Presentation Title: Sorting Algorithm Performance Comparison: Insertion Sort vs Selection Sort

---

Presenter Notes for Slide 1
Welcome everyone. Today's presentation dives into a detailed performance analysis of two fundamental sorting algorithms—insertion sort and selection sort. This analysis is part of my final project for CSCI 681, Spring 2024. We'll evaluate not only their runtime behavior but also statistical and profiling-based insights. I'll show how these algorithms behave under different inputs and compare them to Python's built-in sorted() function to establish a performance benchmark.

Presenter Notes for Slide 2
Sorting algorithms are foundational in computer science and widely used across various systems, from databases to UI rendering. Insertion Sort and Selection Sort are among the simplest to implement, yet they reveal meaningful differences in behavior under various conditions. This project focuses exclusively on these two to allow a detailed comparison. The study includes performance metrics like wall clock and CPU time, swap and comparison count, profiling data, and statistical tests to determine significance.

Presenter Notes for Slide 3
To make our comparison fair and informative, we generated arrays of size 100 under multiple conditions: random, sorted, reversed, and arrays with duplicates. We tested both integers and floating-point numbers to observe sensitivity to data types. Python was chosen for rapid implementation and analysis, and we used standard libraries like time, resource, and cProfile. Each configuration was run 100 times to gather a robust dataset that would allow for statistical analysis using scipy.stats.

Presenter Notes for Slide 4
Our goal was to measure both algorithmic efficiency and system-level resource usage. We recorded:

- Wall clock time, to capture real elapsed time
- CPU time, to isolate processor activity
- The number of swaps and comparisons, which provide insight into algorithmic cost
- Profiling data to analyze which parts of the algorithm were most expensive
  Together, these metrics offer a well-rounded view of how each algorithm performs.

Presenter Notes for Slide 5
Beyond raw performance, we tested variations with and without Python's optimization flags to examine interpreter-level effects. We used Python’s built-in sorted() as a gold standard for performance benchmarking. Input sensitivity testing involved mixing input orders and data types. This allowed us to understand not just average behavior, but edge-case performance and stability, making our study more thorough and realistic.

Presenter Notes for Slide 6
Now let’s look at the runtime performance. Insertion Sort outperformed Selection Sort on partially sorted and random data. This is expected—Insertion Sort benefits from order because it minimizes shifts. Selection Sort, in contrast, scans the entire array repeatedly, leading to a constant cost regardless of initial order. Wall clock and CPU times showed similar trends, although Python’s garbage collection introduced minor variations across trials.

Presenter Notes for Slide 7
Swap and comparison counts offer insight into algorithmic behavior. Selection Sort performs n2n^2 comparisons and up to nn swaps. Insertion Sort, on the other hand, reduces swaps when the data is partially sorted. This makes it especially suitable for nearly ordered datasets. The data confirmed this: Insertion Sort's average swap count was significantly lower for sorted inputs, aligning with its theoretical efficiency.

Presenter Notes for Slide 8
Let’s examine the statistical central tendencies. The mean, median, and range were computed across all configurations. These statistics help highlight trends—such as how Insertion Sort had a lower median runtime on sorted data, while Selection Sort maintained a consistent, albeit slower, median across all input types. The range values showed that Selection Sort was more stable but also consistently slow, while Insertion Sort had more variance but potential for much faster performance.

Presenter Notes for Slide 9
Here are our box plots, which were recommended by the Statistical Consulting Center. These plots better visualize the data spread and outliers. Insertion Sort shows tighter boxes and fewer outliers in partially sorted cases, while Selection Sort’s boxes are wider but stable. This visual clarity is important: bar graphs alone don’t reveal how often extreme values appear, which can skew the mean.

Presenter Notes for Slide 10
To back our observations with statistical rigor, we performed independent t-tests on the runtime

and swap/comparison counts. We found that the p-values were below the 0.05 threshold for most configurations, indicating significant differences between the two algorithms. This confirms that the observed improvements with Insertion Sort aren't just random variation—they're statistically meaningful.

Presenter Notes for Slide 11
Using Python's cProfile, we profiled each algorithm to identify where time was spent. Selection Sort spent most time in swapping routines, while Insertion Sort spent more in nested comparisons. This profiling confirmed our assumptions about algorithmic cost. The profiler also showed that function call overhead was non-trivial in Python, adding about 3–5% overhead, especially for float operations.

Presenter Notes for Slide 12
Next, we benchmarked against Python's built-in sorted() function, which uses Timsort—a hybrid of merge sort and insertion sort. As expected, it significantly outperformed both custom algorithms, often by 10x. This highlights that while academic algorithms are good learning tools, real-world systems demand optimized, hybrid, and compiled approaches for performance-critical tasks.

Presenter Notes for Slide 13
When comparing results under different optimization settings, Python's interpreter flags had negligible impact. Even with PYTHONOPTIMIZE, the runtime differences were under 1%. This shows that most Python optimization occurs at the implementation level, not through runtime flags. In contrast, a C implementation might show much more noticeable gains under compiler optimization.

Presenter Notes for Slide 14
One interesting observation was data type sensitivity. Sorting float arrays took consistently longer than int arrays. This is due to higher computation cost for floating point comparisons and potential caching differences. It emphasizes that not only input order but also data representation affects performance—something that needs to be considered in real-world applications where precision is critical.

Presenter Notes for Slide 15
To conclude: Insertion Sort is better suited for datasets that are already partially sorted. It minimizes swaps and executes faster under such conditions. Selection Sort is simpler and more predictable, but slower. Understanding these trade-offs helps developers choose the right tool for

the job. When speed matters, built-in or hybrid sorts are preferred. However, knowing the mechanics of basic algorithms provides valuable insight into how and why optimizations work.

Presenter Notes for Slide 16
Our project had a few limitations: Python is an interpreted language, so results include interpreter overhead. We used small datasets due to time constraints. Also, concurrency and parallelism weren't tested. For future work, we propose:

- Implementing Merge and Quick Sort for a broader comparison
- Porting the code to C or Java for better optimization
- Adding memory profiling and multithreaded benchmarks
  These improvements would yield deeper and more scalable insights.

Presenter Notes for Slide 17
Here are the references that guided this study. Timsort's documentation helped us understand the built-in benchmark. We also relied on statistical references for t-testing and Python documentation for profiling tools. All analysis and plots were generated using Python tools.

Presenter Notes for Slide 18
That concludes the presentation. Thank you for your attention. I hope this comparative analysis offered useful insights into sorting performance. I now welcome any questions or feedback. I'd be happy to hear suggestions for extending or refining this project further.