Optimal Sorting with Persistent Comparison Errors

Barbara Geissmann
Department of Computer Science, ETH Zürich, Switzerland
barbara.geissmann@inf.ethz.ch

Stefano Leucci
Department of Algorithms and Complexity, Max Planck Institute for Informatics, Germany[*]
`https://www.stefanoleucci.com`
stefano.leucci@mpi-inf.mpg.de

Chih-Hung Liu
Department of Computer Science, ETH Zürich, Switzerland
chih-hung.liu@inf.ethz.ch

Paolo Penna
Department of Computer Science, ETH Zürich, Switzerland
paolo.penna@inf.ethz.ch

Abstract

We consider the problem of sorting n elements in the case of *persistent* comparison errors. In this problem, each comparison between two elements can be wrong with some fixed (small) probability p, and *comparisons cannot be repeated* (Braverman and Mossel, SODA'08). Sorting perfectly in this model is impossible, and the objective is to minimize the *dislocation* of each element in the output sequence, that is, the difference between its true rank and its position. Existing lower bounds for this problem show that no algorithm can guarantee, with high probability, *maximum dislocation* and *total dislocation* better than $\Omega(\log n)$ and $\Omega(n)$, respectively, *regardless of its running time.*

In this paper, we present the first $O(n \log n)$-*time* sorting algorithm that guarantees both $O(\log n)$ *maximum dislocation* and $O(n)$ *total dislocation* with high probability. This settles the time complexity of this problem and shows that comparison errors do not increase its computational difficulty: a sequence with the best possible dislocation can be obtained in $O(n \log n)$ time and, even without comparison errors, $\Omega(n \log n)$ time is necessary to guarantee such dislocation bounds.

In order to achieve this optimality result, we solve two sub-problems in the persistent error comparisons model, and the respective methods have their own merits for further application. One is how to locate a position in which to insert an element in an almost-sorted sequence having $O(\log n)$ maximum dislocation in such a way that the dislocation of the resulting sequence will still be $O(\log n)$. The other is how to simultaneously insert m elements into an almost sorted sequence of m different elements, such that the resulting sequence of $2m$ elements remains almost sorted.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases approximate sorting, comparison errors, persistent errors

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.49

Related Version A full version of the paper is available at `https://arxiv.org/abs/1804.07575`.

Funding Research supported by SNF (project number 200021_165524).

Acknowledgements The authors wish to thank Peter Widmayer for many insightful discussions.

[*] Part of this work was completed while the author was affiliated with ETH Zürich.

1 Introduction

We study the problem of *sorting* n distinct elements under *persistent random comparison errors*, where each comparison is wrong with some fixed (small) probability p, and the errors are independent over all possible pairs of elements. There are two types of comparison errors, *persistent* and *non-persistent*. For non-persistent errors, it is possible to repeat the same comparison several times and each result is wrong with probability p independently of the others. In the 80s and 90s, non-persistent errors have received considerable attention, and it has been shown that the perfectly sorted sequence can be computed in $O(n \log n)$ time with high probability. For persistent errors the repetition of a single comparison always yields the same outcome and this makes *impossible* to consistently recover the *perfectly sorted* sequence, as we explain below. The goal is therefore that of computing an *almost sorted* sequence. This seems a challenging task as all known algorithms have rather high running time and only recently a sub-quadratic running time has been achieved (see below for details). In particular, whether an optimal $O(n \log n)$ running time is sufficient for sorting with *persistent* comparison errors is a fundamental open question.

The above persistent-errors model is a well-studied theoretical abstraction of the errors that arise in hardware architectures. Here, avoiding these errors requires involved fault-tolerant mechanisms that reduce performances and increase manufacturing costs and energy consumption. Recently, a contrasting trend of simplifying hardware architectures has emerged: errors are traded for cheaper manufacturing costs, lower energy consumption, or better performances. The study of sorting algorithms with persistent comparison errors has been also motivated in [5, 13] by experts comparing items according to their importance, by ranking in sports where comparisons correspond to matches between teams, and –more generally– by situations where one wants to aggregate noisy comparisons into a global ranking and repeating a comparison is impossible or too expensive.

A common way to measure the quality of an output sequence in terms of sortedness, is to consider the *dislocation* of an element, which is the difference between its position in the output and its position in the correctly sorted sequence. In particular, a reasonable measure is the *maximum dislocation* of any element in the sequence or the *total dislocation* of the sequence, i.e., the sum of the dislocations of all n elements.

To see why sorting with persistent errors is much more difficult than the case in which comparisons can be repeated, note that in the latter case there is a trivial $O(n \log^2 n)$ time solution to sort perfectly with high probability (simply repeat each comparison $O(\log n)$ times and take the majority of the results). Instead, in the model with persistent errors, it is impossible to sort perfectly as, for any constant p, no algorithm can achieve a maximum dislocation that is smaller than $\Omega(\log n)$ w.h.p., or total dislocation smaller than $\Omega(n)$ in expectation [10]. This problem has been extensively studied in the literature, and several algorithms have been devised with the goal of sorting *quickly* with small dislocation (see Table 1). Unfortunately, even though all the algorithms achieve the best possible maximum dislocation of $\Theta(\log n)$, they use a truly superlinear number of comparisons (specifically, $\Omega(n^c)$ with $c \geq 1.5$), and/or require significant amount of time (namely, $O(n^{3+c_p})$ where c_p is a big constant that depends on p). This naturally suggests the following question:

What is the time complexity of sorting optimally with persistent errors?

In this work, we answer this basic question by showing the following result:

There exists an algorithm with ***optimal running time*** $O(n \log n)$ *which achieves simultaneously* ***optimal maximum dislocation*** $O(\log n)$ *and* ***optimal total dislocation*** $O(n)$*, both* ***with high probability****.*

Table 1 The existing approximate sorting algorithms and our result. The constant c_p in the exponent of the running time of [5] depends on the error probability p and it is typically quite large. We write $\Omega(f(n))$ w.h.p. (resp. exp.) to mean that no algorithm can achieve dislocation $o(f(n))$ with high probability (resp. in expectation).

Upper bounds

Running Time	Max Dislocation	Tot Dislocation	Reference
$O(n^{3+c_p})$	$O(\log n)$ w.h.p.	$O(n)$ w.h.p.	[5]
$O(n^2)$	$O(\log n)$ w.h.p.	$O(n \log n)$ w.h.p.	[13]
$O(n^2)$	$O(\log n)$ w.h.p.	$O(n)$ expected	[10]
$\tilde{O}(n^{3/2})$	$O(\log n)$ w.h.p.	$O(n)$ expected	[11]
$O(n \log n)$	$O(\log n)$ w.h.p.	$O(n)$ w.h.p.	**this work**

Lower bounds

Any	$\Omega(\log n)$ w.h.p.	$\Omega(n)$ expected	[10]

The dislocation guarantees of our algorithm are optimal, due to the lower bound of [10], while the existence of an algorithm achieving a maximum dislocation of $d = O(\log n)$ in time $T(n) = o(n \log n)$ would immediately imply the existence of an algorithm that sorts n elements in $T(n) + O(n \log \log n) = o(n \log n)$ time, even in the absence of comparison errors, thus contradicting the classical $\Omega(n \log n)$ lower bound for comparison-based algorithms.[1]
Along the way to our result, we consider the problem of *searching with persistent errors*, defined as follows:

We are given an approximately sorted sequence S, and an additional element $x \notin S$. The goal is to compute, under persistent comparison errors, an approximate rank *(position) of x which differs from the true rank of x in S by a* small *additive error.*

For this problem, we show an algorithm that requires $O(\log n)$ time to compute, w.h.p., an approximate rank that differs from the true rank of x by at most $O(\max\{d, \log n\})$, where d is the maximum dislocation of S. For $d = \Omega(\log n)$ this allows to insert x into S without any asymptotic increase of the maximum (and total) dislocation in the resulting sequence. Notice that, if d is also in $O(n^{1-\epsilon})$ for any constant $\epsilon > 0$, this is essentially the best we can hope for, as an easy decision-tree lower bound shows that any algorithm must require $\Omega(\log n)$ time. Finally, we remark that [13] considered the variant in which the original sequence is *sorted*, and the algorithm must compute the correct rank. For this problem, they present an algorithm that runs in $O(\log n \cdot \log \log n)$ time and succeeds with probability $1 - f(p)$, with $f(p)$ vanishing as p goes to 0. As by-product of our result, we can obtain the optimal $O(\log n)$ running time with essentially the same success probability. Similarly to other prior related works, all our results apply when p is below a sufficiently small constant, e.g., $p < 1/20$ in [13]. For technical simplicity, throughout this work we assume $p < 1/32$, though the results hold for $p < 1/16$ as in [10].[2]

[1] Indeed, the smallest d elements of a sequence S having dislocation $d = 2^{o(\log n)}$ can be found in time $O(d \log d)$ using any $O(n \log n)$-time sorting algorithm on the first $2d$ elements of S. Removing those elements and repeating the above procedure $O(\frac{n}{d})$ times, would allow to sort in $T(n) + O(\frac{n}{d} \cdot d \log d) = o(n \log n)$ time.

[2] Except for the derandomization technique of Section 5, all our results also hold for the case in which each comparison is wrong with an adversarially chosen and unknown probability in $[0, p]$.

1.1 Main Intuition and Techniques

Approximate Sorting

In order to convey the main intuitions behind our $O(n \log n)$-time optimal-dislocation approximate sorting algorithm, we consider the following ideal scenario: we already have a perfectly sorted sequence A containing a random half of the elements in our input sequence S and we, somehow, also know the position in which each element $x \in S \setminus A$ should be inserted into A so that the resulting sequence is also sorted (i.e, the *rank* of x in A). If these positions alternate with the elements of A, then, to obtain a sorted version of S, it suffices to *merge* S and $S \setminus A$, i.e., to simultaneously insert all the elements of $S \setminus A$ into their respective positions of A. Unfortunately, we are far from this ideal scenario for several reasons: first of all, multiple elements in $S \setminus A$, say δ of them, might have the same rank in A. Since we do not know the order in which those elements should appear, this will already increase the dislocation of the merged sequence to $\Omega(\delta)$. Moreover, due to the lower bound of [10], we are not actually able to obtain a perfectly sorted version of A and we are forced to work with a permutation of A having dislocation $d = \Omega(\log n)$, implying that the natural bound on the resulting dislocation can be as large as $d \cdot \delta$. This is bad news, as one can show that $\delta = \Omega(\log n)$. However, it turns out that the number of elements in $S \setminus A$ whose positions lie in a $O(\log n)$-wide interval of A is still $O(\log n)$, w.h.p., implying that the final dislocation of A is just $O(\log n)$.

But how do we obtain the approximately sorted sequence A in the first place? We could just recursively apply the above strategy on the (unsorted) elements of A, except that this would cause a blow-up in the resulting dislocation due to the constant hidden by the big-O notation. We therefore interleave merge steps with invocations of (a modified version of) the sorting algorithm of [10], which essentially reduces the dislocation by a constant factor, so that the increase in the worst-case dislocation will be only an *additive* constant per recursive step.

An additional complication is due to the fact that we are not able to compute the exact ranks in A of the elements in $S \setminus A$. We therefore have to deal, once again, with approximations that are computed using the other main contribution of this paper: *noisy binary search trees*, whose key ideas are described in the following.

Noisy Binary Search

As a key ingredient of our approximate sorting algorithm, we need to *merge* an almost-sorted sequence with a set of elements, without any substantial increase in the final maximum dislocation. More precisely, given a sequence S with maximum dislocation d and an element $x \notin S$, we want to compute an *approximate rank* of x in S, i.e., a position that differs by $O(\max\{d, \log n\})$ from the position that x would occupy if the elements $S \cup \{x\}$ were perfectly sorted. This same problem has been solved optimally in $O(\log n)$ time in the easier case in which errors are *not persistent* and S is already sorted [9]. The idea of [9] is to locate the correct position of x using a binary decision tree: ideally each vertex v of the tree *tests* whether x appears to belong to a certain *interval* of S and, depending on the result, one of the children of v is considered next. Since these intervals become narrower as we move from the root towards the leaves (that are in a one-to-one correspondence with positions of S) we eventually discover the correct rank of x in S. In order to cope with failures, this process is allowed to *backtrack* when inconsistent comparisons are observed, thus repeating some of the comparisons involving ancestors of v. Moreover, to obtain the correct result with high probability, a logarithmic number of consistent comparisons with a leaf are needed before the algorithm terminates.

Notice how the above process relies on the fact that it is possible to gather more information on the true relative position of x by repeating a comparison multiple times (in fact, it is trivial to design a simple $O(\log^2 n)$-time algorithm in this error model). Unfortunately, this is no longer the case when errors are *persistent*. To overcome this problem we design a *noisy binary search tree* in which testing whether x belongs to the interval associated with a vertex v also causes the interval itself to *grow* thus ensuring that, in future tests involving v, x will always be compared with different elements. This, however, is a source of other difficulties: first, the intervals of the descendants of v also need to be suitably updated to account for the new elements in v's interval. Moreover, since intervals are now *dynamic*, it is possible for multiple tests on the same vertex to report different results even when no comparison errors occur: this is because an interval that did not initially contain x might eventually grow into one that does. Finally, since growing intervals need to overlap, one also has to be careful in avoiding repeated comparisons arising from *unrelated* vertices (i.e., vertices that are not in ancestor–descendant relation in the tree). We overcome these problems by using two search trees that initially comprise of disjoint intervals and ensure that all the vertices exhibiting the problematic behaviours discussed above will be confined to only one of the two trees: in some sense, we guarantee that one of two search trees will behave similarly to the one of [9], where leaves now represent groups of $O(\log n)$ positions in S.

1.2 Related work

Sorting with *persistent errors* has been studied in several works, starting from [5] who presented the first algorithm achieving optimal dislocation (matching lower bounds appeared only recently in [10]) by finding a maximum-likelihood permutation of the input elements given the observed errors. The algorithm in [5] uses only $O(n \log n)$ comparisons and is able to handle any constant comparison error probability $p \in (0, \frac{1}{2})$ (later improved to $p \leq \frac{1}{2} - \Omega(\frac{\log\log n}{\log n})^{\frac{1}{6}}$ in [17]), but unfortunately its running time $O(n^{3+c_p})$ is quite large. For example, if we require the algorithm to succeed with a probability of $1-1/n$, the analysis in [5] yields $c_p = \frac{110525}{(1/2-p)^4}$. On the contrary, all subsequent faster algorithms [10,11,13] – see Table 1 – use a number of comparisons which is asymptotically equal to their respective running time and work for a smaller range of values of p (i.e., $p \leq \frac{1}{20}$ in [13] and $p < \frac{1}{16}$ in [10,11]).

Other works considered error models in which repeating comparisons is possible, although expensive. For example, [4] studied algorithms which use a *bounded number of rounds* for some "easier" versions of sorting (e.g., distinguishing the top k elements from the others). Note that each round consists of a set of comparison operations, where it is possible to compare the same pair of elements several times using independent comparisons like in the non-persistent model; Also, the comparisons made in each round are decided a priori, i.e., they do not depend on the results of the comparisons in this round. In each round, a fresh set of comparison results is generated, and each round consists of $\delta \cdot n$ comparisons. They evaluate the algorithm's performance by estimating the number of "misclassified" elements and also consider a variant in which errors now correspond to missing comparison results.

In general, sorting in presence of errors seems to be computationally more difficult than the error-free counterpart. For instance, [1] provides algorithms using *subquadratic* time (and number of comparisons) when errors occur only between elements whose difference is at most some fixed threshold. Also, [8] gives a *subquadratic* time algorithm when the number k of errors is known in advance.

As mentioned above, an easier error model is the one with *non-persistent* errors, meaning that the same comparison can be *repeated* and the errors are independent, and happen with some probability $p < 1/2$. In this model it is possible to sort n elements in time $O(n \log(n/q))$, where $1 - q$ is the success probability of the algorithm [9] (see also [2,12] for the analysis of the classical Quicksort and recursive Mergesort algorithms in this error model).

More generally, computing with errors is often considered in the framework of a two-person game called *Rényi-Ulam Game*. In this game a questioner tries to identify an unknown object x from a universe U by asking yes-or-no questions to a responder, but some of the answers might be wrong. The case in which $U = \{1, \dots, n\}$, the questions are of the form "is $s > x$?", and each answer is independently incorrect with probability $p < \frac{1}{2}$ has been considered by [9], where the authors provide a binary search algorithm that succeeds with probability $1 - \delta$ and requires $O(\log \frac{n}{\delta})$ worst-case time. In [3], the authors then showed how to find s using an optimal amount of queries up to additive polylogarithmic terms. The variant in which responder is allowed to adversarially lie up to k times has been proposed by Rényi [15] and Ulam [18], which has then been solved by Rivest et al. [16] using only $\log n + k \log logn + O(k \log k)$ question, which is tight. Among other results, near-optimal strategies for the distributional version of the game have been devised in [7]. For more related results on the topic, we refer the interested reader to [14] for a survey and to [6] for a monograph.

1.3 Paper Organization

The rest of this work is organized as follows: in Section 2 we give some preliminary definitions; then, in Section 3, we present our noisy binary search algorithm, which will be used in Section 4 to design an optimal randomized sorting algorithm. Finally, in Section 5, we briefly argue on how our sorting algorithm can be adapted so that it does not require any external source of randomness. Due to space limitations, this manuscript only includes the core parts of the analysis of our sorting algorithm. We refer the reader to the full version of the paper for the formal analysis of other claims of Section 4, and of the results in the remaining sections. Moreover, Section 4 makes use of an improved analysis of the sorting algorithm of [10] which can also be found in the full version of the paper.

2 Preliminaries

According to our error model, elements possess a true total linear order, however this order can only be observed through noisy comparisons. In the following, given two distinct elements x and y, we will write $x \prec y$ (resp. $x \succ y$) to mean that x is smaller (resp. larger) than y according to the true order, and $x < y$ (resp. $x > y$) to mean that x appears to be smaller (resp. larger) than y according to the observed comparison result.

Given a sequence or a set of elements A and an element x (not necessarily in A), we define $\text{rank}(x, A) = |\{y \in A : y \prec x\}|$ as the *true rank* of element x in A (notice that ranks start from 0). Moreover, if A is a sequence and $x \in A$, we denote by $\text{pos}(x, A) \in [0, |A| - 1]$ the *position* of x in A (notice that positions are also indexed from 0), so that the *dislocation* of x in A is $\text{disl}(x, A) = |\text{pos}(x, A) - \text{rank}(x, A)|$, and the *maximum dislocation* of the sequence A is $\text{disl}(A) = \max_{x \in A} \text{disl}(x, A)$.

For $z \in \mathbb{R}^+$, $\ln z$ and $\log z$ refer to the natural and the binary logarithm of z, respectively.

3 Noisy Binary Search

Given a sequence $S = \langle s_0, \dots, s_{n-1} \rangle$ of n elements with maximum dislocation $d \geq \log n$, and an additional element x not in S, we want to compute in time $O(\log n)$ an *approximate rank* of x in S, that is, a position where to insert x in S while preserving a $O(d)$ upper bound on dislocation of the resulting sequence. More precisely, we want to compute index r_x such that $|r_x - \text{rank}(x, S)| = O(d)$, in presence of persistent comparison errors: Errors between x and the elements in S happen independently with probability p, and whether the comparison between x and an element $y \in S$ is correct or erroneous does not depend on the position of y in S, nor on the actual permutation of the sorted elements induced by their order in S (i.e., we are not allowed to pick the order of the elements in S as a function of the comparison errors involving x). We do not impose any restriction on the errors of comparisons that do not involve x.

In the following, we will show an algorithm that computes such a rank r_x in time $O(\log n)$. This immediately implies that $O(\log n)$ time also suffices to insert x into S so that the resulting sequence $\langle s_0, \dots, s_{r_x - 1}, x, s_{r_x}, s_{n-1} \rangle$ still has maximum dislocation $O(d)$.

▶ Remark 1. The $O(\log n)$ running time is asymptotically optimal for all $d = n^{1-\epsilon}$, for constant $\epsilon < 1$, since a $\Omega(\log n - \log d) = \Omega(\log n)$ decision-tree lower bound holds even in absence of comparison errors.

In the following, for the sake of simplicity, we let $c = 10^3$ and we assume that $n = 2cd \cdot 2^h - 1$ for some non-negative integer h. Moreover, we focus on $p \le \frac{1}{32}$ even though this restriction can be easily removed to handle all constant $p < \frac{1}{2}$, as we argue at the end of the section.

We consider the set $\{0, \dots, n\}$ of the possible ranks of x in S and we subdivide them into $2 \cdot 2^h$ ordered *groups* $g_0, g_1, \dots$ each containing *cd* contiguous positions, namely, group g_i contains positions *cid*, …, $c(i+1)d - 1$. Then, we further partition these $2 \cdot 2^h$ groups into two ordered sets G_0 and G_1, where G_0 contains the groups g_i with even i ($i \equiv 0 \pmod 2$) and G_1 the groups g_i with odd i ($i \equiv 1 \pmod 2$). Notice that $|G_0| = |G_1| = 2^h$. In the next section, for each G_j, we shall define a *noisy binary search tree* T_j, which will be the main ingredient of our algorithm.

3.1 Constructing T_0 and T_1

Let us consider a fixed $j \in \{0, 1\}$ and define $\eta = 2\lceil \log n \rceil$. The tree T_j comprises of a binary tree of height $h + \eta$ in which the first $h + 1$ levels (i.e., those containing vertices at depths 0 to h) are complete and the last η levels consists of 2^h paths of η vertices, each emanating from a distinct vertex on the $(h+1)$-th level. We index the leaves of the resulting tree from 0 to $2^h - 1$, we use $h(v)$ to denote the depth of vertex v in T_j, and we refer to the vertices v at depth $h(v) \ge h$ as *path-vertices*. Each vertex v of the tree is associated with one *interval* $I(v)$, i.e., as a set of contiguous positions, as follows: for a leaf v having index i, $I(v)$ consists of the positions in g_{2i+j}; for a non-leaf path-vertex v having u as its only child, we set $I(v) = I(u)$; finally, for an internal vertex v having u and w as its left and right children, respectively, we define $I(v)$ as the interval containing all the positions between $\min I(u)$ and $\max I(w)$, endpoints included.

Moreover, each vertex v of the tree has a reference to two *shared pointers* $L(v)$ and $R(v)$ to positions in $\mathbb{Z} \setminus \bigcup_{g_i \in G_j} g_i$. Intuitively, $L(v)$ (resp. $R(v)$) will always point to positions of S occupied by elements that are *smaller* (resp. *larger*) than all the elements s_i with $i \in I(v)$. For each leaf v, let $L(v)$ initially point to $\min I(v) - d - 1$ and $R(v)$ initially point to $\max I(v) + d$. A non-leaf path-vertex v shares both its pointers with the corresponding pointers of its only child, while a non-path vertex v shares its left pointer $L(v)$ with the left pointer of its left child, and its right pointer $R(v)$ with the right pointer of its right child. See Figure 1 for an example.

Notice that we sometimes allow $L(v)$ to point to negative positions and $R(v)$ to point to positions that are larger than $n - 1$. In the following we consider all the elements s_i with $i < 0$ (resp. $i \ge n$) to be copies a special $-\infty$ (resp. $+\infty$) element such that $-\infty \prec x$ and $-\infty < x$ in every observed comparison (resp. $+\infty \succ x$ and $+\infty > x$).

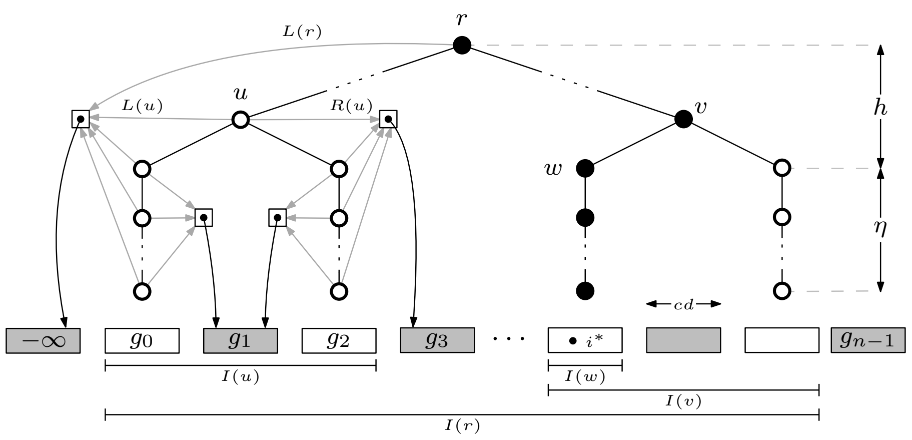

Figure 1 An example of the noisy tree T_0. On the left side the shared pointers $L(\cdot)$ and $R(\cdot)$ are shown. Notice how $L(r)$ (and, in general, all the $L(\cdot)$ pointers on the leftmost side of the tree) points to the special $-\infty$ element. Good (resp. bad) vertices are shown in black (resp. while). Notice that, since $i^* \in I(w)$, $T^* = T_0$ and all the depicted vertices are either good or bad.

3.2 Walking on T_j

The algorithm will perform a discrete-time random walk on each T_j. Before describing such a walk in more detail, it is useful to define the following operation:

▶ **Definition 2** (test operation). *A* test *of an element x with a vertex v is performed by (i) comparing x with the elements $s_{L(v)}$ and $s_{R(v)}$, (ii) decrementing $L(v)$ by 1 and, (iii) incrementing $R(v)$ by 1. The tests succeeds if the observed comparison results are $x > s_{L(v)}$ and $x < s_{R(v)}$, otherwise the test fails.*

The walk on T_j proceeds as follows. At time 0, i.e., before the first step, the *current* vertex v coincides with the root r of T_j. Then, at each time step, we *walk* from the current vertex v to the next vertex as follows:

1. We test x with all the children of v and, if *exactly one* of these tests succeeds, we *walk to the corresponding child.*
2. Otherwise, if *all the tests fail*, we *walk to the parent* of v, if it exists.

In the remaining cases we "walk" from v to itself. We also define $\tau = 240\lfloor \log n \rfloor$ and we stop the walk as soon as one of the following two conditions is met:
Success: The current vertex v is a leaf of T_j. In this case we say that the walk *returns* v;
Timeout: The τ-th time step is completed and the success condition is not met.
It turns out that at least one of the walks on T_0 and T_1 will succeed w.h.p., while the other can either succeed or timeout. If any of the walks succeeds and returns v, we output any position in the interval $I(v)$. Otherwise, we return an arbitrary position. We are then to prove the following result, whose analysis can be found in the full version of the paper:

▶ **Theorem 3.** *Let S be a sequence of n elements having maximum dislocation at most $d \geq \log n$ and let $x \notin S$. Under our error model, an index $r_x \in [\text{rank}(x, S) - \alpha d, \text{rank}(x, S) + \alpha d]$ can be found in $O(\log n)$ time with probability at least $1 - O(n^{-6})$, where $\alpha > 1$ is an absolute constant.*

To conclude this section, we remark that our assumption that $p \leq \frac{1}{32}$ can be easily relaxed to handle any constant error probability $p < \frac{1}{2}$. This can be done by modifying the test operation so that, when x is tested with a vertex v, the majority result of the comparisons between x and the set $\{s_{L(v)}, s_{L(v)-1}, \ldots, s_{L(v)-k+1}\}$ (resp. x and the set $\{s_{R(v)}, s_{R(v)+1}, \ldots, s_{R(v)+k-1}\}$) of η elements is considered, where k is a constant that only depends on p. Consistently, the pointers $L(v)$ and $R(v)$ are shifted by k positions, and the group size is increased to $k \cdot c$. Notice how our description for $p \le \frac{1}{32}$ corresponds exactly to the case $k = 1$. The only difference in the statement Theorem 3 is that α is no longer an absolute constant, but rather, it depends (only) on the value of p.

4 Optimal Sorting Algorithm

4.1 The algorithm

Here we present an optimal sorting algorithm that, given a sequence S of n elements, computes, in $O(n \log n)$ worst-case time, a permutation of S having maximum dislocation $O(\log n)$ and total dislocation $O(n)$, w.h.p. In order to avoid being distracted by roundings, we assume that n is a power of two.[3] Our algorithm will make use of the noisy binary search of Section 3 and of the `WindowSort` algorithm [10]. For our purposes, we need the following *stronger* version of the original result in [10], in which the bound on the total dislocation was only given in expectation:

▶ **Theorem 4.** *Consider a set of n elements that are subject to random persistent comparison errors. For any dislocation d, and for any (adversarially chosen) permutation S of these elements such that* $\mathrm{disl}(S) \le d$, *`WindowSort`(S, d) requires $O(nd)$ worst-case time to compute, with probability at least $1 - \frac{1}{n^4}$, a permutation of S having maximum dislocation at most $c_p \cdot \min\{d, \log n\}$ and total dislocation at most $c_p \cdot n$, where c_p is a constant depending only on the error probability $p < \frac{1}{32}$.*

We give a brief description of `WindowSort` and prove the above theorem in Section 5 of the full version of the paper. Notice that `WindowSort` also works in a stronger error model in which S can be chosen adversarially after the comparison errors between all pairs of elements have been randomly fixed, as long as its maximum dislocation is at most d. In the remaining of this section, we assume $p \le 1/32$ in order to be consistent with Section 3, though both the above theorem and the algorithm we are going to present will only require $p < 1/16$.[4] Using the noisy binary search in Section 3, we now define an operation that allows us to add a linear number of elements to an almost-sorted sequence without any asymptotic increase in the resulting dislocation, as we will formally prove in the sequel. More precisely, if A and B are two disjoint subsets of S, we denote by `Merge`(A, B) the sequence obtained as follows:

- For each $x \in B$, compute an index r_x such that $|\mathrm{rank}(s, A) - r_x| \le \alpha d$. This can be done using the noisy binary search of Section 3, which succeeds with probability at least $1 - \frac{1}{|A|^6}$.
- Insert *simultaneously* all the elements $x \in B$ into A in their computed positions r_x, breaking ties arbitrarily. Return the resulting sequence.

Our sorting algorithm, that we call `RiffleSort` (see the pseudocode in Algorithm 1), works as follows. For $k = \frac{\log n}{2}$, we first partition S into $k+1$ subsets $T_0, T_1, \ldots, T_k$: Each T_i, with $1 \le i \le k$, contains $2^{i-1}\sqrt{n}$ elements chosen uniformly at random from $S \setminus \{T_{i+1}, T_{i+2}, \ldots, T_k\}$, and $T_0 = S \setminus \{T_1, T_2, \ldots, T_k\}$ contains the remaining $n - \sqrt{n}\sum_{i=1}^{k} 2^{i-1} = \sqrt{n}$ elements. As its

[3] This assumption can be easily removed by adding dummy $+\infty$ elements to S. Since `WindowSort`, the noisy binary search of Section 3, and ultimately our algorithm will also work when p is an upper bound on the error probability, it is not necessary to simulate errors when comparisons involving dummy elements are performed.

[4] In fact, Theorem 4 is the only reason preventing our novel sorting algorithm to work for any constant $p \in [0, \frac{1}{2})$.

Algorithm 1 `RiffleSort(S)`.

```
1 $T_0, T_1, \dots, T_k \leftarrow$ partition of $S$ computed as explained in Section 4.1;
2 $S_0 \leftarrow$ WindowSort($T_0, \sqrt{n}$);
3 foreach $i = 1, \dots, k = \frac{\log n}{2}$ do
4     $S_i \leftarrow$ Merge($S_{i-1}, T_i$);
5     $S_i \leftarrow$ WindowSort($S_i, \gamma \cdot c_p \cdot \log n$);
6 return $S_k$;
```

first step, `RiffleSort` will approximately sort T_0 using `WindowSort`, and then it will alternate merge operations with calls to `WindowSort`. On one hand the merge operations allow us to iteratively grow the set of approximately sorted elements to ultimately include all the elements in S but, on the other hand, each operation also increases the dislocation by a constant factor. This is a problem since the rate at which the dislocation increases is faster than the rate at which new elements are inserted. The role of the sorting operations is exactly to circumvent this issue: each `WindowSort` call has the effect of locally rearranging the elements, so that all newly inserted elements are now closer to their intended positions, causing (an upper bound to) the resulting maximum dislocation to increase by only an additive constant. The corresponding pseudocode is shown in Algorithm 1, in which $\gamma \geq \max\{202\alpha, 909\}$ is an absolute constant (recall that α is the constant from Theorem 3).

4.2 Analysis

▶ **Lemma 5.** *The worst-case running time of Algorithm 1 is $O(n \log n)$.*

Proof. Clearly the random partition $T_0, \dots, T_k$ can be computed in time $O(n \log n)$,[5] and the first call to `WindowSort` requires time $O(|T_0| \cdot \sqrt{n}) = O(n)$ (see Theorem 4). We can therefore restrict our attention to the generic i-th iteration of the for loop. The call to `Merge`(S_{i-1}, T_i) can be performed in $O(|S_i| \log n)$ time since, for each $x \in T_i$, the required approximation of $\text{rank}(x, S_{i-1})$ can be computed in time $O(\log |S_{i-1}|)$ and $|T_i| = |S_{i-1}| \leq n$, while inserting the elements of T_i in their computed ranks requires linear time in $|S_{i-1}| + |T_i| = |S_i|$. The subsequent execution of `WindowSort` with $d = O(\log n)$ requires time $O(|S_i| \log n)$, where the hidden constant does not depend on i. Therefore, for a suitable constant c, the time spent in the i-th iteration is $c|S_i| \log n$ and total running time of Algorithm 1 can be upper bounded by:

$$c \sum_{i=1}^{k} |S_i| \log n = c\sqrt{n} \log n \cdot \sum_{i=1}^{k} 2^i < 2^{k+1} c\sqrt{n} \log n = 2cn \log n. \qquad ◀$$

The following lemma, that concerns a thought experiment involving urns and randomly drawn balls, will be useful to upper bound the dislocation of the sequences returned by the `Merge` operations. Since it can be proved using arguments that do not depend on the details of `RiffleSort`, we omit its proof, which can be found in the full version of the paper.

[5] The exact complexity depends on whether we can sample u.a.r. an integer from a range in $O(1)$ time. If this is not the case, then integers can be generated bit-by-bit using rejection, and the total number of required random bits will be $O(n)$ with probability at least $1 - n^{-2}$, as shown in the full version of this paper. To maintain a worst-case upper bound on the running time also in the unlikely event that $\Theta(n \log n)$ bits do not suffice, we can stop the algorithm and return any arbitrary permutation of S.

▶ **Lemma 6.** *Consider an urn containing $N = 2M$ balls, M of which are white, while the remaining M are black. Balls are iteratively drawn from the urn without replacement until the urn is empty. If N is sufficiently large and $9 \log N \le k \le \frac{N}{16}$ holds, the probability that any contiguous subsequence of at most $100k$ drawn balls contains k or fewer white balls is at most N^{-6}.*

We can now show that, if A and B contain randomly selected elements, the dislocation of `Merge`(A, B) is likely to be at most a constant factor larger than the dislocation of A:

▶ **Lemma 7.** *Let A be a sequence containing m randomly chosen elements from S and having maximum dislocation at most d, with $\log n \le d = o(m)$. Let B be a set of m randomly chosen elements from $S \setminus A$. Then, for a suitable constant γ, and for large enough values of m, merge(A, B) has maximum dislocation at most γd with probability at least $1 - m^{-4}$.*

Proof. Let $\beta = \max\{\alpha, 9/2\}$, $S' =$ `Merge`(A, B), and $S^* = \langle s^*_0, s^*_1, \dots, s^*_{2m-1}\rangle$ be the sequence obtained by sorting S' according to the true order of its elements. Assume that:

- all the approximate ranks r_x, for $x \in B$, are such that $|r_x - \text{rank}(x, A)| \le \beta d$; and
- all the contiguous subsequences of S^* containing no more than $2\beta d + 2$ elements in A have length at most $200\beta d + 200$.

We will show in the sequel that the above assumptions are likely to hold.

Pick any element $x \in S'$. We now show that our assumptions imply that the dislocation of x in S' is at most $201d$. An element $y \in B$ can affect the final dislocation of x in S' only if one of the following two (mutually exclusive) conditions holds: (i) $y \prec x$ and $r_y \ge r_x$, or (ii) $y \succ x$ and $r_y \le r_x$. All the remaining elements in B will be placed in the correct relative order w.r.t. x in S', and hence they do not affect the final dislocation of x. If (i) holds, we have:

$$r_x - \beta d \le r_y - \beta d \le \text{rank}(y, A) \le \text{rank}(x, A) \le r_x + \beta d,$$

while, if (ii) holds, we have:

$$r_x - \beta d \le \text{rank}(x, A) \le \text{rank}(y, A) \le r_y + \beta d \le r_x + \beta d,$$

and hence, all the elements $y \in B$ that can affect the dislocation of x in S' are contained in the set $Y = \{y \in B : r_x - \beta d \le \text{rank}(y, A) \le r_x + \beta d\}$.

We now upper bound the cardinality of Y. Let y^- be the $(r_x - \beta d - 1)$-th element of A; if no such element exists, then let $y^- = s^*_0$. Similarly, let y^+ be the $(r_x + \beta d)$-th element of A; if no such element exists, then let $y^+ = s^*_{2m-1}$. Due to our choice of y^- and y^+ we have that $\forall y \in Y, y^- \preceq y \preceq y^+$, implying that all the elements in Y appear in the contiguous subsequence $\overline{S}$ of S^* having y^- and y^+ as its endpoints. Since no more than $2\beta d + 2$ elements of A belong to $\overline{S}$, our assumption guarantees that $\overline{S}$ contains at most $200\beta d + 200$ elements. This implies that the dislocation of x in S' is at most $\beta d + |Y| \le \beta d + |\overline{S}| \le 201\beta d + 200 \le \gamma d$, where the last inequality holds for large enough n once we choose $\gamma = 202\beta$.

To conclude the proof we need to show that our assumptions hold with probability at least $1 - |S'|^{-4}$. Regarding the first assumption, for $x \in B$, a noisy binary search returns a rank r_x such that $|r_x - \text{rank}(x, A)| \le \alpha d \le \beta d$ with probability at least $1 - O(\frac{1}{m^6})$. Therefore the probability that the assumption holds is at least $1 - O(\frac{1}{m^5})$.

Regarding our second assumption, notice that, since the elements in A and B are randomly selected from S, we can relate their distribution in S^* with the distribution of the drawn balls in the urn experiment of Lemma 6: the urn contains $N = 2m$ balls each corresponding to an element in $A \cup B$, a ball is white if it corresponds to one of the $M = m$ elements of A, while a black ball corresponds one of the $M = m$ elements of B. If the assumption does not hold, then there exists a contiguous subsequence of S^* of at least $200\beta d + 200$ elements that contains at most $2\beta d + 2$ elements from A. By Lemma 6 with $k = 2\beta d + 2$, this happens with probability at most $(2m)^{-6}$ (for sufficiently large values of n). The claim follows by using the union bound. ◀

We can now use Lemma 7 and Theorem 4 together to derive an upper bound to the final dislocation of the sequence returned by Algorithm 1.

▶ **Lemma 8.** *The sequence returned by Algorithm 1 has maximum dislocation $O(\log n)$ and total dislocation $O(n)$ with probability at least $1 - \frac{1}{n\sqrt{n}}$.*

Proof. For $i = 1, \ldots, k$, we say that the i-th iteration of Algorithm 1 is *good* if the sequence S_i computed at its end has both (i) maximum dislocation at most $c_p \log n$, and (ii) total dislocation at most $c_p|S_i|$. As a corner case, we say that iteration 0 is good if S_0 also satisfies conditions (i) and (ii) above, which happens with probability at least $1 - \frac{1}{|S_0|^4} \geq 1 - \frac{1}{n^2}$ as shown by Theorem 4.

We now focus on a generic iteration $i \geq 1$ and show that, assuming that iteration $i-1$ is good, iteration i is also good with probability at least $1 - \frac{1}{n^2}$. Since iteration $i-1$ was good, the sequence S_{i-1} has maximum dislocation $c_p \log n$ and hence, by Lemma 7, the sequence resulting from call to `Merge`(S_{i-1}, T_i) returns a sequence with dislocation at most $\gamma c_p \log n$ with probability at least $1 - \frac{1}{|T_i|^4} \geq 1 - \frac{1}{n^2}$. If this is indeed the case, we have that the sequence S_i returned by the subsequent call to `WindowSort` satisfies (i) and (ii) with probability at least $1 - \frac{1}{|S_{i+1}|^4} \geq 1 - \frac{1}{n^2}$ (see Theorem 4). The claim follows by using the union bound on the $k = O(\log n)$ iterations, and by noticing that the returned sequence is exactly S_k. ◀

We can therefore state the following result, which follows directly from Lemma 8 and Lemma 5:

▶ **Theorem 9.** *Consider a set of n elements that are subject to random persistent comparison errors. For any (adversarially chosen) input permutation of these elements, `RiffleSort` is a randomized algorithm that returns, in $O(n \log n)$ worst-case time, a sequence having maximum (resp. total) dislocation $O(\log n)$ (resp. $O(n)$), w.h.p.*

5 Derandomization

In Section 4 we showed how it is possible to design a *randomized* algorithm that approximately sorts a sequence S of n elements achieving simultaneously asymptotically optimal maximum dislocation, total dislocation, and running time, w.h.p.[6] In this section we sketch how `RiffleSort` can be adapted to obtain a *deterministic* algorithm with the same asymptotic guarantees on running time, dislocation, and success probability (over the comparison errors), as long as the order of the elements in S does not depend of the comparison errors.[7]

In order to run `RiffleSort`, we need to partition the input sequence S into a collection of random sets $T_0, T_1, \ldots, T_k$ where $k = \frac{\log n}{2}$ and each T_i contains $m = \sqrt{n} \cdot 2^{i-1}$ elements that are chosen uniformly at random from the $n - \sqrt{n} \sum_{j=i+1}^{k} 2^{i-1} = 2m$ elements in $S \setminus \bigcup_{j=i+1}^{k} T_j$.

[6] The *randomized* result also holds when each comparison c has an adversarially chosen and unknown probability of error $p_c \in [0, p]$. The deterministic result holds if $p_c \in [p_0, p]$ for some constant $p_0 > 0$.

[7] An adversary could make the algorithm fail by first observing all comparison results among the input elements, and then choosing a suitable input permutation S. In other words, our result holds if the comparison errors do not depend on the element values nor on the positions in S of the involved elements.

Notice also that this is the only step in the algorithm that is randomized. To obtain a version of `RiffleSort` that does not require any external source of randomness, i.e., that depends only on the input sequence and on the comparison results, we will generate such a partition by exploiting the intrinsic random nature of the comparison results. As shown in the full version of the paper, with probability at least $1 - \frac{1}{n^3}$, the partition $T_0, \ldots, T_k$ can be found in $O(n)$ time using only $6n$ random bits. Moreover, with a technique similar to that of [11], it is possible to simulate "almost-fair" coin flips by xor-ing together a sufficiently large number of comparison results. Indeed, we can associate the two possible results of a comparison with the values 0 and 1, so that each comparison behaves as a Bernoulli random variable whose (unknown) parameter is either p or $1 - p$. We can then use the following fact: let $c_1, \ldots, c_k$ be $k = \Theta(\log n)$ independent Bernoulli random variables such that $P(c_i = 1) \in \{p, 1-p\}$ $\forall i = 1, \ldots, k$, then $|\Pr(c_1 \oplus c_2 \oplus \cdots \oplus c_k = 0) - \frac{1}{2}| \le \frac{1}{n^4}$. Therefore, if we consider the set A containing the first $7k$ elements from S and we compare each element in A to all the elements in $S \setminus A$, we obtain a collection of $7k(n - 7k) \ge 6kn$ comparison results (for sufficiently large values of n) from which we can generate $6n$ almost-fair coin flips. A coupling argument shows that, with probability at least $1 - \frac{6kn}{n^4} - \frac{1}{n^3} > 1 - \frac{1}{n^2}$, all these almost-fair coin flips behave exactly as unbiased random bits, and they suffice to select a partition $T_0, \ldots, T_k$ of $S \setminus A$. It is now possible to use `RiffleSort` on $S \setminus A$ to obtain a sequence S' having maximum dislocation $d = O(\log n)$ and total dislocation $O(n)$. This requires time $O(n \log n)$ and succeeds with probability at least $1 - |S \setminus A|^{-\frac{3}{2}} > 1 - 3n^{-\frac{3}{2}}$ since $|S \setminus A| \ge \frac{n}{2}$.

What is left to do is to reinsert all the elements of A into S' without causing any asymptotic increase in the total and in the maximum dislocation. While one might be tempted to use the result of Section 1, this is not actually possible since the errors between the elements in A and the elements in S' now depend on the permutation S'. In the full version of this work we show a simple, but slower, $O(n)$-time strategy to compute $\text{rank}(x, S')$ with an additive error of $O(\log n)$, even when S' is adversarially chosen as a function of the errors. Since A contains $O(\log n)$ elements, simultaneously reinserting them in S' affects the maximum dislocation by at most an $O(\log n)$ additive term, while their combined contribution to the total dislocation is at most $O(\log^2 n)$. We summarize the discussion of this section in the following theorem:

▶ **Theorem 10.** *Consider a set of n elements that are subject to random persistent comparison errors. For any input permutation of these elements that is chosen independently of the errors, there exists a deterministic algorithm that returns, in $O(n \log n)$ worst-case time, a sequence having maximum (resp. total) dislocation $O(\log n)$ (resp. $O(n)$), w.h.p.*

References

1 Miklós Ajtai, Vitaly Feldman, Avinatan Hassidim, and Jelani Nelson. Sorting and selection with imprecise comparisons. *ACM Transactions on Algorithms*, 12(2):19, 2016.

2 Laurent Alonso, Philippe Chassaing, Florent Gillet, Svante Janson, Edward M Reingold, and René Schott. Quicksort with unreliable comparisons: a probabilistic analysis. *Combinatorics, Probability and Computing*, 13(4-5):419–449, 2004.

3 Michael Ben-Or and Avinatan Hassidim. The Bayesian Learner is Optimal for Noisy Binary Search (and Pretty Good for Quantum as Well). In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 221–230, 2008. `doi:10.1109/FOCS.2008.58`.

4 Mark Braverman, Jieming Mao, and S Matthew Weinberg. Parallel algorithms for select and partition with noisy comparisons. In *Proc. of the 48th Annual ACM Symposium on Theory of Computing (STOC)*, pages 851–862. ACM, 2016.

5 Mark Braverman and Elchanan Mossel. Noisy Sorting Without Resampling. In *Proceedings of the 19th Annual Symposium on Discrete Algorithms*, pages 268–276, 2008. arXiv:0707.1051.

6 Ferdinando Cicalese. *Fault-Tolerant Search Algorithms - Reliable Computation with Unreliable Information.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2013.

7 Yuval Dagan, Yuval Filmus, Daniel Kane, and Shay Moran. The entropy of lies: playing twenty questions with a liar. *CoRR*, abs/1811.02177, 2018. arXiv:1811.02177.

8 Peter Damaschke. The Solution Space of Sorting with Recurring Comparison Faults. In *Combinatorial Algorithms - 27th International Workshop, IWOCA 2016, Helsinki, Finland, August 17-19, 2016, Proceedings*, pages 397–408, 2016.

9 Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with Noisy Information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994. doi:10.1137/S0097539791195877.

10 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Sorting with Recurrent Comparison Errors. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation (ISAAC 2017)*, volume 92 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ISAAC.2017.38.

11 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal Dislocation with Persistent Errors in Subquadratic Time. In *Proc. of the 35th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 96 of *LIPIcs*, pages 36:1–36:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.STACS.2018.36.

12 Petros Hadjicostas and KB Lakshmanan. Recursive merge sort with erroneous comparisons. *Discrete Applied Mathematics*, 159(14):1398–1417, 2011.

13 Rolf Klein, Rainer Penninger, Christian Sohler, and David P. Woodruff. Tolerant Algorithms. In *Proc. of the 19th Annual European Symposium on Algorithm (ESA)*, pages 736—-747, 2011.

14 Andrzej Pelc. Searching games with errors - fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1-2):71–109, 2002.

15 Alfréd Rényi. On a problem of information theory. *MTA Mat. Kut. Int. Kozl. B*, 6:505–516, 1961.

16 Ronald L. Rivest, Albert R. Meyer, Daniel J. Kleitman, Karl Winklmann, and Joel Spencer. Coping with Errors in Binary Search Procedures. *J. Comput. Syst. Sci.*, 20(3):396–404, 1980. doi:10.1016/0022-0000(80)90014-8.

17 Aviad Rubinstein and Shai Vardi. Sorting from Noisier Samples. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 960–972, 2017. doi:10.1137/1.9781611974782.60.

18 Stanislav M. Ulam. Adventures of a Mathematician, 1976.