Optimal Resilient Sorting and Searching in the Presence of Memory Faults[*]

Irene Finocchi [†] Fabrizio Grandoni [‡] Giuseppe F. Italiano [‡]

Abstract

We investigate the problem of reliable computation in the presence of faults that may arbitrarily corrupt memory locations. In this framework, we consider the problems of sorting and searching in optimal time while tolerating the largest possible number of memory faults. In particular, we design an $O(n \log n)$ time sorting algorithm that can optimally tolerate up to $O(\sqrt{n \log n}\,)$ memory faults. In the special case of integer sorting, we present an algorithm with linear expected running time that can tolerate $O(\sqrt{n}\,)$ faults. We also present a randomized searching algorithm that can optimally tolerate up to $O(\log n)$ memory faults in $O(\log n)$ expected time, and an almost optimal deterministic searching algorithm that can tolerate $O((\log n)^{1-\epsilon})$ faults, for any small positive constant ϵ, in $O(\log n)$ worst-case time. All these results improve over previous bounds.

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

1 Introduction

The need for reliable computations in the presence of memory faults arises in many important applications. In fault-based cryptanalysis, for instance, some optical and electromagnetic perturbation attacks [6, 32] work by manipulating the non-volatile memories of cryptographic devices, so as to induce very timing-precise controlled faults on given individual bits: this forces the devices to output wrong ciphertexts that may allow the attacker to determine the secret keys used during the encryption. Induced memory errors have been effectively used in order to break cryptographic protocols [6, 7, 35], smart cards and other security processors [1, 2, 32], and to take control over a Java Virtual Machine [19].

Applications that make use of large memory capacities at low cost also incur problems of memory faults and reliable computation. Indeed, *soft memory errors* due to power failures,

[*]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 Projects WEB MINDS ("Wide scalE Broadband MIddleware for Networks Distributed Services") and ALGO-NEXT ("Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments"). A preliminary version of this work was presented at the *33rd International Colloquium on Automata, Languages and Programming* (ICALP'06).

[†]Dipartimento di Informatica, "Sapienza" Università di Roma, via Salaria 113, 00198, Roma, Italy. Email: `finocchi@di.uniroma1.it`.

[‡]Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy. Email: {`grandoni,italiano`}`@disp.uniroma2.it`.

radiation and cosmic rays tend to increase with memory size and speed [20, 27, 33], and can therefore have a harmful effect on memory reliability. Although the number of these faults could be reduced by means of error checking and correction circuitry, this imposes non-negligible costs in terms of performance (as much as 33%), size (20% larger areas), and money (10% to 20% more expensive chips). For these reasons, this is not typically implemented in low-cost memories. Data replication is a natural approach to protect against destructive memory faults. However, it can be very inefficient in highly dynamic contexts or when the objects to be managed are large and complex: copying such objects can indeed be very costly, and in some cases we might not even know how to do that (for instance, when the data is accessed through pointers, which are moved around in memory instead of the data itself, and the algorithm relies on user-defined access functions). In these cases, we cannot assume either the existence of *ad hoc* functions for data replication or the definition of suitable encoding mechanisms to maintain a reasonable storage cost. As an example, consider Web search engines, which need to store and process huge data sets (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 bit flips in the index, that may become responsible of erroneous answers to keyword searches [13, 21]. In all these scenarios, it makes sense to assume that it must be the algorithms themselves, rather than specific hardware/software fault detection and correction mechanisms, that are responsible for dealing with memory faults. Informally, we have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a failure, and 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 (at least) on the set of uncorrupted values.

The problem of computing with unreliable information has been investigated in a variety of different settings, including the *liar model* [3, 8, 14, 23, 24, 28, 29, 30, 31, 34], fault-tolerant sorting networks [4, 25, 26, 36], resiliency of pointer-based data structures [5], parallel models of computation with faulty memories [11, 12, 22]. In [18], we introduced a *faulty-memory random access machine*, i.e., a random access machine whose memory locations may suffer from memory faults. In this model, an adversary may corrupt up to δ memory words throughout the execution of an algorithm. The algorithm cannot distinguish corrupted values from correct ones and can exploit only $O(1)$ *safe* memory words, whose content never gets corrupted. Furthermore, whenever it reads some memory location, the read operation will temporarily store its value in the safe memory. The adversary is adaptive, but has no access to information about future random choices of the algorithm: in particular, loading a random memory location into safe memory can be considered an atomic operation. More details about the model are contained in [16].

In this paper we address the problems of resilient sorting and searching in the faulty-memory random access machine. In the *resilient sorting* problem we are given a sequence 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. The resilient sorting problem is to order correctly the set of uncorrupted keys. This is the best that we can achieve in the presence of memory faults, since we cannot prevent keys corrupted at the very end of the algorithm execution from occupying wrong positions in the output sequence. In the *resilient searching* problem we are given a sequence of n keys on which we wish to perform membership queries. The keys are stored in increasing order, but some keys may be corrupted (at any instant of time) and thus may occupy wrong positions in the sequence. Let x be the key to be searched for. The resilient searching problem is either to find a key equal to x, or to determine that there is no correct key equal to x. Also in this case, this is the best we can hope for, because memory faults can make x appear or disappear in the sequence at any time.

In [18] we contributed a first step in the study of resilient sorting and searching. In particular, we proved that any resilient $O(n \log n)$ comparison-based sorting algorithm can tolerate the corruption of at most $O(\sqrt{n \log n}\,)$ keys and we presented a resilient algorithm that tolerates $O(\sqrt[3]{n \log n}\,)$ memory faults. With respect to searching, we proved that any $O(\log n)$ time deterministic searching algorithm can tolerate at most $O(\log n)$ memory faults and we designed an $O(\log n)$ time searching algorithm that can tolerate up to $O(\sqrt{\log n}\,)$ memory faults.

The main contribution of this paper is to close the gaps between upper and lower bounds for resilient sorting and searching. In particular:

- We design a resilient sorting algorithm that takes $O(n \log n + \delta^2)$ worst-case time to run in the presence of δ memory faults. This yields an algorithm that can tolerate up to $O(\sqrt{n \log n}\,)$ faults in $O(n \log n)$ time: as proved in [18], this bound is optimal.
- In the special case of integer sorting, we present a randomized algorithm with expected running time $O(n + \delta^2)$: thus, this algorithm is able to tolerate up to $O(\sqrt{n}\,)$ memory faults in expected linear time.
- We prove an $\Omega(\log n + \delta)$ lower bound on the expected running time of resilient searching algorithms: this extends the lower bound for deterministic algorithms given in [18].
- We present an optimal $O(\log n + \delta)$ time randomized algorithm for resilient searching: thus, this algorithm can tolerate up to $O(\log n)$ memory faults in $O(\log n)$ expected time.
- We design an almost optimal $O(\log n + \delta^{1+\epsilon'})$ time deterministic searching algorithm, for any constant $\epsilon' \in (0, 1]$: this improves over the $O(\log n + \delta^2)$ bound of [18] and yields an algorithm that can tolerate up to $O((\log n)^{1-\epsilon})$ faults, for any small positive constant ϵ.

Notation. We recall that δ is an upper bound on the total number of memory faults. We also denote by α the *actual* number of faults that happen during a specific execution of an algorithm. Note that $\alpha \leq \delta$. We say that a key is *faithful* if its value is never corrupted by any memory fault, and *faulty* otherwise. A sequence is *faithfully ordered* if its faithful keys are sorted, and *k-unordered* if there exist k (faithful) keys whose removal makes the remaining subsequence faithfully ordered (see Figure 1). Given a sequence X of length n, we use $X[a\,;\,b]$, with $1 \leq a \leq b \leq n$, as a shortcut for the subsequence $\{X[a], X[a+1], \ldots, X[b]\}$. Two keys $X[p]$ and $X[q]$, with $p < q$, form an *inversion* in the sequence X if $X[p] > X[q]$. Note that, for any two keys forming an inversion in a faithfully ordered sequence, at least one of them must be faulty. A sorting or merging algorithm is called *resilient* if it produces a faithfully ordered sequence.

Organization. The remainder of this paper is organized as follows. In Section 2 we present our improved comparison-based sorting algorithm: the key-ingredient of the algorithm is a refined merging procedure, which is described in Subsection 2.1. Section 3 is devoted to

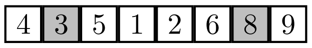

Figure 1: An input sequence X, where white and gray locations indicate faithful and faulty values, respectively. Sequence X is not faithfully ordered, but it is 2-unordered since the subsequence X' obtained by removing elements $X[4] = 1$ and $X[5] = 2$ from X is faithfully ordered. Elements $X'[1] = 4$ and $X'[2] = 3$ form an inversion in X'.

integer sorting, and shows how to derive both deterministic (Subsection 3.1) and randomized (Subsection 3.2) algorithms. Searching is discussed in Section 4: we prove a lower bound on randomized searching (Subsection 4.1), and then we provide an optimal randomized algorithm (Subsection 4.2) and an almost-optimal deterministic algorithm (Subsection 4.3). The paper ends with a list of concluding remarks.

2 Optimal Resilient Sorting in the Comparison Model

In this section we describe a resilient sorting algorithm that takes $O(n \log n + \delta^2)$ worst-case time to run in the presence of δ memory faults. This yields an $O(n \log n)$ time algorithm that can tolerate up to $O(\sqrt{n \log n}\,)$ faults: as proved in [18], this bound is optimal if we wish to sort in $O(n \log n)$ time, and improves over the best known resilient algorithm, which was able to tolerate only $O(\sqrt[3]{n \log n}\,)$ memory faults [18]. We first present a fast resilient merging algorithm, that may nevertheless fail to insert all the input values in the faithfully ordered output sequence. We next show how to use this algorithm to solve the resilient sorting problem within the claimed $O(n \log n + \delta^2)$ time bound.

2.1 The Purifying Merge Algorithm

Let X and Y be the two faithfully ordered sequences of length n to be merged. The merging algorithm that we are going to describe produces a faithfully ordered sequence Z and a disordered fail sequence F in $O(n + \alpha\,\delta)$ worst-case time. It will be guaranteed that $|F| = O(\alpha)$, i.e., that only $O(\alpha)$ keys can fail to get inserted into Z.

The algorithm, called `PurifyingMerge`, uses two auxiliary input buffers of size $(2\delta + 1)$ each, named $\mathcal{X}$ and $\mathcal{Y}$, and an auxiliary output buffer of size δ, named $\mathcal{Z}$. The merging process is divided into rounds: the algorithm maintains the invariant that, at the beginning of each round, input buffer $\mathcal{X}$ ($\mathcal{Y}$) is filled with the first values of the corresponding input sequence X (Y), while the output buffer is empty. Note that in the later rounds of the algorithm an input sequence might not contain enough values to fill in completely the corresponding input buffer, which thus can contain less than $(2\delta + 1)$ values.

Each round consists of merging the content of the input buffers into the output buffer, until either the output buffer becomes full, or the input buffers become empty, or an inconsistency in the input keys is found. In the latter case, we perform a *purifying step*, where two keys are moved to the fail sequence F, and then we restart the round from scratch. Otherwise, the content of $\mathcal{Z}$ is appended to Z, and a new round is started.

We now describe a generic round in more detail. By i' and j' we denote the number of elements in $\mathcal{X}$ and $\mathcal{Y}$ at the beginning of the round, respectively. The algorithm fills buffer $\mathcal{Z}$ by scanning the input buffers $\mathcal{X}$ and $\mathcal{Y}$ sequentially. Let i, j, and p be the running indices on $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$. Initially $i = j = p = 1$. We call $\mathcal{X}[i]$, $\mathcal{Y}[j]$, and $\mathcal{Z}[p]$ the *top keys* of $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$, respectively. All the indices and the top keys are stored in the $O(1)$ size safe memory. In each merging step we perform the following operations. If $\mathcal{X}$ and $\mathcal{Y}$ are both empty (i.e., $i > i'$ and $j > j'$), or $\mathcal{Z}$ is full (i.e., $p > \delta$), the round ends. Otherwise, assume $\mathcal{X}[i] \leq \mathcal{Y}[j]$ or $\mathcal{Y}$ is empty (the other case being symmetric). If $i = i'$, $\mathcal{X}[i]$ is the last value remaining in $\mathcal{X}$, and thus also in X: in this case we copy $\mathcal{X}[i]$ into $\mathcal{Z}[p]$, and we increment i and p. Otherwise ($i < i'$), we check whether $\mathcal{X}[i] \leq \mathcal{X}[i+1]$ (*inversion check*). If the check succeeds, $\mathcal{X}[i]$ is copied into $\mathcal{Z}[p]$ and indices i and p are incremented. If the check fails, i.e., $\mathcal{X}[i] > \mathcal{X}[i+1]$, we perform a purifying step on $\mathcal{X}[i]$ and $\mathcal{X}[i+1]$:

(1) we move these two keys to the fail sequence F,

(2) we append two new keys (if any) from X at the end of buffer $\mathcal{X}$, and

(3) we restart the merging process of the buffers $\mathcal{X}$ and $\mathcal{Y}$ from scratch by resetting all the buffer indices properly (in particular, the output buffer $\mathcal{Z}$ is emptied).

Thanks to the comparisons between the top keys and to the inversion checks, the top key of $\mathcal{Z}$ is always smaller than or equal to the top keys of $\mathcal{X}$ and $\mathcal{Y}$ (considering their values stored in safe memory): we call this *top invariant.*

At the end of the round, we check whether all the remaining keys in $\mathcal{X}$ and $\mathcal{Y}$ are larger than or equal to the top key of $\mathcal{Z}$ (*safety check*). If the safety check fails on $\mathcal{X}$, the top invariant guarantees that there is an inversion between the current top key $\mathcal{X}[i]$ of $\mathcal{X}$ and another key remaining in $\mathcal{X}$: in that case, we execute a purifying step on those two keys. We proceed analogously if the safety check fails on $\mathcal{Y}$. If all the checks succeed, the content of $\mathcal{Z}$ is flushed to the output sequence Z and the input buffers $\mathcal{X}$ and $\mathcal{Y}$ are refilled with an appropriate number of new keys taken from X and Y, respectively.

Lemma 1 *The output sequence Z is faithfully ordered.*

Proof. We say that a round is *successful* if it terminates by flushing the output buffer into Z, and *failing* if it terminates by adding keys to the fail sequence F. Since failing rounds do not modify Z, it is sufficient to consider successful rounds only. Let $\mathcal{X}'$ and X' be the remaining keys in $\mathcal{X}$ and X, respectively, at the end of a successful round. The definition of $\mathcal{Y}'$ and Y' is similar. We will show that:

(1) buffer $\mathcal{Z}$ is faithfully ordered, and

(2) all the faithful keys in $\mathcal{Z}$ are smaller than or equal to the faithful keys in $\mathcal{X}' \cup X'$ and $\mathcal{Y}' \cup Y'$.

The lemma will then follow by induction on the number of successful rounds.

Let us first show (1). We denote by $\widetilde{\mathcal{Z}}$ the ordered sequence of values inserted into $\mathcal{Z}$ at the time of their insertion. The sequence $\widetilde{\mathcal{Z}}$ must be sorted. In fact, consider any two consecutive elements $\widetilde{\mathcal{Z}}[h]$ and $\widetilde{\mathcal{Z}}[h+1]$ of $\widetilde{\mathcal{Z}}$. It is sufficient to show that $\widetilde{\mathcal{Z}}[h] \leq \widetilde{\mathcal{Z}}[h+1]$. Let $\mathcal{X}[i]$ and $\mathcal{Y}[j]$ be the top keys of $\mathcal{X}$ and $\mathcal{Y}$, respectively, right before $\widetilde{\mathcal{Z}}[h]$ is inserted into $\mathcal{Z}$. Without loss of generality, let us assume $\mathcal{X}[i] \leq \mathcal{Y}[j]$ (the other case being symmetric). As a consequence, $\mathcal{X}[i]$ is copied into $\mathcal{Z}$, the algorithm verifies that $\mathcal{X}[i] \leq \mathcal{X}[i+1]$ (otherwise the corresponding inversion check would fail), and $\mathcal{X}[i+1]$ becomes the next top key of $\mathcal{X}$. Then

$$\widetilde{\mathcal{Z}}[h] = \mathcal{X}[i] = \min\{\mathcal{X}[i], \mathcal{Y}[j]\} \leq \min\{\mathcal{X}[i+1], \mathcal{Y}[j]\} = \widetilde{\mathcal{Z}}[h+1].$$

It then follows that $\mathcal{Z}$ is faithfully ordered, being $\widetilde{\mathcal{Z}}[h] = \mathcal{Z}[h]$ for each faithful key $\mathcal{Z}[h]$.

Consider now (2). Let $z = \widetilde{\mathcal{Z}}[k]$ be the largest faithful key in $\mathcal{Z}$, and x be the smallest faithful key in $\mathcal{X}' \cup X'$. We will show that $z \leq x$ (if one of the two keys does not exist, there is nothing to prove). Note that x must belong to $\mathcal{X}'$. In fact, all the faithful keys in $\mathcal{X}'$ are smaller than or equal to the faithful keys in X'. Moreover, either $\mathcal{X}'$ contains at least $(\delta+1)$ keys (and thus at least one faithful key), or X' is empty. All the keys in $\mathcal{X}'$ are compared with $\widetilde{\mathcal{Z}}[\delta]$ during the safety check. In particular, $x \geq \widetilde{\mathcal{Z}}[\delta]$ since the safety check was successful. From the order of $\widetilde{\mathcal{Z}}$, we obtain $\widetilde{\mathcal{Z}}[\delta] \geq \widetilde{\mathcal{Z}}[k] = z$, thus implying $x \geq z$. A symmetric argument shows that z is smaller than or equal to the smallest faithful key y in $\mathcal{Y}' \cup Y'$. □

We now summarize the performance of algorithm `PurifyingMerge`.

Lemma 2 *Given two faithfully ordered sequences of length n, algorithm* `PurifyingMerge` *merges the sequences in $O(n+\alpha\,\delta)$ worst-case time. The algorithm returns a faithfully ordered sequence Z and a fail sequence F such that $|F| = O(\alpha)$.*

Proof. The faithful order of Z follows from Lemma 1. The two values discarded in each failing round form an inversion in one of the input sequences, which are faithfully ordered. Thus, at least one of such discarded values must be corrupted, proving that the number of corrupted values in F at any time is at least $|F|/2$. This implies that $|F|/2 \leq \alpha$ and that the number of failing rounds is bounded above by α. Note that at each round we spend time $\Theta(\delta)$. When the round is successful, this time can be amortized against the time spent to flush δ values to the output sequence. We therefore obtain a total running time of $O(n+\alpha\,\delta)$. □

2.2 The Sorting Algorithm

We first notice that a naive resilient sorting algorithm can be easily obtained from a bottom-up iterative implementation of mergesort by taking the minimum among $(\delta+1)$ keys per sequence at each merge step. We call this `NaiveSort`.

Lemma 3 **[18]** *Algorithm* `NaiveSort` *faithfully sorts n keys in $O(\delta\, n \log n)$ worst-case time. The running time becomes $O(\delta\, n)$ when $\delta = \Omega(n^{\epsilon})$, for some $\epsilon > 0$.*

In order to obtain a more efficient sorting algorithm, we will use the following merging subroutine, called `ResilientMerge` (see also Figure 2). We first merge the input sequences using algorithm `PurifyingMerge`: this produces a faithfully ordered sequence Z and a disordered fail sequence F. We sort F with algorithm `NaiveSort` and produce a faithfully ordered sequence F'. We finally merge Z and F' using the algorithm `UnbalancedMerge` of [18], which has the following time bound:

Lemma 4 **[18]** *Two faithfully ordered sequences of length n_1 and n_2, with $n_2 \leq n_1$, can be merged faithfully in $O(n_1 + (n_2+\alpha)\,\delta)$ worst-case time.*

We now analyze the running time of algorithm `ResilientMerge`.

Lemma 5 *Algorithm* `ResilientMerge`*, given two faithfully ordered sequences of length n, faithfully merges the sequences in $O(n+\alpha\,\delta)$ worst-case time.*

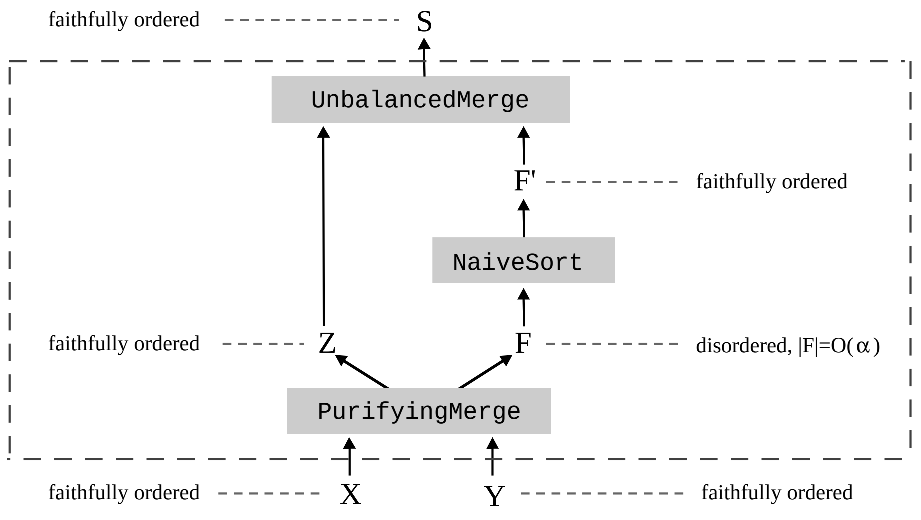

Figure 2: Our resilient merging algorithm.

Proof. By Lemma 2, algorithm `PurifyingMerge` requires time $O(n + \alpha\,\delta)$ and produces a disordered fail sequence F of length $O(\alpha)$. Since $\delta = \Omega(\alpha)$, the total time required by algorithm `NaiveSort` to produce a faithfully sorted sequence F' from F is $O(\alpha\,\delta)$ by Lemma 3. Finally, by Lemma 4 algorithm `UnbalancedMerge` takes time $O(|Z|+(|F'|+\alpha)\,\delta) = O(n+\alpha\,\delta)$ to merge Z and F'. The total running time immediately follows. □

Algorithm `ResilientMerge` has the property that only the keys corrupted while merging may be out of order in the output sequence. Hence, if we plug this algorithm into an iterative bottom-up implementation of mergesort, we obtain the following:

Theorem 1 *There is a resilient algorithm that faithfully sorts n keys in $O(n \log n + \alpha\,\delta)$ worst-case time and linear space.*

This yields an $O(n \log n)$ time resilient sorting algorithm that can tolerate up to $O(\sqrt{n \log n}\,)$ memory faults. As shown in [18], no better bound is possible.

3 Resilient Integer Sorting

In this section we consider the problem of faithfully sorting a sequence of n integers in the range $[0, n^c - 1]$, for some constant $c \geq 0$. We will present a randomized algorithm with expected running time $O(n + \delta^2)$: thus, this algorithm is able to tolerate up to $O(\sqrt{n}\,)$ memory faults in expected linear time. Our algorithm is a resilient implementation of (least significant digit) `RadixSort`, which works as follows. Assume that the integers are represented in base b, with $b \geq 2$. At the i-th step, for $1 \leq i \leq \lceil c \log_b n \rceil$, we sort the integers according to their i-th least significant digit using a linear-time, stable bucket sorting algorithm (with b buckets).

We can easily implement `RadixSort` in faulty memory whenever the base b is constant: we keep an array of size n for each bucket and store the address of those arrays and their current length (i.e., the current number of items in each bucket) in the $O(1)$-size safe memory. Since there is only a constant number of buckets, we can conclude:

Lemma 6 *There is a resilient algorithm that faithfully sorts n polynomially bounded integers in $O(n \log n)$ worst-case time and linear space, while tolerating an arbitrary number of memory faults.*

Proof. The value v_i of the i-th digit of a given integer v influences the position of the integer itself only in the i-th step, when we have to determine to which bucket v must be appended. Let us call the value of v_i at that time its *virtual value*. Clearly, the algorithm correctly sorts the sequence according to the virtual values of the digits. The claim follows by observing that the real and virtual values of faithful elements are equal. □

In order to make `RadixSort` run in linear time, we need $b = \Omega(n^\gamma)$, for some constant $\gamma \in (0, 1]$. Unfortunately, if the number of buckets is not constant, the trivial approach above does not work. In fact, we would need more than linear space to store the arrays. More important, $O(1)$ safe memory words would not be sufficient to store the initial address and the current length of the b arrays. We will next show how to overcome both problems.

Consider the i-th round of `RadixSort`. Assume that, at the beginning of the round, we are given a sequence of integers faithfully sorted up to the $(i-1)$-th least significant digit. The problem is to fill in each bucket j, $j \in \{0, 1, \ldots, b-1\}$, with the faithfully sorted subsequence of values with the i-th digit equal to j. Eventually, the ordered concatenation of the buckets' content gives the input sequence (faithfully sorted up to the i-th digit) for the next round.

The core of our method is the way we fill in the buckets. We achieve this task by associating to each bucket a hierarchy of intermediate buffers. Each time, during the scan of the input sequence, we find an integer κ bounded to the j-th bucket, we execute a *bucket-filling* procedure that copies κ into one of the buffers associated to that bucket. Only from time to time our procedure moves the values from the buffers to the bucket.

Observe that we cannot store the address and current length of the buffers and of the buckets in safe memory (since this would take $\Omega(b)$ safe memory words). We solve the problem with the addresses by packing variables in the faulty memory so as that a unique address β, kept in safe memory, is sufficient to reach any variable in our data structure. In order to circumvent the problem with buffers' and buckets' lengths, we will use redundant variables, defined as follows. A *redundant $|p|$-index p* is a set of $|p|$ positive integers. The *value* of p is the majority value in the set (or an arbitrary value if no majority value exists). Assigning a value x to p means assigning x to all its elements: note that both reading and updating p can be done in linear time and constant space (using, e.g., the algorithm in [9]). If $|p| \geq 2\delta + 1$, we say that p is *reliable* (i.e., we can consider its value faithful even if p is stored in faulty memory). A *redundant $|p|$-pointer p* is defined analogously, with positive integers replaced by pointers.

We periodically restore the ordering inside the buffers by means of a (bidirectional) `BubbleSort`, which works as follows: we compare adjacent pairs of keys, swapping them if necessary, and alternately pass through the sequence from the beginning to the end (*forward pass*) and from the end to the beginning (*backward pass*), until no more swaps are performed. Interestingly enough, bidirectional `BubbleSort` is resilient to memory faults and its running time depends only on the disorder of the input sequence and on the actual number of faults occurring during its execution.

Lemma 7 *Given a sequence of length n containing at most k values which are out of order, algorithm* `BubbleSort` *faithfully sorts the sequence in $O(n + (k + \alpha)\, n)$ worst-case time, where α is the number of memory faults happening during the execution of the algorithm.*

Proof. Let us call *scan* of the input sequence the execution of a forward pass followed by a backward pass, and let α_i be the number of memory faults happening during the i-th scan. Recall that, in safe-memory systems, bidirectional `BubbleSort` reduces the number of out-of-order values at least by one at each scan (while this does not hold for standard `BubbleSort`). By essentially the same arguments, after the j-th scan, at most $k + \sum_{i=1}^{j} \alpha_i - j$ (faithful and faulty) values are out of order. Thus, the algorithm halts within at most $(k + \alpha + 1)$ scans. When this happens, all the faithful values are sorted. $\square$

The rest of this section is organized as follows. In Section 3.1 we describe a deterministic integer sorting algorithm of running time $O(n + b\,\delta + \alpha\,\delta^{1+\epsilon})$, and space complexity $O(n + b\,\delta)$. In Section 3.2 we describe a randomized integer sorting algorithm with the same space complexity, and expected running time $O(n + b\,\delta + \alpha\,\delta)$.

3.1 A Deterministic Algorithm

Consider the subsequence of $\tilde{n}$ integer keys which need to be inserted into a given bucket j during the i-th iteration of `RadixSort`. We remark that such subsequence is faithfully sorted up to the i-th digit (since the i-th digit is equal for all the keys considered, and the remaining digits are faithfully sorted by the properties of the algorithm). From the discussion above, it is sufficient to describe how the bucket considered is filled in. In this section we will describe a procedure to accomplish this task in $O(\tilde{n} + \delta + \tilde{\alpha}\delta^{1+\epsilon})$ time and $O(\tilde{n} + \delta)$ space, for any given constant $\epsilon > 0$, where $\tilde{\alpha}$ is the actual number of faults affecting the procedure considered. Filling in each bucket with this procedure yields an integer sorting algorithm with $O(n + b\,\delta + \alpha\,\delta^{1+\epsilon})$ running time and $O(n + b\,\delta)$ space complexity.

For the sake of simplicity, let us first describe a bucket-filling procedure of running time $O(\tilde{n} + \delta + \tilde{\alpha}\delta^{1.5})$. Moreover, let us implement the bucket considered with an array $\mathcal{B}_0$ of length $(n + \delta)$. We will later show how to reduce the space usage to $O(\tilde{n} + \delta)$ by means of doubling techniques. Besides the output array $\mathcal{B}_0$, we use two buffers to store temporarily the input keys: a buffer $\mathcal{B}_1$ of size $|\mathcal{B}_1| = 2\delta + 1$, and a buffer $\mathcal{B}_2$ of size $|\mathcal{B}_2| = 2\sqrt{\delta} + 1$. All the entries of both buffers are initially set to a value, say $+\infty$, that is not contained in the input sequence. We associate a redundant index p_i to each $\mathcal{B}_i$, where $|p_0| = |\mathcal{B}_1| = 2\delta + 1$, $|p_1| = |\mathcal{B}_2| = 2\sqrt{\delta} + 1$, and $|p_2| = 1$ (see Figure 3). Index p_i points to the first free position of bucket $\mathcal{B}_i$. Note that only p_0 is reliable, while p_1 and p_2 could assume faulty values.

We store all the $\mathcal{B}_i$'s and p_i's consecutively in the unsafe memory in a given order, so that we can derive the address of each element from the address of the first element. Observe that this is possible since each buffer and index occupies a fixed number of memory words. We pack analogously the $\mathcal{B}_i$'s and p_i's corresponding to different buckets. This way, a unique address β is sufficient to derive the addresses of all the items in our data structure: we maintain such address β in the safe memory.

The bucket-filling procedure works as follows. Let κ be the current key which needs to be inserted in the bucket considered. Key κ is initially appended to $\mathcal{B}_2$. Whenever $\mathcal{B}_2$ is full (according to index p_2), we *flush* it as follows:

(1) we remove any $+\infty$ from $\mathcal{B}_2$ and sort $\mathcal{B}_2$ with `BubbleSort` considering the i least significant digits only;

(2) we append $\mathcal{B}_2$ to $\mathcal{B}_1$, and we update p_1 accordingly;

(3) we reset $\mathcal{B}_2$ and p_2.

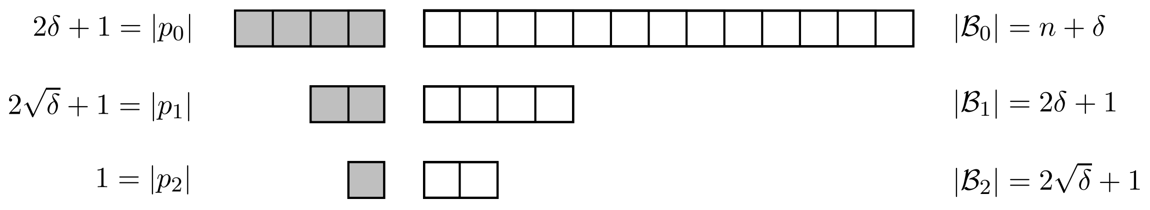
$2\delta + 1 = |p_0|$
$2\sqrt{\delta} + 1 = |p_1|$
$1 = |p_2|$
$|\mathcal{B}_0| = n + \delta$
$|\mathcal{B}_1| = 2\delta + 1$
$|\mathcal{B}_2| = 2\sqrt{\delta} + 1$

Figure 3: Indexes (on the left) and buckets (on the right) in the bucket-filling procedure.

Whenever $\mathcal{B}_1$ is full, we *flush* it in a similar way, moving its keys to $\mathcal{B}_0$. We flush buffer $\mathcal{B}_j$, $j \in \{1, 2\}$, also whenever we realize that the index p_j points to an entry outside $\mathcal{B}_j$ or to an entry of value different from $+\infty$ (which indicates that a fault happened either in p_j or in $\mathcal{B}_j$ after the last time $\mathcal{B}_j$ was flushed). Note that, because of faults, the total number of elements moved to $\mathcal{B}_0$ can be larger than $\tilde{n}$. However, there can be at most one extra value per fault affecting the buffers. This is the reason why we set the size of $\mathcal{B}_0$ to $n + \delta$.

In order to ensure that each faithful value is eventually contained in the corresponding bucket, we force the flushing of all the buffers when the very last element in the input sequence is scanned. Since this last step does not increase the asymptotic running time, we will not mention it any further.

Notice that all the faithful input keys eventually appear in $\mathcal{B}_0$, faithfully sorted up to the i-th digit. In fact, all the faithful keys in $\mathcal{B}_j$, $j \in \{1, 2\}$, at a given time precede the faithful keys not yet copied into $\mathcal{B}_j$. Moreover we sort $\mathcal{B}_j$ before flushing it. This guarantees that the faithful keys are moved from $\mathcal{B}_j$ to $\mathcal{B}_{j-1}$ in a first-in-first-out fashion.

Lemma 8 *The bucket-filling procedure above takes $O(\tilde{n} + \delta + \tilde{\alpha}\,\delta^{1.5})$ worst-case time, where $\tilde{n}$ is the number of keys inserted and $\tilde{\alpha}$ the actual number of faults affecting the procedure.*

Proof. Consider the cost paid by the algorithm between two consecutive flushes of $\mathcal{B}_1$. Let α' and α'' be the number of faults in $\mathcal{B}_1$ and p_1, respectively, during the phase considered. If no fault happens in either $\mathcal{B}_1$ or p_1 ($\alpha' + \alpha'' = 0$), flushing buffer $\mathcal{B}_1$ costs $O(|\mathcal{B}_1|) = O(\delta)$ time. If the value of p_1 is faithful, the sequence is $O(\alpha')$-unordered: in fact, removing the corrupted values from $\mathcal{B}_1$ produces a sorted subsequence. Thus sorting $\mathcal{B}_1$ costs $O((1 + \alpha')\delta)$ time. Otherwise ($\alpha'' > \sqrt{\delta}$), the sequence $\mathcal{B}_1$ can be $O(\delta)$-unordered and sorting it requires $O((1 + \delta + \alpha')\delta) = O(\delta^2)$ time. Thus, the total cost of flushing buffer $\mathcal{B}_1$ is $O(\tilde{n} + \delta + \tilde{\alpha}/\sqrt{\delta}\,\delta^2 + \tilde{\alpha}\,\delta) = O(\tilde{n} + \delta + \tilde{\alpha}\,\delta^{1.5})$. By a similar argument, the total cost of flushing buffer $\mathcal{B}_2$ is $O(\tilde{n} + \sqrt{\delta} + \tilde{\alpha}(\sqrt{\delta})^2 + \tilde{\alpha}\sqrt{\delta}) = O(\tilde{n} + \sqrt{\delta} + \tilde{\alpha}\delta)$. The claimed running time immediately follows. □

Saving Space. The bucket-filling procedure described above uses $O(n + \delta)$ space. The space usage can be easily reduced to $O(\tilde{n} + \delta)$ via doubling, without increasing the asymptotic running time. We replace $\mathcal{B}_0$ in our data structure with a $(2\delta+1)$-pointer p to $\mathcal{B}_0$. By the same argument used before, the address of p can be derived from the unique address β stored in safe memory. We initially impose $|\mathcal{B}_0| = \delta$. When $\mathcal{B}_0$ is full, we create a new array $\mathcal{B}'_0$ of size $|\mathcal{B}'_0| = 2|\mathcal{B}_0|$, we copy $\mathcal{B}_0$ into the first $|\mathcal{B}_0|$ entries of $\mathcal{B}'_0$, and we make p point to $\mathcal{B}'_0$. The amortized cost per update does not increase, and the space complexity of the new bucket-filling procedure is $O(\tilde{n}+\delta)$.

Multi-Level Buffering. The deterministic running time can be improved by increasing the number of buffers and by choosing more carefully the buffer sizes: specifically, with $k \geq 2$ buffers, we can achieve a $O(k\,\tilde{n}+\delta+\alpha\,\delta^{2^k/(2^k-1)})$ running time. This yields an integer sorting algorithm with $O(n+b\,\delta+\alpha\,\delta^{1+\epsilon})$ worst-case running time, for any small positive constant ϵ.

In more detail, instead of two buffers, we use a constant number $k \geq 2$ of buffers, $\mathcal{B}_1$, $\mathcal{B}_2,\ldots,\mathcal{B}_k$, each one with its own redundant-index. The size of buffer $\mathcal{B}_i$, $i = 1, 2, \ldots, k$, is:

$$|\mathcal{B}_i| = 2\delta^{(2^k-2^{i-1})/(2^k-1)} + 1.$$

The size of the redundant-indexes is $|p_k| = 1$, and $|p_i| = |\mathcal{B}_{i+1}|$ for $i = 0, 1, \ldots, k-1$. The bucket-filling procedure works analogously: the elements are first stored in $\mathcal{B}_k$, and then gradually moved to larger and larger buffers, up to $\mathcal{B}_0$.

Let α_i be the total number of faults occurring in p_i and $\mathcal{B}_i$. In particular, by α'_i and α''_i we denote the faults occurring in $\mathcal{B}_i$ and p_i, respectively. Consider the cost paid by the algorithm between two consecutive flushes of $\mathcal{B}_i$. Following the analysis of Section 3.1, if no fault occurs, such cost is $O(|\mathcal{B}_i|)$. Otherwise ($\alpha_i > 0$), if p_i is faithful, the cost is $O(|\mathcal{B}_i|+\alpha'_i|\mathcal{B}_i|) = O(\alpha_i|\mathcal{B}_i|)$. In the remaining case ($\alpha''_i \geq |p_i|/2$), such cost is $O(|\mathcal{B}_i|^2+\alpha'_i|\mathcal{B}_i|)$, but it can be amortized over the $\Omega(|p_i|)$ faults in p_i. Altogether, flushing buffer $\mathcal{B}_i$, $i \in \{1, 2, \ldots, k\}$, costs

$$O(\tilde{n}+|\mathcal{B}_i|+\alpha_i(|\mathcal{B}_i|+|\mathcal{B}_i|^2/|p_i|)) = O(\tilde{n}+(1+\alpha_i)\delta^{(2^k-2^{i-1})/(2^k-1)}).$$

$$O(\tilde{n}+|\mathcal{B}_i|+\alpha_i(|\mathcal{B}_i|+|\mathcal{B}_i|^2/|p_i|)) = O(\tilde{n}+\delta^{(2^k-2^{i-1})/(2^k-1)}+\alpha_i\,\delta^{2^k/(2^k-1)}).$$

Thus the time complexity of the algorithm is

$$O(k\,\tilde{n}+\delta+\sum_{i=1}^{k}(1+\alpha_i)\delta^{(2^k-2^{i-1})/(2^k-1)}) = O(k\,\tilde{n}+\delta+\tilde{\alpha}\,\delta^{2^k/(2^k-1)}).$$

Lemma 9 *For every $k \geq 2$, there is a deterministic bucket-filling procedure of time complexity $O(k\,\tilde{n}+\delta+\tilde{\alpha}\,\delta^{2^k/(2^k-1)})$, where $\tilde{n}$ is the number of keys inserted and $\tilde{\alpha}$ the actual number of faults affecting the procedure.*

Note that, for $k = 2$, we improve over the result of Lemma 8. Indeed, the size of the buffers in that case has been chosen so as to minimize the time complexity of the randomized procedure of next section.

Theorem 2 *For any constant $\epsilon \in (0,1)$, and assuming $\delta = O(n^{1-\gamma})$ for some constant $\gamma > 0$, there is a deterministic resilient algorithm that faithfully sorts n polynomially bounded integers in $O(n+\alpha\,\delta^{1+\epsilon})$ time and $O(n)$ space.*

Proof. It is sufficient to implement `RadixSort` with base $b = \Theta(n^{\gamma})$, via the randomized bucket-filling procedure mentioned above with $k = \lceil \log_2 \frac{1+\epsilon}{\epsilon} \rceil$. Consider the i-th step of `RadixSort`. Let n_j denote the number of integers copied into bucket j, and let α_j be the actual number of memory faults affecting the execution of the j-th instance of the bucket-filling procedure. The running time of the i-th step is $O(\sum_{j=0}^{b-1}(n_j + \delta + \alpha_j\, \delta^{2^k/(2^k-1)})) = O(n + b\,\delta + \alpha\,\delta^{1+\epsilon}) = O(n + \alpha\,\delta^{1+\epsilon})$. The claim on the running time follows by observing that the total number of such steps is $O(\log_b n^c) = O(\log_{n^{\gamma}} n) = O(1)$. The space usage is $O(\sum_{j=0}^{b-1}(n_j + \delta)) = O(n + b\,\delta) = O(n)$. □

3.2 A Randomized Algorithm

We now show how to reduce the (expected) running time of the bucket-filling procedure to $O(\tilde{n} + \delta + \tilde{\alpha}\,\delta)$, by means of randomization. We build up on the simple deterministic bucket-filling procedure of running time $O(\tilde{n} + \delta + \tilde{\alpha}\,\delta^{1.5})$ described before. Notice that a few corruptions in p_1 can lead to a highly disordered sequence $\mathcal{B}_1$. Consider for instance the following situation: we corrupt p_1 twice, in order to force the algorithm to write first δ faithful keys in the second half of $\mathcal{B}_1$, and then other $(\delta+1)$ faithful keys in the first half of $\mathcal{B}_1$. This way, with $2(\sqrt{\delta}+1)$ corruptions only, one obtains an $O(\delta)$-unordered sequence, whose sorting requires $O(\delta^2)$ time. This can happen $O(\alpha/\sqrt{\delta})$ times, thus leading to the $O(\alpha\,\delta^{1.5})$ term in the running time.

The idea behind the randomized algorithm is to try to avoid pathological situations. Specifically, we would like to detect early the fact that many values after the last inserted key are different from $+\infty$. In order to do that, whenever we move a key from $\mathcal{B}_2$ to $\mathcal{B}_1$, we select an entry uniformly at random in the portion of $\mathcal{B}_1$ after the last inserted key: if the value of this entry is not $+\infty$, the algorithm flushes $\mathcal{B}_1$ immediately.

Lemma 10 *The bucket-filling procedure above takes $O(\tilde{n} + \delta + \tilde{\alpha}\,\delta)$ expected time, where $\tilde{n}$ is the number of keys inserted and $\tilde{\alpha}$ the actual number of faults affecting the procedure.*

Proof. Let α' and α'' be the number of faults in $\mathcal{B}_1$ and p_1, respectively, between two consecutive flushes of buffer $\mathcal{B}_1$. Following the proof of Lemma 8 and the discussion above, it is sufficient to show that, when we sort $\mathcal{B}_1$, the sequence to be sorted is $O(\alpha' + \alpha'')$-unordered in expectation. In order to show that, we will describe a procedure which obtains a sorted subsequence from $\mathcal{B}_1$ by removing an expected number of $O(\alpha' + \alpha'')$ keys.

First remove the α' corrupted values in $\mathcal{B}_1$. Now consider what happens either between two consecutive corruptions of p_1 or between a corruption and a reset of p_1. Let $\widetilde{p_1}$ be the value of p_1 at the beginning of the phase considered. By A and B we denote the subset of entries of value different from $+\infty$ in $\mathcal{B}_1$ at index larger than $\widetilde{p_1}$, and the subset of keys added to $\mathcal{B}_1$ in the phase considered, respectively. Note that, when A is large, the expected cardinality of B is small (since it is more likely to select randomly an entry in A). More precisely, the probability of selecting at random an entry of A is at least $|A|/|\mathcal{B}_1|$. Thus the expected cardinality of B is at most $|\mathcal{B}_1|/|A| = O(\delta/|A|)$.

The idea behind the proof is to remove A from $\mathcal{B}_1$ if $|A| < \sqrt{\delta}$, and to remove B otherwise. In both cases the expected number of keys removed is $O(\sqrt{\delta}\,)$. At the end of the process, we obtain a sorted subsequence of $\mathcal{B}_1$. Since p_1 can be corrupted at most $O(\alpha''/\sqrt{\delta}\,)$ times, the total expected number of keys removed is $O(\alpha' + \sqrt{\delta}\,\alpha''/\sqrt{\delta}\,) = O(\alpha' + \alpha'')$. □

By essentially the same arguments as in the proof of Theorem 2, we obtain the following result.

Theorem 3 *Assuming $\delta = O(n^{1-\gamma})$ for some constant $\gamma > 0$, there is a randomized resilient algorithm that faithfully sorts n polynomially bounded integers in $O(n + \alpha\,\delta)$ expected time and $O(n)$ space.*

4 Resilient Searching in the Comparison Model

In this section we prove upper and lower bounds on the resilient searching problem in the comparison model. Namely, we first prove an $\Omega(\log n+\delta)$ lower bound on the expected running time, and then we present an optimal $O(\log n + \delta)$ expected time randomized algorithm. Finally, we describe an $O(\log n + \delta^{1+\epsilon'})$ time deterministic algorithm, for any constant $\epsilon' \in (0, 1]$. Both our algorithms improve over the $O(\log n + \delta^2)$ deterministic bound of [18].

4.1 A Lower Bound for Randomized Searching

We now show that every comparison-based searching algorithm, even randomized ones, which tolerates up to δ memory faults must have expected running time $\Omega(\log n + \delta)$ on sequences of length n, with $n \geq \delta$.

Theorem 4 *Every (randomized) resilient searching algorithm must have expected running time $\Omega(\log n + \delta)$.*

Proof. An $\Omega(\log n)$ lower bound holds even when the entire memory is safe. Thus, it is sufficient to prove that every resilient searching algorithm takes expected time $\Omega(\delta)$ when $\log n = o(\delta)$. Let $\mathcal{A}$ be a resilient searching algorithm. Consider the following (feasible) input sequence I: for an arbitrary value x, the first $(\delta + 1)$ values of the sequence are equal to x and the others are equal to $+\infty$. Let us assume that the adversary arbitrarily corrupts δ of the first $(\delta + 1)$ keys before the beginning of the algorithm. Since a faithful key x is left, $\mathcal{A}$ must be able to find it.

Observe that, after the initial corruption, the first $(\delta + 1)$ elements of I form an arbitrary (unordered) sequence. Suppose by contradiction that $\mathcal{A}$ takes $o(\delta)$ expected time. Then we can easily derive from $\mathcal{A}$ an algorithm to find a given element in an unordered sequence of length $\Theta(\delta)$ in sub-linear expected time, which is not possible (even in a safe-memory system). □

4.2 Optimal Randomized Searching

In this section we present a resilient searching algorithm with optimal $O(\log n + \delta)$ expected running time. Let I be the sorted input sequence and x be the key to be searched for. At each step, the algorithm considers a subsequence $I[\ell; r]$. Initially $I[\ell; r] = I[1; n] = I$. Let $C > 1$ and $0 < c < 1$ be two constants such that $cC > 1$. The algorithm has a different behavior depending on the length of the current interval $I[\ell; r]$. If $r - \ell > C\delta$, the algorithm chooses an element $I[h]$ uniformly at random in the central subsequence of $I[\ell; r]$ of length $(r - \ell)c$, i.e., in $I[\ell'; r'] = I[\ell + (r - \ell)(1 - c)/2;\ \ell + (r - \ell)(1 + c)/2]$ (for the sake of simplicity, we neglect ceilings and floors). If $I[h] = x$, the algorithm simply returns the index h. Otherwise, it continues searching for x either in $I[\ell; h-1]$ or in $I[h+1; r]$, according to the outcome of the comparison between x and $I[h]$.

Consider now the case $r - \ell \leq C\delta$. Let us assume that there are at least 2δ values to the left of ℓ and 2δ values to the right of r (otherwise, it is sufficient to assume that $X[i] = -\infty$ for $i < 1$ and $X[i] = +\infty$ for $i > n$). If x is contained in $I[\ell - 2\delta; r + 2\delta]$, the algorithm returns the corresponding index. Else, if both the majority of the elements in $I[\ell - 2\delta; \ell]$ are smaller than x and the majority of the elements in $I[r; r+2\delta]$ are larger than x, the algorithm returns no. Otherwise, at least one of the randomly selected values must be faulty: in that case the algorithm simply restarts from the beginning.

Note that the variables needed by the algorithm require total constant space, and thus they can be stored in safe memory.

Theorem 5 *The algorithm above performs resilient searching in $O(\log n + \delta)$ expected time.*

Proof. Consider first the correctness of the algorithm. We will later show that the algorithm halts with probability one. If the algorithm returns an index, the answer is trivially correct. Otherwise, let $I[\ell; r]$ be the last interval considered before halting. According to the majority of the elements in $I[\ell - 2\delta; \ell]$, x is either contained in $I[\ell + 1; n]$ or not contained in I. This is true since the mentioned majority contains at least $(\delta + 1)$ elements, and thus at least one of them must be faithful. A similar argument applied to $I[r; r + 2\delta]$ shows that x can only be contained in $I[1; r-1]$. Since the algorithm did not find x in $I[\ell+1; n] \cap I[1; r-1] = I[\ell+1; r-1]$, there is no faithful key equal to x in I.

Now consider the time spent in one iteration of the algorithm (starting from the initial interval $I = I[1; n]$). Each time the algorithm selects a random element, either the algorithm halts or the size of the subsequence considered is decreased by at least a factor of $2/(1+c) > 1$. So the total number of selection steps is $O(\log n)$, where each step requires $O(1)$ time. The final step, where a subsequence of length at most $4\delta + C\delta = O(\delta)$ is considered, requires $O(\delta)$ time. Altogether, the worst-case time for one iteration is $O(\log n + \delta)$.

Thus, it is sufficient to show that in a given iteration the algorithm halts (that is, it either finds x or answers no) with some positive constant probability $P > 0$, from which it follows that the expected number of iterations is constant. Let $I[h_1], I[h_2], \ldots, I[h_t]$ be the sequence of randomly chosen values in a given iteration. If a new iteration starts, this implies that at least one of those values is faulty. Hence, to show that the algorithm halts, it is sufficient to prove that all those values are faithful with positive probability.

Let $\overline{P}_k$ denote the probability that $I[h_k]$ is faulty. Consider the last interval $I[\ell; r]$ in which we perform random sampling. The length of this interval is at least $C\,\delta$. So the value $I[h_t]$ is chosen in a subsequence of length at least $c\,C\,\delta > \delta$, from which we obtain

$$\overline{P}_t \leq \delta/(c\,C\,\delta) = 1/(c\,C).$$

Consider now the previous interval. The length of this interval is at least $2C\delta/(1+c)$. Thus

$$\overline{P}_{t-1} \leq (1+c)/(2\,c\,C).$$

More generally, for each $i = 0, 1, \ldots, (t-1)$, we have

$$\overline{P}_{t-i} \leq ((1+c)/2)^i / (cC).$$

Altogether, the probability P that all the values $I[h_1], I[h_2], \dots, I[h_t]$ are faithful is equal to

$$\prod_{i=0}^{t-1}(1-\overline{P}_{t-i})$$

and thus

$$P \geq \prod_{i=0}^{t-1}\left(1-\frac{1}{c\,C}\left(\frac{1+c}{2}\right)^i\right) \geq \left(1-\frac{1}{c\,C}\right)^{\sum_{i=0}^{t-1}(\frac{1+c}{2})^i} \geq \left(1-\frac{1}{c\,C}\right)^{\frac{2}{1-c}} > 0,$$

where we used the fact that $(1-xy) \geq (1-x)^y$ for every x and y in $[0,1]$. □

4.3 Almost Optimal Deterministic Searching

In this section we describe a deterministic resilient searching algorithm that requires $O(\log n + \alpha\,\delta^\epsilon)$ worst-case time, for any constant $\epsilon \in (0,1]$. We first present and analyze a simpler version with running time $O(\log n + \alpha\sqrt{\delta})$. By refining this simpler algorithm we will obtain the claimed result.

Let I be the faithfully ordered input sequence and let x be the key to be searched for. Before describing our deterministic algorithm, which we refer to as `DetSearch`, we introduce the notion of *k-left-test* and *k-right-test* over a position p, for $k \geq 1$ and $1 \leq p \leq n$. In a k-left-test over p, we consider the neighborhood of p of size k defined as $I[p-k\,;\,p-1]$: the test *fails* if the majority of keys in this neighborhood is larger than the key x to be searched for, and *succeeds* otherwise. A k-right-test over p is defined symmetrically on the neighborhood $I[p+1\,;\,p+k]$. Note that in the randomized searching algorithm described in the previous section we execute a $(2\delta+1)$-left-test and a $(2\delta+1)$-right-test at the end of each iteration. The idea behind our improved deterministic algorithm is to design less expensive left and right tests, and to perform them more frequently.

The $\mathbf{O}(\log \mathbf{n} + \alpha\,\sqrt{\delta})$ algorithm. The basic structure of the algorithm is as in the classical deterministic binary search: at each step, the algorithm considers an interval $I[\ell; r]$ and updates it as suggested by the central value $I[(\ell+r)/2]$. The algorithm may terminate returning the position $(\ell+r)/2$ if $I[(\ell+r)/2] = x$. The left and right boundaries ℓ and r are kept in safe memory. Initially $I[\ell; r] = I[1; n] = I$. At appropriate times, the algorithm performs $(2\sqrt{\delta}+1)$-tests and $(2\delta+1)$-tests. During the tests, the algorithm takes care of checking by exhaustive search whether any of the considered keys is equal to x: if this is the case, it terminates returning the position of that key. Otherwise, the outcome of the tests is used for updating the following four additional indexes, which are also stored in safe memory:

- Two left and right boundaries, ℓ_1 and r_1, witnessed by $(2\sqrt{\delta}+1)$-tests: these indexes are such that, at the time when the tests were performed, they suggested that x should belong to $I[\ell_1; r_1]$. Namely, the majority of the values in $I[\ell_1 - (2\sqrt{\delta}+1)\,;\,\ell_1 - 1]$ were smaller than x and the majority of the values in $I[r_1+1\,;\,r_1+(2\sqrt{\delta}+1)]$ were larger than x.

- Two left and right boundaries, ℓ_2 and r_2, witnessed by $(2\delta+1)$-tests: these indexes are such that, if $x \in I$, then we are guaranteed that $x \in I[\ell_2; r_2]$.

Initially, $I[\ell_1; r_1] = I[\ell_2; r_2] = I[\ell, r] = I[1; n] = I$. We now describe how the indexes are updated after each test.

Every $\sqrt{\delta}$ searching steps, we perform a $(2\sqrt{\delta}+1)$-left-test over the left boundary ℓ and a $(2\sqrt{\delta}+1)$-right-test over the right boundary r of the current interval $I[\ell; r]$. We have the following cases:

1. Both tests succeed: we set $\ell_1 = \ell$ and $r_1 = r$ (see Figure 4a).

2. Exactly one test fails: we revert to the smallest interval suggested by the failed test and by the last $(2\sqrt{\delta}+1)$-tests previously performed. Say, e.g., that the $(2\sqrt{\delta}+1)$-left-test over position ℓ fails: in that case we set $r = r_1 = \max\{\ell_1, \ell - (2\sqrt{\delta}+1) - 1\}$ and $\ell = \ell_1$, as shown in Figure 4b (note that the value of ℓ_1 remains the same). The case when a $(2\sqrt{\delta}+1)$-right-test fails is symmetric.

3. Both tests fail: in this case the two tests contradict each other, and we perform $(2\delta+1)$-tests over positions ℓ and r. Notice that at least one the $(2\sqrt{\delta}+1)$-tests must disagree with the corresponding $(2\delta+1)$-test, since $(2\delta+1)$-tests cannot be misleading. If both the $(2\sqrt{\delta}+1)$-left-test and the $(2\sqrt{\delta}+1)$-right-test disagree with their corresponding $(2\delta+1)$-tests, we set $\ell_2 = \ell_1 = \ell$ and $r_2 = r_1 = r$ (see Figure 4c). If only the left tests disagree, we set $\ell = \ell_1 = \ell_2 = \min\{r + (2\delta+1) - 1, r_2\}$ and $r = r_1 = r_2$ (see Figure 4d). The case where only the right tests disagree is symmetric.

Every δ searching steps during which no contradicting $(2\sqrt{\delta}+1)$-tests are performed, besides the final $(2\sqrt{\delta}+1)$-tests, we also perform $(2\delta+1)$-tests and appropriately update the indexes ℓ_2 and r_2. Two $(2\delta+1)$-tests are finally performed when the algorithm is about to terminate with a negative answer, i.e., when $\ell = r$ and $I[\ell] \neq x$: in this case, if both the left and the right test over position ℓ succeed, the algorithm terminates returning no, otherwise the indexes are updated and the search continues as described above.

Analysis. In order to analyze the correctness and the running time of algorithm `DetSearch`, we will say that a position p (or, equivalently, a value $I[p]$) is *misleading* if $I[p]$ is faulty and guides the search towards a wrong direction. A k-left-test over a position p is misleading if the majority of the values in $I[p-k\,;\,p-1]$ are misleading. Misleading k-right-tests are defined similarly. We note that $(2\delta+1)$-tests cannot be misleading, because there can be at most δ faulty values.

Theorem 6 *Algorithm* `DetSearch` *performs resilient searching in* $O(\log n + \alpha\sqrt{\delta}\,)$ *worst-case time.*

Proof. To show correctness, it is sufficient to prove that, in case of a negative answer, there is no correct key equal to x. The algorithm may return no only if $\ell = r$ and if both the $(2\delta+1)$-tests over position ℓ succeed: since $(2\delta+1)$-tests cannot be misleading, if the tests over ℓ succeed and $x \in I$, then x must be necessarily found in $I[\ell-(2\delta+1)\,;\,n] \cap I[1\,;\,\ell+(2\delta+1)] = I[\ell-(2\delta+1); \ell+(2\delta+1)]$. The correctness follows by noting that during the $(2\delta+1)$-tests the algorithm checks by exhaustive search whether any of the considered keys is equal to x (in that case, it would have terminated with a positive answer).

We now discuss the running time. We consider the search process as divided into δ-steps: at the end of a δ-step, two $(2\delta+1)$-tests over the current left and right boundaries are performed and all the indexes are appropriately updated. The time spent in a δ-step includes

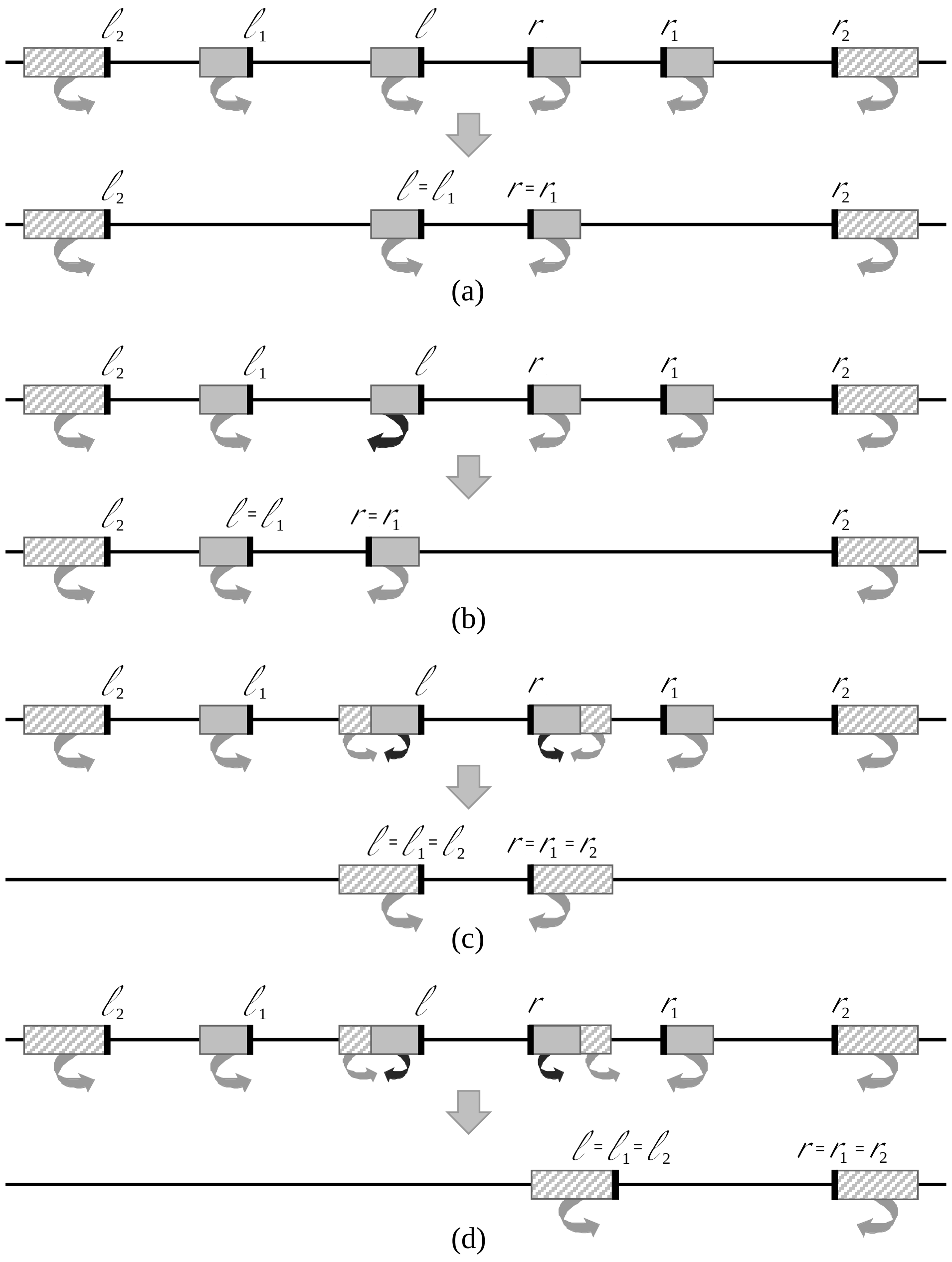

Figure 4: Index update after (a) two successful $(2\sqrt{\delta}+1)$-tests over positions ℓ and r; (b) a failing $(2\sqrt{\delta}+1)$-left-test over position ℓ and a successful $(2\sqrt{\delta}+1)$-right-test over position r; (c) two failing $(2\sqrt{\delta}+1)$-tests over positions ℓ and r, both contradicted by the corresponding $(2\delta+1)$-tests; (d) two failing $(2\sqrt{\delta}+1)$-tests over positions ℓ and r, such that only the left test is contradicted by the corresponding $(2\delta+1)$-test. The missing cases are symmetric.

the $O(1)$ time for each standard searching step, the $O(\sqrt{\delta})$ time for each $(2\sqrt{\delta}+1)$-test, and the $O(\delta)$ time for the final $(2\delta+1)$-tests. Since $(2\sqrt{\delta}+1)$-tests take place every $\sqrt{\delta}$ searching steps and the number of searching steps is at most δ, the total time for the δ-step is $O(\delta)$. Since $(2\delta+1)$-tests cannot be misleading, at the end of any δ-step the proper search direction is always recovered. Notice that a δ-step ends either due to two contradicting $(2\sqrt{\delta}+1)$-tests or because δ standard searching steps have been performed.

We first bound the time spent in δ-steps that terminate due to contradicting $(2\sqrt{\delta}+1)$-tests. Let ℓ and r be the left and right boundaries on which the contradicting $(2\sqrt{\delta}+1)$-tests are performed. Then, at least one of the $(2\sqrt{\delta}+1)$-tests must be misleading and the interval checked during the test must contain $\Theta(\sqrt{\delta})$ faulty values. If the $(2\sqrt{\delta}+1)$-left-test is misleading, the search proceeds either in $I[\ell\,;\,r]$ (see Figure 4c) or in $I[\min\{r+(2\delta+1)-1, r_2\}\,;\,r_2]$ (see Figure 4d). In both cases, the $\Theta(\sqrt{\delta})$ faulty values in $I[\ell-(2\sqrt{\delta}+1)\,;\,\ell-1]$ are eliminated from the interval in which the search proceeds. We can therefore charge the $O(\delta)$ time spent in the δ-step to these faulty values. The case where the $(2\sqrt{\delta}+1)$-right-test is misleading is symmetric. We will therefore have at most $O(\alpha/\sqrt{\delta}\,)$ δ-steps of this kind, requiring $O(\delta)$ time each, and the total running time for these steps will be $O(\alpha\sqrt{\delta}\,)$.

We now bound the time spent in δ-steps that terminate because δ standard searching steps have been performed. In these δ-steps, no contradicting $(2\sqrt{\delta}+1)$-tests are ever performed. We assume that the algorithm takes at some point a wrong search direction (*mislead search*) and we bound the running time for a mislead search, i.e., the cost of the sequence of steps performed by the algorithm before the recovery.

We first analyze the running time for a mislead search assuming that the proper search direction is recovered by a $(2\sqrt{\delta}+1)$-test. It is enough to consider the case where the algorithm encounters a misleading value $I[p]$ that guides the search towards positions larger than p: then, p will become a misleading left boundary (the case of a misleading right boundary is symmetric). Consider the time when the next $(2\sqrt{\delta}+1)$-left-test is performed, and let ℓ be the left boundary involved in the test. Note that it must be $p \le \ell$ and, since p is misleading, then ℓ must be a misleading left boundary, as well. Due to the hypothesis that the proper search direction is recovered by a $(2\sqrt{\delta}+1)$-test, the $(2\sqrt{\delta}+1)$-left-test over ℓ cannot be misleading, and it must have failed detecting the error. This implies that the incorrect search wasted only $O(\sqrt{\delta}\,)$ time, which can be charged to the faulty value $I[\ell]$. Since $I[\ell]$ is out of the interval on which the search proceeds (see Figure 4b), each faulty value can be charged at most once due to this kind of error, and we will have at most α incorrect searches of this kind. The total running time for these mislead searches will thus be $O(\alpha\sqrt{\delta}\,)$.

We next analyze the running time for a mislead search when the $(2\sqrt{\delta}+1)$-tests are not able to recover the proper search direction, i.e., when during the search we encounter a misleading $(2\sqrt{\delta}+1)$-test. In this case, the error will be detected at most δ steps later, when the next $(2\delta+1)$-tests are performed. Using arguments similar to the previous case, we will now show that there must exist $\Theta(\sqrt{\delta}\,)$ faulty values that are eliminated from the interval in which the search proceeds. We only consider the case where the algorithm encounters a misleading $(2\sqrt{\delta}+1)$-left-test over a left boundary p (the case of a misleading $(2\sqrt{\delta}+1)$-right-test is symmetric). Assume without loss of generality that the test guides the search towards positions larger than p, and consider the time when the next $(2\delta+1)$-left-test is performed: let ℓ be the left boundary involved in the test (it must be $p \le \ell$). As far as the algorithm works, the $(2\delta+1)$-left-test over position ℓ is performed just after a $(2\sqrt{\delta}+1)$-test. Since the $(2\sqrt{\delta}+1)$-left-test over p was misleading, and since ℓ is still a left boundary immediately before the $(2\delta+1)$-left-test, also the $(2\sqrt{\delta}+1)$-test over ℓ must have been misleading. Instead, the $(2\delta+1)$-left-test over ℓ cannot guide the search towards a wrong direction, and thus it must fail, detecting the error. We can charge the $O(\delta)$ time spent for the incorrect search to the $\Theta(\sqrt{\delta})$ faulty values in $I[\ell-(2\sqrt{\delta}+1)\,;\,\ell-1]$, that are out of the interval on which the search proceeds. We will therefore have at most $O(\alpha/\sqrt{\delta}\,)$ incorrect searches of this kind, requiring $O(\delta)$ time each, and the total running time will be again $O(\alpha\sqrt{\delta}\,)$.

By a simple amortization, it is not difficult to see that the time for the correct searches is $O(\log n)$, from which the claimed bound of $O(\log n+\alpha\sqrt{\delta}\,)$ follows. □

The bound in Theorem 6 yields a deterministic resilient searching algorithm that can tolerate up to $O((\log n)^{2/3})$ memory faults in $O(\log n)$ worst-case time. This improves over the algorithm described in [18], that can tolerate only $O((\log n)^{1/2})$ faults, but does not match yet the lower bound $\Omega(\log n+\delta)$ [18]. As we will see in the remainder of this section, we can further reduce the overhead due to coping with memory faults, getting arbitrarily close to the lower bound.

A faster algorithm. The running time can be reduced to $O(\log n+\alpha\,\delta^{\epsilon})$, for any constant $\epsilon\in(0,1]$, by refining the algorithm that we have described above. In particular, we exploit the use of $(2\delta^{i\,\epsilon}+1)$-tests, with $i=1,2,\ldots,(1/\epsilon)$, performed every $\delta^{i\,\epsilon}$ steps. As an example, for $\epsilon=1$ we obtain the algorithm presented in [18], for $\epsilon=1/2$ we obtain the previous algorithm, while for $\epsilon=1/3$ the algorithm will perform $(2\delta^{1/3}+1)$-tests every $\delta^{1/3}$ steps, $(2\delta^{2/3}+1)$-tests every $\delta^{2/3}$ steps, and $(2\delta+1)$-tests every δ steps. Note that the tests may be misleading for each $i<1/\epsilon$. However, the degree of "unreliability" decreases as i becomes larger.

Besides the indexes ℓ and r of the current interval, the refined algorithm stores in safe memory $(1/\epsilon)$ additional pairs of indexes witnessed by the different tests. In total we have a constant number of indexes when ϵ is a constant, and we can store them in safe memory. We will call the indexes in each pair ℓ_i and r_i, for $i=1,2,\ldots,(1/\epsilon)$. The indexes in the i-th pair are such that, at the time when the last $(2\delta^{i\,\epsilon}+1)$-tests were performed, they suggested that x should belong to $I[\ell_i;r_i]$. Every $\delta^{i\,\epsilon}$ steps the algorithm performs $(2\delta^{i\,\epsilon}+1)$-tests over the boundaries of the current interval and appropriately updates the indexes ℓ_j and r_j, for every $j\in[1,i]$, according to the outcome of the tests. This guarantees that, at any time during the execution of the algorithm, the indexes associated to less reliable tests never disagree with the indexes associated to more reliable tests, i.e., $I[\ell_j;r_j]\subseteq I[\ell_i;r_i]$ for every $j\le i$. Similarly to the previous algorithm, if two left and right tests contradict each other, we perform more and more reliable tests over the same boundaries, until two non-contradicting tests are found. Furthermore, two $(2\delta+1)$-tests are performed when the algorithm is about to terminate with a negative answer.

We now generalize the analysis carried out in Theorem 6, proving the following:

Theorem 7 *The refined algorithm* `DetSearch` *performs resilient searching in* $O(\log n+\alpha\,\delta^{\epsilon})$ *worst-case time, for any constant* $\epsilon\in(0,1]$.

Proof. The correctness follows from the same argument discussed in the proof of Theorem 6. Consider now the running time. As in Theorem 6, we think of the search process as divided into δ-steps. We first bound the running time for the mislead searches taking place in δ-steps that terminate because δ standard searching steps have been performed: in these δ-steps, no contradicting tests are ever performed. Let $i\in[1,1/\epsilon]$ be the smallest integer such that the proper search direction is recovered after performing a $(2\delta^{i\,\epsilon}+1)$-test. Such an integer certainly exists, because $(2\delta+1)$-tests cannot be misleading: thus, in the worst case $i = 1/\epsilon$. We distinguish two cases according to the value of i (either equal to or larger than 1).

We first analyze the running time for a mislead search when $i = 1$. In this case, the search direction is recovered by a $(2\delta^\epsilon + 1)$-test after at most δ^ϵ search steps. Let ℓ be the boundary involved in the $(2\delta^\epsilon + 1)$-test that recovers the search direction. As in Theorem 6, we can argue that ℓ is a misleading boundary and charge the $O(\delta^\epsilon)$ time spent for the mislead search to $I[\ell]$. Since $I[\ell]$ is out of the interval on which the search proceeds, each faulty value can be charged at most once due to this kind of error, and we will have at most α incorrect searches of this kind. The total running time for these mislead searches will thus be $O(\alpha\,\delta^\epsilon)$.

We next analyze the running time for mislead searches when $i > 1$. In this case, the time wasted due to a mislead search is $O(\delta^{i\,\epsilon})$, that is the time between consecutive $(2\delta^{i\,\epsilon}+1)$-tests. Using arguments similar to the previous case, we can show that there must exist $\Theta(\delta^{(i-1)\,\epsilon})$ faulty values that are eliminated from the interval in which the search proceeds. Indeed, as far as the algorithm works, the $(2\delta^{i\,\epsilon}+1)$-test that recovers the search direction is performed immediately after a $(2\delta^{(i-1)\,\epsilon}+1)$-test, that must be misleading by definition of i. Thus, at least $(\delta^{(i-1)\,\epsilon}+1)$ values considered in that test must be faulty. We can charge the $O(\delta^{i\,\epsilon})$ time spent for the mislead search to these values, all of which are out of the interval on which the search proceeds. We will therefore have at most $O(\alpha/\delta^{(i-1)\,\epsilon})$ incorrect searches of this kind, requiring $O(\delta^{i\,\epsilon})$ time each. The total running time will be again $O(\alpha\,\delta^\epsilon)$.

A similar analysis can be carried out to prove that δ-steps that terminate due to contradicting tests also give an $O(\alpha\,\delta^\epsilon)$ contribution to the running time. The claimed bound of $O(\log n + \alpha\,\delta^\epsilon)$ follows by noticing that the time for the correct searches is $O(\log n)$. $\square$

The bound in Theorem 7 yields a deterministic resilient searching algorithm that can tolerate up to $O((\log n)^{1-\epsilon'})$ memory faults, for any small positive constant ϵ', in $O(\log n)$ worst-case time, thus getting arbitrarily close to the lower bound.

5 Conclusions and Open Problems

In this paper we have presented sorting and searching algorithms resilient to memory faults that may happen during their execution. We have designed a comparison-based resilient sorting algorithm that takes $O(n\log n)$ worst-case time and can tolerate up to $O(\sqrt{n\log n}\,)$ faults: as proved in [18], this bound is optimal. In the special case of integer sorting, we have presented a randomized algorithm able to tolerate up to $O(\sqrt{n}\,)$ memory faults in expected $O(n)$ time. A thorough experimental study [15] has shown that the algorithms presented here are not only theoretically efficient, but also fast in practice.

With respect to resilient searching, 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.

After this work, we have 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]. Later, Brodal *et al.* have proposed an optimal deterministic static dictionary supporting searches in $\Theta(\log n + \delta)$ worst-case time (thus improving on the bounds given in Section 4.3), and have shown how to use it in a dynamic setting in order to support searches in $\Theta(\log n + \delta)$ worst-case time, updates in $O(\log n + \delta)$ amortized time and range queries in

$O(\log n + \delta + k)$ worst-case time, where k is the size of the output [10].

Possible directions for future research include proving lower bounds for the resilient integer sorting problem and for the amortized time required by insertions and deletions in a resilient binary search tree, as well as improving the upper bound for deterministic integer sorting. Investigating whether it is possible to obtain resilient algorithms that do not assume any knowledge on the maximum number δ of memory faults also deserve additional investigation. Finally, we remark that in this paper we focused on a faulty variant of the standard RAM model: an interesting research direction is designing resilient algorithms for more complex memory hierarchies.

Acknowledgments. We are indebted to the anonymous referees for many useful comments.

References

[1] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. *Proc. 2nd Usenix Workshop on Electronic Commerce*, 1–11, 1996.

[2] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. *Proc. International Workshop on Security Protocols*, 125–136, 1997.

[3] 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.

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

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

[6] 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.

[7] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. *Proc. EUROCRYPT*, 37–51, 1997.

[8] 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.

[9] R. Boyer and S. Moore. MJRTY - A fast majority vote algorithm. University of Texas Tech. Report, 1982. *http://www.utexas.edu/users/boyer/mjrty.ps.Z*

[10] G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jørgensen, G. Moruz and T. Mølhave. Optimal resilient dynamic dictionaries. *Proc. 15th Annual European Symp. on Algorithms* (ESA'07), LNCS 4698, 347–358, 2007.

[11] 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.

[12] 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.

[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. Designing Reliable Algorithms in Unreliable Memories. *Proc. 13th Annual European Symposium on Algorithms* (ESA 2005), 1–8, 2005. INVITED LECTURE.

[17] I. Finocchi, F. Grandoni and G. F. Italiano. Resilient Search Trees. *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms* (SODA'07), 547–555, 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. To appear in *Algorithmica*.

[19] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. *Proc. IEEE Symposium on Security and Privacy*, 154–165, 2003.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

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

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

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

[32] S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Proc. 4th Int. Workshop on Cryptographic Hardware and Embedded Systems*, LNCS 2523, 2–12, 2002.

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

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

[35] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. *Proc. International Conference on Dependable Systems and Networks*, 421–430, 2001.

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