DeltaSort: Incremental repair of sorted arrays with known updates

Shubham Dwivedi
Independent Researcher
shubd3@gmail.com

January 2026

Abstract

Maintaining sorted order under incremental updates is a common requirement in read-heavy systems, yet most production sorting routines are blind to which elements have changed and must repeatedly rediscover partial order from scratch. This paper explores an alternative model in which the sorting routine is explicitly informed of the indices updated since the previous sort. Under this model, we present *DeltaSort*, an incremental repair algorithm for arrays that batches multiple updates instead of applying them independently. *DeltaSort* reduces redundant comparison work and overlapping data movement by exploiting update locality and batching. An experimental evaluation of a Rust implementation shows that *DeltaSort* consistently outperforms repeated binary-insertion and blind native sorting for update batch sizes up to 25%, with multi-fold speedups and a clear crossover point depending on array size where full re-sorting becomes preferable. These results suggest that tighter integration between update pipelines and sorting routines can yield significant performance gains in real incremental-sorting workloads at the cost of increased memory usage to track updated indices.

1 Introduction

Sorting is among the most heavily optimized primitives in modern systems. Standard library implementations—TimSort [14], Introsort [12], and PDQSort [13]—deliver excellent performance for general inputs by exploiting partial order, cache locality, and adaptive strategies. However, these algorithms operate under a *blind* model: they rediscover structure dynamically rather than being explicitly informed about which elements have changed since the previous sort.

In many practical systems, this assumption is unnecessarily pessimistic. Sorted arrays are often maintained incrementally in read-heavy workloads where updates affect only a subset of elements and the indices of those updates can be easily tracked and known. Nevertheless, this information is typically not tracked or utilized, and systems fall back to blind re-sorting or independent incremental repairs using binary-insertion or extract-sort-merge. To address these limitations, this paper makes the following contributions:

1. **Update-aware sorting model:** We formulate an incremental sorting model in which the sorting routine is explicitly informed of the indices updated since the previous sort. Under this model, a natural baseline is repeated binary insertion, where each update is repaired independently using binary search. While correct, this approach exhibits efficiency limitations for batched updates due to repeated full-range searches and overlapping data movement.

2. **DeltaSort algorithm:** We present *DeltaSort*, a coordinated incremental repair algorithm for sorted arrays designed for the update-aware model. DeltaSort batches updates and repairs them jointly, reducing redundant comparison work and overlapping data movement by

exploiting update locality and batching. DeltaSort preserves correctness and does not improve asymptotic complexity, but achieves consistent practical performance improvements over repeated binary insertion and blind native sorting in relevant update regimes.

2 Related Work

Adaptive sorting algorithms exploit existing order in the input to improve performance on nearly sorted data. TimSort [14] and natural merge sort [10] identify monotonic runs dynamically and merge them efficiently, yielding improved performance when such structure is present. A substantial body of work formalizes measures of presortedness and studies sorting algorithms whose complexity depends on these measures rather than input size alone [11]. These approaches, however, must discover structure through full-array scans and do not assume explicit knowledge of which elements were modified.

Dynamic data structures offer a different trade-off. Self-balancing trees such as AVL trees [1], red–black trees [9], B-trees [2], and skip lists [15] support efficient ordered updates with logarithmic cost, but sacrifice contiguous memory layout and cache locality. Library sort [3] reduces insertion cost by maintaining gaps in the array, but addresses online insertion and incurs additional space overhead.

In contrast, this work focuses on maintaining sorted order in arrays under batched updates where the indices of modified elements are explicitly available. This update-aware model enables efficient repair without auxiliary data structures or additional memory, and distinguishes DeltaSort from prior adaptive and incremental approaches.

3 Problem Model

Definition 1 (Update-Aware Sorting)**.** Let $A[0..n-1]$ be an array sorted according to a strict weak ordering defined by a comparator `cmp`. Suppose a set of indices $D = \{d_1, \ldots, d_k\} \subseteq \{0, \ldots, n-1\}$ is given such that the values at these indices may have been arbitrarily modified, while values at all other indices remain unchanged.

The *update-aware sorting problem* is to restore A to a state that is sorted with respect to `cmp`, given explicit knowledge of the set D.

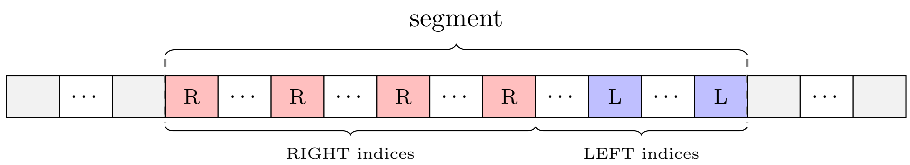
4 DeltaSort Algorithm

4.1 Overview

Definition 2 (Violation)**.** For a dirty index i, we classify its *violation* based on local order:

- **LEFT (L)**: Element **must** move left — $\texttt{cmp}(A[i-1], A[i]) > 0$ (for $i > 0$).
- **RIGHT (R)**: Element **may** move right or stay stable. Essentially, all dirty indices that are not LEFT.

Definition 3 (Segment)**.** A *segment* is a maximal group of dirty indices with structure $(\mathrm{R})^*\mathrm{L}^+$: zero or more RIGHT dirty indices followed by one or more LEFT dirty indices. Clean elements may be interspersed between dirty indices within a segment.



DeltaSort operates in two phases:

1. **Phase 1 (Preparation):** Extract dirty values, sort them, write back to dirty positions in index order. This establishes direction segments in the array which are disjoint and can be repaired independently.

2. **Phase 2 (Repair):** Repair each segment left-to-right, deferring RIGHT indices to a stack until the first LEFT index is encountered. Flush and repair the the RIGHT indexes in stack in LIFO order. Then repair all LEFT indices left-to-right.

4.2 Key Insight: Directional segmentation enables localized repair

The key insight behind DeltaSort is that pre-sorting dirty values induces a *directional segmentation* of updates. After Phase 1, dirty indices partition into disjoint segments of the form $(\mathrm{R})^*\mathrm{L}^+$.

Lemma 4 (Movement Confinement)**.** *Element movement is bounded within each segment: no element crosses a segment boundary.*

Proof. Let S be a segment with RIGHT indices $R_1, \ldots, R_m$ followed by LEFT indices $L_1, \ldots, L_p$. After Phase 1, dirty values are monotonically ordered by index, so $A[R_1] < \cdots < A[R_m] < A[L_1] < \cdots < A[L_p]$.

1. *RIGHT elements cannot pass the leftmost LEFT.* Since $A[R_i] < A[L_1]$ for all i.

2. *LEFT elements cannot pass the rightmost RIGHT.* Since $A[R_m] < A[L_j]$ for all j.

Since no element exits its segment, segments can be repaired independently. $\square$

4.3 Detailed Algorithm

Algorithm 1 DeltaSort

Require: Array $A[0..n-1]$, dirty indices D, comparator `cmp`
Ensure: A is sorted

```
Phase 1: Prepare
dirty ← sort(D)                                   ▷ Sort indices ascending
values ← [A[d] : d ∈ dirty]
values ← sort(values, cmp)
for i ← 0 to |dirty| − 1 do
    A[dirty[i]] ← values[i]
end for

Phase 2: Repair
pending ← []; leftBound ← 0
for p ← 0 to |dirty| − 1 do
    i ← dirty[p]
    if IsLeftViolation(A, i) then
        FlushPending(pending, i − 1)              ▷ Fix pending before LEFT
        leftBound ← FixLeftViolation(A, i, leftBound) + 1
    else
        pending.push(i)                           ▷ Defer RIGHT indices
    end if
end for

FlushPending(pending, n − 1)                      ▷ Fix remaining RIGHT violations
```

```
function IsLeftViolation(A, i)
    return i > 0 ∧ cmp(A[i − 1], A[i]) > 0
end function

function IsRightViolation(A, i)
    return i < n − 1 ∧ cmp(A[i], A[i + 1]) > 0
end function
```

```
function FixLeftViolation(A, i, leftBound)
    t ← BinarySearchLeft(A, A[i], leftBound, i − 1)
    Move(A, i, t)
    return t
end function
```

```
function FlushPending(pending, rightBound)
    while pending ≠ ∅ do
        s ← pending.pop()                                   ▷ Process in LIFO order
        if IsRightViolation(A, s) then
            t ← BinarySearchRight(A, A[s], s + 1, rightBound)
            Move(A, s, t)
        end if
    end while
end function
```

```
function Move(A, from, to)
    v ← A[from]
    if from < to then
        Shift A[from + 1..to] left by one
    else if from > to then
        Shift A[to..from − 1] right by one
    end if
    A[to] ← v
end function
```

4.4 Correctness Proof

Lemma 5 (Segment Boundary Invariant)**.** *Segment boundaries are never violated. For any two consecutive segments S_i and S_{i+1}, all values in S_i remain less than all values in S_{i+1} throughout the repair process.*

Proof sketch. After Phase 1, dirty values are monotonically ordered by index. A segment boundary occurs where a LEFT dirty index is followed by a RIGHT dirty index. At such a boundary, the RIGHT element satisfies $A[d_{i+1}] \geq A[d_{i+1} - 1]$, establishing sorted order. By Lemma 4, movement is confined within segments, so boundaries remain intact. A fully rigorous proof is deferred to future work. □

Lemma 6 (Fixing Does Not Introduce Violations)**.** *When a LEFT or RIGHT element is moved to its correct position via binary search, no new violations are introduced.*

Proof. LEFT fix: Binary search finds position $t < i$ where the element belongs. Elements in $A[t..i - 1]$ shift right by one. The shift preserves relative order. By binary search, the moved element satisfies $A[t - 1] \leq A[t] < A[t + 1]$.

RIGHT fix: Binary search finds position $t > s$ where the element belongs. Elements in $A[s+1..t]$ shift left by one. The shift preserves relative order. By binary search, the moved element satisfies $A[t - 1] < A[t] \leq A[t + 1]$.

In both cases, no new violations are introduced. □

Theorem 7 (Correctness)**.** *DeltaSort produces a correctly sorted array.*

Proof. Phase 2 processes each dirty index exactly once, fixing its violation if any. By Lemma 6, each fix resolves a violation without introducing new ones. After all dirty indices are processed, each segment is internally sorted.

By Lemma 5, segment boundaries maintain sorted order throughout. Since each segment is internally sorted and boundaries preserve global order, the entire array is sorted. □

4.5 Algorithm Analysis

Theorem 8 (Time Complexity)**.** *DeltaSort runs in $O(k \log k + k \log n + M)$ time, where M is total movement.*

Proof. **Phase 1**: Sort k indices: $O(k \log k)$. Sort k values: $O(k \log k)$. Write back: $O(k)$.

Phase 2: Each dirty index: $O(1)$ direction check, $O(\log n)$ binary search. Total: $O(k \log n)$. Movement: $O(M)$. □

Theorem 9 (Space Complexity)**.** *DeltaSort uses $O(k)$ auxiliary space.*

Theorem 10 (Comparison Optimality)**.** *DeltaSort achieves $O(k \log n)$ comparisons, which matches the information-theoretic lower bound: each of k dirty elements can occupy any of n final positions, requiring $\log_2(n^k) = k \log n$ comparisons to distinguish all configurations.*

Remark 11 (Movement Efficiency). While worst-case movement is $O(kn)$, the segmentation created by Phase 1 tends to reduce movement in practice. When dirty values would otherwise "cross" (one moving left, another right over the same positions), Phase 1 reassigns them to minimize displacement. Empirical results in §5 demonstrate substantial speedups, validating that movement is typically much less than the worst case.

Table 1: Algorithm complexity comparison.

Algorithm	Comparisons	Movement	Space	Bounded Search?
Native Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	N/A
Binary Insertion	$O(k \log n)$	$O(kn)$	$O(1)$	No
Extract-Sort-Merge	$O(k \log k + n)$	$O(n)$	$O(n)$	N/A
DeltaSort	$O(k \log n)$	$O(kn)$*	$O(k)$	Yes

*Worst case; typically $O(k \cdot \bar{m})$ for average movement distance $\bar{m}$.

5 Experimental Evaluation

5.1 Baseline Algorithms

Native Sort. Re-sort the entire array: $O(n \log n)$ comparisons, $O(n \log n)$ movements.

Binary Insertion (BI). For each $d \in D$: extract all dirty values, then for each: binary search for correct position, reinsert. Cost: $O(k \log n)$ comparisons, $O(kn)$ worst-case movement. Always correct, but searches the full array range for each insertion.

Extract-Sort-Merge (ESM). Extract dirty values, sort them, merge with clean elements. Cost: $O(k \log k + n)$ comparisons, $O(n)$ movement. Always correct but requires $O(n)$ auxiliary space.

5.2 Setup

Primary Implementation. Rust 1.75 (release build with optimizations).

Hardware. Apple M2 Pro, 16GB RAM, macOS 14.

Table 2: Execution time (µs) for $n = 50,000$ elements. DS vs Best shows speedup against the fastest alternative (Native, BI, or ESM).

k	Native	BI	ESM	DeltaSort	DS vs Best
1	439	40	188	**1**	62× faster
5	749	217	293	**27**	8× faster
10	747	525	494	**44**	11× faster
20	962	782	655	**49**	13× faster
50	1,350	1,994	1,307	**98**	13× faster
100	1,867	4,818	2,673	**109**	17× faster
200	2,560	9,022	4,785	**178**	14× faster
500	3,703	22,112	11,655	**782**	5× faster
1,000	4,253	44,481	22,480	**465**	9× faster
2,000	4,158	87,827	43,627	**837**	5× faster
5,000	4,223	211,016	102,198	**1,751**	2.4× faster
10,000	4,539	377,981	176,373	**3,326**	1.4× faster
20,000	**5,042**	615,645	259,835	6,914	1.4× slower

Algorithms.

- **Native**: Rust's `sort_by` (pattern-defeating quicksort)
- **BI**: Binary Insertion (extract-then-insert)
- **ESM**: Extract-Sort-Merge
- **DS**: DeltaSort

Data. User objects with composite key (country, age, name). Array sizes $n \in \{1\text{K}, 10\text{K}, 50\text{K}, 100\text{K}, 500\text{K}, 1\text{M}\}$. Dirty counts $k \in \{1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000\}$. 100 iterations per configuration with 95% confidence intervals reported.

5.3 Results

Table 2 shows timing results for $n = 50,000$ elements. DeltaSort consistently outperforms all alternatives up to approximately $k = 10,000$ (20% of array size), achieving speedups of 7–17× over Native sort in the sweet spot ($20 \le k \le 200$).

Figure 1 visualizes the crossover between DeltaSort and Native sort on a linear scale. The crossover occurs at $k \approx 13,700$ (27% of array size), clearly showing the region where DeltaSort provides substantial speedups.

Figure 2 compares all four algorithms on a log-log scale, revealing the orders-of-magnitude performance gap between DeltaSort and the baseline algorithms (BI and ESM) across the full range of k values.

5.4 Crossover Analysis

A key practical question is: at what delta size should one switch from DeltaSort to Native sort? A binary search was conducted for the crossover point k_c across array sizes from 1K to 1M elements. Table 3 summarizes the results.

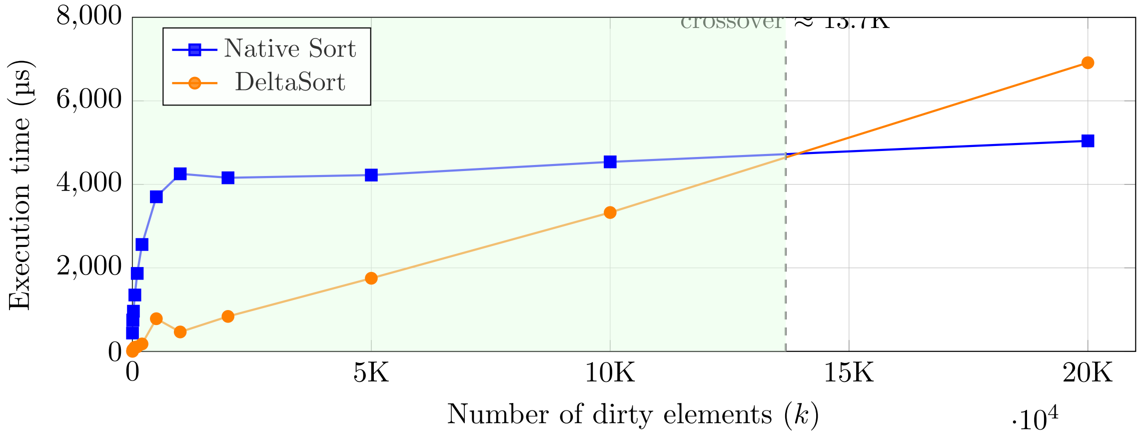


Figure 1: DeltaSort vs. Native Sort for $n = 50,000$. The crossover point occurs at $k \approx 13,700$ (27% of n). The shaded region indicates where DeltaSort is faster.

Table 3: Crossover point k_c where Native sort becomes faster than DeltaSort.

n	k_c	k_c/n
1,000	235	23.5%
2,000	469	23.4%
5,000	1,211	24.2%
10,000	2,813	28.1%
20,000	5,782	28.9%
50,000	13,672	27.3%
100,000	31,251	31.3%
200,000	54,688	27.3%
500,000	105,469	21.1%
1,000,000	156,251	15.6%

The crossover ratio k_c/n ranges from approximately 15% to 31%, with most values falling in the 20–30% range. This suggests a practical rule of thumb:

Use DeltaSort when fewer than ∼25% of elements are dirty; otherwise use Native sort.

5.5 JavaScript Implementation

DeltaSort was also implemented in TypeScript running on Node.js v20 (V8 engine). While the implementation passes all correctness tests, the performance results show higher variance due to JIT compilation behavior, garbage collection pauses, and other engine-level effects. The JavaScript benchmarks will be refined in a future revision to provide more stable measurements. The Rust implementation provides the authoritative performance characterization.

5.6 Analysis

DeltaSort vs. Binary Insertion. DeltaSort dramatically outperforms BI across all tested configurations, with speedups ranging from 8× to over 100×. This is because BI's $O(kn)$ extraction

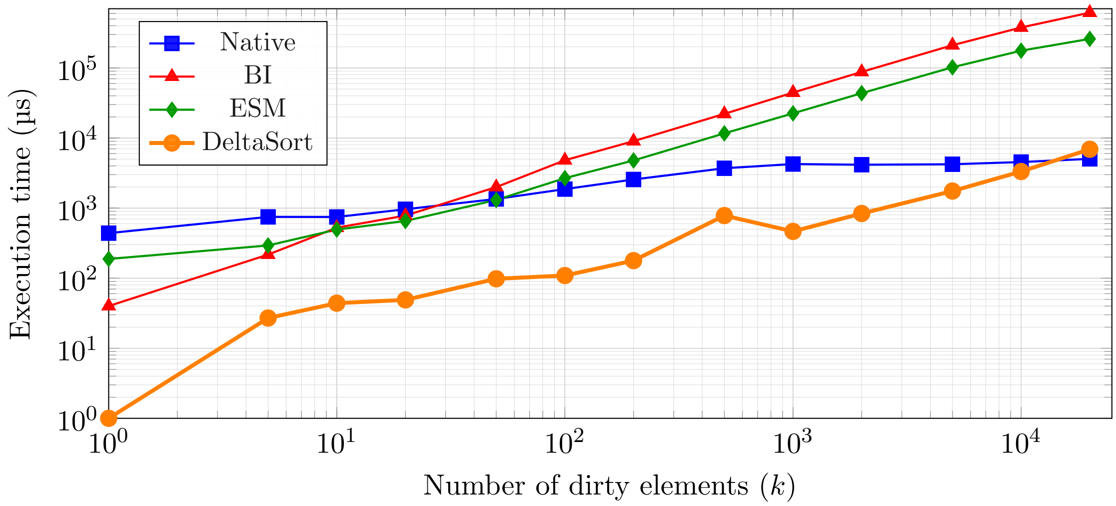


Figure 2: Execution time comparison of all algorithms for $n = 50,000$ (log-log scale). DeltaSort (orange) is consistently fastest across all tested k values except $k = 20,000$ where Native sort wins. Note the orders-of-magnitude gap between DeltaSort and BI/ESM.

and insertion costs dominate, while DeltaSort's segmentation enables coordinated processing that avoids redundant movement.

DeltaSort vs. ESM. DeltaSort also outperforms ESM for all tested k values up to 10,000. ESM's $O(n)$ merge pass becomes competitive only when k is very large, and even then DeltaSort remains faster in these measurements.

DeltaSort vs. Native Sort. The crossover with Native sort occurs at approximately 20–30% dirty elements. Below this threshold, DeltaSort's $O(k \log n)$ complexity and efficient stack-based processing provide substantial speedups. Above this threshold, Native sort's highly optimized $O(n \log n)$ implementation wins.

Algorithm Selection Guide. Based on the Rust benchmarks across array sizes from 1K to 1M elements:

Condition	Recommendation
$k \leq 5$	Binary Insertion (skip Phase 1 overhead)
$5 < k < 0.25n$	**DeltaSort**
$k \geq 0.25n$	Native Sort

6 Future Work

Formal Movement Bounds. A conjecture is that Phase 1's segmentation property reduces total element displacement compared to uncoordinated binary insertion. A formal proof characterizing the expected movement reduction—potentially in terms of the "inversion distance" between original dirty values and their indices—would strengthen the theoretical foundation.

Cache-Aware Analysis. Formalize why bounded search ranges and localized segment processing help despite unchanged asymptotic complexity.

Block Storage. Analyze DeltaSort for B-tree maintenance, where batched updates may reduce node splits.

JavaScript Implementation Refinement. The TypeScript/Node.js implementation shows higher variance due to JIT compilation, garbage collection, and engine-level effects. Future work will characterize and minimize these effects to provide stable JavaScript benchmarks.

Adaptive Hybrid. Runtime selection between DeltaSort and Native sort based on the 25% crossover threshold, potentially with dynamic adjustment based on observed performance.

7 Conclusion

This paper presented DeltaSort, a coordinated incremental repair algorithm for sorted arrays. The key insight is that a pre-sorting phase creates *directional segments*—contiguous groups of dirty elements that all need to move in the same direction—which can then be processed efficiently via stack-based coordination.

The main contributions are:

1. **Segmentation via pre-sorting**: Establishing monotonicity among dirty values creates directional segments that enable coordinated processing.
2. **Stack-based coordination**: LEFT segments are processed immediately with progressive search narrowing; RIGHT segments are deferred to a stack and processed in LIFO order, ensuring stable target positions.
3. **Optimal comparisons**: $O(k \log n)$, matching the information-theoretic lower bound.
4. **Substantial practical speedup**: 5–17× over native sort for delta sizes up to ∼25% of array size, validated through Rust benchmarks across array sizes from 1K to 1M elements.

The crossover point where native sort becomes faster occurs at approximately 25% dirty elements, providing a clear decision boundary for practitioners.

The broader lesson is that exploiting application-level knowledge (which indices changed) enables coordination that blind algorithms cannot achieve. DeltaSort demonstrates that careful coordination—segmentation plus stack-based processing—yields substantial practical gains.

References

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[2] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[3] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $o(n \log n)$. In *Proceedings of the 3rd International Conference on Fun with Algorithms*, pages 16–23, 2004.

[4] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

[5] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[6] Vladimir Estivill-Castro and Derick Wood. A new family of sorting algorithms. *Computing Surveys*, 24(4):441–476, 1992.

[7] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[8] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977.

[9] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 2nd edition, 1998.

[11] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 100(4):318–325, 1985.

[12] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[13] Orson R. L. Peters. Pattern-defeating quicksort. `https://github.com/orlp/pdqsort`, 2021.

[14] Tim Peters. Timsort. Python source code, 2002. Adopted by Java SE 7, Android, and V8.

[15] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.