Optimal Dislocation with Persistent Errors in Subquadratic Time

Barbara Geissmann
Department of Computer Science, ETH Zürich, Switzerland

Stefano Leucci
Department of Computer Science, ETH Zürich, Switzerland

Chih-Hung Liu
Department of Computer Science, ETH Zürich, Switzerland

Paolo Penna
Department of Computer Science, ETH Zürich, Switzerland

Abstract

We study the problem of sorting N elements in presence of *persistent* errors in comparisons: In this classical model, each comparison between two elements is wrong independently with some probability p, but repeating the same comparison gives always the same result. The best known algorithms for this problem have running time $O(N^2)$ and achieve an optimal maximum dislocation of $O(\log N)$ for constant error probability. Note that no algorithm can achieve dislocation $o(\log N)$, regardless of its running time.

In this work we present the first *subquadratic* time algorithm with optimal maximum dislocation: Our algorithm runs in $\widetilde{O}(N^{3/2})$ time and guarantees $O(\log N)$ maximum dislocation with high probability. Though the first version of our algorithm is randomized, it can be derandomized by extracting the necessary random bits from the results of the comparisons (errors).

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

Keywords and phrases Sorting, Recurrent Comparison Errors, Maximum Dislocation

Digital Object Identifier 10.4230/LIPIcs.STACS.2018.36

1 Introduction

We study the problem of *sorting* N distinct elements under *recurrent* random comparison *errors*. In this classical model, each comparison is wrong with some fixed (small) probability p, and correct with probability $1 - p$. The probability of errors are independent over all possible pairs of elements, but errors are recurrent: Repeating the same comparison several times is useless since the result is always the same, i.e., always wrong or always correct.

Because of errors, different sorting algorithms can have different guarantees to output a "nearly sorted" sequence. To measure the quality of an output sequence in terms of sortedness, a common way 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, one can consider the *maximum dislocation* of any element in the permutation or the *total dislocation* of a permutation, i.e., the sum of the dislocations of all n elements. Of course, the running *time* is also an important criteria for evaluating sorting algorithms.

Regarding the *maximum dislocation* and the *running time*, in the recurrent random comparison errors, this is the state of the art:

- Several algorithms [3, 12, 9] guarantee *maximum dislocation* $O(\log N)$ with high probability, though their *running time* is *quadratic* or even *larger* (see Table 1).

Table 1 The running time of previous algorithms which guarantee $O(\log N)$ maximum dislocation (with high probability) and our result. The constant $c(p)$ in [3] depends on both the error probability p, and the success probability of the algorithm, and it is typically quite large. The algorithms in [12, 9] have different guarantees on the total dislocation.

	Braverman and Mossel [3]	Klein et al [12], Geissmann et al [9]	**This work**
Time	$O(N^{3+c(p)})$	$O(N^2)$	$\widetilde{O}(N^{3/2})$

- No algorithm (even randomized) can achieve *maximum dislocation* $o(\log N)$ with high probability, *regardless of its running time* [9].

This suggests naturally the following question:

> *Is there any algorithm with* ***subquadratic running time*** *which achieves* ***optimal maximum dislocation*** $O(\log n)$ *with high probability?*

In this paper we give an affirmative answer to this question.

1.1 Our contribution

We present the first *subquadratic time* algorithm with *optimal maximum dislocation*, namely, an algorithm that runs in $\widetilde{O}(N^{3/2})$ time and returns a sequence of maximum dislocation $O(\log N)$ with high probability (see Table 1). The latter is optimal because, in the model with *persistent* errors, no algorithm (even randomized) can achieve maximum dislocation $o(\log N)$ with high probability [9]. Intuitively speaking, our algorithm (Recursive Window Sort) first picks a random permutation and then performs a number of deterministic operations which use the algorithm in [9] as a subroutine. All recursive steps are *deterministic* and they consist of an algorithm that approximately sorts an input sequence whenever it is *well shuffled* and the errors are *well spread.* The latter condition holds with high probability in the error model we consider, and the starting random permutation serves to have a well shuffled input. The correctness of Recursive Window Sort combines a technical condition that the algorithm in [9] guarantees, combined with an intermediate "merge step" which works well on well shuffled inputs (see Section 2 for an high level description of the algorithm and the main ideas).

Though our first algorithm is *randomized*, it can be "derandomized" in the following sense. By using the results of the comparison errors, the algorithm itself can generate the necessary (almost) random bits to be used in the computation. Note that this is far from trivial for two reasons: (i) The outcome of the comparisons are also used during the computation and (ii) The result of a comparison may tell something about the result of another comparison. Our second major contribution is a *deterministic* algorithm (Derandomized Recursive Window Sort) which still runs in $\widetilde{O}(N^{3/2})$ and that returns a sequence of maximum dislocation $O(\log N)$ with high probability (over the random comparison errors).

Connections with prior work

The algorithm by Braverman and Mossel [3] constructs the maximum likehood permutation, whose computation requires a rather large (though polynomial) running time. Their method in fact uses only $O(N \log N)$ comparisons and is applicable to any $p < 1/2$, while the faster algorithms by Klein et al [12] and Geissmann et al [9] work for p smaller than some absolute small constant (e.g., in [9] $p < 1/16$).

Ajtai et al [1] provide algorithms using *subquadratic* time (and number of comparisons) when errors occur only between elements whose difference is at most some fixed threshold. Damaschke [6] gives also a *subquadratic* time algorithm, by assuming that at most k errors occur. The algorithm returns a sequence up to $O(k)$ inversions and it is based on finding a solution for the feedback arc set (FAST) problem. Coppersmith and Rurda [5] provide a simple algorithm 5-approximation for the weighted FAST problem, if the weights satisfy probability constraints.

An easier error model is the one with *non-recurrent* errors, meaning that the same comparison can be *repeated* and the errors are independent with some probability $p < 1/2$. For this model, Feige et al. [7] gave a sorting algorithm running in $O(N \log(N/q))$ steps, where $1 - q$ is the success probability of the algorithm. Alonso et al. [2] and Hadjicostas and Lakshamanan [11] studied the classical Quicksort and recursive Mergesort algorithms, respectively. Sorting by repeatedly performing random swaps results in Markovian processes which have been studied by Geissmann et al [8, 10].

Finally, computing with errors is often considered in the framework of a two-person game called *Rényi-Ulam Game* (see Pelc's survey [14] and Cicalese's monograph [4]).

1.2 Preliminaries

In this section, we describe the key features of the Window Sort algorithm [9] which will be used a a subroutine of our main algorithm (see Algorithm 1). To this end, we first introduce some notation used throughout the paper.

We consider the problem of sorting N distinct integers which, under the error model considered here, is equivalent to sort a sequence containing the integers $\{1, 2, \ldots, N\}$. For any sequence S, and any element x in the sequence, we define its *rank* as the number of elements smaller than x in S, i.e., $rank(x, S) \triangleq |\{y \in S | \; y < x\}|$. Note that this gives the correct position of x (its rank plus 1) in the correctly sorted sequence, and it only depends on the elements in S (not in the sequence order). In the following we will use $\kappa \geq 1$ to denote a global constant that only depends on the error probability p. For ease of presentation, we assume that $p < \frac{1}{32}$, although our algorithm can be adapted to work for $p < \frac{1}{16}$ (which is a condition needed to successfully run Window Sort). We say that a comparison between an (unordered) pair of elements x, y, with $x < y$, is an *error* if x is (incorrectly) reported to be *larger than* y.

▶ **Definition 1.** We define $ERRORS(x, w, S)$ as the set of errors among the comparisons between element x and every other element y in S with $rank(y, S) \in [rank(x, S) - 4w, rank(x, S) + 4w]$.

▶ **Definition 2.** For a set of elements S, we say that the comparison errors are *well spread* iff, for all $x \in S$ and for all w such that $\kappa \log |S| \leq w \leq n$, we have $|ERRORS(x, w, S)| \leq w/4$.

▶ **Definition 3** (Success). We say that $\langle S, W\rangle$, where S is a sequence and W a window size, satisfies the Success condition if

1. The maximum dislocation of S is at most W;
2. The comparison errors in S are *well spread*.

This condition guarantees that the output of Window Sort will have maximum dislocation $O(\log |S|)$, where the initial window size determines its running time:

▶ **Lemma 4** ([9]). Window Sort *on a sequence S with a starting window size W returns a sequence having maximum dislocation at most $\kappa \log n$ in $O(|S| \cdot W)$ time whenever $\langle S, W\rangle$*

Algorithm 1: Window Sort (on a sequence S of n distinct elements and initial window size W).

```
Initialization: The initial window size is $w = W$. Each element $x$ has two variables
  $wins(x)$ and $computed\_rank(x)$ which are set to zero.
repeat
    1. foreach $x$ at position $l = 1, 2, 3, \dots, n$ in $S$ do
           foreach $y$ whose position in $S$ is in $[l - 2w, l - 1]$ or in $[l + 1, l + 2w]$ do
               if $x > y$ then
                   $wins(x) = wins(x) + 1$
           $computed\_rank(x) = \max\{l - 2w, 0\} + wins(x)$
    2. Place the elements into $S'$ ordered by non-decreasing $computed\_rank$,
       break ties arbitrarily.
    3. Set all $wins$ to zero, $S = S'$, and $w = w/2$.
until $w < 1$;
```

satisfies the Success *condition. Moreover, the expected total dislocation of the returned sequence is $O(n)$.*

▶ **Lemma 5.** *For any sequence S of n elements chosen independently of the errors, the probability (over the comparison errors) that errors are well spread is at least $1 - 1/n^8$.*

2 Warm up

In this section we informally describe some of the ideas used in our algorithm. As a warm up, we consider a simplified (non-optimized) version which is described in Figure 1 and consists of the following steps:

1. Start with a *random permutation* S of the input sequence and split this sequence S into β blocks of the same size.[1]
2. Run Window Sort on each block B_i to obtain a sequence S_i.
3. Combine all the sequences S_i together into a sequence S' as follows: The first element in each S_i will be placed (in arbitrary order) in one of the first β positions of S', the second element in each S_i will be placed in a position between $\beta + 1$ and 2β in S', and so on.
4. Run Window Sort on this new sequence S'.

At this point two observations are in place. First, we did not specify yet some parameters, namely, the number β of blocks (and thus their size N/β), nor the initial window size when we call Window Sort. Both these parameters need to be chosen carefully in order to achieve the desired performance and, in our more complex scheme, they will vary at every recursive call. Second, the initial step where we pick a random permutation of the input (the elements to be sorted) can be implemented more efficiently by distributing directly these elements into the desired number of blocks.

Saving in the running time

One intuition why this scheme should be faster than Window Sort, is that in the first part this algorithm is called on smaller blocks and the running time is thus $\widetilde{O}(N^2/\beta)$, since each

[1] For the sake of simplicity, here and in the rest of the paper, we assume that $|S|$ is a multiple of the block size.

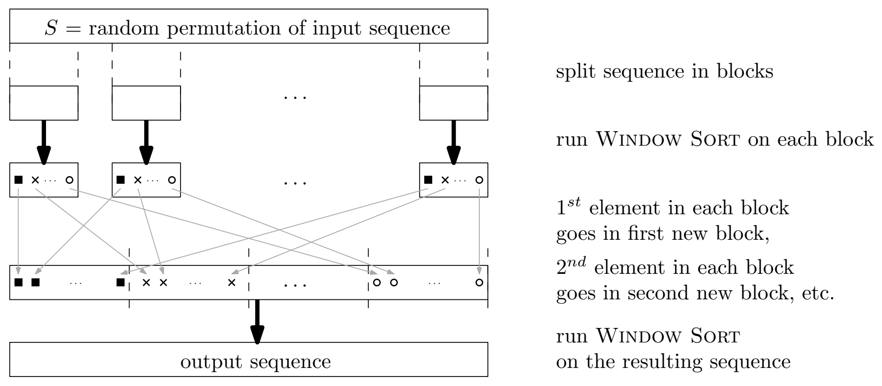
S = random permutation of input sequence
...
...
...
output sequence
split sequence in blocks
run Window Sort on each block
1st element in each block goes in first new block,
2nd element in each block goes in second new block, etc.
run Window Sort on the resulting sequence

Figure 1 The one-level recursion scheme.

block takes $\widetilde{O}((N/\beta)^2)$ time. The last call to Window Sort can be fast if S' has already a bounded maximum dislocation, since in this case we can start the algorithm with a small initial window size.

Need for initial random permutation

To see why we perform this initial step, consider the version in which we start with the sorted sequence, that is, $S = \langle 1, 2, \ldots, N \rangle$. We claim that, in this case, it is very likely that the sequence S' obtained after recombining the blocks has *large dislocation*. Indeed, suppose all the calls to Window Sort sort perfectly each block. Then, $S_1 = \langle 1, 2, \ldots, N/\beta \rangle$ and $S_2 = \langle N/\beta + 1, \ldots, 2N/\beta \rangle$, causing element 2 to be placed in S' in position at least $N/\beta + 1$.

Correctness argument

In order to get the desired bound on the maximum dislocation, we essentially need to show that every call to Window Sort on some *subsequence* of elements will be "successful", that is, the output will have a bounded maximum dislocation. Note that these calls are *not* independent since the results of the comparisons (which depend on the errors) determine the sequence S' in the last call to Window Sort. It turns out, that for any fixed subset of elements and any fixed window size that is large enough (i.e., logarithmic in N), this property holds with *high probability* (w.r.t. N). Moreover, the input sequence S' in the last call of Window Sort involves *all* elements, while the other β calls for the blocks involve randomly chosen subsets of elements (independent of the errors). As all these subsets are polynomially many (we choose β accordingly below), all these calls to Window Sort succeed with high probability too (union bound).

3 The algorithm

We now describe the full version of our algorithm, which we call Recursive Window Sort. Intuitively, our algorithm is a recursive version of the scheme described in Section 2 (see Figure 1), where the only randomized part is the initial shuffling of the input sequence, which is performed only once. We refer to this random permutation of the input sequence as S. All recursive steps are *deterministic* and they consist of an algorithm that approximately sorts an input sequence whenever it is *well shuffled* and the errors are *well spread.*

We next describe the recursive steps of Recursive Window Sort. We denote by N the total number of elements to sort. The behavior of our recursive algorithm varies according

Algorithm 2: RECURSIVE WINDOW SORT (on N distinct elements).

```
Let $S$ be a random permutation the $N$ input elements
Run RECURSIVE STEP on $S$ (with initial depth $d = 0$)
return the resulting sequence
```

Algorithm 3: RECURSIVE STEP (on a sequence S of n_d distinct elements at depth d).

```
Initialization: the maximum depth is $h = \log N$, the values $\beta_d$ and $W_d$ are chosen as in (1)
if $d = h$ then
    Run WINDOW SORT on $S' = S$ with window size $n_d$
    return the resulting sequence
else
    Partition $S$ into $b_d \triangleq \frac{n_d}{\beta_d}$ blocks $B_1, B_2, \dots, B_{b_d}$ each containing $\beta_d$ elements
    foreach block $B_i$ do
        Run RECURSIVE STEP on $B_i$ with depth $d+1$ to obtain $B'_i = \langle b'_{i,1}, b'_{i,2}, \dots, b'_{i,\beta_d} \rangle$
    foreach $j = 1, 2, \dots, \beta_d$ do
        $B''_j = \langle b'_{1,j}, b'_{2,j}, \dots, b'_{b_d,j} \rangle$
    Let $S' = \langle s'_1, s'_2, \dots, s'_{n_d} \rangle = \langle B''_1, B''_2, \dots, B''_{\beta_d} \rangle$
    Run WINDOW SORT on $S'$ with window size $W_d$
    return the resulting sequence
```

to the current depth of recursion. The maximum depth of the recursion is $h = \log N$.[2] In general, a recursive step at depth d sorts an input sequence S of n_d elements[3], by splitting S into blocks of size β_d and recursing on these blocks. Then, it recombines the elements from the blocks in a zip fastener fashion into a single sequence S', and runs WINDOW SORT on S' with window size W_d. We formally describe such a recursive step in Algorithm 3 and RECURSIVE WINDOW SORT in Algorithm 2.

In order to optimize the running time, we shall set the parameters as follows:

$$\beta_d \triangleq n_d^{1-\frac{1}{2^{h-d+1}-1}} \qquad \text{and} \qquad W_d \triangleq 4\kappa \frac{n_d}{\sqrt{\beta_d}} \log N. \tag{1}$$

3.1 Running time

We begin by providing an upper bound on the running time of RECURSIVE WINDOW SORT.

► **Lemma 6.** *The overall running time of* RECURSIVE WINDOW SORT *is* $\widetilde{O}(N^{\frac{3}{2}})$.

Proof. Recall that the running time of WINDOW SORT on an instance of n elements with starting window size W is upper bounded by $c'nW \log n$ for some constant $c' \geq 1$ (Lemma 4). Consider an execution of our algorithm whose depth d defines an *index* $i = h - d$. We now prove by induction on i that its running time T_i is upper bounded by $c(i+1)n_d^{1+2^i/(2^{i+1}-1)} \log N$, where $c = 4\kappa c'$. (Notice that c is a global constant that does not depend on i.)

[2] To avoid being distracted by rounding, we assume that h is an integer.
[3] Here n_d is a function of both N and d.

If $i = 0$ then $d = h$ and the running time coincides with the one of Window Sort, i.e., it is less than $c' n_d^2 \le c n_d^{1+2^0/(2^1-1)}$. This proves the base case.

For $i > 0$ the overall running time is bounded by the sum of: (i) the time required to perform $b_d = \frac{n_d}{\beta_d}$ recursive calls with depth $d{+}1$ (i.e., having index is $h{-}(d{+}1) = (h{-}d){-}1 = i-1$), on instances of size $\beta_d = n_{d+1}$, and (ii) the time required to run Window Sort on an instance of size n_d and initial window size $W_d = 4\kappa \frac{n_d}{\sqrt{\beta_d}} \log N$. By inductive hypothesis, each of the recursive calls requires time $T_{i-1} \le c\, i\, n_{d+1}^{1+\frac{2^{i-1}}{(2^i-1)}} \log N = c i \beta_d^{1+\frac{2^{i-1}}{2^i-1}} \log N$. Thus

$$
\begin{aligned}
T_i &\le \frac{n_d}{\beta_d} \cdot c i \beta_d^{1+\frac{2^{i-1}}{2^i-1}} \log N + 4\kappa c' \frac{n_d^2}{\sqrt{\beta_d}} \log N = c\left(i n_d \beta_d^{\frac{2^{i-1}}{2^i-1}} + n_d^{\frac{3}{2}+\frac{1}{2^{i+2}-2}} \right) \log N \\
&= c\left(i n_d^{1+\frac{2^i}{2^{i+1}-1}} + n_d^{1+\frac{2^i}{2^{i+1}-1}} \right) \log N = c(i+1) n_d^{1+\frac{2^i}{2^{i+1}-1}} \cdot \log N.
\end{aligned}
$$

This completes the proof of the inductive step. By setting $i = h$, and for a sufficiently large N, we obtain $T_h \le c(1 + \log N) N^{1+\frac{N}{2N-1}} \cdot \log N \le 2c(\log N)^2 N^{\frac{3}{2}+\frac{1}{4N-2}}$. Since $N^{\frac{1}{4N-2}} = 2^{\frac{\log N}{4N-2}} = O(1)$ we conclude that $T_h = O(N^{3/2} \log^2 N)$. ◀

3.2 Correctness

Here we will formally prove the correctness of Recursive Window Sort. To this aim, we shall first give a sufficient condition for which, if all executions at depth $d + 1$ return sequences of dislocation $\kappa \log N$, then also the execution at depth d returns a sequence of dislocation at most $\kappa \log N$.

▶ **Definition 7** (Good Blocks)**.** We say that an execution of Recursive Step at depth $d < h$ has Good Blocks if the sequence S to which we apply the recursion satisfies the following condition: For any element x in S, $\left| L_x - \frac{\beta_d}{n_d} G_x \right| \le 2\sqrt{\beta_d} \log N$, for $G_x = rank(x, S)$ and $L_x = rank(x, B_j)$, where B_j is the block of length $\beta_d = n_{d+1}$ containing x.

Note that the input of each execution (recursive call) of Recursive Step is a fixed subset of elements of the initial sequence which *does not depend on the comparison errors.*

▶ **Lemma 8.** *Consider an execution of* Recursive Step *at depth $d < h$ and suppose that the following conditions hold:*

1. *The execution has* Good Blocks *(Definition 7);*
2. *All the executions at depth $d + 1$ return a sequence with maximum dislocation $\kappa \log N$;*
3. *The comparison errors are well spread.*

Then, the considered execution returns a sequence with maximum dislocation $\kappa \log N$.

Proof. By hypothesis 3 together with Lemma 4, it suffices to show that the sequence S' obtained before invoking Window Sort has maximum dislocation at most $W_d = 4\kappa \frac{n_d}{\sqrt{\beta_d}} \log N$.[4] Consider an element $x \in S$, let B_j be the block containing x, and let $\widetilde{L}_x$ be the number of elements preceding x in B'_j (its position in B'_j minus 1). By the hypothesis on

[4] Notice that if the errors in S are well spread, then they are also well spread in S' since the order of the elements is irrelevant in Definition 2.

the executions at depth $d+1$, we know that $|\widetilde{L}_x - L_x| \leq \kappa \log N$ and, since the considered execution has Good Blocks, we can use triangle inequality to write

$$\left|\widetilde{L}_x - \frac{\beta_d}{n_d} G_x\right| \leq 2\sqrt{\beta_d}\log N + \kappa \log N \leq 3\kappa\sqrt{\beta_d}\log N.$$

Let $\widetilde{G}_x$ be the number of elements preceding x in S' (its position minus 1), so that $|\widetilde{G}_x - G_x|$ is the dislocation of x in S'. By definition of S' we have

$$\widetilde{G}_x \geq \frac{n_d}{\beta_d}\widetilde{L}_x \geq G_x - \frac{3\kappa n_d}{\sqrt{\beta_d}}\log N,$$

and similarly

$$\widetilde{G}_x \leq \frac{n_d}{\beta_d}\widetilde{L}_x + \frac{n_d}{\beta_d} \leq G_x + \frac{3\kappa n_d}{\sqrt{\beta_d}}\log N + \frac{n_d}{\beta_d} \leq G_x + \frac{4\kappa n_d}{\sqrt{\beta_d}}\log N\,.$$

Therefore we have shown that $\left|\widetilde{G}_x - G_x\right| \leq \frac{4\kappa n_d}{\sqrt{\beta_d}}\log N$, which concludes the proof. ◀

▶ **Lemma 9.** *The probability that all executions of* Recursive Step *at depth $d < h$ have (jointly)* Good Blocks *is at least $1 - \frac{1}{N^2}$.*

Proof. Fix an element x and a depth $d < h$. Let S and B_j be the sequence of size n_d and its block of size $b_d = n_{d+1}$ containing x. (Both S and B_j are random variables as they depend on the initial random permutation of all N elements.) Since S is a subsequence of a random permutation we can study the distribution of L_x by considering the following:

1. After the random permutation of the N input elements is chosen (thus S and B_j are determined), we randomly permute the elements of S again apart from x (which stays in its position in S and in B_j);
2. We view the previous item as the following experiment. An urn contains G_x black balls (elements smaller than x) and $n_d - G_x - 1$ white balls (elements bigger than x). Out of these $n_d - 1$ balls, choose $\beta_d - 1$ at random and consider the number of chosen black balls (the local rank L_x).

It is well known that, permuting a subsequence of a randomly chosen permutation, gives again a randomly chosen permutation. Therefore the modification of Item 1 is equivalent to the original algorithm. A random permutation of the elements in S determines which of them fall into B_j and thus Item 1 is equivalent to Item 2. The number of chosen black balls is the local rank L_x of x. We hence have that L_x is distributed as an *hypergeometric* random variable of parameters $n_d - 1$, G_x, and $\beta_d - 1$ and we can use the following tail bound [15]:

$$\Pr(|L_x - \mathbb{E}[L_x]| \geq t(\beta_d - 1)) \leq 2e^{-2t^2(\beta_d - 1)}\ ,$$

where $\mathbb{E}[L_x] = \frac{\beta_d - 1}{n_d - 1}G_x$. By choosing $t = 2\sqrt{\frac{\log N}{\beta_d - 1}}$, we obtain

$$\begin{aligned}\Pr(|L_x - \frac{\beta_d}{n_d}G_x| \geq 3\sqrt{\beta_d}\log N) &\leq \Pr(|L_x - \frac{\beta_d - 1}{n_d - 1}G_x| \geq 2\sqrt{\beta_d}\log N)\\ &\leq \Pr(|L_x - \mathbb{E}[L_x]| \geq 2\sqrt{\beta_d}\log N) \leq \Pr(|L_x - \mathbb{E}[L_x]| \geq 2\sqrt{\beta_d - 1}\log N) \leq \frac{2}{N^4}.\end{aligned}$$

Notice that the overall number of elements x for which the above condition must hold is upper bounded by $N \log N$. Indeed, there are $h = \log N$ recursion levels and each level defines a partition of the N elements into blocks (i.e., the total number of elements at each level is N). By the union bound, the probability that all the executions are good is at least $1 - (N \log N)\frac{2}{N^4} > \frac{1}{N^2}$. ◀

The following two lemmas allow us to use Lemma 9 in a recursive fashion.

▶ **Lemma 10.** *For every $d = 0, \dots, h$, it holds that $n_d \geq N^{\frac{1}{2}}$.*

▶ **Lemma 11.** *The errors of all the sequences S' are (jointly) well spread with probability at least $1 - \frac{1}{N^2}$.*

We are now ready to prove the final theorem of this section.

▶ **Theorem 12.** Recursive Window Sort *returns a sequence with maximum dislocation $\kappa \log N$ with probability at least $1 - \frac{1}{N^2}$. Moreover, its running time is $\widetilde{O}(N^{\frac{3}{2}})$ and the expected total dislocation of the returned sequence is $O(n)$.*

Proof. We assume that (i) all the recursive executions of Recursive Step have Good Blocks and that (ii) all the sequences S' used as input for Window Sort have well spread errors. By Lemma 9 and Lemma 11 this happens with probability at least $1 - \frac{2}{N^2}$).

We prove the following claim by induction on $i = h - d$: all the executions of Recursive Step at depth d return a sequence having maximum dislocation $\kappa \log N$.

If $i = 0$, then $d = h$. Consider any execution of Recursive Step at depth h and let S be its input. Notice that Window Sort is invoked on S with window size $n_d = |S|$, meaning that both conditions for Definition 3 are met and hence, by Lemma 4, Window Sort returns a sequence having maximum dislocation $\kappa \log n_d \leq \kappa \log N$.

If $i > 0$, then $d < h$ and we once again focus on any single execution of Recursive Step at depth d having input S. By inductive hypothesis all the executions at depth $d + 1$ returned a sequence having maximum dislocation $\kappa \log N$. This, combined with our assumptions, allows us to invoke Lemma 8 which proves the first claim.
To conclude the proof, notice that the running time is bounded by Lemma 6 and that, by Lemma 4, the sequence returned by the execution of Window Sort at depth $d = 0$ has expected total dislocation $O(n)$. ◀

4 Derandomization

Recursive Window Sort requires as input a random permutation of the N elements. In this section, we show how to derandomize the algorithm. In particular, we show how to generate "*almost random*" bits from the outcome of element comparisons, which can be thought as as biased coins tosses. The derandomized Recursive Window Sort is then as follows: We extract a (random) subset of elements and use them to generate random bits. Then, we use these bits to generate a random permutation of the remaining elements, which allows us to invoke Recursive Step on this permutation. Finally, we reinsert the extracted elements into the approximately sorted sequence, so that the maximum dislocation remains $O(\log N)$. Notice that the sequence returned by Recursive Step (indirectly) depends on the set of extracted elements though the results of their comparisons. We circumvent this problem by providing an algorithm that is able to reinsert a *single* element in *any* sequence having dislocation $O(\log N)$ as long as errors are well spread. We then show how this algorithm can be used to reinsert all the extracted elements without any asymptotic increase in the dislocation. For any two elements x and y we write $x \,\widetilde{<}\, y$ (resp. $x \,\widetilde{>}\, y$) to denote the fact that x *compared* smaller (resp. larger) than y.

4.1 (Re-)Inserting one element

The first key ingredient is an algorithm which reinserts an element in a sequence of n elements of maximum dislocation $O(\log n)$ so that this bound on the dislocation is maintained (up to a multiplicative constant depending on p).

Algorithm 4: InsertPosition (on a sequence $S = \langle s_0, \ldots, s_{n-1} \rangle$ of n distinct elements and on an element x not in S).

```
// Compute a "penalty" function for each possible position.
1. For every index $i = 0, \ldots, n$:
    a. Let $S_{i-} \triangleq \{s_{i-c_2 \log n}, \ldots, s_{i-1}\}$ and $S_{i+} \triangleq \{s_i, \ldots, s_{i+c_2 \log n - 1}\}$.
    b. Let $penalty_i(x, S) \triangleq defeats(x, S_{i-}) + wins(x, S_{i+})$.
// Return the position of minimal penalty.
2. Return any index $i^* \in \arg\min_{i=0,\ldots,n} penalty_i(x, S)$.
```

▶ **Definition 13** (single insertion)**.** The single insertion problem is defined as follows. We are given an arbitrary sequence[5] of n distinct elements, $S = \langle s_0, \ldots, s_{n-1} \rangle$, whose maximum dislocation is at most $c_1 \log n$, for some $c_1 \geq 1$, and another element x distinct from all these elements. The goal is to insert x in a position i^* which still guarantee $c_2 \log n$ maximum dislocation, for $c_2 := \frac{7c_1}{p}$. That is, the sequence $S' = \langle s_0, \ldots, s_{i^*-1}, x, s_{i^*}, \ldots, s_{n-1} \rangle$ has maximum dislocation at most $c_2 \log n$.

In the following, we consider these two quantities:

$$wins(x, Y) \triangleq |\{y \in Y \,:\, x \mathrel{\tilde{>}} y\}| \qquad \text{and} \qquad defeats(x, Y) \triangleq |\{y \in Y \,:\, x \mathrel{\tilde{<}} y\}| \,,$$

where x is an arbitrary element and Y an arbitrary subset of elements. Algorithm Insert-Position (see Algorithm 4 above) solves the single insertion problem with high probability:

▶ **Theorem 14.** *Let S be a sequence of n elements having maximum dislocation $c_1 \log n$. With probability at least $1 - \frac{3}{n^2}$ algorithm* InsertPosition *returns an index i^* such that the sequence $S' = \langle s_0, \ldots, s_{i^*-1}, x, s_{i^*}, \ldots, s_{n-1} \rangle$ has maximum dislocation at most $\frac{7c_1}{p} \log n$.*

Intuitively, the proof of this result is based on the following two facts:
1. When i is *away* from the true (correct) rank of x in S, there is a *large* penalty (Lemma 15);
2. When i is *equal* to the true (correct) rank of x in S, the penalty is *small* (Lemma 16).

▶ **Lemma 15.** *If $|i - r| \geq c_2 \log n$ then $penalty_i(x, S) \geq \frac{1-p}{2}(c_2 - 2c_1) \log n$ with probability at least $1 - \frac{2}{n^{13}}$.*

▶ **Lemma 16.** *$penalty_r(x, S) \leq \frac{1-p}{2}(c_2 - 2c_1) \log n$ with probability at least $1 - \frac{2}{n^2}$.*

Proof of Theorem 14. By Lemma 16 and by union bound on Lemma 15, we conclude that with probability at least $1 - \frac{3}{n^2}$, $penalty_r(x, S) < penalty_i(x, S)$ for every i with $|i - r| \geq c_2 \log n$. In this case, the algorithm returns a index i^*, such that $|i^* - r| < c_2 \log n$. Furthermore, the dislocation of each element between i^* and r in S changes by at most 1, and the dislocation of the other elements is unchanged. ◀

4.2 Generating almost random bits

The result $r \in \{\tilde{<}, \tilde{>}\}$ of comparison between two distinct elements $x, y \in S$ can be seen as a biased coin if we label its faces with $0 \triangleq \tilde{<}$ and $1 \triangleq \tilde{>}$: Since the comparison fails

[5] Note that S can be adversarial and can also be chosen as a function of the comparison results, of the true order of the elements, and of x.

with probability p, but we do not know the correct answer, the coin is biased towards one of its faces, i.e., either it lands on 0 with probability $1-p$ and on 1 with probability p or vice-versa. Moreover, the coin can be tossed at most once, since errors (or lack of thereof) are persistent. Consider now a collection $\mathcal{C}$ of coins whose faces are labeled 0 and 1 and let $\chi_C \in \{0,1\}$ denote the outcome of the coin flip involving coin $C \in \mathcal{C}$. For any subset $C = \{C_1, C_2, \dots\} \subseteq \mathcal{C}$ we compute the exclusive or the results $\chi(C) \triangleq \chi_{C_1} \oplus \chi_{C_2} \oplus \chi_{C_3} \oplus \cdots$ (where $\chi(C) = 0$ if $C = \emptyset$).

The next lemma shows that we can generate an almost random bits with a sufficiently large number of biased coin tosses (comparisons):

▶ **Lemma 17.** *For any choice of the coin biases, and any subset $C = \{C_1, C_2, \dots, \} \subseteq \mathcal{C}$ such that $|C| = \Omega(\log N)$, $\frac{1}{2} - \frac{1}{N^4} \leq P(\chi(C) = 0) \leq \frac{1}{2} + \frac{1}{N^4}$. (For a suitable hidden constant that depends on p).*

Notice that the above lemma holds for any choice of the coin biases (i.e., regardless of true order between the compared elements), therefore we can write the following

▶ **Corollary 18.** *For any collection $\{C^{(1)}, C^{(2)}, \dots, C^{(\eta)}\}$ of pairwise disjoint subset of $\mathcal{C}$, each of size $O(\log n)$, and any $r \in \{0,1\}^\eta$ we have that*

$$\left(\frac{1}{2} - \frac{1}{N^4}\right)^\eta \leq P\left((\chi(C^{(1)}, \dots, \chi(C^{(\eta)})) = r\right) \leq \left(\frac{1}{2} + \frac{1}{N^4}\right)^\eta .$$

Finally, we show that we are able to generate random integers in an interval that closely resemble a discrete uniform distribution.

▶ **Lemma 19.** *Let $\ell \leq N$. It is possible to generate a number z in $0, \dots, \ell-1$ using $O(\log^2 N)$ comparison results. With probability at most $\frac{1}{N^3}$, z will be a spurious result and we say that the* fail *event happens. If the fail event does not happen, then z is uniformly distributed in $0, \dots, \ell - 1$.*

4.3 Derandomized Iterated Windowsort

We are now in a position to describe our deterministic algorithm Derandomized Recursive Window Sort (see Algorithm 5) and its analysis. We have already seen above how to perform and analyze most of the algorithm's steps. We will now give proofs for reinserting many elements at the same time in Step 7, and then present our main theorem for Derandomized Recursive Window Sort. For the rest of this section we let $c_3 = \frac{7\kappa}{p}$. Moreover, we will say that Derandomized Recursive Window Sort *fails* if the fail event of Lemma 19 happens at least once during the execution of the algorithm. The following two lemmas bound the dislocation of the sequence $S^{(4)}$ obtained by reinserting the elements in R after Recursive Window Sort is invoked.

▶ **Lemma 20.** *Suppose* Derandomized Recursive Window Sort *does not fail. Then, with probability at least $1 - \frac{1}{N^2}$, all the sets $R_i = \{r \in R : i \leq rank(r, S^{(1)}) \leq i + 2c_3 \log N\}$, for $0 \leq i < N$, contain at most 6 elements each.*

Proof. If there exists a set R_i that contains 7 or more elements, then there exists a corresponding set $S_j \subseteq S^{(0)}$ that satisfies: (i) $|S_j| \leq 2c_3 \log N + 8$, and (ii) $|R_i \cap S_j| \geq 7$.[6]

[6] Indeed, it suffices to choose $S_j = \{x \in S^{(0)} : j \leq rank(x, S^{(0)}) \leq j + 2c_3 \log_n +7\}$, where $j = i + |\{r \in R\colon rank(r, S^{(1)}) < i\}|$.

Algorithm 5: DERANDOMIZED RECURSIVE WINDOW SORT (on a sequence $S = \langle s_0, \dots, s_{n-1} \rangle$ of n distinct elements).

```
   // Next step generates $\Omega(N)$ comparisons outcomes.
1. Let $s_0$ be the first element in $S$. Compare $s_0$ to every element in $S^{(0)} = S \setminus \{s_0\}$
   // This requires $O(\log^4 N)$ comparison outcomes.
2. Choose a set $R$ of $\log^3 n$ (distinct) random elements from $S^{(0)}$ (see Lemma 19).
   // Next step generates $\Omega(N \log^3 N)$ comparison outcomes.
3. Compare each element in $R$ with each element in $S^{(1)} = S^{(0)} \setminus R$
   // This requires $O(N \log^2 N)$ comparison outcomes (using, e.g., Fisher–Yates shuffle [13]).
4. Obtain a random permutation $S^{(2)}$ of the elements in $S^{(1)}$ (see Lemma 19).
5. Invoke RECURSIVE STEP on $S^{(2)}$ with initial depth 0 to obtain sequence $S^{(3)}$
6. For each element $x \in R$, compute its position $i^*_x$ in $S^{(3)}$ using Algorithm 4.
7. Insert (simultaneously) each $x \in R$ in position $i^*_x$ of $S^{(3)}$ to obtain $S^{(4)}$
   (break ties arbitrarily).
8. Insert $s_0$ in $S^{(4)}$ using Algorithm 4 to obtain $S^{(5)}$. Return $S^{(5)}$.
```

We show that the probability that any single set S_i exists is at most $\frac{1}{n^3}$, so that the claim will immediately follow by using the union bound on the (at most N) values of i.

Notice that, since R is a random subset of elements of $S^{(0)}$, the probability that 7 or more elements from R belong to S_i can be upper bounded by the probability of success of the following experiment: An urn contains $|S^{(0)}| = N - 1$ balls, $|R|$ of which are black; we draw $\eta = |S_i| = 2c_3 \log N + 8$ balls without replacement and we succeed if the number X of drawn black balls is 7 or more.

Since X is distributed as an hypergeometric random variable of parameters $N-1$, $|R|$, and η, we have (for sufficiently large values of N):

$$\begin{aligned}
\Pr(X \geq 7) &= \sum_{j=9}^{\eta} \frac{\binom{|R|}{j}\binom{N-|R|-1}{\eta-j}}{\binom{N-1}{\eta}} \leq \sum_{j=7}^{\eta} |R|^j \cdot \frac{\binom{N-1}{\eta-j}}{\binom{N-1}{\eta}} \\
&= \sum_{j=7}^{\eta} |R|^j \cdot \frac{(N-1)!}{(\eta-j)!(N-1-\eta+j)!} \cdot \frac{\eta!(N-1-\eta)!}{(N-1)!} \\
&= \sum_{j=7}^{\eta} |R|^j \cdot \frac{\eta!}{(\eta-j)!} \cdot \frac{(N-1-\eta)!}{(N-1-\eta+j)!} \leq \sum_{j=7}^{\eta} \left(|R| \frac{\eta}{N-\eta} \right)^j \\
&< \sum_{j=7}^{\eta} N^{-j/2} \leq \int_{x=6}^{\infty} N^{-x/2} \mathrm{d}x = \frac{2}{N^3 \ln N} < \frac{1}{N^3}
\end{aligned}$$ ◀

▶ **Lemma 21.** *Suppose* DERANDOMIZED RECURSIVE WINDOW SORT *does not fail. With probability at least* $1 - \frac{3}{N^2}$*: (i) the maximum dislocation of* $S^{(4)}$ *is at most* $c_3 \log N + 6$ *and (ii) the dislocations of an the element* $yinS^{(3)}$ *increases by at most* 6 *in* $S^{(4)}$.

All the previous lemmas together allow us to state the main result of this section.

▶ **Theorem 22.** *Algorithm 5 is a deterministic algorithm that returns, in* $\widetilde{O}(N^{\frac{3}{2}})$ *time, a sequence with maximum dislocation* $O(\log N)$ *and total dislocation* $O(n)$ *with probability at least* $1 - \frac{1}{N}$.

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 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`.

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

5 Don Coppersmith, Lisa Fleischer, and Atri Rudra. Ordering by weighted number of wins gives a good ranking for weighted tournaments. *ACM Trans. Algorithms*, 6(3):55:1–55:13, 2010. `doi:10.1145/1798596.1798608`.

6 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.

7 Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994. `doi:10.1137/S0097539791195877`.

8 Tomas Gavenciak, Barbara Geissmann, and Johannes Lengler. Sorting by swaps with noisy comparisons. In Peter A. N. Bosman, editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, pages 1375–1382. ACM, 2017. `doi:10.1145/3071178.3071242`.

9 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Sorting with recurrent comparison errors. *Proc of ISAAC 2017*, 2017. To appear. And *ArXiv e-prints arXiv:1709.07249, 2017.*

10 Barbara Geissmann and Paolo Penna. Sort well with energy-constrained comparisons. *ArXiv e-prints arXiv:1610.09223*, 2016.

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

12 Rolf Klein, Rainer Penninger, Christian Sohler, and David P. Woodruff. Tolerant Algorithms. In *Proceedings of the19th Annual European Symposium on Algorithm*, pages 736—-747, 2011.

13 Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998. URL: `http://www.worldcat.org/oclc/312898417`.

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

15 Matthew Skala. Hypergeometric tail inequalities: ending the insanity. *ArXiv e-prints arXiv:1311.5939*, 2013.