Comparative Analysis of Shell Sort and Heap Sort

By Daniyar Abikenov and Damir Almasuly

Introduction

This report presents a detailed comparative analysis of two sorting algorithms: Shell Sort and Heap Sort. The purpose of the comparison is to evaluate their operational principles, computational complexity, performance characteristics, and implementation quality. Shell Sort, proposed by Donald L. Shell in 1959, is a generalization of insertion sort that allows elements to move over long distances in each iteration, thereby reducing large-scale disorder early in the process and achieving subquadratic time complexity. Heap Sort, on the other hand, is a classical algorithm based on the binary heap data structure. It guarantees deterministic $\Theta(n \log n)$ performance in all cases and is particularly effective for large datasets where stability is not required. Both algorithms operate in-place, using $O(1)$ additional space, but neither maintains the stability of equal elements. The sections below present an in-depth examination of each algorithm, their respective strengths and weaknesses, the improvements suggested in the reports, and a comparative conclusion based on theoretical and empirical evidence.

Description and Evaluation of Shell Sort

Shell Sort improves upon the insertion sort by comparing and shifting elements that are far apart from each other during early passes. The algorithm uses a decreasing sequence of gaps—also known as increments—such as the classical Shell sequence (n/2, n/4, ..., 1), Knuth's sequence (1, 4, 13, 40, 121, ...), or Sedgewick's optimized sequence. For each gap, the algorithm performs an insertion sort among elements separated by that distance, progressively reducing large-scale disorder. As the gap diminishes, the array becomes increasingly ordered, and the final pass (with gap = 1) completes efficiently using standard insertion sort. This multi-scale process accelerates sorting by addressing disorder on different levels and allows Shell Sort to outperform simple quadratic algorithms on moderately sized arrays.

Complexity Analysis

The efficiency of Shell Sort is highly dependent on the chosen gap sequence. Its theoretical behavior cannot be expressed in a closed mathematical form since the number of comparisons and swaps varies with the sequence. Nevertheless, established asymptotic bounds can be provided for different cases.

In the best case, when the array is nearly sorted, the time complexity approaches $\Omega(n \log n)$. In this situation, few shifts are needed during the final insertion pass, resulting in a cost close to that of optimal n log n algorithms. For average cases with random data, using Knuth's gap sequence leads to approximately Θ(n^1.5) complexity, while the Sedgewick sequence can improve this to roughly Θ(n^1.33). Thus, Shell Sort ranks among the fastest algorithms that are worse than n log n in asymptotic terms but still highly competitive for mid-sized datasets. In the worst case, particularly with naive sequences (e.g., halving the gap each time), the number of comparisons can approach $n^2/2$, yielding $O(n^2)$ complexity. Despite this, Shell Sort rarely exhibits such poor performance in practice when well-chosen sequences are used.

Shell Sort requires only O(1) additional space beyond the input array, making it memory-efficient and suitable for constrained systems. However, the algorithm is not stable, since long-distance swaps can alter the relative order of equal elements.

Strengths and Weaknesses

Shell Sort offers several notable advantages:

- It is faster than basic $O(n^2)$ algorithms such as bubble sort or insertion sort on moderate datasets.
- It is simple to implement and operates entirely in-place.
- It allows flexibility in tuning via different gap sequences, which can be adapted to input characteristics.

At the same time, the algorithm has inherent limitations:

- Its theoretical analysis is complex, and there is no universally optimal sequence of gaps.
- For large datasets, it scales worse than O(n log n) algorithms, eventually becoming inefficient compared to Heap Sort or Merge Sort.
- It is unstable and thus unsuitable for tasks requiring order preservation.
- In the analyzed implementation, unit testing coverage was limited to small cases, leaving performance at scale insufficiently verified.

Suggested Improvements

The report proposed several optimizations for Shell Sort:

1. **Partial loop unrolling** in the inner loop to reduce branch mispredictions.
2. **Local caching** of repeated array accesses to minimize memory reads.
3. **Dynamic gap tuning** during execution instead of using static predefined sequences.
4. **JVM warm-up iterations** before benchmarking to eliminate noise from compilation and cache effects.

5. **Automatic reset of performance metrics** between test runs to prevent data accumulation from earlier experiments.

Implementation Quality

The Shell Sort project demonstrated strong modularity and clean code structure. Its package design separated algorithm logic, performance metrics, CLI, and JMH benchmarking components, adhering to object-oriented principles. The code followed Java conventions and included concise comments and documentation. Each method had a single responsibility, making the design extensible and maintainable. Only minor issues were noted, such as limited test coverage and missing metric resets, but overall, the implementation was well-organized, correct, and aligned with theoretical expectations.

Description and Evaluation of Heap Sort

Heap Sort is based on the binary heap data structure and proceeds in two main phases: heap construction and repeated extraction. In the first phase, the algorithm builds a max-heap by applying the heapify operation to non-leaf nodes in reverse level order. The heapify function ensures that each subtree satisfies the heap property, where every parent node is larger than its children. Once the max-heap is built, the algorithm enters the extraction phase, repeatedly swapping the root (maximum element) with the last unsorted element, reducing the heap size by one, and re-heapifying the root. The process continues until all elements are sorted in ascending order. The entire procedure operates within the same array and requires no additional memory.

Complexity Analysis

Heap Sort's time complexity is well defined and uniform across all input distributions. Building the heap takes $O(n)$ time according to Floyd's heap construction theorem, as the total cost of heapify operations over all nodes is linear. The extraction phase involves $n-1$ iterations, each requiring $O(\log n)$ operations to restore the heap property. Consequently, the overall time complexity is $\Theta(n \log n)$ in the best, average, and worst cases. The algorithm's auxiliary space complexity is $\Theta(1)$, as it uses only a few helper variables and avoids recursion through an iterative implementation.

Heap Sort's runtime grows predictably with input size. Experimental measurements confirmed that when input size increases by a factor of 10, runtime approximately doubles, consistent with $n \log n$ growth. This makes the algorithm highly reliable for applications requiring consistent performance.

Strengths and Weaknesses

Heap Sort offers several clear advantages:

- Deterministic $\Theta(n \log n)$ complexity across all cases.
- Constant auxiliary space and in-place operation.
- Predictable, robust performance on large datasets.
- No recursive overhead in iterative implementations.

However, it also exhibits certain drawbacks:

- The algorithm is not stable and can change the order of equal elements.
- Its memory access pattern is non-sequential, resulting in poor CPU cache utilization compared to algorithms with linear memory traversal such as Shell Sort.
- It is non-adaptive and does not benefit from pre-sorted data.
- The original implementation contained minor design flaws, such as limited inline documentation and mixing computation with file I/O operations.

Suggested Improvements

The report identified and recommended the following key improvements:

1. **Remove unused variables**, such as an unused recursion depth counter, to reduce stack allocation.
2. **Eliminate disk I/O inside the sorting function** and move CSV writing operations to an external benchmarking module to follow the Single Responsibility Principle.
3. **Use dependency injection for the performance tracker** instead of internal instantiation, improving modularity and testability.
4. **Measure time more granularly**—for example, separately for heap construction and extraction—to gain finer analytical insights.
5. **Add a command-line interface (CLI)** for flexible benchmarking and user-defined experiments, similar to the Shell Sort project.

These adjustments do not alter Heap Sort's asymptotic complexity but improve its runtime constants, modularity, and maintainability. After the modifications, the implementation became cleaner and more scalable.

Implementation Quality

The reviewed Heap Sort implementation was found to be functionally correct and consistent with the theoretical model. The use of an iterative heapify approach avoided recursion overhead and ensured efficiency. Code readability and structure were generally good, with clear variable naming and logical control flow. The main issue identified was the direct inclusion of I/O operations inside the sorting routine, which was later refactored. After separating concerns and introducing dependency injection, the algorithm achieved proper modularity. Test coverage was comprehensive, including sorted, reversed, single-element, and empty arrays. The only missing component was a CLI for dynamic experiments, which was recommended for future development. Overall, the implementation was correct, robust, and efficient, with minor refinements improving its engineering quality.

Comparative Analysis

From both theoretical and empirical perspectives, Heap Sort and Shell Sort exhibit distinct trade-offs.

In terms of asymptotic complexity, Heap Sort is superior, maintaining $\Theta(n \log n)$ performance in all cases. Shell Sort, while potentially achieving near n log n time on nearly sorted data, typically runs in $\Theta(n^{1.5})$ for random inputs and can degrade to $O(n^2)$ with poor gap sequences. Both algorithms require only $O(1)$ extra memory and are non-stable.

In empirical testing, Shell Sort outperformed Heap Sort by 25–30% for small datasets ($n \le 1{,}000$) due to simpler control flow and better cache locality. However, its superlinear scaling caused runtimes to grow faster beyond n ≈ 10,000, where Heap Sort consistently became about twice as fast. The crossover point between them thus depends on dataset size: Shell Sort dominates small to mid-scale workloads, while Heap Sort excels at larger scales.

Regarding memory usage and cache behavior, both algorithms are in-place, but Shell Sort's sequential access pattern provides better cache efficiency, reducing L1 cache misses. Heap Sort's scattered access pattern, jumping between parent and child indices, introduces additional latency on large datasets. This factor contributes to Shell Sort's advantage on small arrays despite its theoretically higher complexity.

In terms of design and modularity, Shell Sort's implementation was architecturally cleaner from the start, featuring separate modules for algorithm logic, metrics tracking, CLI, and JMH benchmarking. Heap Sort's initial implementation mixed computation with tracking and file output, but after refactoring, both achieved comparable modularity. Both adhered to OOP principles, though Shell Sort's project provided a more complete experimental framework out of the box.

Conclusion

Based on both reports, it can be concluded that Shell Sort and Heap Sort each confirm their theoretical characteristics and have distinct domains of practical efficiency. Shell Sort performs exceptionally well on moderately sized datasets, benefitting from small constant factors and cache-friendly behavior. Heap Sort, meanwhile, offers predictable scalability, superior asymptotic performance, and consistency for large datasets. Empirical evidence supports these findings: Shell Sort outperforms Heap Sort on smaller arrays, but Heap Sort overtakes it for larger ones.

Both implementations demonstrated correctness, modularity, and alignment with expected theoretical profiles—Shell Sort showing Θ(n^1.5) behavior and Heap Sort maintaining Θ(n log n). Test coverage confirmed robustness under various input conditions. The improvements proposed in both analyses—particularly those involving modularization, metric management, and benchmark accuracy—further enhanced the reliability and clarity of each implementation.

Ultimately, the choice between Shell Sort and Heap Sort depends on the target use case. Shell Sort is preferable for small and medium-sized datasets where quick, memory-efficient sorting is needed, while Heap Sort is the more suitable option for large-scale sorting tasks requiring consistent and theoretically optimal performance. Both algorithms, as implemented and analyzed, met their design objectives and validated the theoretical expectations established in the respective reports.