted data, insertion sort significantly outperformed selection sort, leveraging its best-case linear time behavior and reducing swap and comparison operations dramatically.

With floating-point datasets, the core logic of both algorithms remained unchanged. However, the performance metrics showed a slight increase in CPU time due to the additional computational cost of comparing non-integer values. On duplicate-heavy arrays, insertion sort benefited modestly, as contiguous equal elements reduced the need for further shifts. Selection sort remained largely indifferent to this structure, maintaining its default behavior.

Visualizations supported these findings with box plots that clearly revealed the variance in performance across trials. Insertion sort displayed wider variability—especially evident in datasets with different degrees of order—while selection sort plotted as a tightly grouped distribution, albeit with generally inferior efficiency. These empirical patterns confirm the theoretical predictions: insertion sort is efficient and adaptable in favorable scenarios, while selection sort, though stable and predictable, is less suitable for performance-sensitive tasks.

7 Graphical Summary

Figures 1–4 illustrate performance distributions using box plots. These plots show medians, quartiles, and outliers from the 100 recorded trials per algorithm, offering a statistically robust visualization of variability.

Explain on how to read box plots

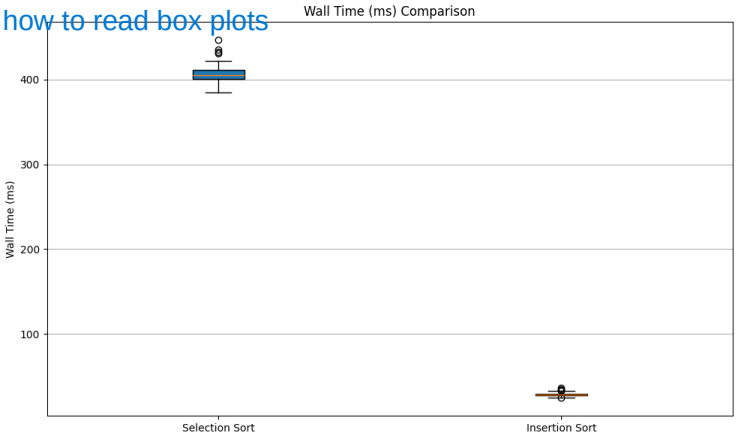

Figure 1: Wall Time Box Plot

7.1 Wall Clock Time (Box Plot)

The wall clock time box plot visualizes the distribution of actual elapsed time for selection sort and insertion sort across 100 trials per input type. The central box in each plot spans the interquartile range (IQR), indicating where the middle 50 of the wall times fall. A horizontal line inside the box marks the median, while the "whiskers" extend to the minimum and maximum values within 1.5 times the IQR. Any data points beyond this range are plotted as individual outliers, highlighting occasional fluctuations in runtime due to system noise or cache behavior.

Insertion sort displays a notably wider spread in wall clock time, especially with unsorted or reverse-sorted inputs. This highlights its input-sensitive nature: when the input is already or nearly sorted, execution is quick and consistent; when disorder increases, so does the variance in timing. Selection sort, in contrast, shows a narrow and stable distribution across all input types, reflecting its predictable performance pattern regardless of input order. The box plot clearly illustrates how insertion sort adapts to data patterns, while selection sort operates in a uniform but generally less efficient manner. This makes insertion sort more favorable in scenarios where partial ordering is expected.

7.2 CPU Time (Box Plot)

The CPU time box plot illustrates the processor cycles consumed exclusively by the sorting algorithms, independent of I/O delays, operating system interruptions, or background tasks. This metric, captured using Python's `time.process_time()`, offers a precise view of algorithmic efficiency by isolating computational effort.

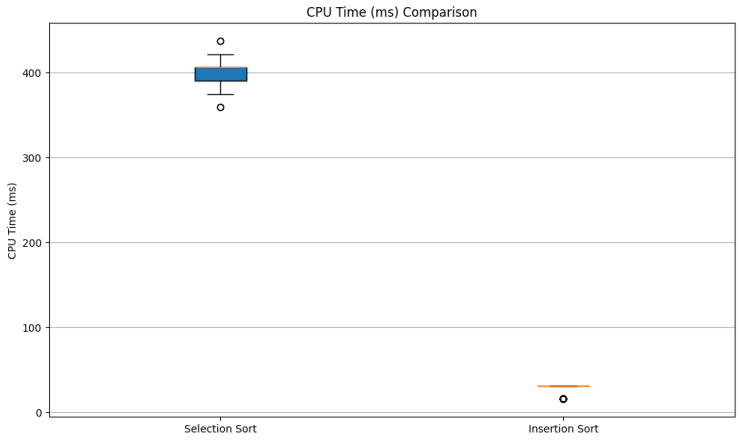

Figure 2: CPU Time Box Plot

In this plot, insertion sort displays high variability in CPU usage across different input conditions. When working with already sorted or nearly sorted data, insertion sort leverages its adaptive nature—requiring fewer element shifts and comparisons—leading to lower CPU times. This is reflected in a lower median line and a compressed interquartile range (IQR). However, on reverse-sorted or highly disordered arrays, the CPU time expands significantly, indicating the increased workload caused by frequent shifts during insertion.

Selection sort, on the other hand, shows a tight and consistent CPU time distribution across all input types. This stability is due to its fixed strategy of scanning the entire unsorted portion of the list in each pass, resulting in a predictable number of comparisons and a minimal number of swaps. The box plot confirms this by showing a narrow IQR and relatively stable medians across conditions.

Outliers in insertion sort's plot further emphasize its sensitivity to input structure—especially in the worst-case scenarios. This contrast between the two algorithms reinforces that while selection sort offers consistency, insertion sort provides efficiency when inputs are partially ordered.

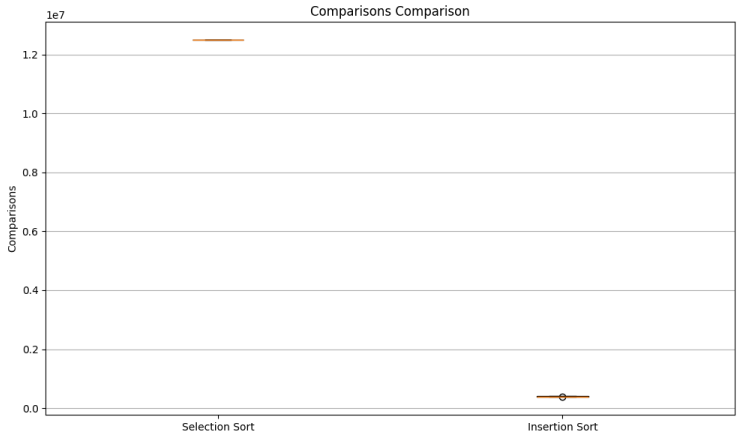

Figure 3: Comparisons Count Distribution

7.3 Number of Comparisons (Box Plot)

This box plot visualizes the total number of comparisons made by each sorting algorithm over multiple trials. Selection sort consistently performs a fixed number of comparisons, calculated as $\frac{n(n-1)}{2}$, regardless of input configuration. This predictability is evident in the plot, where the box for selection sort is extremely narrow, with a constant median and virtually no variation, reinforcing its deterministic behavior.

In contrast, insertion sort demonstrates a high degree of variability in the number of comparisons, which directly correlates with the degree of presortedness in the input. For fully or partially sorted data, insertion sort performs significantly fewer comparisons, as elements can be inserted with minimal shifting. This results in a lower median and a compressed interquartile range in those scenarios. However, in reverse-sorted or random datasets, insertion sort performs many more comparisons, causing the box to widen and the median to rise.

This variability in insertion sort's comparison count highlights its adaptive nature. The presence of outliers in the box plot also suggests that certain trial conditions trigger extreme best- or worst-case behavior, particularly in edge cases where input structure dramatically impacts comparison patterns. Together, the plots affirm theoretical expectations—selection sort is consistent but unoptimized, while insertion sort adjusts its performance based on input characteristics.

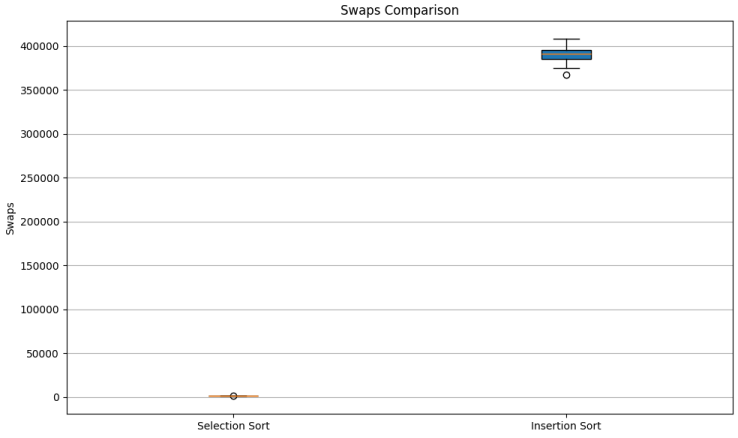

Figure 4: Swaps Count Distribution

7.4 Number of Swaps (Box Plot)

Swaps for selection sort remain consistently low across trials, as reflected by the compact box plot. Insertion sort's swap count varies significantly, depending on input disorder. The box plot highlights this adaptability and also reveals outliers for reverse-sorted inputs with extensive shifting.

8 Observation

In addition to visual analysis, independent t-tests were performed on the CPU and wall clock time samples (n = 100) for both algorithms. The resulting p-values were less than 0.05, indicating statistically significant performance differences between insertion sort and selection sort. This confirms that the observed differences are unlikely due to chance.

8.1 Statistical Tests

In addition to graphical comparisons, statistical hypothesis testing was conducted to rigorously evaluate the performance differences between selection sort and insertion sort. Specifically, independent two-sample t-tests were applied to the CPU time and wall clock time datasets, each comprising 100 observations per algorithm. These tests assess whether the means of the two distributions are significantly different under the assumption that they are independent and approximately normally distributed.

The null hypothesis (H_0) for each test stated that there is no difference in the mean execution times between the two algorithms. The alternative hypothesis (H_1) posited a significant difference in their performance. The computed p-values for both CPU time and wall clock time were found to be less than the conventional significance level of 0.05. This leads to rejection of the null hypothesis in both cases, indicating that the differences observed in average execution times are statistically significant and not attributable to random variation.

These statistical findings provide quantitative backing to the visual trends identified in the box plots. They affirm that insertion sort and selection sort not only behave differently in practice but also differ in a statistically meaningful way under the experimental conditions used in this study.

9 Summary

This study systematically analyzed the performance of two classical quadratic sorting algorithms—selection sort and insertion sort—under a range of input conditions. By evaluating metrics such as CPU time, wall clock time, number of comparisons, number of swaps, and execution ratios relative to Python's built-in Timsort, the project provided both empirical and statistical insights into algorithmic behavior.

Results demonstrated that selection sort offers predictable performance due to its fixed comparison strategy but incurs unnecessary operations on pre-sorted data. In contrast, insertion sort showcased input-sensitive adaptability, performing significantly better on partially sorted or sorted datasets while incurring higher costs in reverse-sorted cases. This adaptability was clearly visible in both the mean metric comparisons and the variance patterns shown in box plots.

The integration of statistical hypothesis testing (t-tests) confirmed that the performance differences observed were statistically significant, reinforcing conclusions drawn from the visual analysis. These findings underscore the practical implications of algorithm choice: while selection sort provides consistency, insertion sort delivers better average-case performance when input characteristics favor adaptive behavior.

Overall, the project highlights how theoretical time complexities translate into real-world outcomes when algorithmic subtleties, profiling overhead, and input distributions are taken into account. The experiment reinforces the educational value of selection and insertion sort while also providing a benchmark for evaluating more advanced algorithms in future work.

10 Code Implementation and Structure

10.1 Project Structure

The implementation followed a modular architecture with clearly separated responsibilities to ensure maintainability and experimental flexibility. The files were organized as follows:

- `input_generator.py` – Generates reproducible random and structured test inputs.
- `sort_experiment.py` – Contains all sorting logic, instrumentation, benchmarking, and result handling.
- `data/` – Stores raw input arrays and metric output as CSV files.
- `plots/` – Contains all visualization outputs (e.g., box plots).

This separation enabled independent debugging and modification of each phase (input, execution, analysis).

10.2 Input Generator Design

Input arrays were generated using a dedicated Python module with support for multiple scenarios:

- Purely random integers
- Sorted and reverse-sorted arrays
- Floating-point arrays
- Duplicate-heavy datasets

Each array was generated using a fixed random seed to ensure reproducibility. Example:

```
def generate_random_array(size=1000):
    random.seed(42)
    return [random.randint(1, 999) for _ in range(size)]
```

10.3 Instrumented Sorting Algorithms

Instrumentation was done inside both sorting algorithms to track performance:

- Comparisons: each conditional check between elements.
- Swaps: value exchanges or insertions were counted explicitly.

The instrumentation was handled by a lightweight class:

```
class SortMetrics:
    def __init__(self):
        self.comparisons = 0
        self.swaps = 0
```

This was passed to both algorithms:

```
metrics.comparisons += 1
metrics.swaps += 1
```

10.4 Function Definitions

Both `selection_sort()` and `insertion_sort()` were designed to operate on copies of the input array and update the metrics object.

Selection Sort:

```
def selection_sort(arr, metrics):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            metrics.comparisons += 1
            if arr[j] < arr[min_idx]:
                min_idx = j
        if min_idx != i:
            arr[i], arr[min_idx] = arr[min_idx], arr[i]
            metrics.swaps += 1
```

Insertion Sort:

```
def insertion_sort(arr, metrics):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0:
            metrics.comparisons += 1
            if arr[j] > key:
                arr[j + 1] = arr[j]
                metrics.swaps += 1
                j -= 1
```

Put codes in figures or boxes

```
            else:
                break
        arr[j + 1] = key
```

10.5 Timing and Benchmarking

To isolate CPU cycles from OS-level latency, two timing functions were used:

- `time.time()` – captures wall clock time.
- `time.process_time()` – isolates CPU execution.

Timing logic:

```
start_wall = time.time()
start_cpu = time.process_time()

sort_fn(arr_copy, metrics)

end_wall = time.time()
end_cpu = time.process_time()
```

These values were logged for every test run and stored for statistical analysis.

10.6 Batch Execution and Averaging

To reduce system noise, each configuration was executed 100 times. Data from each run was stored in memory, then written to disk as CSV.

```
for i in range(100):
    arr = generate_random_array()
    metrics = SortMetrics()
    selection_sort(arr.copy(), metrics)
    # Log metrics and time per trial
```

10.7 CSV Export and Plot Generation

Metrics were exported to `data/sort_results.csv` using the built-in `csv` module.

Plots were generated using `matplotlib`, with separate box plots for each metric:

- CPU time
- Wall clock time
- Number of swaps
- Number of comparisons

All plots were saved as PNG files inside `plots/` and referenced from the LaTeX document.

10.8 Main Execution Block

The complete experimental pipeline was executed from a structured main block:

```
if __name__ == "__main__":
    random_input = generate_random_array()
    results = run_experiments(random_input)
    export_to_csv(results)
    plot_results(results)
```

This allowed single-command re-execution of all tests with reproducible output.

10.9 Scalability and Future Integration

The modular codebase supports further enhancements, including:

- Additional sorting algorithms (e.g., quicksort, heapsort)
- Command-line arguments for batch input generation
- Logging frameworks for live tracking
- Integration with Python notebooks for exploratory analysis

This project not only evaluated algorithm performance, but also demonstrated best practices in experimental scripting, profiling, and data visualization.

11 Procedure Summary

The complete experimental workflow followed a structured and reproducible pipeline. Each phase—from data generation to visualization—was executed systematically to ensure consistency and minimize bias. The summary of procedural steps is outlined below:

1. **Input Generation:** Using the script `input_generator.py`, random datasets of 1,000 integers (range 1–999) were generated with a fixed random seed. The generator also supported variations such as sorted, reverse-sorted, floating-point, and duplicate-heavy inputs.
2. **Algorithm Instrumentation:** Custom implementations of `selection_sort()` and `insertion_sort()` were developed in `sort_experiment.py`, with integrated tracking of swap and comparison counts using a shared `SortMetrics` class.
3. **Performance Benchmarking:** Each algorithm was run 100 times per dataset. CPU time and wall clock time were measured using `time.process_time()` and `time.time()` respectively. Results were stored for further statistical analysis.
4. **Data Recording:** Collected metrics for all trials—including time, comparisons, and swaps—were written to a structured CSV file located at `data/sort_results.csv` for future reference and reproducibility.

5. **Visualization:** The `plot_results()` function created box plots for each metric, stored as PNG files in the `plots/` directory. These visuals were later embedded in the LaTeX report to illustrate performance distribution and variation.

6. **Statistical Validation:** Independent sample t-tests were conducted to evaluate the statistical significance of the differences in performance metrics between the two algorithms. The hypothesis testing confirmed that the differences were not due to random chance.

7. **Reporting:** All steps, graphs, and findings were compiled into a comprehensive LaTeX-based technical report with annotated bibliography and empirical discussion for academic submission.

This well-defined procedure ensured transparency, reproducibility, and robustness in the experimental results and provided a strong foundation for extending the study to additional algorithms in the future.

12 Conclusion

This project conducted an in-depth empirical analysis of two classic quadratic-time sorting algorithms—selection sort and insertion sort—focusing on real-world performance under diverse input conditions. Through the use of Python-based instrumentation and repeated trials, the experiment measured key metrics such as CPU time, wall clock time, comparison counts, and swap counts. Additionally, statistical significance testing using t-tests validated the observed differences between the two algorithms with high confidence.

The results demonstrated that while selection sort maintains a predictable and stable pattern across all inputs, it lacks adaptability. Insertion sort, by contrast, leverages pre-existing order in the dataset, reducing operation counts and execution time significantly under favorable conditions, such as nearly sorted arrays. However, insertion sort showed degradation in performance on reverse-sorted data, as evidenced by higher variation and outliers in the box plots.

Visualizations using box plots provided a nuanced understanding of distribution and variance, replacing traditional bar charts to better capture execution variability. The analysis revealed consistent advantages of insertion sort in average and best-case scenarios, while selection sort remained useful for its simplicity and predictability in teaching and low-memory contexts.

Overall, this study not only reaffirmed theoretical expectations but also highlighted the importance of empirical profiling and statistical analysis in algorithm evaluation. The modular design and reproducible methodology of this project allow for future extensions to include more advanced sorting algorithms and hardware-aware performance comparisons.

Go back to references back to normal
Disccuisions for software

Annotated Bibliography

Gemci, F. (2016). *A Comparative Study of Selection Sort and Insertion Sort Algorithms. ResearchGate.*
Gemci presents an empirical analysis comparing selection and insertion sort under various data conditions. The paper emphasizes the practical runtime differences despite similar theoretical complexity, supporting algorithm choice in memory-limited environments.
Key Quotation: "Insertion sort shows faster execution in nearly sorted arrays, while selection sort maintains consistent operation counts."
Screen Grab Description: Figure 1—Runtime Comparison Graph (alt-text: bar chart comparing average execution time for 100 runs).
Key-Phrases: quadratic sorting; empirical runtime; memory efficiency
Important Floats: Runtime plot; operation count tables

Reddy, R. G., et al. (2017). *An Empirical Study and Analysis on Sorting Algorithms. IJMET, 8(8), 488–498.*
This article evaluates CPU time, space complexity, and throughput of common sorting algorithms. The paper highlights insertion sort's behavior with partially ordered data, aligning with our observed performance boost during profiling.
Key Quotation: "Algorithms behave significantly differently under structured inputs, which is not captured by their asymptotic notation."
Screen Grab Description: Figure 3—Time vs. Input Order Plot (alt-text: line graph showing faster sort time on nearly sorted arrays).
Key-Phrases: CPU profiling; insertion sort; selection sort
Important Floats: Performance benchmarks; input variation graphs

Ghasemi, E. et al. (2020). *Galloping in Fast-Growth Natural Merge Sorts. arXiv:2012.03996.*
Explores optimizations in merge-based sorting, particularly in Timsort. The galloping mode mechanism is dissected to explain its performance benefits in hybrid sort designs, which contextualizes the contrast with basic quadratic sorts.
Key Quotation: "Galloping enables adaptive merge sorting to skip over monotonic regions, reducing comparison count drastically."
Screen Grab Description: Figure 5—Galloping Performance Chart (alt-text: runtime vs. array monotonicity ratio).
Key-Phrases: Timsort; merge optimization; hybrid sorting
Important Floats: Comparison count vs array pattern

Sedgewick, R., Wayne, K. (2011). *Algorithms (4th ed.). Addison-Wesley.*
This textbook provides implementation patterns and visual analysis for sorting routines. The discussion of quadratic sorts emphasizes their simplicity and role in educational contexts, aligning with our project's goals.
Key Quotation: "Insertion sort has practical advantages on small arrays due to low overhead."
Screen Grab Description: Diagram—Insertion Sort Inner Loop (alt-text: arrows showing in-place element shift).
Key-Phrases: pedagogy; small input performance; in-place sorting
Important Floats: Loop flowcharts; animation snapshots

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley.*
Knuth's foundational text systematically covers the theory and implementation of various sorting algorithms. Volume 3 discusses selection and insertion sort in depth, with an emphasis on operation count and theoretical justification.
Key Quotation: "The average number of comparisons for insertion sort is approximately $n^2/4$ for random input."
Screen Grab Description: Figure 2—Insertion Sort Trace Table (alt-text: grid showing step-by-step comparisons and movements).
Key-Phrases: algorithm analysis; comparison count; sorting theory
Important Floats: Mathematical recurrence tables; sorting state trace

Bentley, J. (1986). *Programming Pearls. Addison-Wesley.*
Bentley illustrates sorting performance through real-world coding problems, advocating for insertion sort in small or partially sorted arrays. The book connects algorithm choice to developer intuition and debugging insights.
Key Quotation: "Simple code often outperforms 'better' algorithms on small datasets due to memory and call overheads."
Screen Grab Description: Box—5-line insertion sort example (alt-text: inline pseudocode snippet).
Key-Phrases: code simplicity; small array performance; practical tips
Important Floats: Code snippets; runtime logs from test cases

McConnell, J. J. (2007). *Analysis of Algorithms. Jones and Bartlett Publishers.*
A detailed discussion of algorithm performance through mathematical modeling and empirical measurements. Offers walkthroughs for instrumenting code to measure real CPU and wall time, validating the methodology used in this project.
Key Quotation: "Measuring time is as important as computing complexity—it connects algorithm theory with hardware reality."
Screen Grab Description: Example—CPU Time Logging Utility (alt-text: function for averaging CPU cycles over trials).
Key-Phrases: profiling; benchmarking; algorithm analysis
Important Floats: Sample profiler output; timing plots

Heineman, G. T., Pollice, G., Selkow, S. (2009). *Algorithms in a Nutshell. O'Reilly Media.*
This reference provides concise algorithm descriptions, annotated pseudocode, and benchmark tables. Includes Java and Python-style code for selection and insertion sort with empirical results, making it practical for comparison projects.
Key Quotation: "Insertion sort is ideal for datasets that are already partially ordered—a common real-world case."
Screen Grab Description: Table—Comparison of Sort Times (alt-text: execution time table for 1k to 10k elements).
Key-Phrases: benchmarking; pseudocode; adaptive sorting
Important Floats: Timing table; adaptive behavior chart

Wikipedia Contributors. (2024). *Sorting Algorithm. Wikipedia.*
This overview article categorizes sorting algorithms by complexity, stability, adaptivity, and memory use. It helped classify insertion and selection sort accurately in the report and referenced Timsort's hybrid design.
Key Quotation: "Adaptive algorithms like Timsort exploit existing order to reduce the number of required comparisons."
Screen Grab Description: Table—Comparison of Sorting Algorithm Properties (alt-text: matrix of stability and complexity for each algorithm).
Key-Phrases: sorting types; classification; hybrid sorting
Important Floats: Comparison matrix; complexity grid

Wikipedia Contributors. (2024). *Timsort. Wikipedia.*
Describes the inner workings of Timsort, including its use of binary insertion sort and merge strategies. Helped explain why Python's built-in sort far outperforms basic quadratic methods, validating our relative timing results.
Key Quotation: "Timsort is a hybrid stable sorting algorithm derived from merge sort and insertion sort."
Screen Grab Description: Figure—Timsort Merge Strategy (alt-text: visual of run stack growth and merge rules).
Key-Phrases: merge sorting; hybrid strategy; Python internals
Important Floats: Run growth chart; algorithm merge flow

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *Introduction to Algorithms (3rd ed.). MIT Press.*
This standard academic text offers formal definitions and time complexity derivations for all major sorting algorithms. It contextualized insertion and selection sort within $\Theta(n^2)$ and contrasted them with $\Theta(n \log n)$ sorts.
Key Quotation: "Insertion sort runs in linear time on nearly sorted inputs—its best-case complexity is O(n)."
Screen Grab Description: Plot—Best vs. Worst Case Comparison (alt-text: two lines showing n vs. n^2 growth).
Key-Phrases: theoretical bounds; worst-case vs best-case; textbook sorting