Robust and Adaptive Search*

Yann Disser[1] and Stefan Kratsch[2]

1 **TU Darmstadt, Darmstadt, Germany**
`disser@mathematik.tu-darmstadt.de`
2 **University of Bonn, Bonn, Germany**
`kratsch@cs.uni-bonn.de`

Abstract

Binary search finds a given element in a sorted array with an optimal number of $\log n$ queries. However, binary search fails even when the array is only slightly disordered or access to its elements is subject to errors. We study the worst-case query complexity of search algorithms that are robust to imprecise queries and that adapt to perturbations of the order of the elements. We give (almost) tight results for various parameters that quantify query errors and that measure array disorder. In particular, we exhibit settings where query complexities of $\log n + ck$, $(1 + \varepsilon)\log n + ck$, and $\sqrt{cnk} + o(nk)$ are best-possible for parameter value k, any $\varepsilon > 0$, and constant c.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases searching, robustness, adaptive algorithms, memory faults, array disorder

Digital Object Identifier 10.4230/LIPIcs.STACS.2017.26

1 Introduction

Imagine a large register with n files from which you wish to extract a particular file. All files are indexed by some key and the files are sorted by key value. Not knowing the distribution of the keys, you probably use binary search since looking at $\log n$ keys is best possible in the worst case. Unfortunately, however, other users have accessed files before you and have only returned the files to approximately the right place. As a result, the register is unsorted, but at least each file is within some small number k of positions of where it should be. How should you proceed? If you knew k and n, at what ratio of k vs. n should you resort to a linear search of the register? If you do not know k, can you still do reasonably well? What if the register was recently moved, by packing the files into boxes, but in the process the order of the boxes got mixed up, and now there are large blocks of files that are far away from their correct locations? What if you misread some of the keys? Situations like these are close to searching in a sorted register and there are plenty of parameters that measure closeness to a sorted array, e.g., maximum displacement or minimum block moves to sort, respectively persistent or temporary read errors. We give (almost) optimal algorithms for a large variety of these measures, and thereby establish for each of them exact regimes in which we can outperform a linear search of all elements, or even be almost as good as binary search.

More formally, we study the fundamental topic of comparison-based search, which is central to many algorithms and data structures [20, 24, 31]. In its most basic form, the search problem can be phrased in terms of locating an element e within a given array A. In order to search A efficiently, we need structure in the ordering of its elements: In general, we cannot hope to avoid querying all entries to find e. The most prominent example of an

* A full version of the paper is available at `http://arxiv.org/abs/1702.05932`.

efficient search algorithm that exploits special structure is *binary search* for sorted arrays. Binary search is best-possible for this case. It needs only logarithmically many queries and is thus very well suited for searching extremely large collections of data. However, it heavily relies on perfect order and reliable access to the data. For large and dynamically changing collections of data, both requirements may be difficult to ensure, but it may be reasonable to assume the number of imperfections to be bounded. Accordingly, we ask: *What is the best-possible search algorithm if the data may be disordered or we cannot access it reliably? In what regime of the considered measure is it better than linear search?*

We provide (almost) tight bounds on the query complexity of searching an array A with n entries for an element e in a variety of settings. Each setting is characterized by bounding a different parameter k that quantifies the imperfections regarding either our access to array elements or regarding the overall disorder of the data. Note that one can always resort to linear search, which rules out lower bounds stronger than n comparisons.[1] Table 1 gives an overview of the parameters we analyze and our respective results. Qualitatively, our results can be grouped into three groups of settings leading to different query complexities, and we briefly highlight each group in the following.

The first group contains the parameters k_{sum}, $k_{\max}$, and k_{inv}, which quantify the summed and maximum distance of each element from its position in sorted order and the number of element pairs in the wrong relative order, respectively (detailed definitions can be found the the corresponding sections). For all of these parameters we are able to show that $\log n + ck$ queries are necessary and sufficient, for constant c. Intuitively, this is the best complexity we can hope for: We cannot do better than $\log(n)$ queries, and the impact of k on the query complexity is linear and can be isolated.

The second group of results is with respect to the parameters k_{lies}, k_{faults}, as well as multiple parameters for edit distances that measure the number of element operations needed to sort A. The parameter k_{lies} limits the number of queries that yield the wrong result, and k_{faults} limits the number of array positions that yield wrong query outcomes. For bounded values of k_{lies} and k_{faults} we show that e cannot be found with $\log n + ck$ queries using any binary-search-like algorithm.[2] On the other hand, we provide an algorithm that needs $(1 + 1/c) \log n + ck$ queries, for any $c \geq 1$. For bounded edit distances, it is easy to see that we need n queries if e need not be at its correct position relative to sorted order, since e can be moved anywhere with just 2 edits, forcing us to scan the whole array. If we assume e to be at its correct location, we can carry over the results for k_{lies} and k_{faults} to obtain the same bounds for the edit-distance related parameters k_{rep}, k_{seq}, k_{mov}, and k_{swap}.

Lastly, we consider the parameter k_{ainv} that counts the number of adjacent elements that are in the wrong relative order, as well as several parameters measuring the number of block operations needed to sort A. Intuitively, these settings are much more difficult for a search algorithm, as it takes relatively small parameter values to introduce considerable disorder. For the case that e is guaranteed to be at the correct position, we show that $\sqrt{cnk} + o(nk)$ queries are necessary and sufficient to locate e.

The algorithms for k_{ainv} and related parameters assume that the parameter value is known to the algorithm a priori. In contrast, all our other algorithms are oblivious to the parameter, in the sense that they do not require knowledge of the parameter value as long

[1] Accordingly, all (lower) bounds of the form $f(n, k)$ throughout the paper are to be understood as $\min\{f(n, k), n\}$. A naive bound of n can easily be obtained by scanning the whole array.

[2] We interpret the array as a binary tree (rooted at entry $n/2$, with the two children $n/4$, $3n/4$, etc.), and call an algorithm "binary-search-like" if it never queries a node (other than the root) before querying its parent.

Table 1 Overview of our results, with main results in boldface.[1] (^{o}: even if oblivious to parameter value; ^{c}: for all $c \geq 1$; ^{t}: for tree-algorithms; ^{e}: for $\mathrm{pos}(e) = \mathrm{rank}(e)$)

parameter	description	bounds: lower	bounds: upper
Section 3 – number of imprecise queries			
k_{lies}	wrong outcomes	$\mathbf{\log n + ck}$ [Th. 3]^{ct}	$\mathbf{(1+\frac{1}{c})\log n + (2c+2)k}$ [Th. 2]^{oc}
k_{faults}	indices with wrong outcomes	$\log n + ck$ [Th. 3]^{ct}	$(1+\frac{1}{c})\log n + (2c+2)k$ [Th. 4]^{oc}
Section 4.1 – displacement of elements			
k_{sum}	total displacement	$\mathbf{\log n/k + 2k + \mathcal{O}(1)}$ [Th. 5]^{o}	
k_{max}	maximum displacement	$\mathbf{\log n/k + 3k + \mathcal{O}(1)}$ [Th. 6]^{o}	
Section 4.2 – number of inversions			
k_{inv}	all inversions	$\log n/k+2k+\mathcal{O}(1)$ [Co. 7]	$\log n/k + 4k + \mathcal{O}(1)$ [Co. 7]^{o}
k_{ainv}	adjacent inversions	$\mathbf{\sqrt{8nk} + o(\sqrt{nk})}$ [Th. 10,9]^{e}	
Section 4.3 – element operations needed to sort the array			
k_{rep}	element replacements	$\log n + ck$ [Co. 14]^{cte}	$(1+\frac{1}{c})\log n + (4c+4)k$ [Th. 13]^{oe}
k_{seq}	$n - \lvert\text{max ordered subseq.}\rvert$	$\log n + ck$ [Co. 14]^{cte}	$(1+\frac{1}{c})\log n + (4c+4)k$ [Th. 13]^{oe}
k_{mov}	element moves	$\log n + ck$ [Co. 14]^{cte}	$(1+\frac{1}{c})\log n + (4c+4)k$ [Th. 13]^{oe}
k_{swap}	element swaps	$\log n + ck$ [Co. 14]^{cte}	$(1+\frac{1}{c})\log n + (8c+8)k$ [Th. 13]^{oe}
k_{aswap}	adj. element swaps	$\log n/k+2k+\mathcal{O}(1)$ [Co. 15]	$\log n/k + 4k + \mathcal{O}(1)$ [Co. 15]^{o}
Section 4.4 – block operations needed to sort the array			
k_{bswap}	block swaps	$4\sqrt{nk} + o(\sqrt{nk})$ [Th. 17]^{e}	
k_{rbswap}	equal size block swaps	$2\sqrt{2nk} + o(\sqrt{nk})$ [Th. 18]^{e}	$4\sqrt{nk} + o(\sqrt{nk})$ [Th. 18]^{e}
k_{bmov}	block moves	$2\sqrt{2nk} + o(\sqrt{nk})$ [Th. 19]^{e}	

as the target element e is guaranteed to be present in the array. Note that if e need not be present and we have no bound on the disorder, we generally need to inspect every entry of the array in case we cannot find e. For the parameter k_{lies}, we do not even know how long we need to continue querying the same elements until we may conclude that e is not part of the array. Any of our oblivious algorithms can trade the guarantee that $e \in A$ against knowledge of the parameter value k: Compute from k the maximum number m of queries that it would take without knowing k when $e \in A$. If the algorithm does not stop within m queries then it is safe to answer that e is not in A.

Overall, our results point out several parameters for which a fairly large regime of k (as a function of n) allows search algorithms that are provably better than linear search. For example, while moving only a single element by a lot can lead to bounds of $\Omega(n)$ on the values of several parameters, and hence trivial guarantees, moving many elements by at most k places gives $k_{\text{max}} = k$ and yields better bounds than linear search (roughly) for $k < \frac{n}{3}$, and as good as binary search when $k = \mathcal{O}(\log n)$. Moving only few elements by an arbitrary number of spaces, in turn, still leads to good bounds via parameters such as k_{mov} or k_{swap}, as long as the target is in the correct place. Parameters such as k_{ainv} grow even more slowly, for certain types of disorder, but, on the other hand, only a small regime allows for better than trivial guarantees. While, for each individual parameter we study, there are "easily searchable" instances where the parameter becomes large and makes the corresponding bound trivial, our results often allow for good bounds by resorting to a different parameter.

1.1 Related Work

Our work falls into the area of *adaptive analysis of algorithms*, which aims at a fine-grained analysis of polynomial-time algorithms with respect to structural parameters of the input. An objective of this field is to find algorithms whose running-time dependence on input size and the structural parameters interpolates smoothly between known (good) bounds for special cases and the worst-case bound for general inputs. The topic of adaptive sorting, i.e., sorting arrays that are presorted in some sense, has attracted a lot of attention, see, e.g., [4, 13, 23, 28].

We now discuss results that are specific to searching in arrays. Several authors addressed the question of how much preprocessing, i.e., sorting, helps for searching, if we take into account the total time investment [8, 22, 29]. Fredman [18] gave lower bounds on searching regarding both queries and memory accesses. A classic work of Yao [32] established that the best way of storing n elements in a table such as to minimize number of queries for accessing an element is by keeping the elements sorted, which requires $\log n$ queries, provided that the key space is large enough. Regarding searching in (partially) unordered arrays, there is a nice result of Biedl et al. [5] about insertion sort based on repeated binary searches.

Under appropriate assumptions, namely that array is sorted and its elements are drawn from a known distribution (e.g., searching for a name in a telephone book), one can do much better than binary search, since the distribution allows a good prediction of where the target should be located. In this case $\mathcal{O}(\log\log n)$ queries suffice on average (cf. [31]); to avoid having to query the entire array, previous work suggests combinations of algorithms that perform no worse than binary search in the worst case [10, 6]. Another interesting branch of study is related to search in arrays of more complicated objects such as (long) strings [1, 17] or abstract objects with nonuniform comparison cost [19, 2].

Many papers have studied searching in the presence of different types of errors, e.g., [7, 15, 16, 25], see [11, 27] for surveys. A popular error model for searching allows for a linear number of lies [3, 7, 12, 14, 26], for which Borgstrom and Kosaraju [7] gave an $\mathcal{O}(\log n)$ search algorithm. In contrast, we bound the number of lies separately via the parameter k_{lies}. Rivest et al. [30] gave an upper bound of $\log n + k\log\log n + \mathcal{O}(k\log k)$ queries for this parameter. Their algorithm is based on a continuous strategy for the (equivalent) problem of finding an unknown value in $[1, n]$, up to a given precision, using few yes-no questions. Our algorithm (Theorem 2) uses asymptotically fewer queries if $k_{\text{lies}} = \omega(\log n/\log\log n)$.[3]

The works of Finocchi and Italiano [16] and Finocchi et al. [15] consider a parameter very similar to k_{faults}, with the additional assumption that faults may affect also the working memory of the algorithm, except for $\mathcal{O}(1)$ "safe" memory words. Finocchi and Italiano [16] give a deterministic searching algorithm that needs $\mathcal{O}(\log n + k^2)$ queries. Brodal et al. [9] improve this bound to $\mathcal{O}(\log n+k)$ and Finocchi et al. [15] provide a lower bound of $\Omega(\log n+k)$ even for randomized algorithms. Our results are incomparable as our result for parameter k_{faults} uses only $(1+\frac{1}{c})\log n + (2c+2)k$ queries, getting arbitrarily close to $\log n + O(k)$ (cf. Theorem 4), but does not consider faults in the working memory; the high level approach of balancing progress in the search with security queries is the same as in [9], but more careful counting is needed to get small constants. For parameter k_{lies} we give a simpler algorithm with $2\log n + 4k$ queries and using only $\mathcal{O}(1)$ words of working memory, but it is not clear whether the result can be transferred to k_{faults} without increasing the memory usage.

[3] A technical report of Long [21] claims that the actual tight bound of the algorithm of Rivest et al. [30] is $\mathcal{O}(\log n + k)$, which is consistent with our results.

Finally, we comment on the measures of disorder we adopt in this paper. We study various well-known measures that are mostly folklore. Detailed overviews of measures and their relations were given by Petersson and Moffat [28] and Estivill-Castro and Wood [13]. For the sake of completeness and to get all involved coefficients the full version of this work, accessible at `http://arxiv.org/abs/1702.05932`, provides proofs of all pairwise relations between our parameters; these are depicted in Figure 1.

2 Preliminaries

In this paper we consider the following problem: Given an array A of length n and an element e, find the position of e in A or report that $e \notin A$ with as few *queries* as possible. We use $A[i]$, $i \in 1, \dots, n$ to denote the i-th entry of A. We allow access to the entries of A only via queries to its indices, regarding the relation of the corresponding element to e. We write query(i) for the operation of querying A at index i, and let query$(i) =$ '$<$' (respectively, '$>$' or '$=$') denote the outcome indicating that $A[i] < e$ (respectively $A[i] > e$ or $A[i] = e$). Note that in faulty settings the query outcome need not be accurate.

To keep notation simple, we generally assume the entries of A to be unique unless explicitly stated otherwise. We emphasize that none of our results relies on this assumption. We can then define pos(a) to denote the index of a in A, by setting pos$(a) = i$ if and only if $A[i] = a$. Further, let rank$(a) = |\{i : A[i] < a\}| + 1$ be the "correct" position of a with respect to a sorted copy of A, irrespective of whether or not $a \in A$. We often use an element $a \in A$ and its index pos(a) interchangeably, especially for the target element e. Note that, as discussed in the introduction, for oblivious algorithms we generally assume $e \in A$.

3 Searching with imprecise queries

In this section, we consider the problem of finding the index pos(e) of an element e in a sorted array A of length $n = 2^d$, $d \in \mathbb{N}$ in a setting where queries may yield erroneous results. We say that '$<$' is a lie (the truth) for index i if $A[i] \geq e$ ($A[i] < e$), and analogously for '$>$' and '$=$'. To quantify the number of lies, we introduce two parameters k_{lies} and k_{faults}. The first parameter k_{lies} simply bounds the number of queries with erroneous results, which we interpret as the number of lies allowed to an adversary. The second parameter k_{faults} bounds the number of indices i for which query(i) (consistently) returns the wrong result, allowing the conclusion that $e \notin A$ in case query(e) yields the wrong result. Equivalently, for an unsorted array A, we can require all queries to be truthful and define $k_{\text{faults}}(e)$ to be the number of inversions involving e, i.e., $k_{\text{faults}}(e) = |i : (i < \text{pos}(e) \wedge A[i] > e) \vee (i > \text{pos}(e) \wedge A[i] < e)|$. Observe that both definitions of k_{faults} are equivalent. For clarity, we write k_{faults} when considering the adversarial interpretation, and $k_{\text{faults}}(e)$ when considering it as a measure of disorder of an unsorted array. For both k_{lies} and k_{faults}, we only allow queries to e to yield '$=$'.

The algorithms of this section operate on the binary search tree rooted at index $r = n/2$ that contains a path for each possible sequence of queries in a binary search of the array, and identify nodes of the tree with their corresponding indices. We write $\text{next}_{>}(i)$ and $\text{next}_{<}(i)$ to denote the two successors of node i, e.g., $\text{next}_{>}(r) = n/4$ and $\text{next}_{<}(r) = 3n/4$. Similarly, we write prev(i) to denote the predecessor of i in the binary search tree, and $\text{prev}_q(i) = v$ for the last vertex v on the unique r-i-path such that $\text{next}_q(v)$ also lies on the r-i-path ($\text{prev}_q(i) = \emptyset$ if no such node exists). Intuitively, $\text{prev}_q(i)$ is the last vertex corresponding to an array entry larger (if $q =$"$>$") or smaller (if $q =$"$<$") than $A[i]$. For convenience, query$(\emptyset) = \emptyset$, $\text{prev}_{</>}(r) = \emptyset$, and $\text{next}_{</>}(i) = i$ if i is a leaf of the tree. We further denote

Algorithm 1: Algorithm with $2\log n + 4k_{\text{lies}}$ queries.

```
the algorithm stops once a query yields '='
i ← n/2                                                    // start at the root
while (q ← query(i)) ≠ '=' do          // by definition, '=' cannot be a lie
    i' ← prev_{¬q}(i)   // ∅ if all queries on the path from the root yielded q
    while i ≠ i' ∧ query(i') = q do  // while query(i') contradicts its previous
     outcome...
        i ← prev(i)                                    // ...backtrack towards i'
    if i ≠ i' then              // if we did not backtrack all the way to i'...
        i ← next_q(i)                                  // ...proceed according to q
```

by $d(i, j)$ the length of the path from node i to node j in the search tree. We say that an algorithm *operates on the binary search tree* if no index is queried before its predecessor in the tree.

We start by considering the parameter k_{lies}. If we knew the value of this parameter, we could try a regular binary search, replace every query with $2k_{\text{lies}} + 1$ queries to the same element and use the majority outcome in each step. However, this would give $(2k_{\text{lies}} + 1)\log n$ queries, where ideally we should not use more than $\log n + f(k)$ queries. We first give an algorithm that achieves the separation between n and k_{lies} while being oblivious to the value of k_{lies}. Importantly, the algorithm only needs $\mathcal{O}(1)$ memory words, which also makes it applicable to settings where "safe" memory, that cannot be corrupted during the course of the algorithm, is limited. This algorithm still needs $2\log n + f(k)$ queries, but we will show later how to build on the same ideas to (almost) eliminate the factor of 2.

Intuitively, Algorithm 1 searches the binary search tree defined above, simply proceeding according to the query outcome at each node. In addition, the algorithm invests queries to double check past decisions. We distinguish left and right turns, depending on whether the algorithm proceeds with the left or the right child. In particular, before proceeding, the algorithm queries the last vertex on the path from the root where it decided for a turn in the opposite direction. While an inconsistency to previous queries is detected, i.e., a query to a vertex where it turned right (or left) gives '$>$' (or '$<$'), the algorithm backtracks one step. In this manner, the algorithm guarantees that it never proceeds along a wrong path without the adversary investing additional lies. Note that if the algorithm only ever turned right (or left), i.e., there was no previous turn in the opposing direction, it does not double check any past decisions until the query outcome changes. This is alright since either the algorithm is on the right path or the adversary needs to invest a lie in each step.

▶ **Theorem 1.** *We can find e obliviously using $2\log n + 4k_{\text{lies}}$ queries and $\mathcal{O}(1)$ memory.*

Proof. We claim that Algorithm 1 achieves the bound of the theorem. Note that $\text{prev}_{\neg q}(i)$ only depends on i and not on the outcome of previous queries, therefore, we can determine it with $\mathcal{O}(1)$ memory words. We will show that in each iteration of the outer loop of the algorithm, the potential function $\Phi = 2d(i, e) + 4k$ decreases by at least one for each query, where k is the number of remaining lies the adversary may make. This proves the claim, since $\Phi \geq 0$ and initially $\Phi \leq 2\log n + 4k_{\text{lies}}$. We analyze a single iteration of the outer loop.

Observe that if z is the number of iterations of the inner loop, then the total number of queries is $z + 2$ if the inner loop terminates because $\text{query}(i') = \neg q$, and $z + 1$ if it terminates because $i = i'$. If an iteration of the inner loop is caused by $\text{query}(i')$ being a

Algorithm 2: Algorithm with $(1+\frac{1}{c})\log n + (2c+2)k_{\text{lies}}$ queries.

the algorithm stops once a query yields '='
$i \leftarrow n/2$ // start at the root
while $(q \leftarrow \text{query}(i)) \neq$ '=' **do** // by definition, '=' cannot be a lie
  $i' \leftarrow \text{prev}_{\neg q}(i)$ // $\emptyset$ if all queries on the path from the root yielded q
  while $0 < c\Delta_{i'} < d(i,i') + 1$ **do** // while we do not have sufficient support to proceed...
    query(i') // ...query i' for support
  if $\Delta_{i'} = 0$ **then** // if we ran out of support at i' altogether...
    $i \leftarrow i'$ // ...backtrack to i'
  else // if we have sufficient support at i'...
    $i \leftarrow \text{next}_q(i)$ // ...proceed according to q

lie, then in this iteration $\Delta\Phi \leq 2 - 4 = -2$, and otherwise, $d(i,e)$ is decreased by one and likewise $\Delta\Phi = -2 + 0 = -2$. Overall, the change in potential during the inner loop is always $\Delta\Phi = -2z$. If the inner loop terminates because $i = i'$, then $z \geq 1$ and the total change in potential is $\Delta\Phi \leq -2z \leq -z - 1$, enough to cover all $z + 1$ queries.

Now consider the case that the inner loop terminates because $\text{query}(i') = \neg q$. If $\neg q$ is a lie for i' or q is a lie for i, the adversary invested an additional lie, and even if the last update to i increases $d(i,e)$, the total change in potential is bounded by $\Delta\Phi \leq -2z - 4 + 2 \leq -2z - 2$, enough to cover all $z + 2$ queries. On the other hand, if $\neg q$ is the truth for i' and q is the truth for i, then $e \in \{i', \ldots, i\}$ and i must lie on the unique r-e-path in the search tree (and $i \neq e$). The final update to i thus decreases $d(i,e)$ by 1 and the total change in potential is $\Delta\Phi = -2z - 2$, again enough to cover all $z + 2$ queries. ◀

We now adapt Algorithm 1 to minimize the impact of potential lies on the dependency on $\log n$ in the running time. Intuitively, instead of backing up each query $q \leftarrow \text{query}(i)$ by a query to $\text{prev}_{\neg q}(i)$, we back only one in c queries (cf. Algorithm 2). During the course of the algorithm and its analysis, we let $n_{q,j}$ denote the number of queries (so far) to node j that resulted in $q \in \{<, >\}$ and $\Delta_j := |n_{<,j} - n_{>,j}|$.

▶ **Theorem 2** (∗[4]). *For every $c \geq 1$, we can find e obliviously using $(1+\frac{1}{c})\log n + (2c+2)k_{\text{lies}}$ queries.*

Proof Sketch. We show that Algorithm 2 achieves the bound of the theorem. Instead of backing up every query as in Algorithm 1, Algorithm 2 only invests one supporting query for every c consecutive queries with the same outcome (for integral c). To capture this, our potential function needs to manage additional potential used to account for these irregular backup queries. This gets more involved, as the algorithm will not usually backtrack by increments of c, and we need to allow for fractional contributions to the potential in each query. We introduce a potential function of the form

$$\Phi = (1 + \frac{1}{c})d(i,e) + (2 + \frac{1}{c})L + \frac{1}{c}T + (2c+2)k,$$

where k is the number of lies remaining to the adversary, and L and T will not be formally defined in this proof sketch. Similarly to the proof of Theorem 1, assuming $\Phi \geq 0$ and

[4] Due to space restrictions, proofs for results marked with ∗ are deferred to the full version of this work.

initially $L, T = 0$, we need to show that (on average) the potential function decreases by at least 1 for each query the algorithm makes. We outline intuitively how this is achieved.

The first term of the potential function releases a small amount of potential for every step (in the tree) towards the target element and stores the same amount of potential for every step away from the target. The fourth term releases a lot of potential whenever the adversary invests a lie.

To understand the role of the central terms intuitively, consider an iteration of the algorithm, where it just turned left or right (i.e., moved to a left or right child) and reached node i, and let i' be the last vertex where it turned in the opposite direction. Now if the next query result of q at node i is a lie, the potential function releases a lot of potential that we can use to pay for the query, the possible backup query, and the change in the other terms of the potential. Let us therefore assume that q is the truth for i.

We need to distinguish two situations, depending on whether the turn at i' was correct or not, i.e., whether the (majority of) backup query outcomes were truthful. If the turn at i' was wrong, with every step that the algorithm proceeds down the tree, the distance to the target increases, but on the other hand, the adversary needs to invest a lie for each additional backup query. This means that we can afford to store potential with every new backup query, but need to invest potential for every step down the tree. Accordingly, the contribution to L in the potential function is defined to be $c\Delta_{j'} - d(j, j')$ and there is no contribution to T. If the turn at i' was correct, since q is also the truth for i and since the turns at i and i' are in opposing directions, the algorithm must still be on a path towards the target e. Hence, we can store potential with every step down the tree, but have to invest potential to pay for the backup queries. Accordingly, the contribution to T in the potential function is defined to be $d(j, j') - c(\Delta_{j'} - 1)$ and there is no contribution to L.

Now, at some point earlier in the execution of the algorithm, i' had the role of i and there was a node i'' with the role of i'. Since the algorithm may backtrack to i' in the future, the potential function also needs to remember the contributions of the pair (i', i'') to L and T, as well as the contributions of each earlier such *zig-zag* pair along the path to the root. By carefully defining L and T, we can balance all costs in such a way that, in each iteration of the outer loop of Algorithm 2, the potential reduction is at least equal to the number of queries made during that iteration. ◀

To provide a strong lower bound, we restrict ourselves to algorithms that operate on the binary search tree. Such algorithms interpret the array as a binary tree (rooted at entry n/2, with the two children n/4, 3n/4, etc.), and never query a node (other than the root) before querying its parent.

▶ **Theorem 3** (∗). *For every $c \in \mathbb{N}$ and $k \in \{k_{\text{lies}}, k_{\text{faults}}\}$, no algorithm operating on the search tree can find e with less than $\log n + ck$ queries in general.*[1]

We show how to translate any algorithm with a performance guarantee with respect to k_{lies} to an algorithm with the same guarantee for k_{faults}.

▶ **Theorem 4** (∗). *Let $f : \mathbb{N}^2 \to \mathbb{N}$. If we can find e with $f(n, k_{\text{lies}})$ queries, then we can find e with $f(n, k_{\text{faults}})$ queries.*

4 Searching disordered arrays

In this section, we consider the problem of finding the index $\text{pos}(e)$ of an element e in array A of length $n = 2^d$, $d \in \mathbb{N}$. In contrast to Section 3, we do not assume A to be sorted but

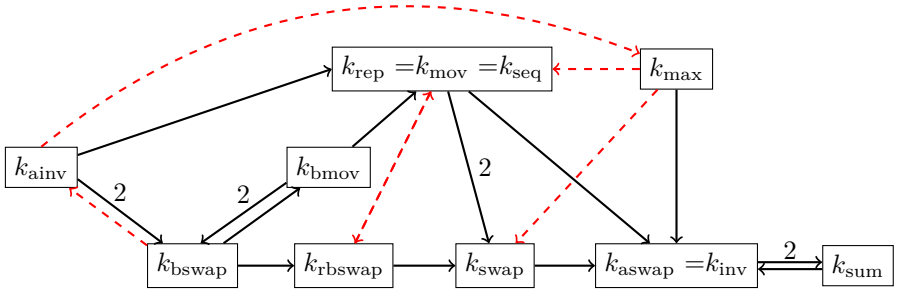

Figure 1 Overview of relations between measures of disorder. A solid black path from k to k' means that $k \leq ck'$, where c is the product of the edge labels along the path ($c = 1$ for unlabeled edges). If there is no solid black path from k to k', then k cannot be bounded by ck' for any constant c. Every arc is proved explicitly in the full paper (dashed red arcs correspond to unboundedness results), and *all* other relationships are implied.

expect all queries to yield correct results. We study a variety of parameters that quantify the disorder of A and provide algorithms and lower bounds with respect to the different parameters. Figure 1 gives the relationship between every pair of parameters.

4.1 Bounded displacement

We first consider the two parameters k_{sum} and $k_{\max}$ that quantify the displacement of elements between A and the sorted counterpart $A^\star$ of A. More precisely, we define $k_{\text{sum}} := \sum_{x \in A} |\operatorname{pos}(x) - \operatorname{rank}(x)|$ and $k_{\max} := \max_{x \in A} |\operatorname{pos}(x) - \operatorname{rank}(x)|$. We obtain the following bounds.

▶ **Theorem 5** (∗). $\log n/k_{\text{sum}} + 2k_{\text{sum}} + \mathcal{O}(1)$ *queries are necessary*[1] *and sufficient to find* e.

▶ **Theorem 6** (∗). $\log n/k_{\max} + 3k_{\max} + \mathcal{O}(1)$ *queries are necessary*[1] *and sufficient to find* e.

4.2 Inversions

We now consider the number of inversions between elements of the array A. More precisely, we define the number of inversions to be $k_{\text{inv}} := |\{i < j : A[i] > A[j]\}|$, and the number of adjacent inversions to be $k_{\text{ainv}} := |\{i : A[i] > A[i+1]\}|$. We have $k_{\text{sum}} \leq k_{\text{inv}} \leq 2k_{\text{sum}}$, therefore the results for k_{sum} (Theorem 5) carry over to k_{inv} with a gap of 2.

▶ **Corollary 7.** *Every search algorithm needs at least* $\log n/k_{\text{inv}} + 2k_{\text{inv}} + \mathcal{O}(1)$ *queries*[1]*, and we can find* e *obliviously with* $\log n/k_{\text{inv}} + 4k_{\text{inv}} + \mathcal{O}(1)$ *queries.*

In general, we cannot hope to obtain results of similar quality for the smaller parameter k_{ainv}; already for $k_{\text{ainv}} = 1$ any search algorithm needs to query all n elements.

▶ **Proposition 8.** *For* $k_{\text{ainv}} \geq 1$*, no algorithm can find* e *with less than* n *queries.*

Fortunately, we can do much better if the target e is guaranteed to be in the correct position relative to sorted order, i.e., if $\operatorname{pos}(e) = \operatorname{rank}(e)$. Note that this restriction still allows us to prove a lower bound on the necessary number of queries that is much larger than all preceding results. We complement this lower bound by a search algorithm that matches it tightly (up to lower-order terms). Both upper and lower bound hinge on the question of how efficiently (in terms of queries) an algorithm can find a good estimate of $\operatorname{rank}(e)$ by querying the array.

▶ **Theorem 9** (∗). *We can find e using $2\sqrt{2nk_{\text{ainv}}} + o(\sqrt{nk_{\text{ainv}}})$ queries if $\text{pos}(e) = \text{rank}(e)$.*

Proof Sketch. We know that if e is in the array then $\text{pos}(e) = \text{rank}(e)$. Since we know neither value beforehand, the algorithm proceeds by determining an estimate of $\text{rank}(e)$ over two stages of queries. For the first stage, we define a block size $p = c \cdot \sqrt{n/k_{\text{ainv}}}$ and query every $(p+1)$-st position. This partitions the array into (so far) unqueried blocks of size p, which we classify into <<-, >>-, <>-, and ><-blocks according to the query outcomes for the two positions adjacent to the block. Taking into account the number k_{ainv} of adjacent inversions, the number of blocks of each type gives rise to an upper and a lower bound on the number of elements that are smaller than e, and hence on the rank of e. Essentially, in <<- and >>-blocks with no adjacent inversion there are p respectively 0 smaller elements, whereas any number between 0 and p is possible if there is at least one adjacent inversion. In ><-blocks any number between 0 and p of smaller elements is possible, but these blocks must contain at least one adjacent inversion. In <>-blocks any number between 0 and p of smaller elements is possible without adjacent inversions, but the number of <>-blocks can be bounded by the number of ><-blocks and hence depending on k_{ainv}. Overall, this leads to a range of positions that certainly contains the position of e, if $e \in A$. Unfortunately, querying this range entirely would not lead to the claimed upper bound.

Upon second inspection, the unknown number of adjacent inversions in ><-blocks has the biggest impact on the size of the range. Accordingly, the second stage performs a binary search on each ><-block, which creates a number of (smaller) >>- and <<-blocks separated by an empty ><-block between two queried positions (where the binary search terminates). Thus, repeating the above analysis after the refinement, all ><-blocks are now of size zero and their impact on the range of possible positions is reduced. The algorithm can now afford to query this range entirely and either find e or be sure that it is not in A. The claimed bound is obtained by choosing the value of c in the block size p in order to balance the cost of making the initial queries and the cost of eventually querying the computed range. ◀

▶ **Theorem 10** (∗). *Every search algorithm needs at least $2\sqrt{2nk_{\text{ainv}}} - o(\sqrt{nk_{\text{ainv}}})$ queries*[1]*, even if $\text{pos}(e) = \text{rank}(e)$.*

Proof Sketch. We outline an adversarial strategy for replying to queries. This strategy needs to avoid revealing the position of e before the claimed number of queries, and simultaneously needs to maintain that at least one realization of the array remains that is consistent with the replies made so far, and that has at most k_{ainv} adjacent inversions and $\text{pos}(e) = \text{rank}(e)$.

At the beginning, our strategy replies with '<' to queries in the first half of A and with '>' to queries in the second half. At some point, we may not be able to place e in one of the two halves anymore: Say that the rightmost unqueried position in the left half has p positions on its right for which we already committed to '<'. Since $\text{pos}(e) = \text{rank}(e)$, the number of elements smaller than e and on its right must be equal to the number of larger elements on its left. Hence, a realization with e in the left half can only exist if there still remains a way of placing p elements larger than e in the left half (leaving the rightmost unqueried position for e), without causing more than k_{ainv} adjacent inversions. This is the case exactly if we can still find a set of (roughly) at most k_{ainv} disjoint blocks of consecutive, unqueried positions in the left half, which contain at least p unqueried positions overall. Analogously, at some point, we may not be able to place e in the left half anymore. Our strategy continues to reply to queries as before, until it has to commit to a realization and reply accordingly. Careful analysis of the strategy yields that committing to a position for e can be avoided until the claimed number of queries have been invested. ◀

4.3 Edit distances

We now consider parameters that bound the number of elementary array modifications needed to sort the given array A. More precisely, a *replacement* is the operation of replacing one element with a new element, and we let k_{rep} be the (minimum) number of replacements needed to obtain a sorted array. A *swap* is the exchange of the content of two array positions, and k_{swap} is the number of swaps needed to sort A. We let k_{aswap} be the number of swaps of pairs of neighboring elements needed to sort A. A *move* is the operation of removing an element and re-inserting it after a given position i, shifting all elements between old and new position by one. We let k_{mov} be the number of moves needed to sort A.

Clearly, starting from a sorted array, we can move e to any position, without using more than a single move or swap, or two replacements involving e. To find e we then have to query the entire array.

▶ **Proposition 11.** *For $k_{\text{mov}} \geq 1$, $k_{\text{swap}} \geq 1$, or $k_{\text{rep}} \geq 2$, no algorithm can find e with less than n queries in general.*

We can obtain significantly improved bounds if the element e remains at its correct position relative to the sorted array. Recall that we can interpret k_{faults} as a measure of disorder via $k_{\text{faults}}(e) = |i : (i < \text{pos}(e) \wedge A[i] > e) \vee (i > \text{pos}(e) \wedge A[i] < e)|$.

▶ **Lemma 12** (∗). *If $\text{rank}(e) = \text{pos}(e)$, then $k_{\text{faults}}(e) \leq \min\{2k_{\text{rep}}, 4k_{\text{swap}}, 2k_{\text{mov}}\}$.*

With this lemma, we can translate the upper bounds of any algorithm for k_{lies}. Before we do, we introduce another measure of disorder, that turns out to be closely related to k_{mov}. We define the parameter k_{seq} to be such that $n - k_{\text{seq}}$ is the length of a longest nondecreasing subsequence in A. It turns out that $k_{\text{mov}} = k_{\text{seq}}$, and we can thus include this parameter in our upper bound.

▶ **Theorem 13** (∗). *Let $f\colon \mathbb{N}^2 \to \mathbb{N}$. If $\text{rank}(e) = \text{pos}(e)$ and we can find e with $f(n, k_{\text{lies}})$ queries, then we can find e with $\min\{f(n, 2k_{\text{rep}}), f(n, 4k_{\text{swap}}), f(n, 2k_{\text{mov}}), f(n, 2k_{\text{seq}})\}$ queries.*

We can also carry over the lower bound from parameter k_{faults}.

▶ **Corollary 14.** *For every $c \in \mathbb{N}$ and $\text{pos}(e) = \text{rank}(e)$, no algorithm operating on the search tree can find e with less than $\log n + ck$ queries in general, for $k \in \{k_{\text{rep}}, k_{\text{swap}}, k_{\text{mov}}, k_{\text{seq}}\}$.*[1]

Finally, we immediately obtain bounds for k_{aswap} from Corollary 7, because $k_{\text{aswap}} = k_{\text{inv}}$.

▶ **Corollary 15.** *Every search algorithm needs at least $\log n/k_{\text{aswap}} + 2k_{\text{aswap}} + \mathcal{O}(1)$ queries*[1]*, and we can find e obliviously with $\log n/k_{\text{aswap}} + 4k_{\text{aswap}} + \mathcal{O}(1)$ queries.*

4.4 Block edit distances

In this section we consider the parameters k_{bswap}, k_{rbswap}, and k_{bmov}, which bound the number of block edit operations needed to sort A. A *block* is defined to be a subarray $A[i, i+1, \ldots, j]$ of consecutive elements. A *block swap* is the operation of exchanging a subarray $A[i, \ldots, j]$ with a subarray $A[i', \ldots, j']$ and vice versa, where $i < j < i' < j'$. Note that a block swap may affect the positions of other elements in case that the two blocks are of different sizes. The parameter k_{bswap} bounds the number of block swaps needed to sort A. For k_{rbswap} we only allow block swaps restricted to pairs of blocks of equal sizes. Finally, for k_{bmov} one of the two blocks must be empty, i.e., only block moves are allowed. For all three parameters one can easily prove that, without further restrictions, search algorithms need to query all positions of an array to find the target.

▶ **Proposition 16.** *For $k_{\text{bswap}} \geq 1$, $k_{\text{rbswap}} \geq 1$, or $k_{\text{bmov}} \geq 1$, no algorithm can find e with less than n queries.*

Complementing this lower bound, for all of three parameters, an upper bound of $\mathcal{O}(\sqrt{nk})$ for finding e when $\text{pos}(e) = \text{rank}(e)$ follows immediately from the results for k_{ainv} of Section 4.2 and the fact that $k_{\text{ainv}} \leq 2k_{\text{bswap}}$ and $k_{\text{bswap}} \leq \min\{k_{\text{rbswap}}, k_{\text{bmov}}\}$. By inspecting the upper and lower bounds proved for k_{ainv}, and adapting the proofs, we are able to obtain tight leading constants in the upper and lower bounds for k_{bmov} and k_{bswap}, and leading constants within a factor of $\sqrt{2}$ for k_{rbswap}.

▶ **Theorem 17** (∗). *If $\text{pos}(e) = \text{rank}(e)$, then $4\sqrt{nk_{\text{bswap}}} + o(\sqrt{nk_{\text{bswap}}})$ queries are necessary*[1] *and sufficient to find e.*

▶ **Theorem 18** (∗). *If $\text{pos}(e) = \text{rank}(e)$, then $2\sqrt{2nk_{\text{rbswap}}} + o(\sqrt{nk_{\text{rbswap}}})$ queries are necessary*[1] *and $4\sqrt{nk_{\text{rbswap}}} + o(\sqrt{nk_{\text{rbswap}}})$ queries are sufficient to find e.*

▶ **Theorem 19** (∗). *If $\text{pos}(e) = \text{rank}(e)$, then $2\sqrt{2nk_{\text{bmov}}} + o(\sqrt{nk_{\text{bmov}}})$ queries are necessary*[1] *and sufficient to find e.*

5 Conclusion

We presented upper and lower bounds for the worst-case query complexity of comparison-based search algorithms that are robust to persistent and temporary read errors, or are adaptive to partially disordered input arrays. For many cases we gave algorithms that are optimal up to lower order terms. In addition, many of the algorithms are oblivious to the value of the parameter quantifying errors/disorder, assuming the target element is present in the array. In most cases, for small values of k, the dependence of our algorithms on the number n of elements is close to $\log n$, with only additive dependency on the number of imprecisions. In other words, these results smoothly interpolate between parameter regimes where algorithms are as good as binary search and the unavoidable worst-case where linear search is best possible.

That said, why should one be interested in, e.g., almost tight bounds relative to the number of block moves that take A to a sorted array, as the bounds are far from binary search? The point is that only the total number of comparisons matter, and having a worse function that depends on a (in this case) much smaller parameter value can be favorable to having a much better function of a large parameter value. E.g., after a constant number of block swaps the parameters $k_{\max}$, k_{sum} etc. may have value $\Omega(n)$ and the guaranteed bound becomes trivial, while running the search algorithm for the case of few block swaps guarantees $\mathcal{O}(\sqrt{n})$ comparisons. Similarly, having tight bounds for the various parameters gives us the exact (worst-case) regime for the chosen parameter (in terms of n) where a sophisticated algorithm can outperform linear search, or even be as good as binary search.

Despite having already asymptotic tightness, it would be interesting to close the gaps between coefficients of dominant terms in upper and lower bounds for some of the cases. Another question would be to find a different restriction than $\text{pos}(e) = \text{rank}(e)$, i.e., the target being in the correct position relative to sorted order, that avoids degenerate lower bounds of $\Omega(n)$ queries for several parameters. A relaxation to allowing a target displacement of ℓ and giving cost in terms of n, k, and ℓ seems doable in most cases, but is unlikely to be particularly insightful. Finally, it seems interesting to study whether randomization could lead to improved algorithms for some of the cases. The analysis of randomized lower bounds requires entirely new adversarial strategies since the adversary must choose an instantiation without access to the random bits of the algorithm.

Acknowledgements. The authors are grateful to several reviewers for their helpful remarks regarding presentation and pertinent literature references.

References

1 Arne Andersson, Torben Hagerup, Johan Håstad, and Ola Petersson. Tight bounds for searching a sorted array of strings. *SIAM J. Comput.*, 30(5):1552–1578, 2000. doi:10.1137/S0097539797329889.

2 Stanislav Angelov, Keshav Kunal, and Andrew McGregor. Sorting and selection with random costs. In *Proceedings of the 8th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 48–59, 2008. doi:10.1007/978-3-540-78773-0_5.

3 Javed A. Aslam and Aditi Dhagat. Searching in the presence of linearly bounded errors. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 486–493, 1991.

4 Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013. doi:10.1016/j.tcs.2013.10.019.

5 Therese C. Biedl, Timothy M. Chan, Erik D. Demaine, Rudolf Fleischer, Mordecai J. Golin, James A. King, and J. Ian Munro. Fun-sort–or the chaos of unordered binary search. *Discrete Applied Mathematics*, 144(3):231–236, 2004. doi:10.1016/j.dam.2004.01.003.

6 Biagio Bonasera, Emilio Ferrara, Giacomo Fiumara, Francesco Pagano, and Alessandro Provetti. Adaptive search over sorted sets. *J. Discrete Algorithms*, 30:128–133, 2015. doi:10.1016/j.jda.2014.12.007.

7 Ryan S. Borgstrom and S. Rao Kosaraju. Comparison-based search in the presence of errors. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 130–136, 1993. doi:10.1145/167088.167129.

8 Allan Borodin, Leonidas J. Guibas, Nancy A. Lynch, and Andrew Chi-Chih Yao. Efficient searching using partial ordering. *Inf. Process. Lett.*, 12(2):71–75, 1981. doi:10.1016/0020-0190(81)90005-3.

9 Gerth Stølting Brodal, Rolf Fagerberg, Irene Finocchi, Fabrizio Grandoni, Giuseppe F. Italiano, Allan Grønlund Jørgensen, Gabriel Moruz, and Thomas Mølhave. Optimal resilient dynamic dictionaries. In *Proceedings of the 15th Annual European Symposium on Algorithms (ESA)*, pages 347–358, 2007.

10 F. Warren Burton and Gilbert N. Lewis. A robust variation of interpolation search. *Inf. Process. Lett.*, 10(4/5):198–201, 1980. doi:10.1016/0020-0190(80)90139-8.

11 Ferdinando Cicalese. *Fault-Tolerant Search Algorithms – Reliable Computation with Unreliable Information*. Springer, 2013. doi:10.1007/978-3-642-17327-1.

12 Aditi Dhagat, Peter Gacs, and Peter Winkler. On playing "twenty questions" with a liar. In *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 16–22, 1992.

13 Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992. doi:10.1145/146370.146381.

14 Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994.

15 Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. *Theor. Comput. Sci.*, 410(44):4457–4470, 2009. doi:10.1016/j.tcs.2009.07.026.

16 Irene Finocchi and Giuseppe F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, 2008. doi:10.1007/s00453-007-9088-4.

17 Gianni Franceschini and Roberto Grossi. No sorting? better searching! *ACM Transactions on Algorithms*, 4(1), 2008. doi:10.1145/1328911.1328913.

18 Michael L. Fredman. The number of tests required to search an unordered table. *Inf. Process. Lett.*, 87(2):85–88, 2003. doi:10.1016/S0020-0190(03)00260-6.

19 Anupam Gupta and Amit Kumar. Sorting and selection with structured costs. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science, (FOCS)*, pages 416–425, 2001. doi:10.1109/SFCS.2001.959916.

20 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

21 Philip M. Long. Sorting and searching with a faulty comparison oracle. Technical Report UCSC-CRL-92-15, University of California at Santa Cruz, 1992.

22 Harry G. Mairson. Average case lower bounds on the construction and searching of partial orders. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 303–311, 1985. doi:10.1109/SFCS.1985.13.

23 Kurt Mehlhorn. Sorting presorted files. In *Proceedings of the 4th GI-Conference on Theoretical Computer Science*, pages 199–212, 1979. doi:10.1007/3-540-09118-1_22.

24 Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984. URL: http://www.mpi-sb.mpg.de/~mehlhorn/DatAlgbooks.html.

25 S. Muthukrishnan. On optimal strategies for searching in the presence of errors. In *Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 1994.

26 Andrzej Pelc. Searching with known error probability. *Theoretical Computer Science*, 63(2):185–202, 1989.

27 Andrzej Pelc. Searching games with errors – fifty years of coping with liars. *Theoretical Computer Science*, 270(1-2):71–109, 2002.

28 Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995. doi:10.1016/0166-218X(93)E0160-Z.

29 Erez Petrank and Guy N. Rothblum. Selection from structured data sets. *Electronic Colloquium on Computational Complexity (ECCC)*, 2004. TR04-085. URL: http://eccc.hpi-web.de/eccc-reports/2004/TR04-085/index.html.

30 Ronald L. Rivest, Albert R. Meyer, Daniel J. Kleitman, Karl Winklmann, and Joel Spencer. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20(3):396–404, 1980.

31 Robert Sedgewick. *Algorithms in* C++ *– Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley-Longman, 1998.

32 Andrew Chi-Chih Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981. doi:10.1145/322261.322274.