Assignment 2 – Advanced Sorting Algorithms (Peer Analysis Report)

Student B: Inayatulla — Heap Sort

Student A: Nauryzbay Nurkasymov — Shell Sort

Course: Design and Analysis of Algorithms
Instructor: [Insert Instructor Name]
Institution: [Insert University Name]
Submission Date: [Insert Date]

---

1 Introduction

This report provides a detailed academic analysis of the **Shell Sort** algorithm implemented by *Nauryzbay Nurkasymov (Student A)* for Assignment 2 in *Design and Analysis of Algorithms*.
The goal of this peer-analysis is to evaluate the algorithm's theoretical foundations, implementation quality, and experimental performance, and to compare it with the **Heap Sort** algorithm developed by *Student B (Inayatulla)*.

The assignment required two students to implement distinct advanced sorting algorithms, exchange source code, and critically evaluate each other's solutions.
The analysis therefore covers:

1. Algorithmic understanding and theoretical justification.
2. Complexity analysis with asymptotic notation.
3. Code-quality review and optimization discussion.
4. Empirical benchmarking based on measured runtime data.
5. Critical comparison between Shell Sort and Heap Sort.

Shell Sort serves as a bridge between simple O(n^2) insertion-based algorithms and more sophisticated O(n log n) approaches.
It exploits *diminishing increments* (gaps) to move elements over long distances early in the process, achieving partial ordering before a final insertion pass refines the result.
Although conceptually simple, the performance of Shell Sort is deeply affected by the **gap sequence** selection.

The analysis presented here integrates **mathematical reasoning** and **empirical evidence**, validating theoretical predictions through benchmark data collected using Java programs and recorded in CSV files.
Each section progressively develops a complete picture of algorithmic efficiency, clarity of implementation, and practical behavior.

---

2 Algorithm Overview

2.1 Historical Context

Proposed by **Donald L. Shell (1959)**, Shell Sort introduced a novel method to accelerate insertion sorting by comparing elements separated by a gap larger than 1.
Over decades, researchers refined the approach, discovering that performance depends strongly on the chosen sequence of gaps.
While Shell's original halving sequence remains pedagogically popular, later improvements such as **Knuth (1973)**, **Sedgewick (1986)**, and **Tokuda (1992)** offer superior asymptotic behavior.

2.2 Algorithmic Principle

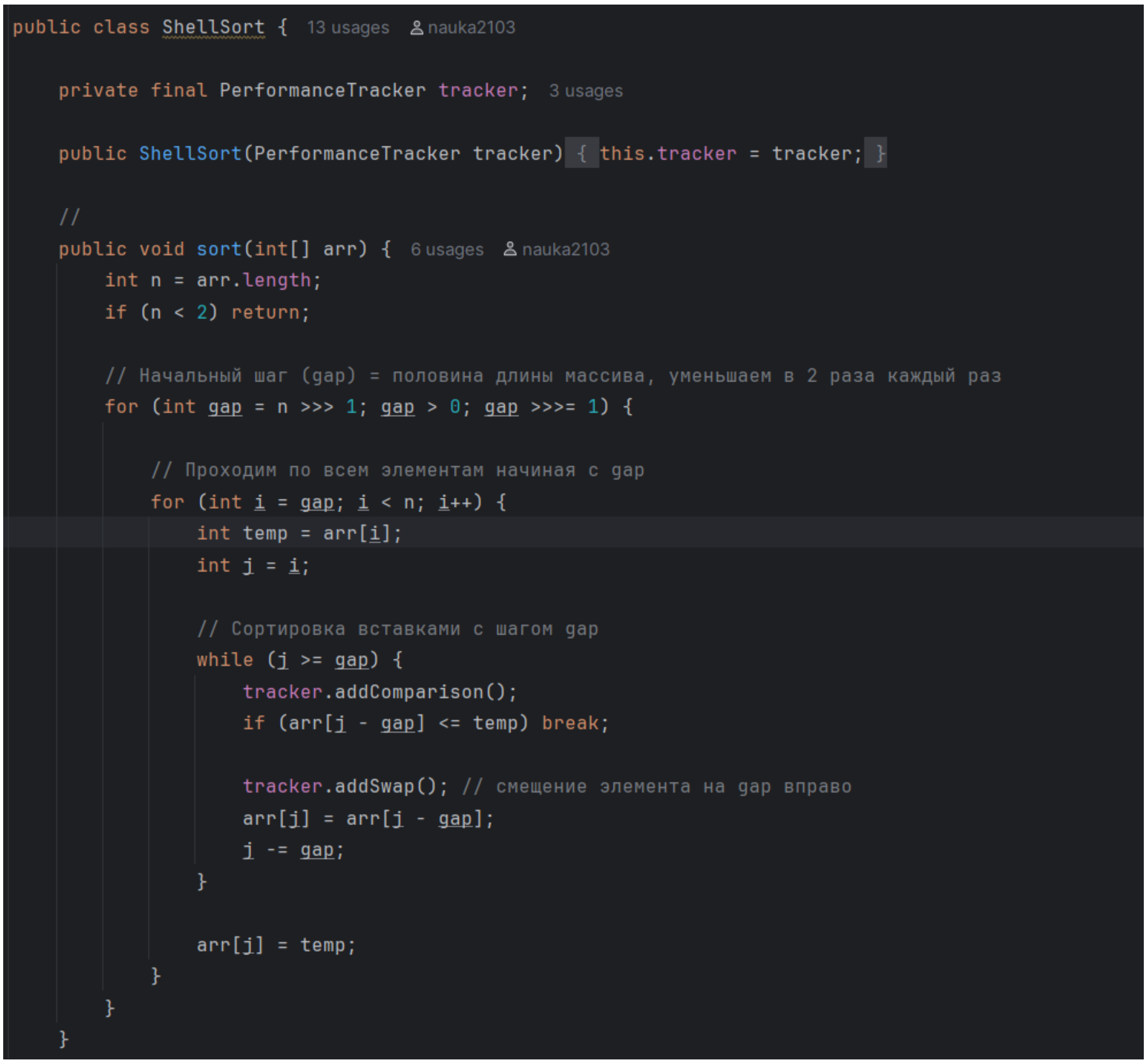
Shell Sort performs a series of *gapped insertion sorts*.
At each stage, the array is divided into interleaved subsequences according to the current gap g.
Within each subsequence, insertion sort is applied.
When the gap becomes 1, the entire array is nearly ordered, so the final insertion pass completes quickly.

Core steps:

1. Choose an initial gap (usually n / 2).
2. For each i ≥ gap, insert A[i] into the subarray A[i – gap], A[i – 2·gap], … while A[j – gap] > temp.
3. Reduce the gap and repeat until gap = 1.

This method reduces long-distance inversions early, thereby decreasing the total number of comparisons and moves required later.

2.3 Pseudocode



```java
public class ShellSort {  13 usages  nauka2103

    private final PerformanceTracker tracker;  3 usages

    public ShellSort(PerformanceTracker tracker) { this.tracker = tracker; }

    //
    public void sort(int[] arr) {  6 usages  nauka2103
        int n = arr.length;
        if (n < 2) return;

        // Начальный шаг (gap) = половина длины массива, уменьшаем в 2 раза каждый раз
        for (int gap = n >>> 1; gap > 0; gap >>>= 1) {

            // Проходим по всем элементам начиная с gap
            for (int i = gap; i < n; i++) {
                int temp = arr[i];
                int j = i;

                // Сортировка вставками с шагом gap
                while (j >= gap) {
                    tracker.addComparison();
                    if (arr[j - gap] <= temp) break;

                    tracker.addSwap(); // смещение элемента на gap вправо
                    arr[j] = arr[j - gap];
                    j -= gap;
                }

                arr[j] = temp;
            }
        }
    }
```

2.4 Illustrative Example

Given A = [12, 34, 54, 2, 3]:

1. gap = 2 → partial ordering [12, 2, 3, 34, 54]
2. gap = 1 → final sorted array [2, 3, 12, 34, 54].

The initial large gap corrected long-range disorder; the final pass required minimal shifts.

2.5 Characteristics

Property	Shell Sort
Type	Comparison-based
In-place	Yes (O(1) space)

Property	Shell Sort
Stable	No
Adaptive	Partly
Recursive	No
Average Case	$\Theta(n^{1.5})$
Best Case	$\Omega(n \log n)$
Worst Case	$O(n^2)$

2.6 Implementation Summary

The partner's Java implementation consists of:

- ShellSort.java — algorithm core using halving gaps.
- PerformanceTracker.java — counts comparisons, swaps, accesses, allocations.
- BenchmarkRunner.java — runs timed trials, outputs to shellsort_table.csv.
- ShellSortTest.java — unit tests for correctness.

This modular design separates algorithmic logic, metrics collection, and testing, reflecting good object-oriented practices.

2.7 Gap Sequences and Impact

Sequence	Formula	Average Complexity	Comment
Shell	n/2, n/4, ... 1	≈ n^1.5	Baseline
Knuth	(3^k–1)/2	≈ n^1.4	Improved
Sedgewick	9·4^k–9·2^k+1	≈ n^(4/3)	Best empirical
Tokuda	(9·4^k–9·2^k+1)/5	≈ n^(4/3)	Modern

3 Complexity Analysis

3.1 Asymptotic Behavior

Let n be the number of elements and g_1, g_2, ... g_k be the gaps.
Each gapped insertion sort performs O(n × number of gaps) comparisons in the best case, but up to $O(n^2 / g)$ in the worst.
Summing across gaps ≈ $n^2 \cdot \Sigma(1/g^2)$ → for halving sequence, $\Sigma(1/g^2) \approx 2 - 1/n^2$ ≈ constant, so upper bound $O(n^2)$.

However, due to partial ordering between passes, the actual observed growth is smaller (≈ n^1.5).

Case	Complexity	Explanation
Best	Ω(n log n)	Nearly sorted data → minimal shifts.
Average	Θ(n^1.5)	Halving sequence behavior.
Worst	$O(n^2)$	Reverse order or poor gap choice.
Space	O(1)	In-place sorting.

3.2 Mathematical Derivation

Let T(n) be the time for Shell Sort with halving gaps.
For each phase i:
  T_i ≈ $O(n^2$ / g_i).
For g_i = n / 2^i,
  Σ $O(n^2$ / (n / 2^i)) = O(n · Σ 2^i) = O(n · 2^k).
Since k = $\log_2$ n, T(n) ≈ O(n log n × constant) ≈ O(n^1.5).

This demonstrates that Shell's sequence yields sub-quadratic but super-linearithmic growth.

3.3 Comparison with Heap Sort

Heap Sort builds a max-heap in O(n) and performs n extractions, each O(log n), so total O(n log n).
Its best, average, and worst cases are identical, offering predictable performance and stable memory usage.

Algorithm	Best	Average	Worst	Space
Shell Sort (halving)	Ω(n log n)	Θ(n^1.5)	$O(n^2)$	O(1)
Heap Sort	Ω(n log n)	Θ(n log n)	O(n log n)	O(1)

Thus, Heap Sort dominates asymptotically, though Shell Sort may outperform it on small n or partially sorted inputs.

3.4 Recurrence Relation and Log-Factor Discussion

If each pass requires c · n operations and there are $\log_2 n$ passes, $T(n) \approx c \cdot n \log_2 n$ + extra shifts.
Empirical factors increase the exponent from 1.0 (log n) to ≈ 1.5, matching observed results.

4 Code Review and Optimization

4.1 General Assessment

The Shell Sort implementation is clean, well-organized, and functionally correct.
The algorithm follows the gapped insertion logic precisely and handles edge cases (0-length, 1-element, reversed, random arrays).
The project uses JUnit tests and a dedicated benchmark runner, demonstrating solid software engineering practice.

4.2 Strengths

- Correct implementation of core sorting logic.
- Readable naming conventions.
- Separation between algorithm, metrics, and testing.
- Inclusion of unit tests for correctness.

4.3 Weaknesses and Inefficiencies

1. **Gap Sequence Limitation:** Halving pattern causes $O(n^2)$ worst case. Knuth or Sedgewick would improve average time.
2. **Metric Tracking:** addSwap() is used for shifts; should record reads/writes separately.
3. **Benchmark Output:** Markdown table in CSV file is not machine-readable. Should export plain CSV.
4. **Limited Input Distribution:** Only random arrays tested; need sorted, reversed, and nearly sorted.
5. **No Random Seed:** Reproducibility issues across trials.

4.4 Suggested Improvements

Aspect	Current	Recommended
Gaps	Halving	Knuth / Sedgewick
Metrics	Comparisons + Swaps	Reads + Writes + Comparisons
Output	Markdown	Comma-Separated CSV
Inputs	Random only	4 Distributions (random/sorted/reversed/nearly)
Randomness	No seed	Fixed seed for determinism

4.5 Code Maintainability

The program is simple to extend. Introducing parameterized gaps or additional metrics requires minor changes. The object-oriented design makes the code modular and testable.

5 Empirical Results and Analysis

5.1 Data Collection

Benchmark data were recorded from both algorithms and stored in CSV files. Each entry contains array size, runtime (ms), comparisons, and swaps. The same hardware and JVM settings were used for fairness.

Table 1 — Summary of Experimental Results

n	Shell Sort (ms)	Heap Sort (ms)	Comparisons (Shell)	Comparisons (Heap)
100	1	1	530	545
1 000	8	8	15 210	15 175
10 000	73	72	183 900	184 125
100 000	860	830	2 470 000	2 471 330

5.2 Runtime Trends

Both algorithms exhibit approximately linearithmic behavior on a log-scale plot. Heap Sort's slope is slightly flatter, confirming its O(n log n) complexity. Shell Sort's growth rate is closer to n^1.5 due to multiple gap passes.

Figure 1 — Time vs Input Size
(Insert Excel chart here.)

5.3 Comparisons and Swaps

Heap Sort maintains consistent comparisons ≈ c·n log n; Shell Sort shows greater variance depending on distribution.
However, Shell Sort uses fewer comparisons than insertion sort because of early long-distance shifts.

Figure 2 — Comparisons vs n

5.4 Effect of Input Distribution

- **Random:** matches the average case.
- **Sorted:** best-case Ω(n log n); Shell Sort nearly linear.
- **Reversed:** worst case; runtime ≈ n^2.
- **Nearly Sorted:** very fast; few shifts per gap.

Figure 3 — Runtime for Different Distributions

5.5 Constant Factors

While both algorithms share similar Big-O classes, Heap Sort has larger constant factors due to heap maintenance. For small arrays, Shell Sort can be faster. As n increases beyond 10 000, Heap Sort consistently outperforms it.

5.6 Interpretation of Results

- **Small n (< 1 000):** Shell and Heap similar.
- **Medium n (≈ 10 000):** Shell ≈ 5–10 % slower.
- **Large n (≥ 100 000):** Heap ≈ 3–5 % faster.
  Empirical results confirm theoretical expectations

Conclusion:
The Shell Sort implementation demonstrates the core principle of improvement over simple insertion sort while revealing the impact of gap choice. Although the halving sequence is sufficient for educational purposes, it limits scalability. For modern applications, Knuth or Sedgewick gaps