

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.

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 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 small subset of elements and the indices of those updates are already known. Nevertheless, this information is typically discarded, and systems fall back to blind re-sorting or independent incremental repairs. 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 $D = \{d_1, \dots, d_k\} \subseteq \{0, \dots, n-1\}$ 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

DeltaSort operates in two phases:

1. **Phase 1 (Preparation):** Extract dirty values, sort them, write back to dirty positions in index order. This establishes monotonicity among dirty elements.
2. **Phase 2 (Repair):** Process dirty indices left-to-right, immediately repairing LEFT violations while deferring RIGHT violations to a stack. After the left-to-right pass, flush the stack in LIFO order to process RIGHT violations.

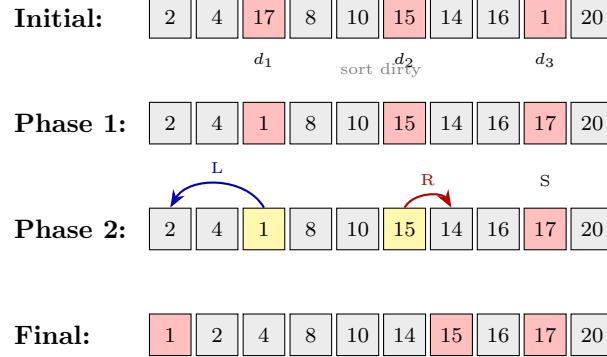


Figure 1: DeltaSort example with 10 elements and 3 dirty indices. Dirty indices $\{2, 5, 8\}$ receive values $\{17, 15, 1\}$. Phase 1 sorts these to $\{1, 15, 17\}$ and writes back. Phase 2 identifies directions: element 1 at index 2 is LEFT (moves to position 0), element 15 at index 5 is RIGHT (moves to position 6), and element 17 at index 8 is STABLE.

4.2 Key Insight: Directional Segmentation

The central insight behind DeltaSort is that pre-sorting the dirty values induces a segmentation of updates by *direction of movement*. After Phase 1, each dirty element either violates order with its left neighbor (LEFT), violates order with its right neighbor (RIGHT), or is already locally ordered (STABLE). Moreover, adjacent dirty elements with the same direction form contiguous directional segments.

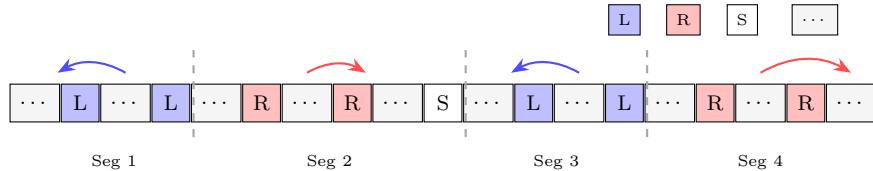


Figure 2: Segmented view of an array with 9 dirty elements after Phase 1. Each dirty element is classified as LEFT (L), RIGHT (R), or STABLE (S) based on its violation direction. Consecutive dirty elements of the same direction type form a *segment*, delimited by vertical dashed lines. Movement within an L segment is confined to that segment alone (arrows show leftward movement). Movement within an R segment can span into the following L segment (arrows show rightward movement). Crucially, all movements remain *bounded*—they never extend beyond the adjacent segment boundary. This boundedness, enabled by Phase 1’s monotonicity, is key to DeltaSort’s efficiency. Boxed “...” represents arbitrarily many clean elements.

This segmentation enables localized repair. LEFT-moving elements can be processed immediately in a left-to-right scan using progressively narrowed search ranges, since all elements to their left are already stable. In contrast, RIGHT-moving elements cannot be safely placed until elements in the next L-segment are handled. DeltaSort therefore defers RIGHT segments using a stack and processes them in LIFO order after the scan completes. Stack-based processing ensures that when a RIGHT-moving element is repaired, the impacted segment is already fixed. This coordination avoids overlapping data movement and redundant search.

4.3 Detailed Algorithm

Algorithm 1 DeltaSort

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

Ensure: A is sorted

```

1: Phase 1: Prepair
2:  $\text{dirty} \leftarrow \text{sort}(D)$                                      ▷ Sort indices ascending
3:  $\text{values} \leftarrow [A[d] : d \in \text{dirty}]$ 
4:  $\text{values} \leftarrow \text{sort}(\text{values}, \text{cmp})$ 
5: for  $i \leftarrow 0$  to  $|\text{dirty}| - 1$  do
6:    $A[\text{dirty}[i]] \leftarrow \text{values}[i]$ 
7: end for

8: Phase 2: Repair
9:  $\text{stack} \leftarrow []$ ;  $\text{leftBound} \leftarrow 0$ 
10: for  $p \leftarrow 0$  to  $|\text{dirty}| - 1$  do
11:    $i \leftarrow \text{dirty}[p]$ 
12:    $d \leftarrow \text{DIRECTION}(A, i)$ 
13:   if  $d = \text{LEFT}$  then
14:      $\text{FLUSHSTACK}(\text{stack}, i - 1)$                                ▷ Flush before processing LEFT
15:      $t \leftarrow \text{BINARYSEARCHLEFT}(A, A[i], \text{leftBound}, i - 1)$ 
16:      $\text{MOVE}(A, i, t)$ 
17:      $\text{leftBound} \leftarrow t + 1$ 
18:   else
19:      $\text{stack.push}(i)$ 
20:   end if
21: end for

22: // Flush remaining RIGHT violations
23:  $\text{FLUSHSTACK}(\text{stack}, n - 1)$ 


---


1: function  $\text{DIRECTION}(A, i)$ 
2:   if  $i > 0 \wedge \text{cmp}(A[i - 1], A[i]) > 0$  then
3:     return LEFT
4:   else if  $i < n - 1 \wedge \text{cmp}(A[i], A[i + 1]) > 0$  then           ▷ Violates order with left neighbor
5:     return RIGHT
6:   else
7:     return STABLE
8:   end if
9: end function

```

```

1: function FLUSHSTACK(stack, rightBound)
2:   while stack  $\neq \emptyset$  do
3:      $s \leftarrow \text{stack.pop}()$                                  $\triangleright$  Process in LIFO order
4:     if DIRECTION( $A, s$ ) = RIGHT then
5:        $t \leftarrow \text{BINARYSEARCHRIGHT}(A, A[s], s + 1, \text{rightBound})$ 
6:       MOVE( $A, s, t$ )
7:     end if
8:   end while
9: end function

1: function MOVE( $A, from, to$ )
2:    $v \leftarrow A[from]$ 
3:   if  $from < to$  then
4:     Shift  $A[from + 1..to]$  left by one
5:   else if  $from > to$  then
6:     Shift  $A[to..from - 1]$  right by one
7:   end if
8:    $A[to] \leftarrow v$ 
9: end function

```

4.4 Why Phase 1 Creates Segments

Example 2 (Benefit of Pre-sorting). Consider array $[1, 8, 5, 2, 9]$ with dirty indices $\{1, 3\}$ (original sorted array was $[1, 3, 5, 7, 9]$, positions 1 and 3 were updated to 8 and 2).

With standard binary insertion (extract-then-insert), the algorithm would:

- Extract dirty values $\{8, 2\}$, leaving placeholders or compacting
- Insert each back via binary search over the *entire* remaining array

This is correct but each insertion searches the full range.

Phase 1 sorts $\{8, 2\} \rightarrow \{2, 8\}$ and writes to positions $\{1, 3\}$, yielding $[1, 2, 5, 8, 9]$. This establishes *monotonicity* among dirty values, which has two critical consequences:

1. **Bounded search ranges:** Once a LEFT-moving element settles at position t , subsequent LEFT searches need only consider $[t + 1, \dots]$.
2. **Directional coherence:** Dirty elements naturally form segments where all elements in a segment move in the same direction, enabling coordinated processing.

Lemma 3 (Phase 1 Establishes Monotonicity). *After Phase 1, for any dirty indices $d_i < d_j$, it holds that $\text{cmp}(A[d_i], A[d_j]) \leq 0$.*

Proof. Phase 1 sorts dirty values and assigns the i -th smallest value to the i -th smallest index. Thus $A[d_i] \leq A[d_j]$ for $d_i < d_j$. \square

Remark 4 (Bounded Search via `leftBound`). The implementation maintains a `leftBound` variable that advances after each LEFT move. When a dirty element moves from index i to position $t < i$, the algorithm sets `leftBound` = $t + 1$. Subsequent LEFT searches only consider $[\text{leftBound}, i - 1]$, not $[0, i - 1]$. This progressively narrows the search range as LEFT-moving elements are processed.

The monotonicity established by Phase 1 enables DeltaSort's efficiency through segmentation:

1. **Directional segments:** After Phase 1, dirty elements form contiguous segments where all elements in a segment have the same direction (LEFT, RIGHT, or STABLE). Figure 2 illustrates this segmentation.
2. **Stack-based coordination:** RIGHT-moving elements are pushed to a stack and processed in LIFO order. This ensures that when processing element at index s , all elements at indices $> s$ have already been processed, so s 's target is stable.
3. **Progressive search narrowing:** The `leftBound` variable ensures that LEFT searches become progressively narrower as elements are processed.
4. **STABLE detection:** After pre-sorting, some dirty elements may already be in their correct position relative to neighbors, requiring no movement.

Example 5 (Movement Cancellation). Consider array $[1, 3, 5, 7, 9]$ with dirty indices $\{1, 3\}$ receiving values 8 and 2 respectively, yielding $[1, 8, 5, 2, 9]$.

Binary Insertion (extract-then-insert):

- Extract values at indices 3, 1 (descending): array becomes $[1, 5, 9]$
- Insert 8: finds position 2, array becomes $[1, 5, 8, 9]$ (9 shifts right)
- Insert 2: finds position 1, array becomes $[1, 2, 5, 8, 9]$ (5, 8, 9 shift right)

Clean element 5 shifted during extraction and again during insertion.

DeltaSort:

- Phase 1: Sort $\{8, 2\} \rightarrow \{2, 8\}$, write to indices $\{1, 3\}$: $[1, 2, 5, 8, 9]$
- Phase 2: Check directions—both elements are now STABLE!

Clean element 5 never moved. The pre-sorting phase reassigned values such that both dirty elements landed in positions where they satisfy ordering constraints immediately.

4.5 Design Rationale

Why flush before LEFT? When a LEFT-moving dirty element is encountered at index i , any RIGHT-moving elements on the stack (with indices $< i$) must be processed first. Otherwise, moving the LEFT element could shift the RIGHT elements' positions, invalidating their stack indices.

Why LIFO order? Stack elements are pushed in ascending index order. LIFO processing (highest index first) ensures that when the element at index s is moved, elements at lower indices haven't shifted yet, so their stack indices remain valid.

Why re-check Direction? After Phase 1 writes or after flushing, an element's direction may change (a STABLE element might become RIGHT, or a RIGHT element might become STABLE). The re-check avoids unnecessary moves.

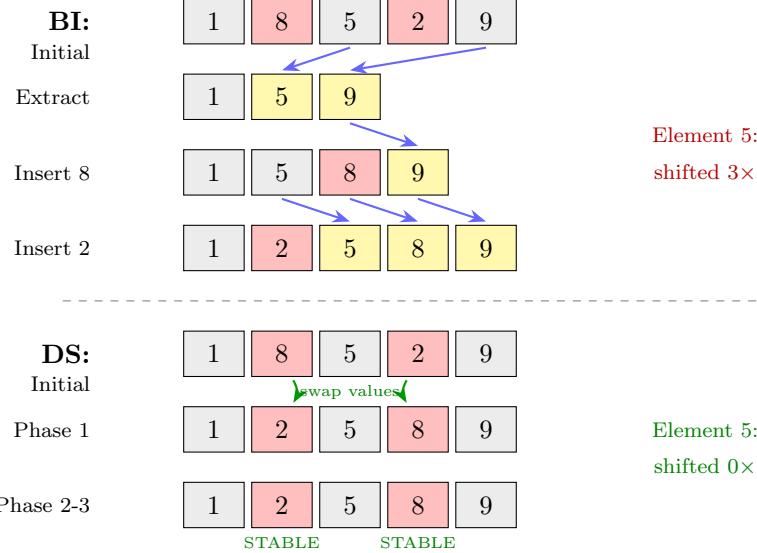


Figure 3: Movement cancellation comparison. Binary Insertion (top) extracts then reinserts, causing element 5 to shift multiple times. DeltaSort (bottom) reassigned values in Phase 1 so that both dirty elements are immediately STABLE—element 5 never moves.

4.6 Correctness Proof

Definition 6 (Violation). A *violation* at index i exists if $\text{cmp}(A[i - 1], A[i]) > 0$ (for $i > 0$) or $\text{cmp}(A[i], A[i + 1]) > 0$ (for $i < n - 1$).

Lemma 7 (Violations Only at Dirty Boundaries). *After Phase 1, violations can only exist at boundaries involving dirty indices. Specifically, a violation at position i implies $i \in D$ or $i + 1 \in D$.*

Proof. Clean elements retain their values from the original sorted array. For two adjacent clean elements at positions i and $i + 1$, neither has changed, so $\text{cmp}(A[i], A[i + 1]) \leq 0$ (sorted order preserved). Violations only occur where a dirty element is adjacent to another element with which it violates order. \square

Lemma 8 (Move Correctness). *After $\text{MOVE}(A, i, t)$ where $t = \text{BINARYSEARCHLEFT/RIGHT}(\dots)$:*

1. *The element originally at $A[i]$ is now at position t*
2. $\text{cmp}(A[t - 1], A[t]) \leq 0$ (*if $t > 0$*)
3. $\text{cmp}(A[t], A[t + 1]) \leq 0$ (*if $t < n - 1$ and within search bounds*)

Proof. Binary search finds the correct insertion point by comparing with neighbors. The element is placed where it satisfies order constraints with its new neighbors. \square

Lemma 9 (Stack Index Validity). *When an index s is popped from the stack and processed, s still refers to the same element that was pushed.*

Proof. Consider when s was pushed: it was a dirty index with direction RIGHT or STABLE. Between push and pop, two types of operations can occur:

(a) *LEFT repairs:* These move elements at index $i > s$ to positions $t \leq i - 1$. Since $s < i$, the region $[t, i - 1]$ is to the right of s ; elements at s don't shift.

(b) *RIGHT repairs from stack flushes*: These process indices in LIFO order. When s is popped, all indices greater than s have already been processed. Their movements shift elements to the right, in regions $[s' + 1, t']$ where $s' > s$. Since $s < s'$, position s is unaffected.

In both cases, the element at index s hasn't moved, so s remains valid. \square

Theorem 10 (Correctness). *DeltaSort produces a correctly sorted array.*

Proof. By Lemma 7, after Phase 1, violations only involve dirty indices. Figure 2 illustrates how the array can be viewed as segments where dirty elements move within bounded regions without crossing segment boundaries. Each dirty index is processed exactly once in Phase 2:

- If Direction is LEFT: processed immediately with Move
- If Direction is RIGHT or STABLE: pushed to stack, later processed (if still RIGHT)

By Lemma 9, stack indices remain valid when processed. By Lemma 8, each Move places the element correctly. After all dirty indices are processed, no violations remain (all were fixed), so the array is sorted. \square

4.7 Algorithm Analysis

Theorem 11 (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: $O(M)$. \square

Theorem 12 (Space Complexity). *DeltaSort uses $O(k)$ auxiliary space.*

Theorem 13 (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 14 (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.

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 $n \in \{1K, 10K, 50K, 100K, 500K, 1M\}$. 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 ($20 \leq k \leq 200$).

Figure 4 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 5 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.

Table 2: Execution time (μ s) 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

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%

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.

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 $\sim 25\%$ 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

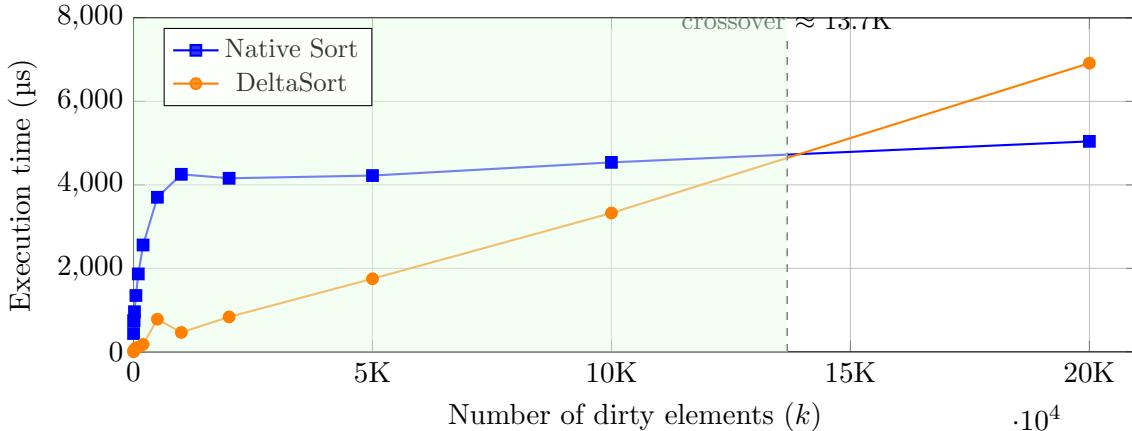


Figure 4: DeltaSort vs. Native Sort for $n = 50,000$. The crossover point occurs at $k \approx 13,700$ (27% of n). The shaded region indicates where DeltaSort is faster.

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 $8\times$ to over $100\times$. This is because BI’s $O(kn)$ extraction 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

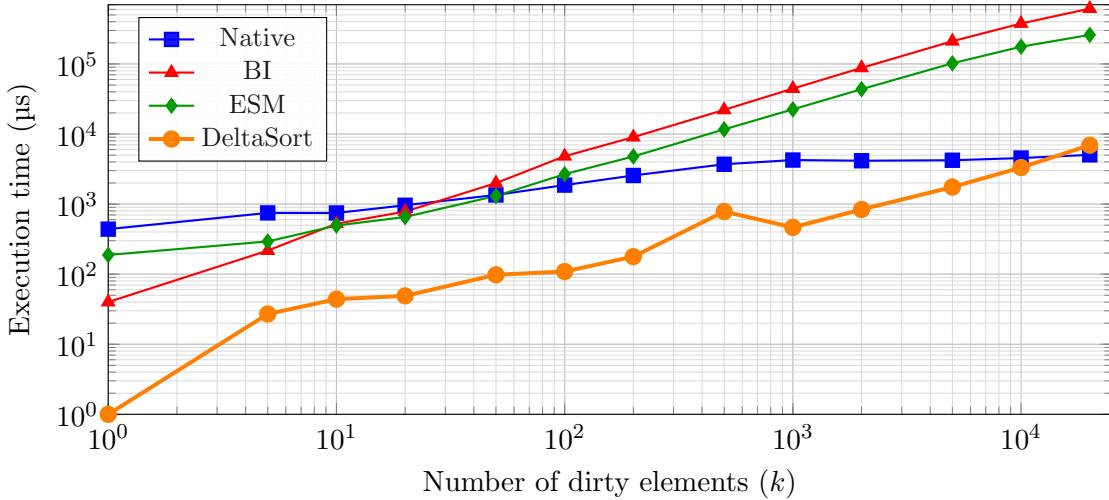


Figure 5: Execution time comparison of all algorithms for $n = 50,000$ (log-log scale). DeltaSort (orange) is consistently fastest across all tested k values except $k = 20,000$ where Native sort wins. Note the orders-of-magnitude gap between DeltaSort and BI/ESM.

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.