DeltaSort: Incremental repair of sorted arrays with known updates

Shubham Dwivedi
Independent Researcher
shubd3@gmail.com

January 2026

Abstract

Reading sorted data is a fundamental requirement in almost all data processing systems. When dealing with large sorted datasets that require great read performance, sorting-on-read is not feasible. A standard approach is to have a derived sorted read-replica that is updated with the latest snapshot asynchronously whenever the system-of-record gets updated. For updating read-replicas, most production systems resort to either full re-sorting or Binary-Insertion-Sort or Extract-Sort-Merge. In this paper, we present why the traditional approaches are sub-optimal and how we can do better. This paper also formulates an alternative model in which sorting routine is explicitly informed of the updated indices since the previous sort. Under this model, we present *DeltaSort*, an efficient incremental repair algorithm for arrays. We present experimental evidence that shows that *DeltaSort* outperforms existing approaches for a wide range of update batch sizes. These results suggest that tighter integration between update pipelines and sorting routines can yield significant performance gains in real incremental-sorting workloads. We also explore limitations of this approach and identify a clear crossover point, depending on array size, where full re-sorting becomes preferable.

1 Introduction

Sorting is among the most optimized primitives in modern systems backed by decades of deep research. Standard library implementations—TimSort [10], Introsort [8], and PDQSort [9] 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 if needed. Nevertheless, this information is typically not tracked or utilized, and systems fall back to blind re-sorting or sub-optimal repair approaches using Binary-Insertion-Sort or Extract-Sort-Merge. As an example, consider a web application that shows a real-time game leaderboard. Being able to efficiently update the sorted view on updates directly impacts the user experience. The traditional approaches do not serve us very well—full re-sort can block the main UI thread for long durations, while Binary-Insertion-Sort can be slow for a large number of updates. 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 updated indices since the previous sort. Under

this model, Binary-Insertion-Sort and Extract-Sort-Merge are standard baseline algorithms because they help maintain sorted order by processing incremental updates.

2. **DeltaSort algorithm:** We present *DeltaSort*, an 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 element movement by exploiting update locality and batching. DeltaSort achieves multi-fold speed ups over traditional approaches and also beats Native-Sort for a wide range of update batch sizes.

2 Related Work

Adaptive sorting algorithms exploit existing order in the input to improve performance on nearly sorted data. TimSort [10] and natural merge sort [6] 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 [7]. 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 [5], B-trees [2], and skip lists [11] 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 updated elements since the last sort are explicitly available. This update-aware model enables efficient repair without auxiliary data structures, 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`. Let $D = \langle d_1, d_2, \ldots, d_k \rangle$ be a sequence of indices such that

$$0 \leq d_1 < d_2 < \cdots < d_k \leq n-1,$$

and 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

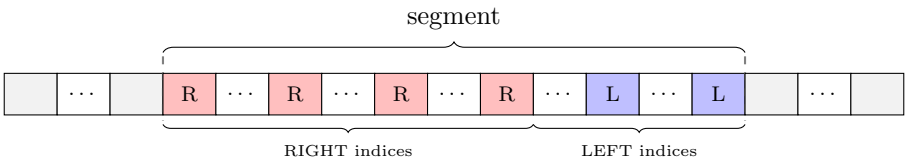
4.1 Overview

Definition 2 (Violation)**.** For an updated 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.

Every updated index is either a LEFT or RIGHT violation. Violation is only applicable to updated indices and has no meaning for clean indices.

Definition 3 (Segment)**.** A *segment* is a **maximal subarray** of D whose elements follow the pattern $(\mathrm{R})^*(\mathrm{L}^+ \mid \epsilon)$: zero or more RIGHT violations followed by one or more LEFT violations, or the empty suffix if the sequence ends.



DeltaSort operates in two phases:

1. **Phase 1 (Preparation):** Extract updated values, sort them, and write back to updated indices in index order. This establishes directional 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 RIGHT indices 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 updated values induces a *directional segmentation* of updates. After Phase 1, updated indices partition into disjoint segments of the form $(\mathrm{R})^*(\mathrm{L}^+ \mid \epsilon)$.

Lemma 4 (Movement Confinement)**.** *Element movement during repair phase 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. □

4.3 Detailed Algorithm

Algorithm 1 DeltaSort

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

Phase 1: Prepare
`updated` $\leftarrow$ sort(U) ▷ Sort indices ascending
`values` $\leftarrow [A[u] : u \in$ `updated`$]$
`values` $\leftarrow$ sort(`values`, `cmp`)
for $i \leftarrow 0$ **to** |`updated`| $- 1$ **do**
    A[`updated`[i]] $\leftarrow$ `values`[i]
end for

Phase 2: Repair
`pendingRight` $\leftarrow$ []; `leftBound` $\leftarrow 0$
for $p \leftarrow 0$ **to** |`updated`| $- 1$ **do**
    $i \leftarrow$ `updated`[p]
    if GetDirection(A, i) = Left **then**
        while `pendingRight` $\neq \emptyset$ **do** ▷ Fix pending before LEFT
            FixRightViolation(A, `pendingRight.pop()`, $i - 1$)
        end while
        `leftBound` $\leftarrow$ FixLeftViolation($A, i,$ `leftBound`) $+ 1$
    else
        `pendingRight.push`(i) ▷ Defer RIGHT
    end if
end for

while `pendingRight` $\neq \emptyset$ **do** ▷ Fix pending RIGHT violations
    FixRightViolation(A, `pendingRight.pop()`, $n - 1$)
end while

function GetDirection(A, i)
    return $i > 0 \wedge$ `cmp`$(A[i-1], A[i]) > 0$? Left : Right
end function
function FixLeftViolation(A, i, `leftBound`)
    $t \leftarrow$ BinarySearchLeft($A, A[i],$ `leftBound`$, i - 1$)
    Move(A, i, t)
    return t
end function
function FixRightViolation(A, i, `rightBound`)
    if $i < n - 1 \wedge$ `cmp`$(A[i], A[i+1]) > 0$ **then** ▷ Check if fixing is needed
        $t \leftarrow$ BinarySearchRight($A, A[i], i + 1,$ `rightBound`)
        Move(A, i, t)
    end if
end function

4.4 Correctness Proof

Lemma 5 (Initial Segment Boundary Order)**.** *For any two consecutive segments induced after Phase 1, let i be the index of the last updated element in first segment and j be the index of the first updated element in the second segment. Then* `cmp`$(A[i], A[j]) \leq 0$ *holds immediately after Phase 1.*

Proof. By construction, Phase 1 extracts all updated values, sorts them according to `cmp`, and writes them back to updated indices in increasing index order. Thus, for any two updated indices $d_p < d_q$, we have `cmp`$(A[d_p], A[d_q]) \leq 0$ after Phase 1. Since $i < j$ are consecutive updated indices belonging to adjacent segments, the claim follows directly. □

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 and Lemma 4, segment boundaries start with correct order and maintain correct 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. We analyze the two phases separately.

Phase 1. Sorting the k dirty indices and their corresponding values costs $O(k \log k)$ time. Writing the sorted values back requires $O(k)$ time.

Phase 2. Each dirty index is processed once. Direction checks cost $O(1)$ per index, and each repair performs a binary search over a sorted region in $O(\log n)$ time. Thus Phase 2 requires $O(k \log n)$ time. Let M denote the total number of elements moved during repair. The movement cost is $O(M)$.

Total. The overall complexity is $O(k \log n)$ time and $O(M)$ data movement. □

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

Proof. Phase 1 stores k dirty indices and k dirty values. Phase 2 maintains a pending stack of at most k indices. No $O(n)$ auxiliary structures are required. □

Remark 10 (Movement Efficiency). While worst-case movement is $O(kn)$, the segmentation created by Phase 1 tends to reduce movement in practice in the average case. Empirical results in §5 demonstrate substantial speedups, validating that movement is typically much less than the worst case. We will analyze average-case movement more rigorously in future work.

Here we compare algorithm complexity with other standard baseline approaches which we use in experiments to compare DeltaSort's performance:

- **NativeSort (NS)**: Re-sort the array using the natively available sort function. $O(n \log n)$ comparisons and movements.
- **Binary-Insertion-Sort (BIS)**: Extract dirty values, then for each: binary search for correct position, reinsert. $O(k \log n)$ comparisons, $O(kn)$ worst-case movement. Searches the full array range for each insertion.
- **Extract-Sort-Merge (ESM)**: Extract dirty values, sort them, merge with clean elements. $O(k \log k + n)$ comparisons, $O(n)$ movement. Requires $O(n)$ auxiliary space.

Table 1: Algorithm complexity comparison

Algorithm	Comparisons	Movement	Space
NativeSort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Binary-Insertion-Sort	$O(k \log n)$	$O(kn)$	$O(1)$
Extract-Sort-Merge	$O(k \log k + n)$	$O(n)$	$O(n)$
DeltaSort	$O(k \log n)$	$O(kn)$*	$O(k)$

*Worst case; average case is closer to $O(n)$.

5 Experimental Evaluation

All experiments are run using a Rust implementation of DeltaSort [4] on a synthetic dataset of user objects with composite keys (country, age, name) on an M3 Pro MacBook Pro with 18GB RAM. DeltaSort is compared against three baselines: NativeSort (Rust's `sort_by`), Binary-Insertion-Sort (BIS), and Extract-Sort-Merge (ESM).

5.1 Correctness

Correctness is formally-proven as Theorem 7 and also verified by an extensive set of randomized unit tests [4] across various scales and update sizes. The test routine generates a sorted base array of size N, applies k random updates at random indices, runs DeltaSort, and asserts that the final array is sorted and contains all original elements with updated values. All tests pass successfully.

5.2 Execution Time

The graph below shows execution time (in microseconds) for $n = 50,000$ elements as a function of dirty count k. DeltaSort consistently outperforms all alternatives up to approximately $k = 15K$ (crossover point), achieving significant speedups of 6–20× over NativeSort in the intermediate range. Also note the orders-of-magnitude gap between DeltaSort and the baseline incremental algorithms (BIS and ESM) across the full range of k values.

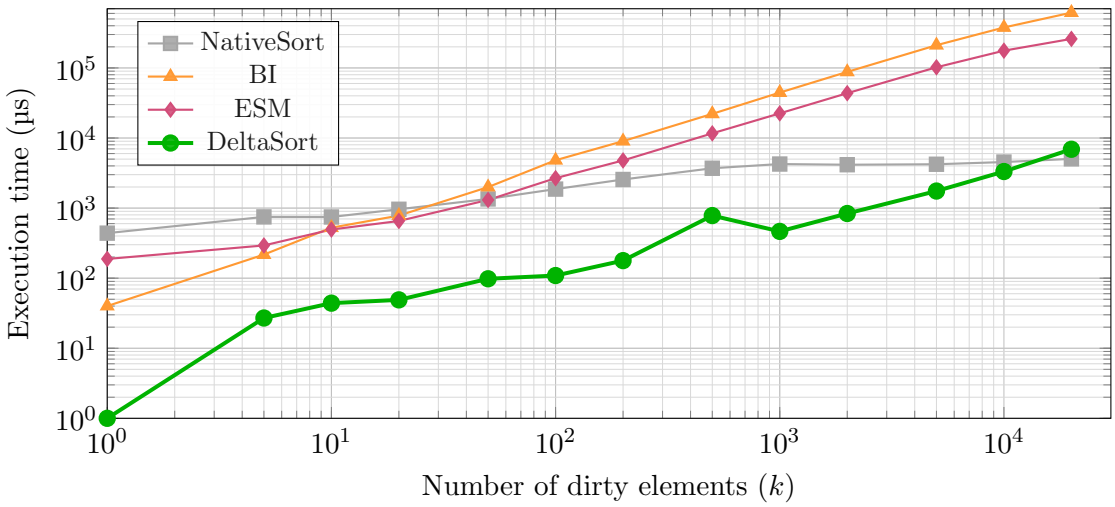


Figure 1: Execution time comparison of all algorithms for $n = 50,000$ (log-log scale)

5.3 Comparator Invocation Count

The following chart shows the number of comparator invocations for each algorithm. DeltaSort and BIS both achieve $O(k \log n)$ comparisons, substantially fewer than NativeSort's $O(n \log n)$ and ESM's $O(k \log k + n)$. The comparison counts for DeltaSort are 10–40% higher than BIS, indicating that DeltaSort's performance advantage comes from reduced data movement (Lemma 4). This also suggests that in scenarios with expensive comparators, the speedups from *DeltaSort* over NativeSort will be even greater.

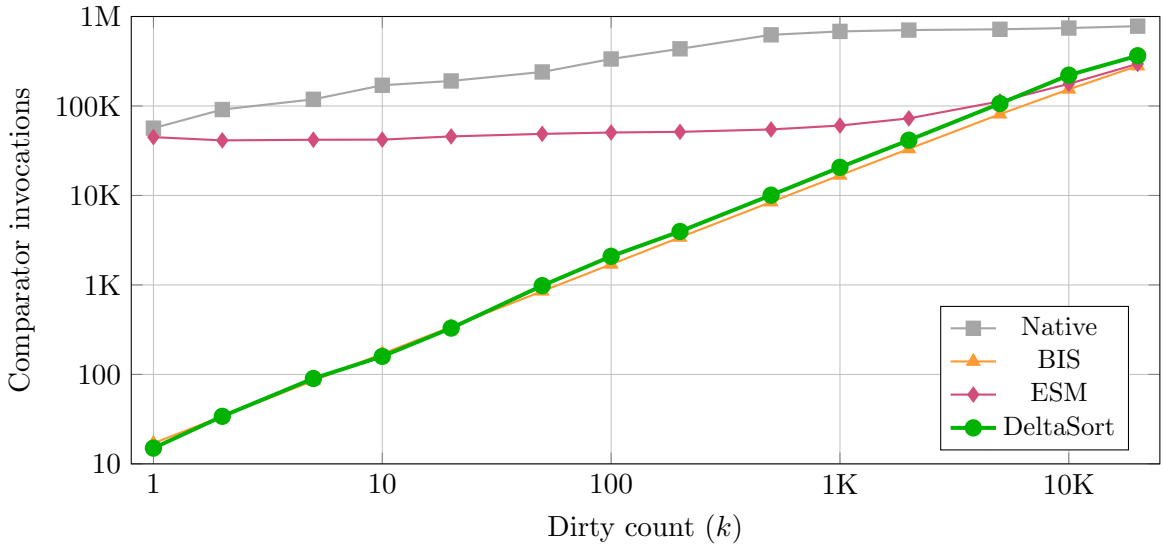


Figure 2: Comparator invocation count for $n = 50{,}000$ elements. DeltaSort and BIS both achieve $O(k \log n)$ comparisons, while Native uses $O(n \log n)$ regardless of k, and ESM uses $O(k \log k + n)$. The similar comparison counts for DeltaSort and BIS confirm that DeltaSort's speedup derives from movement efficiency, not fewer comparisons.

5.4 Crossover Threshold Analysis

A key practical question is: at what delta size should one switch from DeltaSort to NativeSort? A binary search was conducted for the crossover point k_c across array sizes from 1K to 10M elements.

Figure 3 visualizes how the crossover ratio k_c/n varies with array size. The ratio peaks around 31% for medium-sized arrays ($n \approx 50$K) and declines for very large arrays, suggesting that DeltaSort's advantage narrows as arrays grow very large. More study is needed to understand this trend fully, because intuitively the trend should hold up much better than the empirical data seems to suggest.

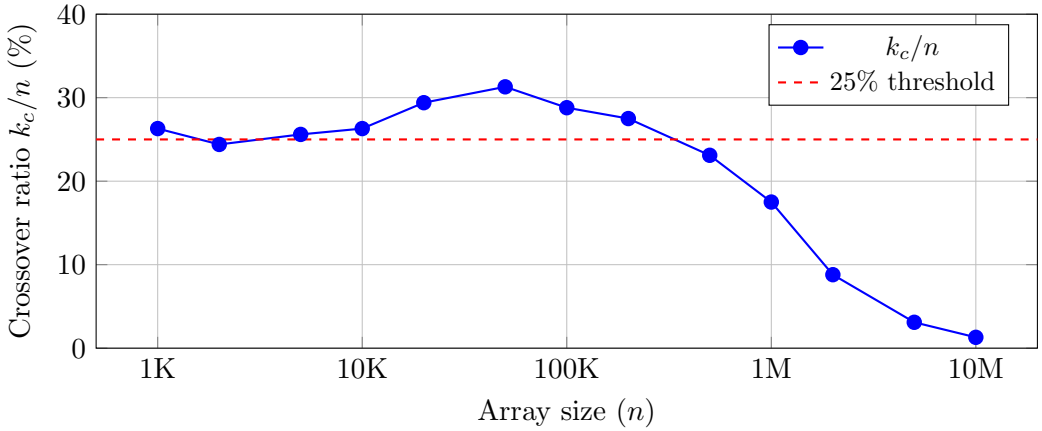


Figure 3: Crossover ratio k_c/n as a function of array size. DeltaSort outperforms Native sort when the dirty fraction is below this curve. The 25% rule of thumb (dashed line) is conservative for small-to-medium arrays but optimistic for very large arrays where the crossover drops significantly.

The key takeaway is that DeltaSort is beneficial for **large** ranges of update sizes. So DeltaSort remains a viable choice even when delta sizes are large. The exact threshold depends on the scenario specifics (array sizes, data types, comparator cost, etc.) but the results suggest that a **25% dirty fraction** is a reasonable rule of thumb for the upper limit of delta size before which DeltaSort remains advantageous.

5.5 Performance in managed execution environments

DeltaSort was also implemented in JavaScript [4] and benchmarked on the V8 engine to evaluate behavior in managed runtimes. Initial results indicate that DeltaSort outperforms NativeSort for some workloads, but the gains are more modest and the crossover point occurs at smaller update sizes. This behavior appears to be driven primarily by the performance characteristics of the native JavaScript sort, which is highly optimized and handles nearly sorted inputs exceptionally well. As a result, the relative advantage of coordinated repair is reduced in this environment compared to Rust implementation. The JavaScript benchmarks are still being refined and will be reported in a later revision. Until then, the Rust implementation provides the primary and authoritative performance characterization.

6 Future Work

This work suggests several directions for future investigation:

- **Average-case movement bounds:** Given that the empirical results suggest multi-fold speedups over NativeSort for the Rust implementation, it would be valuable to derive tighter average-case bounds on data movement M in Theorem 8.
- **Structured workload analysis:** The current evaluation relies on randomized updates, whereas many real workloads exhibit additional structure, such as gradual value changes in leaderboards or localized updates in interactive list views. Studying such patterns may reveal regimes where DeltaSort's advantages are amplified or diminished.
- **Study managed environments:** Performance variance in managed environments warrants deeper investigation. Understanding the impact of factors such as garbage collection, and JIT compilation will help in explaining the observed performance characteristics.
- **Block-structured storage:** Although this work focuses on in-memory arrays, the update-aware model naturally extends to block-structured storage as well. Exploring how DeltaSort-style segmentation interacts with page- or block-based layouts may clarify its applicability to database and external-memory settings.

7 Conclusion

This paper introduced *DeltaSort*, an incremental repair algorithm for maintaining sorted arrays. The central insight is that pre-sorting updated values induces *directional segmentation*: dirty elements naturally partition into segments that can be repaired independently.

DeltaSort leverages this segmentation through stack-based processing. LEFT-moving elements are repaired immediately with progressively narrowed search ranges, while RIGHT-moving elements are deferred and processed in reverse order to ensure stable target positions. This segmentation avoids redundant comparisons and overlapping element movement that arise from repeated binary insertion.

An experimental evaluation in Rust demonstrates that DeltaSort consistently outperforms both blind native sorting and repeated binary insertion across a wide range of array sizes and update volumes. In practice, DeltaSort remains advantageous until approximately 25% of the array is updated, providing a clear and actionable decision boundary.

More broadly, this work highlights the value of integrating application-level update information into core algorithms. When sorting routines are informed of which elements changed, batching and segmentation becomes possible, enabling performance improvements that blind algorithms cannot realize. DeltaSort illustrates how modest structural insight—segmentation combined with disciplined processing order—can yield 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] Shubham Dwivedi. DeltaSort: Reference implementation and benchmarks. `https://github.com/shudv/deltasort`, 2026. Rust and JavaScript implementations with unit tests and benchmark suite.

[5] 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.

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

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

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

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

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

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