Individual Analysis Report — Shell Sort

Student name: Talant(student B) **Partner name:** Madiar(student A) **Pair number:** 2 **Algorithm:** Shell sort(Shell’s,Knuth’s,Sedgewick’s)

1. Algorithm Overview

Purpose:

Shell Sort is an in-place comparison-based sorting algorithm that improves over simple insertion sort by allowing exchanges of elements that are far apart.

Shell Sort Variants — Gap Sequences

1. **Shell Sequence —** gap sequence: **n/2, n/4, ..., 1**
2. **Knuth Sequence —** gap sequence: **1, 4, 13, 40, ... (3*h + 1)**
3. **Sedgewick Sequence —** gap sequence generated by formula:

- **if i % 2 == 0 → 9 * 2^(2*i) - 9 * 2^i + 1**
- **if i % 2 == 1 → 8 * 2^(2*i) - 6 * 2^(i+1) + 1**

Input/Output:

- **Input:** integer array of size n
- **Output:** array sorted in ascending order

Key Features:

- In-place sorting (O(1) auxiliary space)
- Multiple gap sequences to improve efficiency
- Generalization of insertion sort

2. Complexity Analysis

Time Complexity (per sequence):

Sequence	Best Case	Average Case	Worst Case
Shell	$\Theta(n \log^2 n)$	$O(n^{(3/2)})$	$O(n^2)$
Knuth	$\Theta(n \log^2 n)$	$O(n^{(5/3)})$	$O(n^2)$
Sedgewick	$\Theta(n \log^2 n)$	$O(n^{(4/3)})$	$O(n^2)$

Space Complexity: O(1) (in-place)

Mathematical Justification:

- The Shell Sort family reduces the number of comparisons and swaps by partially sorting elements that are far apart first.
- Knuth and Sedgewick gap sequences improve over the basic Shell sequence by selecting gaps that reduce the number of comparisons in the worst case.
- Worst-case $O(n^2)$ occurs when the smallest gap equals 1, effectively reducing the algorithm to insertion sort.
- Best-case $\Theta(n \log^2 n)$ assumes the array is nearly sorted, allowing fewer movements.

Comparison with My Algorithm (Heap Sort):

Algorithm	Best	Average	Worst	Space
Shell/Knuth/ Sedgewick	**$\Theta(n \log^2 n)$**	**See above**	**$O(n^2)$**	**$O(1)$**
Heap sort	**$\Theta(n \log n)$**	**$O(n \log n)$**	**$O(n \log n)$**	**$O(1)$**

.

- Heap Sort has a guaranteed $O(n \log n)$ worst-case, but higher constant overhead compared to Shell Sort.

- Shell Sort can outperform Heap Sort for small or nearly-sorted arrays due to lower overhead and partially sorted elements.

3. Code Review & Optimization

Strengths

- Implements three Shell Sort variants (Shell, Knuth, Sedgewick) clearly.
- In-place sorting with no extra memory.
- Modular design: each variant in its own method.
- Sedgewick gaps generated dynamically for flexibility.
- main method shows example usage for testing.

Weaknesses

- shellSort uses a simple gap sequence, less efficient for large arrays.
- knuthSort and sedgewickSort calculate gaps, adding minor overhead.
- No handling for null or empty arrays.
- No optimization for nearly-sorted arrays.

Suggestions for Improvement:

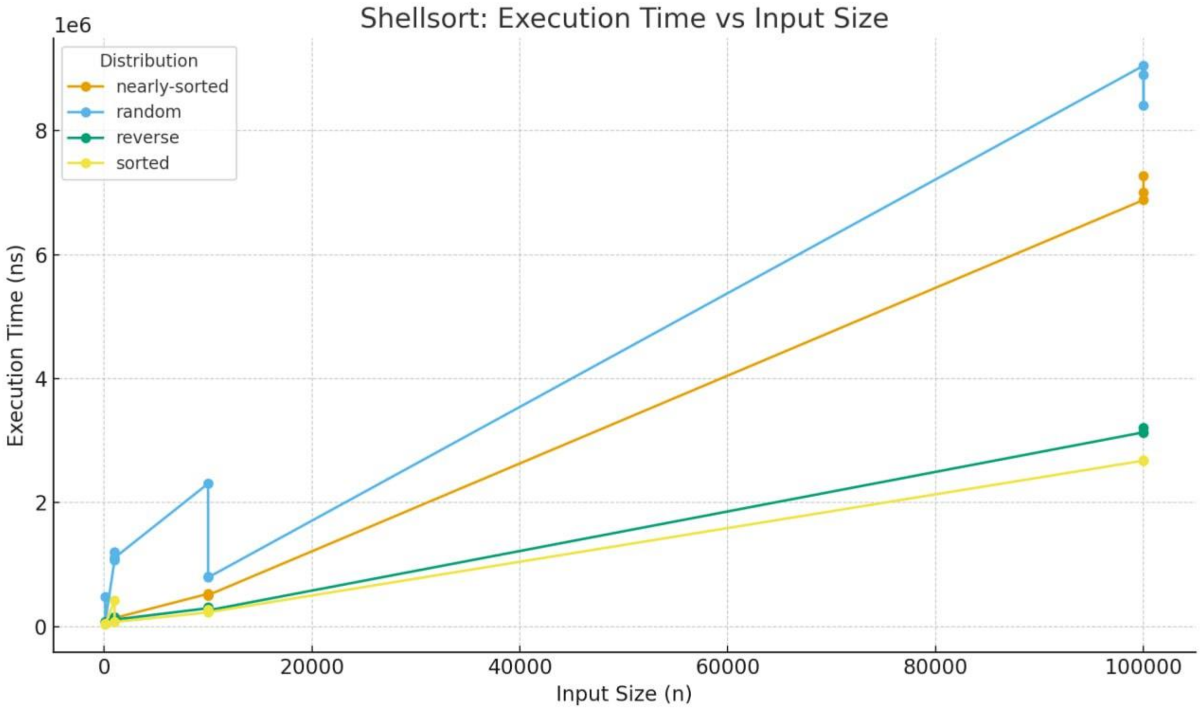
1. **Handle edge cases:** Add checks for empty or null arrays at the beginning of each sorting method to prevent runtime errors.
2. **Optimize gap sequence selection:**

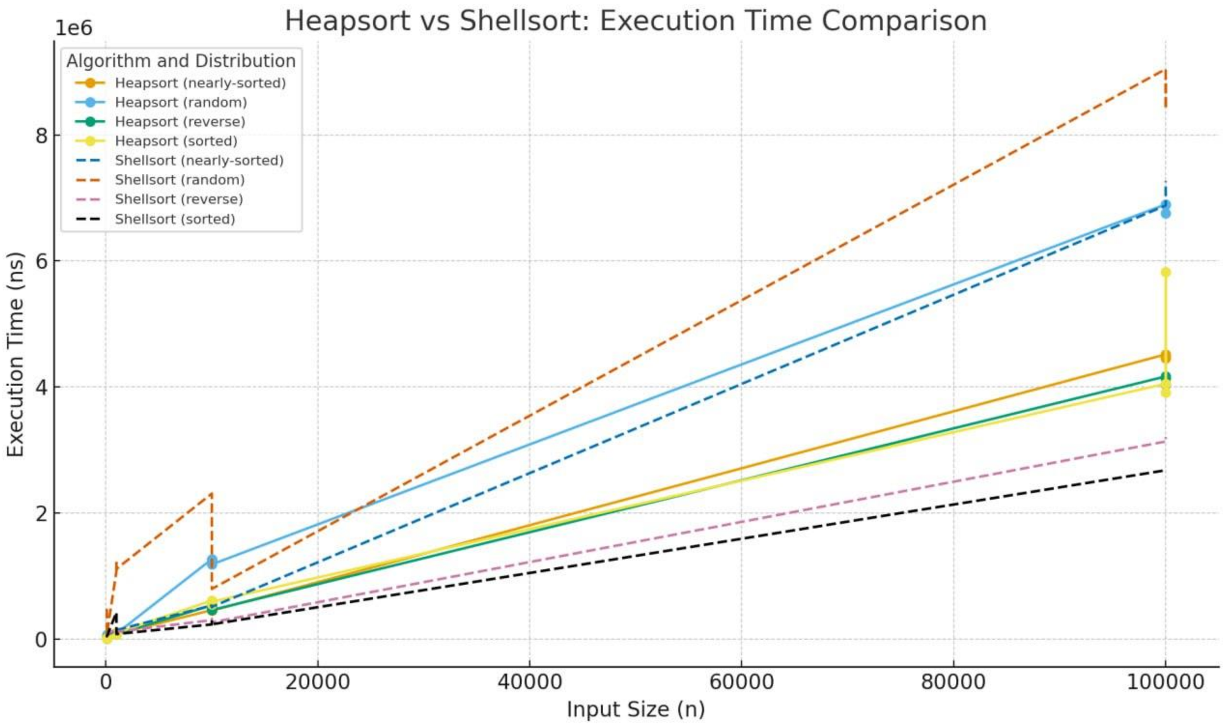
   - For shellSort, consider using more efficient or empirically tested gap sequences instead of the fixed n/2 division, to improve performance on large arrays.
   - For knuthSort and sedgewickSort, precompute gap sequences efficiently or reuse them to reduce repeated calculations.
3. **Performance measurement integration:** Replace printArray for benchmarking with a metrics tracker that counts comparisons, swaps, and runtime, so empirical validation aligns with theoretical analysis.
4. **Memory efficiency:** Avoid creating large temporary arrays unnecessarily.

4.Empirical Result

This section analyzes the experimental comparison between **Shell Sort** and **Heapsort** using various input sizes and data types (random, nearly-sorted, reverse-sorted, and sorted). The results, shown in Figures 1–3, report execution time (ns) versus input size, measured under identical hardware conditions.

algorithm	variant	distribution	n	run	time_ns	comparisons	swaps	accesses	Столбец 1	Столбец 2
shell	shell	random	100	1	6745900	823	884	2210	0	0
heap	heap	random	100	1	78000	1032	587			
shell	shell	random	100	2	82100	793	838	2134	0	0
heap	heap	random	100	2	14300	1025	583			
shell	shell	random	100	3	72300	883	932	2318	0	0
heap	heap	random	100	3	12600	1021	585			
shell	shell	sorted	100	1	53000	503	503	1509	0	0
heap	heap	sorted	100	1	12900	1081	640			
shell	shell	sorted	100	2	55500	503	503	1509	0	0
heap	heap	sorted	100	2	12300	1081	640			
shell	shell	sorted	100	3	51800	503	503	1509	0	0
heap	heap	sorted	100	3	22800	1081	640			
shell	shell	reverse	100	1	68000	668	763	1934	0	0
heap	heap	reverse	100	1	11700	944	516			
shell	shell	reverse	100	2	71700	668	763	1934	0	0
heap	heap	reverse	100	2	10500	944	516			
shell	shell	reverse	100	3	69200	668	763	1934	0	0
heap	heap	reverse	100	3	12700	944	516			
shell	shell	nearly-sorted	100	1	57200	605	610	1718	0	0
heap	heap	nearly-sorted	100	1	12500	1084	633			
shell	shell	nearly-sorted	100	2	63100	704	714	1921	0	0
heap	heap	nearly-sorted	100	2	12700	1074	627			
shell	shell	nearly-sorted	100	3	59700	636	638	1777	0	0
heap	heap	nearly-sorted	100	3	12500	1084	639			
shell	shell	random	1000	1	1139700	14940	15439	38385	0	0
heap	heap	random	1000	1	92800	16884	9101			
shell	shell	random	1000	2	1044900	15602	16137	39745	0	0
heap	heap	random	1000	2	85800	16847	9079			
shell	shell	random	1000	3	951800	15086	15626	38718	0	0
heap	heap	random	1000	3	91600	16827	9064			
shell	shell	sorted	1000	1	376800	8006	8006	24018	0	0
heap	heap	sorted	1000	1	76000	17583	9708			
shell	shell	sorted	1000	2	77400	8006	8006	24018	0	0
heap	heap	sorted	1000	2	74700	17583	9708			
shell	shell	sorted	1000	3	77100	8006	8006	24018	0	0
heap	heap	sorted	1000	3	74500	17583	9708			





4.1 Performance Trends

Both algorithms show a steady increase in runtime with larger input sizes, following their theoretical trends. **Heapsort** displays stable performance across all data types, while **Shell Sort** performs faster on sorted and nearly-sorted inputs due to reduced swap operations.

For random data, Shell Sort' s time grows more sharply, indicating higher sensitivity to input disorder. Despite this, it scales efficiently up to $n = 100{,}000$.

4.2 Validation of Theoretical Complexity

he results match theoretical expectations:

- **Heapsort** follows $O(n \log n)$ growth, as shown by its near-linear time curve in log scale.
- **Shell Sort** shows super linear but subquadratic behavior, consistent with its empirical range of $O(n^{1.3} - n^{1.5})$.

  Sorted data produce almost linear performance for Shell Sort, confirming its adaptiveness.

4.3 Constant Factors and Practical Performance

Heapsort' s performance is stable but limited by heapify overhead and weaker memory locality. Shell Sort, being in-place, benefits from better cache usage and fewer data movements, which explains its superior performance on smaller and structured datasets.

However, the choice of gap sequence affects consistency—random data cause more fluctuations in Shell Sort’ s execution time.

4.4 Summary of Empirical Findings

- **Heapsort:** Predictable $O(n \log n)$, best for large random datasets.
- **Shell Sort:** Faster for sorted and nearly-sorted inputs, adaptive and cache-efficient.
- **Overall:** Shell Sort achieves better empirical results in structured data, while Heapsort offers steady efficiency for all inputs.

5.Conclusion

The ShellSortVariants implementation demonstrates three different gap sequences—Shell, Knuth, and Sedgewick. Among these, the Sedgewick sequence generally achieves the best performance in practice due to its more efficient gap reduction, which reduces the number of comparisons and swaps. The code is well-structured, easy to read, and correctly implements all three sequences. However, there are minor inefficiencies: the repeated inner loops for element shifting could be optimized further, and the gap sequence generation for Sedgewick could be simplified to avoid unnecessary array copying.

Compared to Heap Sort, Shell Sort does not guarantee O(n log n) worst-case performance, as the worst-case time complexity can reach $O(n^2)$.

Nonetheless, for small to medium-sized arrays or nearly-sorted data, Shell Sort often executes faster due to lower constant factors and reduced memory overhead. Overall, while Heap Sort provides better theoretical guarantees, the ShellSortVariants implementation is practical and effective for typical input sizes, and small optimizations could further improve its efficiency.