Anytime Sorting Algorithms (Extended Version)

Emma Caizergues, François Durand, Fabien Mathieu

Anytime Sorting Algorithms (Extended Version)

Emma Caizergues[1,2] , **François Durand**[1] , **Fabien Mathieu**[3]
[1]Nokia Bell Labs, Massy, France
[2]CNRS - Lamsade, Université Paris Dauphine - PSL, Paris, France
[3]Swapcard Lab, Paris, France

Abstract

This paper addresses the anytime sorting problem, aiming to develop algorithms providing tentative estimates of the sorted list at each execution step. Comparisons are treated as steps, and the Spearman's footrule metric evaluates estimation accuracy. We propose a general approach for making any sorting algorithm anytime and introduce two new algorithms: multizip sort and Corsort. Simulations showcase the superior performance of both algorithms compared to existing methods. Multizip sort keeps a low global complexity, while Corsort produces intermediate estimates surpassing previous algorithms.

1 Introduction

Motivation The objective of anytime sorting algorithms is to yield accurate estimates of a list's sorted order with a limited number of comparisons. This problem emerges particularly when the cost of comparing two items outweighs other computational operations.

For example, in scenarios involving human interaction, assigning a utility to each element of the list is not a reliable technique and it is better to proceed by comparing pairs [Giesen *et al.*, 2009]. To illustrate this, consider a supervised learning based on sorted lists (e.g. learning to assert the quality of translations). Providing sorted lists instead of scores is relevant because humans are not good at grading many items, whereas pairwise comparisons are relatively accurate. However, human comparisons are considerably more time-consuming than the other operations, which are performed by computers. To optimize efficiency, it is then crucial to distinguish comparison complexity from global complexity. In this context, anytime sorting has two main applications: instead of performing exact sorting on a small part of the raw input, one may wish to sort approximately more inputs. Uncertainty on the input and processing rates make it desirable to have an anytime approach. Also, users may be worn out after a certain amount of questions and want to stop the sorting process prematurely, so it is essential for the algorithm to generate accurate intermediate estimates that closely approximate the final sorted list.

Similar problems arise in situations where the considered items involve massive datasets requiring data transfers for each comparison [Mesrikhani and Farshi, 2018]. The expense associated with each comparison necessitates the exploration of efficient algorithms that minimize the number of comparisons. Furthermore, practical constraints may require the sorting process to be interrupted before the algorithm's termination, so we need a reliable and accurate estimate of the sorted order based on the progress made thus far.

Related work Limiting the number of comparisons when sorting is a well-known problem [Cormen *et al.*, 2009]. Among many famous algorithms such as quicksort, mergesort, heapsort, binary insertion sort and Shellsort, the Ford-Johnson algorithm [Ford and Johnson, 1959] stands out as very close to the theoretical limit in number of comparisons, and even optimal for certain values [Peczarski, 2004]. These algorithms do not consider a possible premature interruption.

A related problem is that of sorting under partial information. This problem entails determining a total order that is partially known, in the sense that a compatible partial order is available [Cardinal *et al.*, 2010]. Many variants of the sorting problem can be found within the realm of partial information, including scenarios such as ranking the k best values [Dushkin and Milo, 2018]. While the partial information framework does not consider intermediate estimates, similarities with our contributions exist as we propose an algorithm that exploits the partial information extracted so far to determine the next comparison. Furthermore, partial information relates to classical sorts when considering empty information. For example, initializing Algorithm 1 in [Cardinal *et al.*, 2010] with no prior amounts to the classical binary insertion sort, and their Algorithms 2 and 3 are both equivalent to mergesort (in its bottom-up version, which we discuss further in this paper).

Sorting when the probabilistic distribution of the inputs is known is another relevant problem to consider. In [Moran and Yehudayoff, 2016], a variant of insertion sort is proposed as a solution, which has also been employed in [Peters and Procaccia, 2021] for sorting with human comparisons. However, our approach differs in two main aspects. Firstly, our algorithms focus on providing good intermediate estimates throughout the sorting process. Secondly, we assume equal probabilities for all input permutations. In that case, their algorithm is equivalent to binary insertion sort.

The potential interruption in the sorting process places us within the domain of anytime algorithms, which maintain result estimates at all times. Surprisingly, sorting has received

limited attention in this literature, with studies focusing on specific algorithms like selection sort, Shellsort, and quicksort [Grass and Zilberstein, 1996; Horvitz, 1988]. However, the metrics used to measure the estimation error are not always true distances. In our benchmark, we exclude selection sort due to its $O(n^2)$ comparison complexity, while including Shellsort and quicksort for evaluation.

Among related problems, *progressive algorithms* can also be interrupted at any time, but the focus is on provable bounds for worst-case performance rather than on empirical average efficiency [Alewijnse *et al.*, 2014]. Contract-based algorithms also make a trade-off between time and accuracy, but assume that the available time is known in advance [Zilberstein, 1995; Zilberstein, 1996].

One of the most closely related studies is the work by [Giesen *et al.*, 2009], which delves into the link between the number of comparisons performed and the estimation error. Using Spearman's footrule metric to evaluate the error, they show that, for a list of size n, the error after $k \leq n(\ln(n) - 6)$ comparisons is at least $n^2 e^{-k/n-6}$ in the worst case. Conversely, they introduce ASort, an anytime algorithm with guarantee that the error is less than $n^2 e^{-k/6n+1}$. While these results show that ASort is asymptotically optimal in some sense, the ratio between the two bounds above ($e^{5k/6n+7}$) is at least $e^7 > 1000$ and potentially unbounded. This means that the practical performance of ASort remains to be investigated. One of the goals of our work is to provide a practical benchmark for ASort while also introducing new algorithms with superior empirical performance.

Contributions In this paper, we make the following contributions to the field of anytime sorting:

1. We generalize the work on ASort by [Giesen *et al.*, 2009] under the anytime framework and propose the anytime sorting problem, where the *performance profile* of an algorithm is measured by the *Spearman's footrule metric* between its tentative estimates and the perfectly sorted list, as a function of the number of comparisons already performed.
2. We reexamine classical sorting algorithms from the perspective of anytime sorting.
3. We propose simple heuristics called *estimators*, which leverage the available partial information at each step to approximate the final sorted result.
4. We present an enhanced version of the traditional mergesort algorithm called *multizip sort*. This algorithm is specifically designed to improve the anytime sorting capabilities.
5. We introduce the *Corsort* family of anytime sorting algorithms, which rely on estimators to determine the next comparison to perform.
6. Using extensive simulations, we analyze the performance of Corsort, multizip, and ASort.
   - Corsort has a quasi-optimal comparison complexity and provides superior intermediate estimates compared to existing state-of-the-art sorting algorithms. Its main drawback is its global complexity.
   - Multizip sort has a better comparison complexity. It maintains in most cases the second-best intermediate results. Its main drawback is its interruption cost.
   - ASort has less optimal comparison complexity and performance profile. Its main benefits are low global complexity and absence of interruption cost.

A comprehensive summary of the computational complexities is provided in Table 1.

Limitations The anytime sorting algorithms we propose are designed to address specific scenarios, and their suitability depends on the following assumptions:

1. It is impractical or unreliable to assign numerical utilities to the items to sort.
2. The sorting process may be interrupted due to tight time constraints or resource limitations.
3. An approximate ranking is acceptable, although the approximation error does matter.
4. Good empirical performance is required, possibly at the expense of formal guarantees.
5. The cost of comparisons is orders of magnitude higher than other computing operations.
6. Worse case scenarios and non-uniform dataset/inputs are not the point of the study.

If these assumptions do not hold, alternative solutions may be more appropriate. For example, algorithms like Ford-Johnson are well-suited for situations with costly comparisons and a low likelihood of interruption; contract-based algorithms offer better suitability when interruption times are known and formal guarantees are required. ASort offers similar guarantees, even though it is outperformed experimentally.

Furthermore, our proposal does not rely on any specific prior information about the input list. This makes our proposal very generic but also means that if such prior exists, we do not leverage it to reduce the average number of comparisons required, as it is done in [Peters and Procaccia, 2021].

Lastly, we have not provided a formal proof that our Corsort algorithm has an average comparison complexity in $O(n \ln(n))$. This conjecture is only supported by simulations demonstrating its superior performance compared to heapsort, Shellsort, and quicksort/ASort.

Roadmap The rest of the article is organized as follows. Section 2 formally defines anytime sorting, presents a taxonomy of anytime sorting algorithms, and introduces our novel approaches: estimators, multizip sort, and Corsort. Section 3 evaluates the proposed solutions through simulations. Section 4 concludes.

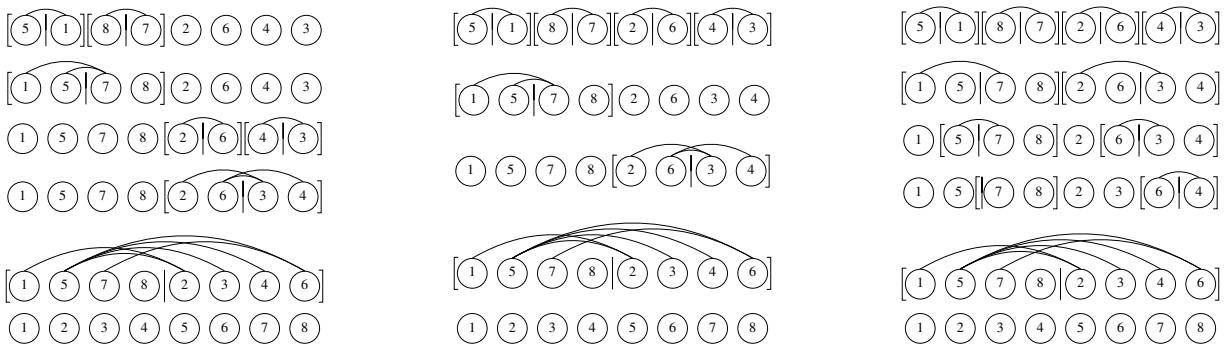
Published version The present paper is an extended version of [Caizergues *et al.*, 2024].

2 Anytime sorting

Formally, we want to sort a list $X = (X[1], \ldots, X[n])$, where $n > 0$, by performing comparisons of the type: is $X[i]$ less than $X[j]$? An *anytime sorting algorithm* is an algorithm capable, after k comparisons have been performed, of returning an estimate X_k of the result. By convention, if k is greater than the number of comparisons needed for the algorithm's termination, then X_k is the sorted list. We measure the error of X_k by computing the Spearman's footrule metric between

	Comparison complexity	Global complexity	Native estimator	Interest(s)
Ford-Johnson	$\sim n\log_2(n)$	$O(n^2)$	No	Comparison complexity
Binary insertion	$\sim n\log_2(n)$	$O(n^2)$	Yes	Comparison complexity
Quicksort	$O(n\ln(n))$	$O(n\ln(n))$	Yes	Fast termination in practice
ASort	$O(n\ln(n))$	$O(n\ln(n))$	Yes	Good native estimator with theoretical guarantees
Top-down merge	$\sim n\log_2(n)$	$O(n\ln(n))$	Yes	Complexities
Bottom-up merge	$\sim n\log_2(n)$	$O(n\ln(n))$	Yes	Good performance profile, complexities
Multizip	$\sim n\log_2(n)$	$O(n\ln(n))$	Yes	Very good performance profile, complexities
Corsort	$O(n^2)^*$	$O(n^4)^*$	No	Best performance profile, comparison complexity*

Table 1: Cost of the algorithms considered in this paper (we exclude heapsort and Shellsort due to their poor termination times, see Figure 5). All complexities are well-known results except for the last two rows. All algorithms can use the estimator ρ when interrupted. Using ρ requires $O(n\ln(n))$ for Corsort, $O(n^3)$ operations for the other algorithms. Native estimators, when they exist, can be used directly.
* Simulations suggest $O(n\ln(n))$ comparisons, yielding a global complexity of $O(n^3\ln(n))$ operations.



(a) Top-down merge execution. (b) Bottom-up merge execution. (c) Multizip sort execution.

Figure 1: Sorting the list $X = (51872643)$ with top-down merge, bottom-up merge and multizip sort. Each edge represents a comparison. The bracket notation [|] delimits two sublists already sorted that are being merged. For more details, cf. Appendix A.1.

X_k and the sorted list [Diaconis and Graham, 1977], i.e. the sum of the absolute differences between the ranks of elements in X_k and in the sorted list : $S_k = \sum_i |r(X_k[i]) - i|$, where $r(x)$ denotes the rank of x in the sorted list. The function $k \to S_k$, which represents the evolution of the error made as the algorithm runs, is called its *performance profile*. Ideally, we are looking for an anytime sorting algorithm whose performance profile is consistently lower than that of the other algorithms tested.

2.1 Classical sorting

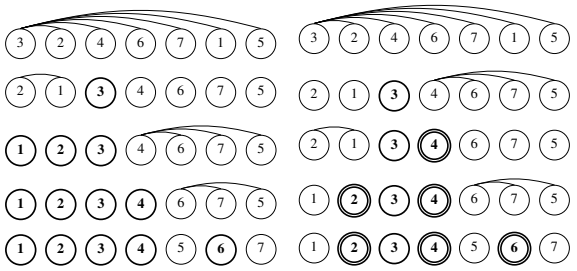
Some classical algorithms can be seen as anytime because they maintain a list that converges to the sorted list and can be used as an estimate X_k. This is the case of quicksort, mergesort, binary insertion sort, and Shellsort, all of which we shall implement in a way that is favorable to the spirit of the original algorithm: for quicksort, for instance, the position of the pivot is updated in the list after each comparison.

Changing the order in which comparisons are made may improve the intermediate estimates X_k. For mergesort, the standard recursive implementation, also called *top-down*, sorts the left part, then the right part of each considered sub-list (recursively), and reunites them by a merge procedure [Cormen *et al.*, 2009, Chapter 2]. However, it is also possible to go through it *bottom-up*, by sorting sub-lists of increasing size [Knuth, 1998, Chapter 5.2.4]. Figure 1 illustrates this difference on an example[1] of size 8: the top-down execution sorts completely the left side of the list before completely sorting the right side of the list, whereas the bottom-up execution starts by sorting all lists of size 2, both on left and right sides. In all classical mergesort implementations, the merge procedure is a single block. We propose a variant of the bottom-up implementation, which we call *multizip sort*, where all merge procedures of a given depth of the recursion tree are interleaved. Figure 1c illustrates the idea: the first part of the execution is unchanged and sorts all lists of size 2. Then, when *bottom-up* sorts the two lists of size 4 sequentially, *multizip* alternates comparisons in the two lists until they are both sorted (cf. Appendix A.1 for details). The rationale for proposing *multizip sort* will be discussed in Section 2.2. Note that all the considered variants of mergesort make the exact same comparisons, but not in the same order. Intuitively, bottom-up and multizip ensure that the information available on the elements of the list (through comparisons) is more balanced. This should be more favorable for the intermediate estimates X_k, as Section 3.2 will verify experimentally.

For quicksort, balancing the comparisons is equivalent to the ASort algorithm [Giesen *et al.*, 2009], using quickselect [Hoare, 1961] as a subroutine for median identification.

[1] For brevity, we omit the commas to note the lists taken as examples in the tables and figures. For example, the list (a, b, c, d) is noted as $(abcd)$.

Specifically, ASort determines the median, separates smaller from larger elements, and proceeds recursively. When utilizing quickselect to identify the median, which uses successive pivots like quicksort, ASort performs precisely the same comparisons as quicksort, albeit not in the same order. This is illustrated in Figure 2.



(a) Quicksort execution. (b) ASort execution.

Figure 2: Sorting the list $X = (3246715)$ with quicksort or ASort. A node appears in bold if it has already been used as a pivot: it partitions the list into smaller elements on the left and larger ones on the right. Intermediate steps that do not perform any comparison are omitted. For ASort (2b), we use quickselect [Hoare, 1961] as median identification subroutine. Each step represents the application of a pivot, and each edge represents a comparison. A node is circled twice if it has been identified as a median. In the third step, since the median 4 has been found, we must compute the median of the left sub-list (213); but note that it is useless to make any comparison with element 3, previously used as a pivot, because it is is already in its final position.

Certain other classical algorithms allow to obtain an estimate X_k by a natural transformation of their current state. For example, heapsort keeps in memory the heap associated to the partially sorted list. To obtain a fair estimate from the algorithm, we can first go through the heap (backwards), then through the elements already sorted (forwards).

Finally, some algorithms like Ford-Johnson, whose internal state does not have a structure naturally close to a list, do not seem to have a natural estimator. Since it is not always trivial to translate the execution of an algorithm into intermediate estimators, we show in the next section how to produce X_k for any comparison-based sorting algorithm.

2.2 Estimators

To make an anytime version of any sort, we propose to use an estimator that ignores the execution details and solely relies on the historical record of the comparisons made.

Let $C_k = \{X[i_1] < X[j_1], \dots, X[i_k] < X[j_k]\}$ be the result of k comparisons. C_k defines a partial order $\preceq_k$ on the elements of the list by transitive closure[2]. We call *estimator* a function that associates to any partial order one of its *linear extensions*, i.e. a compatible total order.

[2]We assume for clarity that all elements are distinct. In case of redundancy, uniqueness can be enforced by appending to each element x its index i in the initial list. For example, $(17, 42, 42)$ should be seen as $((17, 1), (42, 2), (42, 3))$. When sorting the list, it is possible to be in a situation where we already know that $(17, 1) \preceq_k (42, 2)$ but we do not yet know how to compare $(17, 1)$ and $(42, 3)$.

An estimator is optimal if it always finds a total order that minimizes the expected Spearman's footrule metric with a uniform random linear extension of the input partial order. Assuming the linear extensions of $\preceq_k$ are known, this is equivalent to solving an assignment problem [Kuhn, 1955] where the cost of assigning element x to index i is the total error generated by the decision over all extensions. As the metric is a L_1-norm, we can also associate to each element its median rank, which gives the optimal result as long as all medians are distinct integers. However, both approaches are computationally challenging, as enumerating the linear extensions is known to be #P-complete [Brightwell and Winkler, 1991].

In practice, we propose to use classical *score-and-sort* heuristics [Calauzenes and Usunier, 2020; Robertson, 1997]: each element gets a score that reflects its estimated ranking in the list, and we return the list associated with sorting the scores. Recall that global complexity is distinct from comparison complexity: we assume that sorting n numerical scores is much faster than comparing two elements of the list. In case of ties, we return the elements in their original order. By a slight abuse of language, we also call *estimator* the score function itself.

To build our heuristics, we define descendants and ancestors as follows: if x is an element of the list, $d_k(x) = |\{y \in X : y \preceq_k x\}|$ and $a_k(x) = |\{y \in X : x \preceq_k y\}|$ are respectively the number of known descendants and ancestors of x in $\preceq_k$ (x is included in both sets by convention). A simple way to compute d_k and a_k for a given k is to build the transitive closure of C_k, which can be done in $O(n^3)$ using the Flyod-Marshall algorithm [Cormen *et al.*, 2009, chapter 25.2]. If one wishes d_k and a_k for all values of k, it is best to maintain the sets of descendants and ascendants: after each new comparison, update the descendants of the ancestors of the greater element with the descendants of the smaller element, and conversely (cf Appendix A.2 for details). The cost of this incremental approach is $O(n^2)$ for each new comparison performed.

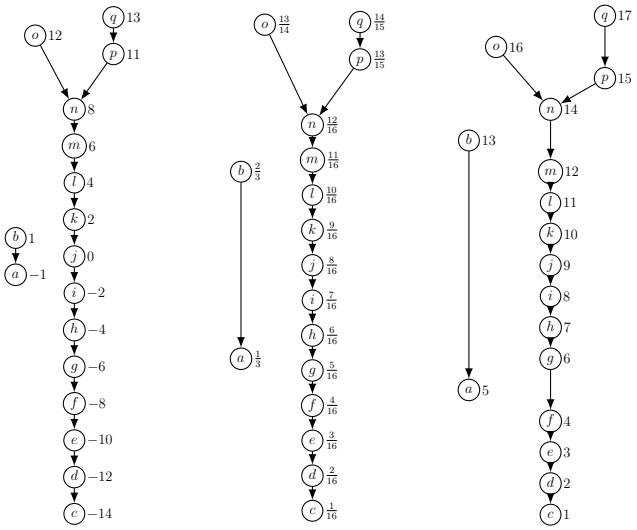
Using d_k and a_k, we consider the following score functions:

- Δ_k, defined by $\Delta_k(x) = d_k(x) - a_k(x)$. Δ_k assigns to each x a score that reflects the average between its lowest and highest possible positions.
- ρ_k, defined by $\rho_k(x) = d_k(x)/(d_k(x) + a_k(x))$. ρ_k positions x as if its descendants and ancestors were on average regularly spaced within the whole list.

If there is no ambiguity, we will omit the index k. Δ, ρ, but also the median rank m are valid estimators: one can check that if an element is greater than another in a partial order, then its score is higher, which ensures that the returned estimate is a linear extension. In particular, when the result of all comparisons is known, sorting the elements according to the score function returns the sorted list.

Figure 3 exhibits a partial order where the three estimators give distinct results. Here, ρ is better than Δ because it positions better the small chain component (nodes a and b) with respect to the large Y-shaped one (nodes c to q). However, it is still surpassed by the median rank m, which is optimal here because all medians are distinct (cf. above).

Figure 3 illustrates that Δ and ρ distort their estimate when the partial order is a combination of chains and Y-shaped com-



(a) Δ scores. (b) ρ scores. (c) m scores.

Estimator	Returned estimate	$\overline{S}$
Δ	$(cdefghiajbklmnpoq)$	16.2
ρ	$(cdefgahijklbmnpoq)$	14.0
m	$(cdefaghijklmbnpoq)$	13.9

Figure 3: Example of score-based selection of a linear extension. The input partial order, represented by the edges of the graph, is the transitive closure of: $a \prec b$, $c \prec d \prec \ldots \prec n \prec o$, $n \prec p \prec q$. Heights are proportional to scores. Returned estimates and expectation $\overline{S}$ of the error S are also provided for completeness.

ponents. Yet, during a standard mergesort execution, a typical partial order is a combination of chains (the sorted sublists) and one Y-shaped component (the ongoing merge). We introduced the multizip sorting earlier to avoid this scenario, so that a typical partial order will be multiple Y-shaped components of similar size (the ongoing merge procedures). In essence, multizip is a variant of mergesort tailored for Δ and ρ.

Multizip complexity As multizip schedules the same comparisons as mergesort, its comparison complexity is equivalent to $n \log_2(n)$ even in worst case, which is asymptotically optimal [Cormen *et al.*, 2009]. Each step requires $O(1)$ computing operations so the global complexity is $O(n \ln(n))$. If an interruption occurs, we need to build the transitive closure of the comparisons made, which is done in $O(n^3)$.

In our preliminary work [Caizergues *et al.*, 2023], we found out that ρ generally outperforms Δ as an estimator, and therefore we adopt it as the default choice. However, Δ can serve another purpose, as elaborated in Section 2.3.

2.3 Comparison-Oriented sort

We call *Corsort* (*Comparison-ORiented Sort*) a sorting algorithm that, at each step of its execution, selects the next comparison based on the current partial order $\preceq_k$. We assume that only pairs that are not comparable yet (according to $\preceq_k$) are chosen until convergence is achieved.

The heart of a Corsort, i.e., the choice of the next comparison, aims at two objectives. Firstly, it must ensure fast termination, i.e., minimize the total number of comparisons. This is a problem of *sorting under partial information*, which

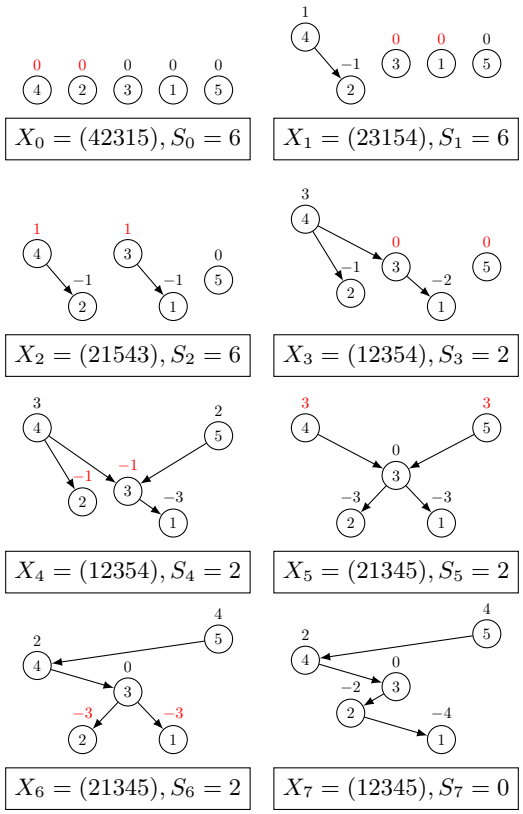


Figure 4: Execution of Corsort-Δ on the list $X = (42315)$. Each step k depicts the partial order after k comparisons. The next comparison to perform is visually highlighted by the two vertices whose values of Δ_k (displayed above each element) are in red. Each subfigure represents ρ_k as each element's height and gives the corresponding estimate X_k based on ρ_k and the associated error S_k.

amounts to choosing a comparison whose result is as uncertain as possible [Cardinal *et al.*, 2010]. To this end, we propose to rely on closeness for a score function. The rationale is that close scores should indicate a high uncertainty regarding the outcome of the comparison. Secondly, to improve the quality of intermediate estimates X_k, we need to acquire information on elements for which we have little, as these create uncertainty that will be reflected in the error S_k. We introduce $I_k(x) = d_k(x) + a_k(x)$ to represent the amount of "information" acquired on an element x, and we wish to compare elements for which I_k is low. In our preliminary work [Caizergues *et al.*, 2023], we compared several variants of Corsort and we selected the following one, called Corsort-Δ:

- The next comparison is made by seeking a pair of incomparable items whose Δ scores are as close as possible.
- We use I_k for tie-breaking. Specifically, among the pairs that minimize Δ-closeness, we pick up one that minimizes $\max(I_k(x), I_k(y))$.

The intermediate estimates X_k can be given by any estimator. In practice, we use ρ. For the pseudocode, cf. Appendix A.3. As an example, Figure 4 shows the execution of the selected Corsort on the list $X = (42315)$. As we argued earlier that ρ is better than Δ, one may wonder why Δ is preferred

for the choice of the comparison. An intuition that explains the advantage of Δ is the need of balance between termination and intermediate estimates: ρ can take $O(n^2)$ distinct values, against $O(n)$ for Δ. As a result, Δ generates more ties than ρ (a typical case is X_4 on Figure 4: $\rho_4(2) < \rho_4(3)$ but $\Delta_4(2) = \Delta_4(3)$), so the tie-breaking rule, which targets accuracy of the estimate, is used more often. Experimentally, this gives a better performance profile than a ρ-based Corsort, at the price of a (slightly) longer termination.

Complexity Corsort never compares a given pair twice, so it makes at most $n(n-1)/2$ comparisons. At each step, one needs to compute the next pair, compare, then update a and d, for a cost of $O(n^2)$ operations, which sums up to a global complexity in $O(n^4)$. The cost of interruption, a score-and-sort in $O(n \ln(n))$, is comparatively negligible.

3 Evaluation

To compare the performance of ASort, multizip sort and Corsort with the classical sorting algorithms discussed in Section 2.1, we have developed an open-source Python package specifically designed for creating anytime sorting algorithms and evaluating their effectiveness [Caizergues *et al.*, 2023] (cf. Appendix B).

3.1 Uninterrupted behavior

Figure 5 shows the termination time (measured in number of comparisons) for the different algorithms. The curves for top-down merge, bottom-up merge, and multizip sort coincide as they represent different scheduling variations of mergesort. Similarly, the curves for ASort and quicksort are identical as they involve the same comparisons. We consider values of the list size n ranging from 8 to 2048. For each point, we generate 10,000 random lists and we record the number of comparisons required for a complete sort. The y-axis shows the relative deviation from the theoretical lower bound of $\log_2(n!) = n\log_2(n) - n/\ln(2) + \log_2(2\pi n)/2 + o(1)$ [Cormen *et al.*, 2009]: a curve closer to 0 indicates closer proximity to the optimal performance. To give a comprehensive view of the results distribution, each algorithm's median is depicted by a dark curve, while the 2.5% to 97.5% quantiles form a 95% confidence interval represented by a light area.

We observe that Heapsort requires almost twice as many comparisons as necessary, and Shellsort (here implemented with Ciura's gap sequence [Ciura, 2001]) approximately 60% more. Quicksort shows better performance with less than 30% overhead for the studied values of n, but relying on a pivot selection introduces high variance. The remaining four sorting algorithms demonstrate even lower overhead: 6% for Corsort, 2% for mergesort, 0.6% for binary insertion, and 0.3% for Ford-Johnson. These algorithms also exhibit negligible variance. Based on these findings, we conclude that Corsort emerges as a promising candidate, surpassing all other algorithms except those with comparison complexities asymptotically equivalent to $n\log_2(n)$ [Ford and Johnson, 1959]. Moreover, our simulations confirm the asymptotical optimality of mergesort, further reinforcing the position of its multizip variant as a strong candidate for anytime sorting.

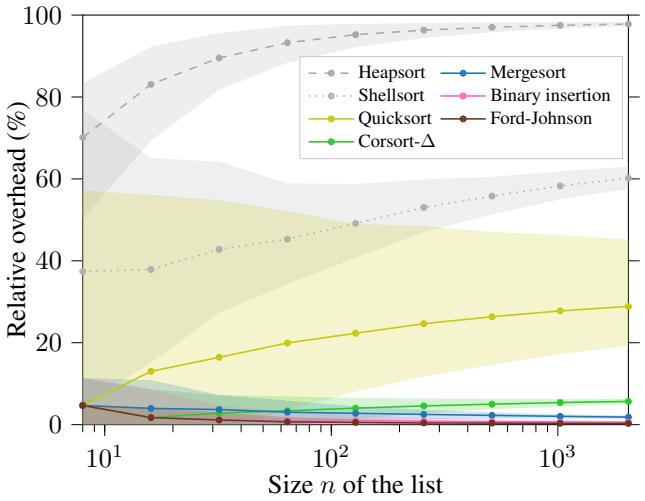


Figure 5: Number of comparisons required for algorithm termination, as a relative overhead compared to the information-theoretic lower bound. Each point is obtained by sorting 10,000 random lists.

3.2 Performance profiles

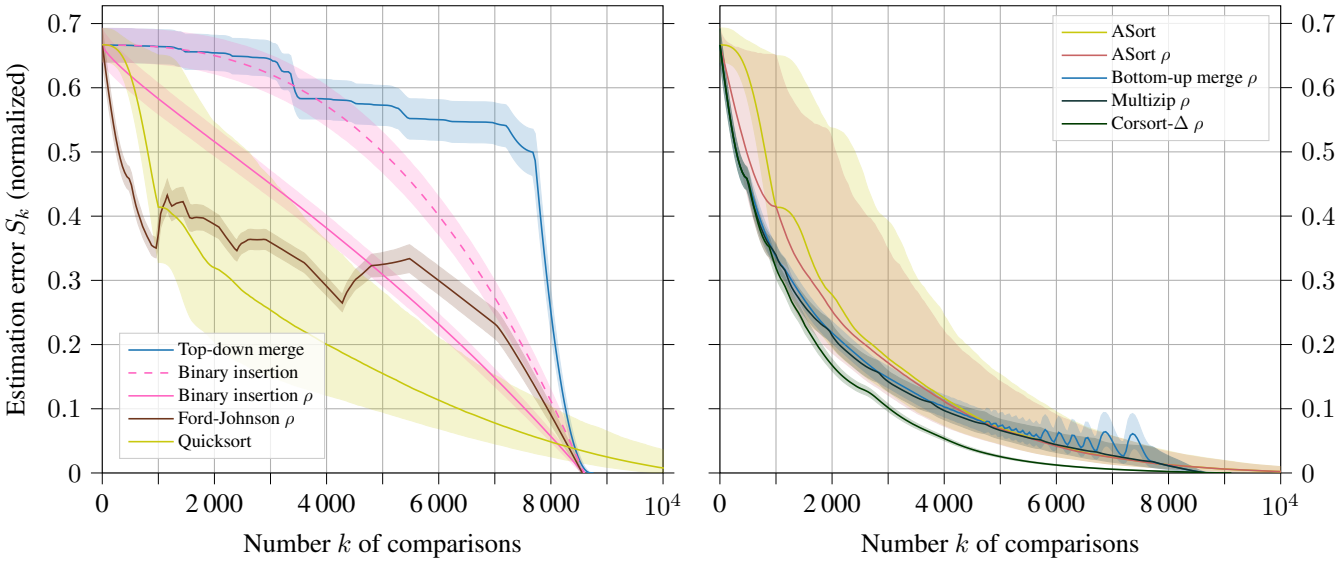
Figure 6 displays the performance profiles of the sorting algorithms. Heapsort and Shellsort were excluded due to their poor termination times (cf. Section 3.1). The Ford-Johnson algorithm employs the estimator ρ. Quicksort is examined in three variants: its natural implementation; ASort; ASort version equipped with ρ. Three variants of mergesort are also considered: the natural top-down implementation; bottom-up merge; multizip sort. The last two are both equipped with ρ. Binary insertion sort is explored with both its natural estimator and the estimator ρ. Finally, as discussed in Section 2.3, Corsort utilizes Δ for selecting the next comparison (with additional criteria in case of ties) and ρ for yielding estimates.

For each algorithm, we sort 10,000 lists of size $n = 1000$. After each specified number of comparisons k, we interrupt the algorithm and compute the estimation error S_k. As for Figure 5, we represent the median and a 95% confidence interval. The error is normalized by the maximal distance $\lfloor n^2/2 \rfloor$. The bounds from [Giesen *et al.*, 2009] are not displayed because they are irrelevant for the considered parameters: the lower bound starts from 0.005 and rapidly vanishes, while the ASort guaranteed upper bound is always above 1.

First and foremost, Corsort stands out with its remarkable performance profile. It consistently exhibits a decreasing trend with minimal variance, and outperforms other algorithms except for termination. Corsort emerges as a highly commendable anytime sorting algorithm, providing accurate estimates within a reasonable timeframe.

Multizip sort demonstrates notable performance, ranking second almost constantly in terms of the performance profile. It offers a compelling option in scenarios where global speed is of utmost importance, with a termination cost of $O(n \ln(n))$ total operations (excluding the interruption cost).

ASort also exhibits a good profile, especially if one focuses on the median values, for which it is slightly better than multizip for $k \in [5700, 7700]$. Its main drawback is a significant variance, shared with Quicksort, primarily explained



(a) Sorting algorithms not designed for anytime interruption

(b) Anytime sorting algorithms

Figure 6: Performance profiles for $n = 1000$. Each curve is obtained by sorting 10,000 random lists. The error S_k is normalized by the maximal distance $\lfloor n^2/2 \rfloor$. Results are dispatched on two subfigures for better visibility. For the computation times, cf. Appendix B.2.

by their pivot selection process. In contrast, the other sorting algorithms showcase greater consistency in their performance: binary insertion sorts, mergesorts, and Ford-Johnson exhibit relatively low variances, while Corsort demonstrates virtually negligible variance.

Corsort, multizip, and ASort are anytime by design and outperform algorithms such as binary insertion sort and Ford-Johnson. Yet, the differences between them are also significant, as illustrated by Table 2, which links the number of comparisons and the estimation error on a few values.

# of comparisons	Corsort	Multizip	ASort
4000	5.4 (5.7)	9.7 (11.4)	11.7 (24.4)
6000	1.2 (1.3)	4.4 (6.2)	4.3 (9.0)
8000	0.2 (0.2)	1.1 (2.0)	1.3 (3.4)

Table 2: Normalized error (in percent) depending on the number of comparisons for $n = 1000$. Each entry shows the median value, then the 97.5% quantile in parenthesis. Numbers are rounded to the first decimal place.

We also observe that the use of the estimator ρ contributes to an improved performance profile. Figure 6 only shows the case of binary insertion and ASort to limit the number of curves, but we actually observed the same phenomenon for all algorithms from Table 1 that have a native estimator. However, the impact of ρ is limited for ASort, where the gain is relatively small and limited to the left part of the profile.

For bottom-up merge, ρ induces a non-monotonic behavior towards the later stages of the sorting process. Indeed, ρ is optimal when the partial order is a set of independent chains, but ρ is inaccurate when there are also (non degenerated) Y-shapes, like in Figure 3. Near completion of the bottom-merge algorithm, the partial orders fluctuate between these two configurations, which leads to the observed phenomenon. For example, the final two "bumps" in Figure 6 correspond to the merge of quarters of the list. In contrast, by performing a round-robin on merge operations, our improved algorithm multizip stays monotonic and remains consistently below bottom-up merge equipped with ρ.

4 Conclusion

We have formalized the problem of anytime sorting and introduced novel approaches to address it. Our proposed estimators allow to transform any comparison-based sorting algorithm into an anytime algorithm. We have introduced the innovative algorithms of multizip sort and Corsort. Through extensive simulations, their outstanding performance has been demonstrated, showcasing their efficiency in terms of termination time and the quality of intermediate estimates. Our work advances the understanding and effectiveness of anytime sorting algorithms. In particular, despite its lack of proven upper bound and global complexity, we believe that Corsort *de facto* replaces ASort as a benchmark for empirical evaluation of future anytime sorting algorithms.

The relative novelty of our approach opens up several avenues for future research. Exploring different scoring functions could potentially enhance Corsort even further, and also improve the performance profiles of classical sorting algorithms like Ford-Johnson. Formally proving that Corsort has an $O(n \ln(n))$ comparison complexity poses another challenge. Additionally, narrowing the gap between existing theoretical bounds and empirical observations is a promising direction for future research.

Acknowledgment The work presented in this paper has been carried out at LINCS (https://www.lincs.fr/).

References

[Alewijnse *et al.*, 2014] S. P. A. Alewijnse, T. M. Bagautdinov, M. de Berg, Q. W. Bouts, A. P. ten Brink, K. Buchin, and M. A. Westenberg. Progressive Geometric Algorithms. In *Proceedings of the Thirtieth Annual Symposium on Computational Geometry*, pages 50–59, 2014.

[Brightwell and Winkler, 1991] Graham Brightwell and Peter Winkler. Counting linear extensions is #P-complete. In *Proceedings of the Twenty-Third ACM Symposium on Theory of Computing - STOC '91*, pages 175–181, 1991.

[Caizergues *et al.*, 2023] Emma Caizergues, François Durand, and Fabien Mathieu. Corsort: Comparaison ORiented Sort. https://emczg.github.io/corsort/, 2023.

[Caizergues *et al.*, 2024] Emma Caizergues, François Durand, and Fabien Mathieu. Anytime sorting algorithms. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence, IJCAI-24*, 2024.

[Calauzenes and Usunier, 2020] Clement Calauzenes and Nicolas Usunier. On ranking via sorting by estimated expected utility. In *Advances in Neural Information Processing Systems - NeurIPS '20*, volume 33, pages 3699–3710, 2020.

[Cardinal *et al.*, 2010] Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M. Jungers, and J. Ian Munro. Sorting under partial information (without the ellipsoid algorithm). In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing - STOC '10*, pages 359–368, 2010.

[Ciura, 2001] Marcin Ciura. Best increments for the average case of Shellsort. In *Fundamentals of Computation Theory: Thirteenth International Symposium*, pages 106–117, 2001.

[Cormen *et al.*, 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[Diaconis and Graham, 1977] Persi Diaconis and Ronald L Graham. Spearman's footrule as a measure of disarray. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 39(2):262–268, 1977.

[Dushkin and Milo, 2018] Eyal Dushkin and Tova Milo. Top-k sorting under partial order information. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1007–1019, 2018.

[Ford and Johnson, 1959] Lester R. Ford and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387–389, 1959.

[Giesen *et al.*, 2009] Joachim Giesen, Eva Schuberth, and Miloš Stojaković. Approximate sorting. *Fundamenta Informaticae*, 90(1–2):67–72, 2009.

[Grass and Zilberstein, 1996] Joshua Grass and Shlomo Zilberstein. Anytime algorithm development tools. *SIGART Bulletin Special Issue on Anytime Algorithms and Deliberation Scheduling*, 7(2):20–27, 1996.

[Hoare, 1961] C. A. R. Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[Horvitz, 1988] Eric J. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence - AAAI '88*, page 111–116, 1988.

[Knuth, 1998] Donald E. Knuth. *The Art of Computer Programming, Volume 3 (2nd Ed.): Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[Kuhn, 1955] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[Mesrikhani and Farshi, 2018] Amir Mesrikhani and Mohammad Farshi. Progressive sorting in the external memory model. *The CSI Journal on Computer Science and Engineering*, 15(2), 2018.

[Moran and Yehudayoff, 2016] Shay Moran and Amir Yehudayoff. A note on average-case sorting. *Order*, 33(1):23–28, 2016.

[Peczarski, 2004] Marcin Peczarski. New results in minimum-comparison sorting. *Algorithmica*, 40(2):133–145, 2004.

[Peters and Procaccia, 2021] Dominik Peters and Ariel D Procaccia. Preference elicitation as average-case sorting. In *Proceedings of the AAAI Conference on Artificial Intelligence - AAAI '21*, volume 35(6), pages 5647–5655, 2021.

[Robertson, 1997] Stephen E. Robertson. The probability ranking principle in IR. *Journal of Documentation 33*, pages 294–304, 1997.

[Zilberstein, 1995] Shlomo Zilberstein. Optimizing decision quality with contract algorithms. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence - IJCAI '95*, pages 1576–1582, 1995.

[Zilberstein, 1996] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73, 1996.

Appendix A Details of presented algorithms

A.1 Multizip sort

For the sake of clarity and conciseness, we present the pseudocode of multizip sort (Algorithm 1) in a version that does not sort the list in place. It uses two subalgorithms:

- Split (Algorithm 2) takes several lists and cuts each of them into two lists of equal sizes (up to one element). In multizip sort, it is called enough times so that all lists end up being of size 0 or 1; in particular, all of them are trivially sorted.

- Multizip merge (Algorithm 3) takes an even number of sorted lists $Y_1, \dots, Y_{2\ell}$ and merges them two by two, as in the classical bottom-up merge sort. The particularity here is that instead of completely merging Y_1 and Y_2 ("closing one zipper"), then Y_3 and Y_4 ("closing another zipper"), etc, we perform only the first comparison for each merge operation, then only the second comparison for each merge operation, etc. (metaphorically closing all zippers in parallel, little by little, in a round-robin fashion).

Algorithm 1 Multizip sort

Input: List X.
Output: List X sorted in ascending order.

$Y_1 := X$.
$\ell := 1$.
while $\exists i \in \{1, \dots, \ell\}, |Y_i| > 1$ **do**
    $Y_1, \dots, Y_{2\ell}$:= Split$(Y_1, \dots, Y_\ell)$.
    $\ell := 2\ell$.
while $\ell > 1$ **do**
    $Y_1, \dots, Y_{\ell/2}$:= Multizip merge$(Y_1, \dots, Y_\ell)$.
    $\ell := \ell/2$.
return Y_1.

Algorithm 2 Split

Input: Lists $Y_1, \dots, Y_\ell$.
Output: Lists $Z_1, \dots, Z_{2\ell}$, where for each $i \in \{1, \dots, \ell\}$, Y_i = concatenate(Z_{2i-1}, Z_{2i}), $|Z_{2i-1}| = \lceil |Y_i|/2 \rceil$, and $|Z_{2i}| = \lfloor |Y_i|/2 \rfloor$.

for $i = 1$ to ℓ **do**
    $y_i := |Y_i|$.
    $m_i := \lceil y_i/2 \rceil$.
    $Z_{2i-1} := [Y_i[1], \dots, Y_i[m_i]]$.
    $Z_{2i} := [Y_i[m_i + 1], \dots, Y_i[y_i]]$.
return $Z_1, \dots, Z_{2\ell}$.

Algorithm 3 Multizip merge

Input: An even number of lists $Y_1, \dots, Y_{2\ell}$, where each list Y_i is sorted in ascending order.
Output: Lists $Z_1, \dots, Z_\ell$, where each list Z_i consists of the elements of Y_{2i-1} and Y_{2i}, sorted in ascending order.

$Z_i := [], \ \forall i \in \{1, \dots, \ell\}$.
while $\exists i, Y_i \neq []$ **do**
    for $i = 1$ to ℓ **do**
        if $Y_{2i-1} = []$ or $Y_{2i} = []$ **then**
            Z_i = concatenate(Z_i, Y_{2i-1}, Y_{2i}).
            Empty Y_{2i-1} and Y_{2i}.
        else
            if $Y_{2i-1}[1] \leq Y_{2i}[1]$ **then**
                Z_i.append(Y_{2i-1}.pop(1)).
            else
                Z_i.append(Y_{2i}.pop(1)).
return $Z_1, \dots, Z_\ell$.

In Figure 1c, we presented the execution of multizip sort on the list (51872643), which we recall in Figure 7 for ease of reading. Here we detail this example.

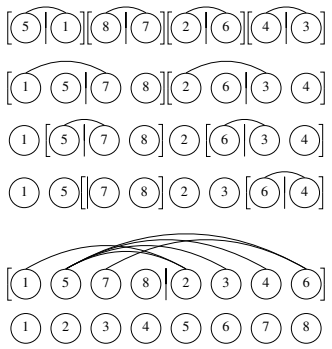


Figure 7: Sorting the list $X = (51872643)$ with multizip sort. Each edge represents a comparison. The bracket notation [|] delimits two sublists already sorted that are being merged.

After enough splitting operations, the initial list is decomposed into sublists of size 0 or 1. Since, for simplicity, we chose an example whose size is a power of two, the decomposition contains only sublists of size 1 and no empty list. Then the merging process begins (the line numbers below refer to Figure 1c or 7).

- First multizip merge:
  - Line 1: $Y_1 = [5]$, $Y_2 = [1]$, $Y_3 = [8]$, $Y_4 = [7]$, $Y_5 = [2]$, $Y_6 = [6]$, $Y_7 = [4]$, $Y_8 = [3]$. Compare $(5, 1)$, $(8, 7)$, $(2, 6)$, and $(4, 3)$, like in bottom-up merge sort.
- Second multizip merge:
  - Line 2: $Y_1 = [1, 5]$, $Y_2 = [7, 8]$, $Y_3 = [2, 6]$, $Y_4 = [3, 4]$. Compare $(1, 7)$ and $(2, 3)$.
  - Line 3: $Z_1 = [1]$, $Y_1 = [5]$, $Y_2 = [7, 8]$, $Z_2 = [2]$, $Y_3 = [6]$, $Y_4 = [3, 4]$. Compare $(5, 7)$ and $(6, 3)$.
  - Line 4: $Z_1 = [1, 5]$, $Y_1 = []$, $Y_2 = [7, 8]$, $Z_2 = [2, 3]$, $Y_3 = [6]$, $Y_4 = [4]$. For Z_1, no more comparaison is

needed because Y_1 is empty. Compare $(6, 4)$ to finish merging Y_3 and Y_4 into Z_2.

- Third multizip merge:
  - Line 5: $Y_1 = [1, 5, 7, 8]$, $Y_2 = [2, 3, 4, 6]$. Compare $(1, 2)$, $(5, 2)$, $(5, 3)$, $(5, 4)$, $(5, 6)$, and $(7, 6)$, like in top-down or bottom-up merge sort.

Note that in multizip sort, the first multizip merge is always identical to the first round of merging operations in bottom-up merge sort. Indeed, each sublist is of size 0 or 1, hence each merging operation involves at most one comparison, which implies that there is no difference between treating each merging operation as a block and performing the necessary comparisons in a round-robin fashion. On the other hand, the last merging operation (between the two sorted half-lists) is always the same for the three variants: top-down merge sort, bottom-up merge sort, and multizip sort. Hence to show the difference between bottom-up merge sort and multizip sort, it is necessary to have an example with three levels of merging, such as the one we present.

A.2 Partial orders and estimators

Algorithm 4 (Partial order update) performs the update of a matrix M representing a partial order, when we acquire the information of a new comparison $i \prec j$. In the paper, it is used in two contexts.

- Corsort calls Algorithm 4 after each comparison and uses the updated partial order to choose the next pair to compare.
- The anytime adaptation of any sorting algorithm, defined in Section 2.2, builds the partial order only in case of interruption, by iterating Algorithm 4 for every comparison in the history.

Algorithm 4 Partial order update

Input: A matrix M such that $M_{k,\ell} = +1$ if $k \preceq \ell$, $M_{k,\ell} = -1$ if $k \succ \ell$, and $M_{k,\ell} = 0$ if we have not compared k and ℓ yet. A pair (i, j).
Output: The transitive closure of M with the additional comparison $i \preceq j$.

```
descendants_i := {k, M_{k,i} = +1}.
ascendants_j := {ℓ, M_{j,ℓ} = +1}.
for k ∈ descendants_i do
    for ℓ ∈ ascendants_j do
        M_{k,ℓ} := +1.
        M_{ℓ,k} := −1.
return M.
```

Let M be defined with the convention of Algorithm 4 and i be the index of an element. To compute the number of ascendants $a(i)$ or descendants $d(i)$, it suffices to read the row or column i of matrix M:

$$a(i) = \sum_{j=1}^{n} \mathbb{1}[M_{i,j} = +1],$$

$$d(i) = \sum_{j=1}^{n} \mathbb{1}[M_{j,i} = +1].$$

The computation of $I(i)$, $\Delta(i)$, and $\rho(i)$ is straightforward:

$$I(i) = d(i) + a(i),$$
$$\Delta(i) = d(i) - a(i),$$
$$\rho(i) = \frac{d(i)}{d(i) + a(i)}.$$

A.3 Corsort

Algorithm 5 describes Corsort-Δ ρ, which uses Δ to choose the next comparison to perform and ρ to estimate the sorted list. Some variants can be obtained by replacing Δ and ρ with other estimators. A detailed description of Corsort is available in Section 2.3.

Algorithm 5 Corsort-Δ ρ

Input: List X of size n.
Output: List X sorted in ascending order if the algorithm ends with no interruption. An estimate of the sorted list otherwise.

```
M := the identity matrix of size n, representing the empty partial order on X.
while there is no interruption and ∃(i,j), M_{i,j} = 0 do
    Update Δ and I according to M.
    S := {(i, j), M_{i,j} = 0}.
    (i, j) := arg min_S (|Δ(i) − Δ(j)|, max(I(i), I(j))).
    if X[i] > X[j] then
        (i, j) := (j, i).
    M := Partial order update(M, (i, j)).
Update ρ according to M.
return X sorted by ascending value of ρ.
```

Appendix B Implementation

B.1 The Corsort package

The Python code of this article has been published on GitHub[3] and PyPI[4] under GNU General Public license v3 [Caizergues *et al.*, 2023]. The documentation[5] provides detailed instructions for package installation and for the replication of our results.

In particular, the *Notebooks* section of the documentation gives step-by-step intructions to reproduce all the experiments we describe.

- The *Termination times*[6] notebook explains the computation of Figure 5.
- The *Comparison of Corsort variants*[7] notebook explains our selection of the Corsort-Δ ρ variant.
- The *Impact of estimators*[8] notebook compares natural estimators (if they exist) and the external estimators Δ and ρ.

[3] https://github.com/emczg/corsort
[4] https://pypi.org/project/corsort/
[5] https://emczg.github.io/corsort/
[6] https://emczg.github.io/corsort/notebooks/termination.html
[7] https://emczg.github.io/corsort/notebooks/corsorts.html
[8] https://emczg.github.io/corsort/notebooks/rho_delta.html

- The *Performance profiles*[9] notebook explains the computation of Figure 6.
- The *Examples with a chain and a Y-shape*[10] notebook gives toy examples that illustrate the behavior of estimators. In particular, it explains how the example of Figure 3 was built.

B.2 Running time

For our benchmark of Figure 6, we sort $10{,}000$ random lists of size $1{,}000$. We choose this size of list for two reasons. First, our framework addresses applications where the comparisons are expensive, for example because they are manually performed by human experts, so the typical use-case does not concern lists whose size is orders of magnitude higher. Secondly, since this benchmark consists in interrupting the algorithms at each step to check the current Spearman distance to the ground truth, it slows down the process; without this artificial additional delay, it is realistic to use the algorithms with larger lists.

In the table below, we give the running time for each algorithm. The hardware used was a AMD Ryzen Threadripper 1950X (32 cores). Note that actual running time (i.e. in seconds or minutes) is not the primary focus of the paper.

Algorithm	Time
Top-down merge	10 min 03 s
Binary insertion	8 min 21 s
Binary insertion ρ	21 min 02 s
Ford-Johnson ρ	24 min 52 s
Quicksort	10 min 13 s
ASort	10 min 33 s
ASort ρ	39 min 56 s
Bottom-up merge ρ	16 min 39 s
Multizip ρ	16 min 42 s
Corsort-Δ ρ	78 min 35 s

[9] https://emczg.github.io/corsort/notebooks/profiles.html

[10] https://emczg.github.io/corsort/notebooks/examples_with_a_chain_and_a_y_shape.html