Resilient Quicksort and Selection

Maxim Babenko[1,2,⋆] and Ivan Pouzyrevsky[1,2,⋆⋆]

[1] Moscow State University, Russia
[2] Yandex, Russia

Abstract. We consider the problem of sorting a sequence of n keys in a RAM-like environment where memory faults are possible. An algorithm is said to be δ-*resilient* if it can tolerate up to δ memory faults during its execution. A resilient sorting algorithm must produce a sequence where every pair of uncorrupted keys is ordered correctly. Finocchi, Grandoni, and Italiano devised a δ-resilient deterministic mergesort algorithm that runs in $O(n \log n + \delta^2)$ time. We present a δ-resilient randomized algorithm (based on quicksort) that runs in $O(n \log n + \delta\sqrt{n \log n})$ expected time and its deterministic variation that runs in $O(n \log n + \delta\sqrt{n} \log n)$ worst-case time. This improves the previous known result for $\delta > \sqrt{n} \log n$.

Our deterministic sorting relies on the notion of an approximate k-th order statistic. For this auxiliary problem, we devise a deterministic algorithm that runs in $O(n + \delta\sqrt{n})$ time and produces a key (either corrupted or not) whose order rank differs from k by at most $O(\delta)$.

1 Introduction

1.1 Preliminaries

Recent trends in algorithm engineering indicate a rising demand for reliable computations. Applications that make use of large memory capacities at low cost face problems of memory faults [18,15]. Indeed, unpredictable failures known as *soft memory errors* tend to happen more often with the increase in memory size and speed [13,16]. Contemporary memory devices such as SRAM and DRAM units are unreliable due to a number of factors such as power failures, radiation, cosmic rays etc. The content of a cell in an unreliable memory can be silently altered and for standard memory circuits there is no direct way for detecting these types of corruptions.

Corrupted content in memory cells can affect many standard algorithms. For instance, during a typical binary search in a sorted array a single corruption encountered in an early stage of the search can cause the search path to end $\Omega(N)$ locations away from its correct position. Data replication can help to combat corruptions but is not always feasible since time and space overheads incurred by storing and fetching replicated values can be significant. Memory corruptions

⋆ maxim.babenko@gmail.com
⋆⋆ ivan.pouzyrevsky@gmail.com

E. Hirsch et al. (Eds.): CSR 2012, LNCS 7353, pp. 6–17, 2012.

are of particular concern for applications dealing with massive amounts of data since such applications typically run for very long periods of time and are thus more likely to encounter memory cells containing corrupted data.

To design an algorithm that is provably resilient to memory corruptions, Finocchi and Italiano [12] introduced the *faulty memory random access machine*, which is based on a traditional RAM model. In this faulty model, memory corruptions can occur at *any time* and at *any place* in memory during the execution of an algorithm, and corrupted memory cells cannot be distinguished from uncorrupted cells. It is assumed that there is an adaptive adversary that chooses how, where, and when corruptions occur. This model has a parameter δ that is an upper bound on the number of corruptions the adversary can perform during a single run of the algorithm. Motivated by the fact that processor registers are considered incorruptible, $O(1)$ reliable memory locations are provided.

This model also extends to randomized computations, where, as defined in [11], the adversary does not see the random bits used by an algorithm. An algorithm is said to be *resilient* if it works *correctly* despite memory faults. The notion of correctness is stated explicitly for each possible kind of problem. For instance, a correct resilient sorting algorithm must output all uncorrupted elements in sorted order (while corrupted elements can appear at arbitrary positions in the output).

Throughout the paper we use the notion of a *trivially resilient storage*, which keeps its values in unreliable memory but still guarantees safety. This is achieved by replicating each stored value in $2\delta + 1$ consecutive cells. Since at most δ of copies can be corrupted, the majority of these $2\delta + 1$ elements remain uncorrupted. The correct value can be retrieved in $O(\delta)$ time and $O(1)$ space with the majority algorithm given [1] (which scans the copies keeping a single majority candidate and a counter in a reliable memory).

The above notion of trivial resiliency is extended to algorithms. Each algorithm that operates in a usual, non-faulty RAM model can be run in a faulty environment and produce correct results. To this aim all its memory (containing the input, the output, and the intermediate data) is kept in a trivially resilient storage. This trick, however, comes at a cost — it multiplies the complexity by δ. We shall be interested in alternative approaches that do not incur the δ-factor overhead.

1.2 Previous Work

Several important results were achieved in the faulty RAM model [9]. Concerning resilient dynamic data structures, search trees that support searches, insertions and deletions in $O(\log n + \delta^2)$ amortized time were introduced in [10]. A resilient priority queue was proposed in [14] supporting both insert and delete-min operations in $O(\log n + \delta)$ amortized time. Furthermore, in [2] a resilient dynamic dictionary implementing search, insert and delete operations in $O(\log n + \delta)$ expected amortized time was developed. Recently, Brodal et al. in [4] addressed the counting problem in faulty memory proposing a number of algorithms with various tradeoffs and guarantees.

For the sorting problem, the following definition is crucial (cf.[12,11]).

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

Given a sequence S, a correct δ-resilient sorting algorithm must tolerate up to δ faults, i.e. produce a faithfully ordered sequence obtained by permuting the elements of S. (In presence of memory corruptions, getting a faithfully ordered sequence is the best we can hope for.) In [11] Finocchi and Italiano devised a deterministic δ-resilient algorithm that sorts n keys in $O(n \log n + \delta^2)$ time. They also proved in [12] that under certain additional assumptions a resilient comparison-based sorting algorithm must perform at least $\Omega(n \log n + \delta^{2-\varepsilon})$ comparisons to sort a sequence of length n when up to $\delta \leq n^{2/(3-2\varepsilon)}$ (for $\varepsilon \in [0, 1/2]$) corruptions may happen.

In [17,7,8] a number of empirical studies were conducted showing that resilient algorithms are of practical interest. The problem of combining external memory and resilient algorithms is considered in [3].

1.3 Our Contribution

Firstly, we address sorting problem. In contrast to a merge-based approach developed by Finocchi, Grandoni, and Italiano, we investigate how the divide-and-conquer strategy fits in the faulty memory model. In Section 2 we propose an $O(n \log n + \delta\sqrt{n \log n})$-time randomized δ-resilient algorithm that sorts n keys and its deterministic variation that runs in $O(n \log n + \delta\sqrt{n} \log n)$ worst-case time. For $\delta > \sqrt{n} \log n$, the method we propose is asymptotically faster than one in [11].

It may seem suspicious that our running time beats the lower bound from [12] but one should not worry since our algorithm employs explicit data replication and the analysis in [12] does not apply to this case. More details are given in Appendix. Based on this we stress that there is a possible gap between the actual complexity of the sorting problem in resilient setting and the bound proved in [12]. This also indicates that a careful usage of data replication may be a key to improving performance of resilient algorithms. In our case we were able to improve performance for large δ.

To devise deterministic variation of quicksort we had to deal with the selection problem. Hence in Section 3 we consider the problem of approximating the k-th order statistic in a faulty environment. That is, our selection algorithm runs in $O(n + \delta\sqrt{n})$ deterministic time, tolerates up to δ faults, and produces a key whose rank is off the desired k by at most $O(\delta)$. To the best of our knowledge, this is the first result of such kind.

2 Randomized Sorting

In this section we develop a randomized resilient sorting algorithm (denoted by Randomized-Resilient-Sort) that runs in $O(n \log n + \delta\sqrt{n \log n})$ expected time.

In contrast to Finocchi et. al. we develop a quicksort-like sorting algorithm. The main obstacle for an algorithm of such kind is the lack of a resilient stack: the adversary can mislead the recursive invocation and hence leave a part of the sequence unsorted or processed more than once. In order to overcome this problem, one could use a trivially resilient stack (i.e., as earlier, replicate each pushed value $2\delta + 1$ times and use majority selection during pops). Quicksort performs $\Theta(n)$ stack operations during its execution, which immediately leads to $\Theta(\delta n)$ overhead. We propose simple modifications to quicksort that balance resiliency and running time.

For our purposes the following notion will be of use:

Definition 2. *A routine that gets a sequence S of length n and computes some permuted sequence S' is said to be a* robust *sorting algorithm if the following properties are met: (i) if no memory faults occur during the execution, then the routine runs in $O(n \log n)$ time and returns a sorted sequence (ii) otherwise the routine still terminates in $O(n \log n)$ time (without any guarantees about the output).*

Note that robustness does not imply resilience. In fact, the output of a robust routine may be arbitrarily off from the desired (due to memory faults).

The standard quicksort sorting algorithm is not robust since it employs recursion. Recursive algorithm may behave unpredictably when run in a faulty memory environment since a single stack corruption may cause the algorithm to crash or, even worse, to enter an infinite loop. To overcome this issue, we maintain a watchdog timer (implemented as a tick counter stored in a reliable memory). If more than $cn \log n$ ticks elapse (where c is a sufficiently large constant chosen based on the worst-case analysis), the algorithm terminates prematurely. In this case at least one memory fault had occurred, so case (ii) from the definition applies. We denote the resulting robust sorting routine by Robust-Sort. An alternative approach to devise such a routine would be to use a non-recursive mergesort.

Randomized-Resilient-Sort is recursive; it takes a sequence S of length n and an additional parameter t. We choose $t := \lceil \sqrt{n/\log n} \rceil$ (where n is the length of the input at the top-most level of recursion). This t is treated as a fixed parameter for the rest of the algorithm.

If $n \leq t$, then apply Robust-Sort to S and validate its result by iterating over S and comparing each element with the previous one (keeping the latter in a reliable memory). If the sequence looks correctly ordered, then terminate Randomized-Resilient-Sort. Otherwise restart by applying Robust-Sort again.

Now suppose $n > t$. Like in a usual randomized quicksort, choose a pivot p in S (independently and uniformly) and perform Partition of S around p. Let the resulting subsequences be L (containing the values less than p) and R (containing the values greater than p). If either $|L| < n/4$ or $|R| < n/4$, then restart by picking another pivot and applying Partition again.

Algorithm 1. RANDOMIZED-RESILIENT-SORT(S, t)

```
1: Let n := |S|
2: if n ≤ t then
3:    repeat
4:       S' := ROBUST-SORT(S)
5:    until S' is ordered correctly
6:    return S'
7: else
8:    repeat
9:       Choose a random pivot p in S
10:      (L, R) := PARTITION(S, p)
11:   until |L| ≥ n/4 and |R| ≥ n/4
12:   L' := RANDOMIZED-RESILIENT-SORT(L, t)
13:   R' := RANDOMIZED-RESILIENT-SORT(R, t)
14:   return L' ∘ (p) ∘ R'
15: end if
```

Otherwise recurse to L and R to obtain sorted subsequences L' and R'. Use trivially resilient storage for storing stack frames during recursive invocations. Finally output the concatenation of L', p, and R'.

Theorem 1. RANDOMIZED-RESILIENT-SORT *produces a faithfully ordered sequence and runs in* $O(n \log n + \delta\sqrt{n \log n})$ *expected time.*

Proof. The fact that the resulting sequence is faithfully ordered is obvious, so we shall focus on estimating the time complexity.

Let n be the length of initial sequence S, n' be the length of current sequence in recursive invocation and also let n'' denote the length of the sequence in the parent call to RANDOMIZED-RESILIENT-SORT (if any). Consider case when $n' \le t$, which is handled in Steps 3–6. Note that $n'' > t$ and Step 11 ensures that $n' \ge n''/4 > t/4$. Hence during the whole computation Step 4 is applied to at most $4n/t$ distinct and non-overlapping ranges. Not counting restarts due to failed validation, this takes $O(t \log t \cdot 4n/t) = O(n \log n)$ time (since $t < n$). There are at most δ restarts of Step 4, each taking $O(t \log t) = O(\sqrt{n \log n})$ additional time.

Next consider the time spent to maintain the resilient stack. Due to the check in Step 11 the recursion tree is nearly-balanced. Each leaf deals with a range of length from $t/4$ to t, so (as indicated above) there are at most $4n/t$ leaf calls. The total number of nodes in the recursion tree is also $O(n/t)$, which is $O(\sqrt{n \log n})$. Therefore RANDOMIZED-RESILIENT-SORT performs $O(\sqrt{n \log n})$ push and pop operations in total, each taking $O(\delta)$ time.

It remains to bound the time spent in Steps 8–11. Each single PARTITION takes $O(n')$ time but there may be multiple restarts due to unbalanced sizes of L and R. Without memory faults, the uniform choice of the pivot ensures that the correct sizes are observed with probability about $1/2$ (see, e.g. [5]). Hence the expected number of repetitions is constant. When faults are possible, the

analysis becomes more intricate since the adversary may corrupt keys and thus force the algorithm to produce unbalanced partitions.

Without corruptions, it takes $O(1)$ attempts to find a pivot p with rank in $[3n'/8, 5n'/8]$. Now if the adversary corrupts less than $n'/8$ keys during this iteration, p's rank remains in $[n'/4, 3n'/4]$, so the check in Step 11 succeeds (here we use the fact that our random bits are unknown to the adversary). Assume the contrary, i.e. at least $\alpha = n'/8$ keys are corrupted (perhaps leading to an unbalanced partition). Such a "large corruption" costs $O(n') = O(\alpha)$ (sic!) time (before another good pivot whose rank is in $[3n'/8, 5n'/8]$ is picked). Summing over all large corruption cases and using the fact that the sum of αs is bounded by δ, one concludes that the additional overhead incurred by the adversary is $O(\delta)$, which is negligible.

The desired bound of $O(n \log n + \delta\sqrt{n \log n})$ follows by summing up the above estimates. □

3 Resilient Selection

In order to devise a deterministic variation of the quicksort we have to learn to select a good pivot (like a median). From our perspective, following notions will be of use:

Definition 3. *Let S be a sequence of n* distinct *items. Let $\mathrm{rk}_S(x) = \mathrm{rk}(x)$ denote the* rank *of x, i.e. the number of items in S that are less than or equal to x. The k-th order statistic of S is an element $x \in S$ such that $\mathrm{rk}_S(x) = k$. For $k = \lceil n/2 \rceil$, the k-th order statistic is referred to as the* median.

In a usual RAM model finding the k-th order statistic in a sequence S of length n is a widely studied problem. For instance, in [5] one can find a randomized $O(n)$ expected time algorithm and also a deterministic $O(n)$ worst-case time "median of medians" algorithm. Other methods are known that achieve better asymptotic constants for the number of comparisons, see e.g. [6].

In a faulty environment it is impossible to compute the order statistic exactly. Indeed, the adversary may corrupt keys just before the algorithm stops running thus destroying the correct answer. To overcome this issue, we relax the notion as follows:

Definition 4. *Let S be a sequence. Then x is a Δ-*approximate k-th order statistic *if $|\mathrm{rk}_S(x) - k| \leq \Delta$.*

Note that in the above definition x need not be an element of S. This observation is important since in the faulty memory model no algorithm can guarantee to output an item belonging to the initial S (as corrupted and uncorrupted values are indistinguishable). Extending the above argument one can see that for $\Delta < \delta$ computing a Δ-approximate k-th order statistic is impossible since the adversary can alter an element's rank for up to δ by corrupting elements less than or greater than the element.

Algorithm 2. PIVOT(S, m)

```
Let n := |S|
Split S into k := ⌈n/m⌉ chunks C_1, ..., C_k of length m or m + 1 each
for all chunks C_i do
    repeat
        Compute x_i := ROBUST-MEDIAN(C_i)
    until x_i passes validation as the median
    Put x_i into a trivially resilient storage
end for
return TRIVIALLY-RESILIENT-MEDIAN(x_1, ..., x_k)
```

There are algorithms for computing a Δ-approximate k-th order statistic. Trivially resilient approach would be to replicate each element in $2\delta + 1$ copies and run the standard linear time algorithm with resilient operations. This would result in $O(\delta n)$ time and $O(\delta n)$ extra space. In particular case of median selection, we will refer to this algorithm as TRIVIALLY-RESILIENT-MEDIAN.

Interestingly, if we allow $\Delta = O(\delta)$, then it is possible to solve the problem in $O(n + \delta\sqrt{n})$ worst-case deterministic time and $O(\delta\sqrt{n})$ space. Our method is based on the "median of medians" algorithm [5]. However, extra care is taken to make the algorithm resilient to memory faults.

As in Section 2 we define an analogous notion of robustness with the same guarantees. Specifically, robust median selection routine either computes a median in linear time when there are no memory faults or terminates prematurely in $O(n)$ time without any guarantees on the output in the presence of memory faults. To devise robust median selection routine we equip a standard algorithm with a watchdog timer (as in Section 2). We denote the resulting robust routine by ROBUST-MEDIAN.

We use ROBUST-MEDIAN (which runs in $O(n)$ time) and also TRIVIALLY-RESILIENT-MEDIAN (which runs in $O(\delta n)$ time) to construct a new procedure called PIVOT.

PIVOT gets a sequence S of length n and computes a value p as follows (a pseudocode shown in Algorithm 2). The input sequence is divided into k *chunks* of length either m or $m + 1$. This can be achieved by using a Bresenham-like partitioning of n. We treat m as a parameter and set $k := \lceil n/m \rceil$. For each chunk C_i, run ROBUST-MEDIAN, denote its output by x_i, and store the latter in a reliable memory. Due to memory faults x_i may differ from the true median. Run a *validation of* x_i, that is, traverse C_i and count the number of elements in C_i that are less than x_i. If $\mathrm{rk}_{C_i}(x_i) \neq \lfloor \frac{1}{2}|C_i| \rfloor$, then restart ROBUST-MEDIAN computation for this particular chunk C_i.

Finally we get a value x_i that "looks like" a true median of C_i. (Strictly speaking, x_i is the median of C_i', where C_i' consists of values of C_i that were observed by the algorithm during the validation pass.) At this point the algorithm stores x_i in a trivially resilient storage (i.e. with replication factor $2\delta+1$) and proceeds to the next chunk. When all the chunks are processed, we get a sequence of k

values $x_1, \ldots, x_k$ (each stored in a trivially resilient storage). The algorithm runs Trivially-Resilient-Median for these values and outputs the result.

Lemma 1. *Let α be the number of memory faults occurred during the execution. Assuming that $2k + m < n/5$,* Pivot *takes $O(n + \alpha m + \delta n/m)$ time and the resulting value p obeys*

$$\mathrm{rk}_S(p) \geq n/5 - \alpha \quad \textit{and} \quad \mathrm{rk}_S(p) \leq 4n/5 + \alpha \tag{1}$$

Proof. Pivot takes $O(km + \delta k) = O(n + \delta n/m)$ plus the time incurred by failed validation passes. There are at most α such passes, each taking $O(m)$ time. The total time bound follows.

To prove (1), consider the chunks $C_1, \ldots, C_k$ comprising S. For each chunk C_i, the algorithm computes (and stores in a trivially resilient storage) a value x_i, which is a candidate for the median of C_i that had passed validation. This way, x_i may be viewed as the median of some fixed sequence C'_i (consisting of values of C_i as they were observed by the algorithm during the validation pass). Such C'_i is in a sense "virtual": it may happen that at no moment of time the current C_i coincides with C'_i. Since $|C'_i|$ is either m or $m + 1$ there are at least $\lfloor m/2 \rfloor$ elements in C'_i that are less than or equal to x_i.

The returned value p is the median of $x_1, \ldots, x_k$ (the exact one since it is computed in a trivially resilient fashion). Therefore at least $\lfloor k/2 \rfloor$ values among $x_1, \ldots, x_k$ are less than or equal to p. Let S' denote the sequence obtained by concatenating $C'_1, \ldots, C'_k$. From the above estimates we can conclude that at least $\Delta = \lfloor m/2 \rfloor \cdot \lfloor k/2 \rfloor$ values in S' are less than or equal to p. Then $\Delta \geq (k-1)(m-1)/4 \geq (n - 2k - m + 1)/4 \geq n/5$ (since $2k + m < n/5$). Finally note that S and S' differ in at most α positions. The proof for lower bound follows. The proof for upper bound with can be obtained by reversing all inequalities with proper modifications. □

Note that in (1) ranks are considered w.r.t. the initial sequence S (before any corruptions took place). However one can easily see that (1) remains true if S denotes the values of the input sequence as they were observed at some (possibly different) moments of time. Indeed, in the above proof S and S' still differ in at most α positions (since each mismatch corresponds to a memory fault).

Corollary 1. *A pivot p satisfying (1) can be found in $O(n + \delta\sqrt{n})$ time.*

Proof. Choose $m := \lfloor \sqrt{n} \rfloor$. Observe that $k = O(\sqrt{n})$ and $\alpha \leq \delta$. We may assume that n is large enough so $2k + m < n/5$ and Lemma 1 applies. □

Now we are ready to present Resilient-Select (see Algorithm 3 for a pseudocode). It takes a sequence S of length n (which is assumed to be large enough), a parameter number k, and finds, in $O(n + \delta\sqrt{n})$ time, an $O(\delta)$-approximate k-th order statistic in S.

Three cases are to be considered, depending on the magnitude of n.

1. **Tiny case**: $n \leq 5\delta$. The algorithm outputs an arbitrary key, e.g. $S[1]$.

Algorithm 3. RESILIENT-SELECT(S, k)

```
1: Let n := |S|, m := ⌊√n⌋
2: if n ≤ 5δ then {tiny case}
3:    return S[1]
4: else if 5δ < n ≤ 20δ then {small case}
5:    return PIVOT(S, m)
6: else {large case}
7:    p := PIVOT(S, m)
8:    (L, R) := PARTITION(S, p)
9:    if |L| ≤ k then
10:      return RESILIENT-SELECT(L, k)
11:   else
12:      return RESILIENT-SELECT(R, k − |L|)
13:   end if
14: end if
```

2. **Small case**: $5\delta < n \le 20\delta$. Apply PIVOT to S (using $m := \lfloor\sqrt{n}\rfloor$) and output the resulting value.
3. **Large case**: $n > 20\delta$. First, invoke PIVOT (using, as above, $m := \lfloor\sqrt{n}\rfloor$) and denote the resulting value by p. Second, perform PARTITION of S around p into subsequences L and R; that is, enumerate the elements of S and put those not exceeding p into L and the others into R. (For simplicity's sake, we assume that all elements are distinct. For coinciding elements, a more careful choice between L and R is needed to avoid unbalanced partitions. These details are quite technical and we omit them due to the lack of space.) These two new sequences L and R are stored in the usual, faulty memory. Their lengths $|L|$ and $|R|$, however, are stored in reliable (safe) memory. Third, if $k \le |L|$, then recurse to L. Otherwise reset $k := k - |L|$ and recurse to R. (These steps are standard for divide-and-conquer selection algorithms.) The tail recursion in RESILIENT-SELECT can be easily eliminated, hence there is no need for a stack.

Theorem 2. RESILIENT-SELECT *runs in* $O(n+\delta\sqrt{n})$ *time and returns an* $O(\delta)$*-approximate* k*-th order statistic.*

Proof. Let S be the input sequence where an approximate k-th order statistic is to be found. Due to memory faults the sequence may be altered during execution; we shall denote its current state by S'.

For the tiny case, any returned value would satisfy as approximation. For the small and large case we use induction to show that the returned value p is indeed a 20δ-approximate k-order statistic. Specifically, we claim that two inequalities hold (where α is the number of faults occurred during the invocation):

$$\min S \le p \le \max S \quad \text{and} \quad |\mathrm{rk}_S - k| \le 20\alpha.$$

The small case is an inductive base. It follows from Lemma 1 and Corollary 1 that $\mathrm{rk}_S(p) \ge n/5 - \alpha \ge 1$ and $\mathrm{rk}_S(p) \le 4n/5 + \alpha < 20\delta \le n$. Hence p is

between $\min S$ and $\max S$. For any $x \in S$, $|\mathrm{rk}_S(p) - \mathrm{rk}_S(x)| \leq 4n/5 + \alpha \leq 20\alpha$. In particular $|\mathrm{rk}_S(p) - k| \leq 20\alpha$, as desired.

Now consider a large case. Note that PARTITION reads each element of S exactly once. Hence one may imagine a sequence S' consisting of values in S as they were observed by PARTITION. Also we may assume that L and R form a partition of S' (not S) around p. Indeed, all memory faults that occur in L or R and affect the values written by PARTITION may be effectively "postponed" until RESILIENT-SELECT calls itself recursively. In other words, these faults are regarded as occurring in the latter recursive invocation.

Let γ (respectively β) be the total number of faults occurred in RESILIENT-SELECT from start until the recursive invocation is made (respectively during the recursive invocation). Suppose RESILIENT-SELECT recurses to L. Then by the inductive assumption the returned value x satisfies $|\mathrm{rk}_L(x) - k| \leq 20\beta$. Observe the equality $\mathrm{rk}_L(x) = \mathrm{rk}_{S'}(x)$ (here we use the fact that $\min L \leq x \leq \max L$). Altogether this implies $|\mathrm{rk}_{S'}(x) - k| \leq 20\beta$.

Sequences S and S' differ by at most γ elements (the latter is the upper bound for the number of memory faults occurred during RESILIENT-SELECT not counting the recursion). Therefore $|\mathrm{rk}_S(x) - \mathrm{rk}_{S'}(x)| \leq \gamma$, so $|\mathrm{rk}_S(x) - k| \leq 20\beta + \gamma \leq 20(\beta + \gamma) = 20\delta$.

The case when RESILIENT-SELECT recurses to R is analogous.

Finally let us estimate the complexity of RESILIENT-SELECT. The i-th level of recursion, which is applied to a sequence of length n_i, takes $O(n_i + \delta\sqrt{n_i})$ time (see Corollary 1). Summing over all levels one gets $O\left(\sum_i n_i + \delta \sum_i \sqrt{n_i}\right)$. Each recursive call reduces the current length n_i to $n_{i+1} \leq 4n_i/5 + \delta \leq 4n_i/5 + n_i/20 = 17n_i/20$. Since the decrease is exponential, the estimate becomes $O(n + \delta\sqrt{n})$, as claimed. □

Corollary 2. *There exists a deterministic $O(n)$-time $O(\delta)$-approximate median selection algorithm that tolerates up to $\delta = O(\sqrt{n})$ memory faults.*

4 Deterministic Sorting

Now we are ready to present a deterministic variation of quicksort (denoted by DETERMINISTIC-RESILIENT-SORT) with $O(n \log n + \delta\sqrt{n} \log n)$ worst-case running time. We effectively combine ideas from Section 2 and Section 3.

Let n be the length of the input sequence (which is assumed to be sufficiently large). The outline is the same as in the randomized algorithm in Section 2 except for three aspects. First, at the top level we choose $t := \lceil \sqrt{n} \rceil$ (in contrast to $t := \lceil \sqrt{n/\log n} \rceil$ in the randomized approach). Second, instead of selecting a random pivot we use deterministic PIVOT procedure in conjunction with PARTITION to ensure that resulting partition is nearly-balanced (i.e. there are at least 1/10 of elements in each part). Third, we perform a fallback to ROBUST-SORT when the length of the current segment becomes smaller than $15t$ rather than t. The latter is done to satisfy the preconditions of Lemma 1, namely, to make sure that

Algorithm 4. DETERMINISTIC-RESILIENT-SORT(S, t)

```
1: Let n := |S|
2: if n ≤ 15t then
3:    repeat
4:       S' := ROBUST-SORT(S)
5:    until S' is ordered correctly
6:    return S'
7: else
8:    repeat
9:       Let p := PIVOT(S, t)
10:      (L, R) := PARTITION(S, p)
11:   until |L| ≥ n/10 and |R| ≥ n/10
12:   L' := DETERMINISTIC-RESILIENT-SORT(L, t)
13:   R' := DETERMINISTIC-RESILIENT-SORT(R, t)
14:   return L' ∘ (p) ∘ R'
15: end if
```

$2t + \lceil n'/t \rceil < n'/5$ for the current length n'. (Indeed, $2t + \lceil n'/t \rceil \le 3t < n'/5$ when $n' > 15t$.) For a pseudocode, see Algorithm 4.

Theorem 3. DETERMINISTIC-RESILIENT-SORT *produces a faithfully ordered sequence and runs in* $O(n \log n + \delta\sqrt{n}\, \log n)$ *deterministic time.*

Proof. Once again, the fact that the resulting sequence is faithfully ordered is obvious, so we focus on estimating the time complexity.

Consider the recursion tree of DETERMINISTIC-RESILIENT-SORT. The latter is nearly-balanced (by the same argument as in the proof of Theorem 1) and thus contains $O(n/t)$ leaves. Hence ROBUST-SORT, which is invoked at the bottom level, costs $O(n/t \cdot t \log t + \delta t \log t)$ time.

At each level of the recursion tree some calls to PIVOT and PARTITION are made. Since PARTITION takes linear time, it contributes $O(n \log n)$ plus the time spent in additional restarts incurred by failed validations. To estimate PIVOT's total time, let N be the total length of segments for which PIVOT is invoked. Clearly $N = O(n \log n)$ (since the tree is nearly-balanced). Then by Lemma 1 all PIVOTs take $O(N + \delta t + \delta N/t)$ time (not counting additional restarts).

Consider additional restarts occurring in Steps 8–11. As in in the proof of Theorem 1, without corruptions PIVOT returns p whose rank is in $[n'/5, 4n'/5]$ (where n' is the current length of the segment). Now if the adversary corrupts less than $n'/10$ keys during the iteration, p's rank remains in $[n'/10, 9n'/10]$, so the check in Step 11 succeeds. Assume the contrary, i.e. at least $\alpha = n'/10$ keys are corrupted (perhaps leading to an unbalanced partition). Such a "large corruption" increases N by 10α, which adds up to $O(\delta)$. Hence the overhead is $O(\delta + \delta^2/t)$, which is negligible.

Summing up these estimates and plugging in $t = \sqrt{n}$ yields the desired time bound. □

References

1. Boyer, R.S., Moore, J.S.: MJRTY - A Fast Majority Vote Algorithm (1982)
2. Brodal, G.S., Fagerberg, R., Finocchi, I., Grandoni, F., Italiano, G.F., Jørgensen, A.G., Moruz, G., Mølhave, T.: Optimal Resilient Dynamic Dictionaries. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 347–358. Springer, Heidelberg (2007)
3. Brodal, G.S., Jørgensen, A.G., Mølhave, T.: Fault Tolerant External Memory Algorithms. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 411–422. Springer, Heidelberg (2009)
4. Brodal, G.S., Jørgensen, A.G., Moruz, G., Mølhave, T.: Counting in the Presence of Memory Faults. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 842–851. Springer, Heidelberg (2009)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd revised edn. The MIT Press (2001)
6. Dor, D.: Selection Algorithms. Ph.D. thesis, Tel-Aviv University (1995)
7. Ferraro-Petrillo, U., Finocchi, I., Italiano, G.: Experimental Study of Resilient Algorithms and Data Structures. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 1–12. Springer, Heidelberg (2010)
8. Ferraro-Petrillo, U., Grandoni, F., Italiano, G.: Data Structures Resilient to Memory Faults: An Experimental Study of Dictionaries. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 398–410. Springer, Heidelberg (2010)
9. Finocchi, I., Grandoni, F., Italiano, G.: Designing reliable algorithms in unreliable memories. Computer Science Review 1(2), 77–87 (2007)
10. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient search trees. In: Proc. SODA 2007, pp. 547–553. Society for Industrial and Applied Mathematics (2007)
11. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal resilient sorting and searching in the presence of memory faults. Theor. Comput. Sci. 410, 4457–4470 (2009)
12. Finocchi, I., Italiano, G.F.: Sorting and searching in the presence of memory faults (without redundancy). In: Proc. STOC 2004, pp. 101–110. ACM (2004)
13. Hamdioui, S., Ars, Z.A., Van De Goor, A.J., Rodgers, M.: Dynamic Faults in Random-Access-Memories: Concept, Fault Models and Tests. J. Electron. Test. 19, 195–205 (2003)
14. Jørgensen, A.G., Moruz, G., Mølhave, T.: Priority Queues Resilient to Memory Faults. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 127–138. Springer, Heidelberg (2007)
15. Li, X., Huang, M.C., Shen, K., Chu, L.: A realistic evaluation of memory hardware errors and software system susceptibility. In: Proc. USENIX 2010. USENIX Association (2010)
16. May, T.C., Woods, M.H.: Alpha-particle-induced soft errors in dynamic memories. IEEE Transactions on Electron Devices 26(1), 2–9 (1979)
17. Ferraro-Petrillo, U., Finocchi, I., Italiano, G.F.: The Price of Resiliency: A Case Study on Sorting with Memory Faults. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 768–779. Springer, Heidelberg (2006)
18. Schroeder, B., Gibson, G.A.: A large-scale study of failures in high-performance computing systems. In: DSN 2006: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006), pp. 249–258. IEEE Computer Society, Los Alamitos (2006)