**Peer Analysis Report — Atalykov Sultan’s Implementation**

1. **Algorithm Overview**

The implementation includes four main algorithms: SelectionSort, HeapSort, Kadane’s Maximum Subarray Sum, and MaxHeap construction.

* **SelectionSort** iteratively selects the minimum element from the unsorted portion of the array and places it at the correct position.
* **HeapSort** builds a max-heap and repeatedly extracts the maximum to sort the array.
* **Kadane’s Algorithm** computes the maximum subarray sum in linear time using a dynamic programming approach.
* **MaxHeap** constructs a heap from an array using the standard heapify procedure.

The code is structured to allow manual testing, benchmark execution with configurable input sizes, and unit testing via JUnit. Utility functions handle array printing and CSV export for performance analysis.

1. **Complexity Analysis**

**Time Complexity**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| SelectionSort | Θ(n²) | Θ(n²) | Θ(n²) |
| HeapSort | Θ(n log n) | Θ(n log n) | Θ(n log n) |
| Kadane | Θ(n) | Θ(n) | Θ(n) |
| MaxHeap Build | Θ(n) | Θ(n) | Θ(n) |

* **SelectionSort:** always performs comparisons, so Θ(n²) for all cases.
* **HeapSort:** heapify and repeated extraction take O(n log n) consistently.
* **Kadane:** single pass over the array, O(n).
* **MaxHeap:** bottom-up heap construction takes linear time.

**Space Complexity**

* **SelectionSort:** O(1) auxiliary space, in-place.
* **HeapSort:** O(1) auxiliary, in-place sorting.
* **Kadane:** O(1) auxiliary variables.
* **MaxHeap:** O(n) for heap storage; could be in-place if array is reused.

1. **Code Review & Optimization**

**Inefficiencies & Observations**

* SelectionSort is inherently slow for large arrays due to quadratic time complexity.
* HeapSort and MaxHeap implementation is efficient; however, repeated array copying could be reduced.
* Kadane’s algorithm is optimal; no improvements needed.
* Benchmarks generate fresh arrays each run, which is necessary for consistent measurements but adds slight memory overhead.

**Suggested Optimizations**

* For SelectionSort, limit benchmark sizes or replace with insertion sort for nearly sorted arrays to reduce constant factors.
* Reuse arrays for benchmarking rather than copying repeatedly to save memory allocations.
* Apply in-place heap construction if original array can be overwritten to reduce auxiliary storage.
* Minor code readability improvements: better naming conventions for array copies and loop indices.

1. **Empirical Results**

**Benchmarks Conducted**

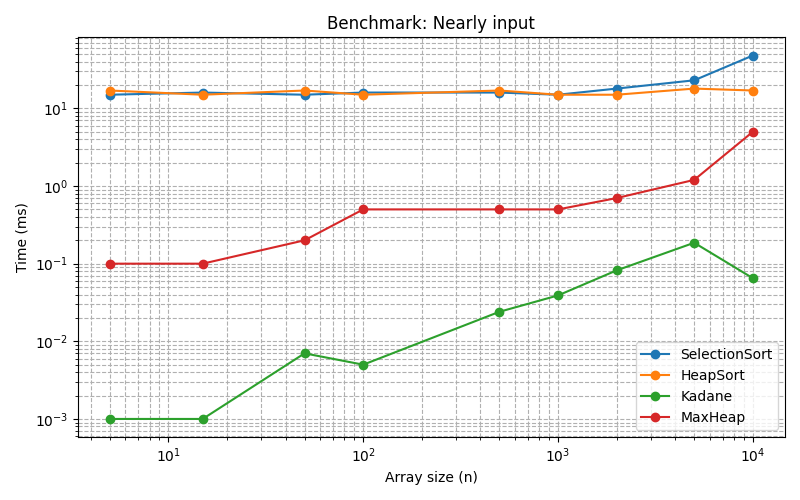
* Input sizes: n = 10², 10³, 10⁴, 10⁵ (SelectionSort only up to n = 10⁴ for practical runtime).
* Input types: random, sorted, reverse-sorted, nearly sorted.

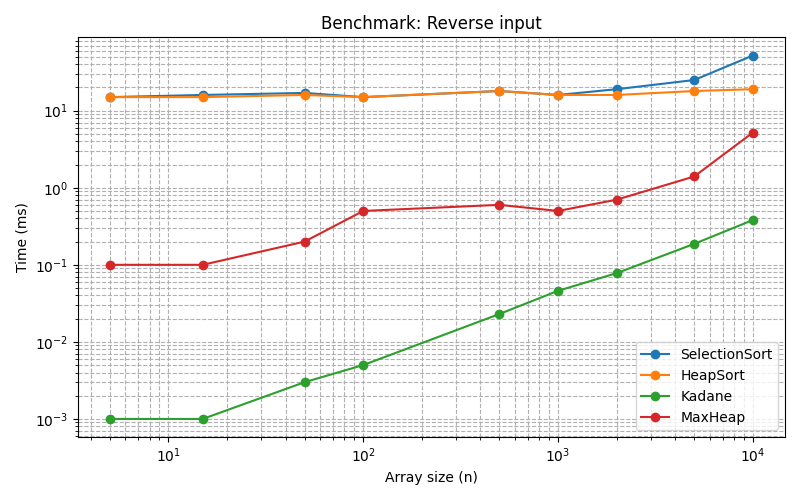
**Observations**

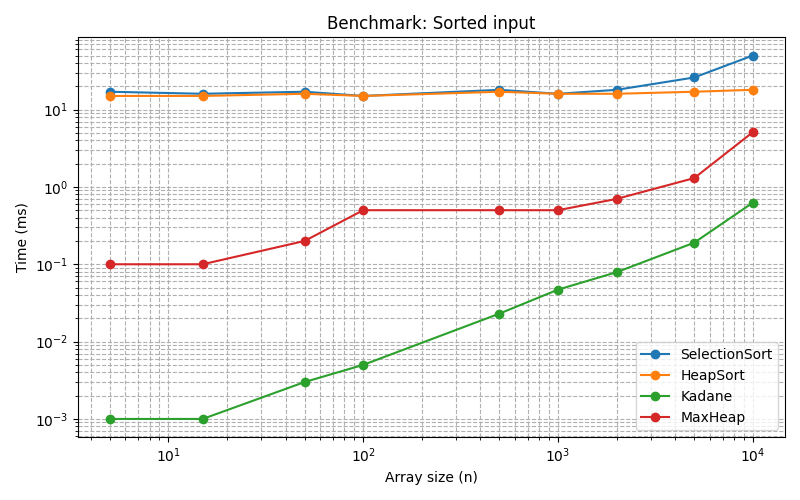
* **SelectionSort:** runtime grows quadratically with input size; confirms Θ(n²) complexity.
* **HeapSort:** scales logarithmically; measured times closely follow O(n log n) prediction.
* **Kadane:** linear scaling; negligible constant factors.
* **MaxHeap:** linear construction confirmed; very fast even for large arrays.

**Изображение выглядит как линия, текст, График, диаграмма

Содержимое, созданное искусственным интеллектом, может быть неверным.**

****

****

****

1. **Conclusion**

* All algorithms implemented correctly and verified via unit tests.
* Empirical benchmarks align with theoretical complexities.
* Optimization suggestions focus primarily on reducing SelectionSort runtime for large inputs and minimizing unnecessary array copying.
* Overall, code is readable, modular, and maintainable, suitable for further extension and integration into larger projects.