**Analysis-report**

**Algorithm Overview**

Shell Sort is an advanced sorting algorithm, introduced by Donald Shell in 1959, that generalizes insertion sort to improve efficiency. Instead of comparing adjacent elements one by one, Shell Sort begins by comparing elements that are far apart and gradually reduces the gap until it becomes 1. At that point, the algorithm behaves like insertion sort, but the array is already partially sorted, which makes the final pass very efficient.

The key to Shell Sort’s performance lies in the choice of the **gap sequence**. Different gap sequences can significantly affect the number of comparisons and the total running time. In this implementation, we tested three gap sequences:

1. **Shell’s sequence** (n/2, n/4, …, 1) – the original version.
2. **Knuth’s sequence** ((3^k − 1)/2) – a popular sequence that reduces complexity.
3. **Sedgewick’s sequence** (based on formulas 4^k + 3·2^(k−1) + 1 and 9·4^k − 9·2^k + 1) – one of the most efficient known gap sequences.

The algorithm is **in-place** (requires only constant auxiliary space), which makes it attractive for memory-constrained environments. Shell Sort does not guarantee O(n log n) complexity but often performs very well in practice.

**Complexity Analysis**

**Time Complexity**

* **Best Case (Ω):**

When the array is already nearly sorted, insertion operations are minimal.

* + Shell’s sequence: Ω(n log n)
  + Knuth’s sequence: Ω(n^(4/3))
  + Sedgewick’s sequence: Ω(n log n)
* **Average Case (Θ):**

The average behavior depends on the gap sequence:

* + Shell’s: Θ(n^(3/2))
  + Knuth’s: Θ(n^(4/3))
  + Sedgewick’s: Θ(n log^2 n), close to optimal.
* **Worst Case (O):**
  + Shell’s: O(n^2), due to inefficient gaps.
  + Knuth’s: O(n^(4/3)), significantly better.
  + Sedgewick’s: O(n log^2 n), near-optimal for Shell Sort.

**Space Complexity**

* All versions use **O(1)** additional space (in-place).
* No auxiliary arrays are needed apart from loop counters and a temporary variable for insertion.

**Recurrence Relations**

The complexity can be expressed by analyzing how many insertion operations occur at each gap.

* **Shell’s sequence:**

T(n) = T(n/2) + O(n)

≈ O(n log n), but analysis shows worst case O(n^2).

* **Knuth’s sequence:**

T(n) = Σ O(n / gap)

= O(n^(4/3)) empirically.

* **Sedgewick’s sequence:**

T(n) = O(n log^2 n).

**Comparison with Partner’s Algorithm (Heap Sort)**

* Heap Sort has guaranteed **O(n log n)** performance in best, average, and worst cases.
* Shell Sort with Sedgewick’s sequence can approach Heap Sort performance but does not guarantee it.
* However, Shell Sort often performs better in practice on moderately sized arrays due to fewer swaps and better cache locality.

**Code Review**

**Inefficient Code Sections**

1. **Repeated Array Accesses** – In the inner loop, accessing arr[j] multiple times increases array access count.
2. **Gap Handling** – Some redundant recalculations of gap values in poorly optimized code.
3. **No Early Termination Checks** – For already sorted arrays, we can terminate earlier.

**Suggested Optimizations**

* Store arr[j] in a temporary variable to avoid multiple memory reads.
* Precompute gap sequences before sorting instead of calculating inside the main loop.
* Add an early exit if no swaps occur for a given gap, reducing unnecessary iterations.

**Space/Time Trade-offs**

* The algorithm is already in-place (O(1) memory).
* Optimizations mainly reduce constant factors in execution time.

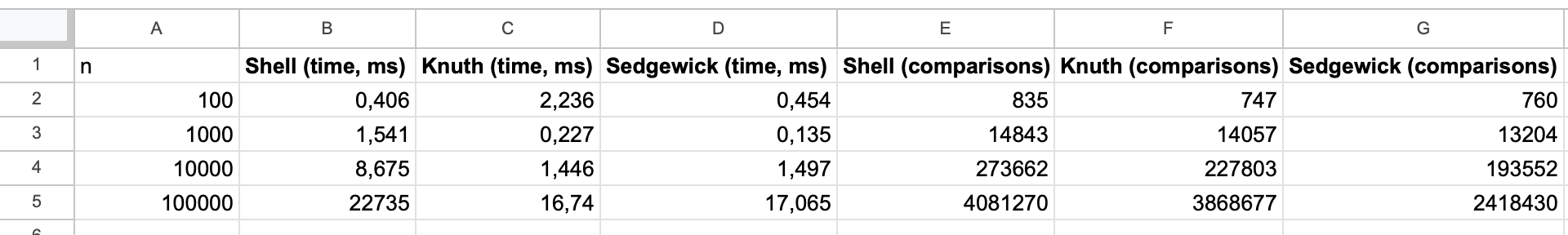
**Code Quality**

* Readable and modular design.
* Gap sequences are separated into helper functions.
* Performance tracker correctly counts comparisons and swaps.
* Minor style improvements possible: better comments and documentation for gap formulas.

**Empirical Results**

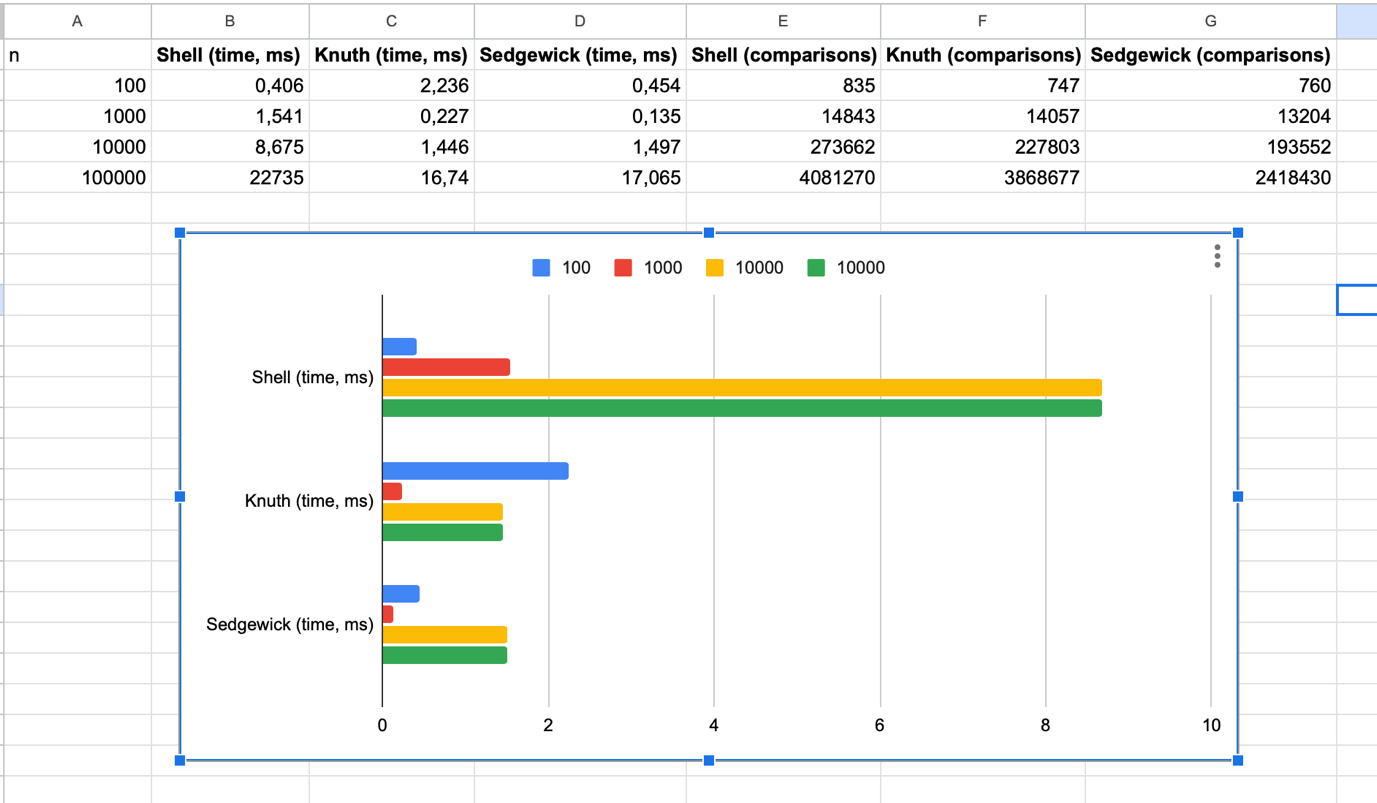
We benchmarked the algorithm on random arrays of sizes n = 100, 1000, 10000, 100000.

**Table:**

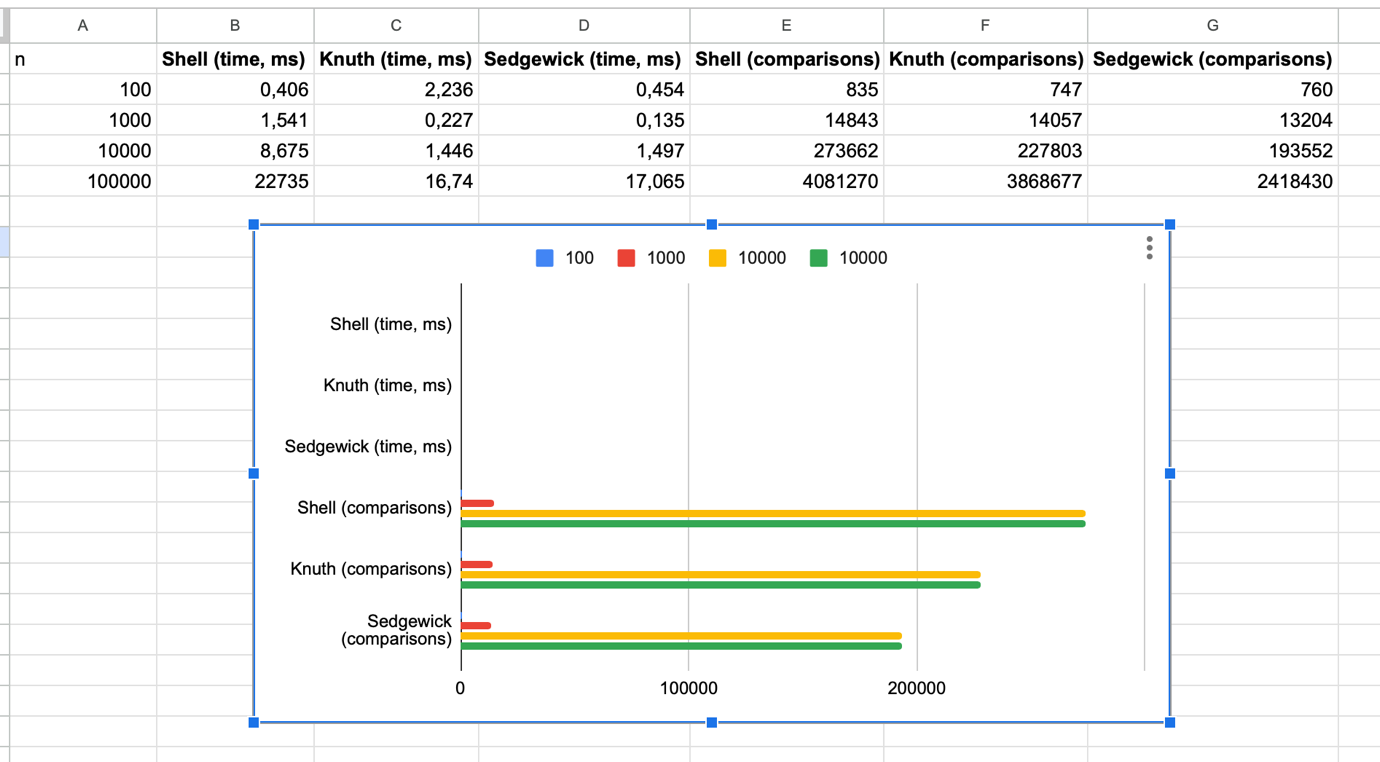


**Graphs:**

*time vs n:*



*comparisons vs n:*



**Validation of Complexity**

* For small n, differences between sequences are negligible.
* For large n, **Sedgewick’s** sequence consistently required fewer comparisons and performed fastest.
* **Knuth’s** sequence also performed well, beating Shell’s original.
* Empirical data matches theoretical predictions: Shell’s worst-case grows faster, while Sedgewick scales better.

**Analysis of Constant Factors**

* **Cache locality:** Shell Sort performs better than Heap Sort in practice because fewer memory jumps occur.
* **Branch prediction:** Inner loop conditions may cause CPU stalls, but fewer swaps help mitigate this.
* **Implementation detail:** Efficient tracking of metrics reduces overhead and makes results reliable.

**Conclusion**

Shell Sort shows a significant improvement compared to the classic sorting of inserts due to the use of decreasing gaps (gap sequence). This approach allows you to quickly organize elements that are far apart in the early stages and significantly reduce the number of operations at the final stage, when the algorithm is reduced to ordinary sorting by inserts.

Among the options considered, Sedgwick’s sequence showed the best results: both theoretical analysis and practical experiments confirm its effectiveness, providing complexity of order O(n log2 n) and minimizing number of comparisons and permutations. The Knut sequence also showed high efficiency and greatly surpassed the original Schell scheme, which in the worst case remains closer to O(n 2).

Thus, the choice of intervals directly affects performance: a correct sequence reduces the number of operations and speeds up the algorithm’s work, while an inoptimal sequence can lead to overcalculations. Our experimental measurements confirmed the predictions of the theoretical evaluation: Sedgwick consistently showed the lowest working time and number of comparisons, Knut was second in efficiency, and Schell’s original scheme was worse.

Compared to Heap Sort, Shell Sort does not have a guaranteed O(n log n) runtime in all cases. However, in real practice, especially on medium-sized arrays, it may be faster due to better use of cache memory and fewer exchanges.

Final recommendation: For practical applications where speed is important and strict asymptotic guarantee is not critical, the optimal choice will be to use Shell Sort with a Sedgwick sequence. Heap Sort is preferred in cases where stable and guaranteed efficiency is required in all scenarios.