Comparative Analysis of HeapSort and ShellSort

1. Introduction

This report presents a comparative analysis of two sorting algorithms: **HeapSort**, implemented by student B, and **ShellSort**, implemented by student A. Both algorithms were benchmarked on arrays of varying sizes (`n = 10², 10³, 10⁴, 10⁵`) and multiple input distributions (`random`, `sorted`, `reversed`, `nearly-sorted`). Performance metrics include execution time, number of comparisons, swaps, and array accesses.

The goal is to evaluate theoretical complexity, practical performance, and identify scenarios where one algorithm outperforms the other.

2. Algorithm Overview

Property	HeapSort (Student B)	ShellSort (Student A)
Algorithm Type	Comparison-based, uses Max-Heap	In-place, gap-based improvement over insertion sort
Worst-case Time Complexity	Θ(n log n)	Depends on gap sequence ($O(n^2)$ for naive gaps, sub-quadratic for Knuth/Sedgewick)
Best-case Time Complexity	Θ(n log n)	Can approach O(n) with ideal gaps and nearly sorted data
Average-case Time Complexity	Θ(n log n)	Gap-dependent; practical performance often very good
Auxiliary Space	O(1) in-place (stack O(log n) if recursive)	O(1) in-place
Stability	Not stable	Not stable
Strengths	Predictable performance for large n, worst-case guarantees	Fast for medium/small n, nearly sorted arrays, benefits from cache locality
Weaknesses	Slightly more memory writes, cache-unfriendly	Worst-case sensitive to gap sequence, complexity can vary

3. Theoretical Comparison

- **HeapSort** guarantees O(n log n) for all input types and sizes. Its efficiency comes from using a binary heap to repeatedly extract the maximum element.
- **ShellSort** improves over insertion sort by moving elements long distances early. Its performance critically depends on the choice of gap sequence. Well-chosen sequences (Knuth, Sedgewick) perform exceptionally well for medium input sizes and nearly sorted arrays, while naive sequences can have $O(n^2)$ worst-case complexity.

Summary: HeapSort is asymptotically superior for large datasets, while ShellSort is often faster in practical cases for smaller or nearly-sorted arrays.

4. Empirical Results

Performance was benchmarked for both algorithms using the following distributions: random, sorted, reversed, and nearly-sorted. Metrics collected: **time (ns), comparisons, swaps, array accesses**.

- **Execution Time:**
  - HeapSort shows stable O(n log n) scaling across all distributions.
  - ShellSort performs best for nearly sorted and small-medium arrays, often outperforming HeapSort in practical runtime.
- **Operation Counts:**
  - HeapSort has consistent comparison counts proportional to n log n.
  - ShellSort's comparisons and swaps vary significantly with input distribution and chosen gap sequence.

Observation: ShellSort benefits from nearly sorted inputs and good gap sequences, while HeapSort maintains stable performance regardless of input type.

5. Practical Implications

Input Type	Preferred Algorithm	Reason
Small n, nearly sorted	ShellSort	Fewer moves, better cache locality
Large n	HeapSort	Predictable runtime, worst-case guarantees
Random input	Depends on gap sequence	ShellSort can be competitive, HeapSort stable

Reversed input	HeapSort	ShellSort may approach worst-case if gaps not optimized

6. Conclusion

- HeapSort offers **predictable O(n log n) performance** across all input types, making it suitable for large-scale data and worst-case sensitive applications.
- ShellSort provides **excellent practical performance** for small-medium datasets and nearly sorted inputs, but its worst-case performance depends on gap sequences.
- The combination of **theoretical analysis and empirical results** validates the strengths and weaknesses of each algorithm: HeapSort for stability and predictability, ShellSort for practical speed and efficiency in specific scenarios.