Anytime Sorting Algorithms

Emma Caizergues\textsuperscript{1,2}, François Durand\textsuperscript{1}, Fabien Mathieu\textsuperscript{3}

\textsuperscript{1}Nokia Bell Labs, Massy, France
\textsuperscript{2}CNRS - Lamsade, Université Paris Dauphine - PSL, Paris, France
\textsuperscript{3}Swapcard Lab, Paris, France
emma.caizergues@nokia.com, francois.durand@nokia.com, fabien@swapcard.com

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 [Alewine 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 $n^2 e^{-k/6n^6} + 1$. While these results show that ASort is asymptotically optimal in some sense, the ratio between the two bounds above ($e^{5k/6n^6} + 1$) 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 merge-sort 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.

Extended version. A version of the paper completed by [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 the Spearman’s footrule metric between $X_k$ and the
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparison complexity</th>
<th>Global complexity</th>
<th>Native estimator</th>
<th>Interest(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford-Johnson</td>
<td>(\sim n \log_2(n))</td>
<td>(O(n^2))</td>
<td>No</td>
<td>Comparison complexity</td>
</tr>
<tr>
<td>Binary insertion</td>
<td>(\sim n \log_2(n))</td>
<td>(O(n^2))</td>
<td>Yes</td>
<td>Comparison complexity</td>
</tr>
<tr>
<td>Quicksort</td>
<td>(O(n \ln(n)))</td>
<td>(O(n \ln(n)))</td>
<td>Yes</td>
<td>Fast termination in practice</td>
</tr>
<tr>
<td>ASort</td>
<td>(O(n \ln(n)))</td>
<td>(O(n \ln(n)))</td>
<td>Yes</td>
<td>Good native estimator with theoretical guarantees</td>
</tr>
<tr>
<td>Top-down merge</td>
<td>(\sim n \log_2(n))</td>
<td>(O(n \ln(n)))</td>
<td>Yes</td>
<td>Complexities</td>
</tr>
<tr>
<td>Bottom-up merge</td>
<td>(\sim n \log_2(n))</td>
<td>(O(n \ln(n)))</td>
<td>Yes</td>
<td>Good performance profile, complexities</td>
</tr>
<tr>
<td>Multizip</td>
<td>(\sim n \log_2(n))</td>
<td>(O(n \ln(n)))</td>
<td>Yes</td>
<td>Very good performance profile, complexities</td>
</tr>
<tr>
<td>Consort</td>
<td>(O(n^2)^{\ast})</td>
<td>(O(n^4)^{\ast})</td>
<td>No</td>
<td>Best performance profile, comparison complexity^{\ast}</td>
</tr>
</tbody>
</table>

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 \(\rho\) when interrupted. Using \(\rho\) requires \(O(n \ln(n))\) for Consort, \(O(n^4)\) operations for the other algorithms. Native estimators, when they exist, can be used directly.

\(^{\ast}\) Simulations suggest \(O(n \ln(n))\) comparisons, yielding a global complexity of \(O(n^3 \ln(n))\) operations.

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 in the extended paper.

sorted list [Diaconis and Graham, 1977], i.e. by the sum of the absolute differences between the ranks of elements in \(X_k\) and in the sorted list: \(S_k = \sum |r(X[k]) - i|\), where \(r(x)\) denotes the rank of \(x\) in the sorted list. The function \(k \rightarrow S_k\), 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 in the extended paper 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.

\(^{1}\)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 \((abcd)\).
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 a subroutine for median identification. 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 already in its final position.

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. 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.

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

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 \( \leq_k \) 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 [Calauzènes 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, \( d_k(x) = \{|y \in X : y \leq_k x\} \) and \( a_k(x) = \{|y \in X : x \leq_k y\} \) are respectively the number of known descendants and ancestors of \( x \) in \( \leq_k \), and \( x \) 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 Floyd-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 ancestors: 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 in the extended paper 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:

- \( \Delta_k \), defined by \( \Delta_k(x) = d_k(x) - a_k(x) \). \( \Delta_k \) assigns to each \( x \) a score that reflects the average between its lowest and highest possible positions.
- \( \rho_k \), defined by \( \rho_k(x) = \Delta_k(x)/(d_k(x) + a_k(x)) \). \( \rho_k \) 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 \( k \). \( \Delta, \rho \), 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, \( \rho \) is better than \( \Delta \) because it pos-
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 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-\( \Delta \):

- The next comparison is made by seeking a pair of incomparable items whose \( \Delta \) scores are as close as possible.
- We use \( I_k \) for tie-breaking. Specifically, among the pairs that minimize \( \Delta \)-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 \( \rho \). For the pseudocode, cf. Appendix A.3 in the extended paper. As an example, Figure 4 shows the execution of the selected Corsort on the list \( X = (42315) \). As we argued earlier that \( \rho \) is better than \( \Delta \), one may wonder why \( \Delta \) is preferred for the choice of the comparison. An intuition that explains the advantage of \( \Delta \) is the need of balance between termination and intermediate estimates: \( \rho \) can take \( O(n^2) \) distinct values, against \( O(n) \) for \( \Delta \). As a result, \( \Delta \) generates more ties than \( \rho \) (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 \( \rho \)-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 \( O(n^3) \). 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 in the extended paper).
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 represent different scheduling variations of mergesort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort. The last two are both equipped with a binary insertion sort.
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 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.

<table>
<thead>
<tr>
<th># of comparisons</th>
<th>Corsort</th>
<th>Multizip</th>
<th>ASort</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>5.4 (5.7)</td>
<td>9.7 (11.4)</td>
<td>11.7 (24.4)</td>
</tr>
<tr>
<td>6000</td>
<td>1.2 (1.3)</td>
<td>4.4 (6.2)</td>
<td>4.3 (9.0)</td>
</tr>
<tr>
<td>8000</td>
<td>0.2 (0.2)</td>
<td>1.1 (2.0)</td>
<td>1.3 (3.4)</td>
</tr>
</tbody>
</table>

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 $\rho$ 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 $\rho$ is limited for ASort, where the gain is relatively small and limited to the left part of the profile.

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.

Acknowledgments

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


