PersiSort: A New Perspective on Adaptive Sorting Based on Persistence

Jens Kristian Refsgaard Schou[*] Bei Wang[†]

Abstract

Adaptive sorting exploits the structure of a partially sorted list—in particular, the sorted segments of a list called runs—to improve its performance. Persistent homology, on the other hand, is a topological data analysis tool that captures a space's topological features at different scales. In this paper, we combine these two seemingly unrelated concepts and introduce a new perspective on adaptive sorting. We introduce a new stable sorting algorithm, referred to as the Persistence Sort (or PersiSort in short), which utilizes the persistence pairs among the local extrema of a list. Given a list of n elements containing r runs with run entropy H, we prove, for the first time, that any adaptive sorting algorithm that uses the two-way-merge subroutine (AdaptMerge) of Carlsson et al. (1990) performs $\mathcal{O}(nH) = \mathcal{O}(n \log r)$ comparisons to merge precomputed runs based on its predetermined merge policy, and is therefore worst-case optimal. Using PersiSort, we then provide a new way to analyze adaptive sorting with a persistence-based arrangement of runs. Unlike previous work such as PowerSort and TimSort, PersiSort does not consider the number of elements in each run but the values of elements in the sorting process. We finally discuss the scenarios when PersiSort outperforms several state-of-the-art adaptive sorting algorithms.

1 Introduction

A sorting algorithm has a basic goal: putting elements from a list into some total order. Adaptive sorting is an active area of research that exploits the structure of a partially sorted list to improve performance. Specifically, it utilizes unique structures in the input called the *runs*, which are segments of the list already in sorted order. Examples of adaptive sorting algorithms include Natural MergeSort [7], TimSort [26], PowerSort [24], and multiway PowerSort [14]. Among those, the first three algorithms use the two-way merges of runs (i.e., AdaptMerge) from Carlsson et al. [7] as subroutines, whereas the multiway PowerSort employs k-way merges of runs.

Persistent homology is a popular tool from topological data analysis (TDA) that captures the topological features of a space at different scales. In its simplest form, given a real-valued function $f : \mathbb{R} \to \mathbb{R}$, persistent homology computes the pairings among local extrema (i.e., local maxima and local minima) of f. These *persistence pairs* encode the topological features of f at different scales.

In this paper, we combine two seemingly unrelated concepts—adaptive sorting and persistence—and introduce a new sorting algorithm, referred to as the Persistence Sort (PersiSort in short, pronounced "Percy sort"), that utilizes the persistence pairs among local extrema of a list. Our contributions include:

- We provide, for the first time, a general worst-case bound for a class of adaptive sorting algorithms. We prove that any adaptive sorting algorithm that uses AdaptMerge of Carlsson et al. [7] (i.e., two-way merges of runs) performs $\mathcal{O}(nH(\ell_1, \ldots, \ell_r))$ comparisons on a list of n elements containing r precomputed runs each with ℓ_i elements, and is, therefore, worst-case comparison optimal. Here, $H(\ell_1, \ldots, \ell_r) = -\sum_{i=1}^{r}(\ell_i/n)\log(\ell_i/n)$ is the entropy of the runs [2].
- Using PersiSort, we provide a new way to analyze adaptive sorting by looking at the arrangement of runs based on the topological notion of persistence. Unlike previous work such as TimSort and PowerSort, PersiSort does not consider the number of elements but the values of elements in the sorting process.
- We demonstrate that PersiSort outperforms several state-of-the-art adaptive sorting algorithms on data distributions where runs have little overlap in their ranges of values. Our experiments suggest ways to improve PowerSort by using AdaptMerge as its merge subroutine.

Finally, we provide an open-source implementation of PersiSort on Github[1].

2 Related Work

Adaptive sorting algorithms. Any sorting algorithm requires worst-case $\Omega(n \log n)$ comparisons to sort a list of n elements. In particular, MergeSort [18] has a $\Theta(n \log n)$ complexity for all inputs. An adaptive sorting algorithm seeks to use the presortedness of the input to make informed decisions about the merges performed.

The first adaptive sorting algorithm uses the number of inversions $\mathrm{Inv}(X)$ in the input list X as a measure of

[*]Aarhus University. jkrschou@gmail.com
[†]University of Utah. beiwang@sci.utah.edu

[1]https://github.com/Nimakii/PersiSort

presortedness. $\mathrm{Inv}(X)$ is the number of pairs of input elements in the wrong order [13]. Given a list X with n elements, Guibas et al. [16] gave the first adaptive sorting algorithm with a complexity $\Omega\left(n \log\left(\frac{\mathrm{Inv}(X)}{n}\right) + n\right)$, using a finger-based balanced search tree. Other adaptive sorting algorithms include BlockSort [21], SplitSort [19], and Adaptive HeapSort [20].

The second type of adaptive sorting algorithm uses *runs* (i.e., presorted subsequences, see Sec. 3.2) as a measure of presortedness. Carlsson et al. [7] introduced AdaptMerge, which uses exponential and binary searches to merge two sorted lists. Given a list X with n elements and r runs, Natural MergeSort [7] detects runs and performs pairwise merges in a balanced way using AdaptMerge, giving a $\Theta(n + n \log r)$ complexity. TimSort [26] puts detected runs on a stack and uses a set of involved rules to decide what and when to merge. TimSort was later shown to have worst-case $\mathcal{O}(n \log n)$ complexity [1] w.r.t. comparison and merge cost. PowerSort [24, 28] is similar to TimSort in the sense that it makes a pass of the input from left to right, and for each new run it detects, it either performs some merges or delays the merges by keeping the runs on a stack. It assigns each adjacent pair of runs a "power" score and applies all delayed merges of higher power. PowerSort has become the standard library sort for CPython since 2022.

Our novel PersiSort algorithm belongs to the second type of adaptive sorting algorithms, where we study the organization of runs in an input list using the notion of persistence [11, 6]. Unlike other adaptive sorting algorithms that focus on the number of elements in each run, PersiSort takes advantage of the values of elements in the sorting process and provides a new perspective on adaptive sorting. Whereas TimSort and PowerSort merge adjacent runs, PersiSort merges runs ordered by persistence pairs.

Persistent homology. Persistent homology is a tool from TDA that captures topological features of data across multiple scales. It has seen a wide range of applications in the study of networks, biological molecules, natural images, time series, etc.; see [9, 23, 25] for introductory texts and surveys. To the best of our knowledge, this is the first time persistence has been utilized in the study of sorting algorithms. The persistent homology of functions from $\mathbb{R}$ to $\mathbb{R}$ is studied in [15] and the windows of [4, 8] are reminiscent but slightly different from the persistence boxes we introduce in Sec. 3.3.

3 Technical Background

3.1 A Review on Persistent Homology

We review the notion of persistent homology in the most straightforward 1-dimensional setting; see [9] for some introductory texts and [10] for a formal treatment.

Let $f : \mathbb{M} \to \mathbb{R}$ be a smooth function defined on a 1-dimensional manifold $\mathbb{M} \subseteq \mathbb{R}$. A point $x \in \mathbb{M}$ is a *critical point* of f if and only if $f'(x) = 0$; otherwise, it is a *regular point*. There are two types of critical points, *local maxima* and *local minima*, which are both *extrema points*. The image of a critical point is a *critical value* of f. A critical point x is *non-degenerate* if $f''(x) \neq 0$. f is a *Morse function* if all its critical points are non-degenerate and have distinct function values. Assume f is a Morse function, the *sublevel set* of f is defined as the pre-image $\mathbb{M}_t := f^{-1}((-\infty, t]) = \{x \in \mathbb{M} \mid f(x) \leq t\}$. To compute the persistent homology, we study the topological changes of $\mathbb{M}_t$ as t increases from $-\infty$ to ∞. This could be considered as the common sweep line idea from computational geometry.

Formally speaking, let m be the number of critical values of f. Let $a_0 < \cdots < a_m$ be a sequence of regular values of f such that each interval (a_i, a_{i+1}) contains exactly one critical value of f. A sublevel set filtration of f is a sequence of sublevel sets connected by inclusions,

$$\mathbb{M}_{a_0} \to \mathbb{M}_{a_1} \to \cdots \to \mathbb{M}_{a_m}.$$

In our setting, 0-dimensional persistent homology studies the topological changes of sublevel sets by applying 0-dimensional homology to this sequence,

$$\mathsf{H}_0(\mathbb{M}_{a_0}) \to \mathsf{H}_0(\mathbb{M}_{a_1}) \to \cdots \to \mathsf{H}_0(\mathbb{M}_{a_n}).$$

The 0-dimensional homology group of a topological space $\mathbb{X}$, denoted as $\mathsf{H}_0(\mathbb{X})$, captures the connected components of $\mathbb{X}$. As t increases, the number of connected components in $\mathbb{M}_t$ only changes when t passes a critical value of f.

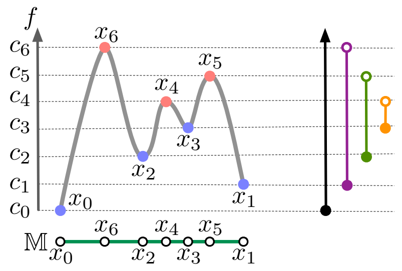

Figure 1: Left: the graph of $f : \mathbb{M} \to \mathbb{R}$, where each point $(x_i, f(x_i))$ is labeled as x_i for simplicity. Right: the 0-dimensional barcode of f based on its sublevel set filtration. Image modified from [29, Fig. 2].

We give an illustrative example in Fig. 1 adapted from [29]. Let x_i denote the critical points and $c_i := f(x_i)$ the critical values of f, ordered as $c_0 < c_1 < \cdots < c_6$ (for readability, we set $c_i = i$). Let $a_0 < a_1 < \cdots < a_7$ be a sequence of regular values of f, where $c_i \in (a_i, a_{i+1})$. As t varies from a_0 to a_7, the 0-dimensional persistent homology encodes the evolution of connected components in $\mathbb{M}_t$. As illustrated in Fig.1 (left), at $t = a_0$, $\mathbb{M}_t$ is empty. At $t = c_0$, a single component appears in $\mathbb{M}_t$; this is referred to as a *birth* event. At $t = c_1, c_2$, and c_3, a 2nd, 3rd, and 4th component appears in $\mathbb{M}_t$, respectively. At $t = c_4$, the 4th component containing x_3 merges with the 3rd component containing x_2. This is referred to as a *death* event: the younger component containing x_3 disappears (dies) while the elder component containing x_2 remains. Similar death events occur at $t = c_5$ and c_6, respectively. Persistent homology pairs the birth and death events as a set of intervals, called *barcode*, shown in Fig. 1 (right), which contains one infinite bar $[c_0, \infty)$ and three finite bars $[c_1, c_6), [c_2, c_5)$, and $[c_3, c_4)$. Since f is assumed to be a Morse function, critical values of f are unique, and each finite bar in the barcode corresponds to a unique *persistence pair* between a local minimum and a local maximum of f, that is, $[x_1, x_6), [x_2, x_5)$, and $[x_3, x_4)$.

In practice, a smooth function $f : \mathbb{R} \to \mathbb{R}$ may be made into a Morse function using simulation of simplicity (SoS) [12]. It assumes arbitrarily small but not vanishing perturbation to f so that critical points become non-degenerate and have distinct function values (i.e., breaking ties consistently).

3.2 Adaptive Sorting and Run Decomposition

Given a list of elements X, adaptive sorting takes advantage of existing *runs* in the list, which are continuous segments already sorted [1]. However, there are some discrepancies in the definitions of runs. Mannila defined the runs as the ascending segments of X [22], whereas Auger et al. [1] defined a run decomposition as an iterative procedure that builds runs based on the local monotonicity; see Fig. 2 for their differences. Following [1], we could either build a run decomposition from left to right (R_+), or from right to left (R_-), and include local extremum we encounter in the current run. We can also consider assigning local extremum arbitrarily to runs, which we avoid. We work with R_+ in this paper. We further assume that runs are organized in alternating monotonic directions (for persistence, see Sec. 3.1). For example, given $X = [1, 2, 3, 2, 5, 4, 3, 1]$, we have $R_+ = [[1, 2, 3], [2, 5], [4, 3, 1]]$ (from left to right) and $R_- = [[1, 2], [3, 2], [5, 4, 3, 1]]$ (from right to left). R_+ is interpreted to be $R_+ = [[1, 2, 3], [], [2, 5], [4, 3, 1]]$, whereas the 1st and the 3rd runs are monotonically increasing and the 2nd empty run and the 4th run are monotonically decreasing;. However, such an interpretation has no impact on the implementation.

We use the number of comparisons to measure a sorting algorithm's complexity. Specifically, let n be the number of elements in an input list X and r the number of runs. The complexity of a sorting algorithm is the number of element comparisons the algorithm performs as a function of n and r. We have the following known result due to [1, 22].

$$\begin{aligned} X &= [12, 10, 7, 5, 7, 10, 14, 25, 36, 3, 5, 11, 14, 15, 21, 22,\\ &\quad 20, 15, 10, 8, 5, 1]\\ R &= [[12, 10, 7, 5], [7, 10, 14, 25, 36],\\ &\quad [3, 5, 11, 14, 15, 21, 22], [20, 15, 10, 8, 5, 1]]\\ R' &= [[12], [10], [7], [5, 7, 10, 14, 25, 36],\\ &\quad [3, 5, 11, 14, 15, 21, 22], [20], [15], [10], [8], [5], [1]] \end{aligned}$$

Figure 2: We repeat the example list X from [22] and provide two different run decompositions R and R' of the list X. R takes monotonic continuous segments following [1]. R' consists of increasing continuous segments in line with [22].

Lemma 1 (Adaptive Sorting Lower Bound) *Any adaptive sorting algorithm on an input of size n with r runs has a worst-case comparison complexity of $\Omega(n + n \log r)$ [1, 22].*

The discrepancy described in Fig. 2 can lead to an asymptotic difference in the statement of the lower bound in Lem. 1 as follows. A strictly decreasing sequence of length n would by [22] be decomposed into $r = n$ singleton increasing sequences, while [1] would find the single decreasing sequence for $r = 1$ runs. For this reason, we use the definition of [1].

An adaptive sorting algorithm may be described by a two-step process. First, the algorithm detects all the runs from an input sequence. Second, it merges the runs in some order determined by a merge policy. There are several adaptive sorting algorithms, such as Natural MergeSort [7], TimSort [26], and PowerSort [24]; see App. A for a detailed review on their merge policies.

3.3 Persistence Pairing Among the Extrema of Runs

The persistence pairing among local extrema of a Morse function can be utilized to study relations among the extremal elements of runs in a list, which is at the core of PersiSort. Let X denote a list of n elements, $X = [x_0, \dots, x_{n-1}]$, where $x_i := X[i]$ (for $0 \le i \le n-1$). For simplicity, assume $x_i \in \mathbb{R}$ and each x_i is unique using the simulation of simplicity (SoS) [12]. X gives rise to a piecewise-linear (PL) function $f : \mathbb{M} \to \mathbb{R}$, where local extrema of f are precisely the extremal elements (extrema) of runs. In other words, the graph of f linearly interpolates among points (i, x_i) (for $0 \le i \le n-1$). Applying sublevel set persistent homology to f gives rise to a persistence pairing among extrema of runs.

Given a list X, a *local maximum* x_i at index i satisfies $x_{i-1} < x_i$ and $x_i > x_{i+1}$. A *local minimum* x_i satisfies $x_{i-1} > x_i$ and $x_i < x_{i+1}$. Depending on their neighbors, x_0 and x_{n-1} are boundary extrema (maximum or

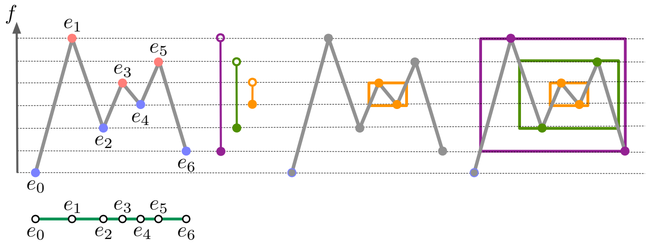

Figure 3: Left: three finite persistence pairs among the extrema: $[e_4, e_3)$, $[e_2, e_5)$, and $[e_6, e_1)$. Middle: a persistence pair $[e_4, e_3)$ with its corresponding persistence box in orange. Right: the nesting of three persistence boxes involving their corresponding persistence pairs.

minimum). Computing persistence pairs of f then gives rise to persistence pairs among the extrema of runs. As shown in Fig. 3 (left), given a list X that contains six extrema (i.e., e_0, e_2, e_4, e_6 are local minima, e_1, e_3, e_5 are local maxima), we obtain three finite persistence pairs: $[e_4, e_3), [e_2, e_5), [e_6, e_1)$; their corresponding bars in the barcode are in orange, green, and purple, respectively.

A persistence pair naturally gives rise to a new concept, a *persistence box*. It is a rectangular box around a persistence pair that is stretched horizontally to include the nearest projections of the local extrema of the pair onto the neighboring runs. See Fig. 3 for an example. Persistence boxes interact with one another, reflecting the relations among runs in a list. A pair of persistence boxes may be *disjoint* or *nested*, or they may *intersect*. In particular, a pair of persistence boxes intersect if their intersection is nonempty and not nested. We describe in Lem. 2 that computing the persistence pairs of X takes linear time.

Lemma 2 *Given a list X of n elements with r runs, the persistence pairs can be computed in $n + \mathcal{O}(r)$ comparisons. If the list of extrema of X is given, then the persistence pairs can be obtained in $\mathcal{O}(r)$ comparisons.*

Proof. Given an element $x_i \in X$, determining whether it is a local extremum relies on its two neighboring elements x_{i-1} and x_{i+1}. Therefore, the local extrema can be computed in a single scan using $n-1$ comparisons.

Assume X contains r runs and E stores the indices of $r+1$ extrema in X. The algorithm to compute persistence pairs proceeds iteratively. During each iteration, it detects pairings among neighboring extrema (referred to as *neighboring persistence pairs*) in E and removes them from the list of extrema E. The algorithm terminates when E is empty or contains one unpaired extremum.

For simplicity, let $e_j := E[j]$ denote an element in the current list of extrema. It has two neighboring extrema e_{j-1} and e_{j+1}. Its *pairing candidate* is one of its neighboring extrema that is closest in terms of its value. Determining the pairing candidate of e_j requires a single comparison between e_{j-1} and e_{j+1}. Two neighboring extrema are paired if they are each other's pairing candidate. The pairing candidate of a boundary extremum is always its neighboring extremum, thus requiring no comparison (e.g., the pairing candidate of e_0 is always e_1 and the pairing candidate of e_r is always e_{r-1}).

Every extremum e_j is removed once from E during some iteration. When e_j is removed, it triggers its neighbor (not paired with e_j) to update its pairing candidate. This takes $\mathcal{O}(1)$ operation. Therefore, processing $r+1$ extrema requires $\mathcal{O}(r)$ comparisons. The comparison complexity is therefore $n + \mathcal{O}(r)$. □

Observation 1 *Intuitively persistence pairings are computed via a sweep line going from $-\infty$ to ∞, but sweep line algorithms require sorting the event points, i.e. extremal values, implying that the comparison complexity of a standard sweep line algorithm would be $\Omega(r \log r)$, excluding the $n-1$ comparisons to find the extremal values. As such, our approach from Lem. 2 is a $\log r$ multiplicative factor improvement.*

We give an illustrative example in Fig. 4. Here, at the beginning of the 1st iteration shown in Fig. 4 (top left), E contains 10 extrema of X, $E = \{e_1, \ldots, e_{10}\}$. The boundary minimum e_0 has a pairing candidate e_1, denoted as $e_0 \to e_1$. The local maximum e_1 has a pairing candidate e_2 since e_2 is closer to e_1 than e_0, therefore $e_1 \to e_2$. Similarly, we have $e_2 \to e_1$, $e_3 \to e_4$, $e_4 \to e_5$, $e_5 \to e_4$, $e_6 \to e_5$, $e_7 \to e_8$, $e_8 \to e_9$, $e_9 \to e_8$, and $e_{10} \to e_9$. Since we have $e_1 \leftrightarrow e_2$, $e_4 \leftrightarrow e_5$, $e_8 \leftrightarrow e_9$ we obtain three neighboring persistence pairs $[e_2, e_1)$, $[e_4, e_5)$, and $[e_8, e_9)$, shown in orange, red, and purple, respectively, see Fig. 4 (top left). Removing the extrema involved in these pairs gives rise to an updated list of extrema at the beginning of the 2nd iteration, $E = \{e_0, e_3, e_6, e_7, e_{10}\}$. The pairing candidates of their neighboring extrema are also updated. See Fig. 4 (top middle).

For instance, as shown in Fig. 4 (top middle), when extrema from the pair $[e_4, e_5)$ are removed from E, we update the pairing candidates of their neighbors e_3 and e_6 to obtain $e_3 \to e_6$ and $e_6 \to e_3$. During the 2nd iteration, we obtain a new neighboring persistence pair $[e_6, e_3)$ since $e_3 \leftrightarrow e_6$. During the 3rd and final iteration, we obtain a final neighboring pair $[e_{10}, e_7)$, shown in teal in Fig. 4 (top right) . Each persistence pair is enclosed by a colored persistence box, shown in Fig. 4 (bottom).

However, corner cases involving the boundary extrema require some care. Based on the algorithm described in Lem. 2, a boundary extremum may be involved in a pair that is not a proper persistence pair. As illustrated in Fig. 5, e_9 is a boundary maximum, but the pair $[e_8, e_9)$ is not a proper persistence pair: e_8 gives rise to a new component, which is not killed at

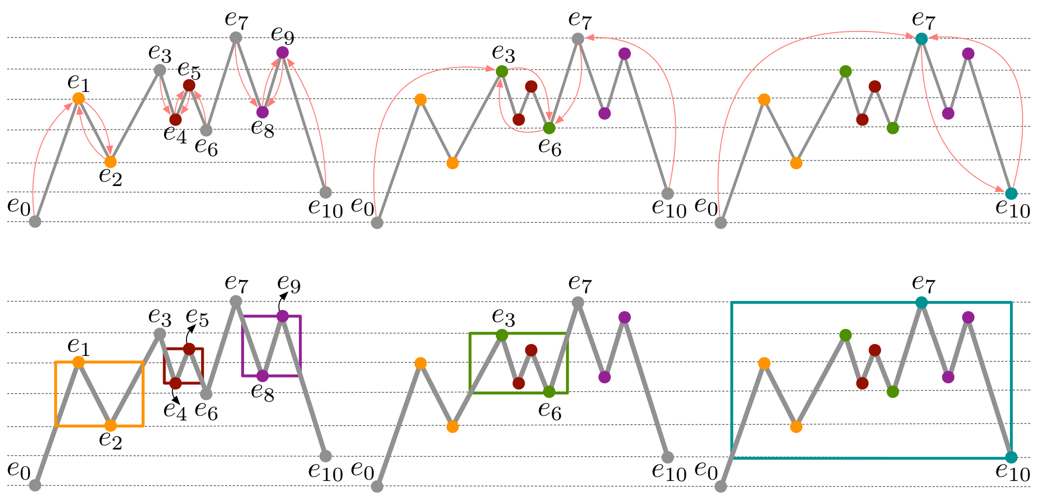

Figure 4: An illustration of computing persistence pairs. Top: a pink arrow from each e_i points to its pairing candidate during each iteration. Bottom: Persistence boxes surrounding persistence pairs were detected during each iteration. From left to right: the 1st iteration returns pairs shown in orange, red, and purple, respectively; the 2nd iteration returns a pair in green; and the 3rd and final iteration returns a pair in teal.

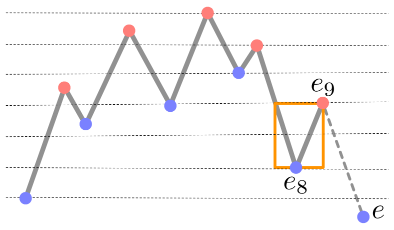

Figure 5: Adding a dummy run (dotted line) for a boundary extremum.

e_9. Such a pair can be made into a proper persistence pair conceptually by adding a dummy run adjacent to the boundary extremum that extends beyond the global minimum. A boundary minimum can be handled similarly by adding an adjacent dummy run that extends beyond the global maximum.

In the case of duplicates, SoS is not actually implemented in PersiSort due to its non-negligible overhead, instead we employ simple rules to handle the pairings in a way that remains consistent with persistence. Specifically, in the case of equally valued pairing candidates, the maximum would first consider the candidate with the smaller index, and the minimum would first consider the candidate with the larger index.

3.4 AdaptMerge and FingerMerge

The key idea behind PersiSort is performing a pair of three-way merges—referred to as FingerMerge—around persistence pairs. Carlsson, Levcopoulos, and Petersson [7] introduced a merging procedure that uses exponential and binary search [3] to achieve the optimal number of comparisons when merging two sorted lists. This is referred to as AdaptMerge in [7], it is also known as galloping, which is employed by TimSort [1, 26]. FingerMerge employs AdaptMerge twice to perform a three-way merging of three sorted lists. We review the idea behind AdaptMerge for completeness. We slightly modify in Fig. 6 an example from [7, Fig. 1]. The input to a merging algorithm consists of two sorted lists, A and B. The output is a sorted list C. Each entry in C is a consecutive subsequence of A or B, e.g., $C[0] = A[0, 4]$ and $C[1] = B[0, 0]$. We obtain the merged sequence by reporting the elements in these subsequences in the order in which they appear in B.

$$\begin{aligned}
A &= [1, 2, 3, 4, 5, 7, 8, 11] \\
B &= [6, 9, 10, 12, 13, 14] \\
C &= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] \\
&= [A[0, 4], B[0, 0], A[5, 6], B[1, 2], A[7, 7], B[3, 5]]
\end{aligned}$$

Figure 6: An example of the input and output of a merging algorithm adapted from [7, Fig. 1]. We write $A[i_1, i_2]$ to represent the elements of A at indices i_1 through i_2.

We now describe AdaptMerge applied to two sorted list A and B; w.l.o.g., we assume that $a_0 := A[0] < b_0 := B[0]$. Consider the example in Fig. 6. To compute C, we find the positions in which A and B have to be split, where portions of the other sequence should be inserted. In other words, we compute the elements in A and B that would receive new successors in the resulting sequence. For example, number 5 from A receives a new successor 6 from B in the resulting sequence C; number 10 from B receives a new successor 11 from A in C, and so on. The intuition behind AdaptMerge is that if there are large consecutive portions in A and B in which all elements would keep their original successor after merging [7], then these elements do not need to be examined entirely during merging. AdapteMerge "uses exponential and binary search to pass such portions as fast as possible in our search for the next element that would receive a new successor" [7].

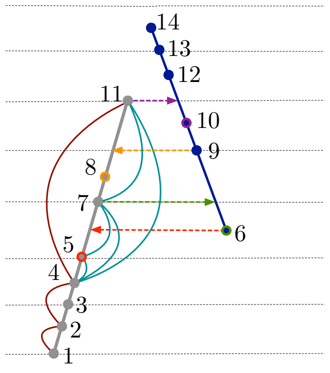

Figure 7: An illustration of AdaptMerge algorithm. The deep red curve illustrates the exponential search (by indices), whereas the teal curve illustrates the binary search. Elements from both sorted lists are laid out in increasing values, resembling a run.

Following [7], let $\{a_{i_0}, a_{i_1}, \ldots, a_{i_p}\}$ be a set of elements in A that will receive new successors. For example, this set equals $\{5, 8, 11\}$ in A. Similarly, $\{b_{j_0}, b_{j_1}, \ldots, b_{j_q}\}$ is a set of elements in B that will receive new successors; this would be $\{6, 10\}$ in B. If A and B are two sorted sequences of length n and m, respectively, we define $\text{Rank}(b_j, A) = max\{\ell \mid 0 \leq \ell \leq n, b_j > a_\ell\}$, for $0 \leq j \leq m-1$. This means the maximum element in A is smaller than (thus closest to) b_j. $\text{Rank}(b_j, A)$ tells us where the split of A and the actual merge from B has to start.

As illustrated in Fig. 7, for the example from Fig. 6, AdaptMerge starts by computing $i_0 = \text{Rank}(b_0, A) = 4$ by an exponential and binary search forward in A based on element indices. Deep red curves illustrate the exponential search, whereas teal curves illustrate the binary search. The red dotted arrow illustrates the computing process i_0, and $A[i_0]$ is highlighted as a red point. Then $C[0] = A[0, i_0] = A[0, 4]$. Second, we compute $j_0 = \text{Rank}(a_{i_0+1}, B) = 0$ by an exponential and binary search forward in B. And we set $C[1] = B[0, j_0] = B[0, 0]$ (see the green dotted arrow and the green point). Third, we compute $i_1 = \text{Rank}(b_{j_0+1}, A) = 6$ by an exponential and binary search forward in X starting from $A[i_0 + 1]$, and set $C[2] = A[i_0 + 1, i_1] = A[5, 6]$ (see the orange dotted arrow and the orange point). We continue to perform exponential and binary searches, alternating between A and B. We start the search from where the last element is found to receive a new successor. When one of the sequences is finished, the next empty entry in C is set to the remaining portion of the nonempty sequence (e.g., $B[3, 5]$ is copied over). For completeness, the pseudocode of AdaptMerge is included in App. B. The following complexity of AdaptMerge is obtained by studying the worst case lower bound on the number of comparisons performed by a merging algorithm in Lem. 3 and Thm. 4 of [7].

Lemma 3 ([7]) *Applying AdaptMerge to two sorted lists of lengths $n_1 \leq n_2$ has a worst-case comparison complexity $\mathcal{O}\left(n_1 \log\left(\frac{n_1+n_2}{n_1}\right)\right)$.*

Observation 2 *In particular, AdaptMerge of two sorted lists where all elements of one list have smaller values than all elements of the other performs $\mathcal{O}(\log n_1)$ comparisons (assuming $n_1 \leq n_2$).*

We define three-way FingerMerge(A,B,C) to be AdaptMerge(AdaptMerge(A,B),C); see App. B for pseudocode for both merge algorithms.

Recall that we study the complexity of a sorting algorithm using the *comparison cost*, also used by the Natural MergeSort and its subroutine AdaptMerge [7]. PowerSort [24] and multiway PowerSort [14], on the other hand, are optimized with regards to the *merge cost*. To merge two runs of lengths n_1 and n_2, AdaptMerge has a merge cost of $\mathcal{O}(n_1 + n_2)$ and a comparison cost of $\mathcal{O}\left(n_1 \log\left(\frac{n_1+n_2}{n_1}\right)\right)$ for $n_1 \leq n_2$. These two costs are equivalent in the worst case. We explore the Merge Tree that encodes the merging order in Sec. 4.

4 New Result: Optimal Bound for Adaptive Sorting

Based on our discussion of AdaptMerge in Sec. 3.4, we prove, for the first time, that any adaptive sorting algorithm that uses AdaptMerge (i.e., two-way merges of runs) as a subroutine is worst-case optimal. In particular, given a list of n elements containing r (precomputed) runs, such an algorithm has a worst-case comparison complexity of $\mathcal{O}(n \log r)$.

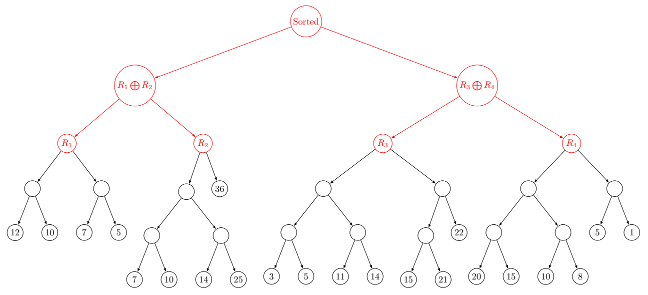

Figure 8: An exemplar Merge Tree of the list in Fig. 2. AdaptMerge(R_1, R_2) is represented by $R_1 \bigoplus R_2$.

To analyze the comparison complexity of a merge-based sorting algorithm, we can view the intermediate steps as a tree, referred to as a *Merge Tree* of a sorting algorithm[2]. In a Merge Tree, leaves represent (sublists of) single elements, internal nodes represent intermediate sorted sublists, and the root represents the entire sorted list. Given an input list X of n elements, the classic MergeSort first divides X into n (sorted) sublists of one element and then repeatedly merges sublists to produce sorted ones until only one sublist remains. Therefore, the classic MergeSort produces an almost perfectly balanced Merge Tree. On the other hand, any iterative merge-based sorting algorithm that inserts elements into a sorted list one at a time can be represented by a maximally unbalanced Merge Tree.

We extend the notion of a Merge Tree to adaptive sorting by introducing the *Adaptive Merge Tree*, which is a Merge Tree whose leaves represent nonempty sorted lists (runs). In Fig. 8, we represent an Adaptive Merge Tree (in red) as a subtree of a Merge Tree (in red and black) for the example in Fig. 2. Not every Merge Tree contains an Adaptive Merge Tree as a subtree. Here, R_1, R_2, R_3 and R_4 (in red) are the four runs from the run decomposition R in Fig. 2. Each subtree rooted at R_i (in black) is a Merge Tree that shows how (an instance of) a MergeSort would have constructed R_i from the input. Given an input list of n element with r runs, the run decomposition requires $n-1$ comparisons. Using an Adaptive Merge Tree, we produce an upper bound in Thm. 4. Assume that we have an input list of n elements containing r precomputed runs, where an ith run contains ℓ_i elements.

Theorem 4 *Any adaptive sorting algorithm that uses AdaptMerge as a subroutine performs $\mathcal{O}(nH(\ell_1,\dots,\ell_r)) = \mathcal{O}(n\log r)$ comparisons to merge precomputed runs based on its predetermined merge policy. In the case when runs are not precomputed, the comparison complexity is $n+\mathcal{O}(nH(\ell_1,\dots,\ell_r)) = n+\mathcal{O}(n\log r)$.*

Proof. We assume X to be a list of n elements containing r runs, and each run contains $\ell_i \geq 2$ elements. We assume the runs have been precomputed and the merge policy has been predetermined, therefore we start with an established Adaptive Merge Tree T. Using AdaptMerge as a subroutine, we report on the number of comparisons needed to merge the sublists from the leaves to the root of T.

Let $f : T \to \mathbb{Z}$ be a function that assigns to each node $v \in T$ the number of comparisons performed to reach its corresponding sublist from its children. For any leaf v, set $f(v) = 0$. The comparison complexity of the algorithm is the number of comparisons needed to arrive at the root o of T, that is, $f(o)$. We prove that $f(o) \leq cn(H(\ell_1,\dots,\ell_r))$ by induction on the size of the tree. Here, the constant $c > 0$ comes from the $\mathcal{O}$ notation of Lem. 3.

[2]This is an entirely different concept from the merge tree of a scalar field commonly used in TDA; see [29] for a survey.

If we have a single run ℓ_1, $H(\ell_1) = 0$, which is trivial. We thus start with an base case (BS) with two runs of size ℓ_1 and ℓ_2, where $\ell_1 + \ell_2 = n$. This corresponds to an Adaptive Merge Tree T that contains a root o with two leaves that correspond to runs of lengths ℓ_1 and ℓ_2 respectively (w.l.o.g., assuming $\ell_1 \leq \ell_2$). A single AdaptMerge is applied to the two runs and according to Lem. 3, the comparison cost is

$$\begin{aligned}
f(o) &\leq c\ell_1 \log\left(\frac{\ell_1+\ell_2}{\ell_1}\right) \\
&< c\ell_1 \log\left(\frac{\ell_1+\ell_2}{\ell_1}\right) + c\ell_2 \log\left(\frac{\ell_1+\ell_2}{\ell_2}\right) \\
&= -c\ell_1 \log\left(\frac{\ell_1}{\ell_1+\ell_2}\right) - c\ell_2 \log\left(\frac{\ell_2}{\ell_1+\ell_2}\right) \\
&= cn\left(H(\ell_1,\ell_2)\right)
\end{aligned}$$

For the induction hypothesis (IH), we assume the bound holds for trees with $r-1$ runs or less.

For the induction step, we need to show the bound holds for trees with r runs. Let T be a given Adaptive Merge Tree and o be its root. Let o_L be the root of the left subtree over n_L elements whose leaves (runs) are indexed by an index set I_L. Similarly, let o_R be the root of the right subtree over n_R list elements whose leaves (runs) are indexed by an index set I_R. By construction, $n_L + n_R = n$, $|I_L| + |I_R| = r$, $\sum_{i\in I_L} \ell_i = n_L$ and $\sum_{j\in I_R} \ell_j = n_R$. Assume *w.l.o.g.* that $n_L \leq n_R$.

The comparison cost needed to reach the root o is obtained from the comparison cost required to reach its children o_L and o_R plus the cost of merging their corresponding sublists, according to Lem. 3. Since $|I_L| \leq r-1$ and $|I_R| \leq r-1$, the IH holds for both the left and right subtree.

To complete the induction we first look at the entropy function $H(h_1,\dots,h_k)$, denoted as $H(\{h_i\}_{i\in I})$ for simplicity (for all i in the index set I). For $h_i > 0$ and $m = \sum_{i=1}^k h_i$, we have,

$$\begin{aligned}
mH(h_1,\dots,h_k) &= \sum_{i=1}^k h_i \log\left(\frac{m}{h_i}\right) \\
&= \log(m)\sum_{i=1}^k h_i - \sum_{i=1}^k h_i \log(h_i) \\
&= m\log(m) - \sum_{i=1}^k h_i \log(h_i) \qquad (1)
\end{aligned}$$

Returning to our induction and apply Eqn. 1 to the left and right subtrees,

$$\begin{aligned}
f(o) &\leq f(o_L) + f(o_R) + cn_L \log\left(\frac{n_L + n_R}{n_L}\right) \\
&\leq cn_L H(\{\ell_i\}_{i \in I_L}) + cn_R H(\{\ell_j\}_{j \in I_R}) \\
&\quad + cn_L \log\left(\frac{n_L + n_R}{n_L}\right) \\
&= cn_L \log(n_L) - c \sum_{i \in I_L} \ell_i \log(\ell_i) \\
&\quad + cn_R \log(n_R) - c \sum_{j \in I_R} \ell_j \log(\ell_j) \\
&\quad + cn_L \log\left(\frac{n}{n_L}\right) \\
&= cn_R \log(n_R) + cn_L \log(n) - c \sum_{i=1}^{r} \ell_i \log(\ell_i) \\
&< cn_R \log(n) + cn_L \log(n) - c \sum_{i=1}^{r} \ell_i \log(\ell_i) \\
&= cn \log(n) - c \sum_{i=1}^{r} \ell_i \log(\ell_i) \\
&= cnH(\ell_1, \dots, \ell_r),
\end{aligned}$$

concluding the induction.

Finally, the right-hand side of the upper bound $\mathcal{O}(n \log(r))$ follows from the fact that $H(\ell_1, \dots, \ell_r)$ is maximized when $\ell_i = n/r$, in which case $nH(n/r, \dots, n/r) = \sum_{i=1}^{r} n/r \log r = n \log r$. $\square$

5 New Result: Persistence Sort

We now introduce the novel PersiSort algorithm. The key idea is performing a pair of three-way merges—referred to as FingerMerge—around persistence pairs, that is, merging the three consecutive runs that intersect a persistence box (or two successive runs involving boundary extrema). On a high level, the algorithm identifies persistence pairs with multiple iterations (described in the proof of Lem. 2). It applies FingerMerge to runs that intersect each persistence pair.

Given an input list X with n elements in r runs, we first identify the set of extrema E from X. We then compute the initial set of (neighboring) persistence pairs. We repeat the following procedure until the list is sorted, i.e., when there are at most two extrema in E.

1. For each persistence pair:
   - Perform FingerMerge on the two or three runs that intersect the pair.
     - ∗ If the pair contains boundary extrema, perform a two-way merge;
     - ∗ Otherwise, perform a three-way merge;
   - Remove the pair of extrema from the set of extrema E.
2. Recompute the persistence pairs by updating the pairing candidates (i.e., neighboring extrema that are closest in terms of values).

Using Fig. 9 as an illustrative example (cf., Fig. 4), we first identify the set of extrema $E = \{e_0, \dots, e_{10}\}$ and denote the ten runs as $R_1, \dots, R_{10}$. We compute the initial set of neighboring persistence pairs, whose persistence boxes are visualized as colored boxes in (a). During the 1st iteration, we perform a three-way merge (FingerMerge) of runs intersecting the box for each box. For example, we would merge R_1, R_2, R_3 that intersect the orange box defined by the pair $[e_2, e_1)$. We would then merge R_4, R_5, R_6 that intersect the red box defined by the pair $[e_4, e_5)$, followed by merging R_8, R_9, R_{10} that intersect the purple box defined by the pair $[e_8, e_9)$. We would remove the corresponding extrema from E, resulting in $E = \{e_0, e_3, e_6, e_7, e_{10}\}$. We then recompute the persistence pairs among E, producing a single pair $[e_6, e_3)$ whose green persistence box is visualized in (b). During the 2nd iteration, we merge the three runs intersecting the green box (b), remove the corresponding extrema, and update $E = \{e_0, e_7, e_{10}\}$. During the final iteration, we merge the remaining two runs that intersect the teal box (c), producing a sorted list visualized in (d).

To analyze the comparison complexity of PersiSort, we introduce the notion of *dynamic box depth* of an element $x \in X$, which is the number of persistence boxes it belongs to across iterations, denoted as $d(x)$. As shown in Fig. 9 (cf., Fig. 4), an element $x_i \in X$ belongs to the orange box during the 1st iteration, and the teal box in the 3rd iteration, therefore it has a box depth $d(x_i) = 2$. $x_j \in X$ belongs to the orange box in the 1st iteration, the green box in the 2nd iteration, and the teal box in the 3rd iteration, therefore, it has a box depths $d(x_j) = 3$. $x_k \in X$ belongs to the orange box in the 1st iteration, then it dynamically "moves" into the green box during the 2nd iteration (to somewhere close to x_j, not visualized here), and stays within the teal box during the 3rd iteration, therefore $d(x_k) = 3$. Therefore, the dynamic box depth $d(x)$ captures the number of FingerMerge operations an element x will participate in. We then need the following Lem. 5 and Lem. 6.

Lemma 5 *FingerMerge implicitly computes the persistence boxes.*

Proof. We use Fig. 10 to illustrate the relation between FingerMerge and persistence boxes. First, w.l.o.g., assume $[e_{p+1}, e_p)$ is a persistence pair that does not involve a boundary extremum. The pair intersects three runs R_p, R_{p+1}, and R_{p+2} (with number of elements $\ell_p, \ell_{p+1}, \ell_{p+2}$, respectively). Computing the persistence box of such a pair is equivalent to finding the predecessor of e_{p+1} in R_p and the successor of e_p in R_{p+2}.

Using FingerMerge, we apply AdaptMerge to the runs

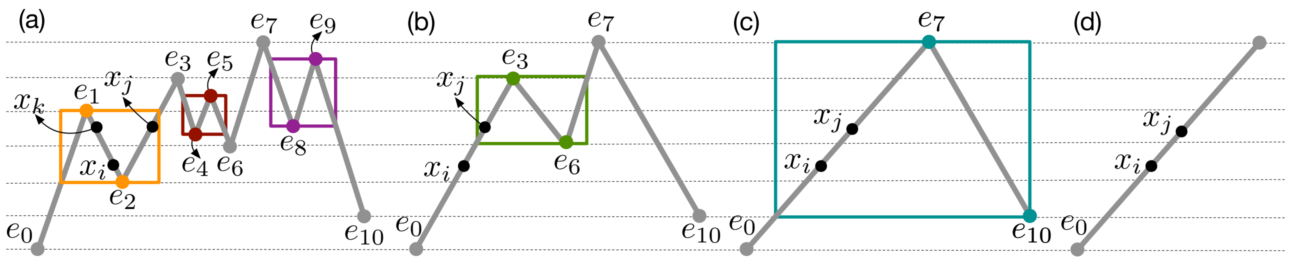

Figure 9: A step-by-step illustration of the PersiSort algorithm.

R_p and R_{p+1}. This process involves finding the maximum element x in R_p that is smaller than e_{p+1} using exponential and binary search (c.f., Fig. 7). x is the location where R_p needs to be split and the actual merge from R_{p+1} has to start. Identifying element x is illustrated by a red dotted arrow from e_{p+1}, and x is shown as a red point. A key observation is that such a process implicitly identifies the lower left corner of the persistence box.

Assume x is located at index i_x in R_p, then the exponential and binary search discovers x in $\mathcal{O}(\log i_x) = \mathcal{O}(\log \ell_p)$ comparisons. After merging R_p with R_{p+1} into R', FingerMerge merges R' with R_{p+2}. In fact, it merges R' starting from index $i_x + 1$ in R' (i.e., the index of e_{p+1} in R'), since all elements in R_{p+2} are larger than e_{p+1}. When R' is exhausted of elements, we have found the successor of e_p in R_{p+2}. This is equivalent to finding the upper right corner of the persistence box, as desired. □

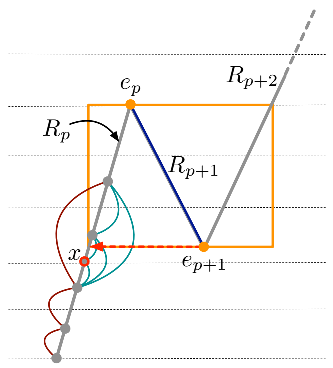

Figure 10: An illustration of the FingerMerge algorithm. The deep red curve illustrates the exponential search, whereas the teal curve illustrates the binary search.

With the above Lemma, we are ready to present an analysis of PersiSort that uses our new computational model. As illustrated in Fig. 9, dynamic box depth is insufficient for analyzing PersiSort since the algorithm requires extra work to detect the boundaries of persistence boxes using FingerMerge. For example, detecting the boundary of the orange box along the run R_p requires $\log(\ell_p)$, which is an overestimation based on the exponential and binary search along R_p. We know from the Adaptive Merge Tree structure that there are r runs represented by leaves, denoted as $R_1, \ldots R_r$; and $r-1$ intermediate sorted sublists represented by internal nodes, denoted as $R_{r+1}, \ldots, R_{2r-1}$. Let $\ell_i = |R_i|$ for $1 \leq i \leq 2r-1$.

Lemma 6 *Using PersiSort on a list of n elements with r runs, the number of comparisons performed is*

$$n + \mathcal{O}\left(r + \sum_{x \in X} d(x) + \sum_{i=1}^{2r-1} \log \ell_i\right).$$

Proof. First, recall in Lem. 2 that all persistence pairs can be computed in $n + \mathcal{O}(r)$ comparisons, where $n-1$ comparisons are needed to first locate the runs.

Second, the dynamic box depth $d(x)$ of an element $x \in X$ captures the number of FingerMerge operations an element x may participate in. However, x may not be compared during the FingerMerge process. The term $\sum_{x \in X} d(x)$ is thus an overestimation of the contribution of elements to the comparison complexity.

Finally, PersiSort by design performs a FingerMerge on runs intersecting a persistence box during an iteration. We know from Lem. 5 that an element outside a persistence box will contribute logarithmically (in the number of elements outside the box) to the comparison complexity. We can upper bound the number of elements outside of the persistence boxes by the total number of elements across the original and intermediate runs, that is, $\sum_{i=1}^{2r-1} \log \ell_i$. This concludes the comparison complexity analysis of PersiSort. □

Together with Thm. 4, Lem. 6 now provides an alternative comparison complexity analysis of PersiSort.

A desirable property of a sorting algorithms is to be *stable*. Stability is defined as follows: if elements $x = y$, and x precedes y in the initial ordering, then this ordering is preserved after sorting. AdaptMerge is stable, making FingerMerge and subsequently PersiSort stable as well. Although PersiSort does not merge runs sequentially like PowerSort and TimSort, it always merge adjacent runs, which is sufficient for a simple inductive proof.

6 Adaptive Sorting Implementations

We discuss implementation details on several popular sorting algorithms: the sorting algorithm implemented in Python (referred to as Python Sort), TimSort, and PowerSort. We implement PersiSort and TimSort in-house. Following [1, 28], we use the number of comparisons performed to quantify the complexity of these sorting algorithms. We count the number of element comparisons via a custom class for our in-house implementations.

TimSort. We use an in-house Python implementation of TimSort based on the description of [1]. The original TimSort uses galloping, a version of AdaptMerge, discussed in Sec. 3.4.

TimSort is a sequential adaptive sorting algorithm that maintains a stack of runs and applies AdaptMerge to selected pairs of runs. Specifically, it merges the top run and the 2nd run from the top or the 2nd and 3rd runs from the top under a merge policy. It ensures that the sizes of the runs on the stack form an exponentially increasing sequence.

Python Sort. We compare against the sorting algorithm used in Python version 3.11.2[3] at the time of writing. This version of Python implements the PowerSort of Munro and Wild [24].

PowerSort [24] is essentially based on TimSort but with a different merge policy, see App. A. In Python version 3.11.2, PowerSort is the standard library sort, which makes it easy for us to get a comparison. On the other hand, the implementation in Python is highly optimized for time, unlike our in-house PersiSort and TimSort implementations. To differentiate the Python sorting algorithm from the PowerSort described below, this algorithm is referred to as **Python v3.11.2** in our experiments.

PowerSort. Sebastian Wild provided an educational implementation of PowerSort [27] that we used as PowerSort in our experiments. This version is not as highly optimized as the Python standard library version, but we can control which merge subroutine is used. In the original code provided by Wild, a classic $\mathcal{O}(n_1 + n_2)$ merge routine is used (for merging a pair of lists of lengths n_1 and n_2 respectively). For our experiments, we replace it with AdaptMerge to more fairly compare the merge policies of PersiSort and PowerSort.

7 Data Distributions

To empirically compare the adaptive sorting algorithms described in Sec. 6, we introduce six data distributions (also referred to as run configurations) that illustrate different behaviors of the algorithms under scrutiny. The six data distributions include Staircase, Isolated Points, Super Nesting, Uniformly Random, Ultra Nesting, and TimSort Nemesis, presented left to right in Fig. 11. A list sampled from a data distribution is visualized as a PL function: the x-axis represents the indices of list elements, and the y-axis represents their values; elements are visualized as blue points connected by edges following the input order, where runs are easily visible as monotonic segments of the PL curve. We also include one additional data distribution, Overlapping Staircase, that is a variant of Staircase. The lists sampled from these distributions differ by the amount of overlap between runs in their range of values and, subsequently, the dynamic box depths of elements in the lists. We describe the intuition behind these data distributions and the performance of PersiSort on them; see Tab. 12 for an overview.

We sample 100 lists from each data distribution and report the median number of comparisons. For lists sampled from the Staircase, Isolated Points, and Super Nesting distributions, we first vary the number of elements and then the number of runs; see Fig. 13. For lists sampled from Uniformly Random, Ultra Nesting, and TimSort Nemesis, we only vary the number of elements as we cannot control the number of runs; see Fig. 14.

When we vary the number of elements, we hard code 50 runs and let the number of elements range from 150 to 2950 in increments of 100 for a total of 29 data points. Similarly, when we vary the number of runs, we hard code 3000 elements and let the number of runs range from 10 to 750 in steps of 25 for 30 total data points.

Observation 3 (Disjoint Values) *AdaptMerge on two sorted lists A, B of lengths $n_1 \leq n_2$ has a worst-case comparison complexity of $n_1 \log\left(\frac{n_1+n_2}{n_1}\right)$ by Lem. 3. However, if the values in the runs are disjoint, w.l.o.g., assuming $A[n_1 - 1] \leq B[0]$, then AdaptMerge would use worst case $\mathcal{O}(\log n_1)$ comparisons and simply prepends A to B.*

We use Obs. 3 to create data distributions where the benefits of AdaptMerge over regular merge subroutines are maximized.

Staircase. The Staircase distribution is designed such that an element x in the list has a constant dynamic box depth $d(x)$. This is achieved by creating monotonically increasing runs of length three, and monotonically decreasing runs of length $n/(r/2) - 3$ in an alternating fashion, while ensuring that all elements of run i have smaller values than those of run $i + 1$. This ensures that $d(x) = 1$ for all elements. When running PersiSort on the staircase distribution, an accumulator run is created, and the initial runs are merged into it one at a time. This behavior is equivalent to a maximally unbalanced Adaptive Merge Tree. By Obs. 3 and Lem. 6,

[3]`https://www.wild-inter.net/posts/powersort-in-python-3.11`, accessed on December 2, 2023

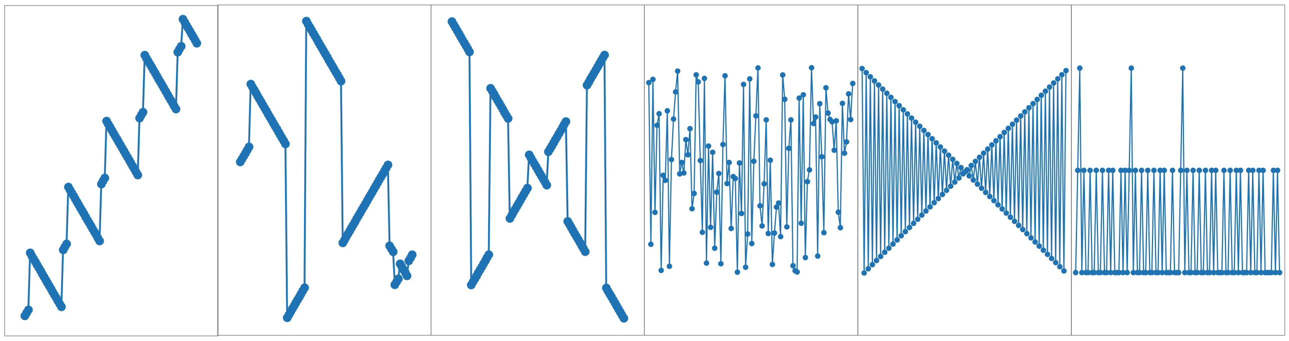

Figure 11: Six types of data distributions. From left to right: Staircase, Isolated Points, Super Nesting, Uniformly Random, Ultra Nesting, and TimSort Nemesis.

Distribution	Disjoint Values	Box depth	PersiSort
Staircase	Yes	1	$\mathcal{O}(n + r\log(rn))$
Isolated Points	Initially	$\mathcal{O}(r)$	$\mathcal{O}(n + n\log r)$
Super Nesting	No	$\mathcal{O}(r)$	$\mathcal{O}(n + r\log n)$
Ultra Nesting ‡	No	$\mathcal{O}(n)$	$\mathcal{O}(n\log n)$
Overlapping Staircase ‡	Variable	$\mathcal{O}(r)$	$\mathcal{O}(n + n\log r)$
Uniformly Random ‡	No	$\mathcal{O}(n)$	$\mathcal{O}(n\log n)$
TimSort Nemesis ‡	No	$\mathcal{O}(n)$	$\mathcal{O}(n\log n)$

Figure 12: Overview of our data distributions and the comparison complexity using PersiSort. n is the number of elements in a list with r runs. "Disjoint Values" means that runs contain values that are disjoint. We did not perform experiments by varying r with distributions marked with a ‡.

the comparison complexity of the merges can roughly be described as

$$\mathcal{O}\left(\sum_{i=1}^{2r-1}\log\left(i\cdot n/r\right)\right) = \mathcal{O}\left(r\log n + r\log r\right)$$

for a total comparison complexity of $\mathcal{O}(n + r\log(rn))$.

Isolated Points. We introduce the Isolated Points distribution to investigate the impact of the disjoint runs on the performance. By partitioning a list of unique integers into r continuous subsets of random sizes and then uniformly shuffling them, we obtain elements in a list of stochastic dynamic box depth. The purpose of this is to show that, in general, the performance of PersiSort is consistent with initially disjoint runs. It follows from basic probability theory that uniformly random data has runs of expected constant length, which means that $r = \Theta(n)$ and PersiSort has a comparison complexity of $\mathcal{O}(n + nH) = \mathcal{O}(n + n\log r)$.

Super Nesting and Ultra Nesting. Super Nesting is designed for most elements to have high dynamic box depth while maintaining non-intersecting runs. A way to envision the distribution is to have an "X" shape and remove elements such that there are r pieces of equal lengths, and the projection onto either axis is injective. The lowest level persistence pair throughout the execution of PersiSort is between the endpoints of the innermost run/piece of the "X" shape. At each level, PersiSort performs $\mathcal{O}(\log n)$ comparisons to append the neighboring pieces for a total comparison complexity of $\mathcal{O}(n + r\log n)$. Ultra Nesting is the most extreme version of Super Nesting, where each run has length two.

Overlapping Staircase. We are interested in exploring how increasing the dynamic box depth $d(x)$ affects the performance of PersiSort. To do so, we generalize the Staircase distribution to the Overlapping Staircase distribution, where adjacent runs have increasing overlap in their ranges of values with a controlled parameter. The overlap parameter s controls the length of the short runs, and by "pulling down" the runs, we have the same s values in three adjacent runs. This means that the benefit of FingerMerge is neutralized. As s tends to n/r, the data become a maximally entangled "zigzag", for instance, $s = n/r = 3$ produces data that look like $1, 2, 3, 2, 1, 0, 1, 2, 3, \ldots$. Here, PersiSort will produce an accumulator and merge adjacent length three lists into it. In this specific case that contains very few unique values, it would be much more effective to count the occurrences of each value using Counting Sort or Bucket Sort.

TimSort Nemesis. It is conjectured by [5, 24] that the worst-case input for TimSort is the following recursive sequence:

$$\mathcal{R}(n) = \begin{cases} \langle n\rangle & \text{if } n \le 3; \\ \mathcal{R}(n/2) :: \mathcal{R}(n/2-1) :: \langle 1\rangle & \text{if } 2|n; \\ \mathcal{R}\left(\frac{n-1}{2}\right) :: \mathcal{R}\left(\frac{n-1}{2}-1\right) :: \langle 2\rangle & \text{otherwise,} \end{cases}$$

where :: denotes list concatenation, $2|n$ means $n \equiv 0 \mod 2$ and $\langle k\rangle$ is the list $[0, 1, 2, \ldots, k-1]$.

PersiSort faces the same difficulties with this distribution as with maximally overlapping staircases, the dynamic box depth of an element in the list is $\mathcal{O}(n)$, which by Lem. 6 gives an $\mathcal{O}(n\log n)$ upper bound.

Uniformly Random. Following the footsteps of [5, 24], we sample real numbers from the interval $[0, 1]$ uniformly randomly. This distribution has very short runs,

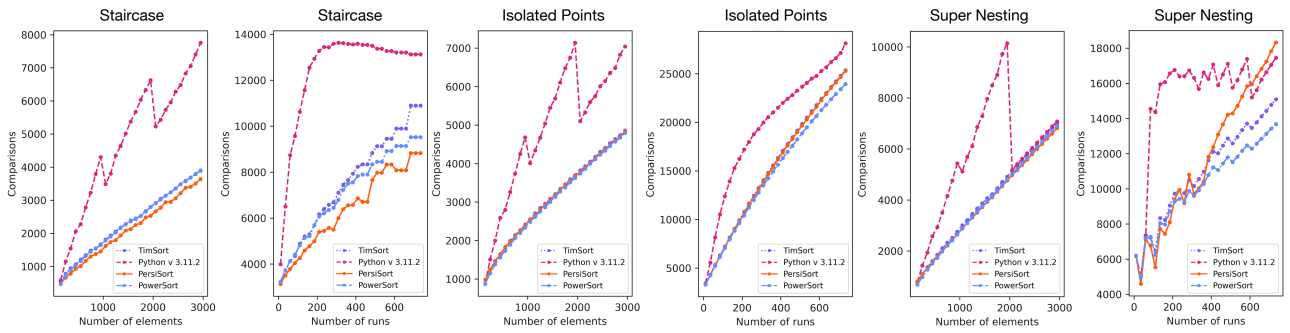

Figure 13: Number of comparisons with Staircase, Isolated Points, and Super Nesting distributions.

in expectation, where the benefit of using an adaptive sorting algorithm is negligible.

8 Experimental Results

We compare the number of comparisons of PersiSort empirically against several adaptive sorting algorithms, including TimSort [1], Python Sort (the PowerSort implementation used in Python version 3.11.2, which does not use AdaptMerge), and PowerSort [24, 27]. Notably, our PowerSort implementation uses AdaptMerge as a subroutine, while Python Sort does not. We use the data distributions described in Sec. 7.

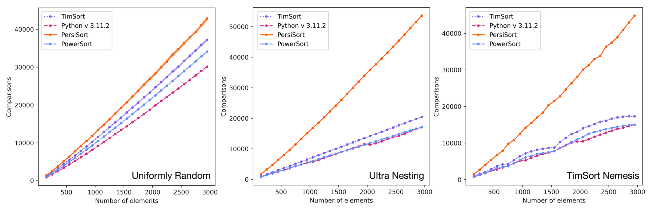

Figure 14: Number of comparisons with Uniformly Random, Ultra Nesting, and TimSort Nemesis distributions.

Highlighted results. As shown in Fig. 13, PersiSort outperforms state-of-the-art adaptive sorting algorithms—PowerSort, TimSort, and Python Sort—on the Staircase data distributions, where runs have no overlap in their ranges of values. This seems reasonable since the merge policy of PersiSort considers the extrema values of the runs. In contrast, TimSort and PowerSort consider the number of elements in the runs when deciding which runs to merge. Meanwhile, PersiSort performs comparably with PowerSort and TimSort on Isolated Points and (partially) on Super Nesting distributions (see Fig. 13). However, it is also clear from Fig. 14 that PersiSort will not replace PowerSort as the standard Python library sorting algorithm. Nevertheless, PersiSort provides a new perspective on adapting sorting based on TDA.

Additional results. We experiment further by increasing the overlap between runs, creating a data distribu-

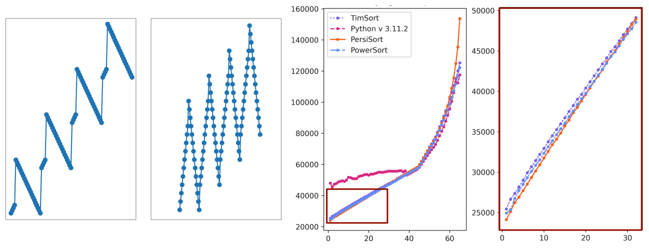

Figure 15: Number of comparisons with the Overlapping Staircase distribution. From left to right: a list sampled from the distribution with a small amount of overlap (before the elbow point); a list sampled from the distribution with a large amount of overlap (after the elbow point); the number of comparisons with increasing number of runs on the x-axis; a zoomed-in view of the red box from the comparison plot.

tion from the Overlapping Staircase, see Fig. 15. We create a list of 20,000 elements in 300 runs and let the overlap between runs vary from 1 through 60 on the x-axis. As the amount of overlap increases, the advantage of PersiSort degrades. We observe an elbow point at $r = 40$ because the dynamic box depth increases drastically from 3 ($r = 40$) to 11 ($r = 60$), increasing the comparison complexity dramatically.

9 Discussion

PersiSort is well-suited for parallelism, which is left for future work. Introducing parallelism will, however, have no impact on the number of comparisons performed. In cases where the data points have high dynamic box depths, the theoretical performance of AdaptMerge is comparable to that of simpler merge algorithms, which suggests that a highly optimized implementation of AdaptMerge can have practical merit.

We observe in many cases that Python Sort (CPython standard library implementation of PowerSort v 3.11.2) performs equal or worse than our in-house implementation of PowerSort. Our experimental results thus hint at the possibility of using AdaptMerge as a merge subroutine for Python Sort.

Finally, it is vital for PersiSort that its merge subroutine is AdaptMerge. If a simple linear merge subroutine is used, the comparison complexity becomes $\mathcal{O}\left(n^2\right)$, which is also the worst-case number of element moves performed by PersiSort. It would be interesting to investigate the comparison complexity of PersiSort if AdaptMerge is replaced by other types of merge subroutines (e.g., [17]).

Acknowledgements

JK was supported by the Independent Research Fund Denmark (DFF) grant 9131-00113B. BW was partially supported by grants from National Science Foundation (NSF) DMS-2301361 and IIS-2145499.

References

[1] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau. On the worst-case complexity of TimSort. In Y. Azar, H. Bast, and G. Herman, editors, *Proceedings of the 26th Annual European Symposium on Algorithms*, volume 112 of *LIPIcs*, pages 4:1–4:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[2] J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[3] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[4] R. Biswas, S. Cultrera di Montesano, H. Edelsbrunner, and M. Saghafian. Geometric characterization of the persistence of 1D maps. *Journal of Applied and Computational Topology*, 2023.

[5] S. Buss and A. Knop. Strategies for stable merge sorting. ArXiv preprint arXiv:1801.04641, 2018.

[6] G. Carlsson, A. J. Zomorodian, A. Collins, and L. J. Guibas. Persistence barcodes for shapes. *Proceedings of the Eurographs/ACM SIGGRAPH Symposium on Geometry Processing*, pages 124–135, 2004.

[7] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In T. Asano, T. Ibaraki, H. Imai, and T. Nishizeki, editors, *Algorithms*, pages 251–260, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[8] S. C. di Montesano, H. Edelsbrunner, M. Henzinger, and L. Ost. Dynamically maintaining the persistent homology of time series. ArXiv preprint arXiv:2311.01115, 2023.

[9] H. Edelsbrunner and J. Harer. Persistent homology - a survey. *Contemporary Mathematics*, 453:257–282, 2008.

[10] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction.* American Mathematical Society, 2010.

[11] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28:511–533, 2002.

[12] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9:66–104, 1990.

[13] A. Elmasry and M. L. Fredman. Adaptive sorting: an information theoretic perspective. *Acta Informatica*, 45:33–42, 2008.

[14] W. C. Gelling, M. E. Nebel, B. Smith, and S. Wild. Multiway powersort. *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 190–200, 2023.

[15] M. Glisse. Fast persistent homology computation for functions on $\mathbb{R}$. ArXiv preprint arXiv:2301.04745, 2023.

[16] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977.

[17] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1), 1972.

[18] D. Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching.* Addison-Wesley, 1973.

[19] C. Levcopoulos and O. Petersson. Splitsort—an adaptive sorting algorithm. *Information Processing Letters*, 39(4):205–211, 1991.

[20] C. Levcopoulos and O. Petersson. Adaptive heapsort. *Journal of Algorithms*, 14(3):395–413, 1993.

[21] C. Levcopoulos and O. Petersson. Exploiting few inversions when sorting: Sequential and parallel algorithms. *Theoretical Computer Science*, 163(1-2):211–238, 1996.

[22] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, 1985.

[23] E. Munch. A user's guide to topological data analysis. *Journal of Learning Analytics*, 4(2):47–61, 2017.

[24] J. I. Munro and S. Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In Y. Azar, H. Bast, and G. Herman, editors, *Proceedings of the 26th Annual European Symposium on Algorithms*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 63:1–63:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[25] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, and H. A. Harrington. A roadmap for the computation of persistent homology. *EPJ Data Science*, 6(17), 2017.

[26] T. Peters. Timsort description. `https://svn.python.org/projects/python/trunk/Objects/listsort.txt`, May 2023.

[27] S. Wild. Educational powersort implementation. `https://colab.research.google.com/drive/13jJDr7dcEz2Ub48TzOT-DaJYCNc5UduR?usp=sharing#scrollTo=KorRRNz_ulvA`.

[28] S. Wild. PyCon US 2023 Talk on Powersort. https://www.wild-inter.net/posts/powersort-pycon-talk, 2023.

[29] L. Yan, T. B. Masood, R. Sridharamurthy, F. Rasheed, V. Natarajan, I. Hotz, and B. Wang. Scalar field comparison with topological descriptors: Properties and applications for scientific visualization. *Computer Graphics Forum (CGF)*, 40(3):599–633, 2021.

A Merge Policies of Adaptive Sorting Algorithms

An adaptive sorting algorithm was described by Wild [28] as a two-step procedure. First, the algorithm detects all the runs. Second, it merges the runs in some order determined by a merge policy. The order in which the runs are merged defines a binary tree, referred to as a *Merge Tree*, where leaves represent runs, internal nodes represent intermediate sorted sublists, and the root represents the entire sorted list.

For comparative analysis, we review merge policies of a number of adaptive sorting algorithms. We assume the input is a list with n elements in r runs.

Natural MergeSort [7] has a simple merge policy independent of the runs' values and sizes. Simply put, it builds a (balanced) Merge Tree by merging adjacent runs in the tree. We omit Natural MergeSort from our experiments because it behaves similarly to TimSort.

TimSort [1, 26] determines its merge policy by maintaining a stack of runs. Runs are added to the stack based on the order in which they are discovered. Merging runs from the stack is based on the four rules described below. When the last run is added to the stack, the algorithm collapses the stack by merging runs from top to bottom. The main idea behind these merge triggering rules is that they "balance the run lengths as closely as possible, while keeping a low bound on the number of runs we have to remember" [26].

At any time, let h denote the height of the stack R. Let R_k ($1 \leq k \leq h$) be the k-th run from the top of the stack, and let $\ell_k := |R_k|$ be the number of elements in R_k. TimSort's merge policy is as follows (based on Algorithm 3 of [1]). Initially, we perform a run decomposition of an input list and set the stack to be empty. At each step of the iteration (until all runs are handled), we remove a run from the run decomposition and push it to the stack. We follow the following four rules to trigger merges:

- If $h \geq 3$ and $\ell_1 > \ell_3$ then merge runs R_2 and R_3;
- else if $h \geq 2$ and $\ell_1 > \ell_2$ then merge runs R_1 and R_2;
- else if $h \geq 3$ and $\ell_1 + \ell_2 \geq \ell_3$ then merge runs R_1 and R_2.
- else if $h \geq 4$ and $\ell_2 + \ell_3 \geq \ell_4$ then merge runs R_1 and R_2.

The analysis of TimSort was proved to be difficult. Auger et al. [1] showed that the runs on the stack are of exponentially increasing size, and TimSort performs $\mathcal{O}(n + n\log r)$ comparisons.

PowerSort was introduced by Munro and Wild [24] and is currently used in Python version 3.11.2. It follows the sequential left-to-right nature of TimSort with some changes in the merge policy. Let R_1 and R_2 be two adjacent runs of lengths ℓ_1 and ℓ_2 respectively. They start at list indices i_1 and i_2 respectively, that is, $R_1 = X[i_1, i_1 + \ell_1 - 1]$ and $R_2 = X[i_2, i_2 + \ell_2 - 1]$. The *power* of the boundary between R_1 and R_2 is defined as

$$p(R_1, R_2) = \max\{\ell \in \mathbb{N} : \lfloor 2^\ell \cdot (i_1 + \ell_1/2)/n \rfloor = \lfloor 2^\ell \cdot (i_2 + \ell_2/2)/n \rfloor\}$$

Similarly to TimSort, PowerSort scans the runs. Assuming there is a run stack $R_0, R_1, \ldots$, and we have discovered a new run R. The algorithm compares the power between R and runs in the stack. If $p(R_0, R) < p(R_0, R_1)$, then R_0 is popped from the stack and merged with R, resulting in R'. Moving forward, if $p(R_0, R) < p(R_1, R_2)$ then R_1 is popped and merged into R'' and so on. Like TimSort, when there are no more runs to process, the stack is collapsed into a single sorted list by merging runs from top to bottom. The comparison complexity of PowerSort [24] is $\mathcal{O}(n + n\log r)$.

B Pseudocode

We provide pseudocode for AdaptMerge, FingerMerge, and PersiSort. The pseudocode of AdaptMerge is included in Fig. 16. By convention, $A[3, 2]$ is an empty list, and $A[3, 3]$ is a single element list. Under these conventions, exponential search for, say, 5 in the list $B = [7, 8, 9]$ returns index -1 such that no elements of B are added to the output C. In the next iteration, the algorithm searches for the predecessor of 7 in, say, $A = [5, 6, 10]$, which will find the predecessor 6 at index 1 and extend C by $A[0, 1] = [5, 6]$. Furthermore, if the exponential part of the exponential search overshoots the end of a list, then it should "round down" to the end of the list and continue to the binary search phase.

```
AdaptMerge($A$, $B$):
    $C$ = empty list of length $n_a + n_b$
    $i_0 = j_0 = 0$
    while not ($i_0 == n_a - 1$ or $j_0 == n_b - 1$):
        $i_1$ = ExponentialSearch($B[j_0], A[i_0, n_a - 1]$)
        C.extend($A[i_0, i_1]$)
        $i_0 = i_1$
        $j_1$ = ExponentialSearch($A[i_0], B[j_0, n_b - 1]$)
        C.extend($B[j_0, j_1]$)
      $j_0 = j_1$
    C.extend($A[i_0, n_a]$)
    C.extend($B[j_0, n_b]$)
    return C
```

Figure 16: Pseudocode for the AdaptMerge algorithm of Carlsson et al. [7].

The three-way FingerMerge calls AdaptMerge twice, as shown in Fig. 17. The algorithm also needs to ensure that the monotonicity of A, B, C, and AB is the same, which can easily be solved with start and end pointers to the lists. This, however, makes FingerMerge ill-suited for algorithms where reverses are costly.

```
FingerMerge($A$, $B$, $C$):
    $AB$ = AdaptMerge($A, B$)
    return AdaptMerge($AB$,$C$)
```

Figure 17: Pseudocode for the FingerMerge algorithm.

The pseudocode of PersiSort is shown in Fig. 18. Given an input list X with n elements in r runs, we denote the sequence of runs as $R_0, R_1, \ldots, R_{r-1}$. We first identify the set of extrema E from X (line 2). We then compute the initial set of (neighboring) persistence pairs (line 3). We repeat the following procedure until the list is sorted, i.e., when there are at most two extrema in E (line 4).

1. For each persistence pair (line 5):
    - Perform FingerMerge on the two or three runs intersecting the pair (lines 6-11).
        * If the pair contains boundary extrema, perform a two-way merge (lines 6-9);
        * Otherwise, perform a 3-way merge (line 11);
    - Remove the pair of extrema from the set of extrema E (line 12).
2. Recompute the persistence pairs by updating the pairing candidates (line 13).

```
1  PersistenceSort($X$)
2      $E$ = DetectExtrema($X$)
3      PersistencePairs = ComputePairs($E$)
4      while $|E| > 2$:
5          for pair in PersistencePairs:
6              if $e_0$ in pair:
7                  AdaptMerge($R_0$, $R_1$)
8              elif $e_{r-1}$ in pair:
9                  AdaptMerge($R_{-2}$, $R_{-1}$)
10             else:
11                 FingerMerge($R_{pair-1}$, $R_{pair}$, $R_{pair+1}$)
12         E.remove(pair)
13         PersistencePairs = RecomputePairs($E$, $X$)
```

Figure 18: The pseudocode for the PersiSort algorithm. R_{-1} means the last run, and R_{-2} is the second to last run. R_{pair} is the run in the current configuration that contains the persistence pair, and $R_{pair\pm1}$ indicate the runs before and after the current run, respectively.