Noisy Sorting Without Searching: Data Oblivious Sorting with Comparison Errors

Ramtin Afshar ✉
University of California, Irvine, USA

Michael Dillencourt ✉
University of California, Irvine, USA

Michael T. Goodrich ✉
University of California, Irvine, USA

Evrim Ozel ✉
University of California, Irvine, USA

Abstract

We provide and study several algorithms for sorting an array of n comparable distinct elements subject to probabilistic comparison errors. In this model, the comparison of two elements returns the wrong answer according to a fixed probability, $p_e < 1/2$, and otherwise returns the correct answer. The ***dislocation*** of an element is the distance between its position in a given (current or output) array and its position in a sorted array. There are various algorithms that can be utilized for sorting or near-sorting elements subject to probabilistic comparison errors, but these algorithms are not data oblivious because they all make heavy use of noisy binary searching. In this paper, we provide new methods for sorting with comparison errors that are data oblivious while avoiding the use of noisy binary search methods. In addition, we experimentally compare our algorithms and other sorting algorithms.

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

Keywords and phrases sorting, algorithms, randomization, experimentation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.1

Supplementary Material *Software (Source Code)*: `https://github.com/UC-Irvine-Theory/NoisyObliviousSorting`

1 Introduction

Given an array, A, of n distinct comparable elements, we study the problem of efficiently sorting A subject to noisy probabilistic comparisons. In this framework, which has been extensively studied [2, 5, 7–9, 14, 16, 17, 19, 21, 23, 31], the comparison of two elements, x and y, results in a true result independently according to a fixed probability, and otherwise returns the opposite (false) result. In the case of ***persistent*** errors [2, 7–9, 17], the result of a comparison of two given elements, x and y, always returns the same result. In the case of ***non-persistent*** errors [5, 14, 16, 19, 23, 31], however, the probabilistic determination of correctness is determined independently for each comparison, even if it is for a pair of elements, (x, y), that were previously compared.

Motivation for sorting with comparison errors comes from multiple sources, including applied cryptography scenarios where cryptographic comparison protocols can fail with known probabilities (see, e.g., [6, 20, 33]). In such cases, reducing the noise from comparison errors can be computationally expensive, and the framework advanced in our paper offers an alternative, possibly more efficient approach, where a higher error rate is tolerated while still achieving the ultimate goal of sorting or near-sorting with high probability. Further, other applications of sorting with comparison errors include ranking objects in online forums via group A/B testing [32].

Since it is not possible to always correctly sort an array, A, subject to persistent comparison errors, we follow the formulation of Geissmann *et al.* [7–9], and define the ***dislocation*** of an element, x, in an arary, A, as the absolute value of the difference between x's index in A and its index in the correctly sorted permutation of A. Further, define the ***maximum dislocation*** of A as the maximum dislocation for the elements in A, and let the ***total dislocation*** of A be the sum of the dislocations of the elements in A. By known lower bounds [7–9], the best a sorting algorithm can achieve under persistent comparison errors is a maximum dislocation of $O(\log n)$ and a total dislocation of $O(n)$. Thus, coming close to such asymptotic maximum and total dislocation guarantees should be the goal for a sorting algorithm in the presence of persistent comparison errors.

Given the cryptographic applications of noisy comparisons, we desire sorting algorithms that are ***data oblivious***, which support privacy-preserving cryptographic protocols. A sorting algorithm is data oblivious if its memory access pattern does not reveal any information about the data values being sorted. Unfortunately, existing efficient algorithms for sorting with noisy comparisons are not data oblivious. Indeed, they all make use of noisy binary search [8], which is a data-sensitive random walk in a binary search tree, e.g., see Geissmann, Leucci, Liu, and Penna [8], Feige, Raghavan, Peleg, and Upfal [5], and Leighton, Ma, and Plaxton [19]. Instead, we desire efficient sorting algorithms that tolerate noisy comparisons and avoid the use of noisy binary search, so as to be data oblivious (i.e., privacy preserving if comparisons are done according to a data-hiding protocol).

Related Prior Results. Problems involving probabilistic comparison errors can trace their roots back to a classic problem by Rényi [27] of playing a two-person game where player A poses yes/no questions to a player B who lies with a given probability; see a survey by Pelc [24]. Notable prior results include a paper by Pippenger [25] on computing Boolean functions with probabilistically noisy gates and work by Yao and Yao [34] on sorting networks built from noisy comparators. There is also considerable work on searching when the total number of faulty comparisons is bounded rather than considering probabilistic noisy comparisons, including the work by Kenyon-Mathieu and Yao [15] and Rivest, Meyer, Kleitman, Winklmann, and

Spencer [28]. Also of note is work by Karp and Kleinberg [14], who study binary searching for a value $x \in [0, 1]$ in an array of biased coins ordered by their biases.

Braverman and Mossel [2] introduce a persistent-error model, where comparison errors are persistently wrong with a fixed probability, $p < 1/2 - \varepsilon$, and they achieve a sorting algorithm that in our framework runs in $O(n^{3+f(p)})$ time, where $f(p)$ is some function of p, with maximum expected dislocation $O(\log n)$ and total dislocation $O(n)$. Klein, Penninger, Sohler, and Woodruff [17] improve the running time to $O(n^2)$, but with $O(n \log n)$ total dislocation w.h.p. The running time for sorting in the persistent-error model optimally with respect to maximum and total dislocation was subsequentially improved to $O(n^2)$, $O(n^{3/2})$, and ultimately to $O(n \log n)$, in a sequence of papers by Geissmann, Leucci, Liu, and Penna [7–9], all of which are not data oblivious because they make extensive use of noisy binary searching, which amounts to a random walk in a binary search tree.

Our Results. In this paper, we provide data-oblivious sorting algorithms that tolerate persistent noisy comparisons. In addition, we empirically compare our algorithms to other sorting algorithms, including the worst-case optimal algorithm, Riffle sort, by Geissmann, Leucci, Liu, and Penna [8], which is not data oblivious, but it achieves an optimal maximum and total dislocation under noisy comparisons. It runs in $O(n \log n)$ time, but it makes use of noisy binary search.

In addition to providing theoretical analysis for some of our algorithms, we empirically study all of our algorithms by measuring the effect of changing the amount of noise and the input size on the amount of dislocation, inversions, and number of comparisons. Our experiments show that for all of the data-oblivious algorithms we provide in this paper, the maximum and total dislocations are comparable to the optimal bounds of $O(\log n)$ and $O(n)$ respectively for the best algorithms that are not data oblivious. Moreover, we include experimental results for some standard sorting algorithms such as insertion sort, quick sort, and shell sort, for which we provide emipirical evidence that all of our algorithms significantly outperform these other algorithms in terms of the maximum and total dislocation metrics. These results indicate that our algorithms are able to combine the properties of both having a good tolerance to noisy comparisons while also being data-oblivious.

2 Window-Sort

Our first sorting algorithm is a version of window-sort [7], which will be useful as a subroutine in our other algorithms. We describe the pseudo-code at a high level in Algorithm 1, for approximately sorting an array of size n that has maximum dislocation at most $d_1 \leq n$ so that it will have maximum dislocation at most $d_2 = d_1/2^k$, for some integer $k \geq 1$, with high probability as a function of d_2.

Algorithm 1 Window-Sort($A = \{a_0, a_1, \ldots, a_{n-1}\}, d_1, d_2$)

```
for w ← 2d_1, d_1, d_1/2, ..., 2d_2 do
    foreach i ← 0, 1, 2, ..., n − 1 do
        r_i ← max{0, i − w} + |{a_j < a_i : |j − i| ≤ w}|
    Sort A (deterministically) by nondecreasing r_i values (i.e., using r_i as the
      comparison key for a_i)
return A
```

In addition to implementing window-sort data obliviously, we provide a new analysis of window-sort, which allows us to apply it in new contexts. We begin this new analysis with the following lemma, which establishes the progress made in each iteration of window-sort.

▶ **Lemma 1.** *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, has maximum dislocation at most d' prior to an iteration of window-sort for $w = 2d'$ (line 1 of Algorithm 1), then after this iteration, A will have maximum dislocation at most $d'/2$ with probability at least $1 - n2^{-d'/8}$.*

Proof. Let a_i be an element in A. Let W denote the window of elements in A for which we perform comparisons with a_i in this iteration; hence, $2d' \leq |W| \leq 4d'$. Because A has maximum dislocation d', by assumption, there are no elements to the left (resp., right) of W that are greater than a_i (resp., less than a_i). Thus, a_i's dislocation after this iteration depends only on the comparisons between a_i and elements in its window. Let X be a random variable that represents a_i's dislocation after this iteration, and note that $X \leq Y$, where Y is the number of incorrect comparisons with a_i performed in this iteration. Note further that we can write Y as the sum of $|W|$ independent indicator random variables and that $\mu = E[Y] = p_e|W| \leq d'/4$. Thus, if we let $R = d'/2$, then $R \geq 2\mu$; hence, we can use a Chernoff bound as follows:

$$\Pr(X > d'/2) \leq \Pr(Y > d'/2) = \Pr(Y > R) \leq 2^{-R/4} = 2^{-d'/8}.$$

Thus, with the claimed probability, the maximum dislocation for all elements of A will be at most $d'/2$, by a union bound. ◀

This implies the following.

▶ **Theorem 2.** *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, of size n has maximum dislocation at most $d_1 \geq \log n$, then executing Window-Sort(A, d_1, d_2) runs in $O(d_1 n)$ time. Further, we can execute Window-Sort(A, d_1, d_2) data-obliviously to result in A having maximum dislocation of $d_2/2$ with probability at least $1 - 2n2^{-d_2/8}$, where $d_2 = d_1/2^k$, for some integer $k \geq 1$.*

Proof. For the running time and data obliviousness, note that we can perform the deterministic sorting step using a data-oblivious sorting algorithm (e.g., see [12]) in $O(n \log n)$ time. The windowed comparison steps (step 3 of Algorithm 1) are already data-oblivious and their running times form a geometric sum adding up to $O(d_1 n)$; hence, the total time for all the deterministic sorting steps (step 4 of Algorithm 1) is $O((\log(d_1/d_2))n \log n)$, which is at most $O(d_1 n)$ for $d_1 \geq \log n$.

For the maximum dislocation bound, note once $w = 2d_2$ and the array A prior to this iteration has maximum dislocation at most d_2, then it will result in having maximum dislocation at most $d_2/2$ with probability at least $1 - n2^{-d_2/8}$, by Lemma 1. Thus, by a union bound, the overall failure probability is at most

$$n\left(2^{-d_2/8} + 2^{-2d_2/8} + 2^{-4d_2/8} + \cdots + 2^{-d_1/8}\right) \quad < \quad n2^{-d_2/8} \sum_{i=0}^{\infty} 2^{-i} \quad = \quad 2n2^{-d_2/8}.$$

◀

In terms of efficiency, we note that our data-oblivious implementation of window-sort is only time-efficient for small subarrays; hence, we need to do more work to design an efficient data-oblivious sorting algorithm.

3 Window-Merge-Sort

In this section, we describe a simple algorithm for sorting with noisy comparisons, which achieves a maximum dislocation of $O(\log n)$. Our window-merge-sort method is a windowed version of merge sort; hence, it is deterministic but not data oblivious. Nevertheless, it does avoid using noisy binary search.

Suppose we are given an array, A, of n elements (we use n to denote the original size of A, and N to denote the size of the subproblem we are currently working on recursively). Our method runs in $O(n \log^2 n)$ time and we give the pseudo-code for this method in Algorithm 2, with $d = c \log n$ for a constant $c \geq 1$ set in the analysis.

Algorithm 2 Window-Merge-Sort$(A = \{a_0, a_1, \ldots, a_{N-1}\}, n, d)$

```
if N ≤ 6d then
    return Window-Sort(A, 4d, d)
Divide A into two subarrays, A_1 and A_2, of roughly equal size
Window-Merge-Sort(A_1, n, d)
Window-Merge-Sort(A_2, n, d)
Let B be an initially empty output list
while |A_1| + |A_2| > 6d do
    Let S_1 be the first min{3d, |A_1|} elements of A_1
    Let S_2 be the first min{3d, |A_2|} elements of A_2
    Let S ← S_1 ∪ S_2
    Window-Sort(S, 4d, d)
    Let B' be the first d elements of (the near-sorted) S
    Add B' to the end of B and remove the elements of B' from A_1 and A_2
Call Window-Sort(A_1 ∪ A_2, 4d, d) and add the output to the end of B
return B
```

Our method begins by checking if the current problem size, N, satisfies $N \leq 6d$, in which case we're done. Otherwise, if $N > 6d$, then we divide A into 2 subarrays, A_1 and A_2, of roughly equal size and recursively approximately sort each one. For the merge of the two sublists, A_1 and A_2, we inductively assume that A_1 and A_2 have maximum dislocation at most $3d/2 = (3c/2) \log n$. We then copy the first $3d$ elements of A_1 and the first $3d$ elements of A_2 into a temporary array, S, and we note that, by our induction hypothesis, S contains the smallest $3d/2$ elements currently in A_1 and the smallest $3d/2$ elements currently in A_2. We then call Window-Sort$(S, 4d, d)$, and copy the first d elements from the output of this window-sort to the output of the merge, removing these same elements from A_1 and A_2. Then we repeat this merging process until we have at most $6d$ elements left in $A_1 \cup A_2$, in which case we call window-sort on the remaining elements and copy the result to the output of the merge. The following lemma establishes the correctness of this algorithm.

▶ **Lemma 3.** *If A_1 and A_2 each have maximum dislocation at most $3d/2$, then the merge of A_1 and A_2 has maximum dislocation at most $3d/2$ with probability at least $1 - N2^{-d/8}$.*

Proof. By Lemma 1 and a union bound, each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most $d/2$, with at least the claimed probability. So, let us assume each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most

$d/2$. Consider, then, merge step i, involving the i-th call to Window-Sort$(S, 4d, d)$, where S consists of the current first $3d$ elements in A_1 and the current first $3d$ elements in A_2, which, by assumption, contain the current smallest $3d/2$ elements in A_1 and current smallest $3d/2$ elements in A_2. Thus, since this call to window-sort results in an array with maximum dislocation at most $d/2$, the subarray, B_i, of the d elements moved to the output in step i includes the $d/2$ current smallest elements in $A_1 \cup A_2$. Moreover, the first $d/2$ elements in B_i have no smaller elements that remain in S. In addition, for the $d/2$ elements in the second half of B_i, let S' denote the set of elements that remain in S that are smaller than at least one of these $d/2$ elements. Since the output of Window-Sort$(S, 4d, d)$ has maximum dislocation at most $d/2$, we know that $|S'| \leq d/2$ Moreover, the elements in S' are a subset of the smallest $d/2$ elements that remain in S and there are no elements in $(A_1 \cup A_2) - S$ smaller than the elements in S' (since S includes the $3d/2$ smallest elements in A_1 and A_2, respectively. Thus, all the elements in S' will be included in the subarray, B_{i+1}, of d elements output in merge step $i+1$. In addition, a symmetric argument applies to the first $d/2$ elements with respect to the d elements in B_{i-1}. Therefore, the output of the merge of A_1 and A_2 will have maximum dislocation at most $3d/2$ with the claimed probability. ◀

Window-merge-sort clearly runs in $O(n \log^2 n)$ time. This gives us the following.

▶ **Theorem 4.** *Given an array, A, of n distinct comparable elements, one can deterministically sort A in $O(n \log^2 n)$ time subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p., assuming that the block size B is at least $\log n$.*

This method is not data oblivious, however. For example, in a merge of two subarrays, A_1 and A_2, if each element in A_1 is less than all the elements in A_2, then with high probability the merge will take almost all the elements from A_1 before taking any elements from A_2.

4 Window-Oblivious-Merge-Sort

In this section, we describe a deterministic data-oblivious sorting algorithm that can tolerate noisy comparisons, which uses our data-oblivious window-sort only for small subarrays. Our method is an adaptation of the classic odd-even merge-sort algorithm [1] to the noisy comparison model, and it runs in $O(n \log^3 n)$ time, and achieves a maximum dislocation of $O(\log n)$, set in the analysis. We give our algorithm in Algorithm 3, with $d = c \log n$, where c is a constant set in the analysis.

Note that, assuming d is $O(\log n)$, the running time for window-merge is characterized by the recurrence, $T(n) = 2T(n/2) + n \log n$, which is $O(n \log^2 n)$; hence, the running time for window-odd-even-sort is characterized by the recurrence, $T(n) = 2T(n/2) + n \log^2 n$, which is $O(n \log^3 n)$.

The correctness of window-merge is proved using induction and the 0-1 principle, which is that if a data-oblivious algorithm can sort an array of 0's and 1's, then it can sort any array[1] [18]. Let n be a power of 2, and consider the elements of each of A_1 and A_2 arranged in two columns with even indices in the left column and odd indices in the right column. (See Figure 1.) By the 0-1 principle, if A_1 and A_2 each have maximum dislocation at most d, then, for each arrangement of A_1 and A_2, the difference between the number of 1's in the left column and the number of 1's in the right column is at most $d+1$.

[1] It is straightforward to show that the 0-1 principle holds for our noisy sorting setting as well.

Algorithm 3 Window-Odd-Even-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, d$)

```
if N ≤ 6d then
    return Window-Sort(A, 4d, d)
Divide A into two subarrays, A_1 and A_2, of roughly equal size
Window-Odd-Even-Sort(A_1, n, d)
Window-Odd-Even-Sort(A_2, n, d)
B ← Window-Merge(A_1, A_2, d)
return B

Window-Merge(A_1, A_2, d):
if |A_1| + |A_2| ≤ 6d then
    return Window-Sort(A_1 ∪ A_2, 4d, d)
Let A_1^o (resp., A_1^e) be the subarray of A_1 of elements at odd (resp., even) indices
Let A_2^o (resp., A_2^e) be the subarray of A_2 of elements at odd (resp., even) indices
B_1 ← Window-Merge(A_1^e, A_2^e, d)
B_2 ← Window-Merge(A_1^o, A_2^o, d)
Let B be the shuffle of B_1 and B_2, so its even (resp., odd) indices are B_1 (resp., B_2)
for i = 0, 1, 2, ..., |B|/d do
    Window-Sort(B[id : id + 6d], 4d, d)
return B
```

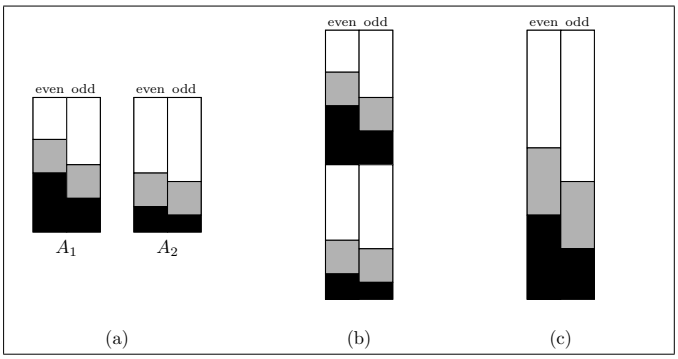

Figure 1 `Window-Merge` (a) Subarrays A_1 and A_2. (b) A before the merge. (c) A after the merge.

Next stack the two-column arrangement of A_1 on top of that for A_2 and note that our window-merge algorithm recursively sorts each column, which, by induction will each have maximum dislocation d. That is, by the 0-1 principle, each column will consist of a contiguous sequence of 0's, followed by a sequence of length at most $2d$ comprising a mixture of 0's and 1's, followed by a contiguous sequence of 1's. Further, by how we began our arrangement, the difference between the number of 1's in the left column and the number of 1's in the right column in the full arrangement of A_1 and A_2 is at most $2d + 2$. Thus, all the unsortedness is confined to a region of at most $4d + 2$ consecutively-indexed elements in the merged sequence, which are then completely contained in a region of $5d$ consecutively-indexed elements that begin at a multiple of d. Our window-merge method is guaranteed to call window-sort for this region of elements, bringing its maximum dislocation to be at most d. We observe that there are other calls to window-sort as well, but these will not degrade the sortedness of this

region. Thus, the result is that the maximum dislocation of the merged list is at most d.

This gives us the following.

▶ **Theorem 5.** *Given an array, A, of n distinct comparable elements, one can deterministically and data-obliviously sort A in $O(n \log^3 n)$ time, subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p.*

We note that the only randomization here is in the comparison model. The algorithm for Theorem 5 is deterministic. If we are willing to use a randomized algorithm, however, we can achieve a faster running time.

5 Randomized Shellsort

In this section, we describe a randomized data-oblivious sorting method that runs in $O(n \log n)$ time. The method is the simple randomized Shellsort algorithm of Goodrich [10], which we review in an appendix in Algorithm 4. It is based on peforming region compare-exhanges between subarrays of equal size, which, for a constant $c \geq 1$ set in the analysis, consists of constructing c random matchings between the elements of the two subarrays and performing compare-exchange operations between the matched elements. We study the dislocation reduction properties of randomized Shellsort empirically.

6 Annealing Sort

We briefly review here the annealing sort algorithm (see Algorithm 5 in an appendix), first introduced by Goodrich [11], which is a randomized data-oblivious sorting algorithm, and uses the simulated annealing meta-heuristic that involves following an **annealing schedule** defined by a **temperature sequence** $T = (T_1, T_2, \ldots T_t)$ and a **repetition sequence** $R = (r_1, r_2, \ldots, r_t)$. This algorithm essentially uses a randomized round-robin strategy of scanning the input array A and performing, for each $i = 1, 2, \ldots, n$, a compare-exchange operation between $A[i]$ and $A[s]$ where s is a randomly chosen index not equal to i. At each round j, the temperature T_j is then used to determine how far apart the candidate comparison elements with indices i and s should be at each time step. Following the simulated annealing metaheuristic, the temperatures in the annealing schedule decrease over time, and each random choice is repeated r_j number of times in round j. In our experiments, we follow the same three-phase annealing schedule used in the analysis of this algorithm in [11].

7 Experiments

To empirically test the performance of our algorithms under persistent noisy errors, we implemented each of the algorithms described in Sections 2–6, along with RIFFLESORT, which is a non-data-oblivious noisy sorting algorithm introduced by Geissman, Leucci, Liu, and Penna [8] that we review in an appendix in Algorithm 6. We also compare our algorithms to the standard and well-known insertion sort, randomized quicksort, and Shellsort [29] algorithms, e.g., see [3, 13]. For completeness, we include pseudo-code for these classic algorithms in an appendix in Algorithm 7.

We have also considered a variant of randomized Shellsort, which we denote by RANDOMIZEDSHELLSORTNO2S3S that does not include the 2 hop and 3 hop passes (lines 7-8 in Algorithm 4), as we do not think that they are necessary for the algorithm to perform well in practice. For standard Shell sort, we used the Pratt sequence [26], which uses a gap

sequence consisting of all products of powers of 2 and 3 less than the array size, and we denote this algorithm by shellSortPratt.

Parameter configurations. The riffleSort algorithm uses a parameter c to determine the group sizes during noisy binary search. Geissmann, Leucci, Liu, and Penna [8] assume $c = 10^3$ in their analysis; however, we set $c = 5$ so that the algorithm works with the input sequence sizes we use. We also set a parameter, h, of riffle-sort, which affects the height of the noisy binary search tree, to be $\log(\lfloor \frac{n+1}{5d} \rfloor)$, where d is the maximum dislocation of the input sequence given to the noisy binary search tree. For all other parameters, we follow the values used in [8]. We have also made a significant and potentially risky modification to the noisy binary search algorithm described by Geissmann, Leucci, Liu, and Penna [8] so that it works in a practical setting. In particular, while the original description of this subroutine fixes an upper bound $\tau = \lfloor 240 \log n \rfloor$ on the total number of steps performed in a noisy binary search random walk, we found that this resulted in unreasonably long running times for the input sequence sizes we used, and we instead lowered this upper bound to $\tau = \lfloor 7 \log n \rfloor$ in our implementation. Despite using lower τ, the algorithm surprisingly produces good dislocation bounds, while taking significantly less time. Because of its reliance on noisy binary searching, riffle-sort is not data-oblivious, so we used its performance as the best achievable empirical dislocation bounds, which our data-oblivious methods compare against.

For annealing sort, we follow the annealing schedule and constants used in [4], which defines additional parameters h, g_{scale}, and finds suitable values for them alongside the existing parameters c and q defined by Goodrich [11], all of which affect the temperature and repetition schedules used in the algorithm; hence, we set $h = 1$, $g_{scale} = 0$, $c = 10$, and $q = 1$.

For windowMergeSort and windowOddEvenMergeSort, we set $d = \log n$. Though a larger constant multiple of $\log n$ is required for the theoretical proofs of these algorithms, we found that this wasn't necessary in practice; in fact we observed that $d = \log n$ resulted in lower inversions and dislocations in our experiments.

Lastly, for windowSort, we set $d_1 = n/2$ and $d_2 = \log n$, and for randomizedShellSort, we set $c = 4$.

Experimental setup. We implemented each algorithm in C++[2], and compared the performance of each algorithm by measuring the total dislocations, maximum dislocations, and the number of inversions of the output arrays, as well as the total number of pairwise comparisons that were done. Each data point in the following plots correspond to the average of 5 runs of the algorithm with random input sequences of integers.

To implement noisy persistent comparisons, we make use of tabulation hashing [22, 30]. In our tabulation hashing setting, we let f denote the number of bits to be hashed, and $s \leq f$ be a block size, and $t = \lceil f/s \rceil$ be the number of blocks. We initialize a two dimensional $t \times 2^s$ array, A, with random q bit integers. Given a key, c, with f bits, we partition f into t blocks of s bits. For our experiments, we set $f = 64$, $s = 8$, and $q = 14$. If c_i represents the i-th block, the hash value $h(c)$ will be derived using the lookup table as follows:

$$h(c) = A[0][c_0] \oplus A[1][c_1] \oplus \cdots \oplus A[t][c_t]$$

[2] Our implementations of all algorithms can be found at `https://github.com/UC-Irvine-Theory/NoisyObliviousSorting`.

For simulating a noisy comparison, given two 4-Byte Integers, $x < y$, we first concatenate the numbers to get the key, $c = (x \cdot 2^{32}) + y$. Then, we hash c to derive $h(c)$, a random $q = 14$ bit integer. We determine that the comparison of these two numbers is noisy if and only if $h(c) \leq p \cdot 2^q$, where p is the noise probability, and output the result of the comparison accordingly.

We performed two sets of experiments: one with a varying probability p of comparison error and fixed input size $n = 32768$, and the other with varying input size n and a fixed probability $p = 0.03$ of comparison error. In our experiments, p takes on values $(2^{-1}, 2^{-2} \ldots, 2^{-10})$, and n takes on values $(2^{16}, 2^{15} \ldots, 2^7)$.

Results and analysis. We first consider experiments with varying p, and compare the maximum dislocations, total dislocations and inversions between each algorithm. We see from Figure 2 that all of the data-oblivious algorithms we describe in this paper have maximum and total dislocations that are inline with the theoretical optimal bounds of $O(\log n)$ and $O(n)$ respectively, as well as RIFFLESORT, particularly when $p < 0.1$. For example, we see that WINDOWODDEVENMERGESORT tends to be the best-performing data-oblivious algorithm for different values of p, achieving a total dislocation of at most $\approx 35\,300$, a maximum dislocation of at most 12, and at most $\approx 19\,200$ total inversions for values of $p < 0.1$.

We see that all of the non-standard algorithms tend to form an S-shaped curve, in terms of their dislocation bounds, such that as p starts to increase, the number of dislocations and inversions start to increase slowly, then there is a sharper increase after we reach $p > 0.1$. As expected, we see that the highest dislocation and inversions is when $p = 0.5$, which is the worst-case scenario for p (for any value of $p > 0.5$, reversing the output should result in a sequence with lower dislocation). In particular, as p goes from $1/32$ to $1/2$, all of the non-standard algorithms go from having up to 100 maximum dislocation and $\approx 28\,900$ total dislocation to having up to $\approx 30\,000$ maximum dislocation and ≈ 345 million total dislocation. These results match our theoretical analyses in this paper, as we assume that $p \leq 1/16$ in order to prove bounds for the dislocation. The proof for the version of RIFFLESORT we use assumes similar bounds for p [8]. On the other hand, we see that our implementations of insertion sort, quick sort, and Shell sort do not have the tendency to form an S-curve, and their inversion and dislocation counts are significantly higher compared to our algorithms.

In Figure 3, we see the effect of varying p and n on the number of comparisons made during the algorithm. We see that the number of comparisons tends to grow smaller as p increases in RIFFLESORT, INSERTIONSORT and SHELLSORTPRATT. When the input size is varied, we see that RIFFLESORT, INSERTIONSORT and QUICKSORT use the fewest number of comparisons. Notably, we see that RANDOMIZEDSHELLSORT (and its variant without 2 and 3-hop passes), as well as ANNEALINGSORT, are the best-performing data-oblivious algorithms in terms of the number of comparisons. Overall, we found that RIFFLESORT was the best-performing algorithm in both sets of experiments; however, it uses the noisy binary search subroutine and is thus not a data-oblivious algorithm.

We also consider how the dislocation is distributed across the output array for each algorithm. In Figure 4, we see the average dislocation across different array indices for 5 runs of each algorithm with input sequences of size 16384, and $p = 0.03$. For each output array, we grouped the indices into 128 bins and took the average dislocation inside each bin. From this figure we can see the significant difference in dislocation counts between the standard sorting algorithms insertion sort, Shellsort and quick sort, compared to the other algorithms we implemented. All of the standard sorting algorithms have bins with over 2000 dislocation

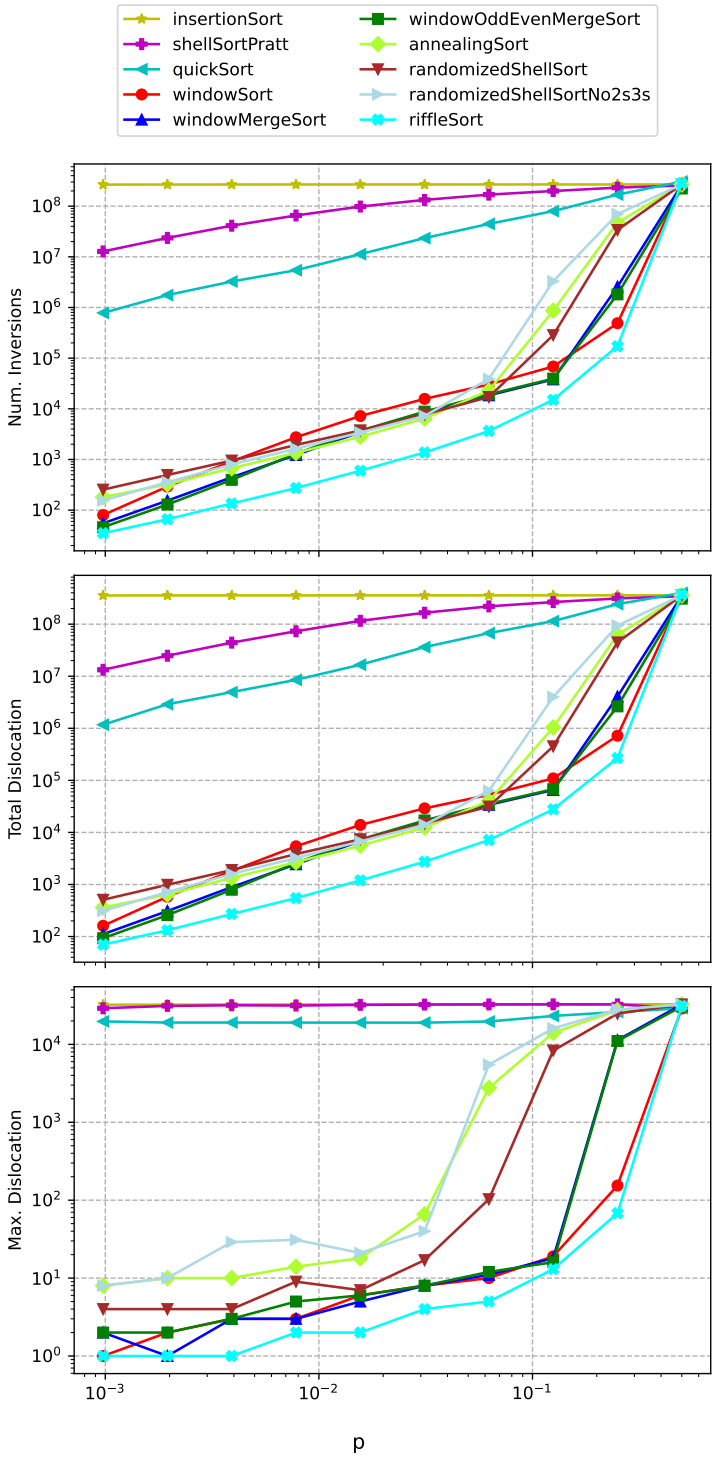

Figure 2 Effect of varying the comparison error probability p on the inversion and dislocation counts, with input sequences of size 32768.

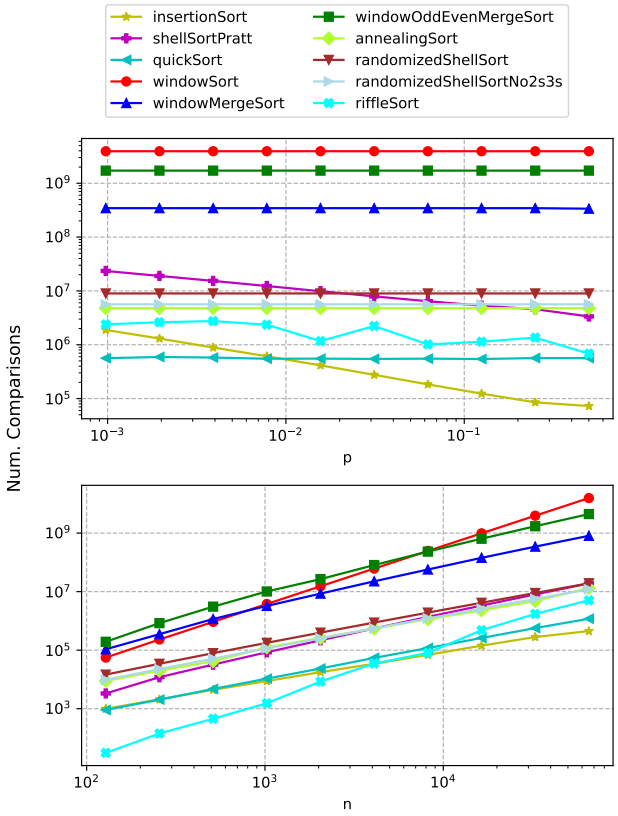

Figure 3 Effect of varying the comparison error probability p and the input size n on the number of comparisons.

on average, whereas none of the other algorithms have any bins with over 1.2 dislocations on average, with RIFFLESORT having less than 0.2 dislocations on average accross all of its bins.

While the distribution of dislocation is similar accross most algorithms, we see that insertion sort has most of its dislocation at the two ends of the array, whereas QUICKSORT has a few bins with high dislocation and has lower dislocation for most of the remaining bins.

8 Conclusions and Future Work

We introduced the sorting algorithms Window-Merge-Sort and Window-Odd-Even-Sort, both of which are tolerant to noisy comparisons, with the latter also being data-oblivious, with the key difference from existing algorithms being that we do not require use of a noisy binary search subroutine for either algorithms. We then provided both theoretical and experimental analyses, comparing our algorithms to some standard well-known sorting algorithms, and saw that our algorithms perform well in a practical setting as well. Interestingly, we found that the data-oblivious algorithms Annealing sort and Randomized Shellsort performed quite well under noisy comparisons in our experiments, though we have not provided a theoretical analysis for either of these algorithms. Therefore one possible direction for future work could be to prove similar bounds for these two algorithms.

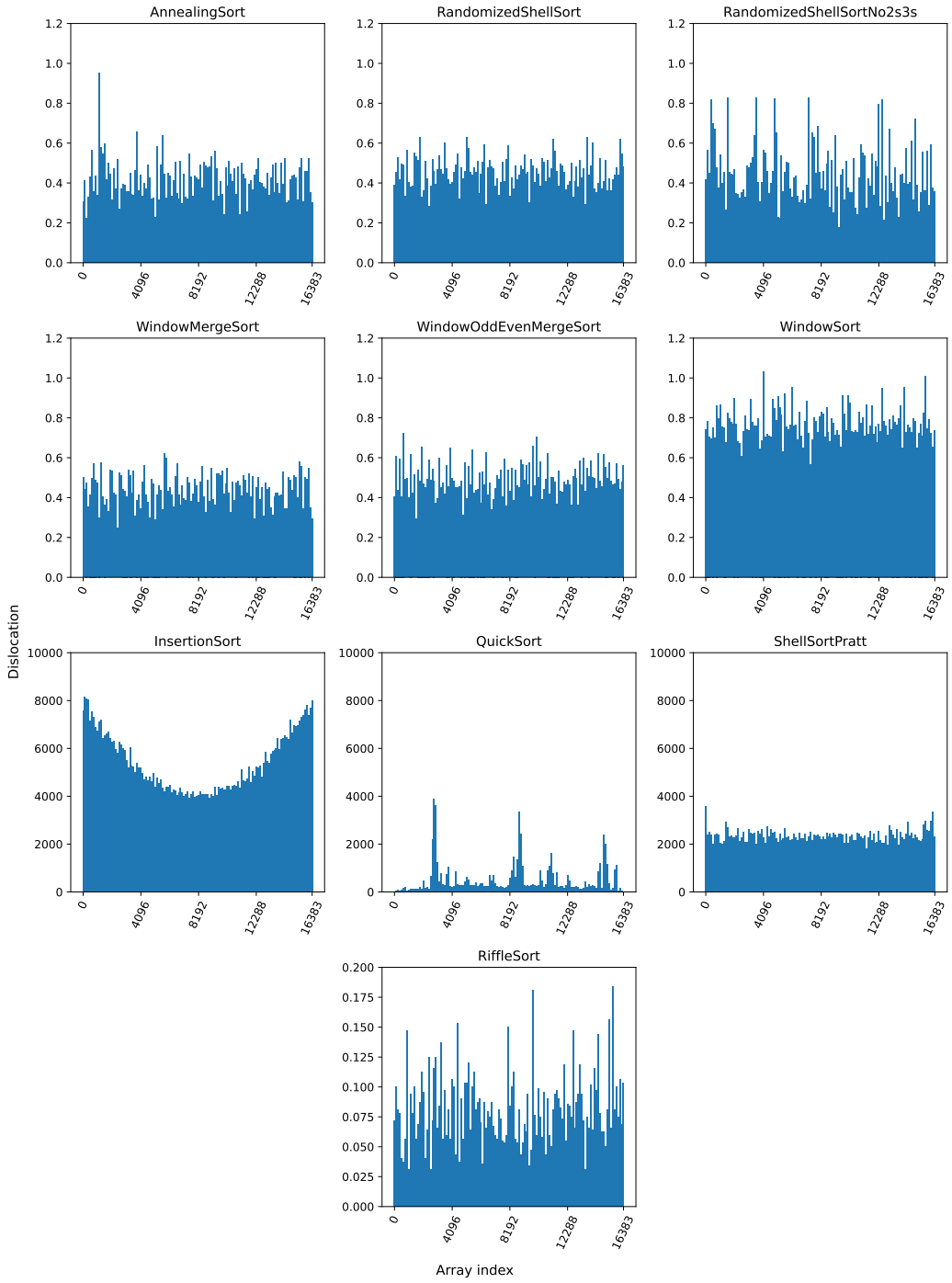

Figure 4 Averaged dislocation counts at different array indices over 5 runs for each algorithm on input sequences of size 16384, and $p = 0.03$. Each bar in the histogram corresponds to a bin of 128 indices.

References

1 Kenneth E Batcher. Sorting networks and their applications. In *Proc. of the Spring Joint Computer Conference (AFIPS)*, pages 307–314. ACM, 1968. URL: http://doi.acm.org/10.1145/1468075.1468121, doi:10.1145/1468075.1468121.

2 Mark Braverman and Elchanan Mossel. Noisy sorting without resampling. In *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 268–276, 2008.

3 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4/e edition, 2022.

4 Kris Vestergaard Ebbesen. On the practicality of data-oblivious sorting. Master's thesis, Aurhus Univ., Denmark, 2015.

5 Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994.

6 Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In David Naccache, editor, *Topics in Cryptology CT-RSA*, pages 457–471. Springer, 2001.

7 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Sorting with recurrent comparison errors. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th Int. Symp. on Algorithms and Computation (ISAAC)*, volume 92 of *LIPIcs*, pages 38:1–38:12, 2017. doi:10.4230/LIPIcs.ISAAC.2017.38.

8 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal sorting with persistent comparison errors. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th European Symposium on Algorithms (ESA)*, volume 144 of *LIPIcs*, pages 49:1–49:14, 2019.

9 Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal dislocation with persistent errors in subquadratic time. *Theory of Computing Systems*, 64(3):508–521, 2020. This work appeared in preliminary form in STACS'18.

10 Michael T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, December 2011. URL: http://doi.acm.org/10.1145/2049697.2049701, doi:10.1145/2049697.2049701.

11 Michael T. Goodrich. Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons. *Algorithmica*, pages 1–24, 2012. URL: http://dx.doi.org/10.1007/s00453-012-9696-5, doi:10.1007/s00453-012-9696-5.

12 Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *46th ACM Symposium on Theory of Computing (STOC)*, page 684–693, 2014. doi:10.1145/2591796.2591830.

13 Michael T Goodrich and Roberto Tamassia. *Algorithm Design and Applications*, volume 363. Wiley, 2015.

14 Richard M Karp and Robert Kleinberg. Noisy binary search and its applications. In *18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 881–890, 2007.

15 Claire Kenyon-Mathieu and Andrew C Yao. On evaluating boolean functions with unreliable tests. *International Journal of Foundations of Computer Science*, 1(01):1–10, 1990.

16 Kamil Khadiev, Artem Ilikaev, and Jevgenijs Vihrovs. Quantum algorithms for some strings problems based on quantum string comparator. *Mathematics*, 10(3):377, 2022.

17 Rolf Klein, Rainer Penninger, Christian Sohler, and David P. Woodruff. Tolerant algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *European Symposium on Algorithms (ESA)*, pages 736–747. Springer, 2011.

18 Donald E Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

19 Tom Leighton, Yuan Ma, and C. Greg Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54(2):265–304, 1997. doi:10.1006/jcss.1997.1470.

20 Wen Liu, Shou-Shan Luo, and Ping Chen. A study of secure multi-party ranking problem. In *Eighth ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, volume 2, pages 727–732, 2007. doi:10.1109/SNPD.2007.367.

21 Cheng Mao, Jonathan Weed, and Philippe Rigollet. Minimax rates and efficient algorithms for noisy sorting. In Firdaus Janoos, Mehryar Mohri, and Karthik Sridharan, editors, *Proceedings of Algorithmic Learning Theory*, volume 83 of *Proceedings of Machine Learning Research*, pages 821–847, 2018.

22 Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):1–50, jun 2012. doi:10.1145/2220357.2220361.

23 Andrzej Pelc. Searching with known error probability. *Theoretical Computer Science*, 63(2):185–202, 1989. doi:10.1016/0304-3975(89)90077-7.

24 Andrzej Pelc. Searching games with errors—fifty years of coping with liars. *Theoretical Computer Science*, 270(1):71–109, 2002. doi:https://doi.org/10.1016/S0304-3975(01)00303-6.

25 Nicholas Pippenger. On networks of noisy gates. In *26th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 30–38, 1985. doi:10.1109/SFCS.1985.41.

26 Vaughan Ronald Pratt. *Shellsort and sorting networks*. PhD thesis, Stanford University, Stanford, CA, USA, 1972.

27 Alfréd Rényi. On a problem in information theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 6:505–516, 1961. See https://mathscinet.ams.org/mathscinet-getitem?mr=0143666.

28 R.L. Rivest, A.R. Meyer, D.J. Kleitman, K. Winklmann, and J. Spencer. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20(3):396–404, 1980. doi:https://doi.org/10.1016/0022-0000(80)90014-8.

29 D. L. Shell. A high-speed sorting procedure. *Comm. ACM*, 2(7):30–32, July 1959. URL: http://doi.acm.org/10.1145/368370.368387, doi:10.1145/368370.368387.

30 Mikkel Thorup. Fast and powerful hashing using tabulation. *Commun. ACM*, 60(7):94—-101, jun 2017. doi:10.1145/3068772.

31 Ziao Wang, Nadim Ghaddar, and Lele Wang. Noisy sorting capacity. *arXiv*, abs/2202.01446, 2022. arXiv:2202.01446.

32 Ya Xu, Nanyu Chen, Addrian Fernandez, Omar Sinno, and Anmol Bhasin. From infrastructure to culture: A/B testing challenges in large scale social networks. In *21th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 2227–2236, 2015. doi:10.1145/2783258.2788602.

33 Andrew C. Yao. Protocols for secure computations. In *23rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 160–164, 1982. doi:10.1109/SFCS.1982.38.

34 Andrew C. Yao and F. Frances Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14(1):120–128, 1985. doi:10.1137/0214009.

A Some Existing Sorting Algorithms

In this section, we review some existing sorting algorithms that we included in our tests.

A.1 Randomized Shellsort

The first existing sorting algorithm we review is the randomized Shellsort of Goodrich [10], which we give in Algorithm 4. This algorithm is data oblivious.

Algorithm 4 Random-Shellsort($A = \{a_0, a_1, \ldots, a_{n-1}\}$)

```
for o = n/2, n/2^2, n/2^3, ..., 1 do
    Let A_i denote subarray A[io .. io + o − 1], for i = 0, 1, 2, ..., n/o − 1.
    begin a shaker pass
        Region compare-exchange A_i and A_{i+1}, for i = 0, 1, 2, ..., n/o − 2.
        Region compare-exchange A_{i+1} and A_i, for i = n/o − 2, ..., 2, 1, 0.
    begin an extended brick pass
        Region compare-exchange A_i and A_{i+3}, for i = 0, 1, 2, ..., n/o − 4.
        Region compare-exchange A_i and A_{i+2}, for i = 0, 1, 2, ..., n/o − 3.
        Region compare-exchange A_i and A_{i+1}, for even i = 0, 1, 2, ..., n/o − 2.
        Region compare-exchange A_i and A_{i+1}, for odd i = 0, 1, 2, ..., n/o − 2.
```

A.2 Annealing Sort

The next existing sorting algorithm we review is the annealing-sort method of Goodrich [11], which we review in Algorithm 5. This algorithm is also data oblivious.

Algorithm 5 Annealing-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, T, R$)

```
for j = 1, 2, ..., t do
    for i = 1, ..., n − 1 do
        for k = 1, 2, ..., r_j do
            Let s be a random integer in the range [i + 1, min(n, i + T_j)]
            if A[i] > A[s] then
                Swap A[i] and A[s]
    for i = n, n − 1, ... 2 do
        for k = 1, 2, ..., r_j do
            Let s be a random integer in the range [max(1, i − T_j), i − 1]
            if A[s] > A[i] then
                Swap A[i] and A[s]
```

A.3 Riffle Sort

We include pseudo-code for the riffle-sort method of Geissman, Leucci, Liu, and Penna [8], which we review in Algorithm 6, for $k = (\log n)/2$ and $\gamma = 2020$. The pseudo-code

uses a subroutine TEST(x, v) (see [8], Definition 1), which checks whether some element x approximately belongs to the interval pointed to by some node v in the noisy binary search tree, which is the main place where this algorithm is not data oblivious.

Algorithm 6 Riffle-Sort$(A = \{a_0, a_1, \ldots, a_{n-1}\})$

```
T_0, T_1, ..., T_k ← Partition(A)
S_0 ← WindowSort(T_0, √n, 1)
for j = 1, ..., k + 1 do
    S_i ← Merge(S_i, T_{i−1})
    S_i ← WindowSort(S_i, 9γ log n, 1)
return S_{k+1}

Partition(A) :
for i = k, ..., 1 do
    T_i ← 2^{i−1}√n elements chosen u.a.r. from A \ {T_{i+1}, ..., T_k}
T_0 ← remaining √n elements in A
return T_0, ..., T_k

Merge(A, B) :
foreach x ∈ B do
    rank_x ← NoisyBinarySearch(A, x)
Insert simultaneously all elements x ∈ B according to rank_x into A
return A

NoisyBinarySearch(A, x) :
Construct noisy binary search trees T_0, T_1 as described in [8], section 3.1.
for j = 0, 1 do
    t ← 7⌊log |A|⌋
    curr ← T_j.root
    while t > 0 do
        if curr is a leaf of T_j then
            return curr
        Call test(x, c) for each child c of node curr.
        if exactly one of the calls pass for some child node c then
            curr ← c
        else
            // all tests have failed
            curr ← curr.parent
        t ← t − 1
return an arbitrary index                    // both walks have timed out
```

A.4 Well-known Sorting Algorithms

For the sake of completeness, we also include pseudo-code for the well-known insertion-sort, quick-sort, and Shellsort algorithms, in Algorithm 7. None of these three algorithms are

data oblivious. One can modify insertion-sort to be data oblivious, however, by continuing the compare-and-swap inner loop process to the beginning of the array in every iteration. Likewise, the Shellsort algorithm can also be modified to be data oblivious in the same manner, since its inner loop is essentially an insertion-sort carried out across elements separated by the gap distance in each iteration.

Algorithm 7 Well-known sorting algorithms, assuming the input array, A, is of size n and indexed starting at 0. We sort A by calling Insertion-Sort(A, n), Quick-sort$(A, 0, n-1)$, or Shell-sort(A, n, G), where G is a non-increasing ***gap sequence*** of positive integers less than n, such as the Pratt sequence [26], which consists of all products of powers of 2 and 3 less than n.

Insertion-sort(A, n):
for $i \leftarrow 1, \ldots, n-1$ **do**
  $j \leftarrow i$
  while $j > 0$ ***and*** $A[j-1] > A[j]$ **do**
    Swap $A[j]$ and $A[j-1]$
    $j \leftarrow j-1$

Quick-sort(A, l, h):
if $l < h$ **then**
  Choose x uniformly at random from the subarray $A[l..h]$
  Partition A into $A[l..p-1]$, $A[p]$, and $A[p+1..h]$, where $A[i] < x$ for $i \in [l, p-1]$, $A[p] = x$, and $A[i] \geq x$ for $i \in [p+1, h]$ (if these subarrays exist)
  Quick-sort$(A, l, p-1)$
  Quick-sort$(A, p+1, h)$

Shell-sort(A, n, G):
foreach $g \in G$ **do**
  for $i \leftarrow g, \ldots, n-1$ **do**
    $j \leftarrow i$
    while $j \geq g$ ***and*** $A[j-g] > A[j]$ **do**
      Swap $A[j]$ and $A[j-g]$
      $j \leftarrow j-g$