The Hong Kong Polytechnic University
Department of Computing

COMP4913 Capstone Project

Report (Interim)

Java-implemented Powersort: Worth the Sacrifice on Maintainability for Performance?

Student Name: YEUNG Tsz Lok

Student ID No.: 22076383D

Programme-Stream Code: 61435-FCS

Supervisor: Prof CAO Yixin

Co-Examiner: Prof LI Bo

2nd Assessor: Prof LIU Yang Veronica

Submission Date: 09 January 2026

Abstract

This interim report details the progress of a capstone project to design and implement a Java-based **Powersort algorithm** by migrating optimized logic from the CPython 3.11 source code. The project addresses the **performance gap** in the Java standard library, which continues to rely on Timsort despite recent advancements in nearly-optimal mergesorts. Key objectives include:

- Translating Powersort's run management logic from Python to Java,
- Establishing correctness validation and evaluate the migrated algorithm's performance,
- Exploring design and implementation processes by adhereing Algorithms Engineering principles.

The project focuses on quantifying execution time and memory overhead across diverse data distributions to determine whether the performance gains justify the potential sacrifice in code maintainability. To date, the literature review and core run-generation logic have been completed, providing a foundation for upcoming integration and statistical analysis.

Contents

1 Literature Review

Stable sorting algorithms are foundational in computer science, with widespread adoption across programming language standard libraries and data processing frameworks. This section surveys the evolution of adaptive mergesort variants, tracing the lineage from Timsort through the theoretical refinements of Powersort to multiway generalizations. These works establish both the theoretical foundations and practical optimizations that inform our Java re-engineering of Powersort.

1.1 Timsort: Adaptive Mergesort in Practice

Timsort, introduced by Peters [1] in 2002, represents a landmark achievement in practical sorting algorithm design. Originally developed for Python but subsequently adopted in Java, Android, JavaScript (V8), and numerous other platforms, Timsort combines conceptual simplicity with empirical effectiveness.

The core innovation lies in exploiting *natural runs*—maximal monotonic subsequences present in real-world data. Rather than treating all input as random, Timsort decomposes the sequence into runs of both non-decreasing and strictly decreasing elements, reversing the latter in-place to maintain stability. A minimum run length (typically 32–64 elements) is computed dynamically and enforced through binary insertion sort, ensuring balanced merge trees on random data while preserving the benefits of longer natural runs in partially sorted inputs.

Timsort's merge policy operates via a stack of pending runs, employing a set of heuristic rules to determine when merges should occur. The rules are designed to maintain invariants on run lengths, specifically ensuring they grow at least as fast as Fibonacci numbers. This keeps the stack height logarithmic while dynamically balancing merge operations. The algorithm further optimizes merges through *galloping mode*—exponential search switching that reduces comparisons when one run dominates during a merge [1].

Critically, Timsort's worst-case complexity remained unproved for over a decade after its introduction, despite its widespread adoption. As Peters noted, the algorithm was announced to run in $O(n \log n)$ without formal justification, raising concerns about the reliability of deployments in production systems.

1.2 Rigorous Analysis of Timsort: Closing the Theoretical Gap

Auger, Jugé, Nicaud, and Pivoteau [2] provided the first rigorous proof that Python's Timsort achieves $O(n \log n)$ worst-case complexity, resolving a fundamental theoretical question. More significantly, they established that Timsort runs in $O(n + n \log \rho)$ time, where ρ is the number of input runs—thereby proving optimal adaptive behavior with respect to presortedness.

Their analysis employs a sophisticated amortized argument using tokens (denoted ♢ and ♡) credited to elements as they enter the stack or decrease in height. Tokens are spent during merges, with the invariant that balances remain non-negative. The key insight is that the merge policy's heuristic rules, while intricate, enforce four invariants on the topmost stack entries (Lemma 5 in [2]):

$$\begin{gathered} r_{i+2} > r_{i+1} + r_i \quad (i \geq 3) \\ r_2 < 3r_3, \quad r_3 < r_4, \quad r_2 < r_3 + r_4 \end{gathered} \tag{1.1}$$

These constraints ensure the stack height remains $O(\log n)$ and that run lengths grow exponentially (Fibonacci-like), bounding total merge costs.

The analysis further reveals that Timsort's execution can be decomposed into *starting sequences* (runs being pushed with subsequent forced merges, costing $O(n)$ total) and *ending sequences* (merges triggered by the heuristic rules, costing $O(n \log \rho)$ via a common pool of tokens). This decomposition elegantly explains Timsort's superior performance on partially sorted data: as $\rho \to 1$ (fully sorted), the algorithm approaches linear time.

Notably, Auger et al. also uncovered an algorithmic bug in Java's Timsort implementation—a violation of the stated merge invariant that had gone undetected. This discovery underscores the importance of rigorous algorithm analysis and motivated subsequent work on more transparent sorting strategies.

1.3 Nearly-Optimal Mergesorts: Powersort and Peeksort

While Auger et al. proved Timsort's $O(n + n \log \rho)$ optimality, Munro and Wild [3] revealed that Timsort does not achieve the best achievable constants. Specifically, Buss and Knop showed that on certain inputs, Timsort incurs at least 1.5× the minimum necessary merge cost (in terms of element moves). This suboptimality motivated the design of algorithms that provably achieve nearly-optimal merging orders while maintaining the practical efficiency requirements of production sorting.

Munro and Wild's key insight is that the problem of finding an optimal merge order is equivalent to constructing a nearly-optimal *alphabetic binary search tree*. Given run lengths $L_1, \ldots, L_r$, the minimum merge cost is:

$$M_{\text{opt}} = H\left(\frac{L_1}{n}, \ldots, \frac{L_r}{n}\right) \cdot n - O(n), \tag{1.2}$$

where $H(p_1, \ldots, p_r) = \sum_{i=1}^{r} p_i \lg(1/p_i)$ is the binary Shannon entropy. This lower bound follows from information theory: sorting r runs of specified lengths requires at least this many comparisons in the worst case (distinct elements).

Munro and Wild present two algorithms that achieve this bound:

Peeksort (top-down): Simulates Mehlhorn's Method 1 for nearly-optimal BSTs. It recursively partitions the array by finding the run boundary closest to the midpoint, then recursively sorts left and right segments. The merge tree naturally mimics an alphabetic tree with optimal search cost

$C \leq H + 2$ (Theorem 1 in [3]). Merge cost is bounded by $H(L/n) \cdot n + 2n - (r + 2)$, optimal up to $O(n)$.

Powersort (stack-based, bottom-up): The preferred algorithm for practical deployment. Powersort assigns each run boundary B_j a *power* P_j, defined as the minimum depth at which a power-of-2 boundary exists in the *midpoint interval* of the two adjacent runs. Formally:

$$P_j = \min \left\{ p \in \mathbb{N} : \exists c \in \mathbb{N}, c \cdot 2^{-p} \in (a_j, b_j] \right\}, \tag{1.3}$$

where $(a_j, b_j]$ is the interval between the midpoints of runs $j-1$ and j. These powers define the node structure of an optimal merge tree, specifically the Cartesian tree of the power sequence (Lemma 4 in [3]).

Powersort maintains a stack of runs in order of weakly increasing powers. When a new run boundary with power P is encountered, if P is smaller than the top power on the stack, all larger powers are popped and merged; otherwise, the new run is pushed. This process implicitly constructs the optimal merge tree left-to-right without explicitly storing all runs or the full tree. The algorithm guarantees stack height $O(\log n)$ and merge cost $\leq H(L/n) \cdot n + 2n$ (Theorem 5 in [3]), matching the information-theoretic lower bound.

Crucially, Powersort's design is *principled* and *simple*. Unlike Timsort's ad-hoc heuristics, Powersort's merging order is theoretically justified, admitting formal proof of optimality without intricate case analyses. This simplicity enabled subsequent generalizations and is instrumental for the present work.

Table 1.1: Comparison of Timsort and Powersort characteristics

Property	**Timsort**	**Powersort**
Worst-case time	$O(n \log n)$	$O(n \log n)$
Adaptive complexity	$O(n + n \log \rho)$	$O(n + n \log \rho)$
Merge cost optimality	Suboptimal ($\geq 1.5\times$ optimal)	Near-optimal ($\leq H \cdot n + 2n$)
Stack height	$\sim 1.44 \lg n$	$\lg n$
Merge policy	Heuristic (5 rules)	Principled (power-based)
Galloping mode	Yes	Optional

Empirical validation by Munro and Wild demonstrates that Powersort incurs negligible overhead compared to standard mergesort on random inputs (no exploitable structure), while achieving 20% speedups on presorted inputs. Crucially, Powersort handles adversarial inputs (e.g., the Buss-Knop worst case for Timsort) optimally, whereas Timsort suffers 30% overhead.

1.4 Multiway Powersort: Generalizing to Modern Hardware

Gelling, Nebel, Smith, and Wild [4] extend Powersort to k-way merging ($k > 2$), motivated by modern CPU-memory disparity. Over decades, CPU speed has increased far faster than memory bandwidth, making memory transfers the dominant cost in internal sorting. Multiway algorithms reduce the number of passes over data, decreasing scanned elements and cache misses.

The theoretical extension is natural: replace the virtual binary tree with a virtual complete k-ary tree and define k-way powers analogously:

$$P_j^{(k)} = \min\left\{p \in \mathbb{N} : \exists c \in \mathbb{N}, c \cdot k^{-p} \in (a_j, b_j]\right\}. \tag{1.4}$$

This yields the following theorem (Theorem 3.3 in [4]):

$$M \leq \frac{1}{\lg k} H(L/n) \cdot n + 2n. \tag{1.5}$$

For $k = 4$ (the practical choice), this implies merge cost is reduced by a factor of $\lg 4 = 2$ compared to binary Powersort. In terms of comparisons, multiway merging uses a tournament tree with cost $\leq \lceil \lg k \rceil$ per output element, yielding:

$$C \leq \frac{\lceil \lg k \rceil}{\lg k} H(L/n) \cdot n + O(n). \tag{1.6}$$

Stack height remains $O(\log n)$: specifically $(k-1)\lceil \log_k(n) + 1 \rceil$ (Proposition 3.5 in [4]).

Table 1.2: Empirical performance comparison: 4-way vs. 2-way Powersort (n = 10^8 integers, random runs with expected length $\sqrt{n}$, data from [4])

Metric	**4-way**	**2-way**	**Ratio**
L1 cache read misses	90,998,034	168,472,841	54.0%
L1 cache write misses	90,941,221	168,392,912	54.0%
Instructions executed	14.39B	15.59B	92.3%
Cycle estimate	26.83B	36.64B	73.2%
Merge cost (total elements)	678M	1,298M	52.2%
Comparisons	1.40B	1.40B	100.2%

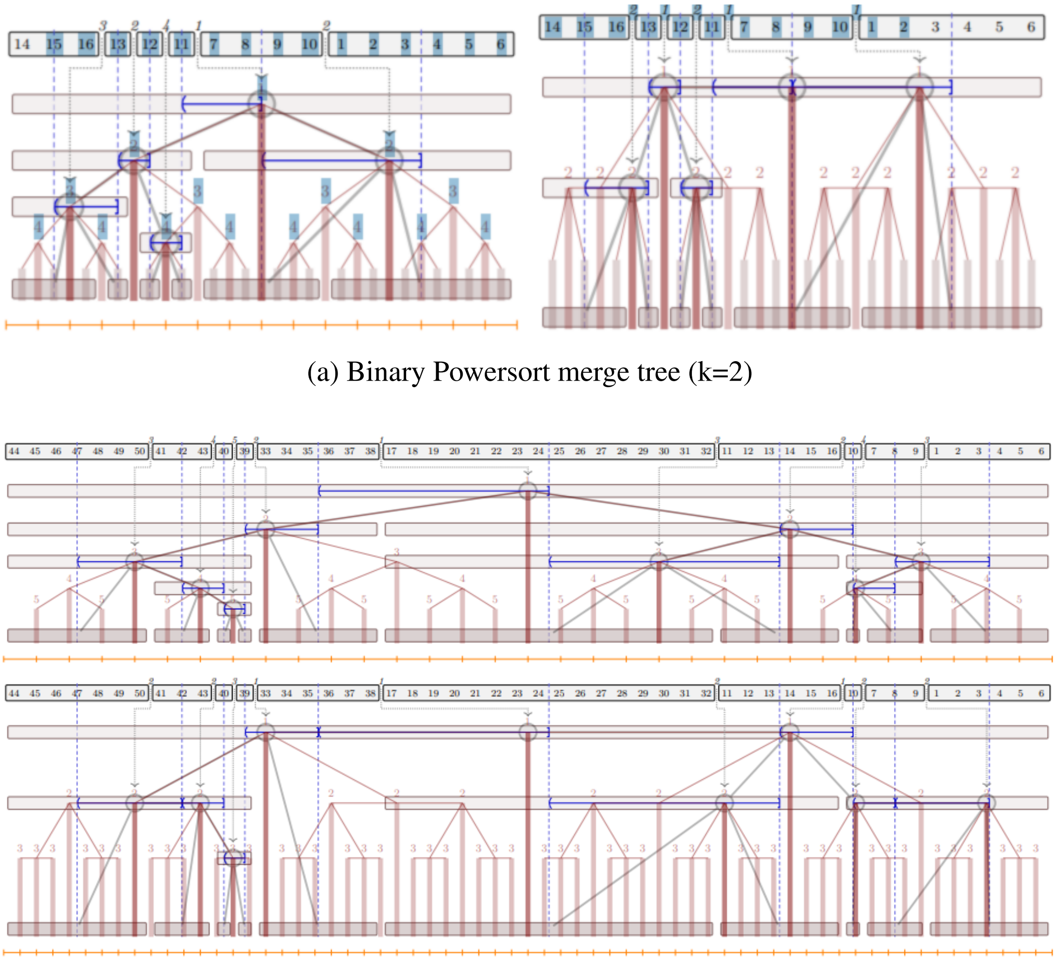
Note: Cycle estimate = instructions + 10 × L1-misses + 100 × LL-misses

Gelling et al. also provide the first efficient implementation of 4-way Powersort in C++, demonstrating empirically that:

- 4-way Powersort achieves 15–20% speedup over binary Powersort across input sizes (10^5 to 10^8 elements).
- Speedups are robust to element type (4-byte integers, 16-byte records) and presortedness levels.
- Scanned elements (memory transfers) are reduced to 54% of binary Powersort, explaining the wall-clock speedup.
- Cachegrind analysis validates that reductions in L1 cache misses (46% decrease) correlate directly with performance improvements.

Table 1.2 presents detailed performance metrics from the C++ implementation, illustrating the substantial reduction in memory operations that drives the observed speedup. The merge cost reduction to 52.2% closely approaches the theoretical optimum of 50% ($1/\lg 4$), while L1 cache misses are reduced by 46%, confirming that memory bandwidth rather than CPU computation is the primary bottleneck.

A key implementation insight is the use of sentinel values (e.g., $+\infty$) to eliminate pointer comparisons in merge inner loops. Additionally, for types unsupporting sentinels, *merge by stages* (merging until one run is exhausted, then recomputing the shortest remaining run) provides competitive performance with minimal code complexity.



(a) Binary Powersort merge tree (k=2)

(b) 4-way Powersort merge tree (k=4)

Figure 1.1: Conceptual illustration of merge tree construction for Powersort variants (adapted from [4], Figure 1). Each run boundary maps to a node in a virtual perfectly balanced tree (shown in light shading). The midpoint intervals (shown as horizontal ranges) determine which tree node each boundary ”snaps to” based on power assignment. The 4-way variant reduces tree depth by combining pairs of adjacent binary tree levels.

1.5 Synthesis and Implications for Java Implementation

The literature review above has established a clear progression: from Timsort’s empirically effective but theoretically opaque heuristics, through Auger et al.’s rigorous analysis revealing its optimality and limitations, to Munro and Wild’s principled Powersort algorithm achieving provable optimality, culminating in Gelling et al.'s multiway generalization addressing modern hardware constraints.

For a Java re-engineering of Powersort, these works collectively provide:

1. **Theoretical foundation:** Auger et al.'s correctness proof and $O(n+n\log\rho)$ optimality guarantee, applicable to any correct implementation of the power-based merging policy.

2. **Algorithmic correctness:** Munro and Wild's principled design admits straightforward implementation and verification, unlike Timsort's intricate heuristics.

3. **Performance ceiling:** Gelling et al.'s 4-way generalization provides a clear path to leveraging modern CPU-memory tradeoffs while maintaining theoretical guarantees.

4. **Implementation guidance:** Empirical insights on sentinel handling, buffer strategies, and memory-transfer optimization from both C++ and Java reference implementations.

Table 1.1 summarizes the key algorithmic differences between Timsort and Powersort, highlighting why the principled power-based approach enables both theoretical guarantees and practical extensions. The progression from heuristic to principled design (Timsort $\rightarrow$ Powersort) and from binary to multiway merging (2-way $\rightarrow$ 4-way) represents algorithmic refinement informed by both theory and hardware evolution.

Our implementation contributes the first enterprise-grade Java port of Powersort, addressing JVM-specific challenges (memory model, object vs. primitive arrays, garbage collection overhead) while validating Gelling et al.'s multiway approach in a different execution environment.

2 Sorting Algorithm Deconstruction

This section provides a detailed examination of the internal mechanisms of stable sorting algorithms, tracing their evolution from classical Mergesort through Timsort's adaptive heuristics to Powersort's principled optimization. We analyze each algorithm's paradigm, merge policy, and inherent trade-offs, illustrated with concrete examples demonstrating how their distinct approaches handle the same input data.

2.1 Classical Mergesort: The Divide-and-Conquer Foundation

2.1.1 Algorithm Paradigm

Mergesort, invented by John von Neumann in 1945, epitomizes the *divide-and-conquer* algorithmic paradigm. The algorithm recursively bisects the input array until reaching base cases of size one, then systematically merges these trivial sorted subarrays back into larger sorted segments.
The three phases are:

1. **Divide**: Split the array at the midpoint $\lfloor n/2 \rfloor$, creating two subarrays of approximately equal size.
2. **Conquer**: Recursively apply Mergesort to each subarray until reaching subarrays of size 1 (which are trivially sorted).
3. **Combine**: Merge two sorted subarrays into a single sorted array via linear-time two-way merge.

2.1.2 Merge Policy

Mergesort's merge policy is *non-adaptive*: it always bisects at the computed midpoint regardless of data properties. This deterministic structure guarantees a perfectly balanced binary tree of depth $\lceil \lg n \rceil$, yielding the canonical $O(n \log n)$ time complexity.
The two-way merge operation scans both sorted subarrays left-to-right, repeatedly selecting the smaller of the two current elements. This requires temporary storage of size $n/2$ (the smaller subarray is copied to auxiliary space).

2.1.3 Characteristics and Pitfalls

Strengths:

- **Predictable performance**: $\Theta(n \log n)$ comparisons and $\Theta(n \log n)$ data movements in all cases (best, average, worst).

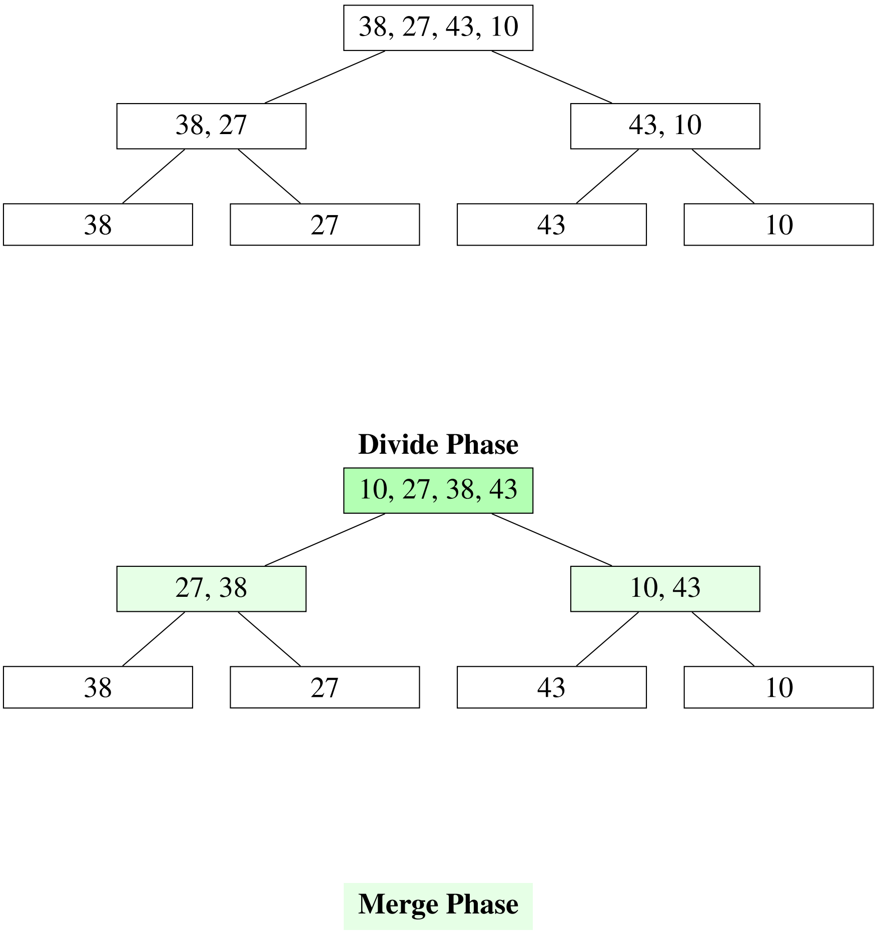


Figure 2.1: Mergesort recursion tree for input [38, 27, 43, 10]. Left: divide phase recursively splits array at midpoints. Right: merge phase combines sorted subarrays bottom-up (green indicates merging operations).

- **Stability**: Equal elements maintain relative order due to tie-breaking favoring the left subarray.
- **Parallelizable**: Independent recursive branches can execute concurrently.

Pitfalls:

- **No adaptivity**: Fully sorted input still requires $\Theta(n \log n)$ operations; cannot exploit pre-existing order.
- **Memory overhead**: Requires $O(n)$ auxiliary space for merging.
- **Cache behavior**: Top-down recursion exhibits poor spatial locality on large arrays.

2.2 Timsort: Adaptive Hybrid Mergesort

2.2.1 Algorithm Paradigm

Timsort, designed by Tim Peters in 2002 for Python's standard library [1], represents a paradigm shift toward *adaptive sorting*. Rather than imposing a fixed divide-and-conquer structure, Timsort exploits *natural runs*—existing monotonic subsequences in the data—to minimize merge operations.

The algorithm proceeds in a single left-to-right scan:

1. **Run Detection**: Identify the next maximal ascending ($a_i \leq a_{i+1}$) or descending ($a_i > a_{i+1}$) run. Descending runs are reversed in-place to maintain stability.
2. **Minrun Enforcement**: If a natural run has length less than `minrun` (computed dynamically from n), extend it to `minrun` elements via binary insertion sort.
3. **Stack Management**: Push the run onto a stack and invoke merge policy heuristics to determine when to merge pending runs.
4. **Final Merge**: After scanning the entire array, merge all remaining runs on the stack.

2.2.2 Computing Minrun

The `minrun` parameter is chosen in range $[32, 64]$ such that $n/\texttt{minrun}$ is close to (but not exceeding) a power of 2. This ensures balanced merges when no natural runs exist. The algorithm takes the top 6 bits of n and adds 1 if any lower bits are set.

Example: For $n = 2112 = \texttt{0x840}$:

- Top 6 bits: $\texttt{0x21} = 33$
- Remaining bits: nonzero
- $\texttt{minrun} = 33$ (not 32), yielding $2112/33 = 64 = 2^6$ balanced merges

2.2.3 Merge Policy: The Stack Invariants

Timsort maintains a stack of pending runs, with the merge policy governed by two heuristic invariants on the topmost three run lengths A, B, C (where C is the most recently added):

$$\begin{aligned} &\text{Invariant 1:} \quad A > B + C \\ &\text{Invariant 2:} \quad B > C \end{aligned} \tag{2.1}$$

When a violation occurs, the algorithm merges either A with B or B with C, choosing whichever pairing involves the smaller run (ties favor C for cache locality). These invariants ensure:

- Run lengths grow at least as fast as Fibonacci numbers: $r_i \geq F_{i+2}$ (where F_k is the k-th Fibonacci number).
- Stack height is bounded by $\lfloor \log_\phi(n) \rfloor \approx 1.44 \lg n$, where $\phi = (1 + \sqrt{5})/2$.

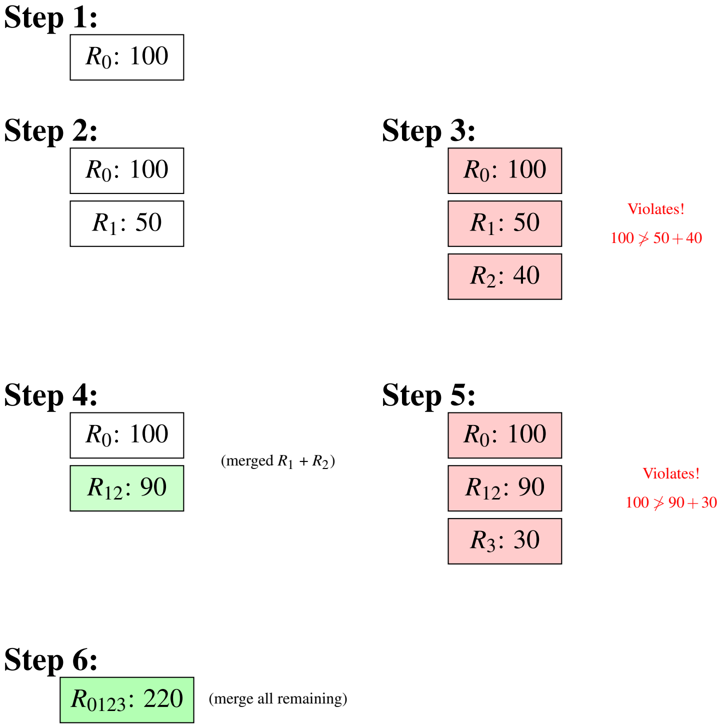


Figure 2.2: Timsort stack evolution for runs of lengths [100, 50, 40, 30]. Red indicates invariant violations; green indicates merge operations. The stack configuration dynamically enforces Fibonacci-like growth to bound stack height.

2.2.4 Galloping Mode

When merging two runs A and B, Timsort initially proceeds in *one-pair-at-a-time mode*, comparing heads of A and B and advancing the smaller element. If one run consistently wins (i.e., provides `MIN_GALLOP` $= 7$ consecutive elements), the algorithm switches to *galloping mode*.

In galloping mode, instead of linear scanning, Timsort uses exponential search to locate where $A[0]$ belongs in B (or vice versa):

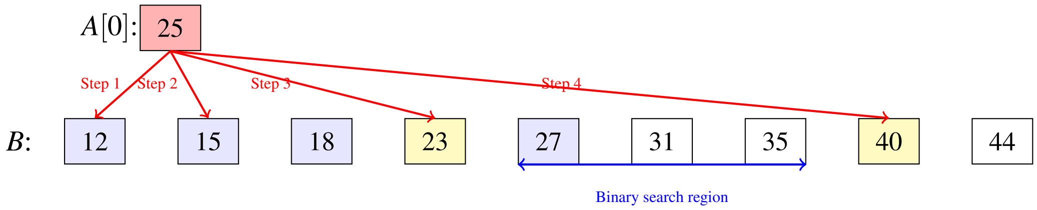
1. Compare $A[0]$ with $B[2^j - 1]$ for $j = 0, 1, 2, \ldots$ until finding the bracketing interval.
2. Perform binary search within the identified $O(2^j)$-sized range.
3. Copy all elements before the found position in one chunk, then continue merging.

Galloping is adaptive: the threshold `min_gallop` decreases when galloping pays off and increases when it doesn't, preventing overhead on random data where long winning runs are improbable.

2.2.5 Characteristics and Pitfalls

Strengths:

- **Adaptive complexity**: $O(n + n \log \rho)$ where ρ is the number of runs; approaches $O(n)$ for nearly-sorted data.
- **Practical efficiency**: Exploits real-world data patterns (partial order, duplicates, recurring sequences).



Galloping finds 25 $\in B$ at index 4:
12, 15, 18, 23 copied in bulk,
then $A[0] = 25$ inserted.

Figure 2.3: Timsort galloping mode: locating $A[0] = 25$ in array B via exponential search (red arrows at positions $2^j - 1$: 0, 1, 3, 7) followed by binary search in the identified interval (blue). Yellow highlights mark the bracketing comparisons.

- **Widespread adoption**: Standard library sort in Python, Java, Android, V8 JavaScript.

Pitfalls:

- **Heuristic complexity**: Five distinct merge rules with numerous edge cases; difficult to verify correctness rigorously.
- **Suboptimal merge cost**: Buss & Knop (2018) demonstrated adversarial inputs incurring $\geq 1.5\times$ optimal merge cost.
- **Galloping overhead**: When $A[0]$ belongs at $B[2]$, galloping requires 4 comparisons vs. 3 for linear search—a 33% increase.
- **Discovered bugs**: Auger et al. [2] uncovered an invariant violation in Java's implementation that went undetected for years.

2.3 Powersort: Principled Optimality

2.3.1 Algorithm Paradigm

Powersort, introduced by Munro and Wild [3] in 2018, retains Timsort's adaptive run-detection framework but replaces the heuristic merge policy with a *theoretically justified* approach rooted in optimal binary search tree construction. The key innovation is assigning each run boundary a *power*—its intended depth in a conceptual merge tree—based on a virtual perfectly balanced tree overlaid on the array.

Like Timsort, Powersort executes a single left-to-right scan with run detection and stack-based merging. The critical difference lies in the merge triggering condition: instead of Fibonacci-based heuristics, Powersort maintains the simple invariant that powers on the stack must be *weakly increasing*.

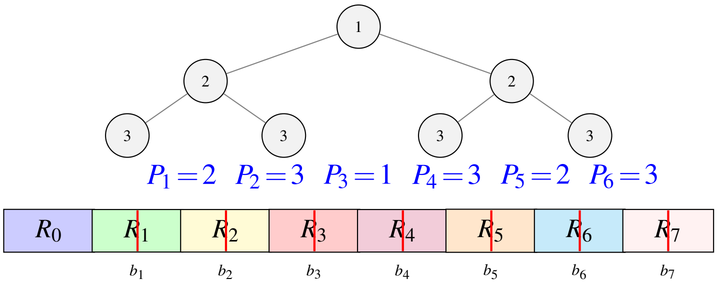
2.3.2 Power Assignment

Given an array of n elements with detected run boundaries at indices $0 = b_0 < b_1 < \cdots < b_r = n$, each boundary b_j (for $1 \leq j < r$) is assigned a power P_j defined as:

$$P_j = \min\left\{ p \in \mathbb{N} : \exists c \in \mathbb{N}, c \cdot 2^{-p} \in \left(\frac{b_{j-1}+b_j}{2n}, \frac{b_j+b_{j+1}}{2n} \right] \right\} \tag{2.2}$$

Intuitively, P_j represents the depth at which boundary b_j "snaps to" a node in a virtual complete binary tree of depth $\lceil \lg n \rceil$ rooted at $[0,n]$. The midpoint interval $\left(\frac{b_{j-1}+b_j}{2}, \frac{b_j+b_{j+1}}{2}\right)$ determines which tree level contains a dyadic rational (multiple of 2^{-p}) within this range.

Geometric Interpretation: Imagine a perfectly balanced binary tree where each internal node at depth p corresponds to a dyadic fraction $c \cdot 2^{-p}$. The power of a run boundary is the shallowest tree level containing a node whose fractional position falls within the boundary's "influence region" (its midpoint interval). This assignment implicitly constructs a Cartesian tree on the power sequence, which approximates the optimal merge tree for the given run lengths [3].



Each boundary ”snaps to” nearest tree node.
Power = depth of that node (gray circles show depths 1, 2, 3).

Figure 2.4: Powersort power assignment for 8 runs. Virtual binary tree (gray) overlays the array; each boundary b_j receives power P_j equal to the depth of the nearest tree node within its midpoint interval (blue labels). Lower powers indicate earlier merges; $P_3 = 1$ marks the final top-level merge.

2.3.3 Merge Policy: The Power Invariant

Powersort maintains a stack of pending runs, each tagged with its left boundary's power. The merge invariant is elegantly simple:

> *Powers on the stack must be weakly increasing from bottom to top.*

When a new run with power P is detected:

1. While the topmost stack entry has power $\geq P$, pop and merge the top two entries.
2. The merged run inherits the smaller of the two powers.
3. Push the new run onto the stack with power P.

This procedure implicitly constructs the Cartesian tree defined by the power sequence [3], which provably yields near-optimal merge cost.

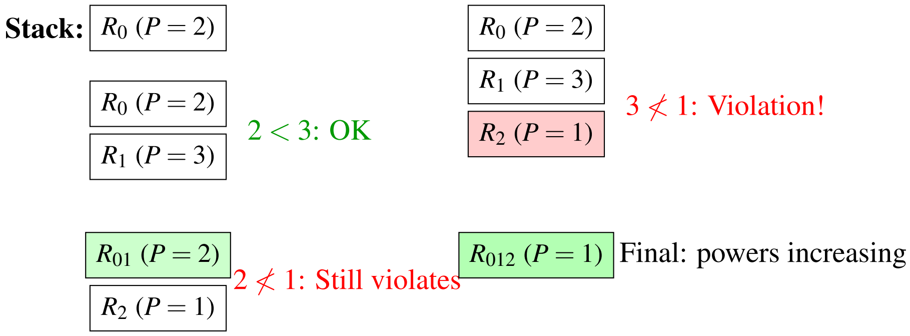


Figure 2.5: Powersort stack evolution with power invariant. When R_2 with $P = 1$ is added (Step 3), it violates the invariant ($3 \not< 1$), triggering cascading merges until powers are weakly increasing. The merged run inherits the minimum power.

2.3.4 Characteristics and Pitfalls

Strengths:

- **Provably optimal**: Merge cost bounded by $H(L/n) \cdot n + 2n$, within $O(n)$ of the information-theoretic lower bound.
- **Simplicity**: Single, intuitive invariant; no complex case analysis or magic constants.
- **Reduced stack space**: Exactly $\lceil \lg n \rceil$ entries (vs. $\lfloor 1.44 \lg n \rfloor$ for Timsort).
- **Extensibility**: Power concept generalizes naturally to k-way merging for $k > 2$ (see [4]).

Pitfalls:

- **No built-in galloping**: While galloping can be added orthogonally, Powersort's design doesn't inherently optimize for skewed merges.
- **Power computation overhead**: Computing midpoint intervals and finding dyadic rationals adds per-boundary cost (though empirically negligible).
- **Recent adoption**: As of 2024, Powersort is replacing Timsort in CPython but not yet adopted in Java or JavaScript standard libraries.

2.4 Comparative Summary

Figure 2.6 visualizes the conceptual evolution from classical Mergesort's rigid structure through Timsort's adaptive heuristics to Powersort's theoretically grounded approach. Each step represents a refinement: Mergesort establishes stability and $O(n \log n)$ worst-case complexity; Timsort adds adaptivity to exploit real-world data patterns; Powersort achieves the same adaptivity with provable optimality and simpler design.

Table 2.1: Algorithmic paradigm comparison: Mergesort, Timsort, and Powersort

Aspect	**Mergesort**	**Timsort**	**Powersort**
Paradigm	Divide-and-conquer	Adaptive hybrid	Principled adaptive
Execution Model	Top-down recursive	Bottom-up iterative	Bottom-up iterative
Run Detection	None (fixed splits)	Natural runs	Natural runs
Merge Trigger	Balanced tree	Fibonacci heuristics	Power invariant
Stack Height	$O(\log n)$ implicit	$\sim 1.44 \lg n$	$\lg n$
Adaptivity	None	High (empirical)	Optimal (proven)
Merge Optimality	Balanced	Suboptimal ($\geq 1.5\times$)	Near-optimal ($H \cdot n + 2n$)
Galloping	No	Yes (adaptive)	Optional
Complexity	$\Theta(n \log n)$	$O(n + n \log \rho)$	$O(n + n \log \rho)$

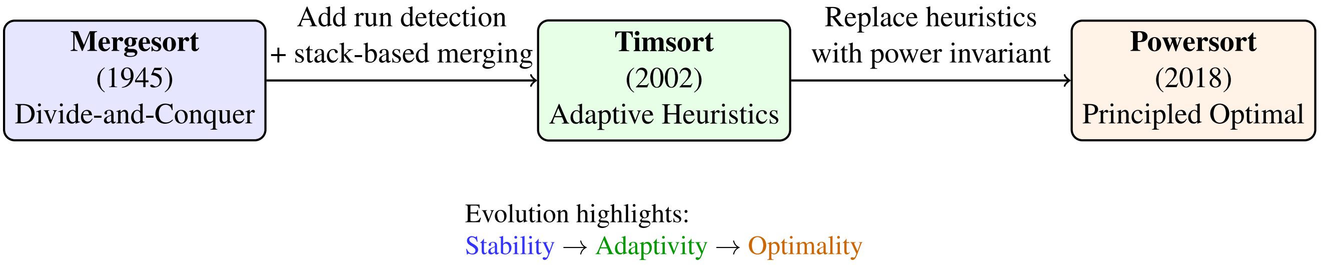


Figure 2.6: Evolution of stable sorting algorithms from Mergesort's foundational divide-and-conquer through Timsort's empirical adaptivity to Powersort's provably optimal merge policy. Each transition addresses limitations of the prior approach while preserving stability and worst-case $O(n \log n)$ complexity.

2.5 Insights gained for Implementation

The progression from Mergesort to Timsort to Powersort reflects an evolution in algorithmic philosophy: from deterministic simplicity, through heuristic pragmatism, to principled optimization. For implementing a Java-based stable sort:

- **Mergesort** provides a reliable baseline with predictable performance but forgoes opportunities to exploit data structure.
- **Timsort** achieves excellent empirical performance on real-world data but at the cost of implementation complexity and subtle correctness concerns.
- **Powersort** offers the "best of both worlds": Timsort's adaptive efficiency with Mergesort's simplicity and provable guarantees.

The power-based approach, extended to k-way merging in Gelling et al. [4], additionally addresses modern hardware constraints by reducing memory transfers—a critical consideration as CPU-memory speed disparity continues to widen. Our Java implementation leverages these insights to provide both theoretical soundness and practical performance optimizations tailored to the JVM execution environment.

3 Environment and Framework

4 Java Powersort Implementation

5 Validation of Algorithm

6 Achieved Objectives

7 Upcoming Works

References

[1] Tim Peters. Cpython list sort. Python Subversion Repository, 2002. Original specification for Timsort. Available at: `https://svn.python.org/projects/python/trunk/Objects/listsort.txt`.

[2] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau. On the worst-case complexity of timsort. *arXiv.org*, 2018. Permalink: `https://julac-hkpu.primo.exlibrisgroup.com/permalink/852JULAC_HKPU/1o78cnh/cdi_arxiv_primary_1805_08612`.

[3] J. I. Munro and S. Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. *arXiv.org*, 2018. Available at: `https://arxiv.org/abs/1805.04154`. DOI: 10.48550/arxiv.1805.04154.

[4] William Cawley Gelling, Markus E. Nebel, Benjamin Smith, and Sebastian Wild. Multiway powersort. *arXiv.org*, 2023. Permalink: `https://julac-hkpu.primo.exlibrisgroup.com/permalink/852JULAC_HKPU/1o78cnh/cdi_arxiv_primary_2209_06909`.