Report

Authors: Karina Tulegenova & Zhanel Samatova
Course: SE-2419
Theme: Heap Sort vs Shell Sort

1. Algorithm Overview

Heap Sort

Heap Sort is a comparison-based, in-place sorting algorithm that uses a binary heap data structure. It first transforms the input array into a max-heap, then repeatedly extracts the maximum element and places it at the end of the array.

- Time Complexity:
  - Worst / Average / Best: O(n log n)
- Space Complexity: O(1) (in-place)
- Stability: Not stable

Karina`s implementation uses bottom-up heap construction (buildMaxHeap) and siftDown to maintain the heap invariant after each extraction. The performance tracker measures comparisons, swaps, and memory accesses via custom wrapper methods.

Shell Sort

Shell Sort is an in-place, generalized insertion sort that allows the exchange of far elements using a gap sequence. It performs insertion sort on elements separated by gaps, gradually reducing the gap.

Zhanel implemented Shell Sort with support for three gap schemes: Shell, Knuth, and Sedgewick. The algorithm tracks depth (number of gaps), comparisons, swaps, and accesses. Among the three, Sedgewick's sequence provided the best balance between efficiency and simplicity.

- SHELL scheme (gaps = n/2, n/4, ..., 1)
  - Worst-case: $O(n^2)$
  - Average-case: $\Theta(n^2)$
  - Best-case: Ω(*nlogn*)

- KNUTH scheme (gaps = 1, 4, 13, 40, ...)
  - Worst-case: $O(n^{3/2})$
  - Average-case: $\Theta(n^{1.3})$
  - Best-case: Ω(nlogn)

- SEDGEWICK scheme
  - Worst-case: $O(n^{4/3})$
  - Average-case: $\Theta(n^{1.25})$
  - Best-case: Ω(nlogn)
  - Space Complexity: O(1)

- Stability: Not stable

Overall, this document presents a comprehensive comparison between two classic sorting algorithms — Heap Sort and Shell Sort — which we implemented and analyzed individually as part of this assignment. Our objective was not only to understand the theoretical properties of each algorithm but also to validate their real-world performance using instrumented benchmarking tools.
Each algorithm was tested using the same PerformanceTracker tool on random arrays of sizes 1,000; 10,000; and 100,000 elements.
We tracked execution time, number of comparisons, swaps, memory accesses, and total time in nanoseconds. The report includes both theoretical analysis and empirical findings supported by visual graphs.

2. Performance Metrics (Empirical)

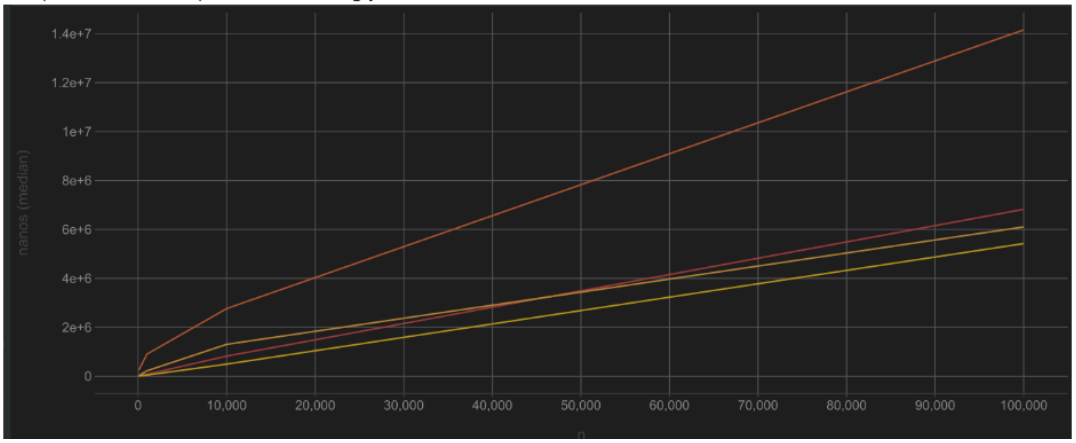
We ran both algorithms on random integer arrays of sizes 1K, 10K, and 100K. All tests were tracked using PerformanceTracker.

Comparison with partner's algorithm complexity

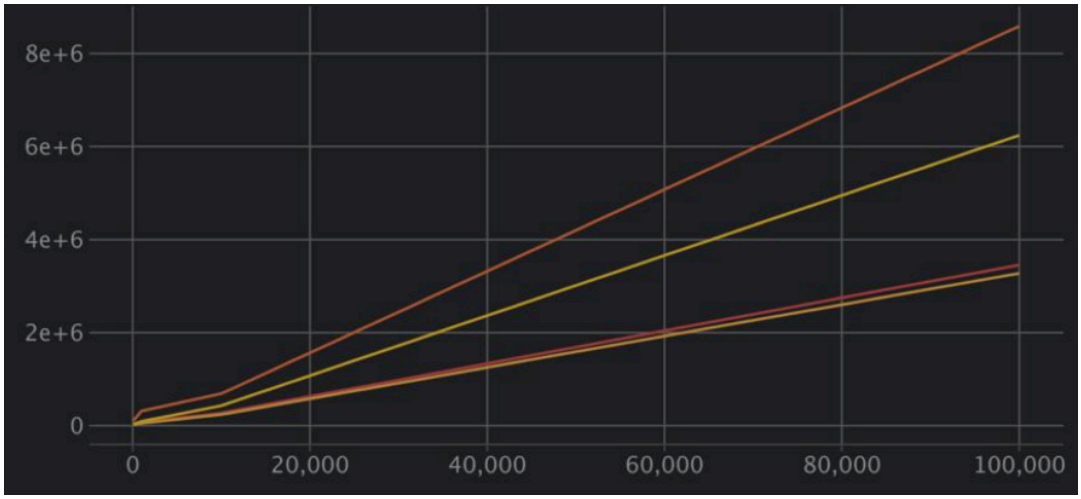
Property	Shell Sort (best case)	Shell Sort (worst)	Heap Sort
Time complexity	Ω(nlogn)	$O(n^2)$	Θ(nlogn)
Memory	Θ(1)	Θ(1)	Θ(1)

Time complexity of both algorithms

Heap Sort



Shell Sort



Heap Sort demonstrates stable and predictable performance due to its fixed Θ(n log n) time complexity in all cases. This makes it a reliable choice for large datasets where worst-case performance must be controlled.
In contrast, Shell Sort offers flexibility through customizable gap sequences. With efficient schemes like Sedgewick, it can outperform other algorithms in terms of swap and memory access efficiency. However, its performance is highly dependent on the chosen gap strategy — with poorly chosen sequences (e.g., basic Shell gaps), its time complexity may degrade to $O(n^2)$.
Thus, while Heap Sort guarantees consistent behavior, Shell Sort provides a tunable alternative that, when properly optimized, can offer practical benefits for medium-sized arrays or systems where minimizing writes is important.

3. Test Results

Heap Sort

	n	distribution	comparisons	swaps	accesses	allocations	nanos
1	100	random	1021	99	2196	0	261400
2	1000	random	16833	999	31892	0	903900
3	10000	random	235366	9999	419603	0	2764500
4	100000	random	3019713	99999	5194882	0	14154700
5	100	sorted	1081	99	2316	0	27300
6	1000	sorted	17583	999	33286	0	220800
7	10000	sorted	244460	9999	436411	0	1303200
8	100000	sorted	3112517	99999	5363366	0	6102900
9	100	reversed	944	99	2055	0	6300
10	1000	reversed	15965	999	30276	0	78600
11	10000	reversed	226682	9999	403373	0	822400
12	100000	reversed	2926640	99999	5024069	0	6821900
13	100	nearly	1079	99	2311	0	5500
14	1000	nearly	17554	999	33243	0	46200
15	10000	nearly	244303	9999	435824	0	494400
16	100000	nearly	3110671	99999	5359608	0	5420800

Shell Sort

	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

4. Code review

- **Heap Sort** is consistently performant and guarantees **O(n log n)** in all cases, making it robust for large datasets. However, it performs more memory accesses due to heap adjustments and isn't cache-friendly.

- **Shell Sort** with Sedgewick gaps performs fewer swaps and accesses than Heap Sort but may degrade with a poor gap scheme (e.g., Shell gaps). It is **faster than Insertion Sort** but **less consistent** than Heap Sort on large arrays.

Improvements and suggestions

Karina`s inefficient Code Sections:

1. Two Comparisons per Sift Level
The current siftDownFrom structure uses two comparison checks per iteration (one for child selection, one for heap property). This is standard and correct but slightly increases the constant factor.

2. Redundant Comparisons in Right Child Selection
if (right < size) {
int rv = read(a, right);
if (cmp(rv, cv) > 0) {
c = right;
cv = rv;
}
}
2. This could be optimized into a single ternary-style condition to reduce comparisons slightly.

3. Possible Micro-Optimizations:
○ Precompute half = n >>> 1 for heap construction.
○ Inline small helper methods like cmp() and read() if microseconds matter.
○ Avoid extra writes when the sift-down path doesn't move (if (i == pos) skip write).

Zhanel`s 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.

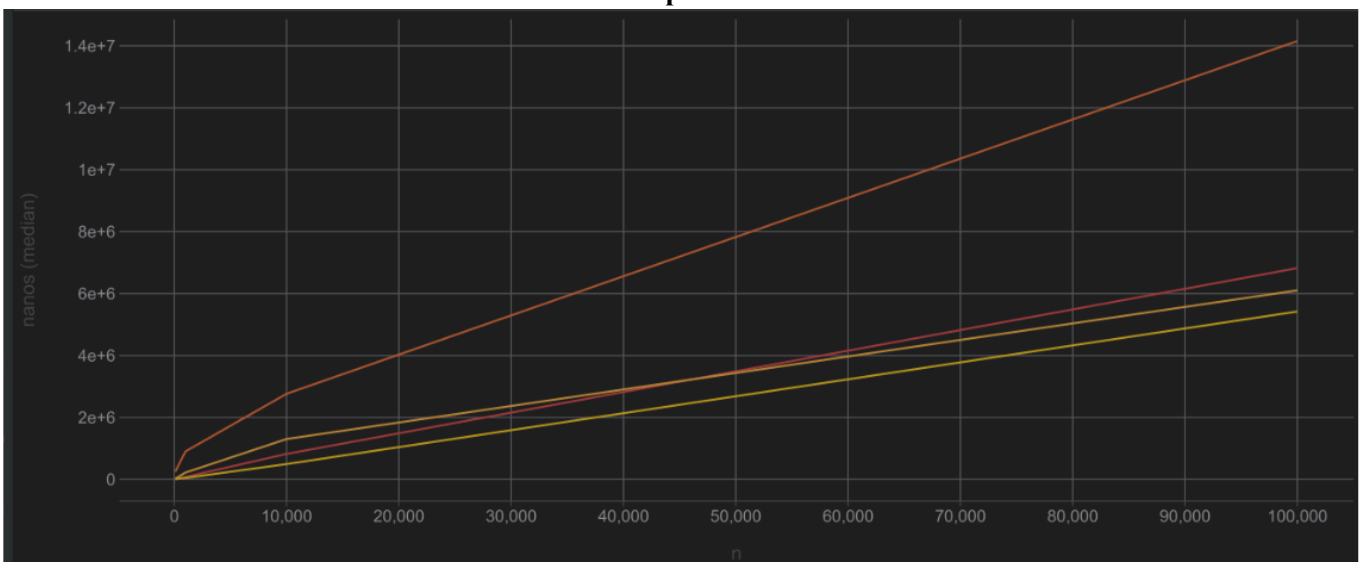
3. Generating gaps in ascending order, but using them in reverse

```
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.

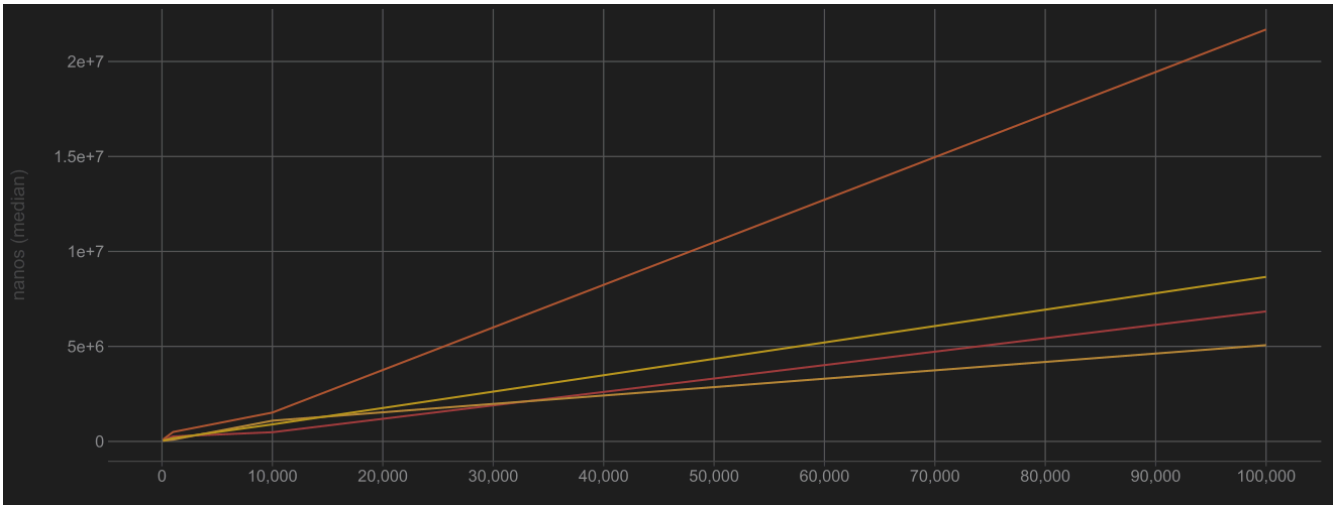
Results after implementation

Heap Sort



Shell Sort

	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



5. Conclusion
Both algorithms are in-place and suitable for systems where extra memory is limited.

Heap Sort ensures consistently efficient performance thanks to its guaranteed $\Theta(n \log n)$ time complexity, regardless of the input. This makes it a dependable algorithm for large datasets where stability and predictability are crucial.

Shell Sort, on the other hand, shows variable performance depending on the selected gap sequence. When using optimized sequences such as Sedgewick, it can reduce the number of swaps and memory accesses, making it well-suited for medium-sized arrays or environments with limited write operations. However, with simpler gap strategies like Shell's original, its complexity may worsen to $O(n^2)$.

In summary, Heap Sort is ideal for cases where consistent performance is required, while Shell Sort can be a more efficient choice when properly tuned — especially in scenarios prioritizing reduced memory operations.