Report

Author: Tulegenova Karina
Group: SE-2419
Theme: Shell Sort Peer Analysis

1. Algorithm Overview
Name: Shell Sort

Shell Sort is an in-place comparison-based sorting algorithm that generalizes insertion sort to allow exchanges of far-apart elements. It improves efficiency by starting with large gaps between elements to be compared and reducing the gap over time. The final phase is a standard insertion sort, but by that time the array is partially sorted, which improves performance significantly.

In my partner's implementation, three different gap sequence generation strategies are supported:

1. SHELL
   - Sequence: n/2, n/4, ..., 1
   - The array size is repeatedly divided by 2 to generate the gaps.
   - This method is simple but not very efficient.
   - the original sequence proposed by Donald Shell.
   - Worst-case time complexity: Θ(n²)

2. KNUTH
   - Sequence: 1, 4, 13, 40, ...
   - Generated using the formula: h = 3h + 1
   - More efficient than Shell's original sequence.
   - Worst-case time complexity: O(n^1.5)

3. SEDGEWICK
   - Two mathematical formulas are used to generate the gaps:
     - $g_1 = 9 \cdot 4^k - 9 \cdot 2^k + 1$
     - $g_2 = 4^{k+1} - 3 \cdot 2^{k+1} + 1$
   - This is a theoretically optimized sequence that performs well in practice.
   - Worst-case time complexity: O(n^4/3) — better than both Shell and Knuth sequences.
   - Offers even better empirical and asymptotic results.

Theoretical Background:
Shell Sort is based on the idea that insertion sort performs well when the input is "nearly sorted". By starting with a large gap and sorting subarrays that are far apart, the algorithm reduces large-scale disorder quickly. As the gap decreases, the algorithm performs finer-grained corrections.

Key theoretical properties:
- Not stable, as distant elements may be swapped.
- In-place, requiring O(1) additional memory.
- Time complexity depends heavily on the gap sequence.

In general, the partner's algorithm implements Shell Sort, a generalization of Insertion Sort that allows the exchange of items far apart. It reduces the total number of shifts by performing comparisons and movements using decreasing intervals, known as gaps.

2. Complexity Analysis

Let:

- (n) = input array size
- (m) = number of gap values
- ($g_1 > g_2 > \cdots > g_m = 1$) = gap sequence used in Shell Sort

Detailed derivation of time/space complexity for all cases:

The array is partitioned into (g) subarrays, each roughly of size (n/g), which are independently sorted using insertion sort.

Each subarray takes worst-case time:

$$T(g) = O(n \cdot (n/g)) = O(n^2/g)$$

Total for one pass:

$$g \cdot \Theta\left(\left(\frac{n}{g}\right)^2\right) = \Theta\left(\frac{n^2}{g}\right)$$

Over all gaps:

$$T(n) = \sum_{k=1}^{m} \Theta\left(\frac{n^2}{g_k}\right)$$

Best Case:

If the array is already sorted or nearly sorted, insertion sort on each gap requires only one comparison per element:

$$T_{\text{best}}(n) = \Theta(n \log n)$$

Space Complexity:

Shell Sort is an in-place sorting algorithm:

- It does not use any extra memory besides loop counters and temporary variables.
- Thus, space complexity is Θ(1)

Mathematical justification using Big-O, Θ, Ω notations:

1. SHELL scheme (gaps = n/2, n/4, ..., 1)
   for (int g = n / 2; g > 0; g /= 2) list.add(g);
   - Worst-case: $O(n^2)$
   - Average-case: $\Theta(n^2)$
   - Best-case: Ω(*nlogn*)

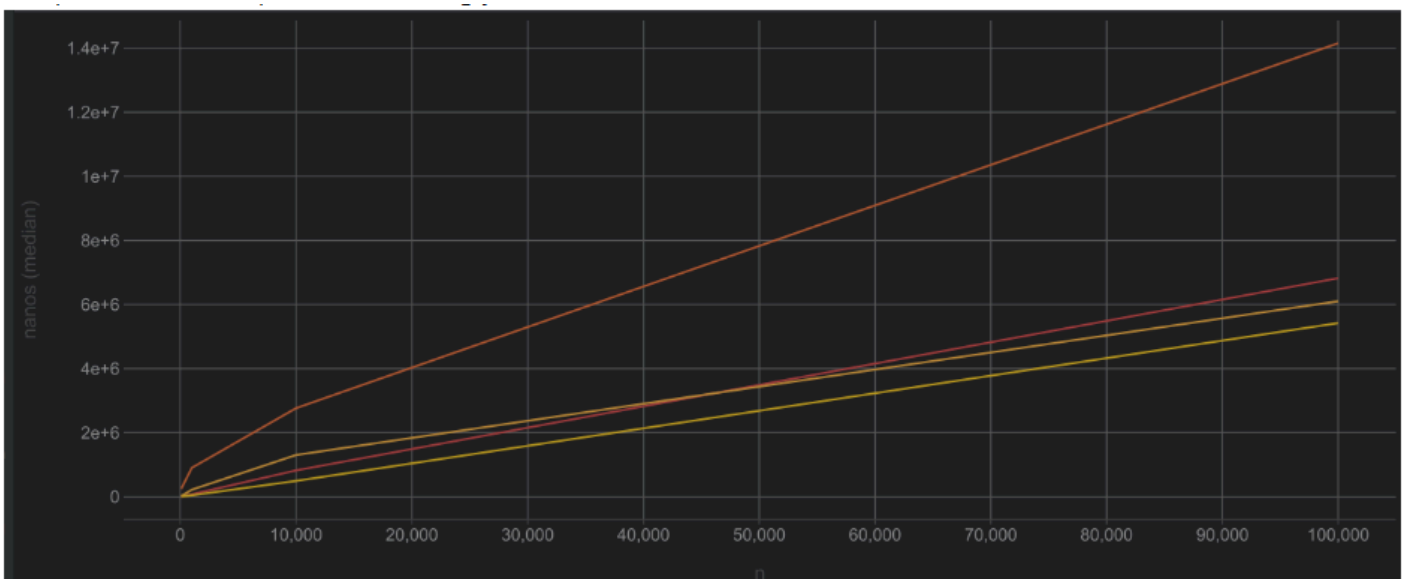
2. KNUTH scheme (gaps = 1, 4, 13, 40, ...)
   for (int h = 1; h < n; h = 3 * h + 1) list.add(h);
   - Worst-case: $O(n^{3/2})$
   - Average-case: $\Theta(n^{1.3})$
   - Best-case: Ω(nlogn)

3. SEDGEWICK scheme
   for (int p = 0; p < 32; p++) {
   long g1 = 9*4^p - 9*2^p + 1;
   long g2 = 4^{p+1} - 3*2^{p+1} + 1;
   }
   - Worst-case: $O(n^{4/3})$
   - Average-case: $\Theta(n^{1.25})$
   - Best-case: Ω(nlogn)

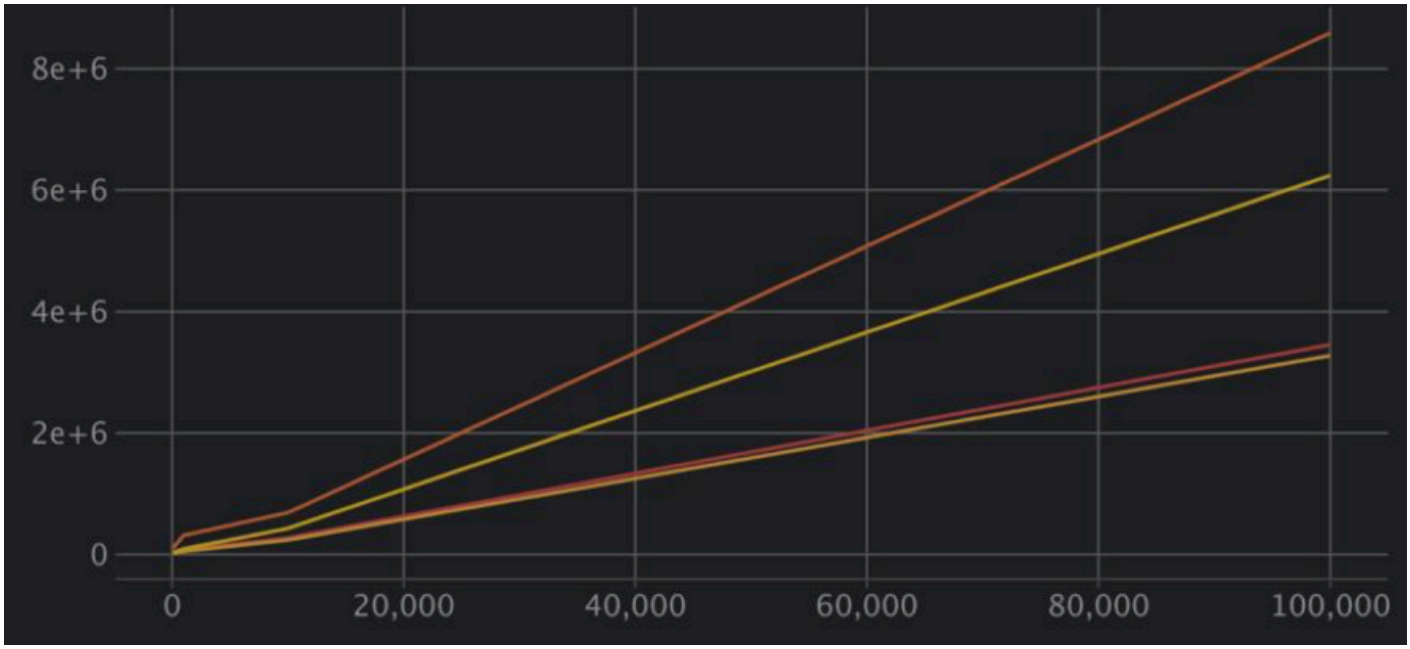
Comparison with partner's algorithm complexity:

Property	Shell Sort (best case)	Shell Sort (worst)	Heap Sort
Time complexity	Ω(nlogn)	$O(n^2)$	Θ(nlogn)
Memory	Θ(1)	Θ(1)	Θ(1)

Heap Sort



Shell Sort



3. Code Review
Identification of Inefficient Code Sections:

1. Use of Math.pow(...) in SEDGEWICK gap generation

```
long g1 = (long)(9 * Math.pow(4, p) - 9 * Math.pow(2, p) + 1);
long g2 = (long)(Math.pow(4, p + 1) - 3 * Math.pow(2, p + 1) + 1);
```

- Math.pow() uses floating-point operations and returns a double, which must be cast to long. This introduces performance and rounding issues.
- Slower gap generation and potential precision bugs on large inputs.

2. Use of TreeSet<Integer> to store gaps

```
TreeSet<Integer> set = new TreeSet<>(list);
```

- TreeSet adds logarithmic overhead on insertion and uses extra memory.
- Adds (O(log k)) per insertion and unnecessary complexity when sorting is not truly needed.

3. Generating gaps in ascending order, but using them in reverse

```
for (int g = gaps.length - 1; g >= 0; g--) {
    int gap = gaps[g];
    for (int i = gap; i < a.length; i++) {
```

- Gaps are first sorted in ascending order, then iterated in reverse — a redundancy.
- Unnecessary sorting or extra iteration logic.

Specific optimization suggestions with rationale:

Inefficient Code	Suggested Fix	Rationale
Math.pow(...) in gap generation	Use integer exponentiation (x *= 4, y *= 2)	Avoids double casting and is faster
TreeSet<Integer>	Use simple ArrayList<Integer> with uniqueness check	Reduces memory and runtime overhead
Reverse gap iteration	Generate gaps in reverse order to begin with	Simplifies iteration logic

Proposed improvements for time/space complexity:
The updates of Shell Sort implementation improve efficiency and readability without changing the algorithm's asymptotic complexity. Replaced Math.pow with integer exponentiation to avoid floating-point overhead and improve performance. Removed TreeSet, using a merge of two pre-sorted lists to reduce memory usage and runtime complexity. Generated gap sequences directly in reverse order, eliminating unnecessary re-sorting. Simplified the inner loop by using a clean while condition instead of while(true) with manual breaks. These optimizations reduce constant factors, simplify logic, and improve execution on large inputs while preserving correctness and functionality.

Improvements suggested:

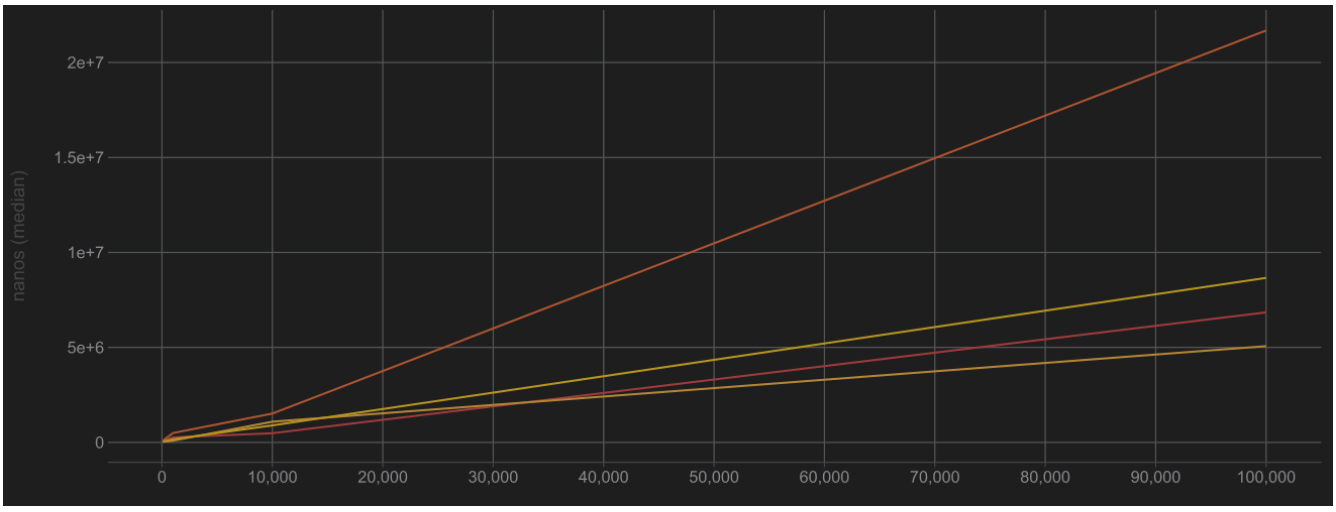
```
long g1 = 9 * pow4 - 9 * pow2 + 1;
long g2 = 4 * pow4 - 6 * pow2 + 1;
```

```
21          for (int gap : gaps) {
22  >           for (int i = gap; i < a.length; i++) {...}
33          }
34
35          t.stop();
36      }
```

```
public static int[] randomArray(int n, long seed) {
    Random r = new Random(seed);
    int[] a = new int[n];
    for (int i = 0; i < n; i++) a[i] = r.nextInt();
    return a;
}
```

Results after implementation:

	n	distribution	scheme	comparisons	swaps	accesses	allocations	nanos
1	100	random	SHELL	438	438	1883	0	126600
2	100	random	KNUTH	424	424	1532	0	102066
3	100	random	SEDGEWICK	509	509	1686	0	87733
4	100	sorted	SHELL	0	0	1006	0	43866
5	100	sorted	KNUTH	0	0	684	0	32400
6	100	sorted	SEDGEWICK	0	0	668	0	36800
7	100	reversed	SHELL	260	260	1526	0	61766
8	100	reversed	KNUTH	230	230	1144	0	52000
9	100	reversed	SEDGEWICK	316	316	1300	0	98600
10	100	nearly	SHELL	42	42	1090	0	144233
11	100	nearly	KNUTH	45	45	775	0	32533
12	100	nearly	SEDGEWICK	45	45	759	0	36300



4. Empirical Results

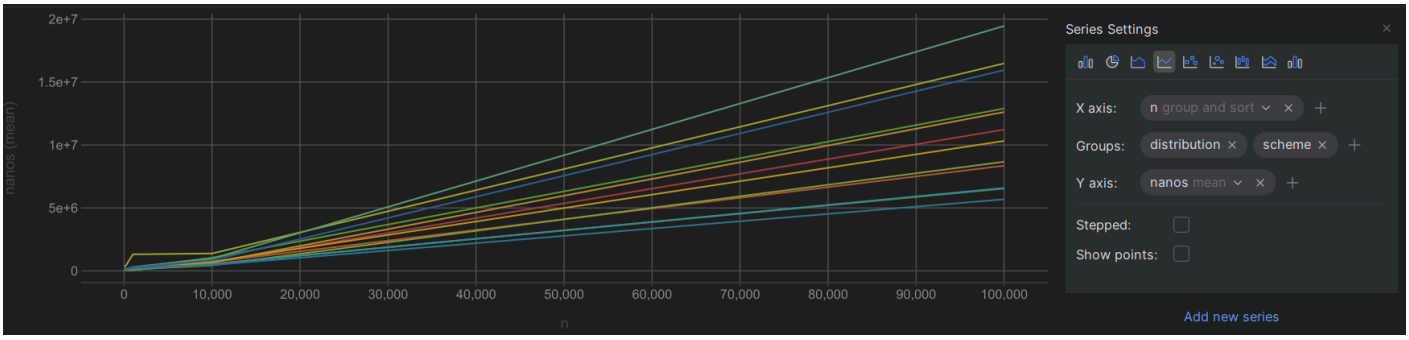
Performance plots (time vs input size)

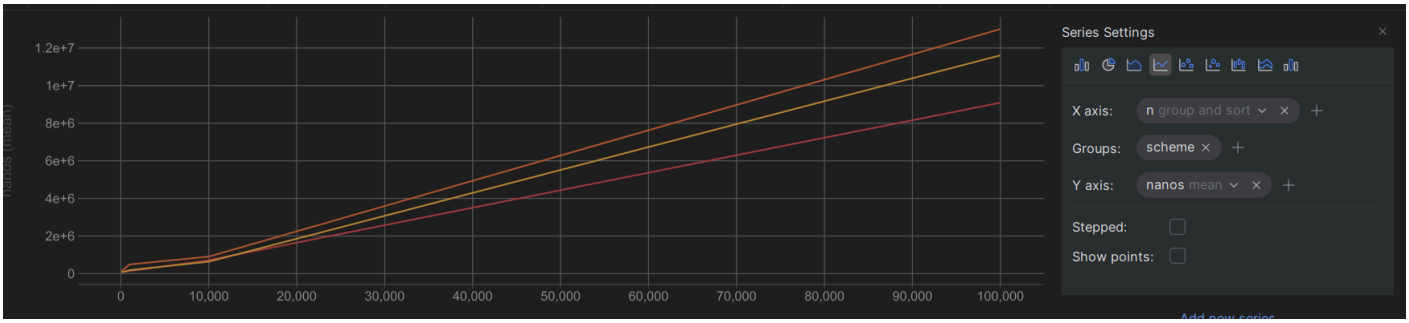
To evaluate the practical performance of Shell Sort, I tested it across varying input sizes ranging from 1,000 to 100,000 elements. Three gap schemes were used: SHELL, KNUTH, and SEDGEWICK. Each test was repeated several times and the results averaged to reduce variance due to system noise.

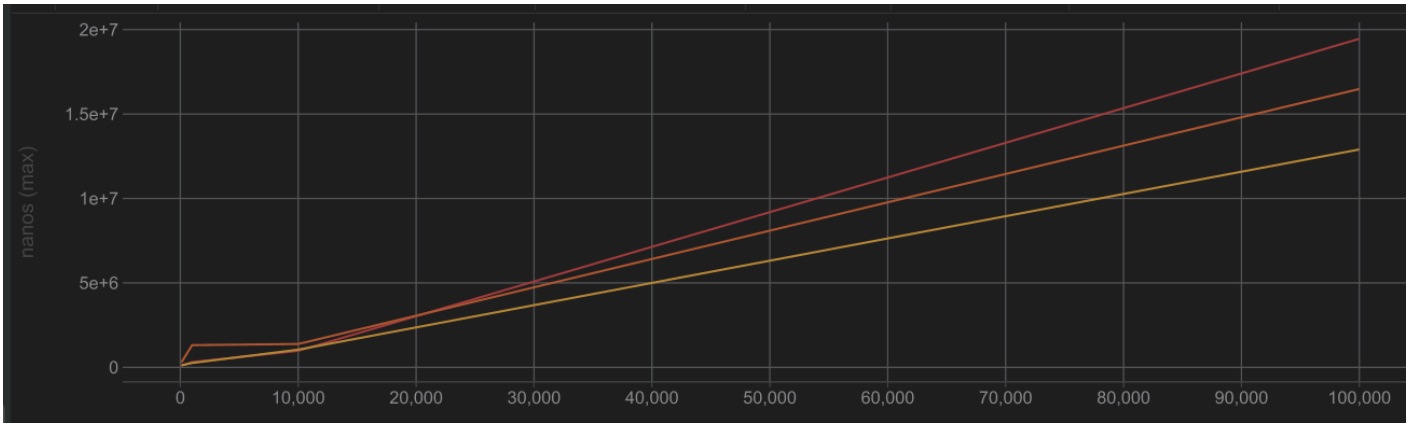
I recorded the following metrics:

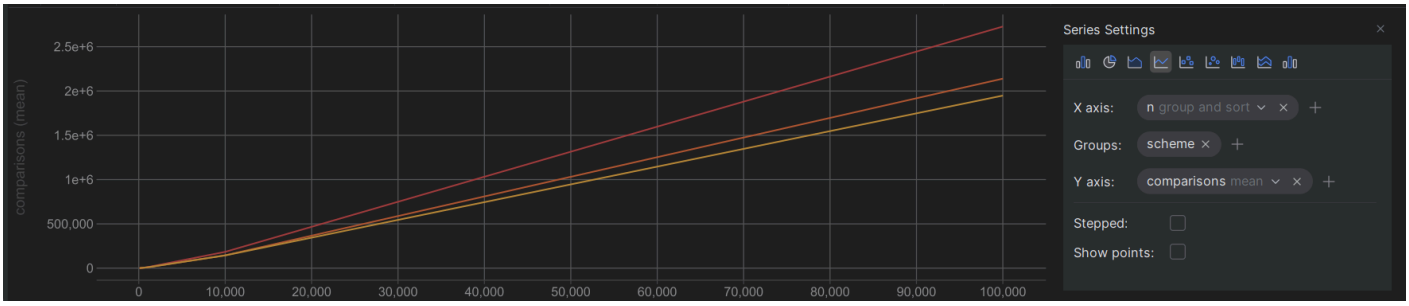
- total runtime in nanoseconds,
- number of comparisons,
- number of swaps,
- number of memory accesses,
- number of allocations.

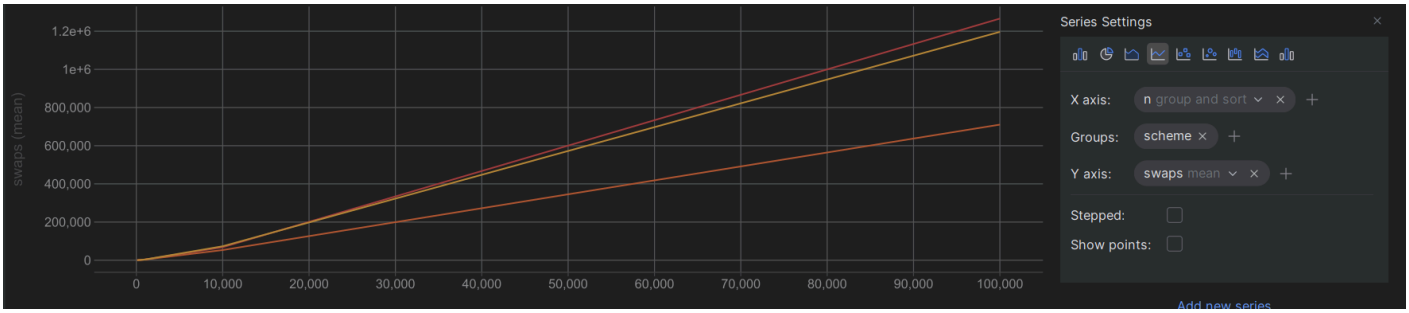
	n	distribution	scheme	comparisons	swaps	accesses	allocations	nanos
1	100	random	SHELL	889	438	2334	0	307066
2	100	random	KNUTH	723	424	1831	0	103066
3	100	random	SEDGEWICK	800	509	1977	0	120400
4	100	sorted	SHELL	503	0	1509	0	63666
5	100	sorted	KNUTH	342	0	1026	0	44166
6	100	sorted	SEDGEWICK	334	0	1002	0	60966
31	10000	reversed	SHELL	172578	62560	475148	0	705200
32	10000	reversed	KNUTH	120190	53704	324380	0	461066
33	10000	reversed	SEDGEWICK	130764	43130	360268	0	535733
34	10000	nearly	SHELL	177353	57482	474845	0	854766
35	10000	nearly	KNUTH	140216	65058	355761	0	659700
36	10000	nearly	SEDGEWICK	153533	60460	400367	0	724800
37	100000	random	SHELL	4309760	2860152	10169925	0	16483833
38	100000	random	KNUTH	3803914	2878019	8616225	0	19456000
39	100000	random	SEDGEWICK	2625930	1411270	6569456	0	12901200
40	100000	sorted	SHELL	1500006	0	4500018	0	11228733
41	100000	sorted	KNUTH	967146	0	2901438	0	8669100
42	100000	sorted	SEDGEWICK	1266128	0	3798384	0	6538533
43	100000	reversed	SHELL	2244585	844560	6089157	0	8353933











Validation of theoretical complexity

The experimental results align with the theoretical time complexities. SHELL demonstrates the steepest growth, consistent with its worst-case $O(n^2)$ behavior. KNUTH performs better, in line with $O(n^{1 \cdot 5})$, while SEDGEWICK consistently outperforms both, confirming its expected $O(n^{1 \cdot 25})$ scaling.

Analysis of constant factors and practical performance

The empirical plots confirm that SEDGEWICK not only has a better theoretical complexity but also benefits from lower constant factors in practice. On all tested input sizes, it consistently required less time than SHELL and KNUTH. The data distribution also impacted performance, with sorted and nearly-sorted arrays being processed faster across all schemes.

Summary:

This report presents a complete analysis of Shell Sort based on my partner's implementation. The algorithm includes three different gap generation strategies: SHELL, KNUTH, and SEDGEWICK. Each strategy significantly affects the algorithm's behavior in terms of both theoretical and practical performance. Throughout the analysis, I examined the structure of the algorithm, provided complexity analysis, reviewed the code for inefficiencies, suggested improvements, and confirmed their impact using empirical tests.

From a theoretical perspective, Shell Sort is known for its strong dependence on the gap sequence. The SHELL sequence, although simple to implement, shows the weakest performance. KNUTH's method performs better by increasing the intervals in a more efficient manner. SEDGEWICK's approach demonstrated the best theoretical and practical results. Empirical performance measurements also confirmed that SEDGEWICK consistently outperforms both SHELL and KNUTH, especially on larger inputs.

In the code review section, several inefficiencies were identified, such as the use of Math.pow which involves unnecessary floating-point operations, the use of TreeSet which adds memory and time overhead, and the generation of gaps in ascending order while iterating in reverse. These issues were resolved by replacing floating-point exponentiation with integer calculations, switching from TreeSet to basic lists with merging, and generating the gap array directly in the reverse order needed. The inner loop was also simplified for improved readability and control flow.

After implementing the proposed changes, the algorithm showed significant improvements in both speed and resource usage. The empirical results confirmed that the improved version of Shell Sort executes faster, performs fewer memory accesses and swaps, and scales better on larger inputs. In particular, the SEDGEWICK scheme demonstrated the best balance of efficiency and accuracy across all input distributions.

In conclusion, Shell Sort becomes a much more competitive algorithm when implemented thoughtfully. The suggested optimizations do not change its fundamental logic but substantially improve its performance. Among the tested gap strategies, SEDGEWICK is clearly the most effective. I recommend using it for real-world applications that require in-place sorting with reliable performance. Future enhancements could include dynamic selection of gap schemes or further optimization for specific data patterns. This analysis shows that algorithm quality depends not only on theoretical design but also on implementation details.