Fragile Complexity of Comparison-Based Algorithms*

Peyman Afshani
Aarhus University
peyman@cs.au.dk

Rolf Fagerberg
University of Southern Denmark
rolf@imada.sdu.dk

David Hammer[1]
University of Southern Denmark
hammer@imada.sdu.dk

Riko Jacob
IT University of Copenhagen
rikj@itu.dk

Irina Kostitsyna
TU Eindhoven
i.kostitsyna@tue.nl

Ulrich Meyer[1]
Goethe University Frankfurt
umeyer@ae.cs.uni-frankfurt.de

Manuel Penschuck[1]
Goethe University Frankfurt
mpenschuck@ae.cs.uni-frankfurt.de

Nodari Sitchinava[2]
University of Hawaii at Manoa
nodari@hawaii.edu

Abstract

We initiate a study of algorithms with a focus on the computational complexity of individual elements, and introduce the *fragile complexity* of comparison-based algorithms as the maximal number of comparisons any individual element takes part in. We give a number of upper and lower bounds on the fragile complexity for fundamental problems, including Minimum, Selection, Sorting and Heap Construction. The results include both deterministic and randomized upper and lower bounds, and demonstrate a separation between the two settings for a number of problems. The depth of a comparator network is a straight-forward upper bound on the worst case fragile complexity of the corresponding fragile algorithm. We prove that fragile complexity is a different and strictly easier property than the depth of comparator networks, in the sense that for some problems a fragile complexity equal to the best network depth can be achieved with less total work and that with randomization, even a lower fragile complexity is possible.

* This material is based upon work performed while attending AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa, in part supported by the National Science Foundation under Grant No. 1745331.
[1] Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.
[2] Supported by the National Science Foundation under Grant No. 1533823.

1 Introduction

Comparison-based algorithms is a classic and fundamental research area in computer science. Problems studied include minimum, median, sorting, searching, dictionaries, and priority queues, to name a few, and by now a huge body of work exists. The cost measure analyzed is almost always the total number of comparisons needed to solve the problem, either in the worst case or the expected case. Surprisingly, very little work has taken the viewpoint of the individual elements, asking the question: *how many comparisons must each element be subjected to?*

This question not only seems natural and theoretically fundamental, but is also practically well motivated: in many real world situations, comparisons involve some amount of destructive impact on the elements being compared, hence, keeping the maximum number of comparisons for each individual element low can be important. One example of such a situation is ranking of any type of consumable objects (wine, beer, food, produce), where each comparison reduces the available amount of the objects compared. Here, classical algorithms like QuickSort, which clearly has fragile complexity $\Omega(n)$ from its partitioning phase, may use up the chosen pivot elements long before the algorithm completes. Another example is sports, where each comparison constitutes a match and takes a physical toll on the athletes involved. If a comparison scheme subjects one contestant to many more matches than others, both fairness to contestants and quality of result are impacted. The selection process could even contradict its own purpose—what is the use of finding a national boxing champion to represent a country at the Olympics if the person is injured in the process? Notice that in both examples above, quality of elements is difficult to measure objectively by a numerical value, hence one has to resort to relative ranking operations between opponents, i.e., comparisons. The detrimental impact of comparisons may also be of less directly physical nature, for instance if it involves a privacy risk for the elements compared, or if bias in the comparison process grows each time the same element is used.

▶ **Definition 1.** We say a comparison-based algorithm $\mathcal{A}$ has *fragile complexity* of $f(n)$ if it performs at most $f(n)$ comparisons for each input element. We also say that $\mathcal{A}$ has *work* of $w(n)$ if it performs at most $w(n)$ comparisons in total. We say that in a particular algorithm $\mathcal{A}$ an element e has fragile complexity of $f_e(n)$ if e participates in at most $f_e(n)$ comparisons when executing $\mathcal{A}$.

In this paper, we initiate the study of algorithms' fragile complexity—comparison-based complexity from the viewpoint of the individual elements—and present a number of upper and lower bounds on the fragile complexity for fundamental problems.

1.1 Previous work

One body of work relevant to what we study here is the study of sorting networks, propelled by the 1968 paper of Batcher [5]. In sorting networks, and more generally comparator networks, the notions of depth and size correspond to fragile complexity and standard worst case complexity,[3] respectively, since a network with depth $f(n)$ and size $w(n)$ easily can be converted into a comparison-based algorithm with fragile complexity $f(n)$ and work $w(n)$.

Batcher gave sorting networks with $\mathcal{O}(\log^2 n)$ depth and $\mathcal{O}(n \log^2 n)$ size, based on clever variants of the MergeSort paradigm. A number of later constructions achieve the same

[3] For clarity, in the rest of the paper we call standard worst case complexity *work*.

bounds [9, 14, 15, 18], and for a long time it was an open question whether better results were possible. In the seminal result in 1983, Ajtai, Komlós, and Szemerédi [1, 2] answered this in the affirmative by constructing a sorting network of $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n \log n)$ size. This construction is quite complex and involves expander graphs [20, 21], which can be viewed as objects encoding pseudorandomness, and which have many powerful applications in computer science and mathematics. The size of the constant factors in the asymptotic complexity of the AKS sorting network prevents it from being practical in any sense. It was later modified by others [7, 11, 16, 19], but finding a simple, optimal sorting network, in particular one not based on expander graphs, remains an open problem. Comparator networks for other problems, such as selection and heap construction have also been studied [4, 6, 13, 17, 22]. In all these problems the size of the network is super-linear.

As comparator networks of depth $f(n)$ and size $w(n)$ lead to comparison-based algorithms with $f(n)$ fragile complexity and $w(n)$ work, a natural question is, whether the two models are equivalent, or if there are problems for which comparison-based algorithms can achieve either asymptotically lower $f(n)$, or asymptotically lower $w(n)$ for the same $f(n)$.

One could also ask about the relationship between parallelism and fragile complexity. We note that parallel time in standard parallel models generally does not seem to capture fragile complexity. For example, even in the most restrictive exclusive read and exclusive write (EREW) PRAM model it is possible to create n copies of an element e in $\mathcal{O}(\log n)$ time and, thus, compare e to all the other input elements in $\mathcal{O}(\log n)$ time, resulting in $\mathcal{O}(\log n)$ parallel time but $\Omega(n)$ fragile complexity. Consequently, it is not clear whether Richard Cole's celebrated parallel merge sort algorithm [8] yields a comparison-based algorithm with low fragile complexity as it copies some elements.

Problem		*Upper*		*Lower*
		$f(n)$	$w(n)$	$f(n)$
Minimum	Determ. (Sec. 2)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\Omega(\log n)$
	Rand. (Sec. 2)	$\langle \mathcal{O}(1)^\dagger, n^\varepsilon \rangle$ $\langle \mathcal{O}(\log_\Delta n)^\dagger, \mathcal{O}(\Delta + \log_\Delta n)^\dagger \rangle$	$\mathcal{O}(n)$	$\langle \Omega(\log_\Delta n)^\dagger, \Delta \rangle$
		$\langle O(\log_\Delta n \log\log \Delta)$, $O(\Delta + \log_\Delta n \log\log \Delta) \rangle^\ddagger$	$\mathcal{O}(n)$	$\Omega(\log\log n)^\ddagger$
Selection	Determ. (Sec. 3)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\Omega(\log n)$
	Rand. (Sec. 3)	$\left\langle \mathcal{O}(\log\log n)^\dagger, \mathcal{O}\left(\sqrt{n}\right)^\dagger \right\rangle$ $\left\langle \mathcal{O}\left(\frac{\log n}{\log\log n}\right)^\dagger, \mathcal{O}(\log n)^\dagger \right\rangle$	$\mathcal{O}(n)$	$\langle \Omega(\log_\Delta n)^\dagger, \Delta \rangle$
Merge	Determ. (Sec. 4)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\Omega(\log n)$
MergeSort	Determ. (Sec. 4)	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n \log n)$	$\Omega(\log^2 n)$
MergeSort	Rand. (Sec. 4)	$\mathcal{O}(\log n)$ whp	$\mathcal{O}(n \log n)$	
Heap Construction	Determ. (Sec. 5)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\Omega(\log n)$

Table 1 Summary of presented results. Notation: $f(n)$ – fragile complexity; $w(n)$ – work; $\langle f_m(n), f_{rem}(n) \rangle$ – fragile complexity for the element of interest (minimum/median) and the remaining elements, respectively; † – holds in expectation, ‡ – holds with high probability.

1.2 Our contribution

In this paper we present algorithms and lower bounds for a number of classical problems, summarized in Table 1. In particular, we study finding the MINIMUM (Section 2), the SELECTION problem (Section 3), and SORTING (Section 4).

Minimum. The case of the deterministic algorithms is clear: using an adversary lower bound, we show that the minimum element needs to suffer $\Omega(\log n)$ comparisons and a tournament tree trivially achieves this bound (Subsection 2.1). The randomized case, however, is much more interesting. We obtain a simple algorithm where the probability of the minimum element suffering k comparisons is doubly exponentially low in k, roughly $1/2^{2^k}$ (see Subsection 2.2). As a result, the $\Theta(\log n)$ deterministic fragile complexity can be lowered to $O(1)$ expected or even $O(1) + \log\log n$ with high probability. We also show this latter high probability case is lower bounded by $\Omega(1) + \log\log n - \log\log\log n$ (Subsection 2.3). Furthermore, we can achieve a trade-off between the fragile complexity of the minimum element and the other elements. We can find the minimum with $O(\log_\Delta n)$ expected fragile complexity while all the other elements suffer $O(\Delta + \log_\Delta n)$ comparisons (Subsection 2.3). Furthermore, this is tight: we show an $\Omega(\log_\Delta n)$ lower bound for the expected fragile complexity of the minimum element where the maximum fragile complexity of non-minimum elements is at most Δ.

Selection. Selection is a special case of the minimum finding problem where we are interested in finding an element of a given rank. As a result, all of our lower bounds apply to this problem as well. Regarding algorithms, the deterministic case is trivial if we allow for $O(n \log n)$ time (via sorting). We show that this can be reduced to $O(n)$ time while keeping the fragile complexity of all the elements at $O(\log n)$ (Section 3). Once again, randomization offers a substantial improvement: e.g., we can find the median in $O(n)$ time and with $O(\log\log n)$ fragile complexity while non-median elements suffer $O(\sqrt{n})$ comparisons, or we can find the median in $O(n)$ time and with $O(\log n/\log\log n)$ fragile complexity while non-median elements suffer $O(\log n)$.

Sorting and other results. The deterministic selection, sorting, and heap construction fragile complexities follow directly from the classical results in comparator networks [2, 6]. However, we show a separation between comparator networks and comparison-based algorithms for the problem of MEDIAN (Section C.1) and HEAP CONSTRUCTION (Section 5), in the sense that depth/fragile complexity of $\mathcal{O}(\log n)$ can be achieved in $\mathcal{O}(n)$ work for comparison-based algorithms, but requires $\Omega(n \log n)$ [4] and $\Omega(n \log\log n)$ [6] sizes for comparator networks for the two problems, respectively. For sorting the two models achieve the same complexities: $\mathcal{O}(\log n)$ depth/fragile complexity and $\mathcal{O}(n \log n)$ size/work, which are the optimal bounds in both models due to the $\Omega(\log n)$ lower bound on fragile complexity for MINIMUM (Theorem 2) and the standard $\Omega(n \log n)$ lower bound on work for comparison-based sorting. However, it is an open problem whether these bounds can be achieved by simpler sorting algorithms than sorting networks, in particular whether expander graphs are necessary. One intriguing conjecture could be that any comparison-based sorting algorithm with $\mathcal{O}(\log n)$ fragile complexity and $\mathcal{O}(n \log n)$ work implies an expander graph. This would imply expanders, optimal sorting networks and fragile-optimal comparison-based sorting algorithms to be equivalent, in the sense that they all encode the same level of pseudorandomness.

We note that our lower bound of $\Omega(\log^2 n)$ on the fragile complexity of MERGESORT (Theorem 14) implies the same lower bound on the depth of any sorting network based on binary merging, which explains why many of the existing simple sorting networks have

$\Theta(\log^2 n)$ depth. Finally, our analysis of MergeSort on random inputs (Theorem 16) shows a separation between deterministic and randomized fragile complexity for such algorithms. In summary, we consider the main contributions of this paper to be:

- the introduction of the model of fragile complexity, which we find intrinsically interesting, practically relevant, and surprisingly overlooked
- the separations between this model and the model of comparator networks
- the separations between the deterministic and randomized setting within the model
- the lower bounds on randomized minimum finding

Due to space constraints, a number of proofs in this paper appear in appendices.

2 Finding the minimum

2.1 Deterministic Algorithms

As a starting point, we study deterministic algorithms that find the minimum among an input of n elements. Our results here are simple but they act as interesting points of comparison against the subsequent non-trivial results on randomized algorithms.

▶ **Theorem 2.** *The fragile complexity of finding the minimum of n elements is $\lceil \log n \rceil$.*

Proof. The upper bound is achieved using a perfectly balanced tournament tree. The lower bound follows from a standard adversary argument. ◀

▶ **Corollary 3.** *For any deterministic algorithm $\mathcal{A}$ that finds the median of n elements, the fragile complexity of the median element is at least $\lceil \log n \rceil - 1$.*

Proof. By a standard padding argument with $n - 1$ small elements. ◀

2.2 Randomized Algorithms for Finding the Minimum

We now show that finding the minimum is provably easier for randomized algorithms than for deterministic algorithms. We define $f_{\min}$ as the fragile complexity of the minimum and f_{rem} as the maximum fragile complexity of the remaining elements. For deterministic algorithms we have shown that $f_{\min} \geq \log n$ regardless of f_{rem}. This is very different in the randomized world. In particular, we first show that we can achieve $\mathbb{E}[f_{\min}] = O(1)$ and $f_{\min} = O(1) + \log \log n$ with high probability (we later show this high probability bound is also tight, Lemma 10).

▶ **Theorem 4.** *There is a simple randomized algorithm finding the minimum where the minimum participates in k comparisons with probability at most $\frac{2^k}{2^{2^k-1}}$. Note this is a doubly exponential drop in the probability implying $\mathbb{E}[f_{min}] = O(1)$ and $f_{min} \leq O(1) + \log \log n$ whp.*

Proof. We actually find the minimum and the second minimum element, as follows.

1: **procedure** SimpleMininum(X) ▷ Find the minimum and the second minimum in X
2: If $|X| \leq 4$, sort X and **return** the minimum and the second minimum.
3: Let $A \subseteq X$ where each element of X is sampled independently with prob. $1/2$.
4: Recurse on A to find its smallest, a_1, and second smallest, a_2, elements.
5: Let C be the elements of $B := X \setminus A$ that are smaller than a_2.
6: Find the minimum, c_1, of C using a tournament tree. If $C = \varnothing$ then define $c_1 = a_2$.
7: **return** the minimum of X as $\min\{a_1, c_1\}$ and second minimum of X as $\max\{a_1, c_1\}$.

The correctness is easy to see. If $C \neq \varnothing$ then $c_1 < a_2$ and the last line of the algorithm returns the correct value. If $C = \varnothing$, a_1 and a_2 are the first and the second smallest element.

We now focus on its analysis. To be able to distinguish the sets X, A, B and C used in the algorithm over its different recursion levels, we use the superscript $X^{(i)}, A^{(i)}, B^{(i)}, C^{(i)}$ to denote the these sets at depth i of the recursion, more precisely $X^{(1)} = X$, and $X^{(i+1)} = A^{(i)}$.

Let y be the minimum element. Let $\mathbf{t}$ denote the random variable which corresponds to the recursion level at which $y \in B^{(\mathbf{t})}$; informally, $\mathbf{t}$ is the last recursion step where the minimum has survived in the sampling process. Observe that $y \notin X^{(t')}$ for any $t' > \mathbf{t}$. Furthermore, observe that for any integer $t > 0$, $A^{(t)}$ is a random sample of X where each element has been sampled with probability $1/2^t$. Thus, the event $\mathbf{t} = t$ happens if and only if y is sampled in $A^{(t-1)}$ but not in $A^{(t)}$. Thus, $\Pr[\mathbf{t} = t] = 2^{-t+1}.\frac{1}{2} = 2^{-t}$.

Conditioned on $\mathbf{t} = t$, we now compute the probability of the event $|C^{(t)}| = r$, for any integer $r \geq 1$. Let $y = x_1, x_2, \ldots, x_\ell$ be the sorted list of element in $X^{(t)}$, and remember that C consists of the elements smaller than the second smallest sampled element. Conditioned on $\mathbf{t} = t$, we already know that $y = x_1$ is not sampled in $A^{(t)}$ and thus $x_1 \in C^{(t)}$. Now observe that $|C^{(t)}| = r$ means that we sample x_{r+2} in $A^{(t)}$ plus precisely one additional element among $x_2, \ldots, x_{r+1}$ in $A^{(t)}$. The probability of this happening is $\frac{r+1}{2^{r+1}} \leq \frac{r}{2^r}$ (because $r \geq 1$) or 0 if $X^{(t)}$ doesn't have enough elements. Either way, we have

$$\Pr[|C^{(t)}| = r | \mathbf{t} = t] \leq \frac{r}{2^r}. \tag{1}$$

Next, we use a tournament tree on $|C^{(t)}|$ that will impose $\log(r)$ comparisons on the minimum y.

Observe that the events $\mathbf{t} = t$, are mutually exclusive over $t = 1, 2, \ldots$ and we also have $\sum_{t=1}^{\infty} \Pr[\mathbf{t} = t] = 1$. Thus, by law of total probability, we have

$$\Pr[f_{\min} = k] = \sum_{t=1}^{\infty} \Pr[f_{\min} = k | \mathbf{t} = t] \Pr[\mathbf{t} = t] = \sum_{t=1}^{\infty} \Pr[f_{\min} = k | \mathbf{t} = t] \frac{1}{2^t}. \tag{2}$$

It remains to analyze $\Pr[f_{\min} = k | \mathbf{t} = t]$. Observe that conditioned on $\mathbf{t} = t$, we have $f_{\min} = 1 + \log |C^{(t)}|$, which in turn implies that $1 + \log |C^{(t)}| = k$ and thus, $|C^{(t)}| = 2^{k-1}$. By Eq. 1, this happens with probability at most $\frac{2^{k-1}}{2^{2^{k-1}}}$. Combining this with Eq. 2 we get

$$\Pr[f_{\min} = k] \leq \sum_{t=1}^{\infty} \frac{2^{k-1}}{2^{2^{k-1}}} \frac{1}{2^t} \leq \frac{2^{k-1}}{2^{2^{k-1}}}$$

which is precisely what we have claimed. ◀

The major strengths of the above algorithm is the doubly exponential drop in probability of comparing the minimum to too many elements. Based on it, we can design another simple algorithm to provide a smooth trade-off between $f_{\min}$ and f_{rem}. Let $2 \leq \Delta \leq n$ be an integral parameter. We will design an algorithm that achieves $\mathbb{E}[f_{\min}] = O(\log_\Delta n)$ and $f_{\min} = O(\log_\Delta n \cdot \log\log \Delta)$ whp, and $f_{\text{rem}} = \Delta + O(\log_\Delta n \cdot \log\log \Delta)$ whp. For simplicity we assume n is a power of Δ. We build a fixed tournament tree T of degree Δ and of height $\log_\Delta n$ on X. For a node $v \in T$, let $X(v)$ be the set of values in the subtree rooted at v. The following code computes $m(v)$, the minimum value of $X(v)$, for every node v.

1: **procedure** RMININUM$_\Delta(X)$
2: For every leaf v, set $m(v)$ equal to the single element of $X(v)$.
3: For every internal node v with Δ children $u_1, \ldots, u_\Delta$ where the values $m(u_1), \ldots, m(u_\Delta)$ are known, compute $m(v)$ using SIMPLEMINIMUM algorithm on input $\{m(u_1), \ldots, m(u_\Delta)\}$.
4: Repeat the above step until the minimum of X is computed.

The correctness of RMININUM$_\Delta$ is trivial. So it remains to analyze the fragile complexity.

▶ **Lemma 5.** *Let $h = \log_\Delta n$. Then for every $k \geq 2h$, $\Pr[f_{min} = k] \leq \frac{1}{2^{h \cdot 2^{\Omega(k/h)}}}$.*

Proof. There are $h = \log_\Delta n$ levels in T. Let $\mathbf{f}_i$ be the random variable that counts the number of comparisons the minimum participates in at level i of T. We have $\sum_{i=1}^{h} \mathbf{f}_i = k$. Let $f_1, \ldots, f_h$ be integers such that $f_i \geq 1$ and $\sum_{i=1}^{h} f_i = k$. Using Theorem 4, we have

$$\Pr[\mathbf{f}_1 = f_1 \wedge \cdots \wedge \mathbf{f}_h = f_h] \leq \prod_{i=1}^{h} \frac{2^{f_i}}{2^{2^{f_i - 1}}} \leq 2^k \frac{1}{2^{\sum_{i=1}^{h} 2^{f_i - 1}}} \leq 2^k \frac{1}{2^{h \cdot 2^{\Omega(k/h)}}} \tag{3}$$

where the last inequality follows from the inequality of arithmetic and geometric means (specifically, observe that $\sum_{i=1}^{h} 2^{f_i - 1}$ is minimized when all f_i's are distributed evenly).

Now observe that the total number of different integral sequences $f_1, \ldots, f_h$ that sum up to k is bounded by $\binom{h+k}{h}$ (this is the classical problem of distributing k identical balls into h distinct bins). Thus, by union bound and assuming $k \geq 2h$, we have

$$\Pr[f_{\min} = k] \leq \binom{h+k}{h} \cdot 2^k \frac{1}{2^{h \cdot 2^{\Omega(k/h)}}} \leq \left(\frac{O(k)}{h}\right)^h \left(\frac{2^{k/k}}{2^{2^{\Omega(k/h)}}}\right)^h \leq \frac{1}{2^{h \cdot 2^{\Omega(k/h)}}}. \quad \blacktriangleleft$$

▶ **Theorem 6.** *In RMINIMUM$_\Delta$, $\mathbb{E}[f_{min}] = O(\log_\Delta n)$. Furthermore, with high probability, $f_{min} = O(\frac{\log n \log\log \Delta}{\log \Delta})$ and $f_{rem} = O(\Delta + \frac{\log n \log\log \Delta}{\log \Delta})$.*

Proof. First, observe that $\mathbb{E}[f_{\min}] = O(\log_\Delta n)$ is an easy consequence of Lemma 5. Now we focus on high probability bounds. Setting $k = c \cdot h \log\log \Delta$, and $h = \log_\Delta n$, in Lemma 5 will yield $\Pr[f_{\min} \geq k] = \frac{1}{2^{h \cdot 2^{\Omega(c \log\log \Delta)}}} = \frac{1}{2^{h \cdot \log^{O(c)} \Delta}} = \frac{1}{2^{\log n \cdot \log^{O(c)-1} \Delta}}$, which is a high probability bound for $f_{\min}$. To bound f_{rem}, observe that for every non-minimum element x, there exists a highest non-root node v such that $m(v) = x$. x is not passed to ancestors of v and at most suffers Δ additional comparisons in step 3 of RMINIMUM algorithm at node v. Now observe that since x is the minimum at node v we can apply Lemma 5 for the subtree rooted at v. This yields that whp we have $f_{\text{rem}} = \Delta + O(\frac{\log n \log\log \Delta}{\log \Delta})$. ◀

2.3 Randomized Lower Bounds for Finding the Minimum

2.3.1 Expected Lower Bound for the Fragile Complexity of the Minimum.

The following theorem is our main result. However, in this section, we only cover the main proof ideas and the full proof with all its details are in the Appendix B.2.

▶ **Theorem 7.** *In any randomized minimum finding algorithm with fragile complexity of at most Δ for any element, the expected fragile complexity of the minimum is at least $\Omega(\log_\Delta n)$.*

We prove the lower bound for a *deterministic* algorithm $\mathcal{A}$ on a random input of n values, $x_1, \ldots, x_n$ where each x_i is chosen iid and uniformly in $(0, 1)$. By Yao's minimax principle,

the lower bound on the expected fragile complexity of the minimum when running $\mathcal{A}$ also holds for any randomized algorithm.

We prove our lower bound in a model that we call "comparisons with additional information (CAI)": if the algorithm $\mathcal{A}$ compares two elements x_i and x_j and it turns out that $x_i < x_j$, then the value x_j is revealed to the algorithm. Clearly, the algorithm can only do better with this extra information. The heart of the proof is the following lemma which also acts as the "base case" of our proof.

▶ **Lemma 8.** *Consider T values $x_1, \ldots, x_T$ chosen iid and uniformly in $(0, b)$. Consider a deterministic algorithm $\mathcal{A}$ in CAI model that finds the minimum value y among $x_1, \ldots, x_T$. If $T > 1000\Delta$, then with probability at least $\frac{7}{10}$ $\mathcal{A}$ will compare y against an element x such that $x \geq b/(100\Delta)$.*

Proof. By simple scaling, we can assume $b = 1$. Let p be the probability that $\mathcal{A}$ compares y against a value larger than $1/(100\Delta)$. Let I_{small} be the set of indices i such that $x_i < 1/(100\Delta)$. Let $\mathcal{A}'$ be a deterministic algorithm in CAI model such that:

- $\mathcal{A}'$ is given all the indices in I_{small} (and their corresponding values) except for the index of the minimum. We call these the known values.
- $\mathcal{A}'$ minimizes the probability p' of comparing the y against a value larger than $1/(100\Delta)$.
- $\mathcal{A}'$ finds the minimum value among the unknown values.

Since $p' \leq p$, it suffices to bound p' from below. We do this in the remainder of the proof.

Observe that the expected number of values x_i such that $x_i < 1/(100\Delta)$ is $T/(100\Delta)$. Thus, by Markov's inequality, $\Pr[|I_{\text{small}}| \leq T/(10\Delta)] \geq \frac{9}{10}$. Let's call the event $|I_{\text{small}}| \leq T/(10\Delta)$ the good event. For algorithm $\mathcal{A}'$ all values smaller than $1/(100\Delta)$ except for the minimum are known. Let U be the set of indices of the unknown values. Observe that a value x_i for $i \in U$ is either the minimum or larger than $1/(100\Delta)$, and that $|U| = T - |I_{\text{small}}| + 1 > \frac{9}{10}T$ (using $\Delta \geq 1$) in the good event. Because $\mathcal{A}'$ is a deterministic algorithm, the set U is split into set F of elements that have their first comparison against a known element, and set W of those that are first compared with another element with index in U. Because of the global bound Δ on the fragile complexity of the known elements, we know $|F| < \Delta \cdot |I_{\text{small}}| \leq \Delta T/(10\Delta) = T/10$. Thus, when the good event happens, the probability of the minimum being in W is $|W|/|U| = 1 - |F|/|U| \geq \frac{8}{9}$. Combining this with the probability of the good event, by union bound, the probability of the minimum being compared with a value greater than $1/(100\Delta)$ is at least $1 - (1 - \frac{9}{10}) - (1 - \frac{8}{9}) \geq 7/10$. ◀

Based on the above lemma, our proof idea is the following. Let $G = 100\Delta$. We would like to prove that on average $\mathcal{A}$ cannot avoid comparing the minimum to a lot of elements. In particular, we show that, with constant probability, the minimum will be compared against some value in the range $[G^{-i}, G^{-i+1}]$ for every integer i, $1 \leq i \leq \frac{\log_G n}{2}$. Our lower bound then follows by an easy application of the linearity of expectations. Proving this, however, is a little bit tricky. However, observe that Lemma 8 already proves this for $i = 1$. Next, we use the following lemma to apply Lemma 8 over all values of i, $1 \leq i \leq \frac{\log_G n}{2}$.

▶ **Lemma 9.** *For a value b with $0 < b < 1$, define $p_k = \binom{n}{k} b^i (1-b)^{n-k}$, for $0 \leq k \leq n$. Choosing $x_1, \ldots, x_n$ iid and uniformly in $(0, 1)$ is equivalent to the following: with probability p_k, uniformly sample a set I of k distinct indices in $\{1, \ldots, n\}$ among all the subsets of size k. For each $i \in I$, pick x_i iid and uniformly in $(0, b)$. For each $i \notin I$, pick x_i iid and uniformly in $(b, 1)$.*

Remember that our goal was to prove that with constant probability, the minimum will be compared against some value in the range $[G^{-i}, G^{-i+1}]$ for every integer i, $1 \leq i \leq \frac{\log_G n}{2}$. We

can pick $b = G^{-i+1}$ and apply Lemma 9. We then observe that it is very likely that the set of indices I that we are sampling in Lemma 9 will contain many indices. For every element x_i, $i \in I$, we are sampling x_i independently and uniformly in $(0, b)$ which opens the door for us to apply Lemma 8. Then we argue that Lemma 8 would imply that with constant probability the minimum will be compared against a value in the range $(b/G, b) = (G^{-i}, G^{-i+1})$. The lower bound claim of Theorem 7 then follows by invoking the linearity of expectations.

2.3.2 Lower bound for the fragile complexity of the minimum whp.

By Theorem 4 in Subsection 2.2, SimpleMinimum guarantees that the fragile complexity of the minimum is at most $k = c + \log\log n$ with probability at most $\frac{2^k}{2^{2^k-1}} = \frac{2^c \log n}{n^{2^c-1}}$. Here we show that this is almost optimal.

▶ **Lemma 10.** *For $c > 0$ and any deterministic algorithm $\mathcal{A}$, let $p_{\mathcal{A},f}$ be the probability of the minimum having $\geq f = \log\log n + c - \log\log\log n$ fragile complexity under the uniform input distribution. Then $p_{\mathcal{A},f} \geq n^{-2^{c+1}}$ for $n \geq 2^{2^c}$.*

Proof. Let $k = 2^f = 2^c \log n / \log\log n$ and let S be the set of k smallest input values. Let π be a uniform permutation the input and $\pi(S)$ be the permutation of the elements of S in to π. Observe that $\pi(S)$ is a uniform permutation of the elements of S. We reveal the elements not in S to $\mathcal{A}$. So, $\mathcal{A}$ only needs to find the minimum in $\pi(S)$. By Theorem 2 there is at least one "bad" permutation of S which forces algorithm $\mathcal{A}$ to do $\log k = f$ comparisons on the smallest element. Our lemma follows because the probability of $\pi(S)$ being such a bad permutation is at least

$$\frac{1}{k!} > \frac{1}{k^k} > \frac{1}{(2^c \log n)^k} > \frac{1}{(\log^2 n)^k} = \frac{1}{2^{c+1} \log n} = \frac{1}{n^{2^{c+1}}}.$$ ◀

3 Selection and median

The (n, t)-*selection problem* asks to find the t-th smallest element among n elements of the input. The simplest solution to the (n, t)-selection problem is to sort the input. Therefore, it can be solved in $\mathcal{O}(\log n)$ fragile complexity and $\mathcal{O}(n \log n)$ work by using the AKS sorting network [1]. For comparator networks, both of these bounds are optimal: the former is shown by Theorem 2 (and in fact it applies also to any algorithm) and the latter is shown in Appendix C.1.

In contrast, in this section we show that comparison-based algorithms can do better: we can solve Selection deterministically in $\Theta(n)$ work and $\Theta(\log n)$ fragile complexity, thus, showing a separation between the two models. However, to do that, we resort to constructions that are based on expander graphs. Avoiding usage of the expander graphs or finding a simpler optimal deterministic solution is an interesting open problem (see Section 6). Moreover, in Subsection 3.2 we show that we can do even better by using randomization.

3.1 Deterministic selection

▶ **Theorem 11.** *There is a deterministic algorithm for* Selection *which performs $\mathcal{O}(n)$ work and has $\mathcal{O}(\log n)$ fragile complexity.*

Proof sketch. It suffices to just find the median since by simple padding we can generalize the solution for the (n, t)-selection problem.

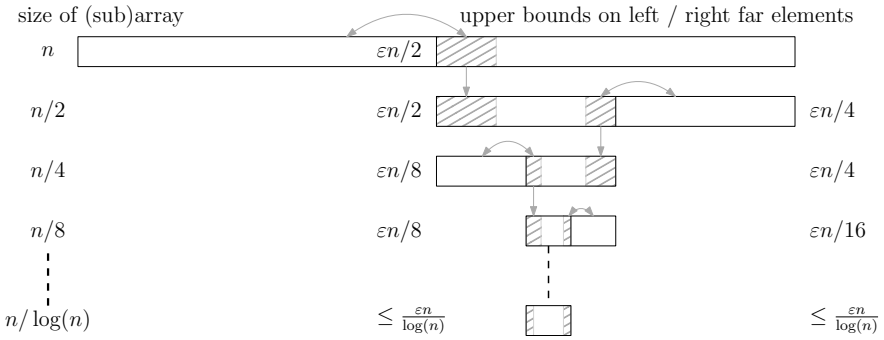

Figure 1 Illustration of the alternating division process using ε-halvers.

We use *ε-halvers* that are the central building blocks of the AKS sorting network. An ε-halver approximately performs a partitioning of an array of size n into the smallest half and the largest half of the elements. More precisely, for any $m \leq n/2$, at most εn of the m smallest elements will end up in the right half of the array, and at most εn of the m largest elements will end up in the left half of the array. Using expander graphs, a comparator network implementing an ε-halver of constant depth can be built [1, 3]. We use the corresponding comparison-based algorithm of constant fragile complexity.

The idea is to use ε-halvers to find roughly $\frac{n}{\log n}$ elements with rank between $(1+\alpha)\frac{n}{2}$ and $(2-\alpha)\frac{n}{2}$ and also roughly $\frac{n}{\log n}$ elements with rank between $\alpha\frac{n}{2}$ and $(1-\alpha)\frac{n}{2}$, for some constant $0 < \alpha < 1$. This is done by repeatedly using ε-halvers but alternating between selecting the left half and the right half (Figure 1). Using these, we filter the remaining elements and discard a constant fraction of them. Then we recurse on the remaining elements. The details are a bit involved, as we have to guarantee that no element accumulates too many comparisons throughout the recursions. We have to do some bookkeeping as well as some additional ideas to provide this guarantee. Details can be found in Appendix C.2. ◀

▶ **Corollary 12.** *There is a deterministic algorithm for partition which performs $\mathcal{O}(n)$ work and has $\mathcal{O}(\log n)$ fragile complexity.*

Proof. At the end of the Selection algorithm, the set of elements smaller (larger) than the median is the union of the respective filtered sets (sets $\mathcal{L}$ and $\mathcal{R}$ in the proof in Appendix C.2) and the first (last) half of the sorted set in the base case of the recursion. Again, simple padding generalizes this to (n,t)-partition for arbitrary $t \neq \frac{n}{2}$. ◀

3.2 Randomized selection

In Appendix C.3 we present the details of an expected work-optimal selection algorithm with a trade-off between the expected fragile complexity $f_{\text{med}}(n)$ of the selected element and the maximum expected fragile complexity $f_{\text{rem}}(n)$ of the remaining elements. In particular, we obtain the following combinations:

▶ **Theorem 13.** *Randomized selection is possible with an expected fragile complexity of the median $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\log\log n)$ and of the remaining elements $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\sqrt{n})$ by Theorem 28 and $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\frac{\log n}{\log\log n})$ and $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\log n)$ by Theorem 29.*

4 Sorting

Recall from Section 1 that the few existing sorting networks with depth $\mathcal{O}(\log n)$ are all based on expanders, while a number of $\mathcal{O}(\log^2 n)$ depth networks have been developed based on merging. Here, we study the power of the mergesort paradigm with respect to fragile complexity. We first prove that any sorting algorithm based on binary merging must have a worst-case fragile complexity of $\Omega(\log^2 n)$. This provides an explanation why all existing sorting networks based on merging have a depth no better than this. We also prove that the standard mergesort algorithm on random input has fragile complexity $\mathcal{O}(\log n)$ with high probability, thereby showing a separation between the deterministic and the randomized situation for binary mergesorts. Finally, we demonstrate that the standard mergesort algorithm has a worst-case fragile complexity of $\Theta(n)$, but that this can be improved to $\mathcal{O}(\log^2 n)$ by changing the merging algorithm to use exponential search. The omitted proofs can be found in Appendix D.

▶ **Theorem 14.** *Any binary mergesort has fragile complexity* $\Omega(\log^2 n)$.

▶ **Theorem 15.** *Standard* MergeSort *with linear merging has a worst-case fragile complexity of* $\Theta(n)$.

▶ **Theorem 16.** *Standard* MergeSort *with linear merging on a randomized input permutation has a fragile complexity of* $\mathcal{O}(\log n)$ *with high probability.*

▶ **Theorem 17.** *Exponential merging of two sequences* $A = (a_1, \ldots, a_n)$ *and* $B = (b_1, \ldots, b_n)$ *has a worst-case fragile complexity of* $\mathcal{O}(\log n)$.

▶ **Corollary 18.** *Applying Theorem 17 to standard* MergeSort *with exponential merging yields a fragile complexity of* $\mathcal{O}(\log^2 n)$ *in the worst-case.*

5 Constructing binary heaps

▶ **Observation 1.** *The fragile complexity of the standard binary heap construction algorithm of Floyd [10] is* $\mathcal{O}(\log n)$.

The above observation is easy to verify (see Appendix E). We note that this fragile complexity is optimal by Theorem 2, since Heap Construction is stronger than Minimum. Brodal and Pinotti [6] showed how to construct a binary heap using a comparator network in $\Theta(n \log\log n)$ size and $\mathcal{O}(\log n)$ depth. They also proved a matching lower bound on the size of the comparator network for this problem. This, together with Observation 1 and the fact that Floyd's algorithm has work $\mathcal{O}(n)$, gives a separation between work of fragility-optimal comparison-based algorithms and size of depth-optimal comparator networks for Heap Construction.

6 Conclusions

In this paper we introduced the notion of fragile complexity of comparison-based algorithms and we argued that the concept is well-motivated because of connections both to real world situations (e.g., sporting events), as well as other fundamental theoretical concepts (e.g., sorting networks). We studied the fragile complexity of some of the fundamental problems and

revealed interesting behavior such as the large gap between the performance of deterministic and randomized algorithms for finding the minimum. We believe there are still plenty of interesting and fundamental problems left open. Below, we briefly review a few of them.

- The area of comparison-based algorithms is much larger than what we have studied. In particular, it would be interesting to study "geometric orthogonal problems" such as finding the maxima of a set of points, detecting intersections between vertical and horizontal line segments, *kd*-trees, axis-aligned point location and so on. All of these problems can be solved using algorithms that simply compare the coordinates of points.
- Is it possible to avoid using expander graphs to obtain simple deterministic algorithms to find the median or to sort?
- Is it possible to obtain a randomized algorithm that finds the median where the median suffers $O(1)$ comparisons on average? Or alternatively, is it possible to prove a lower bound? If one cannot show a $\omega(1)$ lower bound for the fragile complexity of the median, can we show it for some other similar problem?

Acknowledgments

We thank Steven Skiena for posing the original problem, and we thank Michael Bender and Pat Morin for helpful discussions.

References

1 M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. STOC '83, pages 1–9. ACM, 1983. URL: http://doi.acm.org/10.1145/800061.808726.

2 M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, Mar 1983. URL: https://doi.org/10.1007/BF02579338.

3 M. Ajtai, J. Komlós, and E. Szemerédi. Halvers and expanders. In IEEE, editor, *FOCS'92*, pages 686–692, Pittsburgh, PN, October 1992. IEEE Computer Society Press.

4 V. E. Alekseev. Sorting algorithms with minimum memory. *Kibernetika*, 5(5):99–103, 1969.

5 K. E. Batcher. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

6 G. Stølting Brodal and M. C. Pinotti. Comparator networks for binary heap construction. In *Proc. 6th Scandinavian Workshop on Algorithm Theory*, volume 1432 of *LNCS*, pages 158–168. Springer Verlag, Berlin, 1998. doi:10.1007/BFb0054364.

7 V. Chvátal. Lecture notes on the new AKS sorting network. Technical Report DCS-TR-294, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1992, October.

8 R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

9 M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989, October.

10 R. W. Floyd. Algorithm 245: Treesort. *Commun. ACM*, 7(12):701, December 1964. URL: http://doi.acm.org/10.1145/355588.365103.

11 M. T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In David B. Shmoys, editor, *STOC'14*, pages 684–693. ACM, 2014. URL: http://dl.acm.org/citation.cfm?id=2591796.

12 S. Janson. Tail bounds for sums of geometric and exponential variables. *Statistics & Probability Letters*, 135:1–6, 2018. doi:https://doi.org/10.1016/j.spl.2017.11.017.

13 S. Jimbo and A. Maruoka. A method of constructing selection networks with $O(\log n)$ depth. *SIAM Journal on Computing*, 25(4):709–739, 1996.

14 I. Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2(2-3):205–211, 1992.

15 B. Parker and I. Parberry. Constructing sorting networks from k-sorters. *Information Processing Letters*, 33(3):157–162, 30 November 1989.

16 M. S. Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5(1):75–92, 1990.

17 N. Pippenger. Selection networks. *SIAM Journal on Computing*, 20(5):878–887, 1991.

18 V. R. Pratt. *Shellsort and Sorting Networks*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1972.

19 J. I. Seiferas. Sorting networks of logarithmic depth, further simplified. *Algorithmica*, 53(3):374–384, 2009.

20 S.Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *BAMS: Bulletin of the American Mathematical Society*, 43:439–561, 2006.

21 S. P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

22 A. Yao and F. F. Yao. Lower bounds on merging networks. *J. ACM*, 23(3):566–571, 1976.

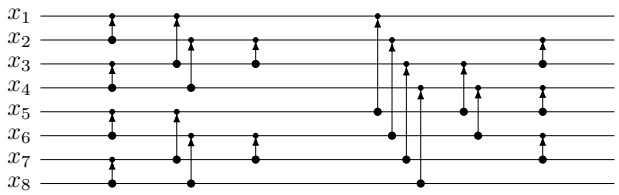

Figure 2 Batcher's Odd-Even-Mergesort network [5]: 8 inputs, depth $f(8)$=6 and size $w(8)$=19.

A Definitions

A.1 Comparator networks

A comparator network $\mathcal{N}$ is constructed of *comparators* each consisting of two inputs and two (ordered) outputs. The value of the first output is the minimum of the two inputs and the value of the second output is the maximum of the two inputs. By this definition a comparator network $\mathcal{N}_n$ on n inputs also consists of n outputs. Figure 2 demonstrates a common visualization of comparator networks with values as horizontal wires and comparators represented by vertical arrows between pairs of wires. Each arrow points to the output that returns the minimum input value. Inputs are on the left and outputs on the right. The *size* of the comparator network is defined as the number of comparators in it, while its *depth* is defined as the number of comparators on the longest path from an input to an output.

We note that a comparator network is straightforward to execute as a comparison-based algorithm by simulating its comparators sequentially from left to right (see Figure 2), breaking ties arbitrarily. If the network has depth $f(n)$ and size $g(n)$, the comparison-based algorithm clearly has fragile complexity $f(n)$ and work $g(n)$.

A.2 Networks for problems

We define the set of inputs to the comparator network $\mathcal{N}$ by $\mathcal{I}$ and the outputs by $\mathcal{N}(\mathcal{I})$. We use the notation $\mathcal{N}(\mathcal{I})^i$ for $0 \leq i \leq n-1$, to represent the i-th output and $\mathcal{N}(\mathcal{I})^{i:j}$ for $0 \leq i \leq j \leq n-1$ to represent the ordered subset of the i-th through j-th outputs.

An (n)-*sorting network* is a comparator network $\mathcal{N}_n$ such that $\mathcal{N}_n(\mathcal{I})^t$ carries the t-th smallest input value for all t. We say such a network solves the (n)-*sorting problem*.

An (n,t)-*selection network* is a comparator network $\mathcal{N}_{n,t}$ such that $\mathcal{N}_{n,t}(\mathcal{I})^0$ carries the t-th smallest input value. We say such a network solves the (n,t)-*selection problem*.

An (n,t)-*partition network* is a comparator network $\mathcal{N}_{n,t}$, such that $\mathcal{N}_{n,t}(\mathcal{I})^{0:t-1}$ carry the t smallest input values.[4] We say such a network solves the (n,t)-*partition problem*.

Clearly, an (n,t)-selection problem is asymptotically no harder than (n,t)-partition problem: let $\mathcal{N}^{\downarrow}_{n,t}(\mathcal{I})$ denote an (n,t)-partition network with all comparators reversed; then $\mathcal{N}'_{n,t}(\mathcal{I}) = \mathcal{N}^{\downarrow}_{t,t-1}(\mathcal{N}_{n,t}(\mathcal{I}))$ is an (n,t)-selection network. However, the converse is not clear: given a value of the t-th smallest element as one of the inputs, it is not obvious how to construct an (n,t)-partition network with smaller size or depth. In Section 3 we will show that the two problems are equivalent: every (n,t)-selection network also solves the (n,t)-partition problem.

[4] Brodal and Pinotti [6] call it an (n,t)-selection network, but we feel (n,t)-partition network is a more appropriate name.

A.3 Rank

Given a set X, the rank of some element e in X, denoted by $rank_X(e)$, is equal to the size of the subset of X containing the elements that are no larger than e. When the set X is clear from the context, we will omit the subscript X and simply write $rank(e)$.

B Details of Finding the Minimum

B.1 Deterministic Algorithm for Finding the Minimum

▶ **Theorem 2.** *The fragile complexity of finding the minimum of n elements is $\lceil \log n \rceil$.*

Proof. The upper bound follows trivially from the application of a balanced tournament tree. We thus focus on the lower bound. Let S be a set of n elements, and $\mathcal{A}$ be a deterministic comparison-based algorithm that finds the minimum element of S. We describe an adversarial strategy for resolving the comparisons made by $\mathcal{A}$ that leads to the lower bound.

Consider a directed graph G on n nodes corresponding to the elements of S. With a slight abuse of notation, we use the same names for elements of S and the associated nodes in graph G. The edges of G correspond to comparisons made by $\mathcal{A}$, and are either black or red. Initially G has no edges. If $\mathcal{A}$ compares two elements, we insert a directed edge between the associated nodes pointing toward the element declared smaller by the adversarial strategy. Algorithm $\mathcal{A}$ correctly finds the minimum element if and only if, upon termination of $\mathcal{A}$, the resulting graph G has only one sink node.

Consider the following adversarial strategy to resolve comparisons made by $\mathcal{A}$: if both elements are sinks in G, the element that has already participated in more comparisons is declared smaller; if only one element is a sink in G, this element is declared smaller; and if neither element is a sink in G, the comparison is resolved arbitrarily (while conforming to the existing partial order).

We color an edge in G red if it corresponds to a comparison between two sinks; otherwise, we color the edge black. For each element i, consider its in-degree d_i and the number of nodes r_i in G (incl. i itself) from which i is reachable by a directed path of only red edges.

We show by induction that $r_i \leq 2^{d_i}$ for all sinks in G. Initially, $r_i = 1 \leq 1 = 2^{d_i}$ for all i. Let algorithm $\mathcal{A}$ compare two elements i and j, where i is a sink, and let the adversarial strategy declare i to be smaller than j. Then, the resulting in-degree of i is $d_i + 1$. If the new edge is black, the number of nodes from which i is reachable via red edges does not change, and the inequality holds trivially. If the new edge is red, the resulting number of nodes from which i is reachable is $r_i + r_j \leq 2^{d_i} + 2^{d_j} \leq 2^{d_i+1}$. Therefore, when $\mathcal{A}$ terminates with the only sink v in G, which represents the minimum element, with degree $d_v \geq \lceil \log r_v \rceil$. The theorem follows by observing that a tournament tree is an instance where $r_v = n$. ◀

B.2 Randomized Lower Bounds for Finding Minimum

In this section we present the omitted proofs from Section 2.3

▶ **Lemma 19.** *For a value b with $0 < b < 1$, define $p_k = \binom{n}{k} b^i (1-b)^{n-k}$, for $0 \leq k \leq n$. Choosing $x_1, \ldots, x_n$ iid and uniformly in $(0,1)$ is equivalent to the following: with probability p_k, uniformly sample a set I of k distinct indices in $\{1, \ldots, n\}$ among all the subsets of size k. For each $i \in I$, pick x_i iid and uniformly in $(0, b)$. For each $i \notin I$, pick x_i iid and uniformly in $(b, 1)$.*

Proof. It is easy to see that choosing $x_1, \ldots, x_n$ iid uniformly in $(0, 1)$ is equivalent to choosing a point X uniformly at random inside an n dimensional unit cube $(0, 1)^n$. Therefore, we will prove the equivalence between (i) the distribution defined in the lemma, and (ii) choosing such point X.

Let Q be the n-dimensional unit cube. Subdivide Q into 2^n rectangular region defined by the Cartesian product of intervals $(0, b)$ and $(b, 1)$, i.e., $\{(0, b), (b, 1)\}^n$ (or alternatively, bisect Q with n hyperplanes, with the i-th hyperplane perpendicular to the i-th axis and intersecting it at coordinate equal to b).

Consider the set R_k of rectangles in $\{(0, b), (b, 1)\}^n$ with exactly k sides of length b and $n - k$ sides of length $1 - b$. Observe that for every choice of k (distinct) indices $i_1, \ldots, i_k$ out of $\{1, \ldots, n\}$, there exists exactly one rectangle r in R_k such that r has side length b at dimensions $i_1, \ldots, i_k$, and all the other sides of r has length $n - k$. As a result, we know that the number of rectangles in R_k is $\binom{n}{k}$ and the volume of each rectangle in R_k is $b^k(1 - b)^k$. Thus, if we choose a point X randomly inside Q, with probability p_k it will fall inside a rectangle r in R_k; furthermore, conditioned on this event, the dimensions $i_1, \ldots, i_k$ where r has side length b is a uniform subset of k distinct indices from $\{1, \ldots, n\}$. ◀

We are ready to prove that the minimum element will have $\Omega(\log_\Delta n)$ comparisons on average.

▶ **Theorem 7.** *In any randomized minimum finding algorithm with fragile complexity of at most Δ for any element, the expected fragile complexity of the minimum is at least $\Omega(\log_\Delta n)$.*

Proof. First, observe that we can assume $n \geq (100,000\Delta)^2$ as otherwise we are aiming for a trivial bound of $\Omega(1)$.

We create an input set of n values $x_1, \ldots, x_n$ where each x_i is chosen iid and uniformly in $(0, 1)$. Let $G = 100\Delta$. Consider an integer i such that $1 \leq i < \frac{\log_G n}{2}$. We are going to prove that with constant probability, the minimum will be compared against a value in the range (G^{-i}, G^{-i+1}), which, by linearity of expectation, shows the stated $\Omega(\log_\Delta n)$ lower bound for the fragile complexity of the minimum.

Consider a fixed value of i. Let S be the set of indices with values that are smaller than G^{-i+1}. Let p be the probability that $\mathcal{A}$ compares the minimum against an x_j with $j \in S$ such that $x_j \geq G^{-i}$. To prove the theorem, it suffices to prove that p is lower bounded by a constant. Now consider an algorithm $\mathcal{A}'$ that finds the minimum but for whom all the values other than those in S have been revealed and furthermore, assume $\mathcal{A}'$ minimizes the probability of comparing the minimum against an element $x \geq G^{-i}$ (in other words, we pick the algorithm which minimizes this probability, among all the algorithms). Clearly, $p' \leq p$. In the rest of the proof we will give a lower bound for p'.

Observe that $|S|$ is a random variable with binomial distribution. Hence $\mathbb{E}[|S|] = nG^{-i+1} > \sqrt{n}$ where the latter follows from $i < \frac{\log_G n}{2}$. By the properties of the binomial distribution we have that $\Pr\left[|S| < \frac{\mathbb{E}[|S|]}{100}\right] < \frac{1}{10}$. Thus, with probability at least $\frac{9}{10}$, we will have the "good" event that $|S| \geq \frac{\mathbb{E}[|S|]}{100} \geq \frac{\sqrt{n}}{100}$.

In case of the good event, Lemma 9 implying that conditioned on S being the set of values smaller than G^{-i+1}, each value x_j with $j \in S$ is distributed independently and uniformly in the range $(0, G^{-i+1})$. As a result, we can now invoke Lemma 8 on the set S with $T = |S|$. Since $n \geq (100,000\Delta)^2$ we have $T = |S| \geq \frac{\sqrt{n}}{100} \geq \frac{100,000\Delta}{100}$. By Lemma 8, with probability at least $\frac{7}{10}$, the minimum will be compared against a value that is larger than G^{-i}.

Thus, by law of total probability, it follows that in case of a good event, with probability $\frac{7}{10}$ the minimum will be compared to a value in the range (G^{-i}, G^{-i+1}). However, as the good event happens with probability $\frac{9}{10}$, it follows that with probability at least $1-(1-\frac{7}{10})-(1-\frac{9}{10}) = \frac{6}{10}$, the minimum will be compared against a value in the range (G^{-i}, G^{-i+1}). ◀

C Details of Selection

C.1 Deterministic selection via comparator networks

In this section, we discuss the (n, t)-selection problem in the setting of comparator networks. We present an upper and a matching lower bound on the size of a comparator network solving the (n, t)-selection problem. In the next section, we consider the problem in the setting of comparison-based algorithms and give an algorithm for selection with the same fragile complexity as in the case of comparator networks, but with total work that is asymptotically smaller. Combined, this shows a separation in power between the two models.

To begin, observe that the (n, t)-selection problem can be solved by sorting the input. Therefore, the (n, t)-selection problem can be solved using a comparator network of size $\mathcal{O}(n \log n)$ and depth $\mathcal{O}(\log n)$ by using the AKS sorting network [2]. Consequently, the (n, t)-selection problem can be solved with $\mathcal{O}(\log n)$ fragile complexity and $\mathcal{O}(n \log n)$ work.

Next, we show that the size of the (n, t)-selection network using the AKS network is asymptotically tight. Before we present the lower bound theorem, we need to introduce some notation and prove two auxiliary lemmas.

Given a set X, we say two elements $x_i, x_j \in X$ are *rank-neighboring* if $|rank(x_i) - rank(x_j)| = 1$. We say a permutation $\hat{\pi}(X)$ is *rank-neighboring* if an ordered sequence X and $\hat{\pi}(X)$ differ in only two elements and these two elements are rank-neighboring. Observe that any permutation $\pi(X)$ is a composition of some number of rank-neighboring permutations.

We define the *signature* of an ordered sequence X with respect to an integer a to be a function $\sigma : \mathbb{Z} \times \mathbb{Z}^{n-1} \to \{0, 1\}^{n-1}$, such that $\sigma(a, (x_1, \ldots, x_{n-1})) = (y_1, \ldots, y_{n-1})$ and for all $1 \leq i \leq n - 1$:

$$y_i = \begin{cases} 0 & \text{if } x_i \leq a \\ 1 & \text{if } x_i > a \end{cases}$$

▶ **Lemma 20.** *For any totally ordered set X of size n, any (n, t)-selection network $\mathcal{N}_{n,t}$, and for any rank-neighboring permutation $\hat{\pi}$: $\sigma(\mathcal{N}_{n,t}(X)) = \sigma(\mathcal{N}_{n,t}(\hat{\pi}(X)))$.*

Proof. Consider the two inputs x_i and x_j in which X and $\hat{\pi}(X)$ differ. During the computation the input values traverse the network until they reach the outputs. During the computation of the network $\mathcal{N}_{n,t}(X)$ element x_i (resp., x_j) starts at the i-th (resp., j-th) input and reaches the i'-th (resp., j'-th) output. During the computation of the network $\mathcal{N}_{n,t}(\hat{\pi}(X))$, element x_i (resp., x_j) starts at the j-th (resp., i-th) input. Let us determine which outputs they reach.

Consider the two paths P_i and P_j that x_i and x_j, respectively, traverse during the computation of $\mathcal{N}_{n,t}(X)$. Since $\hat{\pi}$ is a rank-neighboring permutation, the outputs of every comparator C are the same for $\mathcal{N}_{n,t}(X)$ and $\mathcal{N}_{n,t}(\hat{\pi}(X))$ except for the (set of) comparator(s) C^*, whose two inputs are x_i and x_j. Thus, throughout the computation of $\mathcal{N}_{n,t}(\hat{\pi}(X))$, x_i and x_j only traverse the edges of the paths $P_i \cup P_j$, i.e., the outputs of $\mathcal{N}_{n,t}(\hat{\pi}(X))$ are the same as the outputs of $\mathcal{N}_{n,t}(X)$ everywhere except for, possibly, at the i'-th and j'-th output.

If $rank(x_i) < t$ and $rank(x_j) < t$, or $rank(x_i) > t$ and $rank(x_j) > t$, the signatures of these two outputs are the same. Consequently, $\sigma(\mathcal{N}_{n,t}(X)) = \sigma(\mathcal{N}_{n,t}(\hat{\pi}(X)))$. Otherwise, without loss of generality, let $rank(x_i) = t$ (the case of $rank(x_j) = t$ is symmetric). Then $i' = 0$, and $\mathcal{N}_{n,t}(X)^0 = \mathcal{N}_{n,t}(\hat{\pi}(X))^0 = x_i$. Then, $\mathcal{N}_{n,t}(X)^{j'} = \mathcal{N}_{n,t}(\hat{\pi}(X))^{j'} = x_j$, and it follows that $\sigma(\mathcal{N}_{n,t}(X)) = \sigma(\mathcal{N}_{n,t}(\hat{\pi}(X)))$. ◀

▶ **Lemma 21.** *An (n,t)-selection network can be turned into an (n,t)-partition network.*

Proof. Since every permutation $\pi(X)$ can be obtained from X by a sequence of rank-neighboring permutations, it follows from Lemma 20 that $\sigma(\mathcal{N}_{n,t}(\pi(X))) = \sigma(\mathcal{N}_{n,t}(X))$, i.e., for every permutation of the inputs in the (n,t)-selection network, the same subset of outputs carry the values that are at most t. Thus, the (n,t)-partition network can be obtained from the (n,t)-selection network by reordering (re-wiring) the outputs such that the ones with signature 0 are the first t outputs. ◀

▶ **Theorem 22.** *An (n,t)-selection network for any $t \leq n/2$ has size $\Omega((n-t)\log(t+1))$.*

Proof. Alekseev [4] showed the $\Omega((n-t)\log(t+1))$ lower bound for the size of an (n,t)-partition network, and, by Lemma 21, every (n,t)-selection network also solves the (n,t)-partition problem. ◀

▶ **Corollary 23.** *The size of a comparator network that finds the median of n elements is $\Omega(n \log n)$.*

C.2 Deterministic Selection Algorithms

▶ **Theorem 11.** *There is a deterministic algorithm for* Selection *which performs $\mathcal{O}(n)$ work and has $\mathcal{O}(\log n)$ fragile complexity.*

Proof. Below, we give an algorithm for the median problem. By simple padding of the input, median solves the (n,t)-selection problem for arbitrary $t \neq \frac{n}{2}$.

A central building block from the AKS sorting network is an *ε-halver*. An ε-halver approximately performs a partitioning of an array of size n into the smallest half and the largest half of the elements. More precisely, for any $m \leq n/2$, at most εn of the m smallest elements will end up in the right half of the array, and at most εn of the m largest elements will end up in the left half of the array. Using expander graphs, a comparator network implementing an ε-halver in constant depth can be built [1, 3]. We use the corresponding comparison-based algorithm of constant fragile complexity.

We make the convention that when using ε-halvers, the larger elements placed at the right half and the smaller elements are placed at the left half. We first use an ε-halver on the input array S of length n, dividing it into two subarrays of length $n/2$. Let's call the right half S_1. As an ε-halver does an "approximate" partitioning, S_1 will contain l of the smallest $n/2$ elements by mistake, however, it will contain $n/2 - l$ of the $n/2$ largest with $l \leq \varepsilon n/2$. From this point forward, we apply ε-halvers but alternate between picking the right and then left half. In particular, we apply an ε-halver to S_i (starting from $i = 1$), and set S_{i+1} to be the left half (resp. right half) of the resulting partition if i is odd (resp. if i is even). See Figure 1.

We stop the process after $k = 2\lfloor \frac{\log\log n}{2} \rfloor$ steps; we choose an even k to simplify the upcoming discussions. This results in a set S_k of size $\Theta(n/\log n)$. We sort S_k using an AKS-based sorting algorithm, which takes time $\mathcal{O}(n)$ and has fragile complexity $\mathcal{O}(\log n)$, and we then extract the middle $|S_k|/2$ of these sorted elements as the set R_P ("right pivots").

We claim the rank of every element in R_P is between $(1+\alpha)\frac{n}{2}$ and $(2-\alpha)\frac{n}{2}$ for some absolute constant $0 < \alpha < 1$. To prove this claim, we use the properties of ε-halvers. Consider S_i: we partition S_i into two sets, and select S_{i+1} to be either the left or the right half, depending on the parity of i. Assume S_{i+1} is selected to be the right half (similarly, left half). We mark an element of S_{i+1} as a *left mistake* (similarly, a *right mistake*), if it is among the $|S_i|/2$ smaller (similarly, larger) elements of S_i. We say an element of S_{i+1} is *good*.

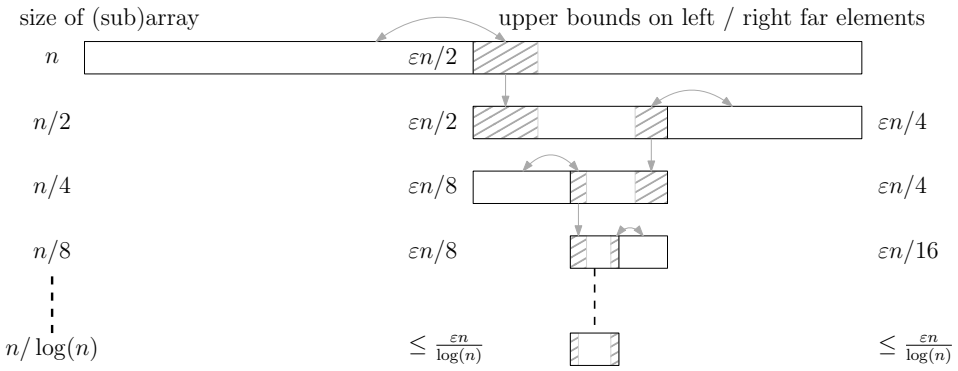

Figure 1 Illustration of the alternating division process using ε-halvers.

Now assume $\varepsilon < 1/64$. We can now use simple induction to show the following: If S_i contains t_ℓ left mistakes and t_r right mistakes, then the left mistakes are the t_ℓ smallest element of S_i and the right mistakes are the t_r largest elements in S_i and furthermore, S_i contains at most $2\varepsilon|S_i|$ left mistakes and $2\varepsilon|S_i|$ right mistakes.

These claims are obviously true for S_1 and S_2 thus assume the hold for S_i; we would like to show that they also hold for S_{i+2}. W.l.o.g, assume S_{i+1} is selected to be the right half after partitioning S_i using an ε-halver. Consider the sorted order of S_i and in particular, the set L containing $|S_i|/2$ largest elements in S_i. As $\varepsilon < 1/64$, it follows that $2\varepsilon|S_i| < |S_i|/2$ and as a result, L contains no left mistakes by our induction hypothesis since all the left mistakes are among the $|S_i|/2$ smallest elements of $|S_i|$. However, L contains all the up to $2\varepsilon|S_i|$ right mistakes. By properties of an ε-halver, S_{i+1} has at most $\varepsilon|S_i|$ elements that do not belong in L. Thus, S_{i+1} contains at most $\varepsilon|S_i|$ left mistakes and at most $2\varepsilon|S_i|$ right mistakes. Crucially, notice that S_{i+2} is obtained by using an ε-halver on L and selecting *the left* half of the resulting partition. A similar argument now shows that S_{i+2} has $\varepsilon|S_i| = 2\varepsilon|S_{i+1}|$ left mistakes but $\varepsilon|S_{i+1}|$ right mistakes. This concludes the inductive proof of our claims. Observe that, as a corollary, at least $1 - 4/\varepsilon > 1/2$ fraction of the elements in S_i are good.

After i steps, we have an array part S_i of length $n_i = n/2^i$ with at most $2\varepsilon n_i$ left mistakes and at most $2\varepsilon n_i$ right mistakes. For a moment assume there are no mistakes in any of the partitioning steps done by ε-halvers. An easy inductive proof implies that in this case, the rank of the elements in S_i, for even i, are between $A_i = (1+\frac{4^{i-1}-1}{34^{i-1}})\frac{n}{2}$ and $B_i = (1+\frac{4^{i-1}+3}{34^{i-1}})\frac{n}{2}$. The claim is clearly true for $i = 2$ and it can be verified for $i + 2$: there are $\frac{n}{2}\frac{1}{4^{i-1}}$ elements between the aforementioned ranks and thus, the ranks of the elements in S_{i+2} will between $A_i + \frac{n}{2}\frac{1}{4^i}$ and $A_i + 2\frac{n}{2}\frac{1}{4^i}$ (partition the range between A_i and B_i into four equal chunks and pick the second chunk). Now observe that $A_i + \frac{n}{2}\frac{1}{4^i} = (1 + \frac{4^{i-1}-1}{34^{i-1}} + \frac{1}{4^i})\frac{n}{2} = (1 + \frac{4^i-1}{34^i})\frac{n}{2}$ and $A_i + 2\frac{n}{2}\frac{1}{4^i} = (1 + \frac{4^{i-1}-1}{34^{i-1}} + 2\frac{1}{4^i})\frac{n}{2} = (1 + \frac{4^i+3}{34^i})\frac{n}{2}$.

However, ε-halvers will likely make plenty of mistakes. Nonetheless, observe that every mistake made by an ε-halver can only change the rank of a good element x in S_k by one: each mistake either involves placing an element that is actually smaller than x to the right of x or an element that is larger than x to the left of x. Furthermore, observe that the total number of elements marked as a mistake is bounded by a geometric series:

$$\sum_{i=1}^{\infty} 4\varepsilon\frac{n}{2^i} \leq 4\varepsilon n < \frac{n}{16}.$$

This in turn implies that the rank of the good elements in S_k is between

$$(1+\frac{4^{k-1}-1}{34^{k-1}})\frac{n}{2}-\frac{n}{16}=(1+\frac{1}{3}-\frac{1}{34^{k-1}}-\frac{1}{8})\frac{n}{2}>(1+\frac{1}{8})\frac{n}{2} \tag{4}$$

and

$$(1+\frac{4^{k-1}+3}{34^{k-1}})\frac{n}{2}+\frac{n}{16}=(1+\frac{1}{3}+\frac{1}{4^{k-1}}+\frac{1}{8})\frac{n}{2}<(1+\frac{7}{8})\frac{n}{2}. \tag{5}$$

Remember that S_k contains $\Theta(n/\log n)$ good elements. We might not know exactly which elements of S_k are good but we know that the middle $|S_k|/2$ elements are certainly good. As discussed, we select these elements as our pivot set R_P. We now compare all elements y of S with some element x from R_P. We evenly distribute the comparisons such that every element of R_P gets compared against at most $\lceil n/|R_P|\rceil = O(\log n)$ elements. If $x \geq y$, we mark y by $\mathcal{R}$. An element marked by $\mathcal{R}$ has rank in S of at least $(1+1/8)n/2$ by Eq. 4, and thus it is guaranteed not to be the median. By Eq. 5, we mark at least $n/16$ elements by $\mathcal{R}$.

We then perform a symmetric process in the first half of S, leading to a set $\overline{S}_k$ of size $\Theta(n/\log n)$ which has a subset L_P (for "left pivots") of $\Theta(n/\log n)$ elements whose rank in S are between $\frac{1}{8}\cdot\frac{n}{2}$ and $\frac{7}{8}\cdot\frac{n}{2}$. We compare all elements y of S with some element x' from L_P and as before distribute the comparisons evenly among the elements of L_P. If $y < x'$, we mark y by $\mathcal{L}$. And as before, an element marked by $\mathcal{L}$ is smaller than the median and at least $n/16$ elements are marked by $\mathcal{L}$.

We can discard an element of mark $\mathcal{L}$ together with an element of mark $\mathcal{R}$ as doing so will not change the median among the remaining elements, as we are discarding an element larger than the median and an element smaller than the median. As a result, we can discard $n/8$ elements.

At first glance, it might feel like we are done. However, we cannot safely recurse on the remaining elements as the elements in S_k and $\overline{S}_k$ have already incurred too many comparisons. This results in a slight complication but it can be averted as follows.

At the beginning we have an input S of n elements. Define $S^{(0)} = S$ and $n^{(0)} = n$ as the top level of our recursion. At depth i of the recursion, we have a set $S^{(i)}$ containing $n^{(i)}$ elements. We then select sets $S_k^{(i)}$ and $\overline{S}_k^{(i)}$ (note that k is also a function of i but to reduce the clutter in the notation, we just use k) and discard $n^{(i)}/8$ elements of $S^{(i)}$ using the above mentioned procedure. The set $S^{(i+1)}$ is defined as the set containing the remaining elements, *excluding* the elements of the sets $S_k^{(i)}$ and $\overline{S}_k^{(i)}$; the elements of these sets are not pruned but they are put aside and we will handle them later.

We stop the recursion as soon as we reach a recursion depth j with $n^{(j)} \leq n/\log n$. At this point, we simply union all the elements in the sets $S_k^{(i)}$ and $\overline{S}_k^{(i)}$, $1 \leq i < j$ and $S^{(j)}$ into a final set $\mathcal{S}$, sort $\mathcal{S}$ and then report the median of $\mathcal{S}$. As discussed, pruning elements does not change the median and thus the reported median is correct. Note that since we always prune a fraction of the elements at each recursive step, we have $j = O(\log\log n)$.

We now analyze the fragile complexity. Invoking an ε-halver incurs $\mathcal{O}(1)$ comparisons on each element, so a full division process incurs $\mathcal{O}(\log\log n)$ comparisons on each element of $S^{(i)}$. Over $j = O(\log\log n)$ recursions, this adds up to $O((\log\log n)^2) = O(\log n)$. Next, the elements in $S_k^{(i)}$ and $\overline{S}_k^{(i)}$ are sorted but they are not sent to the next recursion step so this is simply a one-time cost. These elements suffer an additional $\Theta(\log n)$ comparisons during the pruning phase (to be precise, a subset of them that are selected as "left pivot" or "right pivot" elements) but this cost is also only suffered once. Finally, the remaining elements participate in a final sorting round. Thus, each element participates in $O(\log n)$ comparisons in the worst-case.

Thus, it remains to analyze the work. Invoking an ε-halver incurs a linear number of comparisons. However, as we prune at least a fraction of the elements at each step, we have $n^{(i)} \leq n(\frac{7}{8})^i$. Thus, the total amount of work done by ε-halvers is linear. The same holds for the sorting of $S_k^{(i)}$ and $\overline{S}_k^{(i)}$ as their sizes is bounded by $O(n^{(i)}/\log n^{(i)})$ which forms a geometrically decreasing series. Finally, observe that $|\mathcal{S}| = O(n/\log n)$ since $S^{(j)} = O(n/\log n)$ and $\sum_{i=0}^{j} |S_k^{(i)}| + |\overline{S}_k^{(i)}| = O(n/\log n)$. This implies, we can sort $\mathcal{S}$ in $\mathcal{O}(n)$ time as well. ◀

C.3 Randomized Selection Algorithms

procedure RMEDIAN($X = \{x_1, \ldots, x_n\}, k(n), d(n)$) ▷ Sampling phase
  Randomly sample k elements from X, move them into an array S, sort S with AKS
  Distribute S into buckets $L_b, L_{b-1}, \ldots L_1, C, R_1, \ldots, R_{b-1}, R_b$ as follows:
    set $n_0 = 2\sqrt{k \log n}, n_1 = 3\sqrt{k \log n}, n_i = d \cdot n_{i-1}$
    $C = S[k/2{-}n_0 : k/2{+}n_0]$ median candidates
    $L_i = S[k/2{-}n_i : k/2 - n_{i-1}]$ buckets of elements presumed smaller than median
    $R_i = S[k/2{+}n_{i-1} : k/2{+}n_i]$ buckets of elements presumed larger than median
  ▷ Probing phase
  for $x_i \in X \setminus S$ in random order
    for $j \in [b{-}1, \ldots, 1]$ in order
      $x_A \leftarrow$ arbitrary element in L_j with fewest compares
      $c \leftarrow 1$ if x_A is marked else 2
      if $x_i < x_A$
        add x_i as new pivot to L_{j+c} if $j < b - c$ and mark it, otherwise discard x_i
        stop processing x_i
      $x_B \leftarrow$ arbitrary element in R_j with fewest compares
      $c \leftarrow 1$ if x_B is marked else 2
      if $x_i > x_B$
        add x_i as new pivot to R_{j+c} if $j < b - c$ and mark it, otherwise discard x_i
        stop processing x_i
        ▷ By now it is established that $S[k/2 - n_1] \leq x_i \leq S[k/2 + n_1]$
    add x_i as a median candidate to C
  if $\max(\sum_i |L_i|, \sum_i |R_i|) > n/2$ ▷ Partitioning too imbalanced ⇒ median not in C
    return DETMEDIAN(X)
  if $|C| < \log^4 n$
    sort C with AKS and return median
  $k = \sum_i (|L_i| - |R_i|)$
  if $k > 0$
    add k arbitrary elements from $\bigcup_i R_i$ to C
  else
    add k arbitrary elements from $\bigcup_i L_i$ to C
  return RMEDIAN(C, k(n), b(n))

Just like with the deterministic approach in Section 3.1 / C.2 we restrict ourselves to the special case of median finding. The general (n, t)-selection problem can be solved by initially adding an appropriate number of distinct dummy elements. Note that comparisons of real elements with dummy elements do not contribute to the fragile complexity since we know that all dummy elements are either larger or smaller than all real elements depending on the

value of t. Similarly, the comparison between dummy elements comes for free.

We present RMedian an expected work-optimal median selection algorithm with a trade-off between the expected fragile complexity $f_{\text{med}}(n)$ of the median element and the maximum expected fragile complexity $f_{\text{rem}}(n)$ of the remaining elements. By adjusting a parameter affecting the trade-off, we can vary $\langle \mathbb{E}[f_{\text{med}}(n)], \max \mathbb{E}[f_{\text{rem}}(n)] \rangle$ in the range between $\langle \mathcal{O}(\log\log n), \mathcal{O}(\sqrt{n}) \rangle$ and $\langle \mathcal{O}(\log n/\log\log n), \mathcal{O}(\log n) \rangle$ (see Theorems 28 and 29).

The algorithm (see pseudo code above) takes a totally ordered set X as input. It draws a set S of $k(n)$ random samples, sorts them and subsequently uses the items in S as pivots to identify a set $C \subset X$ of values around the median, such that an equal number of items smaller and greater than the median are excluded from C. Finally, it finds the median of C, which is the median of the initial input, recursively.

The recursion reaches the base case when the input is of size $\mathcal{O}(\text{polylog}\, n)$, at which point it can be sorted to trivially expose the median. RMedian employs two sets L_1 and R_1 of $n_1 = \mathcal{O}(\sqrt{k \log n})$ pivots almost surely below and above the median respectively. All candidates for C are compared to one item in L_1 and R_1 each filtering elements that are either too small or too large. The sizes of L_1, R_1 and C are balanced to achieve fast pruning and low failure probability[5] (see Lemmas 24 and 25).

To reduce the fragile complexity of elements in L_1 and R_1, most elements are prefiltered using a cascade of weaker classifiers L_i and R_i for $2 \leq i \leq b$ geometrically growing in size by a factor of $d(n)$ when moving away from the center. Filtered elements that are classified into a bucket L_i or B_i with $i < b$ are used as new pivots and effectively limit the expected load per pivot. As the median is likely to travel through this cascade, the number of filter layers is a compromise between the fragile complexity f_{med} of the median and f_{rem} of the remaining elements.

We define $k(n)$ and $d(n)$ as functions since they depend on the problem size which changes for recursive calls. If n is unambiguous from context, we denote them as k and d respectively and assume $k(n) = \Omega(n^{\varepsilon})$ for some $\varepsilon > 0$.

▶ **Lemma 24.** *Consider any recursion step and let X be the set of n elements passed as the subproblem. After all elements in X are processed, the center partition C contains the median $x_m \in X$ whp.*

Proof. The algorithm can fail to move the median into bucket C only if the sample $S \subset X$ is highly skewed. More formally, we use a safety margin n_0 around the median of S and observe that if there exists $x_l, x_r \in S_C := S[k/2 - n_0 : k/2 + n_0]$ with $x_l < x_m < x_r$, the median x_m is moved into C.

This fails in case too many small or large elements are sampled. In the following, we bound the probability of the former from above; the symmetric case of too many large elements follows analogously. Consider k Bernoulli random variables X_i indicating that the i-th sample $s_i < x_m$ lies below the median and apply Chernoff's inequality $\Pr[\sum_i X_i > (1+\delta)\mu] \leq \exp(-\mu\delta^2/3)$ where $\mu = \mathbb{E}[\sum_i X_i]$ and $\delta < 1$:

$$\Pr[\neg\exists\, x_r \in S_C \colon x_m < x_r] = \Pr\left[\sum_{i=1}^{k} X_i > k/2 + n_0\right] \leq \exp\left(-\frac{2n_0^2}{3k}\right) \overset{n_0 = \sqrt{2k\log n}}{=} n^{-4/3}.$$

◀

[5] RMedian is guaranteed to select the correct median. Failure results in an asymptotically insignificant increase of expected fragile complexity (see line 21 in the pseudo code, Appendix C.3).

▶ **Lemma 25.** *Each recursion step of* RMedian *reduces the problem size from n to $\mathcal{O}(\sqrt{n \log n})$ whp.*

Proof. The algorithm recurses on the center bucket C which contains the initial sample of size $2n_0 = \mathcal{O}(\sqrt{n \log n})$ and all elements that are not filtered out by L_1 and R_1. We pessimistically assume that each element added to C compared to the weakest classifiers in these filters (i.e., the largest element in R_1 and the smallest in L_1).

We hence bound the rank in X of the R_1's largest pivot; due to symmetry L_1 follows analogously. Using a setup similar to the proof of Lemma 24, we define Bernoulli random variables indicating that the i-th sample s_i is larger than the ℓ-th largest element in X where $\ell = n/2 + 3\sqrt{n \log(n)}$. Applying Chernoff's inequality yields the claim. ◀

▶ **Lemma 26.** *The expected fragile complexity of the median is*

$$\mathbb{E}\left[f_{med}(n)\right] = \mathbb{E}\left[f_{med}\left(\mathcal{O}(\sqrt{n \log n})\right)\right] + \mathcal{O}\Big(\underbrace{\frac{k}{n} \log k}_{\text{Sampled}} + \underbrace{(1 - \frac{k}{n}) \log_d k}_{\text{Not sampled}} + \underbrace{1}_{\text{Misclassified}}\Big).$$

Proof. Due to Lemma 25, a recursion step reduces the problem size from n to $\mathcal{O}(\sqrt{n \log n})$ whp represented in the recursive summand. The remaining terms apply depending on whether the median is sampled or not: with $\Pr[x_m \notin S] = 1 - k/n$ the median is not sampled and moved towards the center triggering a constant number of comparisons in each of the $\mathcal{O}(\log_d n)$ buckets. Otherwise if the median x_m is sampled, it incurs $\mathcal{O}(\log k)$ comparisons while S is sorted. By Lemma 24, the median is then assigned to C whp and protected from further comparisons. The complementary event of x_m being misclassified has a vanishing contribution due to its small probability. ◀

▶ **Lemma 27.** *The expected fragile complexity of non-median elements is*

$$\mathbb{E}\left[f_{rem}(n)\right] = \mathbb{E}\left[f_{rem}\left(\mathcal{O}(\sqrt{n \log n})\right)\right] + \mathcal{O}\Big(\underbrace{\log k}_{\text{Sampled}} + \underbrace{\log_d n}_{\text{Not sampled}} + \max\Big(\underbrace{d^2}_{\substack{\text{Pivot in } R_i \\ i>2}}, \underbrace{\frac{nd}{k}}_{\substack{\text{Pivot in } R_j \\ j\leq 2}}\Big)\Big),$$

where $d(n) = \Omega(\log^{\varepsilon} n)$ for some $\varepsilon > 0$ and $k(n) = \mathcal{O}(n/\log n)$.

Proof. As the recursion is implemented analogously to Lemma 26, we discuss only the contribution of a single recursion step. Let $x \in X \backslash \{x_m\}$ be an arbitrary non-median element.

The element x is either sampled and participates in $\mathcal{O}(\log k)$ comparisons. Otherwise it traverse the filter cascade and moves to C, L_i or R_i requiring $\mathcal{O}(\log_d n)$ comparisons.

If it becomes a median candidate (i.e. $x \in C$), x has a fragile complexity as discussed in Lemma 26 which is asymptotically negligible here. Thus we only consider the case that x is assigned to L_i or R_i and due to symmetry assume without loss of generality that $x \in R_i$. If it becomes a member of the outer-most bucket R_b, it is effectively discarded. Otherwise, it can function as a new pivot element replenishing the bucket's comparison budget. As RMedian always uses a bucket's least-frequently compared element as pivot, it suffices to bound the expected number of comparisons until a new pivot arrives.

Observe that RMedian needs to find an element $y \in (R_{i-2} \cup R_{i-1})$ with $y < x$ in order to establish that $x \in R_i$. This is due to the fact that pivots can be placed near the unfavorable border of a bucket rendering them weak classifiers. We here pessimistically assume that

$y \in R_{i-2}$ for simplicity's sake. By construction the initial bucket sizes n_i grow geometrically by a factor of d as i increases. Therefore, any item compared to bucket R_i continues to the next bucket with probability at most $1/d$. Consequently, bucket R_i with $i > 2$ sustains expected $\mathcal{O}(d^2)$ comparisons until a new pivot arrives.

For the two inner most buckets R_j with $j \in \{1, 2\}$, this is not true as they are not replenished. Bucket R_j ultimately receives a fraction $\mathcal{O}(n_i/k)$ of all items whp, however it is expected to process a factor d more comparisons due to the possibly weak classification in the previous bucket R_{j+1}. Since there are n_i pivots in bucket R_j, each pivot in R_j participates in $\mathcal{O}(kd/n)$ comparisons. ◀

▶ **Theorem 28.** RMEDIAN *achieves* $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\log\log n)$ *and* $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\sqrt{n})$.

Proof. Choose $k(n) = n^\varepsilon$, $d(n) = n^\delta$ with $\varepsilon = 2/3$, $\delta = 1/12$. Then Lemmas 26 and 27 yield:

$$\mathbb{E}[f_{\text{med}}(n)] = \mathbb{E}\left[f_{\text{med}}(\mathcal{O}(\sqrt{n\log n}))\right] + \mathcal{O}(n^{\varepsilon-1}\varepsilon\log n + \frac{\varepsilon}{\delta}) = \mathcal{O}(\log\log n)\,,$$
$$\mathbb{E}[f_{\text{rem}}(n)] = \mathbb{E}\left[f_{\text{rem}}(\mathcal{O}(\sqrt{n\log n}))\right] + \mathcal{O}((\varepsilon + \frac{1}{\delta})\log n + \max(n^{2\delta}, n^{1-\varepsilon+2\delta})) = \mathcal{O}(\sqrt{n})\,.$$

◀

▶ **Theorem 29.** RMEDIAN *achieves* $\mathbb{E}[f_{med}(n)] = \mathcal{O}(\frac{\log n}{\log\log n})$, $\mathbb{E}[f_{rem}(n)] = \mathcal{O}(\log n)$.

Proof. Choose $k(n) = \frac{n}{\log^\varepsilon n}$, $d(n) = \log^\delta n$ with $\varepsilon = \delta = 1/3$. Then Lemmas 26 and 27 yield:

$$\mathbb{E}[f_{\text{med}}(n)] = \mathbb{E}\left[f_{\text{med}}(\mathcal{O}(\sqrt{n\log n}))\right] + \mathcal{O}(\log^{1-\varepsilon} n + \log_{\log^\delta n} n) = \mathcal{O}(\frac{\log n}{\log\log n})\,,$$
$$\mathbb{E}[f_{\text{rem}}(n)] = \mathbb{E}\left[f_{\text{rem}}(\mathcal{O}(\sqrt{n\log n}))\right] + \mathcal{O}(\log n + \max(\log^{2\delta} n + \log^{\varepsilon+\delta} n)) = \mathcal{O}(\log n)\,.$$

◀

▶ **Theorem 30.** *For* $k = \mathcal{O}(n/\log n)$ *and* $d = \Omega(\log n)$, RMEDIAN *performs a total of* $\mathcal{O}(n)$ *comparisons in expectation, implying* $w(n) = \mathcal{O}(n)$ *expected work.*

Proof. We consider the first recursion step and analyze the total number of comparisons. RMEDIAN sorts k elements using AKS resulting in $\mathcal{O}(k\log k) = \mathcal{O}(n)$ comparisons. It then moves $\mathcal{O}(n)$ items through the filtering cascade consisting of buckets of geometrically decreasing size resulting of $\mathcal{O}(1)$ expected comparisons per item. Each bucket stores its pivots in a minimum priority queue with the number of comparisons endured by each pivot as keys. Even without exploitation of integer keys, retrieving and inserting keys is possible with $\mathcal{O}(\log n)$ comparisons each. Hence, we select a pivot and keep it for $d = \Omega(\log n)$ steps, resulting in amortized $\mathcal{O}(1)$ work per comparison. This does not affect f_{med} and f_{rem} asymptotically. Using Lemma 25, the total number of comparisons hence $g(n) = g(\sqrt{n\log n}) + \mathcal{O}(n) = \mathcal{O}(n)$. ◀

D Details of Sorting

Recall from Section 1 that for sorting networks, the few existing networks with depth $\mathcal{O}(\log n)$ are all based on expanders, while a number of $\mathcal{O}(\log^2 n)$ depth networks exist, many of which are based on merging. Here we study the power of the merging paradigm with respect to fragile complexity for comparison-based algorithms, and show that fragile complexity $\mathcal{O}(\log^2 n)$ can be achieved, that this is best possible for any binary MERGESORT, but also that in the expected sense, fragile complexity of standard MERGESORT is $\mathcal{O}(\log n)$. We leave

open the question whether finding a simple sorting algorithm (for instance, not involving expander graphs) with fragile complexity $\mathcal{O}(\log n)$ is easier than finding a sorting network of depth $\mathcal{O}(\log n)$.

D.1 Lower Bound for MergeSort

▶ **Lemma 31.** *Merging of two sorted sequences A and B has fragile complexity at least $\lfloor \log_2 |A| \rfloor + 1$.*

Proof. A standard adversary argument: The adversary designates one element x in B to be the scapegoat and resolves in advance answers to comparisons between A and $B_1 = \{y \in B \mid y < x\}$ by $B_1 < A$ and answers to comparisons between A and $B_2 = \{y \in B \mid x < y\}$ by $A < B_2$. There are still $|A| + 1$ total orders on $A \cup B$ compatible with these choices, one for each position of x in the sorted order of A. Only comparisons between x and members of A can make some of these total orders incompatible with answers given by the adversary. Since the adversary can always choose to answer such comparisons in a way which at most halves the number of compatible orders, at least $\lfloor \log_2 |A| \rfloor + 1$ comparisons involving x have to take place before a single total order is known. ◀

By standard MergeSort, we mean the algorithm which divides the n input elements into two sets of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, recursively sorts these, and then merges the resulting two sorted sequences into one. The merge algorithm is not restricted, unless we specify it explicitly (which we only do for the upper bound, not the lower bound).

▶ **Lemma 32.** *Standard* MergeSort *has fragile complexity $\Omega(\log^2 n)$.*

Proof. In MergeSort, when merging two sorted sequences A and B, no comparisons between elements of A and B have taken place before the merge. Also, the sorted order of $A \cup B$ has to be decided by the algorithm after the merge. We can therefore run the adversary argument from the proof of Lemma 31 in all nodes of the mergetree of MergeSort. If the adversary reuses scapegoat elements in a bottom-up fashion—that is, as scapegoat for a merge of A and B chooses one of the two scapegoats from the two merges producing A and B—then the scapegoat at the root of the mergetree has participated in

$$\Omega(\sum_{i=0}^{\log n} \log 2^i) = \Omega(\sum_{i=0}^{\log n} i) = \Omega(\log^2 n)$$

comparisons, by Lemma 31 and the fact that a node at height i in the mergetree of standard MergeSort operates on sequences of length $\Theta(2^i)$. ◀

We now show that making unbalanced merges cannot improve the fragile complexity of binary MergeSort.

▶ **Theorem 14.** *Any binary mergesort has fragile complexity $\Omega(\log^2 n)$.*

Proof. The adversary is the same as in the proof of Lemma 32, except that as scapegoat element for a merge of A and B it always chooses the scapegoat from the *larger* of A and B. We claim that for this adversary, there is a constant $c > 0$ such that for any node v in the mergetree, its scapegoat element has participated in at least $c \log^2 n$ comparisons in the subtree of v, where n is the number of elements merged by v. This implies the theorem.

We prove the claim by induction on n. The base case is $n = \mathcal{O}(1)$, where the claim is true for small enough c, as the scapegoat by Lemma 31 will have participated in at least one

comparison. For the induction step, assume v merges two sequences of sizes n_1 and n_2, with $n_1 \geq n_2$. By the base case, we can assume $n_1 \geq 3$. Using Lemma 31, we would like to prove for the induction step

$$c \log^2 n_1 + \lfloor \log n_2 \rfloor + 1 \geq c \log^2(n_1 + n_2). \tag{6}$$

This will follow if we can prove

$$\log^2 n_1 + \frac{\log n_2}{c} \geq \log^2(n_1 + n_2)\,. \tag{7}$$

The function $f(x) = \log^2 x$ has first derivative $2(\log x)/x$ and second derivative $2(1-\log x)/x^2$, which is negative for $x > e = 2.71\ldots$. Hence, $f(x)$ is concave for $x > e$, which means that first order Taylor expansion (alias the tangent) lies above f, i.e., $f(x_0)+f'(x_0)(x-x_0) \geq f(x)$ for $x_0, x > e$. Using $x_0 = n_1$ and $x = n_1+n_2$ and substituting the first order Taylor expansion into the right side of (7), we see that (7) will follow if we can prove

$$\frac{\log n_2}{c} \geq 2\frac{\log n_1}{n_1} n_2\,,$$

which is equivalent to

$$\frac{\log n_2}{n_2} \geq 2c\frac{\log n_1}{n_1}\,. \tag{8}$$

Since $n_1 \geq n_2$ and $(\log x)/x$ is decreasing for $x \geq e$, we see that (8) is true for $n_2 \geq 3$ and c small enough. Since $\log(3)/3 = 0.366\ldots$ and $\log 2/2 = 0.346\ldots$, it is also true for $n_2 = 2$ and c small enough. For the final case of $n_2 = 1$, the original inequality (6) reduces to

$$\log^2 n_1 + \frac{1}{c} \geq \log^2(n_1 + 1)\,. \tag{9}$$

Here we can again use concavity and first order Taylor approximation with $x_0 = n_1$ and $x = n_1 + 1$ to argue that (9) follows from

$$\frac{1}{c} \geq 2\frac{\log n_1}{n_1}\,.$$

which is true for c small enough, as $n_1 \geq 3$ and $(\log x)/x$ is decreasing for $x \geq e$. ◀

D.2 Upper Bound for MergeSort with Linear Merging

By *linear merging*, we mean the classic sequential merge algorithm that takes two input sequence and iteratively moves the the minimum of both to the output.

▶ **Observation 2.** *Consider two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$. In linear merging, the fragile complexity of element a_i is at most $\ell + 1$ where ℓ is the largest number of elements from B that are placed directly in front of a_i (i.e. $b_j < \ldots < b_{j+\ell-1} < a_i$).*

▶ **Theorem 15.** *Standard* MergeSort *with linear merging has a worst-case fragile complexity of $\Theta(n)$.*

Proof. *Lower bound $f(n) = \Omega(n)$*: linear merging requires $\mathcal{O}(k)$ comparisons to output a sequence of length k. In standard MergeSort, each element takes part in $\mathcal{O}(\log n)$ merges of geometrically decreasing sizes $n/2^i$ (from root), resulting in $\mathcal{O}(n)$ comparisons.

Upper bound $f(n) = \mathcal{O}(n)$: consider the input sequence $(n, 1, 2, \ldots, n-1)$ where $n = 2^k$. Then every node on the the left-most path of the mergetree contains element n. In each merging step, we receive $A = (1, \ldots, \ell-1, n)$ from the left child, $B = (\ell, \ldots, 2\ell-1)$ from the right, and produce

$$(\underbrace{1, \ldots, \ell-1}_{\text{from } A}, \underbrace{\ell, \ldots, 2\ell-1}_{\text{from } B}, \underbrace{n}_{\text{from } A}).$$

Hence, the whole sequence B is placed directly in front of element n, resulting in $\Theta(\ell)$ comparisons with this element according to Observation 2. Then, the sum of the geometrically increasing sequence length yields the claim. ◀

▶ **Lemma 33.** *Let $X = \{x_1, \ldots, x_{2k}\}$ be a finite set of distinct elements, and consider a random bipartition $X_L, X_R \subset X$ with $|X_L| = |X_R| = k$ and $X_L \cap X_R = \varnothing$, such that $\Pr[x_i \in X_L] = 1/2$. Consider an arbitrary ordered set $Y = \{y_1, \ldots, y_m\} \subset X$ with $m \leq k$. Then $\Pr[Y \subseteq X_L \vee Y \subseteq X_R] < 2^{1-m}$.*

Proof.

$$\Pr[Y \subseteq X_L \vee Y \subseteq X_R] = 2\prod_{i=1}^{m} \Pr[\, y_i \in X_L \,|\, y_1, \ldots y_{i-1} \in X_L \,] = 2\frac{(2k)^{-m}k!}{(k-m)!} \leq 2 \cdot 2^{-m}.$$ ◀

▶ **Theorem 16.** *Standard* MergeSort *with linear merging on a randomized input permutation has a fragile complexity of $\mathcal{O}(\log n)$ with high probability.*

Proof. Let $Y = (y_1, \ldots, y_n)$ be the input-sequence, π^{-1} be the permutation that sorts Y and $X = (x_1, \ldots, x_n)$ with $x_i = y_{\pi^{-1}(i)}$ be the sorted sequence. Wlog we assume that all elements are unique[6], that any input permutation π is equally likely[7], and that n is a power of two.

Merging in one layer. Consider any merging-step in the mergetree. Since both input sequences are sorted, the only information still observable from the initial permutation is the bi-partitioning of elements into the two subproblems. Given π, we can uniquely retrace the mergetree (and vice-versa): we identify each node in the recursion tree with the set of elements it considers. Then, any node with elements $X_P = \{y_\ell, \ldots, y_{\ell+2k-1}\}$ has children

$$X_L = \{x_{\pi(i)} \mid \ell \leq \pi(i) \leq \ell + k - 1\} = \{y_\ell, \ldots, y_{\ell+k-1}\},$$
$$X_R = \{x_{\pi(i)} \mid \ell + k \leq \pi(i) \leq \ell + 2k - 1\} = \{y_{\ell+k}, \ldots, y_{\ell+2k-1}\}.$$

Hence, locally our input permutation corresponds to an stochastic experiment in which we randomly draw exactly half of the parent's elements for the left child, while the remainder goes to right.

This is exactly the situation in Lemma 33. Let N_i be a random variable denoting the number of comparisons of element y_i in the merging step. Then, from Observation 2 and Lemma 33 it follows that $\Pr[N_i = m{+}1] \leq 2^{-m}$. Therefore N_i is stochastically dominated by $N_i \preceq 1{+}Y_i$ where Y_i is a geometric random variable with success probability $p = 1/2$.

Merging in all layers. Let $N_{j,i}$ be the number of times element y_i is compared in the j-th recursion layer and define $Y_{j,i}$ analogously. Due to the recursive partitioning argument, $N_{j,i}$ and $Y_{j,i}$ are iid in j. Let N_i^T be the total number of comparisons of element i, i.e.

[6] If this is not the case, use input sequence $Y' = ((y_1, 1), \ldots, (y_n, n))$ and lexicographical compares.
[7] If not shuffle it before sorting in linear time and no fragile comparisons.

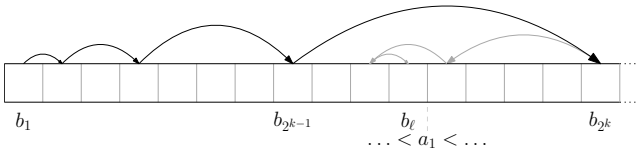

Figure 3 The Exponential search performs k doubling steps and overshoots the target b_ℓ with $b_\ell < a_1 < b_{\ell+1}$. A binary search between $b_{2^{k-1}}$ and b_{2^k} ultimately identifies b_ℓ in $\mathcal{O}(k)$ steps.

$N_i^T \preceq \log_2 n + \sum_{j=1}^{\log_2 n} Y_{j,i}$. Then a tail bound on the sum of geometric variables (Theorem 2.1 in [12]) yields:

$$\Pr\left[\sum_{j=1}^{\log_2 n} Y_{j,i} \geq \lambda \mathbb{E}\left[\sum_{j=1}^{\log_2 n} Y_{j,i}\right] = 2\lambda \log_2 n\right] \overset{[12]}{\leq} \exp\left(-\frac{1}{2}\frac{2\ln n}{\ln 2}[\lambda - 1 - \log\lambda]\right) = n^{-2},$$

where we set $\lambda \approx 3.69$ in the last step solving $\lambda - \log\lambda = 2\log 2$. Thus, we bound the probability $\Pr\left[N_i^T \geq (1+2\lambda)\log_2 n\right] \leq n^{-2}$.

Fragile complexity. It remains to show that with high probability no element exceeds the claimed fragile complexity. We use a union bound on N_i^T for all i:

$$\Pr\left[\max_i\{N_i^T\} = \omega(\log n)\right] \leq n \Pr\left[N_i^T = \omega(\log n)\right] \leq 1/n\,. \qquad \blacktriangleleft$$

D.3 Upper Bound for MergeSort with Exponential Merging

We define *exponential merging* of sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$ as follows: if either A or B are empty, output the other one and stop. Otherwise, assume without loss of generality that m is a power of two and that there exists an $b_i \in Y$ with $a_1 < b_i$, if not append sufficiently many virtual elements $b_\top$ to B with $a_1 < b_\top$. Use an exponential search on B starting in b_1 to find all elements $b_1 < \ldots < b_\ell < a_1$ smaller than a_1. As illustrated in Fig. 3, the exponential search consists of a *doubling phase* which finds the smallest k with $a_1 < b_{2^k}$. Since the doubling phase may overshoot b_ℓ, a *binary search* between $b_{2^{k-1}}$ and y_{b^k} follows. Output $b_1, \ldots, b_\ell, a_1$ and recurse on $A' = [b_{\ell+1}, \ldots, b_m]$ and $B' = [a_2, \ldots, a_n]$ which swaps the roles of A and B.

▶ **Theorem 17.** *Exponential merging of two sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ has a worst-case fragile complexity of $\mathcal{O}(\log n)$.*

Proof. Without loss of generality let n be a power of two and consider a single exponential search finding the smallest k with $a_1 < b_{2^k}$. The element a_1 is compared to all $\{b_{2^i} \mid 1 \leq i \leq k\}$ during the doubling phase. We use an accounting argument to bound the fragile complexity. Element a_1 takes part in every comparison and is charged with $k = \mathcal{O}(\log n)$. It is then charged $\mathcal{O}(\log n)$ comparisons during the binary search between $b_{2^{k-1}}$ and b_{2^k}. It is then moved to the output and not considered again.

The search also potentially interacts with the 2^k elements $b_1, \ldots, b_{2^k}$ by either comparing them during the doubling phase, during the binary search or by skipping over them. We pessimistically charge each of these elements with one comparison. It then remains to show that no element takes part in more than $\mathcal{O}(\log n)$ exponential searches.

Observe that all elements $b_1, \ldots, b_{2^{k-1}}$ are moved to the output and do not take part in any more comparisons. In the worst-case, the binary search proves that element $b_{2^{k-1}+1}$ and its successors are larger than a_1. Hence at most half of the elements covered by the exponential search are available for further comparisons. To maximize the charge, we

recursively setup exponential search whose doubling phases ends in b_{2^l} yielding a recursion depth of $\mathcal{O}(\log n)$.[8] ◀

▶ **Corollary 34.** *Applying Theorem 17 to standard* MergeSort *with exponential merging yields a fragile complexity of* $\mathcal{O}(\log^2 n)$ *in the worst-case.*

E Details of Constructing Binary Heaps

▶ **Observation 1.** *The fragile complexity of the standard binary heap construction algorithm of Floyd [10] is* $\mathcal{O}(\log n)$.

Proof. Consider first an element sifting down along a path in the tree: as the binary tree being heapified has height $\mathcal{O}(\log(n))$ and the element moving down is compared to one child per step, the cost to this element before it stops moving is $\mathcal{O}(\log(n))$. Consider now what may happen to an element x in the tree as another element y is sifting down: x is only hit if y is swapped with the parent of x which implies that y was an ancestor of x. As the height of the tree is $\mathcal{O}(\log(n))$, at most $\mathcal{O}(\log(n))$ elements reside on the path above x. Note that the x may be moved up once as y passes by it; this only lowers the number of elements above x. In total, any element in the heap is hit at most $\mathcal{O}(\log(n))$ times during heapify. ◀

[8] If the following searches were shorter they would artificially limit the recursion depth. If they were longer, too many elements are removed from consideration as only the binary search range can be charged again.