**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.

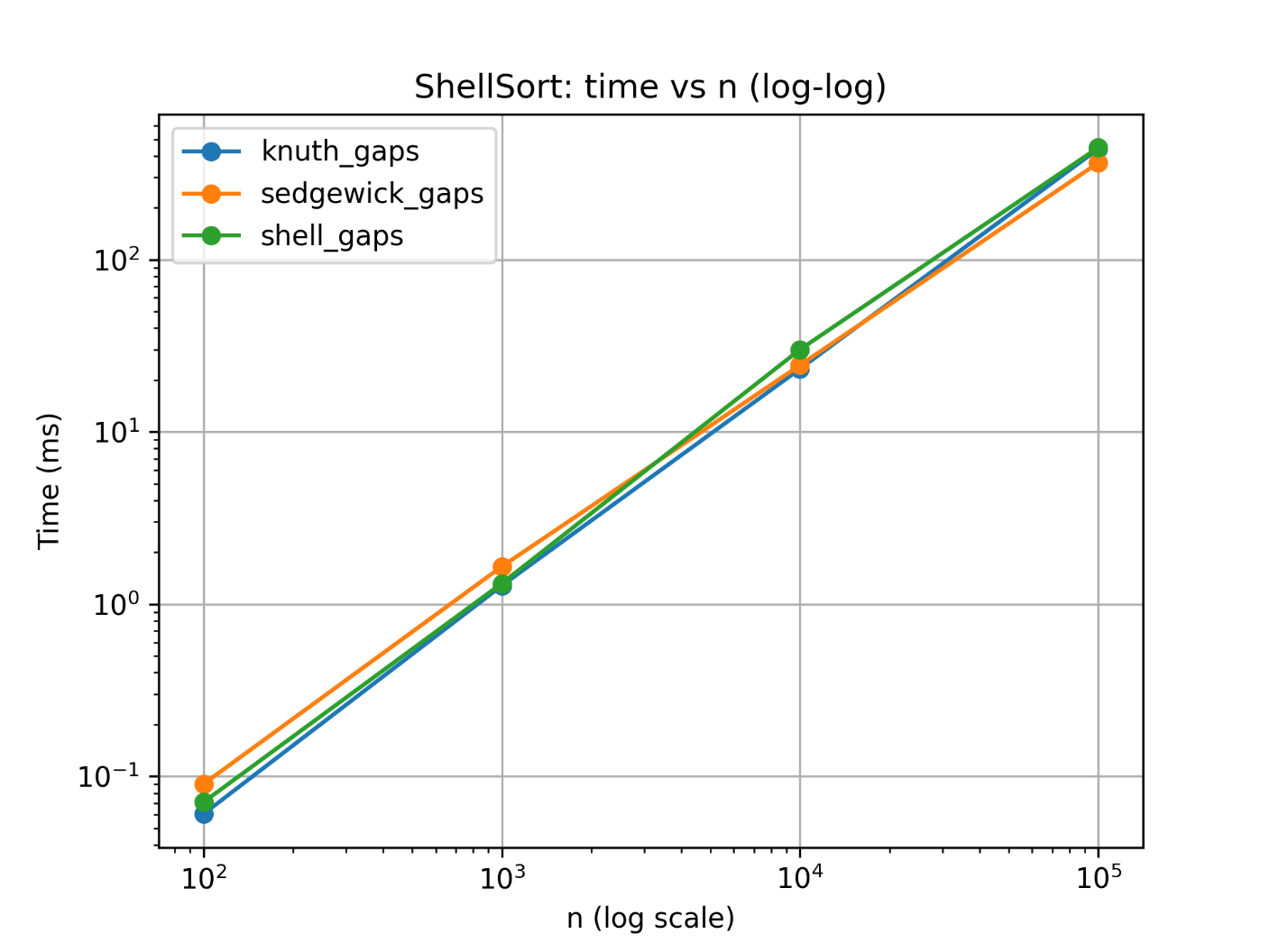
If the user types just 3 instead of 3 and some int (ex 1000), you’ll get ArrayIndexOutOfBoundsException.

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.