ASSIGNMENT 2

SE 2409 - A.Shynggys, K.Mukhammedali

Github: https://github.com/mxkwnz/DAA---Assignment-2.git

EXECUTIVE SUMMARY

This report compares two fundamental quadratic sorting algorithms implemented as part of our peer code review assignment. Both algorithms are correctly implemented with thoughtful optimizations and demonstrate strong understanding of algorithmic principles.

2. ALGORITHM DESCRIPTIONS

SELECTION SORT (Shynggys)

Approach: Find minimum, swap to front, repeat Optimization: Early termination on sorted data

Process:

1. Find minimum element in unsorted portion
2. Swap with first unsorted position
3. Move boundary one step right
4. Repeat until sorted

Characteristics:

- Predictable: Always n(n-1)/2 comparisons
- Minimal swaps: At most n swaps
- Not adaptive: Same performance on all inputs
- Not stable: May change order of equal elements

INSERTION SORT (Kalen)

Approach: Insert each element into sorted portion Optimization: Binary search for insertion position

Process:

1. Take next element from unsorted portion
2. Find correct position in sorted portion (binary search)
3. Shift larger elements right
4. Insert element at correct position
5. Repeat until sorted

Characteristics:

- Adaptive: Faster on nearly-sorted data
- Variable performance: O(n) to O(n²)
- Stable: Preserves order of equal elements
- Online: Can sort as data arrives

3. COMPLEXITY COMPARISON

Metric	Selection Sort (Shynggys)	Insertion Sort (Kalen)
Best Case Time	$\Theta(n^2)$	$\Theta(n)$ ★ WINNER
Average Time	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case Time	$\Theta(n^2)$	$\Theta(n^2)$
Space	$\Theta(1)$	$\Theta(1)$
Comparisons	Always $n^2/2$	n-1 to $n^2/2$
Swaps (Best)	0	0
Swaps (Worst)	n-1 ★ WINNER	$n^2/2$
Stable	No	Yes ★ WINNER
Adaptive	No	Yes ★ WINNER
Predictable	Yes ★ WINNER	No

4. EMPIRICAL PERFORMANCE RESULTS

RANDOM DATA (n=10,000)

Selection Sort: 285 ms ← WINNER Insertion Sort: 385 ms (35% slower)

Reason: Selection Sort has fewer total operations Winner: SELECTION SORT

SORTED DATA (n=10,000)

Selection Sort: 280 ms Insertion Sort: 0.91 ms ← WINNER (300× faster!)

Reason: Insertion Sort achieves O(n) on sorted data

Winner: INSERTION SORT (DOMINANT)

REVERSED DATA (n=10,000)

Selection Sort: 290 ms ← WINNER Insertion Sort: 758 ms (2.6× slower)

Reason: Insertion Sort suffers from excessive shifts Winner: SELECTION SORT

NEARLY-SORTED DATA (n=10,000, 95% sorted)

Selection Sort: 282 ms Insertion Sort: 35 ms ← WINNER (8× faster)

Reason: Insertion Sort adapts to existing order Winner: INSERTION SORT

5. DETAILED PERFORMANCE COMPARISON

FULL BENCHMARK RESULTS

Размер массива	Алгоритм	Random	Sorted	Reversed	Nearly Sorted
100	Selection	0.22 ms	0.21 ms	0.23 ms	0.21 ms
	Insertion	0.32 ms	0.01 ms	0.58 ms	0.04 ms
1,000	Selection	3.5 ms	3.4 ms	3.6 ms	3.4 ms
	Insertion	4.8 ms	0.09 ms	9.2 ms	0.42 ms
10,000	Selection	285 ms	280 ms	290 ms	282 ms
	Insertion	385 ms	0.91 ms	758 ms	35 ms

KEY OBSERVATIONS

1. CONSISTENCY vs ADAPTIVITY • Selection Sort: ±3% variation across all inputs • Insertion Sort: 800× variation (0.91ms to 758ms)
2. SORTED DATA PERFORMANCE • Insertion Sort absolutely dominates (300-400× faster) • This is the most significant difference
3. RANDOM DATA • Selection Sort slightly faster (15-25%) • Both are similarly inefficient at $O(n^2)$
4. REVERSED DATA • Selection Sort significantly faster (2.6×) • Insertion Sort's weakness clearly visible
5. SCALABILITY • Both scale quadratically: 10× size → ~100× time • Confirms $O(n^2)$ complexity for both
6. OPTIMIZATION EFFECTIVENESS

SHYNGGYS'S EARLY TERMINATION (Selection Sort)

Impact on sorted data (n=1,000):

- Without optimization: 3.5 ms
- With optimization: 3.4 ms (3% faster)
- Expected on larger sorted arrays: Much better

Effectiveness: ★★★☆☆

- Works well on sorted data
- Adds overhead for other cases
- Could be simplified to reduce overhead

KALEN'S BINARY SEARCH (Insertion Sort)

Impact on random data (n=1,000):

- Without optimization: 6.8 ms
- With optimization: 4.8 ms (29% faster!)
- Consistent across all input sizes

Effectiveness: ★★★★☆

- Significant 29% speedup
- Doesn't change O(n²) complexity
- Well-implemented and tested

7. SWAP/SHIFT ANALYSIS

Critical difference in memory operations:

Algorithm	Operations	Type
Selection	9,999	Swaps (3 moves each)
Insertion	49,995,000	Shifts (1 move each)

Difference: 5,000× more operations in Insertion Sort!

This explains why Selection Sort wins on reversed data despite both being O(n²).

8. **STRENGTHS & WEAKNESSES SUMMARY**

SELECTION SORT (Shynggys)

Strengths:

 Predictable performance (always O(n²))

 Minimal swaps - only O(n)

 Better on random data (15-25% faster)

 Much better on reversed data (2.6× faster)

 Good when swapping is expensive

 Simple and easy to understand

Weaknesses:

✗ Cannot adapt to sorted data (without optimization)

✗ Always performs n(n-1)/2 comparisons

✗ Not stable

✗ Poor on sorted/nearly-sorted data

INSERTION SORT (Kalen)

Strengths:

 Adaptive - O(n) on sorted data

 Excellent on sorted/nearly-sorted (300× faster!)

 Stable sorting

 Online algorithm

 Binary search optimization (29% faster)

 Versatile for real-world data

Weaknesses:

✗ O(n²) shifts in worst case

✗ Poor on reversed data (2.6× slower)

✗ Slower on random data (25% slower)

✗ Unpredictable performance

9. USAGE RECOMMENDATIONS

USE SELECTION SORT WHEN:

 Swapping is expensive (large objects) Example: Sorting 1MB database records

 Need predictable, consistent performance Example: Real-time systems with timing constraints

 Data is random or reversed Example: Completely unsorted data

 Memory writes are costly Example: Flash memory or SSDs with limited write cycles

 Want simple, easy-to-understand code Example: Teaching or embedded systems

Real-world scenario: Sorting employee records (large objects) where each swap involves moving hundreds of bytes. Selection Sort's O(n) swaps make it ideal.

USE INSERTION SORT WHEN:

 Data is already sorted or nearly-sorted Example: Adding items to a sorted list

 Need stable sorting (preserve order of equal elements) Example: Sorting by multiple criteria

 Online sorting (data arrives in real-time) Example: Sorting streaming data

 Small datasets (n < 100) Example: Subroutine in Timsort

 Unknown input distribution (might be sorted) Example: User input that's often partially sorted

Real-world scenario: Maintaining a sorted list of recent messages as new messages arrive. Insertion Sort handles new additions efficiently.

USE NEITHER WHEN:

✗ Large datasets (n > 10,000) ✗ Performance is critical

✗ Production systems

Better alternatives:

- Quick Sort: O(n log n) average, fastest in practice
- Merge Sort: O(n log n) guaranteed, stable
- Heap Sort: O(n log n) guaranteed, in-place
- Timsort: Hybrid, excellent on real-world data (Python default)

10. DECISION MATRIX

WHEN TO USE WHICH ALGORITHM		
Data Type / Scenario	**Winner**	**Margin**
Sorted	INSERTION ★★★★★	~300× faster
Nearly-Sorted	INSERTION ★★★★	~8× faster
Random	SELECTION ★★	~25% faster
Reversed	SELECTION ★★★	~2.6× faster
Unknown	INSERTION ★★★	More versatile
Need Stability	INSERTION ★★★★★	Only option
Expensive Swaps	SELECTION ★★★★★	~5000× fewer swaps
Predictability	SELECTION ★★★★	Always same behavior
Online Sorting	INSERTION ★★★★★	Only option
Small Arrays	INSERTION ★★★	Slightly better

11. LESSONS LEARNED

FROM IMPLEMENTATION

1. Asymptotic vs Practical Performance • Both are O(n²) but behave very differently • Constant factors and input patterns matter greatly
2. Optimization Trade-offs • Binary search improves constants, not complexity • Early termination can add overhead
3. No Universal "Best" Algorithm • Choice depends on data characteristics • Need to understand the problem domain
4. Empirical Testing is Critical • Our measurements matched theory perfectly • Real-world testing reveals practical differences

FROM PEER REVIEW

1. Code Quality Matters • Both implementations were clean and testable • Good documentation aids understanding
2. Different Approaches, Similar Complexity • Selection: Find and place • Insertion: Take and insert • Both end up at O(n²)
3. Optimization Strategies Differ • Selection: Early exit on sorted • Insertion: Binary search for position • Each targets their algorithm's characteristics

CONCLUSION

Both implementations demonstrate strong algorithmic understanding and professional code quality. Each algorithm has distinct strengths:

OVERALL WINNER: Depends on use case

For general-purpose sorting: INSERTION SORT

- More versatile due to adaptive behavior
- Excellent on common patterns (sorted/nearly-sorted)
- Stable sorting is often required

For specific scenarios: SELECTION SORT

- Better when swaps are expensive
- More predictable timing
- Better on random/reversed data

For production use: NEITHER

- Use O(n log n) algorithms for n > 10,000
- Modern languages provide optimized implementations