Longest Increasing Subsequence under Persistent Comparison Errors

Barbara Geissmann

Abstract
We study the problem of computing a longest increasing subsequence in a sequence $S$ of $n$ distinct elements in the presence of persistent comparison errors. In this model, Braverman and Mossel (Noisy sorting without resampling, SODA 2008, pages 268–276, 2008) every comparison between two elements can return the wrong result with some fixed (small) probability $p$, and comparisons cannot be repeated. Computing the longest increasing subsequence exactly is impossible in this model, therefore, the objective is to identify a subsequence that (i) is indeed increasing and (ii) has a length that approximates the length of the longest increasing subsequence. We present asymptotically tight upper and lower bounds on both the approximation factor and the running time. In particular, we present an algorithm that computes an $O(\log n)$-approximation in $O(n \log n)$ time, with high probability. This approximation relies on the fact that we can approximately sort (Geissmann et al. Optimal Sorting with Persistent Comparison Errors, ArXiv e-prints 1804.07575, 2018) $n$ elements in $O(n \log n)$ time such that the maximum dislocation of an element is $O(\log n)$. For the lower bounds, we prove that (i) there is a set of sequences, such that on a sequence picked randomly from this set every algorithm must return an $\Omega(\log n)$-approximation with high probability, and (ii) any $\log n$-approximation algorithm for longest increasing subsequence requires $\Omega(n \log n)$ comparisons, even in the absence of errors.

Keywords Longest increasing subsequence · Probabilistic persistent comparison errors · Approximation algorithm · Lower bounds
1 Introduction

When dealing with complex systems and large volumes of information, it is often the case that at least part of the involved data will be inconsistent. These inconsistencies can be intrinsic, i.e., they might stem from the fact that the data is obtained from an inherently noisy source (this is typically the case in human-produced data), or they might be the result of corruptions caused by random errors (think, for instance, of random memory faults or communication errors). Indeed, computations in practical architectures are error-prone [8], and even basic algorithmic operations – for instance comparisons – can sometimes return wrong results. A first approach to cope with such bogus results, is trying to detect and correct errors at their occurrence, such that classical algorithms which (only) work in an error-free environment can be used to solve the problems of interest. An alternative to this approach is one that deliberately allows errors to interfere with the execution of an algorithm, with the intent that the computed solution will still be good, at least in an approximate sense. This second approach raises questions like:

Is it possible to design algorithms that cope with errors and return provably good solutions?

On the road to answer this and similar questions, it is important to understand how the classical techniques used to solve basic algorithmic problems can cope with such errors.

There exist many computational problems whose solutions rely on the ability to perform correct comparisons between elements and we consider a model where independent random comparison errors occur, i.e., we assume that every comparison between a pair of two elements fails independently with a small fixed error probability $p < \frac{1}{2}$. To be precise, we assume that a comparison returns the correct order of the elements with probability $1 - p$, and the wrong (i.e., reverse) order of the elements with probability $p$. Such random comparison errors can manifest in two different forms, namely they can be persistent or non-persistent. If comparison errors are persistent, then repeating the comparison between the same two elements multiple times always yields the same result, and therefore the comparison is either always correct or it fails all the time. Note that this is equivalent to saying that comparisons cannot be repeated. On the contrary, if comparison errors are non-persistent, then repeating a comparison several times can yield different results. This makes coping with non-persistent errors considerably easier than with persistent ones, as one can repeat every comparison a sufficiently large number of times and then take the majority result to find out the correct answer with arbitrarily high confidence. In this paper, we focus on computing under persistent comparison errors. This model has been first employed by Braverman and Mossel [6], who studied the problem of sorting. Other work on sorting followed (see [15, 16, 22]) and the model has been studied also for finding the minimum, searching, and linear programming in two dimensions [22]. Further discussion of results in this context is provided in Section 1.1.

The problem of our interest is the problem of computing a longest increasing subsequence in a given input sequence $S$ of distinct elements (hereafter, we refer to such a subsequence as $LIS(S)$, and to its length as $|LIS(S)|$) in the presence of
independent random persistent comparison errors. Computing $LIS(S)$ is a fundamental task that appears naturally in many areas, such as in probability theory and combinatorics [2, 4] or computational biology [11, 30]. Because of the comparison errors, it is impossible in our model to compute $LIS(S)$ correctly. Instead, we seek to return a subsequence that

(i) is indeed increasing and that
(ii) has some guaranteed minimum length in relation to $|LIS(S)|$.

In particular, we are interested in algorithms that return an increasing subsequence of length at least $\frac{1}{r} \cdot |LIS(S)|$, where $r$ is the approximation factor.

We will present an algorithm that returns an $O(\log n)$-approximation on the longest increasing subsequence in $O(n \log n)$ time, with high probability. Moreover, we will prove that this approximation factor is the best possible as $\Omega(\log n)$ is also a lower bound, regardless of the running time, and that any $\log n$-approximation algorithm requires $\Omega(n \log n)$ comparisons, even in the absence of comparison errors. We refer to Section 1.2 for a description of these results in greater detail as well as an overview of the ideas and techniques used in this paper.

1.1 Related Work

There exist several algorithms to compute a longest increasing subsequence if no comparison errors happen. Typically, they are based on a common underlying algorithmic idea and only differ in implementation and whether they return the actual subsequence or only the length: all the algorithms process the elements one by one from left to right and maintain for each length the increasing subsequence of this length that ends with the smallest possible element seen so far. We shall call this algorithmic idea the Core-Algorithm to compute a longest increasing subsequence. The running time of this Core-Algorithm is $O(n \log n)$ (see for instance [5, 7, 13]), in the decision-tree model. This time complexity is tight, as shown in [13]. In RAM model, where one can also inspect the values, the algorithm can be implemented to run in $O(n \log \log n)$ time [9, 29], when using search trees of van Emde Boas [27, 28]. All the results can be parametrized to $O(n \log l)$ or $O(n \log \log l)$, respectively, where $l$ is the length of the longest increasing subsequence.

The longest increasing subsequence of $S$ is also the longest common subsequence between $S$ and the sorted sequence of the elements in $S$. This implies an $O(n^2)$ time (or $O(n^2 / \log n)$ time if optimized) algorithm to find the longest increasing subsequence when using the standard dynamic programming technique that is used to find longest common subsequences [13, 25]. The problem of finding a longest increasing subsequence has also been studied in the streaming model, see for instance [23, 26].

Computing under random persistent comparison errors has been extensively studied for finding the smallest element, for searching, and for sorting (see for instance [6, 15, 16, 22]). A common way to measure the quality of an output sequence in terms of sortedness, is to consider the dislocation of the elements. The dislocation of an element is the absolute difference between its position in the output sequence and its position in the correctly sorted sequence (its rank). Typically, one considers the maximum dislocation of any element in the output sequence and the total dislocation (the
sum of the dislocations of all elements). It has been shown for instance in [17], that there is an algorithm with running time $O(n \log n)$ which achieves simultaneously maximum dislocation $O(\log n)$ and total dislocation $O(n)$ with high probability, and that this is indeed the best one can hope for (i.e., there exist matching lower bounds that show that no possibly randomized algorithm can sort such that, with high probability, the maximum dislocation is $o(\log n)$ or the total dislocation is $o(n)$). A maximum dislocation of $O(\log n)$ implies the following: on the positive side, it is possible to derive the correct relative order of two elements whose ranks differ by at least $\Omega(\log n)$; on the negative side, this is not possible for two elements whose ranks differ by less than $O(\log n)$. The results on the maximum dislocation of sorting are also of interest for the problem of finding the longest increasing subsequence: although computing a longest increasing subsequence is different to sorting, it is still closely related, simply because an increasing subsequence is sorted. Indeed, we will see that our approximation algorithm will use the sorting algorithm of [17] as a black box in order to compute a long increasing sequence.

The sorting algorithm in [17] has one limitation: it has been proven to work only for error probability $p < 1/16$, although experiments in [15] suggest a higher bound close to 1/2. This limitation does not exist for the algorithm in [6], which finds for any $p < 1/2$ a maximum likelihood order of the elements, such that the maximum dislocation is $O(\log n)$ with high probability. However, this comes at the cost of a significantly larger running time $O(n^{3+24c})$, where the constant $c$ depends on both the error probability $p$, and the success probability of the algorithm. For example, for a success probability of $1 - 1/n$, the analysis in [6] yields $c = \frac{110525}{(1/2-p)^4}$. It is indeed an open question whether any algorithm can achieve both optimal bounds of $O(n \log n)$ running time and $O(\log n)$ maximum dislocation for every $p < 1/2$.

In the easier variant with non-persistent comparison errors, one can actually sort in $O(n \log(n/q))$ time, where $1 - q$ is the success probability of the algorithm (see for instance [12]). The impact of such errors on classical sorting algorithms such as Insertionsort, Quicksort, and Mergesort has been analyzed in [3, 19–21]. Other models restrict the comparisons in which errors can happen. For instance, [1] gives a sorting algorithm when errors occur only between elements whose difference is at most some fixed threshold, and [10] provides an algorithm when the total number of errors is known in advance.

### 1.2 Our Contribution

We prove asymptotically tight upper and lower bounds on both the approximation factor and the running time for longest increasing subsequence under persistent comparison errors. For the upper bounds, we define an Approximation-Algorithm that computes an $O(\log n)$-approximation to the longest increasing subsequence of $S$. In fact, it even finds the longest possible increasing subsequence under the implication that we cannot sort better than obtaining an order with maximum dislocation $O(\log n)$. Formally, we prove the following result:

**Theorem 1** (Upper Bounds) *For any input sequence $S$ that contains $n$ distinct elements and any error probability $p < 1/16$, our Approximation-Algorithm computes*
an \(O(\log n)\)-approximation to the longest increasing sequence of \(S\), in \(O(n \log n)\) time, with probability at least \(1 - \frac{1}{n}\).

This result on the upper bound can be generalized to other error models. In fact, if we are given or able to obtain an approximately sorted sequence with maximum dislocation \(d\), then our Approximation-Algorithm will return a \(2d\)-approximation to the longest increasing subsequence. We discuss this point in Section 6.

To prove our lower bound on the approximation factor of any algorithm solving \(LIS(S)\) under persistent comparison errors with high probability, we will identify a small collection of similar sequences that all contain a longest increasing sequence of size \(\Theta(\log n)\). The similarity is such that two sequences are likely to be indistinguishable by any algorithm that observes the comparison results of the elements. Indeed, we show for any algorithm that if it succeeds on one sequence of this collection by returning a constant number of elements of this increasing sequence it must fail on another sequence. In particular, we will prove the following theorem:

**Theorem 2** (Lower Bound – Approximation Factor) There exists a collection of sequences \(S\) (permutations of length \(n\)), a probability distribution on \(S\), and a constant \(c\) depending on \(p\), such that no algorithm can return an \(c \log n\)-approximation of the longest increasing subsequence with probability \(1 - \frac{1}{n}\).

We prove a lower bound of \(\Omega(n \log n)\) on the number of comparisons (which is also a lower bound on the running time) needed to compute an \(\log n\)-approximation by considering the easier case in which all comparisons are correct, and by adapting the techniques used in [13] for proving a similar lower bound for exact (i.e., 1-approximate) algorithms:

**Theorem 3** (Lower Bound – Running Time) Any \(\log n\)-approximation algorithm for longest increasing subsequence requires \(\Omega(n \log n)\) comparisons, even if no errors occur.

### 2 Preliminaries

We assume that we are given an input sequence \(S = \langle s_1, s_2, \ldots, s_n \rangle\) containing \(n\) elements with pairwise distinct values that possess a linear order \(S^{\text{sort}}\). For easier notation we let the sequence implicitly also define the set of its elements, such that we can assume that functions usually defined for sets work on sequences as expected. We also assume, for easier analysis and readability, that \(S\) is a permutation of the integer numbers (elements) \(\{1, \ldots, n\}\). For two distinct elements \(x\) and \(y\) in \(S\), we write \(x < y\) to denote that \(x\) is smaller than \(y\) according to their linear order (resp. \(x > y\) to denote that \(x\) is larger than \(y\)).

In the error model that we consider, we assume that our algorithms can only obtain ordinal information about the elements in the input sequence, but cannot inspect their actual values. In particular, we assume that we can perform pairwise element comparisons to gain information about the linear order of these elements, however,
these comparisons are prone to persistent errors: with probability $1 - p$, the result of the comparison is always correct, and with probability $p$, the result of the comparison is always wrong, where $p < \frac{1}{2}$ is a small fixed constant. We will thus write $x < y$ (resp. $x > y$) to mean that $x$ is observed to be smaller than (resp. larger than) $y$ in the corresponding comparison.

For a given sequence $S$ and an element $x \in S$, we define $\text{pos}(x, S) \in [1, |S|]$ to be the position of $x$ in $S$ (note that positions start from 1), and $\text{pos}(x, S^\text{sort})$ to be the rank of element $x$ in $S$. The dislocation of $x$ in $S$ is then $\text{disl}(x, S) = |\text{pos}(x, S) - \text{pos}(x, S^\text{sort})|$, and the maximum dislocation of $S$ is $\text{disl}(S) = \max_{x \in S} \text{disl}(x, S)$.

For a given sequence $S$, we let $C \in \{<, >\}^\binom{n}{2}$ denote the comparison outcomes that can be observed. For $C = \langle c_{1,2}, \ldots, c_{i,j}, \ldots, c_{n-1,n} \rangle$ with $1 \leq i < j \leq n$, this means that $c_{i,j} = "<"$ if $s_i$ is reported to be smaller than $s_j$ when comparing $s_i$ and $s_j$ and $c_{i,j} = ">"$ otherwise.

We continue the preliminaries with some results on sorting that we will use to prove our upper bound on the approximation factor.

**Theorem 4** (Theorem 3 in [17]) There is an algorithm that approximately sorts, in $O(n \log n)$ worst-case time, $n$ elements subject to random persistent comparison errors with $p < 1/16$ so that the maximum dislocation of the resulting sequence is $c \log n$, for some constant $c$ depending on $p$, with probability $1 - \frac{1}{n}$.

**Lemma 1** Let $S^\text{apx} = \langle s_1, s_2, \ldots, s_n \rangle$. If the maximum dislocation in $S^\text{apx}$ is at most $d$, then for all $1 \leq i < n - 2d$, $s_i$ and $s_{i+2d}$ are in linear order: $\text{pos}(s_i, S^\text{sort}) < \text{pos}(s_{i+2d}, S^\text{sort})$.

**Proof** Since $\text{disl}(S^\text{apx}) \leq d$, $\text{pos}(s_i, S^\text{sort}) \in \{i - d, \ldots, i + d\}$ and $\text{pos}(s_{i+2d}, S^\text{sort}) \in \{i + d, \ldots, i + 3d\}$. These intervals intersect in at most one position, and the claim follows since no two elements can appear in the same position. □

## 3 Upper Bound and Approximation-Algorithm

We will modify the so-called Core-Algorithm (as named in Section 1.1, Related work) that computes a longest increasing subsequence in the absence of comparison errors, such that it computes an $O(\log n)$-approximation with high probability in our error model. Before we do so, we first show that it is possible to identify a $2d$-approximation by looking at $S$ and a sequence $S^\text{apx}$ with maximum dislocation $d$. Since we can sort such that the maximum dislocation is $O(\log n)$ with high probability (see Theorem 4), this implies an $O(\log n)$-approximation on $LIS(S)$.

### 3.1 Upper Bound

The proof of the upper bound is based on the following fact and observation:
Without any comparison errors, the problem of finding $LIS(S)$ is equivalent to the problem of finding a longest common subsequence between $S$ and $S^{sort}$, where $S^{sort}$ is the correctly sorted order of the elements in $S$.

This leads to the following observation. Let $S^{apx}$ be the sequence obtained from approximately sorting $S$ with comparison errors and consider now $S^{apx}$ as the total order over all elements, i.e., for each pair of elements, their comparison result is redefined as their relative order in $S^{apx}$. Furthermore, let $A$ be any algorithm that solves $LIS(S)$ in the absence of errors. If $A$ uses the redefined comparison results, it computes the longest common subsequence $LCS(S, S^{apx})$ between $S$ and $S^{apx}$.

The immediate idea of computing $LCS(S, S^{apx})$ comprises some difficulties, since this subsequence is not necessarily increasing and, on top of that, its length might be smaller than $|LIS(S)|$. However, we can still get a first approximation. Assume that $S^{apx}$ has maximum dislocation at most $d$. Lemma 1 implies that we obtain an increasing subsequence when taking every $2d$-th element of $LCS(S, S^{apx})$. And the maximum dislocation implies that the elements in the subset containing every $2d$-th element of $LIS(S)$ appear in the same relative order in $S^{apx}$, thus $|LCS(S, S^{apx})| \geq \frac{1}{2d}|LIS(S)|$. When put together, we get a $4d^2$-approximation.

This approximation factor can be improved, and it turns out that considering common subsequences whose elements lie (at least) $2d$ positions apart in $S^{apx}$ is actually a good start: By Lemma 1, a common subsequence between $S$ and $S^{apx}$ is increasing if for every pair of adjacent elements in this subsequence their positions in $S^{apx}$ differ by at least $2d$. Therefore, we say that a sequence $S' = (s'_1, s'_2, \ldots, s'_m)$ is $2d$-distant in $S^{apx}$ if

$$pos(s'_i, S^{apx}) + 2d \leq pos(s'_{i+1}, S^{apx}) \quad \text{for } 1 \leq i < m.$$  

(1)

Based on this notion of $2d$-distant subsequences, we prove the following lemma, whose proof can be seen as a general recipe to obtain a $2d$-approximation on longest increasing subsequence.

**Lemma 2** The longest subsequence $S^*$ of $S$ that is $2d$-distant in $S^{apx}$ has length at least

$$|S^*| \geq \frac{1}{2d}|LIS(S)|.$$  

**Proof** Notice that any (increasing) subsequence of $S$ that is $2d$-distant in $S^{apx}$ is automatically also a common (increasing) subsequence of $S$ and $S^{apx}$. This observation suggests the following easy recipe to obtain a $2d$-approximation:

- First, partition the elements of $S^{apx}$ into $2d$ subsets, such that every $2d$-th element gets into the same subset, i.e., let $S^{apx} = (s_1, s_2, \ldots, s_n)$, then the $i$-th subset contains the elements $(s_i, s_{i+2d}, s_{i+4d}, \ldots)$. Thus, we obtain $2d$ new input subsequences based on this partition.
- Then, on every new input subsequence, run any algorithm that computes a longest increasing subsequence if no comparison errors happen, and return the longest result.
By pigeon hole principle and since every input subsequence is now $2d$-distant in $S_{apx}$, the longest result must be a $2d$-approximation on $|LIS(S)|$. The lemma follows immediately, since $S^*$ is at least as long as the best approximation obtained by the recipe.

Note that the above recipe is not optimal in the sense that in many cases, we could do better and find a longer subsequence in $S$ that is still $2d$-distant in $S_{apx}$. In fact, we lose up to a factor $2d$ in the case where $LIS(S)$ is already $2d$-distant in $S_{apx}$, but these elements are equally distributed among all input subsequences. For this reason, we will define an approximation algorithm that finds the longest increasing subsequence in $S$ that is $2d$-distant in $S_{apx}$.

### 3.2 Approximation-Algorithm

Consider the Core-Algorithm described in Algorithm 1 that computes the longest increasing subsequence of the input sequence $S$ in the error-free case. The algorithm processes the input elements one by one, maintaining the longest increasing subsequence found so far. In particular, it maintains a parameter $k$ and an array $L$, such that $k$ is the length of the longest increasing subsequence found so far and $L$ contains an entry for each length 1 to $k$, such that $L[i]$ stores the smallest element processed so far that can be at the end of an increasing subsequence of length $i$.

- The first element is placed to $L[1]$ and $k$ is set to 1.
- Each subsequent element $x$ is placed to $L[j + 1]$, such that $j$ is the largest position where $y = L[j]$ is smaller than $x$.
- If $x$ is placed to $L[k + 1]$, then $k$ is updated to $k + 1$.

---

Algorithm 1 Core-Algorithm($S = \langle s_1, \ldots, s_n \rangle$).

```plaintext
1 $L[1] \leftarrow s_1$;
2 $k \leftarrow 1$;
3 foreach $i = 2, \ldots, n$ do
4     $x \leftarrow s_i$;
5     if $x < L[1]$ then $L[1] \leftarrow x$;
6     else $j \leftarrow \max\{j \leq k : L[j] < x\}$;
7     if $j = k$ then $k \leftarrow k + 1$;
8     $L[j + 1] \leftarrow x$;
9     $prec[x] \leftarrow L[j]$;
10 $lis[1] \leftarrow L[k]$;
11 foreach $i = 2, \ldots, k$ do
12     $lis[i] \leftarrow prec[lis[i - 1]]$
13 return $lis$;
```
Whenever a new element $x$ is placed, put a pointer $prec$ from $x$ to the element in $y = L[j]$, that, by construction, has a lower value than $x$.

In the end, follow these pointers from the top element of the last pile to recover the longest increasing subsequence (in reverse order).

An entry $L[j]$ basically represents the increasing sequence of length $j$ that ends with the smallest possible element processed so far. When an element $x$ is inserted into some position $L[j + 1]$ this means that it is appended to the sequence represented by $L[j]$. Hence, $x$ either increases the longest increasing sequence so far (case $j = k$) or the sequence $L[j + 1]$ gets replaced by this new sequence (case $x < L[j + 1]$).

Our Approximation-Algorithm, as described in Algorithm 2, is obtained by modifying the Core-Algorithm such that it works in our error model.

We first approximately sort (using the algorithm from [17], see also Theorem 4 in the current paper) the elements of $S$ to obtain $S^{apx}$, and we redefine the comparison outcomes based on this total order, i.e., the result of a comparison between two elements now corresponds to their relative order in $S^{apx}$.

To compute a suitable subsequence, we change the algorithm so that it remembers the longest $2d$-distant in $S^{apx}$ subsequences instead of the longest increasing subsequences. This implies that an element $x$ is only appended to an (intermediate) subsequence that ends with element $y$ if $\text{pos}(y, S^{apx}) + 2d \leq \text{pos}(x, S^{apx})$.

For easier analysis, we introduce some additional notation. We call one execution of the lines 5 to 13 of Algorithm 2 an iteration, and number them such that element $s_i$ is considered in iteration $i$. We also say that line 4 corresponds to the first iteration. Furthermore, we denote by $L_t$ and $k_t$ the state and the value of $L$ and $k$ after the $t$-th iteration, respectively, and for any $j \leq k_t$, we call the subsequence $\langle L_t[j], prec[L_t[j]], prec[prec[L_t[j]]], \ldots \rangle$ with length $j$ the implied sequence of $L_t[j]$.

**Lemma 3** For every $t \leq n$, after the $t$-th iteration of our Approximation-Algorithm, every implied sequence is a subsequence of $S$ that is $2d$-distant in $S^{apx}$. Moreover, $\langle L_t[1], \ldots, L_t[k_t] \rangle$ is also $2d$-distant in $S^{apx}$.

**Proof** For any $t$ and $j \leq k_t$, let $S' = \langle s'_1, \ldots, s'_j \rangle$ be the implied sequence of $L_t[j]$. Observe that to every element $s'_i \in S'$, such that $i > 1$, the algorithm has assigned $s'_{i-1}$ as its predecessor. Since the predecessor of any element can only have been processed in an earlier iteration, $S'$ is a subsequence of $S$.

It follows by induction, that the condition on line 10 in Algorithm 2 ensures that $S'$ is $2d$-distant in $S^{apx}$: It is trivial to see for $t = 1$, thus, assume that every implied sequence before the $t$-th iteration is $2d$-distant in $S^{apx}$. If $s_t$ is inserted into $L[j]$ (nothing changes in the opposite case), the implied sequence of $L_t[j]$ is equal to $s_t$ appended to the implied sequence of $L_{t-1}[j - 1]$ (if it exists). By hypothesis and the condition on line 10, $L_t[j]$ is still $2d$-distant,
and since the other implied sequences do not change, the claim also holds after iteration $t$.

That $\langle L_t[1], \ldots, L_t[k_t] \rangle$ is $2d$-distant in $S_{apx}$ also follows by induction: If $L[j]$ changes (thus $L[j']$ does not change for all $j' \neq j$), then by hypothesis and the conditions in lines 7, 9, and 10, $\text{pos}(L_{t-1}[j-1], S_{apx}) + 2d \leq \text{pos}(L_t[j], S_{apx}) < \text{pos}(L_{t-1}[j], S_{apx}) \leq \text{pos}(L_{t-1}[j+1], S_{apx}) - 2d$ (for all those entries that exist).

**Lemma 4** Let $S' = \langle s'_1, \ldots, s'_m \rangle$ be the sequence that our Approximation-Algorithm returns. Then, $S'$ is a longest subsequence of $S$ that is $2d$-distant in $S_{apx}$.

**Proof** Lemma 3 implies that $S'$ is a subsequence of $S$ and $2d$-distant in $S_{apx}$. Let $S^* = \langle s^*_1, \ldots, s^*_m \rangle$ be a longest subsequence of $S$ that is $2d$-distant in $S_{apx}$. We now show that $|S'| \geq |S^*|$. In particular, we show by induction that after iteration $t^*_i$, $\text{pos}(L^*_i[j], S_{apx}) \leq \text{pos}(s^*_i, S_{apx})$. For the base case, consider iteration $t^*_1$, where $s^*_1$ is processed. Either $s^*_1$ gets inserted into some position $j \geq 1$, i.e., $L^*_1[j] = s^*_1$, or not. If it gets inserted, then by conditions in lines 7 or 9 in Algorithm 2, $\text{pos}(L^*_1[1], S_{apx}) \leq \text{pos}(s^*_1, S_{apx})$. If it does not get inserted, then it must hold that $L^*_1[1] = L^*_1-1[1]$ and therefore also $\text{pos}(L^*_1[1], S_{apx}) < \text{pos}(s^*_1, S_{apx})$.

**Algorithm 2: Approximation-Algorithm**

```
1 $S_{apx} \leftarrow$ approximately sort $S$ as shown in [17] ;
2 $d \leftarrow c \cdot \log n$  // $\exists c$ s.t. w.h.p. $\text{disl}(s) \leq c \cdot \log n$ [17];
3 $L[1] \leftarrow s_1$;
4 $k \leftarrow 1$;
5 foreach $i = 2, \ldots, n$ do
6     $x \leftarrow s_i$;
7     if $\text{pos}(x, S_{apx}) < \text{pos}(L[1], S_{apx})$ then $L[1] \leftarrow x$;
8     else
9         $j \leftarrow \text{max}\{j \leq k : \text{pos}(L[j], S_{apx}) < \text{pos}(x, S_{apx})\}$;
10        if $\text{pos}(L[j], S_{apx}) + 2d \leq \text{pos}(x, S_{apx})$ then
11            if $j = k$ then $k \leftarrow k + 1$;
12            $L[j + 1] \leftarrow x$;
13            $\text{prec}[x] \leftarrow L[j]$;
14     end
15 $\text{lis}[1] \leftarrow L[k]$;
16 foreach $i = 2, \ldots, k$ do
17     $\text{lis}[i] \leftarrow \text{prec}[\text{lis}[i - 1]]$;
18 return lis;
```

For the step case, consider iteration $t^*_{i+1}$, where $s^*_{i+1}$ is processed. We can make two observations: (i) the value of $k$ can only increase during the algorithm, and (ii) for any $t' < t$ and $j \leq k_{t'}$ it holds that $\text{pos}(L_{t'}[j], S_{apx}) \geq \text{pos}(L_t[j], S_{apx})$. By these observations, and by induction hypothesis and the assumption that $S^*$ is $2d$-distant in
S^{apx}, we get that \( \text{pos}(L_{i+1}^*[i], S^{apx}) + 2d \leq \text{pos}(s_{i+1}^*, S^{apx}) \). Moreover, Lemma 3 implies that \( \text{pos}(L_{i+1}^*[i], S^{apx}) + 2d \leq \text{pos}(L_{i+1}^*[i] + 1, S^{apx}) \). Thus, if \( s_{i+1}^* \) does not get inserted, it is because \( \text{pos}(L_{i+1}^*[i] + 1, S^{apx}) < \text{pos}(s_{i+1}^*, S^{apx}) \), and if it gets inserted, it will be in some position \( j \geq i + 1 \). In any case, the hypothesis also holds after the iteration iteration \( i_{i+1}^* \), which means that \( S' \) has indeed maximum length.

3.3 Proof of Theorem 1

We now prove the initially stated Theorem 1, which for convenience, we restate here:

**Theorem 1 (Upper Bounds)** For any input sequence \( S \) that contains \( n \) distinct elements and any error probability \( p < 1/16 \), our Approximation-Algorithm computes an \( O(\log n) \)-approximation to the longest increasing sequence of \( S \), in \( O(n \log n) \) time, with probability at least \( 1 - \frac{1}{n} \).

**Proof** Let \( d \in O(\log n) \) according to Theorem 4, such that with probability \( 1 - \frac{1}{n} \), the maximum dislocation in \( S^{apx} \) is at most \( d \). If this is true, by Lemmata 1-4, our Approximation-Algorithm returns a subsequence \( S' \) of \( S \) that is increasing, and that has length at least \( \frac{\text{LIS}(S)}{2d} \in \Omega \left( \frac{\text{LIS}(S)}{\log n} \right) \).

The running time consists of the initial sorting step, which by Theorem 4 takes \( O(n \log n) \) time, and the \( n \) iterations of the algorithm, which take \( O(\log n) \) time each if binary search (using the redefined comparison results) is used to implement line 9. The final construction of the output takes \( O(k) \) time, where \( k \leq \text{LIS}(S) \leq n \) is the length of the approximation.

Finally, recall that our approximation guarantee on longest increasing subsequence relies on the fact that one can sort \( n \) elements such that the maximum dislocation is at most \( O(\log n) \) with high probability. This result is achieved by the algorithm in [17], and holds for \( p < 1/16 \). If the error probability is larger, i.e. \( p \) is between 1/16 and 1/2, then we can achieve asymptotically the same \( O(\log n) \) maximum dislocation with the algorithm in [6], but at the cost of a significantly larger running time. See Section 1.1 for more details.

4 Lower Bound on the Approximation Factor

We continue this paper with a lower bound on the approximation factor, that implies that the upper bound we showed in Theorem 1 is tight up to constant factors. In particular, we prove Theorem 2, which we restate here:

1By modifying this algorithm so that it returns also the mapping from each element in \( S \) to its position in \( S^{apx} \) we can obtain the new comparison results in the same time.
Theorem 2 (Lower Bound – Approximation Factor) There exists a collection of sequences $S$ (permutations of length $n$), a probability distribution on $S$, and a constant $c$ depending on $p$, such that no algorithm can return an $c \log n$-approximation of the longest increasing subsequence with probability $1 - \frac{1}{n}$.

Our proof can be seen as a generalization of the lower bound on the maximum dislocation for sorting (see proof of Theorem 9 in [15]), where it is shown that two elements whose ranks differ by less than $O(\log n)$ are likely to be indistinguishable by any algorithm, and hence to appear in the wrong relative order. Intuitively, the argument there is as follows: consider the sorted sequence and the sequence obtained by swapping two elements, and assume that the comparison outcomes on these sequences look identically. It turns out that the probability of this happening is larger than $\frac{1}{n}$, whenever the rank difference is smaller than $O(\log n)$, since only a small number of comparison outcomes must differ.

This is not enough in our case, since an algorithm could simply ignore such two elements. For instance, consider an increasing sequence of $c$ adjacent elements. If the first and the last element are swapped, the algorithm could simply return the subsequence without these two elements and be almost optimal. A first idea to fix this problem could be to consider the case, where one observes the whole increasing sequence to be reversed. However, to have this happen with probability larger than $\frac{1}{n}$, $c$ needs to be smaller than $O(\sqrt{n \log n})$, thus implying a weaker lower bound.

Instead, we shall use a collection of similar sequences (more than two), such that if an algorithm succeeds on one of these sequences it must fail on another one.

**Proof** We say that an algorithm succeeds if it returns a $(c \log n)$-approximation for some constant $c < \frac{1}{2 \log \frac{1}{1-p}}$, otherwise we say it fails. We shall first define our collection $S$ of similar sequences. Let $\eta := \lceil \frac{\log n}{2 \log \frac{1}{1-p}} \rceil$. Let $S^*$ denote the sequence, in which the largest $\eta$ elements appear first in increasing order and then the remaining elements appear in decreasing order,

$$S^* := (n - \eta + 1, \ldots, n - 1, n, n - \eta, \ldots, 1).$$

Furthermore, for $1 \leq i < \eta$, let $S(i)$ be the sequence obtained from $S^*$ when the largest element is moved to position $i$,

$$S(i) := (n - \eta + 1, \ldots, n - \eta + (i - 1), n, n - \eta + i, \ldots, n - 1, n - \eta, \ldots, 1).$$

Now, let $S := \{S^*, S(1), S(2), \ldots, S(\eta - 1)\}$ (note that $S^* = S(\eta)$) and let $P$ be the uniform distribution over $S$. We will show (proof by contradiction) that no algorithm succeeds on this pair $(S, P)$ with probability at least $1 - \frac{1}{n}$. 

© Springer
Assume towards a contradiction that algorithm A succeeds with high probability on a sequence $S'$ chosen uniformly at random from $S$, i.e.,

$$\Pr(A(S') \text{ succeeds}) = \sum_{i=1}^{\eta} \Pr(A(S(i)) \text{ succeeds}) \cdot \Pr(S' = S(i))$$

$$= \frac{1}{\eta} \sum_{i=1}^{\eta} \Pr(A(S(i)) \text{ succeeds})$$

$$\geq 1 - \frac{1}{n}.$$  

This implies that

$$P := \Pr(A(S^*) \text{ succeeds}) \geq 1 - \frac{\eta}{n}, \quad (2)$$

since by hypothesis

$$\frac{1}{\eta} \cdot P \geq \left(1 - \frac{1}{n}\right) - \frac{1}{\eta} \cdot \sum_{i=1}^{\eta-1} \Pr(A(S(i)) \text{ succeeds}) \cdot \frac{\eta}{n},$$

and is tight in the case where the algorithm succeeds on all input sequences $S' \neq S^*$ with probability 1.

Let $C \in \{\prec, \succ\}^{(n \choose 2)}$, then $A(S, C)$ means that algorithm A runs on sequence $S$ and observes comparison outcomes $C$. Now, consider the set of all comparison outcomes that the algorithm can observe and let

$$C := \{C \in \{\prec, \succ\}^{(n \choose 2)} : A(S^*, C) \text{ succeeds}\}$$

denote the set of all possible comparison outcomes for which $A$ succeeds on input $S^*$. We define $R(S) \in \{\prec, \succ\}^{(n \choose 2)}$ to be the random variable corresponding to the comparison outcomes as they would be observed by the algorithm when the input sequence is $S$. Then, the probability that $A(S^*)$ succeeds is expressed by the total probabilities of the events that $A$ observes comparison outcomes in $C$,

$$P = \Pr(A(S^*) \text{ succeeds}) = \sum_{C \in C} \Pr(R(S^*) = C). \quad (3)$$

Before we continue the proof, we shall first show the following lemma.

**Lemma 5** For every $S \in S \setminus \{S^*\}$ and $C \in \{\prec, \succ\}^{(n \choose 2)}$, it holds that

$$\Pr(R(S) = C) > \Pr(R(S^*) = C) \cdot \left(\frac{p}{1-p}\right)^{\eta}.$$  

**Proof** Consider $S^* = (s_1^*, \ldots, s_n^*)$ and $C$ and let $E(S^*, C)$ be the set of wrong comparison results, i.e., the set of pairs $(s_i^*, s_j^*)$ with $i < j$ such that either $s_i^* < s_j^*$ and $c_{i,j} = \succ$ (i.e., $s_i^* > s_j^*$) or $s_i^* > s_j^*$ and $c_{i,j} = \prec$. Thus,

$$\Pr(R(S^*) = C) = (1 - p)^{\binom{n}{2} - |E(S^*, C)|} \cdot p^{|E(S^*, C)|} = (1 - p)^{\binom{n}{2}} \cdot \left(\frac{p}{1-p}\right)^{|E(S^*, C)|}. $$
Now consider \( S = S(k) = \langle s_1, \ldots, s_n \rangle \) and observe that only the relative order of the pairs \((s_k, s_j)\) with \( k < j \leq \eta \), changed compared to \( S^* \). This implies that there can be at most \( \eta - k < \eta \) additional wrong comparison results, i.e., \(|E(S, C)| = |E(S^*, C)| + \eta\). Therefore, and since \( \frac{p}{1-p} < 1 \) as \( p < \frac{1}{2} \),

\[
\Pr(R(S) = C) = (1 - p)^{\binom{n}{2}} \cdot \left( \frac{p}{1-p} \right)^{|E(S, C)|}
> (1 - p)^{\binom{n}{2}} \cdot \left( \frac{p}{1-p} \right)^{|E(S^*, C)| + \eta}
= \Pr(R(S^*) = C) \cdot \left( \frac{p}{1-p} \right)^{\eta}.
\]

\[ \square \]

**Continuation of the Proof of Theorem 2.** Now notice that in order to succeed, \( A \) needs to return at least two of the first \( \eta \) elements in \( S^* \). Therefore, we can map every \( C \in C \) to a (not necessarily unique) sequence of \( S \) as follows: for each \( C \in C \), let \( i_C \) be the position of the first element that \( A(S^*, C) \) returns and let \( S(C) := S(i_C) \). (Note that \( i_C < \eta \) as otherwise \( A \) does not return at least two elements of the first \( \eta \) elements in \( S^* \).) For each \( S \in S \setminus \{ S^* \} \),

\[
\Pr(A(S) \text{ fails}) \geq \sum_{C \in C: S = S(C)} \Pr(R(S) = C)
> \sum_{C \in C: S = S(C)} \Pr(R(S^*) = C) \cdot \left( \frac{p}{1-p} \right)^{\eta},
\]

where the second inequality follows from Lemma 5. As a consequence, for \( S' \in S \) chosen uniformly at random,

\[
\Pr(A(S') \text{ fails}) \geq \sum_{S \in S \setminus \{ S^* \}} \Pr(S' = S) \cdot \Pr(A(S) \text{ fails})
> \sum_{S \in S \setminus \{ S^* \}} \frac{1}{\eta} \sum_{C \in C: S = S(C)} \Pr(R(S^*) = C) \cdot \left( \frac{p}{1-p} \right)^{\eta}
= \frac{1}{\eta} \left( \frac{p}{1-p} \right)^{\eta} \sum_{S \in S \setminus \{ S^* \}} \sum_{C \in C: S = S(C)} \Pr(R(S^*) = C)
\geq \frac{1}{\eta} \left( \frac{p}{1-p} \right)^{\eta} \sum_{C \in C} \Pr(R(S^*) = C)
\geq \frac{1}{\eta} \left( \frac{p}{1-p} \right)^{\eta} \left( 1 - \frac{\eta}{n} \right),
\]

where from line 3 to line 4 we use that every instance of comparison results is mapped to exactly one sequence, and on the last line we use (2) and (3).
Now, observe that for $n > 2\eta$, we have $(1 - \frac{\eta}{n}) > \frac{1}{2}$ and that, by our choice of $\eta$, 
\[
\left(\frac{p}{1-p}\right)^{\eta} \geq \frac{1}{\sqrt{n}}.
\]
Therefore, again for $n > 2\eta$,
\[
\Pr(A(S') \text{ fails}) > \frac{2\log \frac{1-p}{p}}{\log n} \cdot \frac{1}{\sqrt{n}} \cdot \frac{1}{2} > \frac{1}{n}.
\]
To conclude the proof, note that this is a contradiction to our assumption that $A$ succeeds with probability at least $1 - \frac{1}{n}$.

The lower bound shown in Theorem 2 holds for all deterministic algorithms, but can be expanded to also hold for probabilistic algorithms as explained in the following remark.

**Remark 1** To make the lower bound on the approximation factor work also for any randomized algorithm $A$, we can turn $A$ into a deterministic version by fixing a sequence $\lambda \in \{0, 1\}^t$ random bits that can be used by the algorithm. Thus, for the resulting deterministic algorithm $A_{\lambda}$, the lower bound holds. Let $p_{\lambda}$ be the probability to generate the sequence $\lambda$ of random bits. To lower bound the probability that $A(S')$ fails, where $S'$ is chosen uniformly at random from $S$, one simply needs to sum over all $\lambda$ the probabilities that $A_{\lambda}$ fails multiplied by $p_{\lambda}$, i.e., 
\[
\Pr(A(S') \text{ fails}) = \sum_{\lambda \in \{0, 1\}^t} \Pr(A_{\lambda}(S') \text{ fails}) \cdot p_{\lambda} > \frac{1}{n} \sum_{\lambda \in \{0, 1\}^t} p_{\lambda} = \frac{1}{n}.
\]

### 5 Lower Bound on the Running Time

We complement the upper bound in Theorem 1 by showing that the running time of our Approximation-Algorithm is asymptotically optimal. In [13], it is shown that (in the error-free model) computing the longest increasing subsequence is at least as hard as sorting. We will use this proof as a starting point to show Theorem 3 which we restate here:

**Theorem 3** (Lower Bound – Running Time) Any log $n$-approximation algorithm for longest increasing subsequence requires $\Omega(n \log n)$ comparisons, even if no errors occur.

The proof techniques of the lower bound in [13] are as follows: Assume that we are in the error-free case. Consider the easier problem of deciding on a given sequence $S$ of $n$ distinct elements whether $|LIS(S)| < k$, and consider the comparison tree of an algorithm $A$ with leaves that tell as an answer to this question either “yes” or “no”. Without loss of generality, assume that no useless comparisons are made on a root to a leaf path (i.e., no comparison twice and no comparisons whose outcome is predictable by the outcomes of previous comparisons).

Every leaf $\ell$ can be associated with a partial order implied by a set of linear orderings on $S$ that are consistent with the transitive closure of the comparisons performed on the path from the root to $\ell$. If the answer in a leaf is “yes”, this implies that there
are no $k$ elements of $S$ that are pairwise incomparable in this partial order (i.e., the relative order of every pair is neither tested in any comparison on the path, nor implied by other comparisons), as otherwise, these elements could possibly form an increasing sequence of length $k$. Such a subset of elements is called antichain, while a chain is a subset of elements that are linearly ordered. An important property of chains and antichains is based on the so-called Dillworth theorem:

**Lemma 6** (Lemma 3.1 in [13]) *In any finite partial order, the elements can be partitioned into $m$ chains, where $m$ is the size of the largest antichain.*

This implies that in a “yes”-leaf, the elements can be partitioned into less than $k$ chains, since there is no antichain of size $k$. Furthermore, given such a partition into (less than) $k$ chains, the elements can be sorted with $n \log k + O(n)$ comparisons, think for instance of natural merge sort:

**Lemma 7** (Lemma 3.3 in [13]) *If a linear order is partitioned into $k$ chains, then this linear order can be algorithmically restored with at most $n \lceil \log k \rceil$ comparisons.*

In order to lower bound the number of comparisons needed to end in a “yes”-leaf, algorithm $A$ can be extended to $A^*$ as follows: whenever $A$ concludes to be in a “yes”-leaf, $A^*$ continues to completely sort the elements of $S$ (which requires no more than $n \log k + O(n)$ further comparisons). Let $S(n, k)$ denote the number of linear orderings of the elements in $S$ that end in a “yes”-leaf, i.e., the number of linear orderings such that the longest increasing subsequence in $S$ is strictly smaller than $k$. Then,

$$S(n, k) \geq n! \left(1 - \frac{n}{k}\right),$$

since there are $n!$ different linear orderings and $\binom{n}{k}$ possible subsequences of size $k$ each increasing with probability $1/k!$. The comparison tree corresponding to $A^*$ has thus at least $S(n, k)$ leaves, and therefore must perform at least $\log S(n, k)$ comparisons in its worst case. Therefore and by Lemma 7, algorithm $A$ must perform at least

$$\log S(n, k) - n \log k - O(n)$$

comparisons in its worst case to end up in a “yes”-leaf, which is $\Omega(n \log n)$ when choosing $k = 3 \cdot n^{1/2}$, since in this case $0 < \left(\binom{n}{k}\right)/k! < 1$ converges to 0 with $n$ going to infinity, and therefore $S(n, k) \sim n!$ (see also Theorem 3.5 in [13]).

We can use the above proof techniques to show that every algorithm, that computes a $\log n$-approximation on longest increasing subsequence must perform at least $\Omega(n \log n)$ comparisons.

**Proof of Theorem 3** Let $B$ be an $\log n$-approximation algorithm for $LIS(S)$ under our error model (i.e., we can always simulate our error model in the error-free case) and consider a relaxation of the problem of determining whether $|LIS(S)|$ is smaller than $k \log n$. In this relaxation we require the answer to be “yes” if $|LIS(S)| < k$
and “no” if $|LIS(S)| \geq k \log n$, while we do not impose any restriction on the range $k \leq |LIS(S)| < k \log n$.

It is clear that algorithm $B$ can be used to solve this relaxed problem without increasing the number of needed comparisons. Therefore, the associated comparison tree must reach a leaf corresponding to answer “yes” for all linear orderings on the elements in $S$ that contain no increasing subsequence of length $k$, while the largest antichain in any ordering for which the tree reaches such a leaf is smaller than $k \log n$. This implies, by using Lemmata 6 and 7, that algorithm $B^*$ (now in the error-free case) needs at most $n \log(k \log n) + O(n)$ further comparisons in the worst case to sort the elements in $S$, and $B$ thus needs at least

$$\log S(n, k) - n \log(k \log n) - O(n)$$

comparisons in the worst case to end in a “yes”-leaf, which is in $\Omega(n \log n)$ if we set again $k = 3 \cdot n^{1/2}$. This step follows since by Stirling’s approximation $k! \geq (k/e)^k$ and by using that $n!/(n-k)! \leq n^k$. Hence,

$$S(n, 3 \cdot n^{1/2}) \geq n! \left(1 - \frac{n!}{(n-3 \cdot n^{1/2})!(3 \cdot n^{1/2})!(3 \cdot n^{1/2})!}\right)$$

$$\geq n! \left(1 - \frac{n^{3 \cdot n^{1/2}}}{\left(\frac{9n}{e^2}\right)^{3 \cdot n^{1/2}}}\right)$$

$$\geq n! \left(1 - 1.2^{-3 \cdot n^{1/2}}\right),$$

which is larger than $1/2 \cdot n!$ for $n \geq 2$. Therefore,

$$\log S(n, 3 \cdot n^{1/2}) \geq \log \frac{n!}{2} \geq n \log n - O(n),$$

while

$$n \log(3 \cdot n^{1/2} \log n) \leq \frac{1}{2} n \log n + O(n \log \log n).$$

Finally, we can conclude that our Approximation-Algorithm performs in asymptotically optimal time, since we can always simulate our error model in the error-free case.

### 6 Generalization

In this section, we would like to explain how the upper bound on the approximation factor as shown in Theorem 1 can be generalized. Our Approximation-Algorithm actually succeeds whenever the approximately sorted sequence has maximum dislocation at most some parameter $d$. This implies that the result can be parameterized and also used in other models with comparison errors.
Whenever one can obtain a total order with maximum dislocation $d$, the Approximation-Algorithm is $2d$-approximative.

Consider for instance the so-called threshold-model [1, 14, 18], where comparisons between numbers that differ by more than some threshold $\tau$ are always correct, while those between numbers that differ by less than $\tau$ can fail persistently (with some probability possibly depending on the difference or even adversarially). For instance, it has been shown in [18] that if the input sequence $S$ is a permutation of the numbers $\{1, \ldots, n\}$, then running Quicksort in this error model yields an output sequence with maximum dislocation $2\tau$. Thus, our Approximation-Algorithm finds a $4\tau$-approximation of the longest increasing subsequence in $S$.

7 Conclusion

It turns out that a logarithmic approximation ratio on the problem of computing a longest increasing subsequence is the best one that can be obtained in the presence of persistent comparison errors. In this respect, it is interesting to see that there exist such simple recipes to compute a logarithmic approximation. We have seen in the very beginning (Proof of Lemma 2 in Section 3.1) one recipe that can use as a black box any algorithm that computes a longest increasing sequence if no comparison errors happen. And we have seen afterwards (Section 3.2) another recipe that dives into such an algorithm and changes the rule of when to add (or append) an element to a previously computed increasing subsequence. Note that this approach also works for the very similar patience sort algorithm [5, 24]. As indicated earlier, our Approximation-Algorithm has the advantage to perform much better than $O(\log n)$-approximate on many input sequences and is even optimal in the case where the longest increasing subsequence is already $2d$-distant in $S^{apx}$, whereas this is not necessarily true when using the black-box recipe.

Recall that we assume the elements in the input sequence $S$ to be distinct. If this is not true, then we can observe that our algorithm returns an $O(\log n)$-approximation to the longest non-decreasing subsequence of $S$. In order to still find a strictly increasing sequence, the algorithm would need to have access to an “equality oracle” that correctly tells whether two elements are equal or not.

References

1. Ajtai, M., Feldman, V., Hassidim, A., Nelson, J.: Sorting and selection with imprecise comparisons. ACM Trans. Algorithm. 12(2), 19 (2016)
2. Aldous, D., Diaconis, P.: Longest increasing subsequences: from patience sorting to the baik-deift-johansson theorem. Bull. Am. Math. Soc. 36(4), 413–432 (1999)
3. Alonso, L., Chassaing, P., Gillet, F., Janson, S., Reingold, E.M., Schott, R.: Quicksort with unreliable comparisons: a probabilistic analysis. Comb. Probab. Comput. 13(4-5), 419–449 (2004)
4. Baik, J., Deift, P., Johansson, K.: On the distribution of the length of the longest increasing subsequence of random permutations. J. Am. Math. Soc. 12(4), 1119–1178 (1999)
5. Bespamyatnikh, S., Segal, M.: Enumerating longest increasing subsequences and patience sorting. Inf. Process. Lett. 76(1-2), 7–11 (2000)
6. Braverman, M., Mossel, E.: In: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, pp. 268–276 (2008)
7. Chandramouli, B., Goldstein, J.: Patience is a virtue: revisiting merge and sort on modern processors. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, pp. 731–742 (2014)
8. Clark, G.C., Cain, J.B.: Error-correction Coding for Digital Communications, 1st edn. Springer Publishing Company Incorporated (2013)
9. Crochemore, M., Porat, E.: Fast computation of a longest increasing subsequence and application. Inf. Comput. 208(9), 1054–1059 (2010)
10. Damaschke, P.: The solution space of sorting with recurring comparison faults. In: Combinatorial Algorithms - 27th International Workshop, IWOCA 2016, Helsinki, pp. 397–408 (2016)
11. Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., Salzberg, S.L.: Alignment of whole genomes. Nucleic Acids Res. 27(11), 2369–2376 (1999)
12. Feige, U., Raghavan, P., Peleg, D., Upfal, E.: Computing with Noisy Information. SIAM J. Comput. 23(5), 1001–1018 (1994)
13. Michael, L.: Fredman on computing the length of longest increasing subsequences. Discret. Math. 11(1), 29–35 (1975)
14. Funke, S., Mehlhorn, K., Näher, S.: Structural filtering: a paradigm for efficient and exact geometric programs. Comput. Geom. 31(3), 179–194 (2005)
15. Geissmann, B., Leucci, S., Liu, C.-H., Penna, P.: Sorting with recurrent comparison errors. In: 28th International Symposium on Algorithms and Computation, ISAAC 2017, Phuket, pp. 38:1–38:12 (2017)
16. Geissmann, B., Leucci, S., Liu, C.-H., Penna, P.: Optimal dislocation with persistent errors in sub-quadratic time. In 35th Symposium on Theoretical Aspects of Computer Science, STACS, 2018, Caen France, pp. 36:1–36:13 (2018)
17. Geissmann, B., Leucci, S., Liu, C.-H., Penna, P.: Optimal sorting with persistent comparison errors. In: to appear in 27th Proceedings of the European Symposium on Algorithms, ESA 2019, preliminary version in ArXiv:1804.07575 (2019)
18. Geissmann, B., Penna, P.: Inversions from sorting with distance-based errors. In: Proceedings of 44th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM, 2018, pp. 508–522 (2018)
19. Hadjicostas, P., Lakshmanan, K.B.: Bubble sort with erroneous comparisons. Aust. J. Comb. 31, 85–106 (2005)
20. Hadjicostas, P., Lakshmanan, K.B.: Measures of disorder and straight insertion sort with erroneous comparisons. Ars Comb. 98, 259–288 (2011)
21. Hadjicostas, P., Lakshmanan, K.B.: Recursive merge sort with erroneous comparisons. Discret. Appl. Math. 159(14), 1398–1417 (2011)
22. Klein, R., Penninger, R., Sohler, C., Woodruff, D.P.: Tolerant algorithms. In: Algorithms - ESA 2011 - 19th Annual European Symposium. Proceedings, Saarbrücken, pp. 736–747 (2011)
23. Liben-Nowell, D., Vee, E., Zhu, A.: Finding longest increasing and common subsequences in streaming data. J. Comb. Optim. 11(2), 155–175 (2006)
24. Mallows, C.: Patience sorting. Bullet. Inst. Math. Appl. 9, 216–224 (1973)
25. Masek, W.J., Paterson, M.S.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. 20(1), 18–31 (1980)
26. Muthukrishnan, S.: Data streams: Algorithms and applications. Foundations and Trends in Theoretical Computer Science 1(2), 117–236 (2005)
27. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett. 6(3), 80–82 (1977)
28. Boas van Emde, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Math. Syst. Theory 10, 99–127 (1977)
29. Yang, I.-H., Huang, C.-p., Chao, K.-m.: A fast algorithm for computing a longest common increasing subsequence. Inf. Process. Lett. 93(5), 249–253 (2005)
30. Zhang, H.: Alignment of BLAST high-scoring segment pairs based on the longest increasing subsequence algorithm. Bioinformatics 19(11), 1391–1396 (2003)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.