Nearly Tight Bounds for Discrete Search under Outlier Noise

Anindya De* Sanjeev Khanna* Huan Li* Hesam Nikpey*

Abstract

Binary search is one of the most fundamental search routines, exploiting the hidden structure of the search space. In particular, it cuts down exponentially on the complexity of the search assuming that the search space is monotone. This paper is prompted by a basic question – how does the query complexity of the search problem change if the data has corruption? In particular, we study the powerful *outlier noise model* and assuming a bound on the fraction of such corruptions, establish nearly matching upper and lower bounds for the following problems: (i) binary search on an ordered set of size $[n]$; (ii) search on the posets $\{0,1\}^d$; and (iii) search on the posets $[n]^d$. In all three cases, we use randomization to create robust versions of classical algorithms for these problems that handle corrupted data with relatively small performance penalties, specified as a function of the amount of corruption K. We complement these algorithmic results with almost matching lower bounds that show that no randomized algorithm can solve these problems with a smaller performance hit on the query complexity as a function of K.

1 Introduction

Search problems are among the most fundamental class of computational problems. While it is possible to always perform an exhaustive search on any domain, sometimes algorithms can substantially improve upon this by leveraging underlying structure. Probably, the simplest and most prominent example of this is the *binary search*. In particular, suppose $f : [n] \to \mathbb{R}$ defines a monotone (say increasing) function. Then, given an oracle for f and a target element x, the binary search algorithm makes $\lceil \log n \rceil$ calls to f and finds $i \in [n]$ such that $f(i) = x$ (provided there exists such an i).

The high level question addressed in this paper is the following: *Can we design noise tolerant versions of non-trivial search routines like binary search?* In particular, the usual guarantee of routines like binary search requires that there is *no noise* in the oracle for f, a rather strong and sometimes unrealistic modeling assumption. In this paper, we study the problem when the oracle for f is noisy.

1.1 Our results

This work studies the basic (search) problem in computation – *binary search* – with the goal of making it robust to noise. In particular, we study the powerful *outlier noise model*, where the only assumption we make about the noise is that there is an upper bound on the fraction of corrupted points. We note that there are plenty of works considering stochastic noise [KK07, BM08] or noise that is bounded in magnitude [HS17, SV15, HS16]. We will discuss more about these works in our next subsection.

1.1.1 Noisy binary search

The first problem we consider is the classical problem of binary search. Suppose, we are given oracle access to a function $f : [n] \to \mathbb{R}$ such that f is monotonically increasing, i.e., $f(i) \geq f(j)$ whenever $i \geq j$. Then, given any element x, binary search can make $\lceil \log n \rceil$ queries to f and (i) if there exists i^* such that $f(i^*) = x$, then return i^*; (ii) return $\perp$ otherwise. Now, consider a "corrupted" monotone function where up to K entries have been altered in an arbitrary manner. In other words, we get oracle access not to f but to $\tilde{f} : [n] \to \mathbb{R}$ such that $|x : f(x) \neq \tilde{f}(x)| \leq K$. In other words, the function $\tilde{f}$ defined by the oracle is guaranteed to be K-close to the "true" monotone function f. We refer to the oracle for $\tilde{f}$ as K-corrupted. We are interested in the complexity of searching for an element x given a corrupted oracle $\tilde{f}$. We now make two crucial assumptions about the target element x: (i) the element x itself is not corrupted – i.e., if $f(i) = x$, then $\tilde{f}(i) = f(i)$. (ii) if $f(j) \neq \tilde{f}(j)$, then $\tilde{f}(j) \neq x$. In other words, we do not introduce a corruption whose value is x.

*University of Pennsylvania. Email: {anindyad,sanjeev,huanli,hesam}@cis.upenn.edu.

Assumption (i) is easily seen to be necessary – if we are searching for a corrupted element x, it is easy to see that the query complexity can grow as $\Omega(n)$. To justify Assumption (ii), suppose f is the function $f(i) = i$ and $\tilde{f}$ is the corrupted version which is the same as f except $\tilde{f}(n) = n+1$. Then, note that if the target element is $n+1$, then we expect our algorithm to return "NO" as $n+1 \notin \mathsf{Range}(f)$. On the other hand, given oracle access to $\tilde{f}$ (which is now itself monotone), even with an arbitrary number of queries, it is not possible to tell that $n+1$ is a corruption and not present in $\mathsf{Range}(f)$. Our first main result is stated below (informally).

INFORMAL THEOREM 1.1. *There is a randomized algorithm which given query access to a K-corrupted monotone $\tilde{f}$ and an uncorrupted element $x \in \mathsf{Range}(f)$ has the following guarantee: with $O(K + \log n)$ queries, it produces an index $i^* \in [n]$ such that $f(i^*) = x$. On the complementary side, we show that any randomized algorithm for this problem requires $\Omega(K + \log n)$ queries.*

In fact, we prove a stronger guarantee, namely, there is a randomized algorithm that uses $O(\log(n/K))$ queries to trap the index i^* (if it exists) in an interval of length slightly above $2K$. On the other hand, we show that any randomized algorithm that narrows down i^* to any interval of size $2K$ must necessarily use $\Omega(K)$ queries.

We refer the reader to Theorem 2.1 for the formal theorem of this statement. Note that the above theorem can be interpreted as follows: Suppose $\mathcal{A}$ is a sorted array with K corruptions. Then, we can narrow down the target index to an interval of size approaching $2K$ with $\Omega(\log(n/K))$ queries. *A fortiori*, this implies that the unknown element can be found with queries $O(K + \log n)$. Note that $\log n$ is just the time required for binary search and thus, the additional overhead we require to handle K corruptions is just $O(K)$. Further, the linear dependence on K is necessary. In particular, if $n = K + 1$, then with K corruptions, we have an arbitrary array and thus search in this array will necessarily take $\Omega(K)$ queries. On the complementary side, we also show a sharp threshold behavior at $2K$: namely, narrowing down the target index to an interval of size $2K$ or smaller requires $\Omega(K)$ queries.

To explain the high level idea behind our algorithm, note that in the usual binary search, when we query an element $f(i)$, we move "left" or "right" depending on whether $f(i) > x$ or $f(i) < x$ (and return i if $f(i) = x$). This search routine is of course extremely brittle to noise as it is possible that the point i that the algorithm queries is itself corrupted. To circumvent this problem, we instead query a randomly chosen index j in an interval of length M around i. We now move "left" or "right" depending on whether $f(j) > x$ or $f(j) < x$. It turns out that as long as M exceeds $2K$ by a constant factor, using such a randomly chosen j is a noise-tolerant way to choose a direction in binary search. This lets us essentially locate the target index to an interval of size $O(K)$. Once we have reduced the problem to this case, it suffices to a do a brute-force search. This leads to a query complexity of $O(K + \log n)$.

1.1.2 Noisy search on the hypercube We now move to search over partially ordered sets (posets) – in contrast, binary search is meant to find an item in a totally ordered set. To state the problem, define a function $f : [n]^d \to \mathbb{R}$ to be monotonically increasing if $x \succeq y$ implies $f(x) \geq f(y)$. Here $x \succeq y$ if for all $1 \leq i \leq d$, $x_i \geq y_i$. Note that the grid $[n]^d$ is a basic example of a poset and thus it is natural to ask if we can generalize "noisy binary search" from a totally ordered set to this specific poset. Before we discuss the problem of noisy search on the grid, let us first recall the complexity of search on the grid in the noiseless case. The *monotone search problem* is defined as follows: given oracle access to a monotone function $f : [n]^d \to \mathbb{R}$ and $x \in \mathsf{Range}(f)$, find α such that $f(\alpha) = x$.

For simplicity, we first focus on the case $n = 2$ – i.e., the domain is equivalently $\{0,1\}^d$. The complexity of the monotone search problem over $\{0,1\}^d$ is $\Theta(2^d/\sqrt{d})$ [KTDV96]. The proofs of both upper and lower bounds crucially depend on two combinatorial concepts – *chain* and *anti-chain*.

DEFINITION 1.1. *For a poset $\mathcal{X}$, a* chain *is a sequence $x_1, \ldots, x_N \in \mathcal{X}$ such that $x_1 \preceq x_2 \ldots \preceq x_N$.*

An anti-chain *is a sequence $x_1, \ldots, x_N \in \mathcal{X}$ such that for any $i \neq j$, x_i and x_j are not comparable. In other words, neither $x_i \succeq x_j$ nor $x_j \succeq x_i$.*

The algorithm for monotone search $\{0,1\}^d$ relies on splitting the domain into a so-called *symmetric chain decomposition* (see Definition 3.1). Roughly speaking, a symmetric chain decomposition splits the domain $\{0,1\}^d$ into $\Theta(2^d/\sqrt{d})$ chains, each of size d. With such a chain decomposition, the search is easy – namely, for each chain, do a binary search to find the unknown x. Each chain is of length d and thus the total time for monotone search is $O((2^d \log d)/\sqrt{d})$.

To obtain the lower bound, we rely on the existence of the antichain $\mathcal{X}_{\mathsf{anti}}$ of size $\Omega(2^d/\sqrt{d})$. Note that since an anti-chain is an incomparable set of points, such an anti-chain can be used to embed problem of searching an unordered array of $|\mathcal{X}_{\mathsf{anti}}|$ inside the monotone search problem on $\{0,1\}^d$. This also gives a lower bound of $\Omega(2^d/\sqrt{d})$ on the complexity of the monotone search problem on $\{0,1\}^d$.

We now turn our attention to the noisy search problem on $\{0,1\}^d$. As was the case with the previous subsection, we define $\tilde{f} : \{0,1\}^d \to \mathbb{R}$ to be K-corrupted monotone if there is a "true" monotone function $f : \{0,1\}^d \to \mathbb{R}$ such that f and $\tilde{f}$ differ in at most K places. Similar to Theorem 1.1, we stipulate that (i) the target element x is itself not a corruption; (ii) if $\tilde{f}(i) \neq f(i)$, then the corruption $\tilde{f}(i) \neq x$. Our main result here is informally stated below.

INFORMAL THEOREM 1.2. *There is a randomized algorithm which given query access to a K-corrupted monotone $\tilde{f} : \{0,1\}^d \to \mathbb{R}$ and an uncorrupted element $x \in \mathsf{Range}(f)$ has the following guarantee: with $O(\min\{2^d, \frac{2^d}{\sqrt{d}}(\log d + \log K)\})$ queries, it finds an i^* such that $f(i^*) = x$. On the complementary side, we show that any randomized algorithm to find i^* requires $\Omega(\min\{2^d, \frac{2^d}{\log d\sqrt{d}} \log K\})$ queries.*

Our upper bound crucially uses the notion of symmetric chain decomposition. In particular, we first randomize our chain decomposition by randomly permuting the coordinates. This ensures that the average number of corruptions contained in any chain $\approx O(\log K)$. Now, within each chain, we essentially apply a noise tolerant binary search (Theorem 1.1) to obtain the final upper bound on the complexity.

To obtain our lower bound, we first use Yao's minimax lemma to reduce the problem to showing a lower bound for deterministic algorithms. Subsequently, we leverage the structure of the hypercube, in particular, that (i) the middle layer of $\{0,1\}^d$, i.e., the points with $d/2$ ones are an antichain; (ii) for any a in the middle layer, the number of points b in the next ℓ layers such that $b \succeq a$ can be bounded by $\approx d^\ell$. Roughly speaking, our hard instance starts with a function $\tilde{f}$ with the property that (i) if a is above layer $d/2 + \log_d K$, $\tilde{f}(a) = \infty$ and (ii) if a is below layer $d/2 - \log_d K$, $\tilde{f}(a) = -\infty$. All the other elements are filled with positive integers respecting the order defined by the hypercube. Now, we randomly choose a position a^* from the layers $d/2 - \log_d K$ to $d/2 + \log_d K$ and label $\tilde{f}(a^*) = 0$. We let the target element x be zero – using property (ii) of the hypercube, it is easy to see that $\tilde{f}$ differs from a monotone function f in at most K places. Exploiting the size of the antichains in the hypercube, it is not difficult to show that any deterministic algorithm must essentially now do a brute force search for the target element in the middle $d/2 \pm \log_d K$ layers. This finishes the proof sketch for the lower bound.

1.1.3 Search on the grid $[n]^d$ We now turn to our results for the high-dimensional grid $[n]^d$. Note that vis-a-vis the hypercube, the alphabet size is now n as opposed to 2 (for the hypercube). Similar to the hypercube, it is first instructive to understand the complexity of search *without errors*.

The complexity of the monotone search problem on the grid without errors turns out to be (up to $\log(nd)$ factors) $\Theta(n^{d-1}/\sqrt{d})$ – note that this saves a factor of $n\sqrt{d}$ over the trivial search algorithm whose complexity is $\Theta(n^d)$. While we suspect that this is a folklore fact, we could not find a reference for this. The upper bound can be achieved by doing a symmetric chain decomposition of the grid and performing a binary search on each chain. The lower bound is immediate from the fact that the set $A_{\mathsf{mid}} = \{x : x_1 + \ldots + x_n = \lceil nd/2 \rceil\}$ is an antichain of size $\Omega(n^{d-1}/\sqrt{d})$. As we can always embed unordered search inside an antichain, this gives a lower bound of $\Omega(n^{d-1}/\sqrt{d})$. The next informal theorem describes the complexity of monotone search on $[n]^d$ in presence of errors.

INFORMAL THEOREM 1.3. *There is a randomized algorithm* Search-corrupt-monotone-grid *which given query access to a K-corrupted monotone $\tilde{f} : [n]^d \to \mathbb{R}$ and an uncorrupted element $x \in \mathsf{Range}(f)$ has the following guarantee: with $O(\min\{n^d, n^{d-1}K^{1/d}d + \frac{n^{d-1}d}{K^{1-(1/d)}} \log(n) \log\log(n)\})$ queries, it finds an i^* such that $f(i^*) = x$. On the complementary side, we show that any randomized algorithm to find i^* requires $\Omega(\min\{n^d, n^{d-1}K^{1/d}\sqrt{d}\})$ queries.*

Note that our upper and lower bounds for the K-corrupted monotone search can be off by a factor of about $\sqrt{d}$ in the worst-case. Further, vis-a-vis, Theorem 1.2, the dependence on K is $K^{1/d}$ (as opposed to $\log K$). We also note that our lower bound in this theorem only applies for n greater than some sufficiently large constant ($10e$ suffices, see our proof in Section 4.2), and thus it does not apply to the Boolean cube.

The proof of the lower bound is similar to that of Theorem 1.2. Specifically, we randomly choose a position a^* from the points with ℓ_1-norm between $dn/2 - dK^{1/d}/(20e)$ to $dn/2 + dK^{1/d}/(20e)$ and plant the target there. Then essentially repeating the same argument as Theorem 1.2 yields the lower bound in Theorem 1.3.

The upper bound in Theorem 1.3 follows a somewhat different idea. Namely, we tile the grid $[n]^d$ by, what we refer to as, "cubic diagonals". Roughly speaking, a cubic diagonal is a collection of cubes $\{C_1, \ldots, C_t\}$ such that if $j > i$, then $x_j \succeq x_i$ for any $x_j \in C_j$ and $x_i \in C_i$. We show that the grid can be covered with $O(d(n/K^{1/d})^{d-1})$ such cubic diagonals. Once we achieve such a tiling, we do a binary search on the "diagonals" and brute force search inside the cubes to obtain the final time complexity.

1.2 Related Work We note that the high level question of "making algorithms robust to noise" is by no means new. In particular, several papers in the past have studied the problem of binary search (and its generalizations) where the data is erroneous [Now09, BOH08, KK07]. All these papers model noise as a probabilistic process. As an example, in [KK07], for every comparison query (i, j), the algorithm gets the true order with probability $1 - \delta$ and false order with probability δ. In related vein is the long line of work on noisy sorting. This includes the papers [BM08, RV17, AFHN15, BMP19]. Similar to the work on noisy binary search, the modeling of noise in these papers is also probabilistic.

There are also some early works that have considered the question of solving search problems with an erroneous oracle [RMK$^+$80, DGW92, FRPU94, Pel02]. We refer the reader to [Pel02] for a comprehensive survey on results of this flavor. Of a particular note is the paper by Rivest et al [RMK$^+$80] which studies the problem of finding an unknown value $x \in [n]$ using comparisons of x to any chosen $y \in [n]$, when at most K of the comparisons may be incorrect. The authors give a tight bound of $\log_2 n + K \log_2 \log_2 n + O(K \log_2 K)$ on the number of comparisons needed to determine the value x. However, we remark that their results are incomparable to ours since in their model the adversary can be *adaptive*—the adversary can decide whether to corrupt a comparison based on the comparisons that the algorithm has made so far. In contrast, in our model the adversary is oblivious and has to fix a (corrupted) input beforehand. In fact, in their model randomization does not help, since one can easily show that there is always an optimal strategy that is deterministic. All our algorithms, on the other hand, crucially rely on randomization.

Two recent papers [DFGM17, DFKM21] studied an extension of the problem in [RMK$^+$80], where the target is chosen from a distribution μ on $[n]$ known to the algorithm, and the performance of the algorithm is measured by comparing the number of comparisons (or other kinds of queries) it makes to the entropy of μ. Very recently, the authors also studied optimization of convex functions under outlier noise, where we want to get close to the minimizer of a convex function that is corrupted in a bounded region [DKLN21]. Such a problem may be seen as a continuous analog of the discrete search problems studied in this paper.

1.3 Organization We present our upper and lower bounds for searching in a corrupted monotone array, Boolean cube, and monotone higher-dimensional grid in Sections 2, 3, and 4, respectively.

2 Search in a Corrupted Monotone Array

We consider the problem of locating a given target element in a corrupted monotone array of length n where at most K elements are corrupted. We will show that the query complexity of this problem is $\Theta(\log n/K + K)$. We focus here on the algorithmic upper bound on the query complexity, and defer the lower bound results to the Appendix.

At a high-level, our algorithm performs the search in two phases. In the first phase it traps the target to an interval of length $10K$, and in the second phase, it does an exhaustive search on all of the elements. The theorem below summarizes the exact performance guarantee achieved by our algorithm.

THEOREM 2.1. *Let A be a monotone array of n numbers such that at most K of them are corrupted. Then, there is an algorithm that finds a target t in the array using $O(\log \frac{1}{\delta} \cdot (\log(n/K) + K))$ queries with success probability $1 - \delta$, where $0 < \delta < 1$.*

Proof. We will create a robust version of the binary search that will allow us at each step to reduce the search space by a factor of half as long as it is larger than $10K$. At this point, the algorithm switched to an exhaustive search in the remaining interval. Specifically, the algorithm finds a target t in array A as follows:

1. while $|A| > 10K$

(a) Let m be the mid-point of A.

(b) Compare t with a random element in $A[m - 3K/2, m + 3K/2]$. If the element is t, we are done.

(c) If t is the bigger one, discard from A the elements to the left of index $m - 3K/2$, otherwise discard the elements to the right of index $m + 3K/2$.

2. Perform an exhaustive search on the remaining elements.

Line 1 repeats at most $O(\log n/K)$ times, and in each repetition the algorithm queries one element. So it makes at most $\log n/K$ queries at this step. For Line 2, it makes at most $10K$ queries. So the total query complexity of the above algorithm is $O(\log n/K + K)$.

Now we prove that with constant probability the algorithm correctly finds the target t. We say the algorithm makes an error in the i'th step of Line 1 if the algorithm deletes the part of the array that contains t, and we use e_i to denote the probability of the error. If the algorithm queries an uncorrupted target, error does not happen. So the probability that the algorithm does not make an error in i'th step is at least $1 - \frac{v_i}{3K}$ where v_i is the number of corrupted elements in the middle interval of length $3K$. By union bound, the probability that error never happens in any steps is $1 - \sum_{i=1}^{\log(n/K)} \frac{v_i}{3K}$. As the intervals of length $3K$ are disjoint, we have $\sum_{i=1}^{\log(n/K)} \frac{v_i}{3K} \leq \frac{K}{3K} = 1/3$. So the probability that the algorithm traps t successfully at the end of Line 1 is at least $2/3$. When the algorithm traps the target, it will successfully find the target in Line 2. By repeating the above algorithm for $\log 1/\delta$ times, the algorithm finds t with probability $1 - \delta$ using $O(\log(\frac{1}{\delta}) \cdot (\log(n/k) + K))$ queries. ∎

The idea outlined in the Line 1 of the algorithm can also be extended to show the following result.

THEOREM 2.2. *Let A be a monotone array of n numbers such that at most K of them are corrupted. Then, there is an algorithm that returns an interval of length at most $l = (2 + \epsilon)K$ that contains the target t w.p. $1 - \delta$ using $O(\log(n/K)\log(1/\delta)) + O(1/\epsilon^2 \log(1/\epsilon)\log\log(1/\epsilon)\log(1/\delta))$ queries, for any $\epsilon > 0$ and $0 < \delta < 1$.*

A proof of the above algorithm is provided in Appendix A. Thus even in a corrupted array, essentially binary search can be used to narrow down the target to an interval of length $2K$. Interestingly, we show that a phase transition occurs in the problem at the threshold of $2K$ in that any further reduction in the length of the interval containing the target requires a linear number of queries.

THEOREM 2.3. *For a corrupted monotone array A consisting $2n$ numbers where half of them are corrupted and a target t, any randomized algorithm that makes at most cn queries on A and returns an interval of $(2 - \epsilon)n$, has an error of at least $\frac{\epsilon - c}{2}$.*

A proof of the above lower bound is provided in Appendix B.

3 Search in a Corrupted Monotone Boolean Cube

We consider here the problem of searching for a target t in a monotone d-dimensional Boolean cube where at most K elements have been corrupted. We show matching upper and lower bounds (up to $\log^{O(1)} d$ factors) of

$$\min\left\{\frac{2^d}{\sqrt{d}}\log K,\ 2^d\right\}$$

on the query complexity for finding the target's exact location. As a result, when $K = 2^{\Omega(\sqrt{d})}$, we need to query a (nearly) constant fraction of the points in $\{0,1\}^d$ in order to find the target.

Preliminaries. Let $f : \{0,1\}^d \to \mathbb{Z}$ denote a Boolean cube filled with integers. For two points $x, y \in \{0,1\}^d$, we say x *precedes* y (denoted by $x \preccurlyeq y$) if $x_i \leq y_i, \forall i \in [d]$, and x *succeeds* y (denoted by $x \succcurlyeq y$) if $x_i \geq y_i, \forall i \in [d]$. We also say x *strictly* precedes (succeeds) y if $x \preccurlyeq y$ ($x \succcurlyeq y$) but $x \neq y$, denoted by $x \prec y$ ($x \succ y$). We say x and y are *incomparable* if x neither precedes nor succeeds y.

The boolean cube can be partitioned into $d + 1$ layers. The i-th layer L_i contains all points with Hamming weight i.

For some $x \in \{0,1\}^d$ on layer l and an integer $i \geq l$, we write $\mathsf{S}_{x,i}$ to denote the successors of x on layer i, i.e. $\{y : y \succcurlyeq x\} \cap \mathsf{L}_i$. Similarly for $i \leq l$ we write $\mathsf{P}_{x,i}$ to denote the predecessors of x on layer i, i.e. $\{y : y \preccurlyeq x\} \cap \mathsf{L}_i$. We then define $\mathsf{S}_{x,\leq i} \stackrel{\text{def}}{=} \bigcup_{j=l}^{i} \mathsf{S}_{x,j}$ and $\mathsf{P}_{x,\geq i} \stackrel{\text{def}}{=} \bigcup_{j=i}^{l} \mathsf{P}_{x,j}$.

Problem Setting. Let $f : \{0,1\}^d \to \mathbb{Z}$ be a monotone function such that $f(x) < f(y)$ for all $x \prec y$. An adversary arbitrarily modifies the value of f on at most K points, and obtain a corrupted function $\tilde{f}$. Let $\mathsf{C} \subseteq \{0,1\}^d$ where $|\mathsf{C}| \leq K$ denote the set of corrupted points (i.e. the points where f and $\tilde{f}$ differ). Given query access to $\tilde{f}$, we need to find the location of some target number $t \in \mathbb{Z}$. It is guaranteed that $\tilde{f}(x^*) = t$ for a unique $x^* \in \{0,1\}^d$, and that $x^* \notin \mathsf{C}$.

3.1 An $O(\frac{2^d}{\sqrt{d}}(\log K + \log d))$ Query Algorithm

We will use a *symmetric chain decomposition* of the boolean cube, which was first used in [Han66] for exact learning of a monotone Boolean function.

Definition 3.1. (Chains) *$\mathsf{X} = (x_1, \ldots, x_{|\mathsf{X}|})$ where each $x_i \in \{0,1\}^d$ is a chain if $x_i \prec x_{i+1}$ for all $i < |\mathsf{X}|$. X is consecutive if for all $i < |\mathsf{X}|$, there is no point y with $x_i \prec y \prec x_{i+1}$. It is symmetric if it is consecutive, and for some $0 \leq l \leq d$ we have $x_1 \in \mathsf{L}_l$ and $x_{|\mathsf{X}|} \in \mathsf{L}_{d-l}$.*

Theorem 3.1. (Symmetric Chain Decomposition [Han66]) *There is symmetric chain decomposition $\mathsf{X}_1, \ldots, \mathsf{X}_m$ of the d-dimensional Boolean cube such that*

1. *Each point $x \in \{0,1\}^d$ belongs to exactly one of the X_i's.*
2. *Each X_i is symmetric (thus also consecutive).*

This decomposition can be constructed in $O(2^d)$ time.

A direct consequence is that the number of chains in the decomposition is

$$m = \binom{d}{\lfloor d/2 \rfloor} = O(2^d/\sqrt{d}).$$

Our algorithm works as follows:

1. Construct a symmetric chain decomposition $\mathsf{X}_1, \ldots, \mathsf{X}_m$.
2. Repeat for $10 \log(1/\delta)$ times:
   (a) Pick a random permutation to permute the coordinates of all points in $\mathsf{X}_1, \ldots, \mathsf{X}_m$.
   (b) For each chain X_i, use the noisy binary search algorithm in Theorem 2.2 to narrow it down to an interval of length $100 \log K$ that (allegedly) traps t, with the failure probability set to $\delta = 0.1$.
   (c) Search t in each of the intervals of length $100 \log K$ by brute-force. If found, return the index.

We note that the idea of randomly permuting the coordinates of a fixed chain decomposition was used in [RRSW11] for approximating the influence of a monotone Boolean function.

Theorem 3.2. *The above algorithm finds the target t using $O(\frac{2^d}{\sqrt{d}}(\log K + \log d)\log(1/\delta))$ queries with success probability at least $1 - \delta$.*

We first state a key lemma showing that the chain containing the target is only mildly corrupted in expectation.

Lemma 3.1. *After applying the random permutation to the chains on Line 2a, let X_{i^*} be the chain containing x^* (the location of the target t). Then the expected size of $\mathsf{C} \cap \mathsf{X}_{i^*}$ is at most $20 \log K$ (recall that C is the set of corrupted points).*

Proof. If $|\mathsf{X}_{i^*}| \leq 20 \log K$, the lemma follows immediately. We thus only need to consider the case when $|\mathsf{X}_{i^*}| > 20 \log K$. Let x_j denote the intersection point of X_{i^*} and layer j. Let s, t be the smallest and largest layer intersecting X_{i^*}, and let $l \in [s,t]$ be the layer containing x^*. Conditioned on that X_{i^*} contains x^*, we have that

1. $\forall j \in [s,l]$, x_j follows a uniform distribution on $\mathsf{P}_{x^*,j}$ (predecessors of x^* on layer j).
2. $\forall j \in [l,t]$, x_j follows a uniform distribution on $\mathsf{S}_{x^*,j}$ (successors of x^* on layer j).

Then we have (all expectations/probabilities are conditioned on $x^* \in \mathsf{X}_{i^*}$)

$$\begin{aligned}\mathbf{E}\left[|\mathsf{C} \cap \mathsf{X}_{i^*}|\right] &= \sum_{j=s}^{t} \mathbf{Pr}\left[x_j \in \mathsf{C}\right] \qquad \text{(linearity of expectation)} \\ &= \sum_{j=s}^{l-1} \frac{|\mathsf{C} \cap \mathsf{P}_{x^*,j}|}{|\mathsf{P}_{x^*,j}|} + \sum_{j=l+1}^{t} \frac{|\mathsf{C} \cap \mathsf{S}_{x^*,j}|}{|\mathsf{S}_{x^*,j}|} \qquad \text{(by the above observation).} \end{aligned} \tag{3.1}$$

Now note that $|\mathsf{P}_{x^*,j}| = \binom{l}{l-j}$ and $|\mathsf{S}_{x^*,j}| = \binom{d-l}{j-l}$. To maximize (3.1), the adversary would corrupt as many points as possible in the smallest $\mathsf{P}_{x^*,j}$'s and $\mathsf{S}_{x^*,j}$'s. Using standard calculations of the binomial coefficients, it can be verified that the adversary can corrupt at most $20 \log K$ such sets before its corruption budget K runs out. The upper bound of $20 \log K$ on (3.1) thus follows. ∎

We now prove the theorem.

Proof. [of Theorem 3.2] By Lemma 3.1, the chain X_{i^*} that contains the target has at most $40 \log K$ corrupted elements with probability at least $1/2$. Conditioned on this, by Theorem 2.2 we successfully traps the target in an interval on X_{i^*} of length $100 \log K$ with probability at least 0.9. As a result, we find the target in each iteration with probability 0.45, and thus find the target with probability at least $1-\delta$ by $O(\log(1/\delta))$ repetition. Since the number of the chains is $m = O(\frac{2^d}{\sqrt{d}})$, and each noisy binary search takes $O(\log d + \log K)$ queries by Theorem 2.2, in each iteration we make at most $O(\frac{2^d}{\sqrt{d}}(\log K + \log d))$ queries in total. The statement in the theorem then follows. ∎

3.2 An $\Omega(\min\left\{\frac{2^d}{\log d\sqrt{d}} \log K,\ 2^d\right\})$ Query Lower Bound In this subsection, we provide an $\Omega(\min\left\{\frac{2^d}{\log d\sqrt{d}} \log K,\ 2^d\right\})$ lower bound for the number of queries needed by a randomized algorithm to find a target t in a hypercube of dimension d in the presence of K corrupted elements. We prove it using Yao's minimax theorem. Namely, we define a distribution over the possible inputs and show any deterministic algorithm has a large error over the distribution.

We start by defining a distribution $\mathcal{D}(k)$ over functions $\tilde{f}$. A function $\tilde{f}$ and a target is drawn from $\mathcal{D}(k)$ as follows:

1. Let $x_{\max}$ be the point with all ones, and $x_{\min}$ be the point with all zeroes. Let f be the function that has $f(x) = 0$ if $x \in \mathsf{S}_{x_{\min},\leq (d/2)-k-1}$, $f(x) = 3$ if $x \in \mathsf{P}_{x_{\max},\geq (d/2)+1}$ and $f(x) = 2$ for any other point.
2. Let $y \in \mathsf{S}_{x_{\min},\leq d/2} \cap \mathsf{P}_{x_{\max},\geq (d/2)-k}$ be a random point. $\tilde{f}$ has the same value as f everywhere except y, where $\tilde{f}(y) = 1$.
3. Define all the points that are not compatible with y as corrupted, and return $\tilde{f}$ as the function and 1 as the target.

Note that 1 is always the target, and all the corrupted cells are preceding neighbors of the target in $\mathsf{S}_{x_{\min},\leq d/2} \cap \mathsf{P}_{x_{\max},\geq (d/2)-k}$. From now on, we refer to $\mathsf{S}_{x_{\min},\leq (d/2)-k-1}$ as the first layers, $\mathsf{S}_{x_{\min},\leq d/2} \cap \mathsf{P}_{x_{\max},\geq (d/2)-k}$ as the second layers and $\mathsf{P}_{x_{\max},(d/2)+1}$ as the third layers. Note that second layers contain the middle layer and the k layers below them.

LEMMA 3.2. *Any function $\tilde{f} \in \mathcal{D}(k)$ has at most $\sum_{i=1}^{k} \binom{d/2}{i}$ corrupted cells.*

Proof. Let y be the point for which $\tilde{f}(y) = 1$. Then all the corrupted cells are the preceding cells of y in the second layers. The number of preceding cells of y with hamming distance i is $\binom{||y||_1}{i}$. So the total number of corrupted elements is $\sum_{i=1}^{||y||_1-(d/2-k)} \binom{||y||_1}{i} \leq \sum_{i=1}^{k} \binom{d/2}{i}$ as $||y||_1 \leq d/2$. ∎

LEMMA 3.3. *Any deterministic algorithm with constant success probability over $\mathcal{D}(k)$ should make at least $\Omega(\sum_{i=d/2-k}^{d/2}\binom{d}{i})$ queries.*

Proof. The proof follows from the fact that any deterministic algorithm gets 2 if it doesn't hit the target and gets 1 if it hits the target. So with each miss, the algorithm eliminates one possible cell for the target and gains no other information about other cells. Since the target can be anywhere in the second layers, any algorithm algorithm with constant success probability needs to make $\Omega(\sum_{i=d/2-k}^{d/2}\binom{d}{i})$ queries. ∎

THEOREM 3.3. *Any randomized algorithm that can find a target t in a corrupted monotone Boolean hypercube with a constant success probability needs to make $\min\left\{\frac{2^d}{\log d\sqrt{d}}\log K,\ 2^d\right\}$ queries, where d is the dimension of the hypercube and K is the number of corrupted cells.*

Proof. Let k be the largest integer such that $K \geq \sum_{i=1}^{k}\binom{d/2}{i}$. By Lemma 3.2, we know each draw from $\mathcal{D}(k)$ has at most K corrupted cells. Suppose $k \geq \sqrt{d}$. By Lemma 3.3 and Yao's minimax theorem, the query complexity of any algorithm with constant success probability is at least $\Omega(\sum_{i=d/2-k}^{d/2}\binom{d}{i})$. But each layer within distance of $\sqrt{d}$ from the middle layer has $\Omega(\frac{2^d}{\sqrt{d}})$ cells and there are at least $\sqrt{d}$ layers. So $\Omega(\sum_{i=d/2-k}^{d/2}\binom{d}{i}) = \Omega(2^d)$ and we are done in this case.

Now suppose that $k < \sqrt{d}$. Again, by standard binomial calculations we have $k = \Omega(\log K/\log d)$, and each of the k layers below the middle layer has $\Omega(\frac{2^d}{\sqrt{d}})$ cells. So by Lemma 3.3, any deterministic algorithm with constant success probability needs to make $\Omega(\frac{2^d}{\sqrt{d}}\cdot k)$ queries. Then by Yao's minimax theorem, any randomized algorithm with constant success probability needs to make $\Omega(\frac{2^d}{\sqrt{d}}k) = \Omega(\frac{2^d}{\log d\sqrt{d}}\log K)$ queries and this completes the proof. ∎

4 Search in a Corrupted Monotone Higher-dimensional Grid

We now consider searching a target in a d-dimensional grid indexed by $\{0,1,\ldots,n\}^d$, where when any $d-1$ coordinates are fixed, the numbers are monotone with respect to the remaining coordinate. We are again concerned with the case where at most K cells are corrupted by an adversary.

For two points $x, y \in \{0,1,\ldots,n\}^d$, we say x *precedes* y (denoted by $x \preccurlyeq y$) if $x_i \leq y_i, \forall i \in [d]$, and x *succeeds* y (denoted by $x \succcurlyeq y$) if $x_i \geq y_i, \forall i \in [d]$. We also say x *strictly* precedes (succeeds) y if $x \preccurlyeq y$ ($x \succcurlyeq y$) but $x \neq y$, denoted by $x \prec y$ ($x \succ y$). We say x and y are *incomparable* if x neither precedes nor succeeds y.

4.1 A $\min\left\{O(n^{d-1}K^{1/d}d + \frac{n^{d-1}}{K^{\frac{d-1}{d}}}\tilde{O}(\log(n))), n^d\right\}$ Query Algorithm In this subsection, we provide a $\min\left\{O(n^{d-1}K^{1/d}d + \frac{n^{d-1}}{K^{\frac{d-1}{d}}}\log(n)\log(\log(n))), n^d\right\}$ algorithm for searching a target t in a d-dimensional grid of the form $\{0,1,\ldots,n\}^d$ in the presence of K corrupted elements. We start by some definitions and properties of the grid, and then provide the algorithm.

A set of cells $C \subseteq \{0,\ldots,n\}^d$ is a *cuboid* if it is a Cartesian product of d intervals, i.e. $C = \prod_{i=1}^{d}[l_i, r_i]$ where $l_i \leq r_i, \forall i$. The *volume* of a such a cuboid is the number of cells it contains, given by $\texttt{vol}(C) = \prod_{i=1}^{d}(r_i - l_i + 1)$. A cuboid is a *cube* if all of its sides are of equal length.

A $(3K)^{1/d}$*-tiling* of a d-dimensional grid with side length n is the partitioning of cells into cubes of side length $(3K)^{1/d}$. Namely, the cubes are of the form

$$\prod_{i\in[d]}[(k_i-1)(3K)^{1/d}+1, k_i(3K)^{1/d}] \qquad \text{where } k_i \in [n/(3K)^{1/d}], \forall i.$$

The volume of each cube in the tiling above is $3K$. We will identify the cubes in the tiling by $C_{k_1,\ldots,k_d}$. A cube $C_{a_1,\ldots,a_d}$ succeeds cube $C_{b_1,\ldots,b_d}$ if every cell in $C_{a_1,\ldots,a_d}$ succeeds every cell in $C_{b_1,\ldots,b_d}$. Also, a cube $C_{a_1,\ldots,a_d}$ precedes cube $C_{b_1,\ldots,b_d}$ if every cell in $C_{a_1,\ldots,a_d}$ precedes every cell in $C_{b_1,\ldots,b_d}$.

DEFINITION 4.1. *A cubic diagonal of a grid $\mathcal{G}$ is a set of cubes of the form $\bigcup_{0\leq i<l} C_{k_1+i,\ldots,k_d+i}$ for some $k_1, k_2, \ldots, k_d$ and l. We define l as the size of the cubic diagonal.*

Note that there is a total order on the cubes of a cubic diagonal. So when the cube is uncorrupted, for a cubic diagonal with size l, one can trap the target in a single cube with $2\lceil \log l \rceil$ comparisons, by comparing the target with minimum and maximum cells of a cube. We refer to the most preceding cube of a cubic grid as smallest cube, i.e., the cube for which every cell in it precedes every cell in other cubes of the diagonal. Similarly, we refer to the most succeeding as largest cube.

One can perform the binary search in the corrupted case as well. For a cubic diagonal in the presence of K corruptions, we can use the majority argument as in Theorem 2.2. In order to decide if t succeeds or precedes cells in a cube, we sample a number of cells from the cube and compare them with t. If t is bigger than the majority, we search for t in the succeeding cubes and the current cube, and if t is smaller than the majority, then we search for t in preceding cubes and the current cube. We choose the sample size large enough to successfully trap t with high probability. The probability that a sampled cell in a cube of volume $3K$ is uncorrupted is at least $2/3$, so for a cubic diagonal with length l, by sampling $O(\log((\log l)/\delta))$ from each cube, we can trap the target in at most two cubes with probability at least $1-\delta$. Now we show that we can cover any grid with cubic diagonals.

LEMMA 4.1. *There is a covering of a grid $\mathcal{G}$ with $O(d(\frac{n}{K^{1/d}})^{d-1})$ cubic diagonals.*

Proof. We provide a $(3K)^{1/d}$-tiling of the grid $\mathcal{G}$. Consider all the cubic diagonals for which the smallest cube has a coordinate 1 and the largest cube has a coordinate $n/(3K)^{1/d}$. In other words, the minimum cube moves up on the same coordinate until it hits the end of the grid. Note that a cubic diagonal has one cube if the cube has 1 in one coordinate, and $n/(3K)^{1/d}$ in another coordinate. Let S be the union of all such cubic diagonals. Each cube of the grid is covered by S, so every cell of the grid is covered. There are at most $d(\frac{n}{(3K)^{1/d}})^{d-1}$ candidates for the smallest cube of a cubic diagonal, So $|S| = O(d(\frac{n}{(3K)^{1/d}})^{d-1}) = O(d(\frac{n}{(K)^{1/d}})^{d-1})$. ∎

We are now ready to provide an algorithm for searching a target t in a grid $\mathcal{G}$. The algorithm works as follows:

1. Consider a covering of $\mathcal{G}$ with cubic diagonals as in Lemma 4.1.
2. Perform a noisy binary search on each of the $O(d(\frac{n}{(K)^{1/d}})^{d-1})$ cubic diagonals by sampling from the cubes, and store the candidate cubes. Perform the binary search by querying $O(\log(\log(n)/\delta))$ random independent cells for each cube.
3. Do an exhaustive search on every candidate cube. If the target is found, return the index, otherwise return null.

LEMMA 4.2. *The above algorithm makes at most $O(d(\frac{n}{K^{1/d}})^{d-1}(\log n \log(\log(n)/\delta) + K))$ queries.*

Proof. Performing a binary search on a cubic diagonal takes at most $O(\log n)\log(\log(n)/\delta)$ queries as the length of a cubic diagonal is at most $n/(3K)^{1/d} = O(n)$ and we take $\log(\log(n)/\delta)$ samples when we compare t to a cube. So the algorithm makes $O(d(\frac{n}{K^{1/d}})^{d-1}\log(n)\log(\log(n)/\delta))$ queries at Line 2. The exhaustive searches on cubes take at most $O(d(\frac{n}{K^{1/d}})^{d-1}K)$ queries at Line 3. So the total query complexity of the algorithm is $O(d(\frac{n}{K^{1/d}})^{d-1}(\log n \log(\log(n)/\delta) + K))$. ∎

THEOREM 4.1. *There is a randomized algorithm that makes $\min\{O(d(\frac{n}{K^{1/d}})^{d-1}(\log(n)\log(\log(n)/\delta) + K)), n^d\}$ queries and finds a target t with probability $1-\delta$ in a d-dimensional grid of the form $\{0, 1, \ldots n^d - 1\}$ when at most K elements are corrupted.*

Proof. If $K^{1/d} > n/d$, then we check all the cells of $\mathcal{G}$. Otherwise, by using the algorithm in Lemma 4.2, we make at most $O(d(\frac{n}{K^{1/d}})^{d-1}(\log(n)\log(\log(n)/\delta) + K))$ queries.

Now we prove that the above algorithm finds the target with probability $1-\delta$. Consider the cubic diagonal that contains t. By Chernoff bound and Union Bound, the probability that the algorithm makes no error, i.e., always goes to the right direction in the binary seacrh, is at least $1-\delta$. In this case, the cube that contains t would be among the candidate cubes. So the algorithm would find it at Line 3. Thus the probability of success is at least $1-\delta$. ∎

4.2 A $\min\left\{\Omega(n^{d-1}K^{1/d}\sqrt{d}), \Omega(n^d)\right\}$ Query Lower Bound In this subsection, we prove a lower bound for the problem. We prove our lower bound in the setting that $n > 10e$ and $K > (20e)^d$. Similar to the hypercube case, we first partition the grid into layers. Layer i consists all the cells whose sum of coordinates is i. So there are nd layers. If $i > j$ we say layer i is above layer j and layer j is below layer i. Let L_i be the number of cells in the layer i. We also refer to L_i as the size of the layer i. We know that L_i's are increasing when $i \leq nd/2$ and then starts to decrease from $i = \lceil nd/2 \rceil$ onwards [Sta06]. We refer to layer $\lceil nd/2 \rceil$ as the middle layer, and l middle layers as layers from $\lceil nd/2 \rceil - \lceil l/2 \rceil - 1$ to $\lceil nd/2 \rceil + \lfloor l/2 \rfloor$. We first state a folklore lemma.

LEMMA 4.3. ([STA06]) *The middle layer has $L_{\lceil nd/2 \rceil} = \Theta(\frac{n^{d-1}}{\sqrt{d}})$ cells. Besides, the average size of $n\sqrt{d}$ middle layers is $\Theta(\frac{n^{d-1}}{\sqrt{d}})$.*

Now we can define a hard distribution over the inputs for the problem. A grid $\mathcal{G}$ and a target t is drawn from $\mathcal{D}_G$ as follows:

1. Let $\mathcal{G}$ be any monotone grid such that the elements in the $d(\frac{K^{1/d}}{10e} - 1)$ middle layers are positive, elements in layers below the $d(\frac{K^{1/d}}{10e} - 1)$ middle layers are $-\infty$, and elements in layers above the $d(\frac{K^{1/d}}{10e} - 1)$ middle layers are ∞.
2. Replace a random element of the $d(\frac{K^{1/d}}{10e} - 1)$ middle layers with 0.
3. Return $\mathcal{G}$ as the grid and $t = 0$ as the target.

Note that as we assumed $\frac{K^{1/d}}{10e} > 2$, middle layers in Line 1 contains at least one layer.

LEMMA 4.4. *Any grid $\mathcal{G}$ drawn from $\mathcal{D}_G$ has at most K corrupted cells.*

Proof. The number of corrupted cells is the number of cells that do not agree with $t = 0$. Such cells are the preceding cells of t in the $d(\frac{K^{1/d}}{10e} - 1)$ middle layers. The preceding cells form a subgrid of the main grid. The size of the subgrid is less than the grid with minimum point 0 and $d(\frac{K^{1/d}}{10e} - 1)$ layers. One layer in the latter grid has at most $\binom{d(K^{1/d}/e-1)+(d-1)}{d-1}$ cells, so the grid has at most $d(\frac{K^{1/d}}{10e} - 1)\binom{d(K^{1/d}/e-1)+(d-1)}{d-1} \leq dK^{1/d}(\frac{e \cdot dK^{1/d}/(10e)}{d-1})^{d-1} \leq K$ cells, where we used the fact that $\binom{n}{k} \leq (ne/k)^k$. So the number of the corrupted cells is at most K and the lemma follows. ∎

LEMMA 4.5. *Any deterministic algorithm with constant success probability over $\mathcal{D}_G$ must make at least $\min\left\{\Omega(n^{d-1}K^{1/d}\sqrt{d}), \Omega(n^d)\right\}$ queries.*

Proof. Suppose a deterministic algorithm $\mathcal{A}_D$ makes q queries and find the target with constant probability over $\mathcal{D}_G$. By the construction of $\mathcal{D}_G$, if a cell is not the target, its value is already known. Also, the value of cells not located in the $d(\frac{K^{1/d}}{10e} - 1)$ middle layers are always the same for each grid in $\mathcal{D}_G$. So the algorithm $\mathcal{A}_D$ follows the same path unless it hits the target. Thus, the probability of success is at most $\frac{q}{s} + (1 - \frac{q}{s}) \cdot \frac{1}{s-q}$ where s is the number of cells in the $d(\frac{K^{1/d}}{10e} - 1)$ middle layers. Now we consider two cases. First, suppose $K^{1/d} > n\sqrt{d}$. Then $d(K^{1/d}/e - 1) = \Omega(n\sqrt{d})$. Then by Lemma 4.3, a constant fraction of all cells are located in $d(\frac{K^{1/d}}{10e} - 1)$ middle layers. So, $s = \Omega(n^d)$, and we have $q = \Omega(s) = \Omega(n^d)$.

For the other case, suppose $K^{1/d} \leq n\sqrt{d}$. The by Lemma 4.3, we have $s > \Omega(d(\frac{K^{1/d}}{10e} - 1) \cdot \frac{n^{d-1}}{\sqrt{d}}) = \Omega(n^{d-1}K^{1/d}\sqrt{d})$ where we used the condition $\frac{K^{1/d}}{10e} > 2$. So $\mathcal{A}_D$ must make $\Omega(s) = \Omega(n^{d-1}K^{1/d}\sqrt{d})$ queries. This concludes that any deterministic algorithm $\mathcal{A}_D$ with constant success probability over $\mathcal{D}_G$, must make $\min\left\{\Omega(n^{d-1}K^{1/d}\sqrt{d}), \Omega(n^d)\right\}$ queries. ∎

THEOREM 4.2. *Any randomized algorithm that can find a target t in a monotone grid with constant success probability must make $\min\left\{\Omega(n^{d-1}K^{1/d}\sqrt{d}), \Omega(n^d)\right\}$ queries where d is the dimension of the grid, n is the maximum of a coordinate in the grid and K is the number of corrupted elements in the grid.*

Proof. The proof immediately follows from Yao's minimax theorem in conjunction with Lemma 4.4 and Lemma 4.5. ∎

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. The second author would like to thank Nalin Khanna for a discussion that helped simplify presentation of the algorithmic results in Section 2. This work was supported in part by NSF awards CCF-1763514, CCF-1934876, CCF-2008305, CCF-1910534, CCF-1926872, and CCF-2045128.

References

[AFHN15] Miklós Ajtai, Vitaly Feldman, Avinatan Hassidim, and Jelani Nelson. Sorting and selection with imprecise comparisons. *ACM Transactions on Algorithms (TALG)*, 12(2):1–19, 2015.

[BM08] M. Braverman and E. Mossel. Noisy sorting without resampling. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 268–276. Society for Industrial and Applied Mathematics, 2008.

[BMP19] M. Braverman, J. Mao, and Y. Peres. Sorted top-k in rounds. In *Conference on Learning Theory*, pages 342–382, 2019.

[BOH08] Michael Ben-Or and Avinatan Hassidim. The bayesian learner is optimal for noisy binary search (and pretty good for quantum as well). In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 221–230. IEEE, 2008.

[DFGM17] Yuval Dagan, Yuval Filmus, Ariel Gabizon, and Shay Moran. Twenty (simple) questions. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 9–21, 2017.

[DFKM21] Yuval Dagan, Yuval Filmus, Daniel Kane, and Shay Moran. The entropy of lies: Playing twenty questions with a liar. In *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, pages 1:1–1:16, 2021.

[DGW92] Aditi Dhagat, Péter Gács, and Peter Winkler. On playing "twenty questions" with a liar. In *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 27-29 January 1992, Orlando, Florida, USA*, pages 16–22, 1992.

[DKLN21] Anindya De, Sanjeev Khanna, Huan Li, and Hesam Nikpey. Approximate optimization of convex functions with outlier noise. 2021. To appear in NeurIPS 2021.

[FRPU94] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[Han66] Georges Hansel. Sur le nombre des fonctions booléennes monotones de n variables. *COMPTES RENDUS HEBDOMADAIRES DES SEANCES DE L ACADEMIE DES SCIENCES SERIE A*, 262(20):1088, 1966.

[HS16] Thibaut Horel and Yaron Singer. Maximization of approximately submodular functions. In *Advances in Neural Information Processing Systems*, pages 3045–3053, 2016.

[HS17] Avinatan Hassidim and Yaron Singer. Submodular optimization under noise. In *Conference on Learning Theory*, pages 1069–1122. PMLR, 2017.

[KK07] R. Karp and R. Kleinberg. Noisy binary search and its applications. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 881–890, 2007.

[KTDV96] Boris Kovalerchuk, Evangelos Triantaphyllou, Aniruddha S. Deshpande, and Evgenii Vityaev. Interactive learning of monotone boolean functions. *Inf. Sci.*, 94(1-4):87–118, 1996.

[Now09] R. Nowak. Noisy generalized binary search. In *Advances in Neural Information Processing Systems*, pages 1366–1374, 2009.

[Pel02] Andrzej Pelc. Searching games with errors - fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1-2):71–109, 2002.

[RMK+80] Ronald L. Rivest, Albert R. Meyer, Daniel J. Kleitman, Karl Winklmann, and Joel Spencer. Coping with errors in binary search procedures. *J. Comput. Syst. Sci.*, 20(3):396–404, 1980.

[RRSW11] Dana Ron, Ronitt Rubinfeld, Muli Safra, and Omri Weinstein. Approximating the influence of monotone boolean functions in $o(\sqrt{n})$ query complexity. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 14th International Workshop, APPROX 2011, and 15th International Workshop, RANDOM 2011, Princeton, NJ, USA, August 17-19, 2011. Proceedings*, pages 664–675, 2011.

[RV17] Aviad Rubinstein and Shai Vardi. Sorting from noisier samples. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 960–972. SIAM, 2017.

[Sta06] Richard Stanley. Log-concave and unimodal sequences in algebra, combinatorics, and geometry. *Annals of the New York Academy of Sciences*, 576:500 – 535, 12 2006.

[SV15] Yaron Singer and Jan Vondrák. Information-theoretic lower bounds for convex optimization with erroneous oracles. In *Advances in Neural Information Processing Systems*, pages 3204–3212, 2015.

A A $(2+\epsilon)$-close Algorithm for Trapping a Target in a Corrupted Monotone Array

In this section, we provide an algorithm for trapping a target in an interval of length $(2+\epsilon)K$.

Proof. [of Theorem 2.2] By Line 1 of the algorithm in Theorem 2.1, we can narrow down the interval to $10K$. With the same idea as Theorem 2.1, we take several samples from the middle interval of length $(2+\epsilon/2)K$, and delete the part that does not contain t. The algorithm works as follows:

1. Run Line 1 of Theorem 2.1 so $|A| \leq 10K$.
2. Let $\epsilon' = \epsilon/2$ and $l' = (2+\epsilon')K$.
3. Let m be the mid-point of A.
4. Take $s = \frac{1000}{\epsilon^2}\log\log(\frac{1}{\epsilon})$ independent random samples from $A[m - \lceil l'/2 \rceil, m + \lfloor l'/2 \rfloor]$ and compare it with t, if any of them was equal to t return the index.
5. For $1 \leq i \leq s$, X_i is 1 if t is bigger than the i'th sample and 0 otherwise. Let $X = \sum_i X_i$. If $X \geq s/2$, discard all the elements with index less than $m - \lceil l'/2 \rceil$, otherwise discard all elements with index greater than $m + \lfloor l'/2 \rfloor$.
6. If the length of the remaining array is greater than $(2+\epsilon)K$, go to line 4, otherwise return the interval.

The number of recursions in Line 6 is at most $O(\log 1/\epsilon)$, as in each step, the number of all elements but l' middle elements becomes half, and we only need it to become less than $K\epsilon/2 = K\epsilon'$. For each step, the algorithms makes $O(\frac{1}{\epsilon^2}\log\log(\frac{1}{\epsilon}))$ queries. So the query complexity for this part is $O(1/\epsilon^2 \log(1/\epsilon)\log\log(1/\epsilon))$.

The probability that a sampled element in step 4 is uncorrupted is at least $\frac{(1+\epsilon')K}{(2+\epsilon')K} > 1/2 + \epsilon'/5$ for $\epsilon' < 1/2$. Let $Y_{i,j}$ be a random variable that is 1 if at the i'th recursion of the algorithm, j'th sampled element is uncorrupted and $Y_i = \sum_j Y_{i,j}$. Then $\mathbf{E}[Y_{i,j}] \geq 1/2 + \epsilon'/5$ for each i, j. Note that $\mathbf{E}[Y_i] = (1/2 + \epsilon'/5)s$, so by Chernoff bound, $\mathbf{Pr}\left[Y_i \leq \frac{s}{2}\right] < \frac{0.1}{\log(1/\epsilon)}$. So, by Union bound, with constant probability in all steps of the algorithm, $|Y_i| > s/2$, which means the number of uncorrupted elements is more than half for each step and the algorithm goes to the right direction of the array. So the final interval contains t with constant probability. The probability of success in the first part of the Theorem 2.1 is 2/3, so with constant probability the algorithm finds t. By repeating the algorithm for $\log 1/\delta$ times, we get the desired result. ∎

B A 2-close Lower Bound for Trapping a Target in a Corrupted Monotone Array

In this section, we provide a proof of 2-close lower bound stated in Section 2. Suppose there is a monotone array of $2n$ elements where half of them are corrupted and there is a randomized algorithm $\mathcal{A}$ that makes cn queries and returns an interval of size $(2-\epsilon)n$. We'll show that the algorithm has an error of at least $(\epsilon - c)/2$, means the returned interval does not contain t with probability at least $(\epsilon - c)/2$. To do so, we provide a distribution $\mathcal{D}$ over the inputs (array A and target t) and show for any deterministic algorithm $\mathcal{A}_D$, $\mathbf{Pr}_{(A,t)\sim\mathcal{D}}[t \notin \mathcal{A}_D(A,t)] \geq (\epsilon - c)/2$. Then by Yao's minimax theorem we have the result for the randomized algorithms. From now on we assume $\epsilon > c$, the other case is easy, the algorithm only returns the elements that has not queried.

An element (A, t) of $\mathcal{D}$ is drawn as follows:

1. Let $t = n$.
2. Replace one element of $[0, 1, \ldots, n-1, n+1, n+2, \ldots, 2n-1, 2n]$ with t uniformly at random. Return the resulted array as A, if t falls in the first half of the array, t and the second half of the array are uncorrupted and if t falls in the second half of the array, t and the first half of the array are uncorrupted.

Note that by step 2, every drawn array from $\mathcal{D}$ has less than n corrupted elements.

THEOREM B.1. *Let $\mathcal{A}_D$ be a deterministic algorithm that makes at most cn queries on any array A consisting $2n$ numbers and any target t, and returns an interval of $(2-\epsilon)n$. Then $\mathbf{Pr}_{(A,t)\sim\mathcal{D}}[t \notin \mathcal{A}_D(A,t)] \geq (\epsilon - c)/2$.*

Proof. By definition of $\mathcal{D}$, if the algorithm queries an arbitrary index i, it either finds t or always gets the same value. So any deterministic algorithm proceeds the same path for any instances in $\mathcal{D}$, unless it hits the target. In other words, if the oracle only returns that an element is the target or not, it would be the same as the original oracle. This is true as by knowing that t is not in a position, we already know the value of the queried position. The probability that the algorithm doesn't hit t in cn queries is $\frac{2-c}{2}$. Then it returns the same interval of length $(2-\epsilon)n$ for such instances. The probability that t is not placed in the interval is $\frac{\epsilon-c}{2-c}$. So the probability of error is at least $\frac{(2-c)}{2} \cdot \frac{\epsilon-c}{2-c} = (\epsilon - c)/2$. ∎

The above theorem in conjunction with Yao's minimax theorem proves Theorem 2.3.