Cross-Review Summary (Shell Sort vs. Heap Sort)

Student A: (Zhandos Gilazhev / 4)

Student B: (Shynar Balgatay / 5)

Pair: 2

Algorithm 1 (Shynar's): Heap Sort

Algorithm 2 (Zhandos's): Shell Sort (Sedgwick, Knuth, Shell)

1. Introduction and Goal

The goal of this final document is to compare our two algorithms, Shell Sort (Student A) and Heap Sort (Student B), to determine which one is better for different kinds of tasks.

We compare them based on two main criteria:

1. Reliability (Theory): What is the guaranteed worst-case speed?
2. Practical Speed (Empirical): Which one is faster when running real tests?

2. Theoretical Complexity Comparison Table (Formal Big-O, Θ, Ω)

Case	**Shell Sort (Knuth Sequence)**	**Heap Sort**	**Comparison**
Best Case (Ω)	Ω(n log n)	Ω(n log n)	Equal — both require at least log n comparisons per element
Average Case (Θ)	Θ(n^{3/2})	Θ(n log n)	Shell Sort slightly slower asymptotically, but often faster in practice
Worst Case (O)	$O(n^2)$	O(n log n)	Heap Sort guaranteed better upper bound
Space Complexity	O(1)	O(1)	Both in-place
Data Sensitivity	High	None	Heap Sort stable under any input
Cache Locality	Good	Poor	Shell Sort wins in practical speed

3. Speed and Reliability Comparison

The main difference between these algorithms is **what they guarantee** versus **how fast they actually run.**

Criterion	Shell Sort (Code - Knuth Sequence)	Heap Sort	Conclusion
Worst-Case Guarantee	O(n4/3) or O(n2) (No strong O(nlogn) guarantee)	O(nlogn) (Strictly Guarantee d)	**Heap Sort** is safer and more reliable
Average Speed (Empirica l)	Fastest (Knuth Sequence)	Slower in the average case	**Shell Sort** is faster in practice
Memory Usage	O(1) (In-place)	O(1) (In-place)	They are equal

Key Takeaway: Heap Sort gives better **theoretical security**, but Shell Sort (specifically using the **Knuth** sequence, which proved fastest) is better for practical speed.

4. Visual Comparison Discussion (Graphs Summary)

[Graphs:

Shell Sort: assignment_2-shellsort\docs\performance-plots

Heap Sort: assignment2-heapsort\docs\performance-plots]

For smaller input sizes, Shell Sort (especially Knuth’s sequence) shows faster execution time and fewer memory accesses.

As input size increases, Heap Sort’s time curve grows more predictably, confirming its **O(n log n)** scalability, while Shell Sort’s curves fluctuate depending on the gap sequence.

This visual pattern supports the analytical conclusion that Shell Sort performs better in practice for **mid-range data**, while Heap Sort provides superior **worst-case** guarantees.

5. Bottleneck Analysis (Why They Slow Down)

We identified why each algorithm struggles in certain situations:

- **Shell Sort's Bottleneck:**
    - The Problem: Speed is data-dependent. If we use a bad gap sequence, the algorithm can suddenly degrade to O(n2).

- **Heap Sort's Bottleneck:**
    - The Problem: Poor Cache Locality (Memory Jumping). The algorithm constantly forces the CPU to "jump" across the memory to find the next element in the heap structure.
    - The Effect: These jumps cause Cache Misses, which adds time penalties. This high constant factor makes Heap Sort slower than Shell Sort on average.

6. Optimization and Final Recommendation

We proposed and tested an optimization for Heap Sort to improve its practical speed:

Optimization	Core Idea	Result (Insert your measured data)
"Floating Element" Technique	Replace many expensive **Swaps** (3 memory operations) with fewer, faster **Shifts** (1 memory operation) in the **heapify** function.	**Speed increase of X% and Y% reduction in memory accesses.**

Final Conclusion:

- **Choose Heap Sort if...** you need **absolute reliability** and a guarantee that the running time will never exceed O(nlogn) (e.g., for mission-critical systems).

- **Choose Shell Sort (Knuth) if...** you need **maximum speed** in the average case. It wins empirically because it handles the computer's memory more efficiently.