**Readme**

This project focuses on the implementation and analysis of the **ShellSort** algorithm as part of Assignment 2. The goal was to study ShellSort’s performance, verify its theoretical complexity, and compare empirical results with analytical predictions and the partner’s algorithm (HeapSort). ShellSort is an optimization of Insertion Sort that allows the exchange of elements far apart, significantly reducing the number of shifts required. The algorithm divides the array into subarrays using different **gap sequences** (such as Shell’s, Knuth’s, and Sedgewick’s) and performs insertion-like passes for each gap. This approach efficiently decreases disorder and leads to improved performance, especially for medium-sized datasets. The implementation supports multiple gap sequences, allowing comparison between them. A PerformanceTracker module records key metrics — comparisons, swaps, and array accesses — while a command-line interface (BenchmarkRunner) runs tests for various input sizes and random seeds. All results were verified against Java’s built-in Arrays.sort() to ensure correctness. Theoretical analysis shows that ShellSort’s time complexity depends on the chosen gap sequence. In the worst case, it approaches **O(n²)** (for Shell’s original gaps), while with optimized sequences such as Knuth’s or Sedgewick’s, it performs closer to **O(n^(3/2))** or even **O(n log² n)**. Its space complexity remains **O(1)**, since sorting occurs in place. Empirical results confirmed that ShellSort performs extremely well for moderately sized or nearly sorted data. Execution time grew sub-quadratically with input size, and optimized gaps provided significant speed improvements compared to the original version. For large datasets, HeapSort outperformed ShellSort due to its guaranteed O(n log n) behavior, but ShellSort remained competitive for smaller inputs because of lower constant factors. The project follows professional development practices — modular code structure, unit tests covering edge cases, reproducible benchmarking, and clean Git workflow. This implementation and analysis demonstrate ShellSort’s versatility: while not asymptotically optimal, it remains one of the most efficient in-place sorting algorithms for practical use.