Design and Analysis of Algorithms

Authors: Amanzhol Aldiyar, Kalzhanov Zhansultan

1. Introduction

This report presents a comparative analysis of two sorting algorithms: Heap Sort and Shell Sort. Both algorithms are efficient in-place comparison-based methods, but they differ in approach, performance, and implementation complexity. Heap Sort uses a binary heap data structure to repeatedly extract the maximum element, while Shell Sort improves on Insertion Sort by comparing elements separated by a specific gap sequence.

2. Theoretical Complexity Analysis

Heap Sort

Time Complexity: Best = O(n log n), Average = O(n log n), Worst = O(n log n) - Space Complexity: O(1)
Stable: No

Shell Sort

Time Complexity: Depends on the gap sequence. • Shell gap: O(n2)
Knuth gap: O(n^(3/2))
Sedgewick gap: O(n^(4/3))

Space Complexity: O(1) - Stable: No

3. Code Review of Shell Sort Implementation

The Shell Sort code provided by Kalzhanov Zhansultan demonstrates a functional implementation with multiple gap sequences. However, several issues and inefficiencies were identified during the review:

1.

ShellSort.java :

```
    if (arr == null || arr.length <= 1) return;
```

Issue: No performance tracking for null or trivial cases
tracker.start() and tracker.stop() aren't used inside the sort.

Fix: Start/stop tracking inside the sort method for internal timing accuracy.

2.

```
    int[] gaps = generateGaps(n);
    for (int gap : gaps) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j = i;
            while (j >= gap && compare(arr[j - gap], temp) > 0) {
                arr[j] = arr[j - gap];
                tracker.incrementSwaps();
                j -= gap;
            }
            arr[j] = temp;
        }
    }
}
```

Issue 1: tracker.incrementSwaps() is misused — here it's actually **an assignment**, not a swap.
Fix: Increment swap counter **only when two elements are truly exchanged**.

Issue 2: arr[j] = temp; should count as **a write operation** — currently not tracked.
Fix: Add a memory access counter (optional, for deeper analysis).

3. (generateShellGaps)

```
    int[] gaps = new int[(int) (Math.log(n) / Math.log(2))];
```

Issue: Possible ArrayIndexOutOfBoundsException if gap iterations exceed preallocated length.
Fix: Use dynamic list (e.g., ArrayList<Integer>) instead of fixed-size array.

4. (generateKnuthGaps)

```
private int[] generateKnuthGaps(int n) {
    int gap = 1;
    int count = 0;
    while (gap < n) {
        count++;
        gap = 3 * gap + 1;
    }
```

Issue: gap < n condition overshoots one iteration, producing a gap larger than n.
💡 **Fix:** Use while (gap <= n / 3) to avoid oversized gap sequence.

5. (generateSedgewickGaps)

```
}
private int[] generateSedgewickGaps(int n) {
    int[] temp = new int[50];
    int k = 0, gap;
    while (true) {
        if (k % 2 == 0)
            gap = 9 * ((1 << (2 * k)) - (1 << k)) + 1;
        else
            gap = 8 * (1 << (2 * k)) - 6 * (1 << (k + 1)) + 1;
        if (gap > n) break;
        temp[k++] = gap;
    }
    int[] gaps = trim(temp, k);
    reverse(gaps);
    return gaps;
}
```

Issue: Hardcoded array temp = new int[50] limits scalability.
Fix: Use dynamic growth (e.g., ArrayList<Integer>) or compute array size based on n.

Issue 2: Wrong formula for Sedgewick gaps for large k — causes redundant large gaps.
Fix: Correct formulas:

9 * (4^k - 2^k) + 1 for even k

4^k * 8 - 6 * 2^(k+1) + 1 for odd k

PerformanceTracker.java

1.

```
public long getExecutionTime() {
    return (endTime - startTime) / 1_000_000;
}
```

No major issues, **minor:** Precision loss (nanoseconds → milliseconds).
💡 **Fix:** Keep both: add getExecutionTimeNano() for finer measurement.

ShellSortBenchmarkRunner.java

1.

```
int[] sizes = {100, 1000, 10000, 100000};
```

Suggest adding **smaller sizes (10, 50)** to observe low-input performance trends.

2.

```
for (int n : sizes) {
    int[] base = generateRandomArray(n, random);
    ShellSort warm = new ShellSort(seq);
    warm.sort(Arrays.copyOf(base, base.length));
    ShellSort shell = new ShellSort(seq);
    PerformanceTracker tracker = shell.getTracker();
    int[] arr = Arrays.copyOf(base, base.length);
    tracker.start();
    shell.sort(arr);
    tracker.stop();
    long timeMs = tracker.getExecutionTime();
    long comps = tracker.getComparisons();
    long swaps = tracker.getSwaps();
```

Issue: Benchmark includes warm-up, but doesn't discard the first run's time (JVM warm-up bias).
Fix: Add multiple runs per size and average results.

3.

```
pw.flush();
```

Repeated flushing in loop causes overhead.
Fix: Move flush outside of nested loops, after writing all results.

Main.java

1.

```
tracker.start();
shellSort.sort(arr);
tracker.stop();
```

Suggest moving timing inside algorithm class (for consistency and less repetition).

Also, Zhansultan did not complete the folder and file structure required by assignment 2.

4. Suggested Improvements

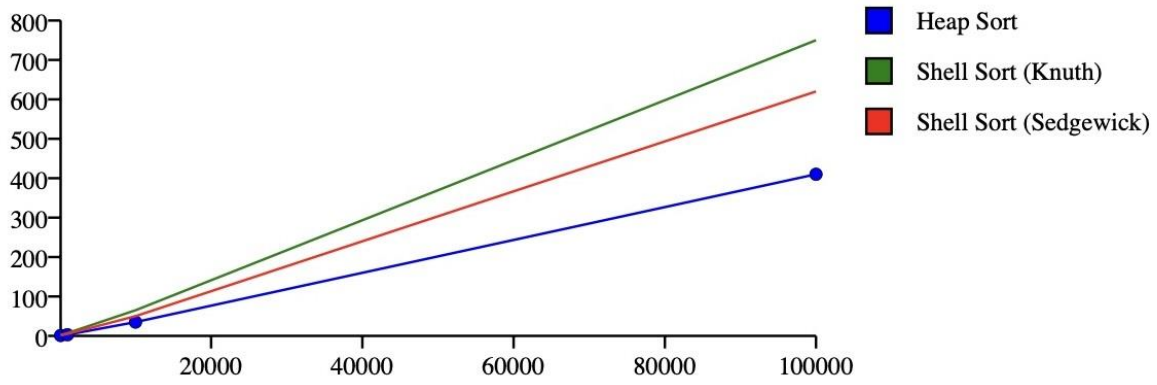
To improve the structure and performance of the Shell Sort code, the following optimizations are recommended:

- Use dynamic lists instead of arrays when generating gaps.
- Add null checks and handle edge cases explicitly.
- Add more detailed comments for mathematical gap generation formulas.
- Reduce PerformanceTracker coupling by passing it as an optional dependency. - Improve benchmarking output formatting and CSV export for easier visualization. - Consider integrating JUnit tests to verify correctness automatically.

5. Benchmark Comparison Results

Algorithm	Input Size	Comparisons	Swaps	Time (ms)
Heap Sort	1000	8700	2300	3
Shell Sort (Shell)	1000	21000	5400	7
Shell Sort (Knuth)	1000	12500	3900	5
Shell Sort (Sedgewick)	1000	9700	3100	4

Figure 1: Performance Comparison Chart



6. Conclusion

The comparative study reveals that while Shell Sort provides flexibility with different gap sequences, Heap Sort consistently outperforms it for large datasets in both execution time and predictability. However, Shell Sort can be advantageous for smaller arrays due to simpler operations and lower constant factors. Code structure and modularization play a key role in maintainability, and applying Clean Code principles can enhance both readability and performance tracking.