**Cross-Review Summary**

As part of the second task, each pair of students implemented and analyzed two sorting algorithms - one simple and one advanced.

My algorithm is ShellSort, an improved version of Insertion Sort which uses several intermediate passes with different steps (gap sequences).

The algorithm of my partner - HeapSort, based on the structure of a binary stack and guaranteeing asymptotic complexity O(n log n) in all cases.

Both implementations were realized with the same code quality standard, modular design (algorithms, metrics, cli, test), metric collection through PerformanceTracker and subsequent validation of correctness and performance.

The main purpose of the comparison is to identify differences in the efficiency, complexity and applicability of each algorithm depending on the size and structure of the data.

ShellSort is an improvement of simple quadratic sorting. His key idea is the sorting of elements that are at a gap distance, which gradually reduces to 1.

This makes it possible to partially organize the array in early iterations and significantly reduce the number of shifts in the final Insertion Sort pass.

HeapSort, on the other hand, uses a binary heap to sequentially extract maximum elements, which provides strictly limited uptime even for the worst input data.

|  |  |  |
| --- | --- | --- |
| Option | **ShellSort** | **HeapSort** |
| **Best case** | O(n log n) | O(n \log n) |
| **Average case** | O(n^{3/2}) | O(n \log n) |
| **Worst case** | O(n^{3/2}) | O(n \log n) |
| **Space complexity** | O(1) | O(1) |
| **Stability** | No | No |

Theoretically, HeapSort has stable asymptotics independent of input data,

while the efficiency of ShellSort depends heavily on the choice of sequence of steps (Shell, Knuth, Sedgewick).

However, small ShellSort arrays are often faster due to easier operations and better use of the cache.

Practical experiments showed that for small array sizes (up to 5,000 elements), ShellSort had shorter run times than HeapSort.

When increasing the data size to 10,000 - 100,000 elements, HeapSort started to take a clear lead, proving its advantage in large input volumes.

For example:

• When n = 1,000, ShellSort was performed in an average of 0.35 ms and HeapSort in an average of 0.5 ms;

• For n = 100,000, ShellSort already required about 18 ms, whereas HeapSort required 12 ms.

By number of comparisons HeapSort kept close to n log n dependence,

while ShellSort showed a more chaotic but still sub-quadratic dynamic depending on the selected gap sequence.

These observations confirmed that ShellSort is better suited for medium and near-sorted data,

a HeapSort is optimal for large and random sets.

Both implementations showed high quality: the code is structured, contains tests and metrics.

HeapSort is implemented strictly according to the theory, with siftDown() function and efficient memory management.

ShellSort is flexible and can experiment with different gap sequences (for example, h\_{k+1} = 3h\_k + 1).

In terms of architecture and code readability, both solutions meet the requirements:

methods are atomic, there is protection from incorrect input data, implemented modularity and metric collection.

Based on the analysis, it can be concluded that:

• HeapSort provides guaranteed performance for any input data,

• ShellSort achieves high operating speed at moderate dimensions thanks to simplicity and better caching.