Sorting and Searching in the Presence of Memory Faults (without Redundancy)*

Irene Finocchi
Dip. di Informatica, Sistemi e Produzione
Università degli Studi di Roma "Tor Vergata"
Via del Politecnico 1, 00133 Roma, Italy
finocchi@disp.uniroma2.it

Giuseppe F. Italiano
Dip. di Informatica, Sistemi e Produzione
Università degli Studi di Roma "Tor Vergata"
Via del Politecnico 1, 00133 Roma, Italy
italiano@disp.uniroma2.it

ABSTRACT

We investigate the design of algorithms resilient to memory faults, i.e., algorithms that, despite the corruption of some memory values during their execution, are able to produce a correct output 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 at most $O((n \log n)^{1/2})$ memory faults. Furthermore, we present one comparison-based sorting algorithm with optimal space and running time that is resilient to $O((n \log n)^{1/3})$ faults. We also prove polylogarithmic lower and upper bounds on fault-tolerant searching.

Categories and Subject Descriptors

F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*sorting and searching*; E.1 [**Data**]: Data structures.

General Terms

Algorithms, reliability, theory.

Keywords

Combinatorial algorithms, sorting, searching, memory faults, memory models.

1. INTRODUCTION

Some of today's applications run on computer platforms with large and inexpensive memories, which are also error-prone [28]. Unfortunately, the correctness of the underlying algorithms may be compromised by the appearance of 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 the output can contain as many as $\Theta(n^2)$ inversions, where n is the length of the input sequence. 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. Computing in the presence of memory faults seems important in many practical scenarios: for instance, sorting data in faulty memories is a common problem in search engines [12]. In this paper we investigate the design of algorithms resilient to memory faults, and present upper and lower bounds for two basic problems: sorting and searching.

1.1 Faulty-memory model

We have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a soft failure [28]. 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 [14]. 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 $O(1)$ *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.

1.2 Related work

The problem of computing with unreliable information

*Work 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 the Italian Ministry of University and Scientific Research (Project "ALINWEB: Algorithmics for Internet and the Web").

STOC'04, June 13–15, 2004, Chicago, Illinois, USA.

has been investigated in a variety of settings. In the *liar model*, started by Ulam–Rényi's game [26, 27], an algorithm asks comparison questions answered by a possibly lying adversary. Many works address the problem of searching with lies [2, 5, 11, 13, 16, 21, 22, 23]. Even when the number of lies grows proportionally with the number of questions (linearly bounded model), searching is easy: Borgstrom and Kosaraju [5], improving over [2, 11, 22], design an $O(\log n)$ searching algorithm. Problems such as sorting and selection have instead drastically different bounds. Lakshmanan *et al.* [17] prove 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 in [25]. In the linearly bounded model, an exponential number of questions is necessary even to test whether a list is sorted [5]. Feige *et al.* study a probabilistic model and present a sorting algorithm correct with probability $\geq 1 - q$ that requires $\Theta(n \log(n/q))$ comparisons [13]. 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, 18, 19, 29]. In this model, comparators may be faulty and 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 [18]. We note that the Assaf-Upfal network makes $\Theta(\log n)$ copies of each item, using fault-free data replicators. Using *data replication* is indeed a natural approach to protect against destructive memory faults: e.g., Aumann and Bender [4] use pointer redundancy to make pointer-based data structures resilient to memory faults. Redundant arrays of inexpensive disks (RAID) are another typical application of data replication [6].

Parallel models of computation with faulty memories have also been studied, e.g., in [7, 8, 9, 15]. 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 [7, 8, 9, 15] 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.

1.3 Our results

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's overhead in terms of both space and running time would also be $\Theta(k)$. This would imply, 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.

In general, using data replication can be very inefficient when the objects to be sorted 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 do it. For instance, common libraries of algorithmic codes (e.g., the LEDA Library of Efficient Data Types and Algorithms [20] or the Standard Template Library [24]) 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 the sorting algorithm assumes neither the existence of *ad hoc* functions for data replication nor the definition of suitable encoding mechanisms to maintain a reasonable storage cost.

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?

We affirmatively answer this question. 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.

Fault-tolerant sorting. We are given a set of n keys that need to be sorted. The value of at most δ keys can be arbitrarily corrupted (either increased or decreased) during the sorting process. A sorting algorithm is *fault-tolerant* if it correctly orders the set of uncorrupted keys. We remark that this is the best 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.

Fault-tolerant searching. We are given a set of n keys on which we wish to perform membership queries. The keys are in increasing order, but up to δ keys may be corrupted and thus occupy wrong positions in the sequence. Let s be a key to be searched for. A *fault-tolerant* searching algorithm works as follows: (1) if there is a correct key equal to s, it answers yes returning the index of s; (2) if there is no key (either correct or faulty) equal to s, it answers no. Note that, if there is a faulty key equal to s, the answer may be either yes or no.

One of the main difficulties in designing efficient fault-tolerant sorting and searching algorithms derives from the fact that positional information is not necessarily reliable in the presence of memory faults: for instance, in searching it may be possible that keys to the left (right) of a faulty key x are larger (smaller) than x.

We say that a sorting or searching algorithm is $[f(n)]$-*resilient* if it can tolerate at most $O(f(n))$ memory faults. The main results of this paper can be summarized as follows.

- We prove that any $O(n \log n)$ comparison-based sorting algorithm can be at most $[(n \log n)^{1/2}]$-resilient to memory faults.

- We present an $O(n \log n)$ comparison-based sorting algorithm that is $[(n \log n)^{1/3}]$-resilient to memory faults.

Both our lower and upper bounds are in fact more general, as we will show in Sections 3 and 4. We also prove polylogarithmic lower and upper bounds on fault-tolerant searching (Section 5). 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. For the sake of simplicity, however, in the course of the analysis we will sometimes refer to their recursive counterpart. Due to lack of space, many details are omitted from this extended abstract.

2. PRELIMINARIES

Given a sequence S, we denote by $S[i; j]$ the subsequence $\{S[i], S[i+1], \ldots, S[j]\}$. The following definitions will be useful throughout.

DEFINITION 1. *A sequence is* faithfully ordered *if its uncorrupted keys are sorted.*

DEFINITION 2. *A sequence is* k-unordered *if k is the minimum number of keys whose removal makes the remaining subsequence sorted.*

Note that each faithfully ordered sequence is k-unordered for some $k \leq \delta$, where $\delta \leq n$ is the upper bound on the number of memory faults.

DEFINITION 3. *A sorting or merging algorithm is* strongly fault tolerant *if it produces a faithfully ordered sequence, i.e., it correctly sorts all of the uncorrupted keys.*

DEFINITION 4. *A sorting or merging algorithm is* k-weakly fault tolerant *if it produces a k-unordered sequence, i.e., if it correctly sorts all but k keys.*

Note that a strongly fault tolerant algorithm is δ-weakly fault tolerant. On the contrary, a δ-weakly fault tolerant algorithm is not necessarily strongly fault tolerant, as uncorrupted keys may not be sorted in the output sequence.

2.1 Naive fault-tolerant sorting and searching

A fault-tolerant algorithm that sorts all the correct keys in $O(\delta \cdot n \log n)$ worst-case time can be easily obtained from mergesort. At each merge step, instead of taking the minimum among two keys, we take the minimum among $2\delta + 2$ keys, $\delta + 1$ per sequence; since there can be at most δ errors, at least one correct key per sequence is considered. An implementation of this approach would require time $O(\delta \cdot n)$ to merge sequences of length n. Furthermore, in order to avoid problems in the recursion stack, we use the standard iterative bottom-up implementation of mergesort, sorting all the sequences of length 2^i before any sequence of length 2^{i+1}, for $i = 1$ up to $\lceil \log n \rceil$. 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-Mergesort` this iterative sorting algorithm that uses the fault-tolerant merging described above as a subroutine. The running time of `Naive-Mergesort` is $O(\delta \cdot n \log n)$ in the worst case, and it becomes $O(\delta \cdot n)$ when $\delta = \Omega(n^{\epsilon})$, for some $\epsilon > 0$.

Similarly to sorting, a single fault-tolerant search step can be easily implemented in $O(\delta)$ time by using a majority argument, i.e., by following the search direction suggested by the majority of $2\delta + 1$ consecutive keys. We will refer to this subroutine as `FT-SearchStep`. Performing $O(\log n)$ fault-tolerant search steps yields a fault-tolerant algorithm that answers membership queries in $O(\delta \log n)$ time.

2.2 Purifying k-unordered sequences

In this section we describe a fault-tolerant algorithm, that we call `Purify`, for extracting a faithfully ordered subsequence from a k-unordered sequence X of length n. The algorithm hinges upon the Cook-Kim division [10].

Algorithm Purify. While we scan sequence X from left to right, we build a stack and a list of discarded keys as follows. We maintain the top of the stack and the index i that scans X in the $O(1)$-size reliable memory. At the i-th step, if $X[i]$ is larger than or equal to the top, we push it onto the stack. Otherwise, we add both the top and $X[i]$ to the list of discarded keys, pop the stack, compute the maximum of the topmost $\delta + 1$ keys, and move it to the top. The reason for computing the maximum is that keys below the discarded top may have been corrupted since they were pushed onto the stack: the possible implications on the correctness of the algorithm will become apparent shortly. Finally, we return the stack and the list of discarded keys.

Analysis. We first show that algorithm `Purify` maintains the following stack invariant:

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

PROOF. We remark that the top of the stack is fault-free, because it is stored in reliable memory. The proof proceeds by induction on the value of index i. The base step, with the stack containing just one element, trivially holds. Assume that the invariant holds at the beginning of the i-th step. If $X[i] \geq top$, $X[i]$ is pushed onto the stack and the invariant remains satisfied by transitivity. If $X[i] < top$, the stack is popped and the invariant may be no longer satisfied if the key below the discarded top got corrupted (namely, if its value was decreased). In this case, let m be the maximum of the topmost $\delta + 1$ keys: we choose m as the new top. Note that at least one of the $\delta + 1$ considered keys is correct: let x be any such correct key. At the time when x was at the top, the invariant was true by inductive hypothesis, and therefore x is still 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. □

LEMMA 1. *Algorithm* `Purify` *computes a faithfully ordered 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 faithfully ordered 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 t is the number of pop operations, it is easy to see that the running time of `Purify` is $O(n+t\cdot\delta)$ and that the final stack has size at least $n-2t$. To prove that $t \leq k+\alpha$, we observe that pop operations correspond to inversions in X and that elements in the inversions are all distinct. Thus, we have t disjoint inverted pairs of keys: to get a sorted subsequence, at least one key from each pair must be eliminated. Assuming $t > k+\alpha$ would therefore contradict either the hypothesis that X is k-unordered or the hypothesis that at most α keys are corrupted during the execution of `Purify`. □

3. STRONGLY FAULT-TOLERANT SORTING

In this section we describe a δ-resilient strongly fault-tolerant sorting algorithm with $O(n\log n+\alpha\cdot\delta^2)$ worst-case running time, where α is the actual number of keys corrupted during the sorting process. In particular, this yields an $O(n\log n)$ time algorithm that is $[(n\log n)^{1/3}]$-resilient to memory faults. The algorithm is based on mergesort, and we first present two fault-tolerant merging subroutines with quite different characteristics: the first one requires time linear in the length of the sequence, but it may be unable to sort faithfully all the correct keys; the second one is slower, yet strongly fault-tolerant. We will use both subroutines as building blocks for our strongly fault-tolerant sorting algorithm.

3.1 Weakly fault-tolerant merge

Let A and B be the sequences to be merged. We assume that A and B are faithfully ordered 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 fault-tolerant merging, called `WFT-Merge`, resembles the classical merging algorithm with the following modifications.

Algorithm WFT-Merge. Let i and j be the indices to arrays 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 and $wait_B$ is incremented by 1; otherwise, if $B[j]$ is added to the output sequence, $wait_B$ is reset and $wait_A$ is incremented. We let the *wait* variables increase up to value $2\delta+1$. If this value is reached by either of them (without loss of generality, say $wait_A$), we reset it and we count the number t of values in the window $W = A[i+1; i+2\delta+1]$ that are smaller than $A[i]$: if $t \geq \delta+1$, we output $A[i]$ and increment index i by 1. The situation when W contains less than $2\delta+1$ values (i.e., we are almost at the end of sequence A) can be easily handled: we omit the details in this paper. The case $wait_B = 2\delta+1$ is symmetric. Indices, *wait* variables, and counter t are all stored in the $O(1)$-size reliable memory.

Analysis. At any time one of the two *wait* variables is zero. The merging process is thus divided into a sequence of *A-bursts* and *B-bursts*: during an A-burst, $wait_A = 0$ while i and $wait_B$ are incremented (B-bursts are symmetric). The bursts have length $\leq 2\delta+1$, and this is crucial in proving that the algorithm is $O(\alpha\cdot\delta)$-weakly fault-tolerant. 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 $k = O(f(\delta))$. Similarly, we will refer to $O(f(\delta))$-weakly fault-tolerant algorithms.

LEMMA 2. *Given two faithfully ordered sequences of total length n, algorithm* `WFT-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 over 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 over corrupted keys. We analyze B-bursts only (the analysis of A-bursts is symmetric). Consider the end of a B-burst, i.e., the time when $wait_A$ is reset. 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 has a value larger than the correct one. If this is the case, there are two possibilities:

- $wait_A < 2\delta+1$. In this case $A[i] < B[j]$. The keys of B that have been added to the output sequence in the last $wait_A$ steps may appear in the wrong position in the output sequence: those errors can be charged over the corrupted value $A[i]$.

- $wait_A = 2\delta+1$. Recall that t is the number of values in the window $W = A[i+1; i+2\delta+1]$ that are smaller than $A[i]$.

  If $t \geq \delta+1$, at least one of the keys of W that are smaller than $A[i]$ must be correct, and the algorithm can deduce that $A[i]$ is corrupted. $A[i]$ is therefore output immediately – in spite of the fact that it may be in a wrong position – so as not to introduce further errors. As in the previous case, the keys from sequence B (at most $2\delta+1$) that will eventually be in a wrong position can be charged over the corrupted value $A[i]$.

  If $t \leq \delta$, the algorithm cannot decide whether $A[i]$ is corrupted or not; however, there must be at least one element of W, say x, that is larger than $A[i]$ and it is correct. Since $A[i] \leq x$, by transitivity the keys of B that have been output during the burst are all 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 $\geq x$. 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 in that case we can charge these possible errors over $A[i]$ itself.

This shows that each corrupted key is charged at most $2\delta+1$ errors, concluding the proof. □

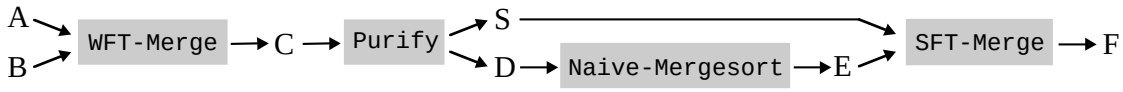

Figure 1: The merging algorithm.

3.2 Strongly fault-tolerant merge

Let A and B be two faithfully ordered 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 fault-tolerant merging, called `SFT-Merge`, repeatedly extracts a key from the shorter sequence B and places it in the correct position w.r.t. the longer sequence A.

Algorithm SFT-Merge. Let i and j be pointers to arrays A and B, respectively. At each step, we extract the minimum in $B[j; j+\delta]$: let $b = B[h]$ be such minimum, for $j \leq h \leq j+\delta$. We shift right all the keys in $B[j; h-1]$, move b to $B[j]$, and increase index j by 1. We scan A left to right, 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 $W = A[i+1; i+2\delta+1]$. If $t \geq \delta+1$, we continue scanning A ($A[i]$ is certainly corrupted). 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 3. *Let A and B be two faithfully ordered sequences of length n_1 and n_2, respectively, with $n_2 \leq n_1$. Algorithm* `SFT-Merge` *faithfully 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.*

PROOF. We first prove that the output sequence is faithfully ordered. 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 fault-tolerant: when b is added to the output sequence, at least $\delta+1$ elements from W are larger than $A[i]$, and thus at least one of them must be correct. Keys smaller than b are returned before b itself, and the fact that their relative order is maintained guarantees that no error is introduced. We now discuss the running time. Extracting b, analyzing W, and rearranging it, 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 over 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 Sorting in the presence of memory faults

In this section we present and analyze a strongly fault-tolerant mergesort-based sorting algorithm, that we call `SFT-Mergesort`. As in Section 2.1, we assume an iterative bottom-up implementation in order to avoid problems due to a non fault-tolerant recursion stack. The merging algorithm used by `SFT-Mergesort` is described in Figure 1.

The input sequences, A and B, are first merged using the linear-time subroutine `WFT-Merge`. The output sequence, C, may not be faithfully ordered, i.e., some correct elements may be in a wrong position. Such errors are recovered by the combined use of `Purify` and `SFT-Merge`. Note that the crux of this merging algorithm is to use the slower strongly fault-tolerant subroutine `SFT-Merge` on two unbalanced sequences, the shorter of which has length proportional to the actual number of corrupted keys (as we will see later). It is easy to see that the entire algorithm is strongly fault-tolerant.

Analysis. Let ℓ be the total length of the faithfully ordered 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 $\alpha = in + thru$. By Lemma 2, `WFT-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 `SFT-Merge` are both $O(\ell + \alpha \cdot \delta^2)$ by Lemmas 1 and 3, respectively. Since $|D| = O(\alpha \cdot \delta)$, then $\delta = \Omega(|D|^{1/2})$ and `Naive-Mergesort` on the set D requires time $O(|D| \cdot \delta) = O(\alpha \cdot \delta^2)$ (see Section 2.1). Thus we can conclude:

LEMMA 4. *The merge step takes time $O(\ell + \alpha \cdot \delta^2)$ to merge two sequences of length ℓ containing α corrupted keys.*

We remark that the sequence S returned by `Purify` is faithfully ordered; more precisely, 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-Mergesort`. Hence, the propagation of errors satisfies the following property:

LEMMA 5. *The merge step returns a faithfully ordered sequence in which only the keys corrupted while merging may be unordered.*

THEOREM 1. *Algorithm* `SFT-Mergesort` *faithfully 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.*

PROOF. Without loss of generality assume that n is a power of 2. Consider the recursion tree of `SFT-Mergesort`. Let x be any node of this tree at level k, for $0 \leq k < \log n$. Let $\text{in}_k(x)$ be the number of errors contained in the sequences to be merged at node x, and let $\text{thru}_k(x)$ be the number of errors introduced while merging at node x. By Lemma 4, the time required for merging at node x satisfies the following equation:

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

By Lemma 5, $\text{in}_k(x) = \text{thru}_{k+1}(x_1) + \text{thru}_{k+1}(x_2)$, where x_1 and x_2 are the children of node x in the recursion tree.

Since α is the total number of memory faults, we have $\sum_{k=0}^{\log n} \sum_{x=1}^{2^k} \text{thru}_k(x) \leq \alpha$. Thus, if we sum up the fault dependent contributions $(\text{in}_k(x) + \text{thru}_k(x))\delta^2$ on the entire recursion tree, we obtain $O(\alpha \cdot \delta^2)$. With standard techniques we can then conclude that the running time of algorithm `SFT-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 `SFT-Mergesort` can actually work without knowledge of a tight upper bound on the number of faults: choosing $\delta = O((n \log n)^{1/3})$ yields an $O(n \log n)$ strongly fault-tolerant sorting algorithm that is $[(n \log n)^{1/3}]$-resilient to memory faults. Moreover, by Lemma 5, if no key is corrupted during the last merge step (i.e., at the root of the recursion tree), both correct and corrupted keys will be ordered in the output sequence.

4. LOWER BOUND ON FAULT-TOLERANT SORTING

In this section we prove lower bounds on strongly fault-tolerant 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 "$x \leq y$?" or "$x > y$?". We first show that $\Omega(n + \delta^{2-\epsilon})$ comparisons are necessary to merge two faithfully ordered 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, 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 algorithm and adversary know the fault upper bound δ.

Adversary's strategy. Let A and B be two faithfully ordered 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, $A_i = A[(i-1)n/\delta;\, i(n/\delta) - 1]$, for $1 \leq i \leq \delta$. B_i is defined similarly. The adversary answers the algorithm's questions as if the sorted sequence were:

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

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 $(\delta^{2-\epsilon}/2)$ 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 $\delta/2$ edges. We first prove two lemmas about sparse sequences.

LEMMA 6. *If the algorithm performs less than $\delta^{2-\epsilon}/2$ comparisons, for some ϵ, $0 \leq \epsilon \leq 1/2$, then there exists at least one sparse sequence of length $2\delta^\epsilon$.*

PROOF. There are $2\delta/(2\delta^\epsilon) = \delta^{1-\epsilon}$ disjoint subsequences of $2\delta^\epsilon$ consecutive clusters. If none of them is sparse, the number of comparisons should be $\geq \delta^{1-\epsilon}(\delta/2) = \delta^{2-\epsilon}/2$, which is a contradiction. □

LEMMA 7. *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 $2\delta^\epsilon$. 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 with each element of B_j, for each A_i and B_j in Y. Recall that $|A_i| = |B_j| = n/\delta$ and that Y consists of δ^ϵ clusters B_j and δ^ϵ clusters A_i. Thus, each A_i must have degree $\geq \delta^\epsilon (n/\delta)^2$ and the total number of edges in the subgraph induced by Y must be $\geq n^2\delta^{2\epsilon-2}$. 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 $n^2\delta^{2\epsilon-2} < \delta/2$, which is clearly a contradiction. □

LEMMA 8. *Given any ϵ, $0 \leq \epsilon \leq 1/2$, any strongly fault-tolerant merging algorithm requires in the worst-case $\Omega(n + \delta^{2-\epsilon})$ comparisons to merge two faithfully ordered 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 if $\delta \leq n^{1/(2-\epsilon)}$ the lower bound trivially holds. Let $\delta > n^{1/(2-\epsilon)}$ and assume that the algorithm has asked less than $\delta^{2-\epsilon}/2$ comparison questions. By Lemma 6, there must be a sparse sequence, say $Y = A_k B_k A_{k+1} \ldots A_{k+\delta^\epsilon-1} B_{k+\delta^\epsilon-1}$, of length $2\delta^\epsilon$. By Lemma 7, let $a \in A_i$ and $b \in B_j$, for some $k \leq i, j < k + \delta^\epsilon$, be two elements of Y that have not been directly compared by the algorithm. We analyze only the case when $i \leq j$ (the analysis when $i > j$ is similar). We will show that both the order in which a precedes b and the order in which b precedes a are consistent with the adversary's answers. The former case is easy, since it holds when no value has been corrupted. In the latter case, consider the sequence of comparisons asked by the algorithm. Let "$x < y$?" be any such question: as far as a and b have been chosen, neither "$a < b$?" nor "$b < a$?" was asked. The adversary claims that:

1. If either $x \notin Y$ or $y \notin Y$, the question has been consistently answered without introducing memory faults: the sequences A and B can be stretched so that, for any $t < k$, elements in A_t and B_t are smaller than elements in Y. Similarly for $t \geq k + \delta^\epsilon$.
2. If $x = a, b$ and $y \in Y$, then y is corrupted (note that $y \neq b, a$).
3. If $y = a, b$ and $x \in Y$, then x is corrupted (note that $x \neq b, a$).
4. If $x \in Y$, $y \in Y$, and none of them coincides with a or b, both x and y are corrupted.

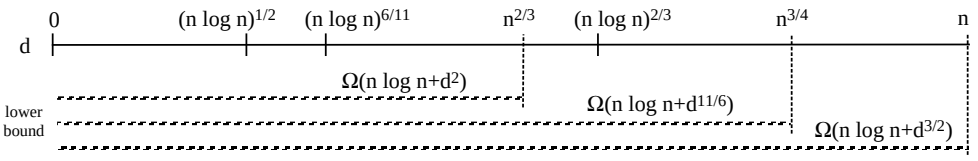

Figure 2: Intervals used in the proof of Theorem 3.

Note that, if the comparison 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, the number of faults is at most δ. Let $Y' \subseteq Y$ be the subset of uncorrupted elements: $Y' \neq \emptyset$, as it contains at least a and b. The adversary claims that the elements in $Y' \cap B$ appear before the elements in $Y' \cap A$ in the correct output sequence: these elements have not been compared against each other – otherwise they would be considered as corrupted – and the comparisons with elements outside Y cannot be used to get information regarding their relative order by transitivity. Thus, the adversary can prove that she always answered consistently all of the questions and the algorithm cannot decide which of the two orders ($a < b$ or $b < a$) is correct. We remark that neither a nor b is faulty, and thus both orders are faithful. □

THEOREM 2. *Given any ϵ, $0 \leq \epsilon \leq 1/2$, any strongly fault-tolerant 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 8. 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 8. □

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

THEOREM 3. *Any $O(n \log n)$ comparison-based strongly fault-tolerant sorting algorithm can be at most $[(n \log n)^{1/2}]$-resilient to memory faults.*

PROOF. By Theorem 2, $\Omega(n \log n + \delta^{2-\epsilon})$ is a lower bound on strongly fault-tolerant sorting when $\delta \leq n^{2/(3-2\epsilon)}$, 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 be δ-resilient, with $\delta \in \Big((n \log n)^{2/3}, n\Big]$.
- $\epsilon = 1/6$: 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 be δ-resilient, with $\delta \in \Big((n \log n)^{6/11}, n^{3/4}\Big]$.
- $\epsilon = 0$: 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 be δ-resilient, with $\delta \in \Big((n \log n)^{1/2}, n^{2/3}\Big]$.

As shown in Figure 2:

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

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

Any optimal comparison-based sorting algorithm can thus be at most $[(n \log n)^{1/2}]$-resilient. □

5. FAULT-TOLERANT BINARY SEARCH

In this section we consider the fault-tolerant binary search problem. Let X be a faithfully ordered sequence of length n containing at most δ corrupted keys, and let s be a key to be searched for in X. We first describe a fault-tolerant searching algorithm with $O(\log n + \alpha \cdot \delta^2)$ query time, where α is the total number of faulty keys in X, and then we show how to improve the query time to $O(\log n + k \cdot \delta \cdot \log(\delta/k))$, where k is the actual number of faulty keys on the search path to s. This yields a $[(\log n)^{1/2}]$-resilient searching algorithm that requires $O(\log n)$ time in the worst case. We also prove that any comparison-based searching algorithm with optimal $O(\log n)$ time can be at most $[\log n]$-resilient to memory faults.

5.1 The query algorithm

In this section we describe a fault-tolerant searching algorithm, called `FT-BinSearch`, with $O(\log n + \alpha \cdot \delta^2)$ query time, where α is the total number of faulty keys in X. The algorithm proceeds in a non fault-tolerant way, checking from time to time if it did some error. If this is the case, it backtracks and corrects the search direction by scanning part of the search path using a (slow) fault-tolerant approach. We present a recursive implementation of `FT-BinSearch`: similarly to sorting, the recursion can be easily unrolled by storing a few array indices in $O(1)$ words of reliable memory. Let a and b denote the left and right indices of the array to be searched for at a generic step. The recursive call `FT-BinSearch`(X, a, b) works as follows.

Step 1. If $a = b$, return yes if and only if $s = X[a]$ and exit the algorithm. Otherwise, starting from $X[(a+b)/2]$, perform h steps of *non fault-tolerant* binary search. Let ℓ and r be the left and right indices of the interval I identified by this search.

Step 2. Let $\mathcal{N}_\ell$ and $\mathcal{N}_r$ be two $(2\delta+1)$-size neighborhoods of $X[\ell]$ and $X[r]$, respectively: i.e., $\mathcal{N}_\ell = X[\ell - 2\delta; \ell]$ and $\mathcal{N}_r = X[r; r + 2\delta]$. If $s \in \mathcal{N}_\ell \cup \mathcal{N}_r$, return yes and exit the algorithm.

Step 3. If $\mathcal{N}_\ell$ contains at least $\delta + 1$ keys larger than s or $\mathcal{N}_r$ contains at least $\delta + 1$ keys smaller than s, restarting from $X[(a+b)/2]$, perform at most h fault-tolerant binary search steps (using `FT-SearchStep` of

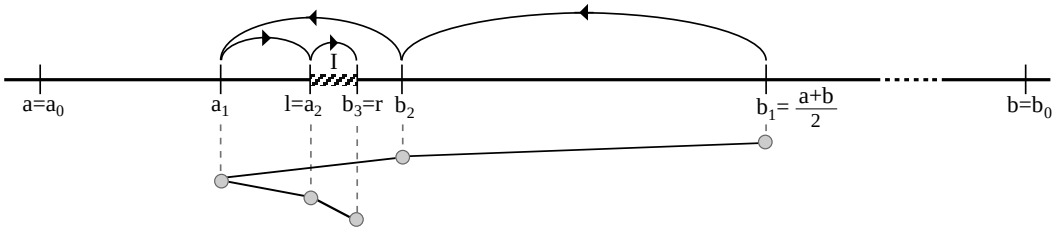

Figure 3: Fault-tolerant binary search with $h = 5$ and correspondence with a binary search tree: a_i's and b_i's are the left and right boundary nodes, respectively.

Section 2.1), until the first difference from the search of Step 1 is found (this can be deduced by comparisons with the values ℓ and r, which are stored in reliable memory). Update ℓ and r to delimit the new interval identified by this search.

Step 4. Return the result of `FT-BinSearch`(X, ℓ, r).

The value h at which we place checkpoints will be determined in the analysis. The boundary cases where $\ell < 2\delta$ or $r > n - 2\delta$ can be easily handled, e.g., by extending array X: before starting the algorithm, we add 2δ keys with value $-\infty$ to the left of $X[0]$ and 2δ keys with value $+\infty$ to the right of $X[n-1]$. At the first invocation, we call `FT-BinSearch`$(X, 2\delta, n + 2\delta - 1)$.

Analysis. Note that, if there were no faults, at the end of Step 1 the search should continue in interval $I = X[\ell\,; r]$. The test in Step 3 simulates two fault-tolerant search steps on $X[\ell]$ and $X[r]$, respectively, using a majority argument. It is easy to see that we have taken a wrong search direction at some point during Step 1 if and only if either the fault-tolerant search step at $X[\ell]$ would continue the search to the left or the fault-tolerant search step at $X[r]$ would continue the search to the right.

LEMMA 9. *Given a faithfully ordered sequence X of length n, algorithm* `FT-BinSearch` *correctly answers membership queries on X in $O(\log n + \alpha \cdot \delta^2)$ worst-case time, where $\alpha \leq \delta$ is the total number of corrupted keys at the end of the algorithm execution.*

PROOF. The algorithm answers yes only if it finds an index i such that the searched key $s = X[i]$. Thus, to show its correctness, we just need to prove that in case of a negative answer there exists no correct key equal to s. This may be false if the algorithm has taken at some point a wrong search direction. However, if the search in Step 1 is not correct, this is detected by the test in Step 3 and the search is then repeated in a fault-tolerant way. This guarantees that the values ℓ and r in the recursive call in Step 4 are always correct.

We now discuss the running time. It is easy to see that Steps 1, 2, and 3 require time $O(h)$, $O(\delta)$, and $O(h \cdot \delta)$, respectively. Each time we perform a fault-tolerant search in Step 3, we eliminate at least one error from the interval in which the search continues, and thus we can execute it at most α times. Counting the running time for Step 3 and for the uncorrect searches in Step 1 separately, we get $O(\alpha \cdot h \cdot \delta)$ total time. 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 $T(n) = T(n/2^h) + O(\delta + h) = O((1 + \delta/h) \log n)$. This yields a total of $O((1+\delta/h) \log n + \alpha \cdot h \cdot \delta)$ time. Choosing $h = \delta$, we get that `FT-BinSearch` answers membership queries in $O(\log n + \alpha \cdot \delta^2)$ worst-case time. □

5.2 Improving the query time

A careful implementation and analysis of Step 3 of algorithm `FT-BinSearch` improves the running time to $O(\log n + k \cdot \delta \cdot \log(\delta/k))$, where k is the actual number of faulty keys on the search path to s.

Note that the keys considered during the non fault-tolerant search of Step 1 naturally correspond to the nodes of a binary search tree (see Figure 3). Call a_i's and b_i's the nodes where we turn right and left during the search, respectively (also called left boundary and right boundary nodes). Clearly, $a_0 = a$ and $b_0 = b$. For brevity, we call a value *misleading* if it is faulty and guides the search to a wrong direction during Step 1. Let us assume that there exists a misleading left boundary node, say a_j (this is detected in Step 3 by the simulation of the fault-tolerant search step on $X[\ell]$). The search should then continue to the left of a_j, all a_k with $k \geq j$ are faulty, and all b_i are not misleading (they might be faulty, though). In order to find the first misleading a_j encountered in Step 1, which is the upmost in the binary search tree and leftmost in the array, we can therefore use a fault-tolerant binary search on the set of the a_i's. However, it is not possible to store the a_i's in reliable memory, as the required size would not be constant: instead, we can recompute the a_i's on the fly by simulating the $O(h)$-time search in Step 1. (We defer the details to the full paper.) The case of a misleading b_i is symmetric. This implementation of Step 3 costs $O((\delta + h) \log h) = O(\delta \log \delta)$ time, decreasing the total query time to $O(\log n + \alpha \cdot \delta \log \delta)$. We can further refine the algorithm and its analysis proving the following result:

THEOREM 4. *Given a faithfully ordered sequence X of length n containing at most δ faulty values, we can correctly search in X in $O(\log n + k\delta \log \frac{\delta}{k})$ worst-case time, where k is the number of faulty values on the correct search path.*

PROOF. Let a_j be the leftmost misleading left boundary node. Let t be the number of left boundary nodes to the right of a_j (all of them must be faulty). Starting from the rightmost left boundary node (i.e., ℓ) and using doubling until we find a non misleading a_i, we can identify a_j in time $O((\delta + h) \log t)$. Let k be the total number of faulty values on the correct search path to s: in the worst case all such values are misleading. On the i-th misleading value, we spend time $O((\delta + h) \log t_i) = O(\delta \log t_i)$ to eliminate t_i faulty keys, and exactly one of these keys is on the correct

search path to s. Summing up the running times of all the fault-tolerant binary searches (at most k), we get

$$\sum_{i=1}^{k} O(\delta \log t_i) = O\left(\delta \log\left(\prod_{i=1}^{k} t_i\right)\right) =$$

$$= O\left(\delta \log\left(\frac{1}{k} \cdot \sum_{i=1}^{k} t_i\right)^k\right) = O\left(k\delta \log \frac{\delta}{k}\right)$$

Hence, the total query time is $O(\log n + k\delta \log(\delta/k))$ when the search path contains k faulty values. □

We remark that $O(\log n + k\delta \log \frac{\delta}{k}) = O(\log n + \delta^2)$ in the worst case. Hence, choosing $\delta = O((\log n)^{1/2})$ yields a searching algorithm with optimal $O(\log n)$ query time that is $[(\log n)^{1/2}]$-resilient to memory faults.

5.3 Lower bound

A simple adversary-based argument shows that $\Omega(\log n + \delta)$ comparisons are necessary to answer membership queries on a faithfully ordered sequence of length n containing up to δ faulty keys. We limit to prove that every fault-tolerant search algorithm needs $\Omega(\delta)$ comparison questions. Combining this with the classical $\Omega(\log n)$ lower bound (absence of faults), gives the desired result. We will show an adversary's answering strategy that leaves at least two possible correct answers if the searching algorithm asks less than $(\delta/2)$ questions.

Adversary's strategy. Let X be a faithfully 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 $i < j$, and no otherwise.

Analysis. Assume that the searching algorithm asks less than $(\delta/2)$ comparisons. Let $\mathcal{C} \subseteq X$ be the set of elements involved in at least one question. Note that $|\mathcal{C}| < 2(\delta/2) = \delta$, because at most two elements per question are used. Since $\delta \leq n$, the subsequence $X \setminus \mathcal{C}$ is not empty, and the comparisons provide no information about it. Based on this, the algorithm cannot decide whether the answer 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 have been consistent with the sequence X. Thus, any fault-tolerant searching algorithm requires in the worst-case $\Omega(\log n + \delta)$ comparisons to search in a faithfully ordered sequence of length n when up to δ values may be corrupted. We therefore have the following theorem:

THEOREM 5. *Any comparison-based searching algorithm with optimal $O(\log n)$ query time can be at most $[\log n]$-resilient to memory faults.*

Acknowledgments. We wish to thank an anonymous reviewer for useful comments.

6. 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 network. *Proc. 31st IEEE Symp. on Foundations of Computer Science* (FOCS'90), 275–284, 1990.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] B. S. Chlebus, L. Gasieniec and A. Pelc. Fast deterministic simulation of computations on faulty parallel machines. *Proc. 3rd Annual European Symp. on Algorithms* (ESA'95), LNCS 979, 89–101, 1995.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] S. Muthukrishnan. On optimal strategies for searching in the presence of errors. *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms* (SODA'96), 680–689, 1996.

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

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

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

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

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

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

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

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