Cross Joint Summary: Insertion Sort vs. Selection Sort

1. Introduction

This report provides a joint comparative analysis of two elementary quadratic sorting algorithms — **Insertion Sort (with Binary Search optimization)** and **Selection Sort (with Early Termination optimization)** — implemented and benchmarked by two students as part of the *Design and Analysis of Algorithms* course.

The shared objective is to evaluate their **theoretical properties**, **empirical performance**, and **code quality**, highlighting both similarities and differences. Both algorithms are simple, in-place, and stable, but differ fundamentally in mechanism:

- Insertion Sort incrementally builds a sorted prefix.
- Selection Sort repeatedly extracts the minimum from the suffix.

2. Algorithm Overview

Insertion Sort Implementation

- **Core steps**: for each element, binary search locates its correct position, elements are shifted, and the key is inserted.
- **Optimizations**:
  - Early sortedness detection.
  - Binary search reduces comparisons.
  - Cached reads for precision in metrics.
- **Strengths**: stable, efficient for nearly sorted inputs, low memory footprint.
- **Dominant cost**: moves (shifts/writes).

Selection Sort Implementation

- **Core steps**: for each pass, find the minimum element in the suffix and swap it into position.
- **Optimizations**:
  - Early termination if suffix is sorted.
  - Caching repeated reads to reduce overhead.
- **Strengths**: predictable swaps, effective on sorted/nearly sorted inputs.
- **Dominant cost**: comparisons.

3. Theoretical Complexity Comparison

Both algorithms share quadratic asymptotics in the average and worst cases but differ in dominant operations.

Table 1. Asymptotic Complexity Comparison

Operation	Insertion Sort	Selection Sort
Best-case time	$\Omega(n)$ — sorted input (no shifts)	$\Omega(n)$ — sorted input detected early
Average-case time	$\Theta(n^2)$ — shifts dominate	$\Theta(n^2)$ — comparisons dominate
Worst-case time	$O(n^2)$ — reversed input, max shifts	$O(n^2)$ — reversed input, max comparisons
Comparisons	$\Theta(n \log n)$ (with binary search)	$n(n-1)/2 = \Theta(n^2)$
Moves (writes)	Up to $n^2/2$ (shifting dominates)	Always $n-1$ swaps ($\Theta(n)$)
Space	$O(1)$ (in-place)	$O(1)$ (in-place)
Stability	Stable	Stable
Adaptivity	Effective on nearly sorted inputs	Effective on sorted/nearly sorted inputs

4. Empirical Performance Results

Benchmarks were conducted on arrays of size n = 100, 1k, 10k, 100k under four input distributions (sorted, random, reversed, nearly sorted). Metrics include runtime (ns), comparisons, moves, reads, and writes.

Table 2. Example Runtime & Metrics (n = 10,000)

Distribution	Insertion Sort (optimized)	Selection Sort (optimized)
Sorted	~5,000 ns; comps ≈ 9,999; moves = 0	~50,000 ns; comps ≈ 9,999; moves = 0
Random	~21.0×10^6 ns; comps ≈ 118k; moves ≈ 25.0×10^6	~20.6×10^6 ns; comps ≈ 119k; moves ≈ 29.9k
Reversed	~31.0×10^6 ns; comps ≈ 113k; moves ≈ 50.0×10^6	~31.0×10^6 ns; comps ≈ 113k; moves ≈ 5.0×10^6

Distribution	Insertion Sort (optimized)	Selection Sort (optimized)
Nearly sorted	~4.25×10^6 ns; comps ≈ 119k; moves ≈ 5.9×10^6	~4.25×10^6 ns; comps ≈ 119k; moves ≈ 5.9×10^6

Observations:

- **Sorted input**: Both achieve Ω(n). Selection Sort benefits dramatically from early termination (~23,000× speedup at n=100k).
- **Random input**: Both confirm $\Theta(n^2)$. Insertion Sort's binary search reduces comparisons by 5–10%, but moves dominate.
- **Reversed input**: Both degrade to $O(n^2)$. Insertion Sort suffers from excessive shifts; Selection Sort suffers from maximum comparisons.
- **Nearly sorted input**: Both improve significantly. Insertion Sort reduces shifts, Selection Sort terminates earlier.

Table 3. Comparative Performance (n = 100,000, random input)

Metric	Insertion Sort	Selection Sort
Time (ns)	~1.55×10^9	~1.49×10^9
Comparisons	~1.52×10^6	~1.52×10^6
Moves/Writes	~2.49×10^9	~2.99×10^4
Reads	~2.49×10^9	~1.01×10^6
Allocations	0 (in-place)	0 (in-place)

Interpretation:

- Insertion Sort's **time correlates with moves**, which reach billions for large n.
- Selection Sort's **time correlates with comparisons**, but swap count remains linear.
- Both implementations align closely with theoretical predictions.

5. Code Quality and Maintainability

Table 4. Code Quality Comparison

Criterion	Insertion Sort	Selection Sort
Readability	Clear modular code; binary search helper	Straightforward min-search logic
Performance tracking	Detailed but some undercount of reads	Captures swaps, reads/writes; minor gaps
Optimization coverage	Binary search, early exit, read caching	Early termination, read caching
Maintainability	Modular, easy to extend with hybrids	Simple, predictable, easy to instrument
Benchmark readiness	CLI tested, JMH recommended	CLI tested, JMH recommended

6. Joint Conclusion

Both **Insertion Sort** and **Selection Sort** are correct, stable, in-place algorithms that validate their theoretical complexity ($\Omega(n)$, $\Theta(n^2)$, $O(n^2)$) through detailed empirical benchmarking.

- **Insertion Sort** emphasizes adaptivity: efficient on nearly sorted data, reduced comparisons via binary search, but dominated by shifts.
- **Selection Sort** emphasizes predictability: fixed swaps, early termination optimization gives dramatic best-case improvement.

Comparative Insights

- Insertion Sort → Better for small or partially ordered datasets.
- Selection Sort → Better when inputs are already sorted or nearly sorted.
- Both remain quadratic and impractical for very large n compared to $O(n \log n)$ algorithms.

Recommendations

1. Extend experiments to hybrid algorithms (e.g., MergeSort with Insertion Sort cutoff).
2. Standardize metric tracking for more precise comparisons.
3. Add JMH microbenchmarks to measure JVM-level constant factors.
4. Visualize results with plots (time vs n^2, time vs n for sorted inputs).