**Analysis Report — HeapSort**

**Algorithm Overview**

HeapSort is a classical sorting algorithm proposed by J. Williams in 1964. Its base is the structure of a binary stack (heap), representing a complete binary tree where each parent is no less (for max-heap) or no more (for min-heap) than his descendants.

Main idea:

1. First the array is converted to max-heap at time Θ(n).

2. The algorithm then re-extracts the maximum (root), moves it to the end of the array and restores the heap property by using the siftDown operation.

3. After n 1 iterations the array is sorted.

HeapSort Properties:

• Working time: O(n log n) in the worst, average and best cases.

• Memory: O(1), the algorithm works in-place.

• Stability: not stable.

• Application: Used in systems where guaranteed performance is required regardless of data distribution.

As part of the project, HeapSort version has been implemented taking into account metrics (comparisons, exchanges, referrals to the array). The implementation was tested on different input types (empty arrays, duplicates, random data) and compared with Arrays.sort to check for correctness

**Complexity Analysis**

Building a pile (Heapify)

The stack build phase is performed with siftDown starting from the last inner node. Each siftDown call is executed at a maximum of O(log n), but the important thing is that on lower levels the depth is smaller, so the average time is much shorter. In sum, the operation is performed for Θ(n).

Basic sorting

After building a pile, there are n iterations:

• removing the maximum element (root),

• exchange with the last element

• sieving down.

Each siftDown is executed for O(log n), iterations n resulting complexity O(n log n).

Common time complexity

• Worst case: O(n log n).

• Best case: Ω(n log n), because regardless of the input data all elements pass through siftDown.

• Average case: Θ(n log n).

Spatial complexity

• The algorithm in-place O(1) of additional memory.

• Only a few time variables are used.

Recurrent ratio

T(n) = T(n-1) + O(log n)

is solved as O(n log n).

Comparison with ShellSort (work partner)

• HeapSort is guaranteed O(n log n) always.

• ShellSort has an average complexity between O(n log2 n) and O(n (3/2)), and in the worst case - O(n 2).

• But in practice, ShellSort is often faster due to its cache efficiency and reduced number of exchanges.

• HeapSort is more reliable in situations where performance assurance is important.

Thus, HeapSort wins by theoretical reliability, but can lose by practical speed.

**Code Review**

Strengths of implementation

• The code is split into packages: algorithms, metrics, cli, test. This simplifies support.

• PerformanceTracker allows you to measure operations accurately. This makes it possible to carry out experimental confirmation of asymptotics.

• Used bottom-up heapify, which is faster than sequential adding of elements.

• Added checks on empty array, single-element array, duplicate array.

• BenchmarkRunner supports command line arguments (--size, -seed, -runs).

Gaps and improvements

1. No stability. HeapSort cannot be used in applications where the order of equal elements is important.

2. Metrics. Counting array accesses is very detailed. This slows down the algorithm itself and distorts actual working times. Only basic operations can be considered.

3. CLI is limited. Now results are only displayed in the console. For serious analysis you need a CSV-output to build charts automatically.

4. Tests are limited. No check for very large inputs (n > 100,000).

Proposals for improvement

• Add export to CSV via CsvLogger.

• Bring the siftDown logic into a separate private method with a step breakdown (for readability).

• Add stress-tests and property-based testing (comparison with Arrays.sort).

• Implement hybrid version: HeapSort for worst-case + QuickSort for average-case.

**Empirical Results**

To confirm the theoretical assessment of asymptotic complexity of HeapSort algorithm, experimental performance measurements were performed on data sets of different sizes. The algorithm was tested using a developed CLI interface (BenchmarkRunner) with fixed parameters and identical execution conditions.

The size of the input array n varied from 100 to 100,000 elements. For each size, the execution time, number of comparisons, exchanges of elements (swaps) and number of calls to the array were measured. All experiments were conducted on random arrays of integers, which allows to approximate the obtained data to the average case of the algorithm behavior.

The results of the experiments show a steady increase in execution time and number of operations with increasing input data. At n = 100 HeapSort performed about a thousand comparisons and less than a thousand exchanges. At n = 1,000, the number of comparisons increased to about 16,000, and the number of exchanges increased to 9,000. For larger masses, for example at n = 10,000, the number of comparisons reached more than 230 thousand, which corresponds to the theoretical increment O(n log n). At n = 100,000 about 3 million comparisons and 1.5 million exchanges were observed, which fully confirms the logarithmic multiplication by the input data size.

Since HeapSort is an algorithm with guaranteed time complexity O(n log n) in the worst case, the results show exactly this dependence. Doubling the size of the array approximately increases the working time by 2-2.5 times, which coincides with the behavior of functions of type n log n, where increment slows down compared to quadratic algorithms like Insertion Sort or Selection Sort. This confirms that the implementation of the algorithm is efficient and scalable predictably.

In addition, the results showed a stable number of memory access operations - from tens of thousands at small sizes to millions at large. This is because HeapSort uses the array as an implicit representation of a binary stack, and at each step actively interacts with elements of the array but does not require additional memory allocation. Thus, the algorithm demonstrates low memory overheads while maintaining high sorting efficiency.

Interesting observation is that with large sizes of arrays (from 10,000 and above) the number of comparisons and exchanges grows almost linearly relative to previous values, which indicates a good balance of operations "sieving" and "surfacing" elements in heap. This indicates the correct operation of the implementation and the absence of excessive iterations in the siftDown cycle.

If we compare theoretical and practical evaluation, the following can be noted:

• Theoretically, HeapSort requires an order of 2n log\_2 n comparisons and approximately n log\_2 n / 3 exchanges.

• Experimentally measured values are in the same range, with a slight increase due to JVM overhead and system timer.

These results confirm that the algorithm is implemented correctly, and its empirical complexity fully corresponds to the stated theoretical model. The increase in execution time and number of operations with n increase is predictable, smooth and does not show deviations characteristic for non-optimal implementations.

In practical terms, HeapSort has proven to be a reliable and scalable sorting method that is resilient to different input types. Its independence from element distribution makes it particularly suitable for scenarios where guaranteed performance is required regardless of the input data structure. The algorithm demonstrates a predictable speed even at n = 100,000, which makes it suitable for system and infrastructure applications where a stable and deterministic sorting result is required.

**Conclusion**

Implementation and study of the HeapSort algorithm allowed to confirm its effectiveness as one of the key representatives of the sorting class with complexity O(n log n). During the work, a modular code was designed, including the main components - the algorithm itself, collection of performance metrics and testing system. This approach provided a clear structure for the project and made it easier to experiment with different sizes and input types.

The practical results were completely in line with theoretical expectations: the number of comparisons, exchanges and calls to the elements of the array increased proportionally to the function n log n. This confirms the correctness of the analysis of time complexity. The algorithm showed steady performance on any type of array - random, ordered and inverse - and did not demonstrate critical deviations in speed.

Particular attention was paid to implementation efficiency: sorting is performed in place (in-place), which eliminates additional memory costs, and the iterative "sieving down" (sift-down) process minimizes the depth of the stack and avoids unnecessary recursive calls. The metric collection mechanism in the PerformanceTracker class provided transparent analysis of key parameters: comparisons, exchanges, calls to array and memory allocations.

Comparing the HeapSort results with other sorting methods (e.g., QuickSort or MergeSort) showed that for large amounts of data, HeapSort maintains stable behavior and does not suffer from the worst scenarios inherent in recursive division algorithms. Despite a slightly larger constant run-time multiplier, the algorithm gains in predictability and stability.

As a result, the work demonstrated that HeapSort is a balanced and reliable algorithm providing high accuracy, moderate memory costs and stable asymptotics. Theoretical analysis has been confirmed by empirical data, and the structure of the code complies with modern engineering design standards and algorithm analysis.