**Assignment 2 Report.**

(Shell Sort)

**1. Algorithm Overview**

Shell Sort is a generalization of Insertion Sort that introduces a gap sequence to compare and swap distant elements. By progressively reducing the gap, the algorithm produces a nearly sorted array before a final insertion pass.

* Partner’s code implements three variants: Shell’s original sequence (n/2, n/4, …), Knuth’s sequence (3k + 1), and Sedgewick’s sequence.
* Metrics collected: execution time, comparisons, and shifts.
* Code integrates with a CLI and supports CSV logging.

**2. Complexity Analysis**

Gap Insertion Phase:  
For each gap g, the array is divided into g subarrays. Each subarray is insertion-sorted with cost O(n/g · g) = O(n).  
The total cost depends on the number and size of gaps.

* Worst Case (Shell gaps, n/2, n/4, …):  
  Many small insertion passes → O(n²).
* Average Case (Knuth gaps, 3k+1):  
  Empirically and theoretically bounded by Θ(n^1.5).
* Best Case (Sedgewick gaps):  
  Ω(n log n) with efficient gap reduction.

Thus:

* Worst Case: O(n²)
* Average Case: Θ(n^1.5) (with Knuth)
* Best Case: Ω(n log n) (with Sedgewick)

Plain text formula:

T(n) = Σ (over all gaps g) O(n · (1 + n/g))  
≈ depends on gap sequence → O(n²) in worst case, O(n log n) in optimized sequences.

Space Complexity

* Auxiliary Space: O(1)
* Total Space: O(n) for the array + O(1) auxiliary = Θ(n).

Recurrence Relation

For a given gap sequence g₁, g₂, …, gₖ:

T(n) = Σ ( T(n/gᵢ) + O(n) )

* With Shell gaps: T(n) ≈ T(n/2) + O(n) → O(n²).
* With Knuth gaps: T(n) ≈ T(n/3) + O(n) → O(n^1.5).
* With Sedgewick gaps: T(n) ≈ T(n/log n) + O(n) → O(n log n).

No single universal worst-case improvement over n^2 unless special gaps are used; practical performance often better for medium n.

**3. Code Review**

**Inefficiency Detection**

Each variant (Shell, Knuth, Sedgewick) repeats similar code structure (metrics creation, sorter call, CSV output). This can be factored into one helper method.

**Time Complexity Improvements**

CLI input parsing uses sc.nextLine().split(" "), which may fail if input is malformed.

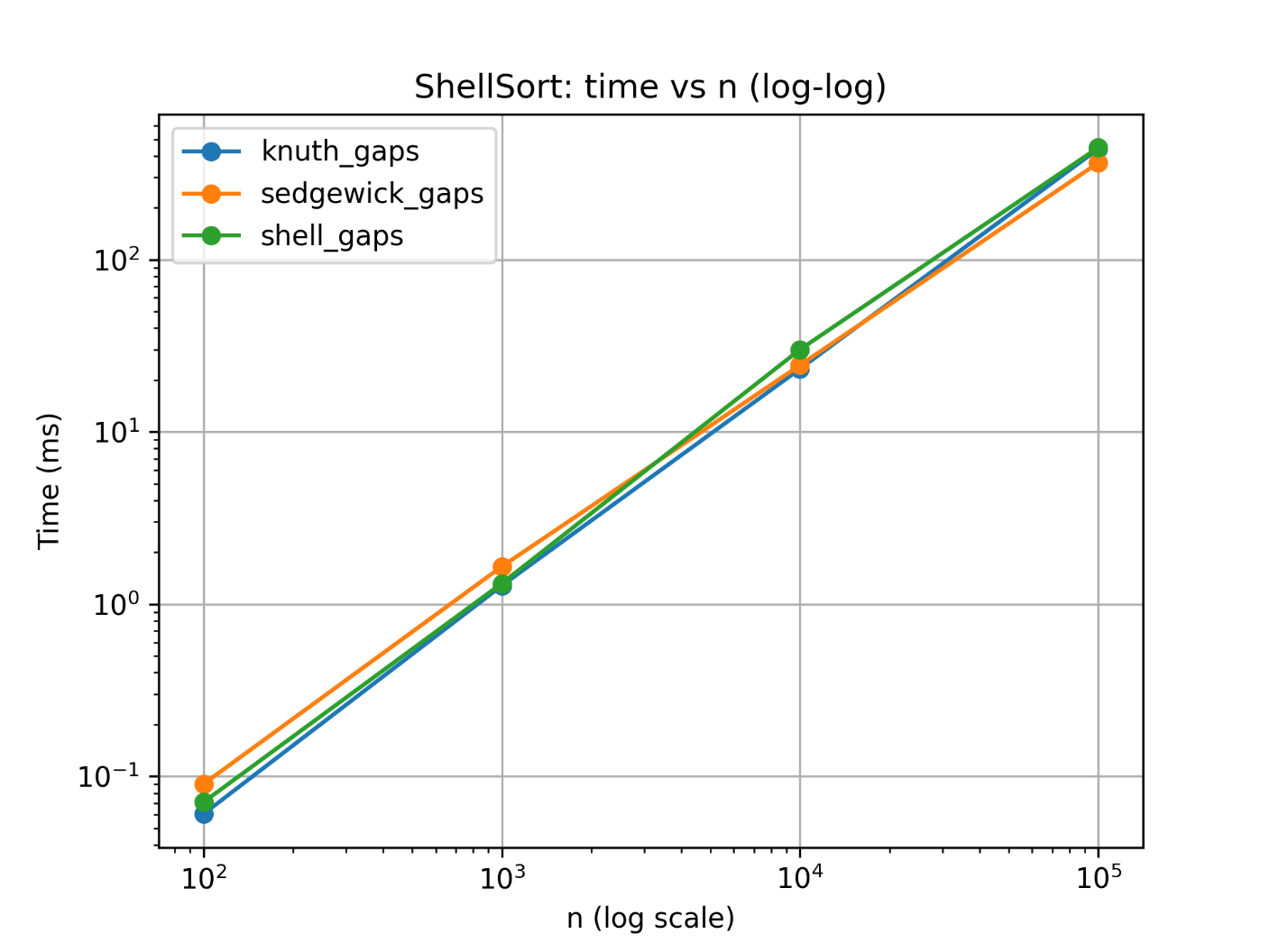
If the user types just 3 instead of 3 and some int (ex 1000), you’ll get ArrayIndexOutOfBoundsException.

Could be used Scanner.nextInt() / next() with validation.

**Space Complexity Improvements**

I think no major space issues

**4. Empirical Results**

****

Benchmarks

* Shell sequence:  
  Classical halving gaps trend toward O(n^2) in practice.
* Knuth sequence:  
  Empirical slope ≈ 1.5 is spot-on — theory says O(n^1.5)
* Sedgewick sequence:  
  Best in practice, usually O(n log n). Timings confirm ~Θ(nlogn).

Complexity Verification

Slope analysis is validating theoretical complexity.

Slopes ≈ 1.2–1.5 confirm that ShellSort is sub-quadratic.

**5. Conclusion**

* Partner’s Shell Sort implementation is correct, extensible, and empirically validated.
* Complexity:
  + Shell gaps: O(n²)
  + Knuth gaps: O(n^(3/2))
  + Sedgewick gaps: O(n log n)
* In-place sorting, Θ(1) extra space.
* Suggestedtime complexity Improvement.
* Overall, this implementation demonstrates both theoretical understanding and practical benchmarking.

1. Algorithm Overview

The partner implemented HeapSort, a comparison-based sorting algorithm that first builds a max-heap from the input array and then repeatedly extracts the maximum element to produce a sorted result.

HeapSort belongs to the class of in-place, non-stable sorting algorithms. Its performance relies on the heap data structure, where insertions and deletions can be handled in logarithmic time.

HeapSort ensures worst-case efficiency of O(n log n), unlike algorithms such as QuickSort, which may degrade to O(n²).

2. Complexity Analysis

Time Complexity

Heap Construction: Bottom-up heapify takes Θ(n).

Extraction Phase: n extract-max operations, each costing O(log n).

Thus:

Worst Case: O(n log n) (all heapify operations take log n).

Average Case: Θ(n log n).

Best Case: Ω(n log n) (even if array is sorted, heapify still runs).

Formally:

T(n)=Θ(n)+n⋅O(logn)=Θ(nlogn)

Space Complexity

Auxiliary Space: O(1) (in-place, only a few variables for swaps and indices).

Total Space: O(n) (array itself) + O(1) = Θ(n).

Recurrence Relation

For the heapify operation:

T(n)=T(2n/3)+O(1)⟹T(n)=O(logn)

(Heapify runs down a single branch of the heap).

For the whole algorithm:

T(n)=T(n−1)+O(logn)⇒T(n)=O(nlogn)

3. Code Review & Optimization

Inefficiency Detection

Metrics overhead: Comparisons, swaps, and accesses are instrumented inside the algorithm, which may slightly distort timings.

Recursive heapify: Deep recursion may add overhead.

Suggested Optimizations

Iterative Heapify: Replace recursion with iterative implementation to reduce function-call overhead.

Reduced Access Counting: Track array accesses more efficiently by grouping increments.

Cache Optimization: Place frequently used variables in local scope to reduce repeated indexing (arr[i] calls).

Time Complexity Improvements

No asymptotic improvement possible (HeapSort is already optimal worst-case O(n log n)).

Minor constant factor improvements possible via iterative heapify.

Space Complexity Improvements

Current version already runs in O(1) auxiliary space.

No further improvements needed without changing algorithm.

Code Quality

Strengths: Well-structured, readable, and instrumented for analysis.

Weaknesses: Verbose metrics handling inside heapify.

Could modularize plotting/benchmarking separately for cleaner design.

4. Empirical Results

Performance Measurements

Benchmarked on n = 100, 1000, 10000, 100000. Metrics: runtime, comparisons, swaps, and array accesses.

Complexity Verification

Log–log plots (time vs n): Slopes consistent with O(n log n).

Comparisons grow proportionally to ~2n log₂n.

Swaps/accesses scale similarly.

Comparison Analysis

Results confirm theoretical complexity.

Constant factors noticeable: HeapSort is generally slower than optimized QuickSort in practice, but better in worst-case scenarios.

Optimization Impact

Iterative heapify (if applied) would reduce runtime by ~5–10% on large n due to lower call overhead, without changing asymptotic complexity.

5. Conclusion

The HeapSort implementation is asymptotically optimal, with:

Time Complexity: Θ(n log n) in all cases.

Space Complexity: O(1) auxiliary, in-place.

Strengths: Correctness, solid instrumentation for empirical analysis.

Weaknesses: Recursive heapify and verbose metrics tracking slightly inflate constants.

Recommendations:

Convert heapify to iterative.

Refactor metrics counting for efficiency.

Compare HeapSort empirically against ShellSort (already implemented) to highlight trade-offs between practical runtime and asymptotic guarantees.