Sorting and searching in faulty memories *

Irene Finocchi §   *Giuseppe F. Italiano* ¶

Abstract

In this paper we investigate the design and analysis of algorithms resilient to memory faults. We focus on algorithms that, despite the corruption of some memory values during their execution, are nevertheless able to produce a correct output at least on the set of uncorrupted values. In this framework, we consider two fundamental problems: sorting and searching. In particular, we prove that any $O(n \log n)$ comparison-based sorting algorithm can tolerate the corruption of at most $O((n \log n)^{1/2})$ keys. Furthermore, we present one comparison-based sorting algorithm with optimal space and running time that is resilient to $O((n \log n)^{1/3})$ memory faults. We also prove polylogarithmic lower and upper bounds on resilient searching.

Keywords: combinatorial algorithms, sorting, searching, memory faults, memory models, computing with unreliable information.

1 Introduction

Some of today's applications run on computer platforms with large and inexpensive memories, which are also error-prone. Physical defects and manufacturing errors can result in hard memory faults, i.e., faults that can permanently affect the correct behavior of memory cells. Environmental conditions such as cosmic rays and alpha particles can also temporarily affect the memory behavior, resulting in unpredictable, random, independent failures known in the literature as soft memory errors [26, 36, 38]. Tests have identified several specific design factors which impact soft error rates, including higher density of memory chips, lower voltage, and higher speed: all these factors are in line with the trend observed in the design of today's highest-speed memory technologies, and thus soft error rates are expected to rise continuously. Implementing error checking and correction circuitry at the board level could contrast this phenomenon, but would also impose non-negligible costs in terms of performance (as much as 33%), size (20% larger areas), and money (10% to 20% more expensive chips). Furthermore, error recovery circuitry can typically correct single bit errors and detect - but not correct - double bit errors: hence, it would not guarantee complete fault coverage.

Table 1 illustrates the mean time between failures of different computing platforms using currently available memory technologies. The table shows an increasing number of failures as

*This work has been partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet") and by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT ("Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments"). A preliminary version of this work was presented at the 36th ACM Symposium on Theory of Computing (STOC'04) [18].

§Dipartimento di Informatica, Università degli Studi di Roma "La Sapienza", Via Salaria 113, 00198 Rome, Italy. E-mail: finocchi@di.uniroma1.it.

¶Dipartimento di Informatica, Sistemi e Produzione, Università degli Studi di Roma "Tor Vergata", Via del Politecnico 1, 00133 Rome, Italy. E-mail: italiano@disp.uniroma2.it.

Main memory size	Failures in time per Mbit	Mean time between failures
512 MB	1000	244 hours
1 GB	1000	122 hours
16 GB	1000	7.6 hours
64 GB	1000	1.9 hours
1 TB	1000	7 minutes
512 MB	5000	48.8 hours
1 GB	5000	24.4 hours
16 GB	5000	1.5 hours
64 GB	5000	22 minutes
1 TB	5000	1.4 minutes

Table 1: Mean time between failures of different computing platforms as a function of the main memory size and of the number of failures in time: 1000 to 5000 failures in time per Mbit (i.e., 1000 to 5000 errors per 10^9 hours of use of one Mbit) is the standard soft error rate for modern memory devices [36].

the memory size becomes larger: a system with Terabytes of memory, such as a large cluster of computing platforms with a few Gigabytes per node, would experience one bit error every few minutes. The demand for larger capacities at low cost arises in many applications. For instance, Web search engines use low cost memories to store their huge data sets (typically of the order of Terabytes), including inverted indices which have to be maintained sorted for fast document access: for such large data structures, even a small failure probability can result in few bit flips in the index, that may become responsible of erroneous answers to keyword searches [13, 20]. Another domain of interest is fault-based cryptanalysis: some novel optical and electromagnetic perturbations attacks work by manipulating the non-volatile memory of cryptographic devices, so as to induce very timing-precise controlled transient faults on given individual bits [5, 32, 35]. This forces the device to output a wrong cipher text that may allow the attacker to determine the secret key used during the encryption.

Computing in the presence of memory faults seems thus important in many practical scenarios, when the correctness of the underlying algorithms may be jeopardized by the appearance of even very few memory faults. Consider for instance mergesort: during the merge step, errors may propagate due to corrupted keys (having value larger than the correct one). Even in the presence of very few errors, in the worst case as many as $\Theta(n)$ keys may be out of order in the output sequence, where n is the number of keys to be merged. Informally, we say that an algorithm is *resilient to memory faults* if, despite the corruption of some memory values before or during its execution, the algorithm is nevertheless able to get a correct output on the set of uncorrupted values. In this paper we investigate the design and analysis of algorithms which are resilient to memory faults, and present upper and lower bounds for two basic problems: sorting and searching. Before presenting in detail our results, we discuss the relevant related work and highlight our faulty memory model.

1.1 Related work

The problem of computing with unreliable information has been investigated in a variety of settings. In the *liar model*, started by Ulam–Rényi's game [34, 37], an algorithm asks comparison questions answered by a possibly lying adversary. Many works address the problem of searching with lies [2, 6, 12, 14, 22, 28, 29, 30]. Even when the number of lies grows proportionally with the number of questions (linearly bounded model), searching can be solved to optimality: Borgstrom and Kosaraju [6], improving over [2, 12, 29], designed an $O(\log n)$ searching algorithm. Problems such as sorting and selection have instead drastically different bounds. Lakshmanan *et al.* [23] proved that $\Omega(n \log n + k \cdot n)$ comparisons are necessary for sorting when at most k lies are allowed. The best known $O(n \log n)$ algorithm tolerates only $O(\log n/\log\log n)$ lies, as shown by Ravikumar in [33]. In the linearly bounded model, an exponential number of questions is necessary even to test whether a list is sorted [6]. Feige *et al.* [14] studied a probabilistic model and presented a sorting algorithm correct with probability $\geq 1-q$ that requires $\Theta(n \log(n/q))$ comparisons. Lies are well suited at modeling transient ALU failures, such as comparator failures. Since memory data get never corrupted in the liar model, fault-tolerant algorithms may exploit *query replication* strategies. We remark that this is not the case in our faulty-memory model.

Other faults that have been considered in the literature are *destructive faults*, which have been first investigated in the context of fault-tolerant sorting networks [3, 24, 25, 39]. In this model, one assumes that comparators can be faulty and can possibly destroy one of the input values. Assaf and Upfal [3] present an $O(n \log^2 n)$-size sorting network tolerant (with high probability) to random destructive faults; this bound is tight, as proved later by Leighton and Ma [24]. We note that the Assaf-Upfal network makes $\Theta(\log n)$ copies of each item, using fault-free data replicators. Using *data replication* or some kind of *redundancy* is indeed a natural approach to protect against destructive memory faults. Redundant arrays of inexpensive disks (RAID) [7] are a typical application of this technique. Aumann and Bender [4] also use pointer redundancy to make pointer-based data structures resilient to memory faults.

In general, using data replication can be very inefficient in highly dynamic scenarios when the objects to be managed are large and complex, such as large database records or long strings: copying such objects can indeed be very costly, and in some cases we might not even know how to accomplish this task. For instance, common libraries of algorithmic codes (e.g., the LEDA Library of Efficient Data Types and Algorithms [27] or the Standard Template Library [31]) typically implement sorting algorithms by means of indirect addressing methods: the objects to be sorted are accessed through pointers, which are moved around in memory instead of the objects themselves, and the algorithm relies on user-defined comparator functions. In these cases, the implementation of sorting algorithms assumes neither the existence of *ad hoc* functions for data replication nor the definition of suitable encoding mechanisms to maintain a reasonable storage cost.

Parallel models of computation with faulty memories have also been studied, e.g., in [8, 9, 10, 21]. Differently from this paper, these works assume the existence of a special fault-detection register that makes it possible to recognize memory errors upon access to a memory location. The aim is to design fast simulations of the fully operational parallel models on the faulty ones: the simulations described in [8, 9, 10, 21] are either randomized or deterministic, and operate with constant or logarithmic slowdown, depending on the model (PRAM or Distributed Memory Machine), on the nature of faults (static or dynamic, deterministic or

random), and on the number of available processors.

1.2 Our model

We have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a failure. In particular, the content of a location can change unexpectedly, i.e., faults may happen *at any time*: real memory faults are indeed highly dynamic and unpredictable [19]. Faulty values may be everywhere in memory, i.e., faults may happen *at any place*. Furthermore, since an entire memory bank may undergo a failure, more than one fault can be introduced at the same time, i.e., faults may happen *simultaneously*. We model this with a *faulty-memory random access machine*, i.e., a random access machine [1] whose memory locations may suffer from memory faults and thus may possibly corrupt the values they contain. We observe that in this model corrupted values are indistinguishable from correct ones. We will assume that the algorithms can exploit only $O(1)$ *reliable* memory words, whose content gets never corrupted: this is not restrictive, since at least registers can be considered fault-free. In particular, we assume throughout the paper that moving variables around in memory is an atomic operation: i.e., whenever we read some memory location and we copy it somewhere else immediately afterwards, the read operation will store that value in reliable memory, so that the written value equals the read value.

In the faulty-memory model considered in this paper, if each value were replicated k times, by majority techniques we could easily tolerate up to $(k-1)/2$ faults; however, the algorithm would present a multiplicative overhead of $\Theta(k)$ in terms of both space and running time. This implies, for instance, that in order to be resilient to $O(\sqrt{n})$ faults, a sorting algorithm would require $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space. The space may be improved using error-correcting codes, but at the price of a higher running time.

It seems thus natural to ask whether it is possible to design algorithms that do not exploit data replication in order to achieve resilience to memory faults: i.e., algorithms that do not wish to recover corrupted data, but simply to be correct on uncorrupted data, without incurring any time or space overhead. E.g., is it possible to sort the correct data in $O(n \log n)$ time and $O(n)$ space in the presence of polynomially many memory faults? In this paper we affirmatively answer this question.

1.3 Our results

Let $\delta \leq n$ denote an upper bound on the number of memory faults that may occur throughout the algorithm execution (note that δ may be a function of the input size n). We consider the following problems:

- *Resilient sorting.* We are given a set of n keys that need to be sorted. The value of some keys can be arbitrarily corrupted (either increased or decreased) during the sorting process. A sorting algorithm is *strongly resilient* if it correctly orders the set of uncorrupted keys. We remark that this is the best that we can achieve in the presence of memory faults: if keys get corrupted at the very end of the algorithm execution, we cannot prevent them from occupying wrong positions in the output sequence.

- *Resilient searching.* We are given a set of n keys on which we wish to perform membership queries. The keys are in increasing order, but some of them may be corrupted

(even during the searching process) and thus occupy wrong positions in the sequence. Let s be a key to be searched for. A *resilient* searching algorithm works as follows:

(1) if there is a correct key equal to s, it answers yes and returns the index of any key equal to s;

(2) if there is no key (either correct or faulty) equal to s, it answers no;

(3) if there are only faulty keys equal to s, the answer may be either yes or no.

One of the main difficulties in designing efficient resilient sorting and searching algorithms derives from the fact that positional information is no longer reliable in the presence of memory faults: for instance, when we search an array whose correct keys are in increasing order, it may be still possible that a faulty key in position x is smaller than some correct keys in positions $< x$, thus guiding the search towards a wrong direction.

The main results of this paper can be summarized as follows:

- We prove that any strongly resilient $O(n \log n)$ comparison-based sorting algorithm can tolerate the corruption of at most $O((n \log n)^{1/2})$ keys.

- We present a strongly resilient $O(n \log n)$ comparison-based sorting algorithm that tolerates $O((n \log n)^{1/3})$ memory faults.

Our lower bound implies that if we have to sort in the presence of $\omega((n \log n)^{1/2})$ memory faults, then we should be prepared to spend more than $O(n \log n)$ time. Both our lower and upper bounds are in fact more general, as we will show in Section 3 and in Section 4, respectively. Similarly, in Section 5 we prove polylogarithmic lower and upper bounds on resilient searching: we design an $O(\log n)$ time searching algorithm that can tolerate up to $O((\log n)^{1/2})$ memory faults, and we prove that any $O(\log n)$ time resilient searching algorithm can tolerate at most $O(\log n)$ memory faults.

All our algorithms are deterministic, do not make use of data replication, and use only $O(1)$ reliable memory. We remark that a constant-size reliable memory may be even not sufficient for a recursive algorithm to work properly: parameters, local variables, return addresses in the recursion stack may indeed get corrupted. This is not a problem if the recursion can be simulated by an iterative implementation using only a constant number of variables. The algorithms presented in this paper have this property, and we will thus use their iterative implementation.

The remainder of this paper is organized as follows. Section 2 introduces preliminary terminology for characterizing the resilience of sorting algorithms with respect to memory faults. In Section 3 we show how to sort in optimal space and running time in the presence of a polynomial number of faults, and in Section 4 we prove a lower bound for strongly resilient sorting in the comparison model. Section 5 presents the results for resilient searching, while Section 6 lists some concluding remarks.

2 Unordered sequences and resilient sorting algorithms

In this section we introduce a framework for studying the resilience of sorting algorithms to memory faults. We will denote by n the number of input keys, by $\delta \leq n$ an upper bound on the number of memory faults that can occur throughout the execution of the algorithm,

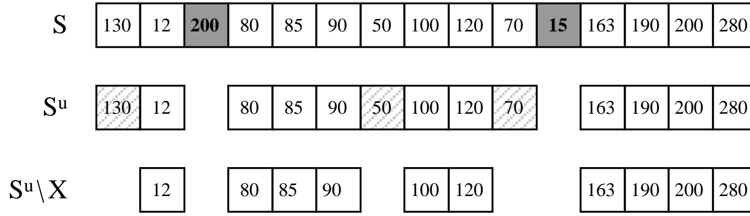

Figure 1: A sequence S with two corrupted keys (in boldface): S is 3-unordered since the removal of keys 130, 50 and 70 makes the uncorrupted keys in the remaining subsequence sorted.

and by $\alpha \leq \delta$ the actual number of faults that occur throughout a specific execution. We characterize the resilience of sorting algorithms to memory faults both in terms of *overhead* in their running times and in terms of *quality* of their solution.

Intuitively, the overhead is the *additional* time spent by the algorithm to cope with memory faults. In order to characterize the quality of the produced solution, we note that even correct keys may be not sorted in the output sequence due to the occurrence of memory faults, i.e., errors can propagate: we therefore characterize the quality of resilient sorting and merging algorithms in terms of *error propagation*, which is going to be defined next. Given a sequence of keys S, we denote by $S^u \subseteq S$ the subset of uncorrupted keys. The following definition will be useful throughout.

Definition 1 *Given an integer $k \geq 0$, a sequence S is k-*unordered *if:*

- *there exists a set $X \subseteq S^u$ such that $|X| = k$ and $S^u \setminus X$ is sorted;*
- *for any set $Y \subseteq S^u$ such that $|Y| < k$, $S^u \setminus Y$ is not sorted.*

An example of k-unordered sequence for $k = 3$ is shown in Figure 1. We remark that, according to Definition 1, a sequence in which only the corrupted keys appear in the wrong order is 0-*unordered*. With a slight abuse of notation, in the remainder of this paper we will say that a sequence is $O(f(\delta))$-unordered, for some function f, to indicate that it is k-unordered for $k = O(f(\delta))$.

Given a k-unordered sequence X of length n, it is always possible to obtain a 0-unordered subsequence of X in $O(n)$ time by removing $\Theta(k)$ keys from X. This can be easily done with the help of the Cook-Kim division algorithm [11]. We now show how to make this algorithm resilient to memory faults, i.e., capable of operating during the occurrence of memory faults.

Algorithm Purify. While we scan the sequence X sequentially, we build a stack S and a list D of discarded keys as follows. We maintain the top of stack S and the index i that scans X in the $O(1)$-size reliable memory. Initially, we push $X[1]$ onto the empty stack. At the i-th step, with $i > 1$, if $X[i]$ is larger than or equal to the top of S, we push $X[i]$ onto S. Otherwise, we have found an inversion and therefore either $X[i]$ or the top of the stack S must be out of order: in this case, we pop S, add the popped item and $X[i]$ to the list D of discarded keys, compute the maximum among the topmost $t = \min\{\delta + 1, |S|\}$ stack keys, and move it to the top of S. After n steps, we return S and D.

The reason for computing the maximum among the topmost stack keys when we pop stack S is that the key below the discarded top may have been corrupted since it was pushed onto S: namely, if its value was decreased since it was pushed, S could be no longer 0-unordered at the end of the algorithm execution. More formally, we can prove that algorithm `Purify` maintains the following invariant:

Invariant 1 (Stack Invariant) *Throughout the algorithm execution, the key on the top of stack S is larger than or equal to all the keys that have not been corrupted since they were pushed onto S.*

Proof. The proof proceeds by induction on the value of the index i that scans X. The base step, with the stack containing just $X[1]$, trivially holds. Assume that the invariant holds by induction at the beginning of the i-th step, $i > 1$. If $X[i] \geq top$, $X[i]$ is pushed onto the stack and the invariant still holds by transitivity: the top of stack S is indeed maintained in reliable memory, and thus a key cannot get corrupted as long as it is at the top of S. If $X[i] < top$, both $X[i]$ and the top of the stack are added to the list of discarded keys. Moreover, the algorithm moves to the top of S the maximum among the topmost t stack keys, where $t = \min\{\delta + 1, |S|\}$. Let m be this maximum. Since m is larger than or equal to the topmost t keys, the invariant remains true when $t = |S| \leq \delta + 1$. Let us now assume that $t = \delta + 1 < |S|$. In this case, at least one of the $(\delta + 1)$ keys considered must be correct, since there can be at most δ errors: let x be any such correct key. At the time when x was at the top of S, the invariant was true by induction, and therefore x must still be larger than or equal to all the uncorrupted keys below its position. Since $m \geq x$, the new top satisfies the invariant with respect to the entire stack. □

The running time of algorithm `Purify` is analyzed in the following lemma:

Lemma 1 *Algorithm* `Purify` *computes a 0-unordered subsequence S of a k-unordered sequence X of length n in $O(n + \delta \cdot (k + \alpha))$ worst-case time, where $\alpha \leq \delta$ is the actual number of memory faults introduced during the execution of* `Purify`*. S has length $\geq n - 2(k + \alpha)$, and only the keys corrupted during the execution of* `Purify` *may be unordered in S.*

Proof. By Invariant 1, the stack S contains a 0-unordered subsequence of X at the end of the execution, because only keys corrupted after they have been pushed onto the stack may be unordered. If ℓ is the number of pop operations performed, it is easy to see that the running time of `Purify` is $O(n + \ell \cdot \delta)$ and that the final stack has size at least $n - 2\ell$. To conclude the proof, we are left to show that $\ell \leq k + \alpha$. Assume by contradiction that $\ell > k + \alpha$. For each pop operation performed, the algorithm must have met one distinct pair of inverted keys: thus, the total number of inverted pairs encountered is at least ℓ. The assumption $\ell > k + \alpha$ would therefore imply either that X is not k-unordered or that more than α keys get corrupted during the execution of `Purify`, clearly a contradiction. □

In the rest of this section we show how to use the notion of k-unordered sequence in order to characterize the quality of resilient sorting algorithms. In particular, we measure the quality of the solution produced by a sorting/merging algorithm that runs in the presence of memory faults with the following definition.

Definition 2 *Let $\mathcal{A}$ be a sorting / merging algorithm that runs in the presence of α memory faults, $\alpha > 0$. We say that algorithm $\mathcal{A}$ has error propagation σ, with $\sigma \geq 0$, if in the worst case $\mathcal{A}$ produces $(\sigma \cdot \alpha)$-unordered sequences.*

We will call a sorting / merging algorithm that has error propagation $\sigma = 0$ (i.e., an algorithm that does not propagate errors) *strongly resilient*: such an algorithm will produce a 0-unordered sequence, i.e., a sequence in which only corrupted keys may be out of order. Similarly, we will call an algorithm with error propagation $\sigma > 0$ *weakly resilient.*

We now characterize the overhead and the quality of the solution of two basic merging algorithms.

Standard merging. Consider the classical merging algorithm, which repeatedly extracts the minimum among the first items of the two sorted n-length sequences A and B to be merged. The algorithm runs in $O(n)$ time and has therefore no overhead. However, its error propagation σ can be as large as $\Theta(n)$, as we are going to show next. Assume that all of the keys in A are smaller than those in B and that, just before merging, the first key of A gets corrupted and becomes larger than all the other values. Clearly, in this example we have only one memory fault, i.e., $\alpha = 1$. However, standard merging returns incorrectly all the n keys of B before any key of A: the output sequence is therefore $O(n)$-unordered, even in the presence of a single memory fault.

Naive strongly resilient merging. Consider the following variant of the classical merging algorithm: at each merge step, instead of taking the minimum among two top keys, take the minimum among the top $(\delta + 1)$ keys of each sequence, for a total of $(2\delta + 2)$ keys. Since there can be at most δ memory faults, at least one correct key per sequence must be looked up. This guarantees that the algorithm has error propagation $\sigma = 0$, at the price of a time overhead of $\Theta(\delta \cdot n)$.

The first example shows that the standard mergesort algorithm has obviously no running time overhead, but its error propagation σ can be $\Theta(n)$. At the other extreme, if we are willing to pay a large overhead, we can easily obtain a strongly resilient sorting algorithm with the help of naive strongly resilient merging. In order to avoid problems in the recursion stack, we use the standard iterative bottom-up implementation of mergesort, i.e., we sort all the sequences of length 2^i before any sequence of length 2^{i+1}, for $i = 1, 2, \ldots \lceil \log n \rceil$[1]. In this way we only need to maintain in $O(1)$ reliable memory words the length of the subsequence being sorted and the position of its left boundary. We will call `Naive-SR-Mergesort` this iterative sorting algorithm, which is analyzed in the following lemma.

Lemma 2 *Algorithm* `Naive-SR-Mergesort` *is strongly resilient and has overhead* $O(\delta \cdot n \log n)$ *on the running time. The overhead becomes* $O(\delta \cdot n)$ *when* $\delta = \Omega(n^\epsilon)$*, for some* $\epsilon > 0$.

Proof. The strong resilience of the iterative sorting algorithm derives from the strong resilience of the merging subroutine. The running time satisfies the recurrence

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(\delta \cdot n)$$

that can be solved with standard techniques. □

Table 2 summarizes the quality and overhead of these two basic algorithms.

[1]In this paper all logarithms are assumed to be to the base 2, unless explicitly noted otherwise.

algorithm	error propagation	overhead
`Standard-Mergesort`	$\Theta(n)$	0
`Naive-SR-Mergesort`	0	$O(\delta \cdot n \log n)$

Table 2: Quality of the solution and overhead of two basic sorting algorithms.

3 Strongly resilient sorting

In this section we describe a strongly resilient sorting algorithm that can tolerate up to δ memory faults in $O(n \log n + \alpha \cdot \delta^2)$ worst-case running time, where $\alpha \leq \delta$ is the actual number of keys corrupted during the sorting process. In particular, this yields an $O(n \log n)$ time algorithm that can tolerate $O((n \log n)^{1/3})$ memory faults. The algorithm is based on mergesort, and we first present two resilient merging subroutines with quite different characteristics: the first, presented in Section 3.1, has no asymptotic overhead, but may be unable to sort all of the correct keys; the second, presented in Section 3.2, is slower, yet strongly resilient. We will use both subroutines as building blocks for our strongly resilient sorting algorithm to be presented in Section 3.3. In the following, given a sequence S of n keys, we will denote by $S[i; j]$ the subsequence $\{S[i], S[i+1], \ldots, S[j]\}$, with $1 \leq i \leq j \leq n$.

3.1 Merging in linear time with error propagation $O(\delta)$

Let A and B be two n-length sequences to be merged. We assume that A and B are 0-unordered and we denote by α their total number of corrupted values, with $\alpha \leq \delta$. We count in α the values corrupted both at the beginning and during the merging process. Our weakly resilient merging, called `WR-Merge`, resembles the classical merging algorithm with the following modifications.

Algorithm WR-Merge. The algorithm scans A and B sequentially. Let i and j be the running indices on A and B, respectively. At each step, in addition to comparing $A[i]$ and $B[j]$ and advancing one of the two indices, the algorithm updates two additional variables, respectively called $wait_A$ and $wait_B$, initialized to 0: $wait_A$ (resp. $wait_B$) counts the number of increments of index j (resp. i) since the last increment of index i (resp. j). If $A[i]$ is added to the output sequence, $wait_A$ is reset to 0 and $wait_B$ is incremented by 1; otherwise, if $B[j]$ is added to the output sequence, $wait_B$ is reset to 0 and $wait_A$ is incremented by 1. We let the *wait* variables increase up to value $(2\delta + 1)$ at most. If this value is reached by either of them we do the following. If $wait_A = 2\delta + 1$ (the case $wait_B = 2\delta + 1$ is symmetric), we consider the window $W = A[i + 1; \min\{i + 2\delta + 1, n\}]$:

- if W contains less than $(2\delta + 1)$ values (i.e., if $i + 2\delta + 1 > n$), we simply reset $wait_A$ to 0;
- if W contains $(2\delta + 1)$ values (i.e., if $i + 2\delta + 1 \leq n$), we count the number t of its values that are smaller than $A[i]$: if $t \geq \delta + 1$, we output $A[i]$ and increment index i by 1. Independently of the value of t, we also reset $wait_A$ to 0.

The indices i and j, the two *wait* variables, and the counter t are all stored in the $O(1)$-size reliable memory.

Analysis. We first note that at any time one of the two *wait* variables must be zero. The merging process is thus divided into a sequence of A-*bursts* and B-*bursts*: during an A-burst, the algorithm keeps on adding to the output sequence values from A, until either $B[j] < A[i]$ or $(2\delta+1)$ values have been returned. Thus, during an A-burst $wait_A = 0$, index i advances, and $wait_B$ is incremented up to $(2\delta + 1)$. B-bursts are symmetric. If an A-burst is followed by a B-burst in the sequence, we call it a *final* A-*burst*. If an A-burst is followed by another A-burst in the sequence, we call the two bursts *consecutive*. The fact that bursts have length $\leq 2\delta + 1$ is crucial in proving that the algorithm has error propagation $O(\alpha \cdot \delta)$.

Lemma 3 *Given two* 0*-unordered sequences of total length* n*, algorithm* `WR-Merge` *merges the sequences in* $O(n)$ *time and returns an* $O(\alpha \cdot \delta)$*-unordered sequence, where* $\alpha \leq \delta$ *is the number of corrupted keys at the end of the algorithm execution.*

Proof. It is easy to see that the running time is $O(n)$. At each step we spend constant time, except when one of the *wait* variables gets value $(2\delta + 1)$: in this case we spend time $O(\delta)$, which can be amortized against the time spent to output the last $(2\delta + 1)$ elements.

We now show that at most $\alpha \cdot (2\delta + 1)$ keys can be in the wrong position in the output sequence. We charge the errors introduced during merging to the corrupted keys, showing that when a corrupted key is added to the output sequence it can be charged at most $(2\delta+1)$ errors. We analyze B-bursts only (the analysis of A-bursts is symmetric). Consider the end of a final B-burst, i.e., the time when key $A[i]$ is added to the output sequence. Notice that $A[i]$ may have prevented some elements of sequence A from being placed correctly in the output sequence only if $A[i]$ is corrupted and its value is larger than the correct one. If this is the case, let $\beta \geq 0$ be the number of consecutive keys of B that have been returned before $A[i]$. We distinguish the following two cases:

► $\beta > 2\delta + 1$: in this case, since bursts have length $\leq (2\delta + 1)$, the previous burst must have been a (non-final) B-burst of length $(2\delta+1)$, at the end of which the algorithm must have checked for possible errors. Let $W = A[i + 1; \min\{i + 2\delta + 1, n\}]$ be the window considered by the algorithm at that time.

- If W contains less than $(2\delta+1)$ values (i.e., if $i+2\delta+1 > n$), than $A[i]$ in the worst case may have prevented all those keys from A from being output at the right time: we can charge those errors to the corrupted value $A[i]$.
- If W contains $(2\delta+1)$ values (i.e., if $i+2\delta+1 \leq n$), the algorithm must have counted the number t of values in W that were smaller than $A[i]$. Since $A[i]$ was not returned at that time, it must have been $t \leq \delta$. Thus, there must be at least one element of W, say x, such that $A[i] \leq x$ and x is correct. Since $A[i] \leq x$, by transitivity the keys of B that have been output during all the $\lceil \beta/(2\delta + 1) \rceil$ consecutive B-bursts are smaller than x, and thus they will appear in the output sequence in the correct position with respect to x and to all the keys of sequence A that are larger than or equal to x. Hence, $A[i]$ may have prevented at most δ keys of sequence A (those smaller than $A[i]$) from being output at the right time, and once again we can charge those possible errors to $A[i]$ itself.

► $\beta \leq 2\delta+1$: in this case it can be either (i) $wait_A = 2\delta+1$, $i+2\delta+1 \leq n$ and $t \geq \delta+1$, or (ii) $wait_A < 2\delta + 1$ and $A[i] < B[j]$. In the first case at least one of the $t \geq \delta + 1$ keys of W that are smaller than $A[i]$ must be correct; the algorithm can deduce that $A[i]$

is corrupted and return it immediately, in spite of the fact that it may be in a wrong position. In both cases, the β keys of B that have been returned during the B-burst may appear in the wrong position in the output sequence: those errors, which are at most $(2\delta + 1)$, can be charged to the corrupted value $A[i]$.

This shows that each corrupted key can be charged at most $(2\delta + 1)$ errors. Since correct keys are never charged, this concludes the proof. □

3.2 Merging in superlinear time without error propagation

Let A and B be two 0-unordered sequences of length n_1 and n_2, respectively, containing α corrupted values, with $\alpha \leq \delta$. Without loss of generality assume that $n_2 \leq n_1$. Our strongly resilient merging, called `Unbalanced-SR-Merge`, repeatedly extracts a key from the shorter sequence B and places it in the correct position with respect to the longer sequence A.

Algorithm Unbalanced-SR-Merge. Let i and j be the running indices on A and B, respectively. At each step, we extract the minimum in $B[j; \min\{j + \delta, n_2\}]$: let $b = B[h]$ be such minimum, for $j \leq h \leq \min\{j+\delta, n_2\}$. We shift right all the keys in $B[j; h-1]$, move b to $B[j]$, and increase index j by 1. Then, we scan A sequentially, starting from position i, and add the keys of A to the output sequence until we find a key such that $A[i] > b$. Since $A[i]$ may be corrupted, returning b just before $A[i]$ may be wrong: we therefore count the number t of keys smaller than $A[i]$ in the window $W = A[i + 1; \min\{i + 2\delta + 1, n_1\}]$. If $t \geq \delta + 1$, we continue scanning A. Otherwise, we partition the window W into two groups: the keys $\leq b$ and the keys $> b$. We rearrange W so that keys $\leq b$ appear before keys $> b$, while the relative order of any two keys in the same group is maintained. We add to the output sequence the keys of W that are $\leq b$, followed by b itself, and start a new step.

Lemma 4 *Let A and B be two 0-unordered sequences of length n_1 and n_2, respectively, with $n_2 \leq n_1$. Algorithm* `Unbalanced-SR-Merge` *merges the sequences in $O(n_1 + (n_2 + \alpha) \cdot \delta)$ time, where $\alpha \leq \delta$ is the number of corrupted keys at the end of the algorithm execution. The output of* `Unbalanced-SR-Merge` *is a 0-unordered sequence.*

Proof. We first prove that the output sequence is 0-unordered. In spite of possible faults, the correct keys of B are extracted in increasing order, because the element b extracted at each step is certainly smaller than or equal to the minimum correct key in $B[j; n_2]$. The choice of the proper position of b with respect to sequence A is also resilient. Indeed, when a key $A[i] > b$ is encountered by the algorithm, a check on the window $W = A[i + 1; \min\{i + 2\delta + 1, n_1\}]$ is performed before adding b to the output sequence. If W contains $t \geq \delta + 1$ keys smaller than $A[i]$, then $A[i]$ is certainly corrupted and the algorithm correctly continues scanning A. Otherwise, if $t \leq \delta$, we consider two cases, depending on whether $i + 2\delta + 1 \leq n_1$ or not.

- $i + 2\delta + 1 \leq n_1$: in this case, W has size $(2\delta + 1)$. Since $t \leq \delta$, at least $(\delta + 1)$ elements from W are larger than or equal to $A[i]$, and thus at least one of them must be correct. Keys of W smaller than b are returned before b itself, and the fact that their relative order is maintained guarantees that no error is introduced.

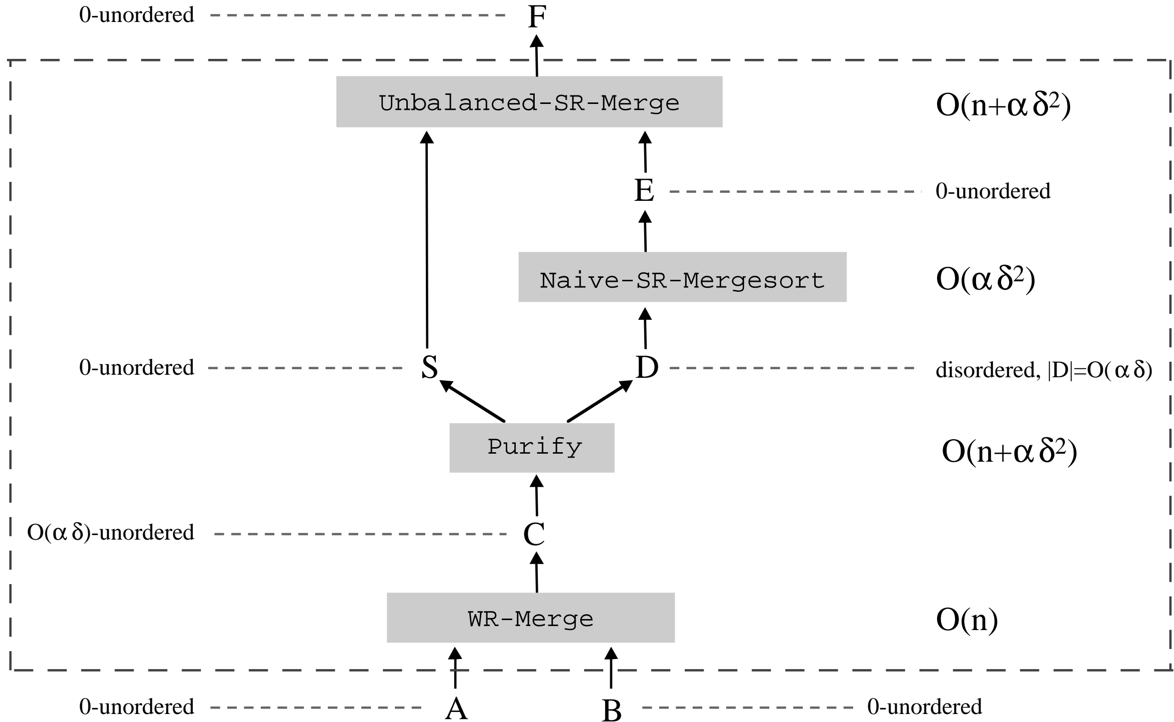

Figure 2: The strongly resilient merging algorithm `SR-Merge`.

- $i + 2\delta + 1 > n_1$: in this case, all the keys of A not yet added to the output sequence are in W, and only those larger than b are correctly not output before b itself.

We now discuss the running time. Extracting b, analyzing and rearranging W, can all be implemented in time $O(\delta)$. Note that if $A[i]$ is corrupted, we may continue scanning A, and therefore more than one window may be analyzed before adding b to the output sequence: we charge the analysis of all these windows to the corrupted element $A[i]$ associated with each of them. Since at most α elements are faulty, this contributes $O(\alpha \cdot \delta)$ to the total running time. All the remaining operations require time $O(n_1 + n_2 \cdot \delta)$. □

3.3 The sorting algorithm

In this section we present and analyze a strongly resilient mergesort-based sorting algorithm, that we call `SR-Mergesort`. As in Section 2, we assume an iterative bottom-up implementation in order to avoid problems due to a non fault-tolerant recursion stack.

The merging algorithm `SR-Merge` that will be used by `SR-Mergesort` is described in Figure 2. The input sequences, A and B, are first merged using the linear-time subroutine `WR-Merge`. The output sequence, C, may not be 0-unordered, i.e., some correct elements may be in a wrong position. Such errors are recovered by the combined use of algorithms `Purify` and `Unbalanced-SR-Merge`. Namely, the disordered list D of keys discarded by `Purify` is first sorted using algorithm `Naive-SR-Mergesort` described in Section 2. The slower strongly resilient subroutine `Unbalanced-SR-Merge` is then used on two 0-unorderded unbalanced sequences, S and E, the shorter of which has length proportional to the actual

algorithm	error propagation	overhead
`Standard-Merge`	$\Theta(n)$	0
`WR-Merge`	$O(\delta)$	0
`Naive-SR-Merge`	0	$O(\delta \cdot n)$
`Unbalanced-SR-Merge`	0	$O((n_2 + \alpha) \cdot \delta)$
`SR-Merge`	0	$O(\alpha \cdot \delta^2)$

Table 3: Quality of the solution and overhead of different merging algorithms: δ, α, n, and n_2 denote the upper bound on the number of memory faults, the actual number of faults that occur throughout a specific execution, the total number of input keys, and the number of keys in the shorter input sequence, respectively.

number of corrupted keys (as we will see later). It is easy to see that the entire algorithm is strongly resilient.

Analysis. Let ℓ be the total length of the 0-unordered sequences A and B to be merged. Let in and $thru$ denote the number of keys corrupted in the input and during merging, respectively, and let $\alpha = in + thru$. The running time of algorithm `SR-Merge` can be stated in terms of ℓ, α and δ as follows:

Lemma 5 *Algorithm* `SR-Merge` *takes time* $O(\ell + \alpha \cdot \delta^2)$ *to merge two sequences of total length* ℓ*, where* $\alpha \leq \delta$ *is the number of corrupted keys at the end of the algorithm execution.*

Proof. By Lemma 3, `WR-Merge` returns an $O(\alpha \cdot \delta)$-unordered sequence in time $O(\ell)$. By Lemma 1, the list D of keys discarded by `Purify` has length $O(\alpha \cdot \delta)$. The running times of `Purify` and `Unbalanced-SR-Merge` are both $O(\ell + \alpha \cdot \delta^2)$ by Lemmas 1 and 4, respectively. Since $|D| = O(\alpha \cdot \delta)$, then $\delta = \Omega(|D|^{1/2})$ and `Naive-SR-Mergesort` on the set D requires time $O(|D| \cdot \delta) = O(\alpha \cdot \delta^2)$ (see Lemma 2). The total running time immediately follows. □

The propagation of errors of algorithm `SR-Merge` satisfies the following property:

Lemma 6 *Algorithm* `SR-Merge` *returns a* 0*-unordered sequence in which only the keys corrupted while merging may be unordered.*

Proof. Consider the 0-unordered sequence S returned by `Purify`: only keys corrupted during the execution of `Purify` may be unordered in S (see Lemma 1). A similar consideration holds for the sequence E returned by `Naive-SR-Mergesort`. Thus, any key corrupted before the execution of algorithm `SR-Merge` will appear in the proper position in the sequences S and E received as input by algorithm `Unbalanced-SR-Merge`: these corrupted, but correctly ordered keys will be treated by `Unbalanced-SR-Merge` as if they were correct. This proves that only the keys corrupted during the execution of algorithm `SR-Merge` may be unordered in the output sequence F. □

Table 3 compares the quality of the solution and the running time overhead of algorithm `SR-Merge` with those of the merging algorithms described before. We are now ready to analyze algorithm `SR-Mergesort`, that uses `SR-Merge` as a subroutine.

Theorem 1 *Algorithm* `SR-Mergesort` *is strongly resilient and sorts* n*-length sequences in* $O(n \log n + \alpha \cdot \delta^2)$ *worst-case time, where* $\alpha \leq \delta$ *is the total number of memory faults.*

algorithm	error propagation	overhead
`Standard-Mergesort`	$\Theta(n)$	0
`Naive-SR-Mergesort`	0	$O(\delta \cdot n \log n)$
`SR-Mergesort`	0	$O(n \log n + \alpha \cdot \delta^2)$

Table 4: Quality of the solution and overhead of algorithm `SR-Mergesort`, compared with the basic sorting algorithms described in Section 2.

Proof. Without loss of generality assume that n is a power of 2. Consider the merge tree associated with algorithm `SR-Mergesort`, whose nodes correspond to the invocations of procedure `SR-Merge`: namely, the root of the merge tree corresponds to the last merge step performed during the execution of `SR-Mergesort`, and the left and right subtrees are defined recursively on the left and right half of the input sequence. Let x be any node of this tree at level k, for $0 \le k < \log_2 n$. Let $\mathrm{in}_k(x)$ be the number of errors contained in the sequences to be merged at node x, and let $\mathrm{thru}_k(x)$ be the number of errors introduced while merging at node x. By Lemma 5, the time required for merging at node x satisfies the following:

$$T_k(x) = O\left(\frac{n}{2^k}\right) + (\mathrm{in}_k(x) + \mathrm{thru}_k(x))\,\delta^2$$

By Lemma 6,

$$\mathrm{in}_k(x) = \mathrm{thru}_{k+1}(x_1) + \mathrm{thru}_{k+1}(x_2)$$

where x_1 and x_2 are the children of node x in the merge tree. Since α is the total number of memory faults, we have

$$\sum_{k=0}^{\log n} \sum_{x=1}^{2^k} \mathrm{thru}_k(x) \le \alpha$$

Thus, if we sum up the fault dependent contributions $(\mathrm{in}_k(x) + \mathrm{thru}_k(x))\delta^2$ on the entire merge tree, we obtain $O(\alpha \cdot \delta^2)$. With standard techniques we can then conclude that the running time of algorithm `SR-Mergesort` is $O(n \log n + \alpha \cdot \delta^2)$. □

A crucial point in the running time analysis is that the slowdown of the merge step depends only on the actual number of faults in the sequence: the fewer the corrupted values, the faster is merging and thus sorting. We remark that algorithm `SR-Mergesort` can actually work without knowledge of a tight upper bound on the number of faults: choosing $\delta = \Theta((n \log n)^{1/3})$ yields an $O(n \log n)$ strongly resilient sorting algorithm that can tolerate $O((n \log n)^{1/3})$ memory faults. Moreover, by Lemma 6, if no key is corrupted during the last merge step (i.e., at the root of the merge tree), both correct and corrupted keys will be ordered in the output sequence.

Table 4 summarizes the quality of the solution and the running time overhead of algorithm `SR-Mergesort` and compares it with the basic sorting algorithms described in Section 2.

4 Lower bound on resilient sorting

In this section we prove lower bounds on strongly resilient merging and sorting in the comparison model. We will assume that the basic comparisons are of the form "$x < y$?": the results can be easily generalized to other kinds of comparisons, such as "$x \le y$?" or "$x > y$?". We

first show that $\Omega(n+\delta^{2-\epsilon})$ comparisons are necessary to merge two 0-unordered sequences of length n containing up to δ faulty values when $\delta \leq n^{2/(3-2\epsilon)}$, for any ϵ, $0 \leq \epsilon \leq 1/2$. To prove this claim, we use an adversary-based argument. A merging algorithm asks comparison questions that the adversary should answer consistently: if challenged at any time, the adversary must exhibit two input sequences and at most δ memory faults such that all her answers are correct (i.e., consistent with the memory image at the time when the comparison was asked). Both the algorithm and the adversary know the fault upper bound δ.

Adversary's strategy. Let A and B be two 0-unordered sequences of length n that need to be merged. Without loss of generality we assume that $n \bmod \delta = 0$. We divide A and B into δ subsequences of n/δ consecutive elements, called $A_1, \ldots A_\delta$ and $B_1, \ldots B_\delta$, respectively. Namely, $A_i = A[1+(i-1)n/\delta;\, i(n/\delta)]$, for $1 \leq i \leq \delta$. B_i is defined similarly. The adversary answers the algorithm's questions as if the sorted sequence were:

$$S = A_1\, B_1\, A_2\, B_2 \ldots B_{\delta-1}\, A_\delta\, B_\delta \tag{1}$$

Analysis. We build a comparison graph $G(V,E)$ as follows: V consists of $2n$ vertices, one for each element of A and B; two vertices are connected by as many edges as the number of times the corresponding elements have been compared. The vertices are grouped into 2δ clusters associated with the subsequences A_i and B_i defined above. To prove the lower bound, we will show that if this comparison graph has less than $\lfloor \delta^{2-\epsilon}/4 - \delta/2 \rfloor$ edges, then there must be at least two possible orderings of the input sequences that are both consistent with all of the adversary's answers.

We say that two clusters A_i and B_j are *consecutive* if either $j=i$ or $j=i-1$. A sequence of consecutive clusters is called *sparse* if the elements in the clusters induce a subgraph with less than $\lceil \delta/2 \rceil$ edges. The *length* of a sequence of clusters is defined as the number of clusters in the sequence. We first prove two lemmas about sparse sequences.

Lemma 7 *If the algorithm performs less than $\lfloor \delta^{2-\epsilon}/4 - \delta/2 \rfloor$ comparisons, for some ϵ, $0 \leq \epsilon \leq 1/2$, then there must be at least one sparse sequence of length $2\lceil \delta^\epsilon \rceil$.*

Proof. Let S be defined as in (1). Let ℓ be the maximum number of disjoint subsequences of length $2\lceil \delta^\epsilon \rceil$ that can be obtained by dividing S into sets of $2\lceil \delta^\epsilon \rceil$ consecutive clusters. Then $\ell = \lfloor 2\delta/(2\lceil \delta^\epsilon \rceil) \rfloor \geq \delta/(\delta^\epsilon+1) - 1 \geq \delta^{1-\epsilon}/2 - 1$. If none of the subsequences is sparse, the number of comparisons should be at least $\ell \cdot \lceil \delta/2 \rceil \geq (\delta^{1-\epsilon}/2 - 1)(\delta/2) = \delta^{2-\epsilon}/4 - \delta/2$, which is a contradiction. □

Lemma 8 *Let $\delta \leq n^{2/(3-2\epsilon)}$, for some ϵ, $0 \leq \epsilon \leq 1/2$. Let Y be a sparse sequence of length $2\lceil \delta^\epsilon \rceil$. Then Y must contain two clusters A_i and B_j such that at least one pair of elements $a \in A_i$ and $b \in B_j$ has not been compared by the algorithm.*

Proof. Assume by contradiction that each element of A_i is compared against each element of B_j, for each A_i and B_j in Y. Recall that $|A_i| = |B_j| = n/\delta$ for each i and j, and that Y consists of a total of $2\lceil \delta^\epsilon \rceil$ clusters, such that $\lceil \delta^\epsilon \rceil$ clusters are in B and $\lceil \delta^\epsilon \rceil$ clusters are in A. Thus, each cluster A_i must have at least $\delta^\epsilon (n/\delta)^2$ incident edges, and the total number of edges in the subgraph induced by Y must be at least $n^2\delta^{2\epsilon-2}$. Since by assumption $\delta \leq n^{2/(3-2\epsilon)}$, we have $n^2\delta^{2\epsilon-2} \geq \delta^{3-2\epsilon}\delta^{2\epsilon-2} = \delta$. Since Y is sparse, it should be also $n^2\delta^{2\epsilon-2} < \lceil \delta/2 \rceil$, which is clearly a contradiction. □

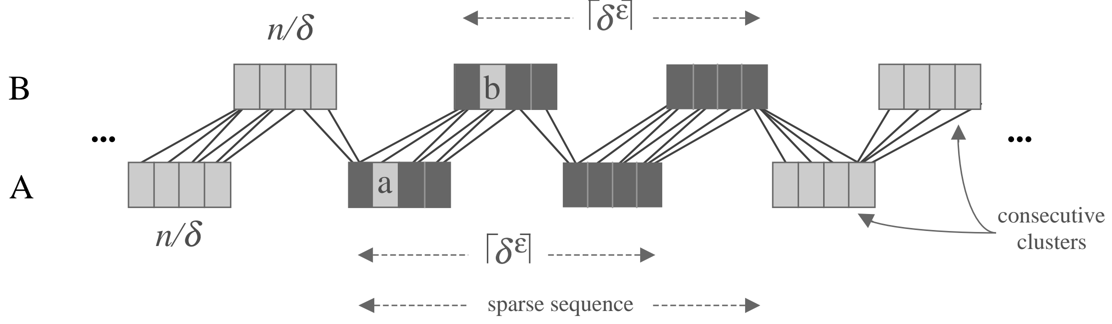

Figure 3: Sparse sequence and elements a and b used in the proof of Lemma 9: dark grey elements are corrupted, light grey elements in the sparse sequence have not been compared against each other.

Lemma 9 *Given any ϵ, $0 \leq \epsilon \leq 1/2$, any strongly resilient merging algorithm requires in the worst-case $\Omega(n + \delta^{2-\epsilon})$ comparisons to merge two 0-unordered sequences of length n when up to $\delta \leq n^{2/(3-2\epsilon)}$ values may be corrupted.*

Proof. We consider only the case where $\delta > n^{1/(2-\epsilon)}$, since the lower bound trivially holds for $\delta \leq n^{1/(2-\epsilon)}$. If $\delta > n^{1/(2-\epsilon)}$, we will show that the merging algorithm needs $\Omega(\delta^{2-\epsilon})$ comparisons in order to determine the correct order of uncorrupted keys.

To prove this, assume that the algorithm performs less than $\lfloor \delta^{2-\epsilon}/4 - \delta/2 \rfloor$ comparisons. Then, by Lemma 7, there must be a sparse sequence of length $2\lceil \delta^{\epsilon} \rceil$, say $Y = A_k B_k A_{k+1} \ldots A_{k+\lceil \delta^{\epsilon} \rceil - 1} B_{k+\lceil \delta^{\epsilon} \rceil - 1}$. By Lemma 8, let $a \in A_i$ and $b \in B_j$, for some $k \leq i, j < k + \lceil \delta^{\epsilon} \rceil$, be two elements of Y that have not been directly compared by the algorithm during its execution. Without loss of generality, assume that $i \leq j$ (the analysis when $i > j$ is similar). We will now show that the algorithm cannot determine the correct order of a and b (even by transitivity). Namely, the adversary can exhibit at most δ memory faults such that both the order in which a precedes b and the order in which b precedes a are consistent with her answers. Consider the comparisons performed by the algorithm, and let "$x < y$?" be any such comparison question. The adversary claims that:

(1) If either $x \notin Y$ or $y \notin Y$, the question has been answered without introducing memory faults.

(2) If $x \in \{a, b\}$ and $y \in Y \setminus \{a, b\}$, then y has been corrupted.

(3) If $y \in \{a, b\}$ and $x \in Y \setminus \{a, b\}$, then x has been corrupted.

(4) If $x \in Y \setminus \{a, b\}$ and $y \in Y \setminus \{a, b\}$, then both x and y have been corrupted.

The comparisons "$a < b$?" and "$b < a$?" are not included in cases (1)–(4) since a and b have not been compared by the algorithm. If the comparison "$x < y$?" involves at most one element of Y (case (1)), there are no memory faults. Otherwise (cases (2)–(4)), at most two values are corrupted: since Y is sparse, there are less than $\lceil \delta/2 \rceil$ comparisons of type (2)–(4) and thus the total number of corrupted values is at most δ.

We now show that, in accordance with cases (1)–(4), either a can precede b or b can precede a in the final ordering. Let $Y^u \subseteq Y$ be the set of uncorrupted values in Y. Since a

and b have never been corrupted, $\{a,b\} \subseteq Y^u$. Note that the values in $Y^u \cap B$ have not been compared against the values in $Y^u \cap A$: otherwise, they would be considered as corrupted and would not belong to Y^u. Furthermore, the comparisons with elements outside Y cannot be used by transitivity to get information on the relative order of $Y^u \cap B$ and $Y^u \cap A$: since the adversary does not need to choose the values in advance, the sequences A and B can always be stretched so that, for any $t < k$ ($t > k + \lceil \delta^\epsilon \rceil$), values in A_t and B_t are smaller (larger) than values in Y. In the final ordering, the values in $Y^u \cap B$ can therefore appear either before or after the values in $Y^u \cap A$, and the algorithm cannot decide which of the two orders ($a < b$ or $b < a$) is correct.

Since neither a nor b is faulty, the algorithm must put them in the correct relative order to get a 0-unordered sequence. We can therefore conclude that any strongly resilient merging algorithm requires at least $\lfloor \delta^{2-\epsilon}/4 - \delta/2 \rfloor = \Omega(\delta^{2-\epsilon})$ comparisons, for any ϵ, $0 \leq \epsilon \leq 1/2$. □

Theorem 2 *Given any ϵ, $0 \leq \epsilon \leq 1/2$, any strongly resilient sorting algorithm requires in the worst-case $\Omega(n \log n + \delta^{2-\epsilon})$ comparisons to sort a sequence of length n when up to $\delta \leq n^{2/(3-2\epsilon)}$ values may be corrupted.*

Proof. If $\delta \leq (n \log n)^{1/(2-\epsilon)}$, the lower bound trivially holds. Otherwise, if $(n \log n)^{1/(2-\epsilon)} < \delta \leq n^{2/(3-2\epsilon)}$, the lower bound on sorting follows from Lemma 9. Indeed, if Theorem 2 were not true, using a sorting algorithm in order to solve the merging problem would contradict the lower bound of $\Omega(\delta^{2-\epsilon})$ that derives from Lemma 9. □

We can now characterize the resilience to memory faults of comparison-based strongly resilient sorting algorithms with optimal running time.

Theorem 3 *Any $O(n \log n)$ comparison-based strongly resilient sorting algorithm can tolerate the corruption of at most $O((n \log n)^{1/2})$ values.*

Proof. By Theorem 2, $\Omega(n \log n + \delta^{2-\epsilon})$ is a lower bound on strongly resilient sorting when the number δ of corrupted values is such that $\delta \leq n^{2/(3-2\epsilon)}$, for any ϵ, $0 \leq \epsilon \leq 1/2$. We consider three different choices for ϵ:

- $\epsilon = 1/2$: this gives the weakest lower bound, i.e., $\Omega(n \log n + \delta^{3/2})$, that holds for each $\delta \leq n$. An $O(n \log n)$ algorithm cannot therefore tolerate the corruption of γ values, with $\gamma \in \big((n \log n)^{2/3}, n\big]$.
- $\epsilon = 1/6$: this gives a lower bound of $\Omega(n \log n + \delta^{11/6})$ that holds for each $\delta \leq n^{3/4}$. An $O(n \log n)$ algorithm cannot therefore tolerate the corruption of γ values, with $\gamma \in \big((n \log n)^{6/11}, n^{3/4}\big]$.
- $\epsilon = 0$: this gives the strongest lower bound, i.e., $\Omega(n \log n + \delta^2)$, that holds for each $\delta \leq n^{2/3}$. An $O(n \log n)$ algorithm cannot therefore tolerate the corruption of γ values, with $\gamma \in \big((n \log n)^{1/2}, n^{2/3}\big]$.

As shown in Figure 4:

$$\Big((n \log n)^{1/2}, n^{2/3}\Big] \cup \Big((n \log n)^{6/11}, n^{3/4}\Big] \cup \Big((n \log n)^{2/3}, n\Big] = \Big((n \log n)^{1/2}, n\Big]$$

Any optimal comparison-based sorting algorithm can thus tolerate the corruption of at most $O((n \log n)^{1/2})$ values. □

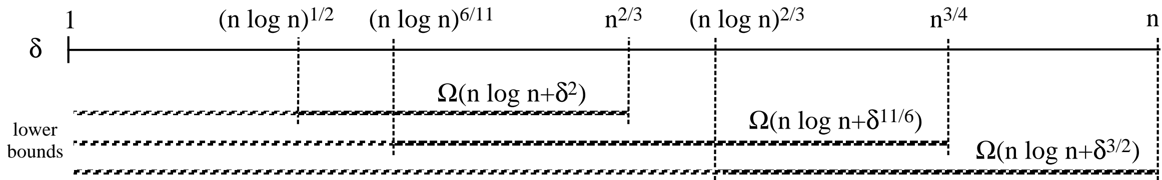

Figure 4: Intervals used in the proof of Theorem 3.

5 Resilient binary search

In this section we consider the resilient binary search problem. Let X be an ordered sequence of length n containing at most δ corrupted keys, which may be possibly out of order: the occurrence of such memory faults may compromise the correctness of the classical binary search algorithm, since it may be possible that a faulty key in position x is smaller than some correct keys in positions $< x$, thus guiding the search towards a wrong direction. Let s be a key to be searched in X. We expect from a resilient searching algorithm to answer yes if there is a correct key equal to s, and no if there is no key (either correct or faulty) equal to s. Note that, if there are only faulty keys equal to s, the answer may be either yes or no.

A naive resilient searching algorithm can be easily implemented by using a majority argument at each search step, i.e., by following the search direction suggested by the majority of $(2\delta + 1)$ keys in consecutive array positions. However, such an algorithm would have a large overhead of $O(\delta \log n)$ on the running time. In Section 5.1 we describe an algorithm that requires $O(\log n + \alpha \cdot \delta)$ time, where $\alpha \leq \delta$ is the actual number of faulty keys in X. This yields a resilient searching algorithm that can tolerate $O((\log n)^{1/2})$ memory faults in $O(\log n)$ time in the worst case. We also prove in Section 5.2 that any comparison-based searching algorithm that runs in $O(\log n)$ time can tolerate the corruption of at most $O(\log n)$ values.

5.1 The search algorithm

Our binary searching algorithm, called `ResilientSearch`, proceeds in a non-resilient way, checking from time to time if it did some mistake. If this is the case, it corrects the search direction using a resilient recovery procedure. Although we present a tail-recursive implementation of algorithm `ResilientSearch`, the recursion can be easily unrolled by storing just a few array indices in $O(1)$ words of reliable memory.

Let a and b denote the indices to the left and right boundaries of the array to be searched for at a generic step. The recursive call `ResilientSearch`(X, a, b) works as follows.

Step 1: interval retrieval. If $a = b$, return yes if and only if $s = X[a]$ and stop. Otherwise, starting from $X[(a + b)/2]$, perform h steps of *non-resilient* binary search (the value of h will be determined later). Let ℓ and r be the indices to the left and right boundaries of the interval I identified by this search.

Step 2: neighborhood search. Let $\mathcal{N}_\ell$ and $\mathcal{N}_r$ be two $(2\delta + 1)$-size neighborhoods of $X[\ell]$ and $X[r]$, respectively: i.e., $\mathcal{N}_\ell = X[\max\{0, \ell - 2\delta\}; \ell]$ and $\mathcal{N}_r = X[r; \min\{r + 2\delta, n - 1\}]$. Check by exhaustive search if $s \in \mathcal{N}_\ell \cup \mathcal{N}_r$: if this is the case, return yes and stop.

Step 3: error recovery. If $\mathcal{N}_\ell$ contains at least $(\delta+1)$ keys larger than s, set $r = \max\{a, \ell - 2\delta\}$ and then $\ell = a$. Otherwise, if $\mathcal{N}_r$ contains at least $(\delta + 1)$ keys smaller than s, set

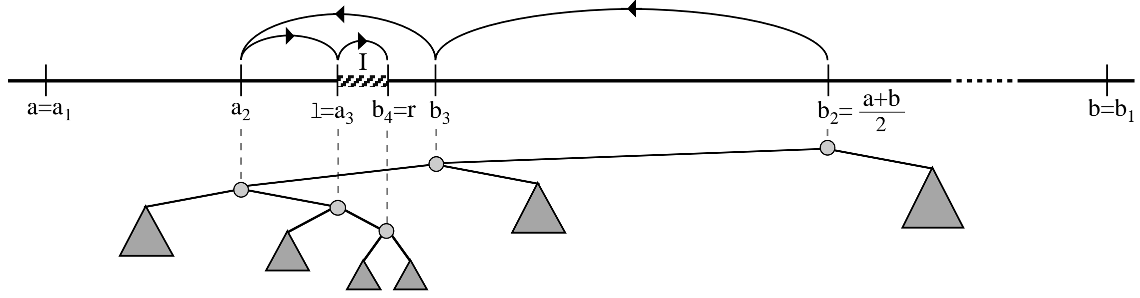

Figure 5: Resilient binary search with $h = 5$ and correspondence with a binary search tree: a_i's and b_i's are the left and right boundary nodes, respectively.

$\ell = \min\{r + 2\delta, b\}$ and then $r = b$.

Step 4: recursive call. Return the result of `ResilientSearch`(X, ℓ, r).

Note that, if there were no faults, at the end of Step 1 the search would continue in interval $I = X[\ell\,;r]$. On the other side, if we took a wrong search direction at some point during Step 1, then the search should continue either in $X[a\,;\ell-1]$ or in $X[r+1\,;b]$. This is detected by the error recovery procedure in Step 3, and implies updating the left and right boundaries before performing the recursive call of Step 4.

Analysis. The keys considered during the non-resilient search of Step 1 correspond to the nodes of a binary search tree, as shown in Figure 5. Call $a_1, \ldots, a_x$ and $b_1, \ldots, b_y$ the nodes where we turn right and left during the search, respectively (also called *left boundary* and *right boundary* nodes). Clearly, $a_1 = X[a]$, $a_x = X[\ell]$, $b_y = X[r]$, and $b_1 = X[b]$. In the following we call a value *misleading* if it is faulty and guides the search towards a wrong direction during Step 1. The following theorem analyzes the correctness and the running time of algorithm `ResilientSearch`.

Theorem 4 *Algorithm* `ResilientSearch` *correctly answers membership queries on a n-length sequence X in $O(\log n + \alpha \cdot \delta)$ worst-case time, where $\alpha \le \delta$ is the number of faulty keys in X at the end of the algorithm execution.*

Proof. The algorithm answers yes only if it finds an index i such that $X[i]$ equals the searched key s. Thus, to show its correctness, we only need to prove that in case of a negative answer there is no correct key equal to s. Assume that the algorithm has taken, at some point during the execution of Step 1, a wrong search direction. In this case, however, one of the tests in Step 3 succeeds, and the left and right boundaries are updated before the recursive call of Step 4. We now prove that, if X contains a correct key equal to s, such a key must be searched in $X[\ell; r]$, where ℓ and r are updated as in Step 3.

First, consider the case where $\mathcal{N}_\ell$ contains at least $(\delta + 1)$ keys larger than s: then, there is at least a key $\lambda \in \mathcal{N}_\ell$ such that $\lambda > s$ and λ is correct. The search of s should then continue in $X[a; p]$, with $p \in [\ell - 2\delta, \ell]$ and $X[p] = \lambda$. The exhaustive search performed in Step 2 guarantees that $s \notin \mathcal{N}_\ell$, and thus proceeding in $X[a; \ell - 2\delta]$ is correct. A symmetric argument holds if the test on $\mathcal{N}_r$ succeeds. This guarantees that the values ℓ and r are always properly set before the recursive call of Step 4.

We now discuss the running time. We first bound the time required for uncorrect searches. Assume that, during the execution of Step 1, the algorithm encounters a misleading left boundary node, say a_j (the case of misleading right boundary nodes is symmetric). Then all a_k with $k \geq j$ must be misleading, and in particular $a_x = X[\ell]$ is misleading, and thus faulty. We charge the time $O(h)$ spent for this uncorrect search to $X[\ell]$. Since $X[\ell]$ is out of the interval $X[a; \max\{a, \ell - 2\delta\}]$ in which the search proceeds further, each faulty key can be charged at most once and we will have at most α uncorrect searches, whose total running time is $O(\alpha \cdot h)$.

We now analyze the running time for the other steps. It is easy to see that Steps 1, 2 and 3 require time $O(\delta + h)$. If the search in Step 1 is correct, the interval on which we recurse in Step 4 has length $(b - a)/2^h$, and thus the running time for the remaining steps is given by the recurrence

$$T(n) = T\left(\frac{n}{2^h}\right) + O(\delta + h) = O\left(\left(1 + \frac{\delta}{h}\right) \log n\right) \qquad (2)$$

Choosing $h = \Theta(\delta)$ and summing up the running time given by (2) with the contribution $O(\alpha \cdot \delta)$ of the uncorrect searches, we obtain a total running time of $O(\log n + \alpha \cdot \delta)$ in the worst case for each membership query.

□

Theorem 4 implies that algorithm `ResilientSearch` can tolerate up to $O((\log n)^{1/2})$ memory faults while still running in $O(\log n)$ optimal time.

5.2 The lower bound

In this section we prove a lower bound on resilient searching in the comparison model. In particular, we use an adversary-based argument to show that $\Omega(\log n + \delta)$ comparisons are necessary to answer membership queries on an ordered sequence X of length n containing up to δ faulty keys. We assume that the basic comparisons are of the form "$x < y$?", where x and y either are keys of X or coincide with the key s that we are searching for. The lower bound holds also if the adversary does not know the fault upper bound δ. The adversary strategy is as follows:

> Let X be an ordered sequence of length n. Let x_i, for $1 \leq i \leq n$, denote the i-th element of X, and let s be the key to be searched for. The algorithm may ask two possible kinds of comparisons: either "$s < x_i$?" or "$x_i < x_j$?", for $1 \leq i, j \leq n$. If the question is of the former type, the adversary always answers yes. If the question is of the latter type, the adversary answers yes if and only if $i < j$.

We will now prove the following theorem:

Theorem 5 *Any comparison-based searching algorithm with optimal $O(\log n)$ search time can tolerate the corruption of at most $O(\log n)$ values.*

Proof. We start by proving a lower bound of $\Omega(\log n + \delta)$ on resilient searching. It suffices to prove that every resilient searching algorithm needs $\Omega(\delta)$ comparisons: combining this with the classical $\Omega(\log n)$ lower bound (absence of faults), yields the desired result.

Assume that the searching algorithm asks less than $(\delta/2)$ comparisons and that the adversary answers according to the strategy described above. Let $\mathcal{C} \subseteq X$ be the set of elements

involved in at least one comparison. Note that $|\mathcal{C}| < 2(\delta/2) = \delta$, because at most two elements per comparison are used. Since $\delta \leq n$, the subsequence $X \setminus \mathcal{C}$ is not empty, and the comparisons provide no information about this subsequence. Based on this, the algorithm cannot decide whether the answer to a membership query for a key s should be yes or no. Indeed, the adversary can exhibit either a correct sequence not containing s or a faulty sequence containing s. This faulty sequence can be obtained as follows. Let x_k be any element in $X \setminus \mathcal{C}$: the adversary claims that $x_k = s$, all the keys in $\mathcal{C}$ are corrupted, and all of them are larger than the keys in $X \setminus \mathcal{C}$. In both cases, all of the adversary's answers are consistent with the sequence X.

This lower bound implies that no resilient searching algorithm with optimal running time can tolerate the corruption of more than $O(\log n)$ values. □

6 Conclusions and open problems

In this paper we have initiated the investigation of resilient sorting and searching algorithms, i.e., algorithms that are able to produce a correct output despite the appearance of memory faults during their execution. By characterizing the running time as a function of the maximum number of memory faults that may occur throughout the algorithm execution, we have obtained several upper and lower bounds. We have proved that any strongly resilient comparison-based sorting algorithm running in $O(n \log n)$ worst-case time can tolerate the corruption of at most $O((n \log n)^{1/2})$ values. We have also presented an algorithm with optimal running time that tolerates up to $O((n \log n)^{1/3})$ memory faults. Similarly, we have designed an $O(\log n)$ time searching algorithm that can tolerate the presence of up to $O((\log n)^{1/2})$ corrupted values, and we have proved that any resilient searching algorithm with optimal running time can tolerate at most $O(\log n)$ memory faults.

After this work, in [16] we have closed the gaps between our upper and lower bounds for sorting designing a comparison-based resilient sorting algorithm that takes $O(n \log n)$ worst-case time and can tolerate up to $O((n \log n)^{1/2})$ faults. A thorough experimental study [15] has shown that our algorithms are not only theoretically efficient, but also fast in practice. With respect to resilient searching, in [16] we have proved matching upper and lower bounds $\Theta(\log n + \delta)$ for randomized algorithms, and we have presented an almost optimal deterministic algorithm that can tolerate up to $O((\log n)^{1-\epsilon})$ faults, for any small positive constant ϵ, in $O(\log n)$ worst-case time, thus getting arbitrarily close to the lower bound presented in this paper. We have also addressed the searching problem in a dynamic setting, designing a resilient version of binary search trees such that search operations, insertions of new keys, and deletions of existing keys can be implemented in $O(\log n + \delta^2)$ amortized time per operation [17].

As possible directions for future research, it would be interesting to improve the running time of the search tree operations and to design other resilient data structures (e.g., priority queues): we remark that classical data structures are highly non-resilient to memory faults, due to either the heavy use of pointers (e.g., balanced binary search trees) or the dependence upon structural and positional information (e.g., implicit heap-based priority queues). Other interesting questions are whether a probabilistic fault model would make it possible to tolerate an even larger number of memory faults than the more restrictive adversarial model considered in this paper, and whether it would be possible to obtain resilient algorithms that do not

assume any knowledge on the maximum number δ of memory faults.

Acknowledgments. We wish to thank Michael Bender, Mark de Berg, Gianfranco Bilardi, Piotr Indyk, Muthu Muthukrishnan, and Mikkel Thorup for enlightening discussions on this topic.

References

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. A. Aslam and A. Dhagat. Searching in the presence of linearly bounded errors. *Proc. 23rd ACM Symp. on Theory of Computing* (STOC'91), 486–493, 1991.

[3] S. Assaf and E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.

[4] Y. Aumann and M. A. Bender. Fault-tolerant data structures. *Proc. 37th IEEE Symp. on Foundations of Computer Science* (FOCS'96), 580–589, 1996.

[5] J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the Advanced Encryption Standard (AES). *Proc. 7th International Conference on Financial Cryptography* (FC'03), LNCS 2742, 162–181, 2003.

[6] R. S. Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. *Proc. 25th ACM Symp. on Theory of Computing* (STOC'93), 130–136, 1993.

[7] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 145–185, 1994.

[8] B. S. Chlebus, A. Gambin and P. Indyk. PRAM computations resilient to memory faults. *Proc. 2nd Annual European Symp. on Algorithms* (ESA'94), LNCS 855, 401–412, 1994.

[9] B. S. Chlebus, A. Gambin and P. Indyk. Shared-memory simulations on a faulty-memory DMM. *Proc. 23rd International Colloquium on Automata, Languages and Programming* (ICALP'96), 586–597, 1996.

[10] B. S. Chlebus, L. Gasieniec and A. Pelc. Deterministic computations on a PRAM with static processor and memory faults. *Fundamenta Informaticae*, 55(3-4), 285–306, 2003.

[11] C. R. Cook and D. J. Kim. Best sorting algorithms for nearly sorted lists. *Communications of the ACM*, 23, 620–624, 1980.

[12] A. Dhagat, P. Gacs, and P. Winkler. On playing "twenty questions" with a liar. *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms* (SODA'92), 16–22, 1992.

[13] M. Farach-Colton. Personal communication. January 2002.

[14] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23, 1001–1018, 1994.

[15] U. Ferraro Petrillo, I. Finocchi, G. F. Italiano. The Price of Resiliency: a Case Study on Sorting with Memory Faults. *Proc. 14th Annual European Symposium on Algorithms* (ESA'06), 768–779, 2006.

[16] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal sorting and searching in the presence of memory faults. *Proc. 33rd ICALP*, LNCS 4051, 286–298, 2006.

[17] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. To appear in *Proc. 18th ACM-SIAM SODA*, 2007.

[18] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). *Proc. 36th ACM Symposium on Theory of Computing* (STOC'04), 101–110, 2004.

[19] S. Hamdioui, Z. Al-Ars, J. Van de Goor, and M. Rodgers. Dynamic faults in Random-Access-Memories: Concept, faults models and tests. *Journal of Electronic Testing: Theory and Applications*, 19, 195–205, 2003.

[20] M. Henzinger. The past, present and future of Web Search Engines. Invited talk. *31st Int. Coll. Automata, Languages and Programming*, Turku, Finland, July 12–16 2004.

[21] P. Indyk. On word-level parallelism in fault-tolerant computing. *Proc. 13th Annual Symp. on Theoretical Aspects of Computer Science* (STACS'96), 193–204, 1996.

[22] D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winklmann. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20:396–404, 1980.

[23] K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Trans. on Computers*, 40(9):1081–1084, 1991.

[24] T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM Journal on Computing*, 29(1):258–273, 1999.

[25] T. Leighton, Y. Ma and C. G. Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54:265–304, 1997.

[26] T. C. May and M. H. Woods. Alpha-Particle-Induced Soft Errors In Dynamic Memories. *IEEE Trans. Elect. Dev.*, 26(2), 1979.

[27] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[28] S. Muthukrishnan. On optimal strategies for searching in the presence of errors. *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms* (SODA'94), 680–689, 1994.

[29] A. Pelc. Searching with known error probability. *Theoretical Computer Science*, 63, 185–202, 1989.

[30] A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoretical Computer Science*, 270, 71–109, 2002.

[31] P. J. Plauger, A. A. Stepanov, M. Lee, D. R. Musser. *The C++ Standard Template Library*, Prentice Hall, 2000.

[32] J. J. Quisquater and D. Samyde. Eddy current for magnetic analysis with active sensor. *Proc. International Conference on Research in SmartCards* (E-Smart'02), 185–194, 2002.

[33] B. Ravikumar. A fault-tolerant merge sorting algorithm. *Proc. 8th Annual Int. Conf. on Computing and Combinatorics* (COCOON'02), LNCS 2387, 440–447, 2002.

[34] A. Rényi. *A diary on information theory*, J. Wiley and Sons, 1994. Original publication: *Napló az információelméletröl*, Gondolat, Budapest, 1976.

[35] S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems* (CHES'02), LNCS 2523, 2–12, 2002.

[36] Tezzaron Semiconductor. *Soft errors in electronic memory - a white paper*, URL: `http://www.tezzaron.com/about/papers/Papers.htm`, January 2004.

[37] S. M. Ulam. *Adventures of a mathematician*. Scribners (New York), 1977.

[38] A.J. Van de Goor. *Testing semiconductor memories: Theory and practice*, ComTex Publishing, Gouda, The Netherlands, 1998.

[39] A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14, 120–128, 1985.