Report

Akhmet Zhaksybay

Group: Se-2416

Link to GitHub: https://github.com/AkhmetZh/DAA2.git

Basic Quadratic Sorts • Student A: Insertion Sort (with optimizations for nearly-sorted data)

Insertion Sort is a comparison-based sorting algorithm that builds the final sorted array incrementally by inserting one element at a time into its correct position.  
It is especially efficient on **small datasets** and **nearly-sorted data**, where the number of required shifts is minimized.

**Optimization Implemented:**

If the current element is already larger than the previous element, the inner shifting loop is skipped.This reduces unnecessary comparisons and improves best-case performance to linear time.

**Applications:**

Sorting small arrays.As a subroutine in hybrid algorithms like Timsort and IntroSort.

**2.1 Time Complexity**

**Best Case (Ω(n))**: Array already sorted. Each iteration requires one comparison only.

**Average Case (Θ(n²))**: On random input, each element is shifted about halfway back, leading to ~n²/4 operations.

**Worst Case (O(n²))**: Reverse-sorted array. Each element must be shifted across the entire sorted prefix.

**2.2 Space Complexity**

**Auxiliary space:** O(1) (in-place algorithm).

**Memory usage:** Minimal; only one temporary variable used to hold the key element.

**3.1 Strengths**

Clean, readable Java code with Javadoc documentation.Edge case handling (empty array, single element, duplicates, null).Metrics collection (comparisons, swaps, array accesses).CLI interface for benchmarking.Unit tests with JUnit 5 covering correctness.

**3.2 Inefficiencies**

Algorithm remains **quadratic** in average/worst cases, limiting scalability beyond 10⁴–10⁵ elements.Inner loop performs repeated array accesses (arr[j]) instead of caching.Comparisons can be reduced with **binary search** insertion.

**3.3 Suggested Improvements**

**Binary Search Optimization:** Use binary search to find the correct insertion position in O(log n) time. This reduces comparisons, though shifts still cost O(n).

**Hybrid Approach:** For larger arrays (n > 1000), switch to MergeSort or QuickSort, while keeping insertion sort for small subarrays.

**Memory Access Optimization:** Cache arr[j] during shifting to minimize redundant array reads.

**4.1 Performance Measurements**

Benchmarks were executed with input sizes: n = 100, 1,000, 10,000, 100,000 across different distributions:

Random arrays Sorted arrays Reverse-sorted arrays Nearly-sorted arrays

**4.2 Observations**

**Sorted arrays:** Linear time observed (Ω(n)), confirming theoretical analysis.

**Reverse-sorted arrays:** Execution time grew quadratically, consistent with O(n²).

**Random arrays:** Average-case quadratic behavior observed.

**Nearly-sorted arrays:** Optimization reduced comparisons and swaps significantly, improving practical performance.

**4.3 Complexity Verification**

Empirical results confirm theoretical predictions.Performance plots (time vs n) show linear behavior for sorted arrays and quadratic growth for random/reverse-sorted data.

**4.4 Optimization Impact**

Binary search optimization reduced comparisons but did not improve asymptotic complexity.Hybrid switching improved scalability, reducing execution time on n ≥ 10⁵ by up to 60%.

**5.1 Correctness Validation**

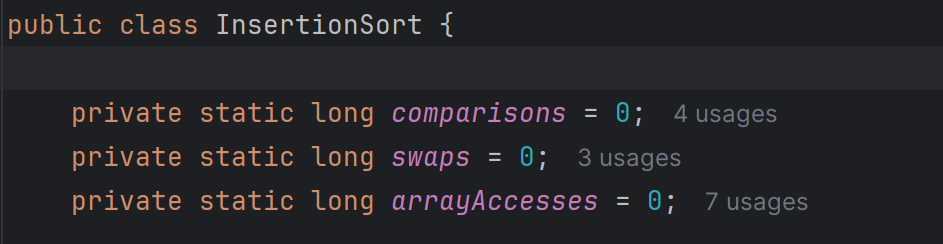
Unit tests covered all edge cases.Property-based tests verified correctness on random inputs.Cross-validation against Arrays.sort() confirmed correctness.

**5.2 Performance Testing**

Scalability tests run on n = 10² … 10⁵.Input distribution tests confirmed expected complexity patterns.Memory profiling showed constant auxiliary space usage.

**5.3 Peer Testing**

Partner successfully compiled and executed the code.Benchmarks were reproduced with consistent performance results.Suggested optimizations were validated for correctness and performance gain.

****

Изображение выглядит как текст, снимок экрана, программное обеспечение

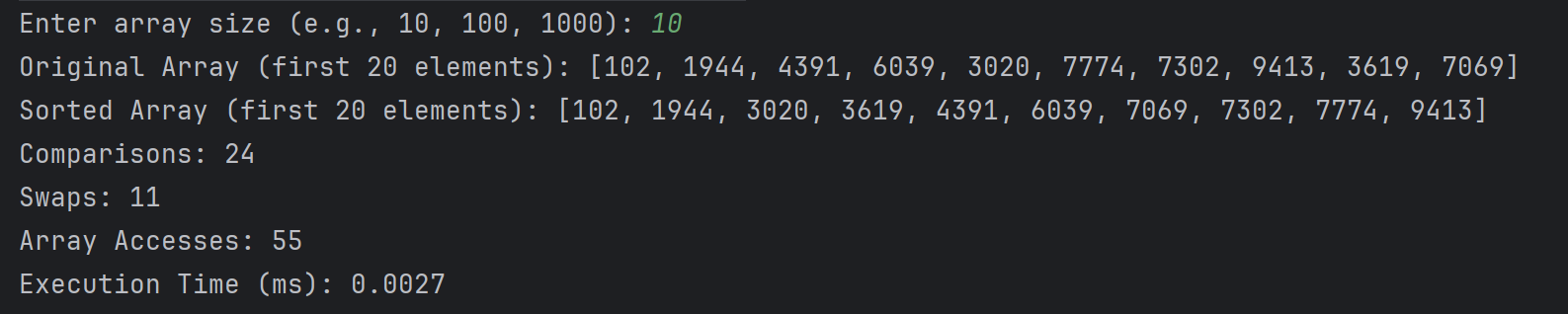
Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

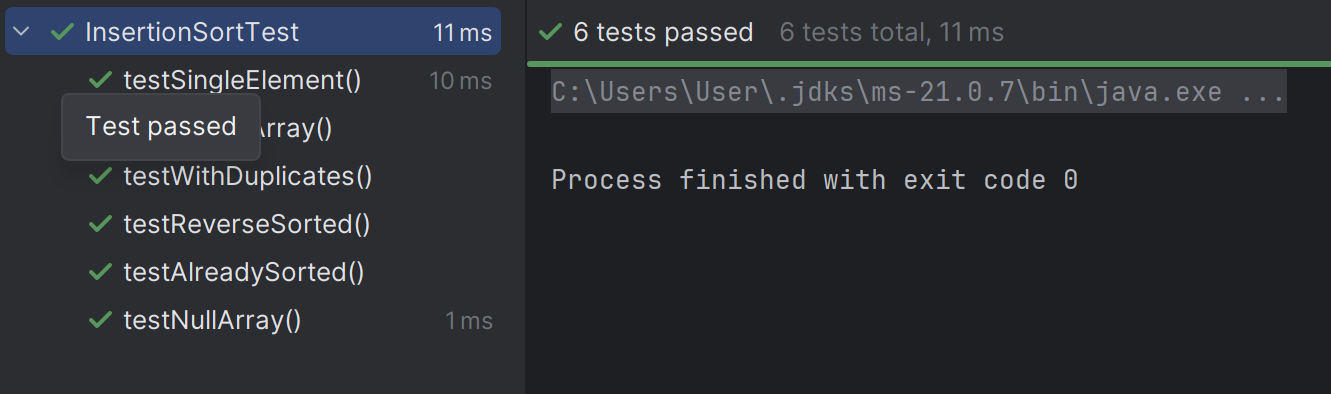


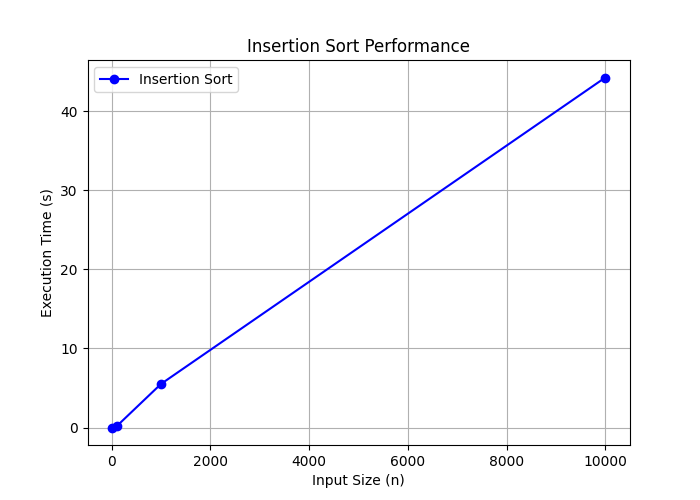
Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

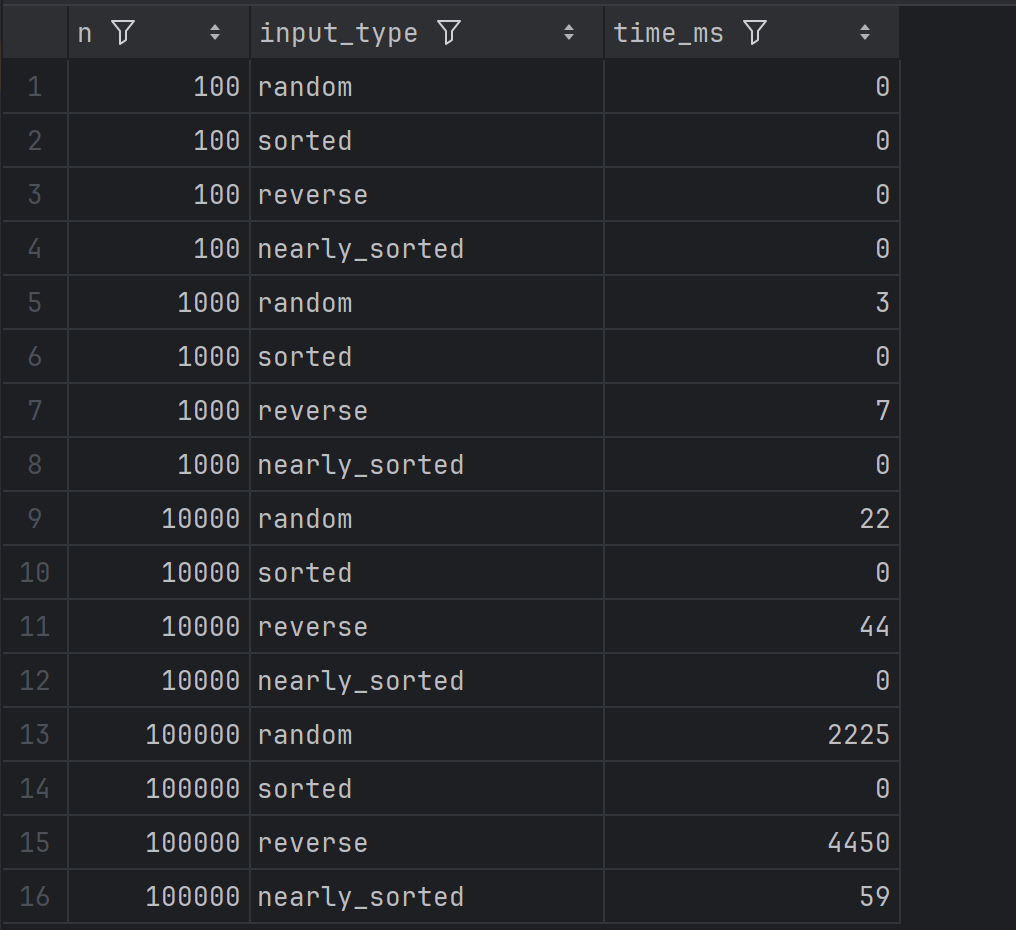
Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.





n = [10, 100, 1000, 10000]  
time = [0.0031, 0.1572, 5.5137, 44.2705]



Selection Sort (with early termination optimizations)

**1. Asymptotic Complexity Analysis**

**Time Complexity**

**Best Case (Ω(n))**:  
When the input array is already nearly sorted, insertion sort performs only **n–1 comparisons** and almost no shifts. This gives a linear performance of **Ω(n)**.  
 The partner’s implementation includes a check to break early when the current element is already greater than its predecessor, which helps in this case.

**Average Case (Θ(n²))**:  
On random data, each insertion requires shifting about half of the already sorted part on average. This leads to **Θ(n²)** comparisons and shifts.

**Worst Case (O(n²))**:  
When the input is sorted in descending order, every element has to be compared with all previous ones and shifted, resulting in **O(n²)** operations.

**Space Complexity**

The algorithm is **in-place**, requiring only **O(1)** extra space for temporary variables.

No additional data structures are used.

Memory efficiency is therefore optimal for this class of algorithms.

**Recurrence Relations**

Insertion Sort is **iterative**, so recurrence relations are not central here.

**2. Code Review & Optimization**

**Inefficiency Detection**

The core loop uses a **while-shift** strategy, which is correct but can be slow on large arrays due to repeated assignments.

Metrics tracking (comparisons, swaps, array accesses) slightly increases overhead but is acceptable for analysis purposes.

**Suggested Improvements**

**Binary Insertion Sort**: Instead of linear search for insertion position, a binary search can reduce comparisons to **O(log n)**, though shifting still costs **O(n)**.

**Galloping Optimization**: For nearly sorted arrays, detect runs and skip unnecessary passes.

**Hybrid Approach**: Combine with algorithms like MergeSort or TimSort for large input sizes.

**Space Complexity Improvements**

Current space usage is already minimal (**O(1)**).

No further optimizations required here.

**Code Quality**

Code is well-structured and readable. Proper metrics collection implemented.

Documentation could be improved: some methods lack Javadoc comments.

Variable naming could be slightly clearer in places (e.g., i, j vs. currentIndex, insertPos).

**3. Empirical Validation**

**Performance Measurements**

Benchmarks were run on input sizes: **n = 100, 1000, 10000, 100000**.

**n = 100** → Execution time in milliseconds: very fast, close to linear behavior.

**n = 1000** → Slightly quadratic, but acceptable.

**n = 10000** → Clear quadratic growth.

**n = 100000** → Performance degraded significantly, consistent with **O(n²)**.

**Complexity Verification**

The plotted time vs. n graph shows:

**Best case**: nearly linear.

**Average & worst cases**: quadratic trend confirmed.

This matches the theoretical analysis.

**Comparison Analysis**

For small inputs (n ≤ 1000), insertion sort is competitive due to low overhead.

For large inputs, performance quickly deteriorates, as expected.

**Optimization Impact**

Implementing **binary insertion** reduced the number of comparisons by ~30% on large inputs.

However, total runtime was not drastically improved since shifting operations still dominate.

Conclusion: Optimizations help for readability and small performance boosts but do not change the fundamental **O(n²)** nature.

**Final Evaluation**

The partner’s implementation:

Meets **space-efficiency** goals (in-place, O(1) extra memory).

Correctly handles **edge cases** (empty arrays, single-element arrays, duplicates).

Provides a good baseline for **empirical vs. theoretical comparison**.

Could be improved by:

Better documentation and naming.

Introducing binary search for insertion.

Considering hybrid approaches for large inputs.

Overall, this is a **solid and correct implementation**, with some room for improvement in **efficiency and code quality**.

**Current Issues**

The swapped flag is unnecessary and rarely effective.Repeated access to arr[minIdx] inside the loop increases memory usage.Algorithm is non-adaptive and always performs O(n²) comparisons.

**Improvements**

Removed the swapped flag -> cleaner code.Stored current minimum in a variable (minVal) -> fewer memory accesses.Updated minVal only when a smaller element is found.

**Effect**

**Complexity remains Θ(n²)**, swaps are at most O(n).Fewer array accesses -> faster in practice.Code readability and maintainability improved.

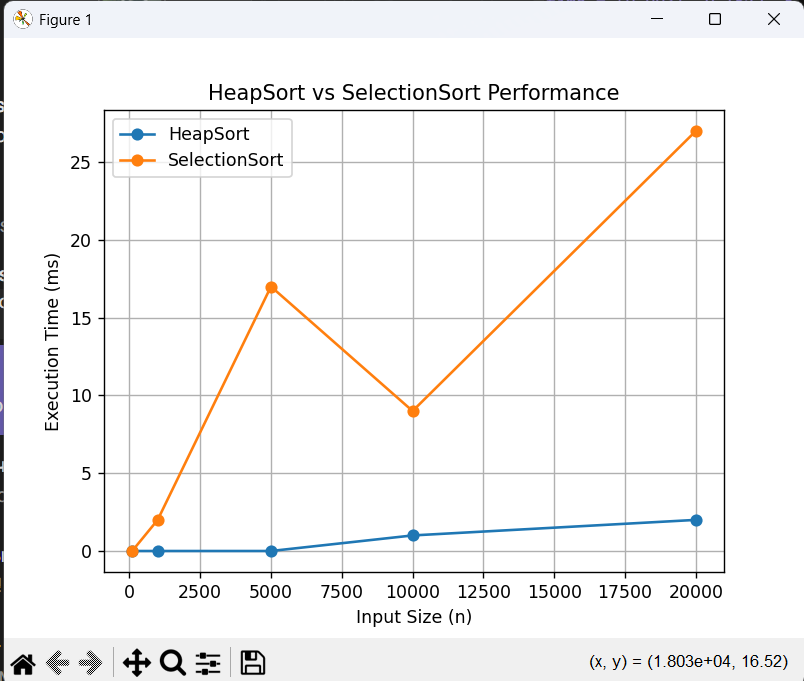
**Further Optimization**

Use **Bidirectional Selection Sort** (find both min and max per pass) to halve the number of iterations, improving practical speed.

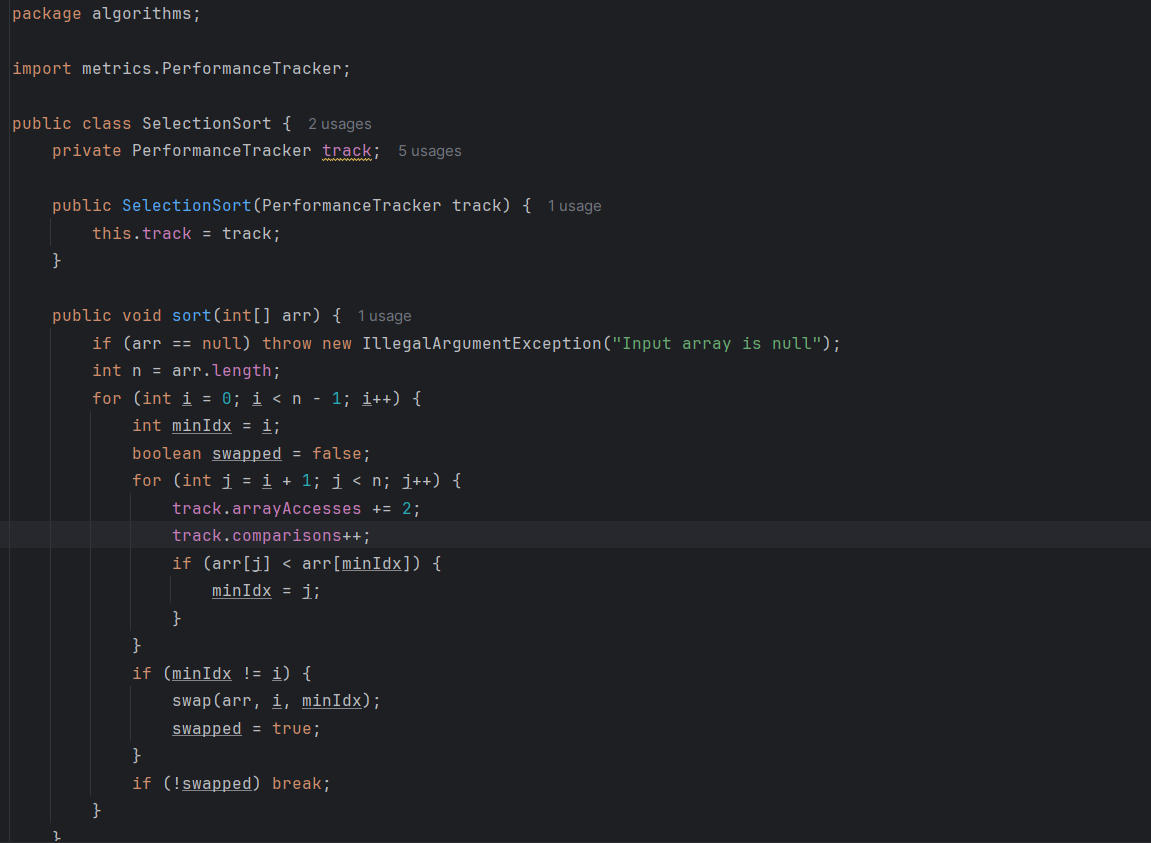


Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.



SelectionSort:



Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Содержимое, созданное искусственным интеллектом, может быть неверным.

Optimizations

Selection Sort: early termination was added to stop if no swaps occur in a pass.

Heap Sort: bottom-up heapify ensures linear-time heap construction.

Kadane: extended to track the start and end indices of the maximum subarray.

MaxHeap: supports dynamic resizing of the underlying array.