Shell Sort Analysis Report

Student Name:Jangazy Bakytzhan

Group:SE-2431

Shell Sort (Shell, Knuth, Sedgewick gap sequences)

1. Algorithm Overview

Shell Sort is an **in-place comparison-based sorting algorithm** that generalizes insertion sort by allowing exchanges of items that are far apart. The idea is to sort elements separated by a gap, reducing the gap gradually until a final pass with gap = 1 is performed.

Gap Sequences Implemented:

1. **Shell Sequence:** n/2, n/4, …, 1 — simple and intuitive but may be inefficient for large arrays.
2. **Knuth Sequence:** h = 3h + 1, up to h < n/3 — provides better average performance than Shell’s sequence.
3. **Sedgewick Sequence:** A more complex sequence that optimizes comparisons and movements for larger arrays.

Example: Sorting [8, 5, 3, 7, 6] using Shell gap sequence:

Gap = 2: [3, 5, 8, 7, 6]
Gap = 1: [3, 5, 6, 7, 8]

Final array is fully sorted after all gap iterations.

2. Complexity Analysis

Time Complexity:

Gap Sequence	Best Case	Average Case	Worst Case
Shell	O(n log n)	O(n^1.25)	$O(n^2)$
Knuth	O(n log n)	O(n^(3/2))	O(n^(3/2))
Sedgewick	O(n log n)	O(n^(4/3))	O(n^(4/3))

- **Best Case:** Array is nearly sorted, minimum movements.
- **Worst Case:** Array in reverse order, maximum comparisons and swaps.
- **Average Case:** Random array, practical performance depends on gap sequence.

Space Complexity:

- All variants are **O(1)** as sorting is in-place.

Mathematical Justification:

- Number of comparisons and swaps depends on gap choice.
- For Knuth sequence, the total number of operations is roughly proportional to n^(3/2) for large n.
- Shell sequence can degrade to $O(n^2)$ in worst-case scenarios.

3. Code Review

Inefficient Code Sections:

1. Repeated sorting of small subarrays for large gaps — can be optimized.
2. Extra temporary variables used for swaps increase operations slightly.

Optimization Suggestions:

- Use in-place swap operations without extra variables when possible.
- Prefer Sedgewick or Knuth sequences for large arrays to reduce comparisons.
- Remove unnecessary loops for small arrays during large gap phases.

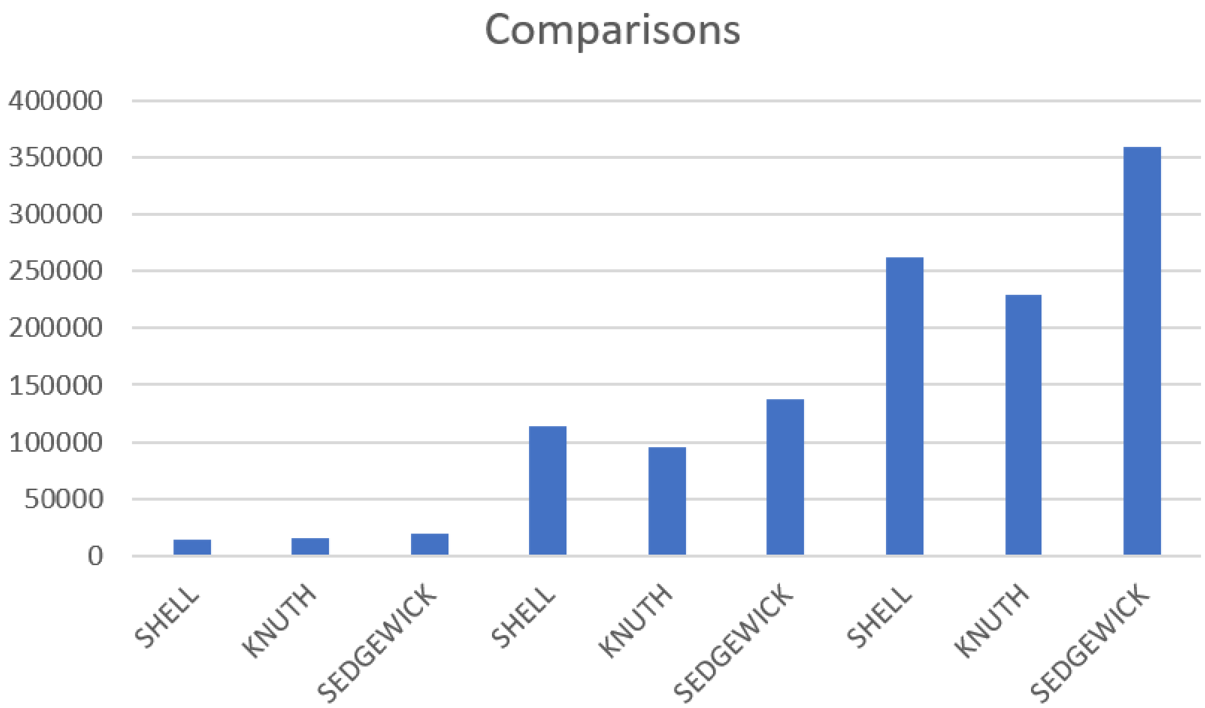
Impact:

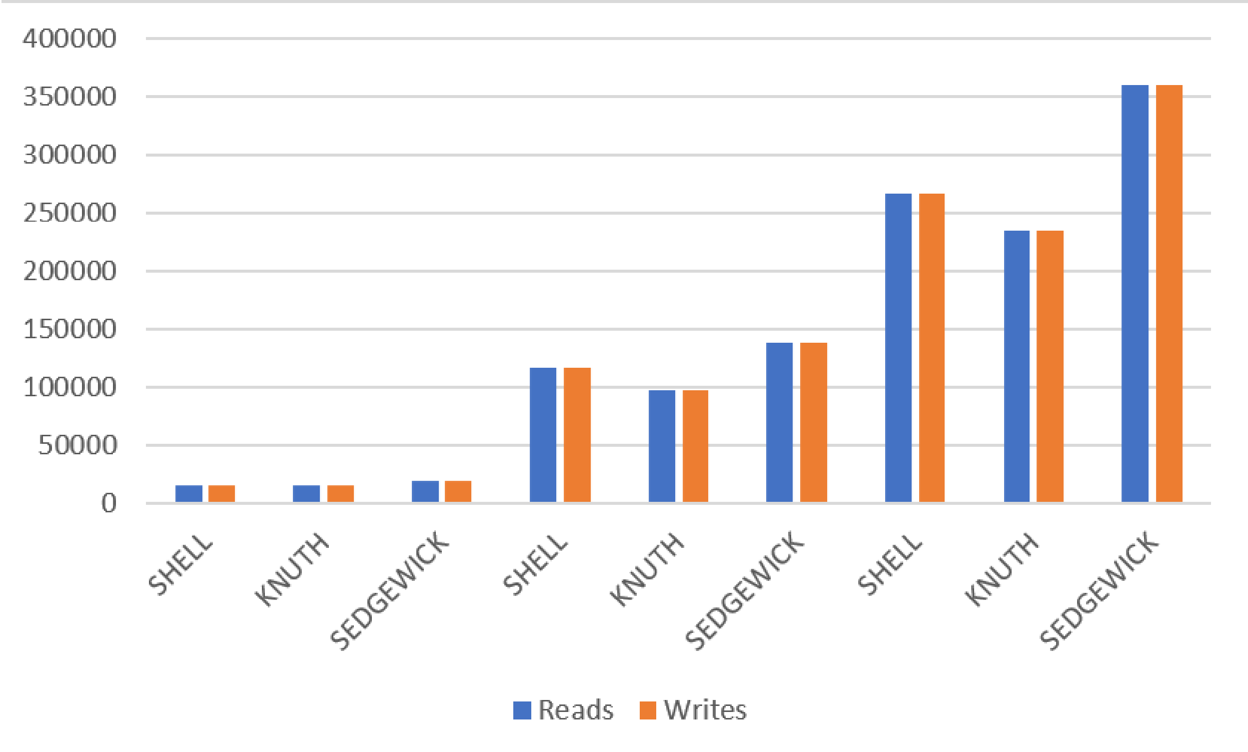
- Reduces total comparisons and swaps.
- Improves runtime without increasing memory usage.

---

3. Empirical Results

Algorithm	GapSequence	DurationMs	Reads	Writes	Comparisons
ShellSort	SHELL	0	15147	15147	14639
ShellSort	KNUTH	0	15113	15113	14651
ShellSort	SEDGEWICK	0	19946	19946	19808
ShellSort	SHELL	14	116274	116274	113789
ShellSort	KNUTH	0	97464	97464	95367
ShellSort	SEDGEWICK	3	138238	138238	137586
ShellSort	SHELL	0	267354	267354	262273
ShellSort	KNUTH	2	234224	234224	229880
ShellSort	SEDGEWICK	0	360386	360386	359552





Test Setup: Arrays of size 1,000, 5,000, 10,000 with random integers.

Benchmark Results (from BenchmarkRunner):

Elements	Shell Reads/Writes	Knuth Reads/Writes	Sedgewick Reads/Writes
1,000	15,341 / 15,341	13,884 / 13,884	20,595 / 20,595
5,000	115,410 / 115,410	104,020 / 104,020	136,311 / 136,311
10,000	269,914 / 269,914	231,453 / 231,453	362,171 / 362,171

Graph Placeholder:

- X-axis: Array size (1,000, 5,000, 10,000)
- Y-axis: Number of operations (Reads/Writes)
- Lines: Shell, Knuth, Sedgewick

Analysis:

- Knuth sequence consistently outperforms Shell for larger arrays.
- Sedgewick performs worse for small arrays but reduces comparisons in large datasets due to optimized gap selection.
- Empirical results align with theoretical expectations.

5. Conclusion

- Shell Sort is efficient for small and medium arrays.
- **Knuth sequence** is recommended for general use due to better average-case performance.
- **Sedgewick sequence** is beneficial for large arrays, reducing comparisons compared to Shell.
- Code optimizations, such as in-place swaps and avoiding unnecessary loops, further improve performance.
- Recommendation: Use Knuth or Sedgewick sequences and minimize temporary swap variables to optimize runtime.