The Entropy of Lies: Playing Twenty Questions with a Liar

Yuval Dagan
Massachusetts Institute of Technology, Cambridge, MA, USA
dagan@mit.edu

Yuval Filmus
Technion – Israel Institute of Technology, Haifa, Israel
yuvalfi@cs.technion.ac.il

Daniel Kane
University of California San Diego, La, Jolla, CA, USA
dakane@ucsd.edu

Shay Moran
Technion – Israel Institute of Technology, Haifa, Israel
smoran@technion.ac.il

Abstract

"Twenty questions" is a guessing game played by two players: Bob thinks of an integer between 1 and n, and Alice's goal is to recover it using a minimal number of Yes/No questions. Shannon's entropy has a natural interpretation in this context. It characterizes the average number of questions used by an optimal strategy in the distributional variant of the game: let μ be a distribution over $[n]$, then the average number of questions used by an optimal strategy that recovers $x \sim \mu$ is between $H(\mu)$ and $H(\mu) + 1$.

We consider an extension of this game where at most k questions can be answered falsely. We extend the classical result by showing that an optimal strategy uses roughly $H(\mu) + kH_2(\mu)$ questions, where $H_2(\mu) = \sum_x \mu(x) \log\log \frac{1}{\mu(x)}$. This also generalizes a result by Rivest et al. (1980) for the uniform distribution.

Moreover, we design near optimal strategies that only use comparison queries of the form "$x \le c$?" for $c \in [n]$. The usage of comparison queries lends itself naturally to the context of sorting, where we derive sorting algorithms in the presence of adversarial noise.

2012 ACM Subject Classification Theory of computation

Keywords and phrases entropy, twenty questions, algorithms, sorting

Digital Object Identifier 10.4230/LIPIcs.ITCS.2021.1

Related Version A full version of the paper is available at `https://arxiv.org/abs/1811.02177`.

Funding *Yuval Filmus*: Taub Fellow – supported by the Taub Foundations. The research was funded by ISF grant 1337/16.

1 Introduction

The "twenty questions" game is a cooperative game between two players: Bob thinks of an integer between 1 and n, and Alice's goal is to recover it using the minimal number of Yes/No questions. An optimal strategy for Alice is to perform binary search, using $\log n$ queries in the worst case.

The game becomes more interesting when Bob chooses his number according to a distribution μ known to both players, and Alice attempts to minimize the *expected* number of questions. In this case, the optimal strategy is to use a Huffman code for μ, at an expected cost of roughly $H(\mu)$.

What happens when Bob is allowed to lie (either out of spite, or due to difficulties in the communication channel)? Rényi [20] and Ulam [25] suggested a variant of the (non-distributional) "twenty questions" game, in which Bob is allowed to lie k times. Rivest et al. [21], using ideas of Berlekamp [4], showed that the optimal number of questions in this setting is roughly $\log n + k \log\log n$. There are many other ways of allowing Bob to lie, some of which are described by Spencer and Winkler [24] in their charming work, and many others by Pelc [19] in his comprehensive survey on the topic.

Distributional "twenty questions" with lies

This work addresses the distributional "twenty questions" game in the presence of lies. In this setting, Bob draws an element x according to a distribution μ, and Alice's goal is to recover the element using as few Yes/No questions as possible on average. The twist is that Bob, who knows Alice's strategy, is allowed to lie up to k times. Both Alice and Bob are allowed to use randomized strategies, and the average is measured according to both μ and the randomness of both parties.

Our main result shows that the expected number of questions in this case is

$$H(\mu) + kH_2(\mu), \qquad \text{where } H_2(\mu) = \sum_x \mu(x) \log\log \frac{1}{\mu(x)},$$

up to an additive factor of $O(k \log k + kH_3(\mu))$, where $H_3(\mu) = \sum_x \mu(x) \log\log\log(1/\mu(x)))$ (here $\mu(x)$ is the probability of x under μ.) See Section 3 for a complete statement of this result.

When μ is the uniform distribution, the expected number of queries that our algorithm makes is roughly $\log n + k \log\log n$, matching the performance of the algorithm of Rivest et al. However, the approach by Rivest et al. is tailored to their setting, and the distributional setting requires new ideas.

As in the work of Rivest et al., our algorithms use only *comparison queries*, which are queries of the form "$x \prec c$?" (for some fixed value c). Moreover, our algoritms are efficient, requiring $O(n)$ preprocessing time and $O(\log n)$ time per question. Our lower bounds, in contrast, apply to *arbitrary* Yes/No queries.

Noisy sorting

One can apply binary search algorithms to implement insertion sort. While sorting an array typically requires $\Theta(n \log n)$ *sorting queries* of the form "$x_i \prec x_j$?", there are situations where one has some prior knowledge about the correct ordering. This may happen, for example, when maintaining a sorted array: one has to perform consecutive sorts, where each sort is not expected to considerably change the locations of the elements. Assuming a distribution Π over the $n!$ possible permutations, Moran and Yehudayoff [17] showed that sorting a Π-distributed array requires $H(\Pi) + O(n)$ sorting queries on average. We extend this result to the case in which the answerer is allowed to lie k times, giving an algorithm which uses the following expected number of queries:[1]

$$H(\Pi) + O(nk).$$

This result is tight, and matches the optimal algorithms for the uniform distribution due to Bagchi [3] and Long [16], which use $n \log n + O(nk)$ queries.

[1] Strictly speaking, this bound holds only under the mild condition that k is at most exponential in n.

Table 1 Query complexities of searching and sorting in different settings, ignoring lower-order terms. All terms are exact upper and lower bounds except for those inside the $O(\cdot)$ and $\Theta(\cdot)$ notations.

Setting	Searching	Sorting
No lies; deterministic	$\log n$ [classical]	$n \log n$ [classical]
No lies; distributional	$H(\mu)$ [classical]	$H(\Pi) + O(n)$ [17]
k lies; deterministic	$\log n + k \log\log n$ [21]	$n \log n + \Theta(nk)$ [3, 16, 15]
k lies; distributional	$H(\mu) + kH_2(\mu)$ [this paper]	$H(\Pi) + \Theta(nk)$ [this paper]

Table 1 summarizes the query complexities of resilient and non-resilient searching and sorting algorithms, in both the deterministic and the distributional settings. To the best of our knowledge, we present the first resilient algorithms in the distributional setting.

On randomness

All algorithms presented in the paper are randomized. Since they only employ public randomness which is known for both players, there exists a fixing of the randomness which yields a deterministic algorithm with the same (or possibly smaller) expected number of queries. However, this comes at the cost of possibly increasing the running time of the algorithm (since we need to find a good fixing of the randomness); it would be interesting to derive an explicit efficient deterministic algorithm with a similar running time.

1.1 Main ideas

Upper bound

Before presenting the ideas behind our algorithms, we explore several other ideas which give suboptimal results. The first approach that comes to mind is simulating the optimal non-resilient strategy, asking each question $2k + 1$ times and taking the majority vote, which results in an algorithm using $\Theta(kH(\mu))$ queries on average.

A better approach is using *tree codes*, suggested by Schulman [22] as an approach for making interactive communication resilient to errors [11, 22, 14]. Tree codes are designed for a different error model, in which we are bounding the *fraction* of lies rather than their absolute number; for an ε-fraction of lies, the best known constructions suffer a multiplicative overhead of $1 + O(\sqrt{\varepsilon})$ [13]. In contrast, we are aiming at an *additive* overhead of $kH_2(\mu)$.

Using a packing bound, one can prove that there exists a (non-interactive) code of expected length roughly $H(\mu) + 2kH_2(\mu)$, coming much closer to the bound that we are able to get (but off by a factor of 2 from our target $H(\mu) + kH_2(\mu)$). The idea, which is similar to the proof of the Gilbert–Varshamov bound, is to construct a prefix code $w_1, \dots, w_n$ in which the prefixes of w_i, w_j of length $\min(|w_i|, |w_j|)$ are at distance at least $2k + 1$ (whence the factor $2k$ in the resulting bound); this can be done greedily. Apart from the inferior bound, two other disadvantages of this approach is that it is not efficient and uses arbitrary queries.

In contrast to these prior techniques, which do not achieve the optimal complexity, might ask arbitrary questions, and could result in strategies which cannot be implemented efficiently, in this paper we design an efficient and nearly optimal strategy, relying on comparison queries only, and utilizing simple observations on the behavior of binary search trees under the presence of lies.

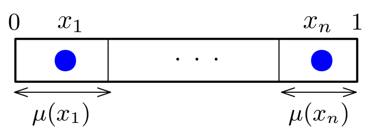

Figure 1 Representing items as centers of segments partitioning the interval $[0, 1]$.

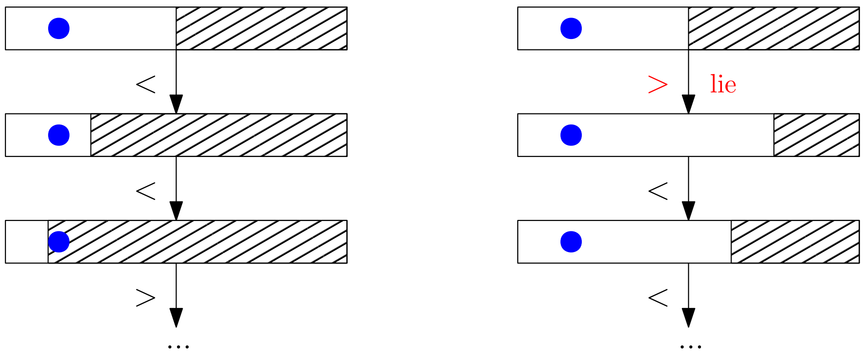

Figure 2 On the left, the operation of the algorithm without any lies. On the right, answerer lied on the first question. As a result, all future truthful answers are the same.

Following the footsteps of Rivest et al. [21], our upper bound is based on a binary search algorithm on the unit interval $[0, 1]$, first suggested in this context by Gilbert and Moore [12]: given $x \in [0, 1]$, the algorithm locates x by first asking "$x < 1/2$?"; depending on the answer, asking "$x < 1/4$?" or "$x < 3/4$?"; and so on. If $x \in [0, 1]$ is chosen uniformly at random then the answers behave like an infinite sequence of random and uniform coin tosses.

In order to apply this kind of binary search to the problem of identifying an unknown element (assuming truthful answers), we partition the unit interval $[0, 1]$ into segments of lengths $\mu(x_1), \ldots, \mu(x_n)$, and label the center of each segment with the corresponding item (see Figure 1). We then perform binary search until the current interval contains a single item. (In the proof, we use a slightly more sophisticated randomized placement of points which guarantees that the answers on *each* element behave like an infinite sequence of random and uniform coin tosses.)

The main observation is that if a question "$x < a$?" is answered with a lie, this will be strongly reflected in subsequent answers (see Figure 2). Indeed, suppose that $x < a$, but Bob claimed that $x > a$. All subsequent questions will be of the form "$x < b$?" for various $b > a$, the truthful answer to all of which is $x < b$. An observer taking notes of the proceedings will thus observe the following pattern of answers: $>$ (the lie) followed by many $<$'s (possibly interspersed with up to $k - 1$ many $>$'s, due to further lies). This is suspicious since it is highly unlikely to obtain many $<$ answers in a row (the probability of getting r such answers is just 2^{-r}).

This suggests the following approach: for each question we will maintain a "confidence interval" consisting of $r(d)$ further questions (where d is the index of the question). At the end of the interval, we will check whether the situation is suspicious (as described in the preceding paragraph), and if so, will ascertain by brute force the correct answer to the original question (by taking a majority of $2k + 1$ answers), and restart the algorithm from that point.

The best choice for $r(d)$ turns out to be roughly $\log d$. Each time Bob lies, our unrolling of the confidence interval results in a loss of $r(d)$ questions. Since an item x requires roughly $\log(1/\mu(x))$ questions to be discovered, the algorithm has an overhead of roughly $kr(\log(1/\mu(x))) \approx k \log\log(1/\mu(x))$ questions on element x, resulting in an expected overhead of roughly $kH_2(\mu)$.

When implementing the algorithm, apart from the initial $O(n)$ time needed to setup the partition of $[0, 1]$ into segments, the costliest step is to convert the intervals encountered in the binary search to comparison queries. This can be done in $O(\log n)$ time per query.

Lower bound

The proof of our lower bound uses information theory: one can lower bound the expected number of questions by the amount of information that the questioner gains. There are two such types of information: first, the hidden object reveals $H(\mu)$ information, as in the setting where no lies are allowed. Second, when the object is revealed, the positions of the lies are revealed as well. This reveals additional $H_2(\mu)$ (conditional) information, as we explain below.

Let d_x denote the number of questions asked for element x. Kraft's inequality shows that any good strategy of the questioner satisfies $d_x \gtrapprox \log(1/\mu(x))$. If the answerer chooses a randomized strategy in which the positions of the lies are chosen uniformly from the $\binom{d_x}{k}$ possibilities, these positions reveal $\log\binom{d_x}{k} \approx k \log d_x \gtrapprox k \log\log(1/\mu(x))$ information given x. Taking expectation over x, the positions of the lies reveal at least $kH_2(\mu)$ information beyond the identity of x.

1.2 Related work

Most of the literature on error-resilient search procedures has concentrated on the non-distributional setting, in which the goal is to give a worse case guarantee on the number of questions asked, under various error models. The most common error models are as follows:[2]

- Fixed number of errors. This is the error model we consider, and it is also the one suggested by Ulam [25]. This model was first studied by Berlekamp [4], who used an argument similar to the sphere-packing bound to give a lower bound on the number of questions. Rivest et al. [21] used this lower bound as a guiding principle in their almost matching upper bound using comparison queries.
- At most a fixed fraction p of the answers can be lies. This model is similar to the one considered in error-correcting codes. Pelc [18] and Spencer and Winkler [24] (independently) gave a non-adaptive strategy for revealing the hidden element when $p \leq 1/4$, and showed that the task is not possible (non-adaptively) when $p > 1/4$. Furthermore, when $p < 1/4$ there is an algorithm using $O(\log n)$ questions, and when $p = 1/4$ there is an algorithm using $O(n)$ questions, which are both optimal (up to constant factors). Spencer and Winkler also showed that if questions are allowed to be adaptive, then the hidden element can be revealed if and only if $p < 1/3$, again using $O(\log n)$ questions.
- At most a fixed fraction p of any *prefix* of the answers can be lies. Pelc [18] showed that the hidden element can be revealed if and only if $p < 1/2$, and gave an $O(\log n)$ strategy when $p < 1/4$. Aslam and Dhagat [2] and Spencer and Winkler gave an $O(\log n)$ strategy for all $p < 1/2$.

[2] This section is heavily based on Pelc's excellent and comprehensive survey [19].

- Every question is answered erroneously with probability p, an error model common in information theory. Rényi [20] showed that the number of questions required to discover the hidden element with constant success probability is $(1 + o(1)) \log n/(1 - h(p))$.

The distributional version of the "twenty questions" game (without lies) was first considered by Shannon [23] in his seminal paper introducing information theory, where its solution was attributed to Fano (who published it later as [9]). The Shannon–Fano code uses at most $H(\mu) + 1$ questions on average, but the questions can be arbitrary. The Shannon–Fano–Elias code (also due to Gilbert and Moore [12]), which uses only comparison queries, asks at most $H(\mu) + 2$ questions on average. Dagan et al. [7] give a strategy, using only comparison and equality queries, which asks at most $H(\mu) + 1$ questions on average.

Sorting

The non-distributional version of sorting has also been considered in some of the settings considered above:

- At most k errors: Lakshmanan et al. [15] gave a lower bound of $\Omega(n \log n + kn)$ on the number of questions, and an almost matching upper bound of $O(n \log n + kn + k^2)$ questions. An optimal algorithm, using $n \log n + O(kn)$ questions, was given independently by Bagchi [3] and Long [16].
- At most a p fraction of errors in every prefix: Aigner [1] showed that sorting is possible if and only if $p < 1/2$. Borgstrom and Kosaraju [5] had showed earlier that even *verifying* that an array is sorted requires $p < 1/2$.
- Every answer is correct with probability p: Feige et al. [10] showed in an influential paper that $\Theta(n \log(n/\epsilon))$ queries are needed, where ϵ is the probability of error.
- Braverman and Mossel [6] considered a different setting, in which an algorithm is given access to noisy answers to all possible $\binom{n}{2}$ comparisons, and the goal is to find the most likely permutation. They gave a polynomial time algorithm which succeeds with high probability.

The distributional version of sorting (without lies) was considered by Moran and Yehudayoff [17], who gave a strategy using at most $H(\mu) + 2n$ queries on average, based on the Gilbert–Moore algorithm.

Paper organization

After a few preliminaries in Section 2, we describe our results in full in Section 3. The proofs are presented in the full version of this paper [8].

2 Definitions

We use the notation $\binom{n}{\leq k} = \sum_{\ell=0}^{k} \binom{n}{\ell}$. Unless stated otherwise, all logarithms are base 2. We define $\overline{\log}(x) = \log(x + C)$ and $\overline{\ln}(x) = \ln(x + C)$ for a fixed sufficiently large constant $C > 0$ satisfying $\log \log \log C > 0$.

Information theory

Given a probability distribution μ with countable support, the entropy of μ is given by the formula

$$H(\mu) = \sum_{x \in \operatorname{supp} \mu} \mu(x) \log \frac{1}{\mu(x)}.$$

Twenty questions game

We start with an intuitive definition of the game, played by a questioner (Alice) and an answerer (Bob). Let U be a finite set of elements, and let μ be a distribution over U, known to both parties. The game proceeds as follows: first, an element $x \sim \mu$ is drawn and revealed to the answerer but not to the questioner. The element x is called the *hidden* element. The questioner asks binary queries of the form "$x \in Q$?" for subsets $Q \subseteq U$. The answerer is allowed to lie a fixed number of times, and the goal of the questioner is to recover the hidden element x, asking the minimal number of questions on expectation.

Decision trees

Let U be a finite set of elements. A *decision tree* T for U is a binary tree formalizing the question asking strategy in the twenty questions game. Each internal node of v of T is labeled by *a query* (or *question*) – a subset of U, denoted by $Q(v)$; and each leaf is labeled by the output of the decision tree, which is an element of U. The semantics of the tree are as follows: on input $x \in U$, traverse the tree by starting at the root, and whenever at an internal node v, go to the left child if $x \in Q(v)$ and to the right child if $x \notin Q(v)$.

Comparison tree

Given an ordered set of elements $x_1 \prec x_2 \prec \cdots \prec x_n$, *comparison questions* are questions of the form $Q = \{x_1, \ldots, x_{i-1}\}$, for some $i = 1, \ldots, n+1$. In other words, the questions are "$x \prec x_i$?" for some $i = 1, \ldots, n+1$. An answer to a comparison question is one of $\{\prec, \succeq\}$. A *comparison tree* is a decision tree all of whose nodes are labeled by comparison questions.

Adversaries

Let $k \geq 0$ be a bound on the number of lies. An intuitive way to formalize the possibility of lying is via an adversary. The adversary knows the hidden element x and receives the queries from the questioner as the tree is being traversed. The adversary is allowed to lie at most k times, where each lie is a violation of the above stated rule. Formally, an adversary is a mapping that receives as input an element $x \in X$, a sequence of the previous queries and their answers, and an additional query $Q \subseteq U$, which represents the current query. The output of the adversary is a boolean answer to the current query; this answer is a *lie* if it differs from the truth value of "$x \in Q$".

We also allow the adversary and the tree to use randomness: a randomized decision tree is a distribution over decision trees and a randomized adversary is a distribution over adversaries.

Computation and complexity

The responses of the adversary induce a unique root-to-leaf path in the decision tree, which results in the output of the tree. A decision tree is *k-valid* if it outputs the correct element against any adversary that lies at most k times.

Given a k-valid decision tree T and a distribution μ on U, the *cost* of T with respect to μ, denoted $c(T, \mu)$, is the maximum, over all possible adversaries that lie at most k times, of the expected[3] length of the induced root-to-leaf path in T. Finally, the k-cost of μ, denoted $c_k(\mu)$, is the minimum of $c(T, \mu)$ over all k-valid decision trees T.

[3] The expectation is also taken with respect to the randomness of the adversary and the tree when they are randomized.

Basic facts

We will refer to the following well-known formula as *Kraft's identity*:

▶ **Fact 1** (Kraft's identity)**.** *Fix a binary tree T, let L be its set of leaves and let $d(\ell)$ be the depth of leaf ℓ. The following applies:*

$$\sum_{\ell \in L(T)} 2^{-d(\ell)} \leq 1.$$

We will use the following basic lower bound on the expected depth by the entropy:

▶ **Fact 2.** *Let T be a binary tree and let μ be a distribution over its leaves. Then*

$$H(\mu) \leq \mathop{\mathbb{E}}_{\ell \sim \mu}\big[d(\ell)\big].$$

In other words, for any distribution μ, $c_0(\mu) \geq H(\mu)$. In fact, it is also known that $c_0(\mu) \leq H(\mu) + 1$.

3 Main results

This section is organized as follows: The lower bound is presented in Section 3.1. Then, the two searching algorithms are presented in Section 3.2, and finally the application to sorting is presented in Section 3.3.

3.1 Lower bound

In this section we present the following lower bound on $c_k(\mu)$, namely, on the expected number of questions asked by *any* k-valid tree (not necessarily a comparison trees).

▶ **Theorem 3.** *For every non-constant distribution μ and every $k \geq 0$,*

$$c_k(\mu) \geq \Big(\mathop{\mathbb{E}}_{x\sim\mu} \log\frac{1}{\mu(x)}\Big) + k\Big(\mathop{\mathbb{E}}_{x\sim\mu} \log\log\frac{1}{\mu(x)}\Big) - (k\log k + k + 1).$$

Proof overview

Consider a k-valid tree; we wish to lower bound the expected number of questions for $x \sim \mu$. Let d_x denote the number of questions asked when the secret element is x. Then, by the entropy lower bound when the number of mistakes is $k = 0$, it follows that *typically*, $d_x \gtrsim \log(1/\mu(x))$. Moreover, the transcript of the game (i.e. the list of questions and answers) determines both x and the positions of the k lies. This requires

$$d_x + k\log(d_x) \gtrsim \log(1/\mu(x)) + k\log\log(1/\mu(x))$$

bits of information. Taking expectation over $x \sim \mu$ then yields the stated bound.

Our proof formalizes this intuition using standard and basic tools from information theory. One part that requires a subtler argument is showing that indeed one may assume that $d_x \gtrsim \log(1/\mu(x))$ for all x. This is done by showing that any k-valid tree can be modified to satisfy this constraint without increasing the expected number of questions by too much.

3.2 Upper bounds

We introduce two algorithms. The first algorithm, presented in Section 3.2.1, is simpler, however, the second algorithm has a better query complexity. The expected number of questions asked by the first algorithm is at most

$$H(\mu) + (k+1)H_2(\mu) + O(k^2 H_3(\mu) + k^2 \log k), \qquad \text{where } H_3(\mu) = \sum_x \mu(x) \log\log\log \frac{1}{\mu(x)}.$$

The second algorithm, presented in Section 3.2.2, removes the quadratic dependence on k, and has an expected complexity of:

$$H(\mu) + kH_2(\mu) + O(kH_3(\mu) + k\log k).$$

In Section 3.2.3 we robustify the guarantees of these algorithms and consider scenarios where the exact distribution μ is not known but only some prior $\eta \approx \mu$, or where the actual number of lies is less than the bound k (whence the algorithm achieves better performance).

3.2.1 First algorithm

Suppose that we are given a probability distribution μ whose support is the linearly ordered set $x_1 \prec \cdots \prec x_n$. In this section we overview the proof of the following theorem:

▶ **Theorem 4.** *There is a k-valid comparison tree T with*

$$c(T,\mu) \le H(\mu) + (k+1)\sum_{i=1}^{n} \mu_i \log \overline{\log} \frac{1}{\mu_i} + O\left(k^2 \sum_{i=1}^{n} \mu_i \log\log \overline{\log} \frac{1}{\mu_i} + k^2 \log k\right),$$

where $\mu_i = \mu(x_i)$.

The question-asking strategy simulates a binary search to recover the hidden element. If, at some point, the answer to some question q is suspected as a lie then q is asked $2k+1$ times to verify its answer. When is the answer to q suspected? The binary search tree is constructed in a manner that if no lies are told then roughly half of the questions are answered $\prec$, and half $\succeq$. However, if, for example, the lie "$x \succeq x_{50}$" is told when in fact $x = x_{10}$, then all consecutive questions will be of the form "$x \prec x_i$?" for $i > 50$, and the correct answer would always be $\prec$. Since no more than k lies can be told, almost all consecutive questions will be answered $\prec$, and the algorithm will suspect that some earlier question is a lie.

We start by suggesting a question-asking strategy using comparison queries which is valid as long as there are no lies, and then show how to make it resilient to lies. Each element x_i is mapped to a point p_i in $[0,1]$, such that $p_1 < p_2 < \cdots < p_n$. Then, a binary search on the interval $[0,1]$ is performed, for finding the point p_i corresponding to the hidden element. The search proceeds by maintaining a *Live* interval, which is initialized to $[0,1]$. At any iteration, the questioner asks whether p_i lies in the left half of the *Live* interval. The interval is updated accordingly, and its length shrinks by a factor of 2. This technique was proposed by Gilbert–Moore [12], and is presented in AuxiliaryAlgorithm 1, as an algorithm which keeps asking questions indefinitely.

The points $p_1, \ldots, p_n$ are defined as follows: first, a number $\theta \in [0, 1/2)$ is drawn uniformly at random. Now, for any element i define $p_i = \frac{1}{2}\sum_{j=1}^{i-1} \mu_j + \frac{1}{4}\mu_i + \theta$.[4] Given θ, let T'_θ denote the infinite tree generated by AuxiliaryAlgorithm 1. Note that whenever *Live* contains just

[4] In the original paper p_i was defined similarly but without the randomization: $p_i = \sum_{j=1}^{i-1} \mu_j + \frac{1}{2}\mu_i$.

Algorithm 1 Randomized Gilbert–Moore.

```
1: $Live \leftarrow [0,1]$
2: loop
3:     $m \leftarrow$ midpoint of $Live$
4:     $X \leftarrow \{i : p_i \geq m\}$
5:     if $x \in X$ then
6:         $Live \leftarrow$ right half of $Live$
7:     else
8:         $Live \leftarrow$ left half of $Live$
9:     end if
10: end loop
```

one point p_i, then (as there are no lies) the hidden element must be x_i. Denote by T_θ the finite tree corresponding to the algorithm which stops whenever that happens. We present two claims about these trees.

First, conditioned on any hidden element x_i, the answers to all questions (except, perhaps, for the first answer) are distributed uniformly and independently, where the distribution is over the random choice of θ. This follows from the fact that all bits of p_i except for the most significant bit are i.i.d. unbiased coin flips.

▷ Claim 5. For any element x_i, let (A_t) be the random sequence of answers to the questions in AuxiliaryAlgorithm 1, containing all answers except for the first answer, assuming there are no lies. The distribution of the sequence (A_t) is the same as that of an infinite sequence of independent unbiased coin tosses, where the randomness stems from the random choice of p_i.

Second, since $\min(p_i - p_{i-1}, p_{i+1} - p_i) \geq \mu_i/4$, one can bound the time it takes to isolate x_i as follows.

▷ Claim 6. For any element x_i and any θ, the leaf in T_θ labeled by x_i is of depth at most $\log(1/\mu_i) + 3$. Hence, if x is drawn from a distribution μ, the expected depth of the leaf labeled x is at most $\sum_i \mu_i \log(1/\mu_i) + 3 = H(\mu) + 3$.

We now describe the k-resilient algorithm: Algorithm 1 (the pseudocode appears as well). At the beginning, a number θ is randomly drawn. Then, two concurrent simulations over T'_θ are performed, and two pointers to nodes in this tree are maintained (recall that T'_θ is the infinite binary search tree). The first pointer, *Current*, simulates the question-asking strategy according to T'_θ, ignoring the possibility of lies. In particular, it may point on an incorrect node in the tree (reached as a result of a lie). Since *Current* ignores the possibility of lies, there is a different pointer, *LastVerified*, which verifies the answers to the questions asked in the simulation of *Current*. All answers in the path from the root to *LastVerified* are verified as correct, and *LastVerified* will always be an ancestor of *Current*. See Figure 3 for the basic setup.

The algorithm proceeds in iterations. In every iteration the question $Q(\mathit{Current})$ is asked and *Current* is advanced to the corresponding child accordingly. In some of the iterations also *LastVerified* is advanced. Concretely, this happens when the depth of *Current* in T'_θ equals $d + r(d)$, where d is the depth of *LastVerified* and $r(d) \approx \log d + k \log\log d$. In these iterations, the answer given to $Q(\mathit{LastVerified})$ is being verified, as detailed next.

Algorithm 1 Resilient-Tree.

```
1: $\theta \leftarrow \mathrm{Uniform}([0, 1/2))$
2: $\mathit{Current} \leftarrow \mathrm{root}(T'_\theta)$
3: $\mathit{LastVerified} \leftarrow \mathrm{root}(T'_\theta)$
4: while $\mathit{LastVerified}$ is not a leaf of $T_\theta$ do
5:     if $x \in Q(\mathit{Current})$ then
6:         $\mathit{Current} \leftarrow$ left-child($\mathit{Current}$)
7:     else
8:         $\mathit{Current} \leftarrow$ right-child($\mathit{Current}$)
9:     end if
10:    $d \leftarrow \mathrm{depth}(\mathit{LastVerified}) + 1$
11:    if $\mathrm{depth}(\mathit{Current}) = d + r(d)$ then
12:        $\mathit{Candidate} \leftarrow$ child of $\mathit{LastVerified}$ which is an ancestor of $\mathit{Current}$
13:        $\mathit{VerificationPath} \leftarrow$ ancestors of $\mathit{Current}$ up to and excluding $\mathit{Candidate}$
14:        if $\mathit{Candidate}$ is a left (right) child and at most $k-1$ vertices in $\mathit{VerificationPath}$
               are left (right) children then
15:            Ask $2k+1$ times the question $x \in Q(\mathit{LastVerified})$
16:            if majority answer is $x \in Q(\mathit{LastVerified})$ then
17:                $\mathit{LastVerified} \leftarrow$ left-child($\mathit{LastVerified}$)
18:            else
19:                $\mathit{LastVerified} \leftarrow$ right-child($\mathit{LastVerified}$)
20:            end if
21:            $\mathit{Current} \leftarrow \mathit{LastVerified}$
22:        else
23:            $\mathit{LastVerified} \leftarrow \mathit{Candidate}$
24:        end if
25:    end if
26: end while
27: return label of $\mathit{LastVerified}$
```

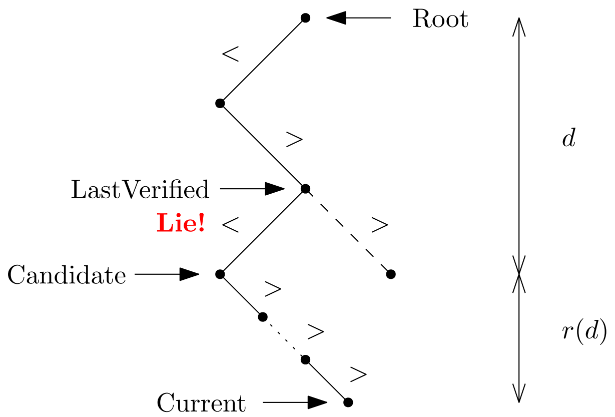

Figure 3 An illustration of Algorithm 1 just before the detection of a lie. The answer at *LastVerified* was a lie ($<$ instead of $>$), and so all answers below *Candidate* (except for any further lies) are $>$. This is noticed since *Current* is at depth $d + r(d)$. The answer at *Candidate* will be verified and found wrong, and so *LastVerified* would move to the sibling of *Candidate* (and so will *Current*), and the algorithm will continue from that point.

The verification process

Next, we examine the verification process when *LastVerified* is advanced. There are two possibilities: first, when the answer to the question $Q(\mathit{LastVerified})$, which was given when *Current* traversed it, is verified to be correct. In that case, *LastVerified* moves to its child which lies on the path towards *Current*. In the complementing case, when the the answer to the question $Q(\mathit{LastVerified})$ is detected as a lie, then *LastVerified* moves to the other child. In that case, *Current* is no longer a descendant of *LastVerified*, hence *Current* is moved up the tree and is set to *LastVerified*.

We now explain how the answer to $Q(\mathit{LastVerified})$ is verified. There are two verification steps: the first step uses no additional questions and the second step uses $2k+1$ additional questions. Usually, only the first step will be used and no additional questions will be spent during verification. In the first verification step one checks whether the following condition holds:

The answer to $Q(\mathit{LastVerified})$ is identical to at least k of the answers along the path from LastVerified to Current.

If this condition holds, then the answer is verified as correct. To see why this reasoning is valid, assume without loss of generality that the answer is $\prec$, and assume towards contradiction that it was a lie. Then, the correct answers to all following questions in the simulation of *Current* are $\succeq$. Since there can be at most $k-1$ additional lies, there can be at most $k-1$ additional $\prec$ answers. Hence, if there are more $\prec$ answers among the following questions then the previous answer to $Q(\mathit{LastVerified})$ is verified as correct.

Else, if the above condition does not hold then one proceeds to the second verification step and asks $2k+1$ times the question $Q(\mathit{LastVerified})$. Here, the majority answer must be correct, since there can be at most k lies.

We add one comment: if the second verification step is taken, one sets $\mathit{Current} \leftarrow \mathit{LastVerified}$ regardless of whether a lie had been revealed (this is performed to facilitate the proof). So, whenever the condition in the first verification step fails to hold then *Current* and *LastVerified* point to the same node in the tree.

The algorithm ends when *LastVerified* reaches a leaf of T_θ, at which point the hidden element is recovered.

Query complexity analysis

Fix an element x_i. We bound the expected number of questions asked when x_i is the hidden element as follows. Define $d \approx \log(1/\mu(x_i))$ as the depth of the leaf labeled x_i in T_θ. We divide the questions into the following five categories:

- Questions on the path P from the root to *Current* by the end of the algorithm, when *Current* reaches depth $d+r(d)$, *LastVerified* reaches depth d, and the algorithm terminates. Hence, there are at most $d+r(d)$ such questions.
- Questions that were ignored due to the second verification step while *Current* was backtracked from a node outside P. This can only happen due to a lie between *Current* and *LastVerified* so there are at most $k \cdot r(d)$ such questions.
- Questions asked $2k+1$ times during the second verification step when *Current* was pointing to a node outside P. This can only happen due to a lie between *Current* and *LastVerified* so there are at most $k \cdot (2k+1)$ such questions.
- Questions that were ignored due to the second verification step, when *Current* was being backtracked from a node in P. By the choice of $r(d)$ there are at most $O(1)$ such questions (on expectation).
- Questions asked $2k+1$ during the second verification step when *Current* was pointing to a node in P. By the choice of $r(d)$ there are at most $O(1)$ such questions (in expectation).

Summing these bounds up, one obtains a bound of

$$\big(d + r(d)\big) + k \cdot r(d) + k \cdot (2k+1) + O(1) + O(1)$$
$$\approx \log(1/\mu_i) + (k+1)\Big(\log\log(1/\mu_i) + k \log\log\log(1/\mu_i) + O(k)\Big).$$

3.2.2 Second algorithm

In this section we overview the proof of the following theorem.

▶ **Theorem 7.** *For any distribution μ there exists a k-valid comparison tree T with*

$$c(T,\mu) \le H(\mu) + k\mathbb{E}_{x\sim\mu}\left[\log\overline{\log}(1/\mu(x))\right] + O(k\mathbb{E}_{x\sim\mu}\left[\log\log\overline{\log}(1/\mu(x))\right] + k\log k).$$

We explain the key differences with Algorithm 1.

- In Algorithm 1, an answer to a question Q at depth d was suspected as a lie if at most k of the $r(d)$ consecutive questions received the same answer as Q. In the new algorithm, we suspect a question Q if *all* the $r'(d)$ consecutive answers are different than Q. This change enables setting $r'(d) \approx \log d$ rather than the previous value of $r(d) \approx \log d + k \log\log d$. Similarly to Algorithm 1, any time a lie is deleted, $r'(d)$ questions are being deleted. Summing over the k lies, one obtains a total of $kr'(d) \approx k \log d$ deleted questions, which is smaller than the corresponding value of $kr(d) \approx k \log d + k^2 \log\log d$ in Algorithm 1.
- In Algorithm 1, the lies were detected in the same order they were told (i.e. in a *first-in-first-out* queue-like manner). This is due to the semantic of the pointer *LastVerified* which verifies the questions one-by-one, along the branching of the tree. In Algorithm 2 the pointer *LastVerified* is removed (only *Current* is used), and the lies are detected in a *last-in-first-out* stack-like manner: only the last lie can be deleted at any point in time. Indeed, as described in the previous paragraph, a lie will be deleted only if all consecutive answers are different (which is equivalent to them being non-lies).
- In Algorithm 1, when an answer is suspected as a lie, the corresponding question Q is repeated $2k+1$ times in order to verify its correctness. This happens after each lie, hence $\Omega(k)$ redundant questions are asked per lie. In Algorithm 2, the suspected question Q will be asked again only *once*, and the algorithm will proceed accordingly. It may however be the case that this process will repeat itself and also the second answer to this question will be suspected as a lie and Q will be asked once again and so on. In order to avoid an infinite loop we add the condition that if the same answer is told $k+1$ times then it is guaranteed to be correct and will not be suspected any more.
- The removal of *LastVerified* forces finding a different method of verifying the correctness of an element x upon arriving at a leaf of T_θ. One option is to ask the question "element $= x$?" $2k+1$ times and take the majority vote, where each $=$ question is implemented using one $\preceq$ and one $\succeq$. This will, however, lead to asking $\Omega(k)$ redundant questions each time x is not the correct element. Instead, one asks "element $= x$?" multiple times, stopping either when the answer $=$ is obtained $k+1$ times, or by the first the answer $\neq$ has obtained more than the answer $=$. The total redundancy imposed by these verification questions throughout the whole search is $O(k)$.

To put the algorithm together, we exploit some simple combinatorial properties of paths containing multiple lies.

3.2.3 A fine-grained analysis of the guarantees

In this section, we present a stronger statement for the guarantees of our algorithms. First, the algorithms do not have to know *exactly* the distribution μ from which the hidden element is drawn: an approximation suffices for getting a similar bound. Recall that the algorithm gets as an input some probability distribution η. This distribution might differ from the true distribution μ. The cost of using η rather than μ is related to $D(\mu\|\eta)$, the Kullback–Leibler divergence between the distributions.

Secondly, the algorithm has stronger guarantees when the actual number of lies is less than k. This is an improvement comparing to the algorithm of Rivest et al. [21] mentioned in the introduction. It will be utilized in the application of sorting, where the searching algorithm is invoked multiple times with a bound on the total number of lies (rather the number of lies per iteration). We present the general statement with respect to Algorithm 2.

▶ **Theorem 8.** *Assume that Algorithm 2 is invoked with the distribution $(\eta_1, \dots, \eta_n)$. Then, for any element x_i, the expected number of questions asked when x_i is the secret is at most*

$$\log(1/\eta_i) + \mathbb{E}[K'] \log \overline{\log} \frac{1}{\eta_i} + O\left(\mathbb{E}[K'] \log\log \overline{\log} \frac{1}{\eta_i} + \mathbb{E}[K'] \overline{\log} k + k\right),$$

where K' is the expected number of lies. (The expectation is taken over the randomness of both parties.)

As a corollary, one obtains Theorem 7 and the following corollary, which corresponds to using a distribution different from the actual distribution.

▶ **Corollary 9.** *Assume that Algorithm 2 is invoked with $(\eta_1, \dots, \eta_n)$ while $(\mu_1, \dots, \mu_n)$ is the true distribution. Then, for a random hidden element drawn from μ, the expected number of questions asked is at most*

$$H(\mu) + k\mathbb{E}_{x\sim\mu}\left[\log \overline{\log}(1/\mu(x))\right] + O(k \mathop{\mathbb{E}}_{\mu}\left[\log\log \overline{\log}(1/\mu(x))\right] + k\overline{\log} k)$$
$$+ D(\mu\|\eta) + O(k \log D(\mu\|\eta)),$$

where $D(\mu\|\eta) = \sum_{x\in \operatorname{supp}\mu} \mu(x) \log \frac{\mu(x)}{\eta(x)}$ is the Kullback–Leibler divergence between μ and η.

Corollary 9 follows from Theorem 8 by bounding $K' \le k$, taking expectation over $x_i \sim \mu$, noting that $\sum_i \mu_i \log(1/\eta_i) = H(\mu) + D(\mu\|\eta)$ and applying Jensen's inequality with the function $x \mapsto \log x$.

3.3 Sorting

One can apply Algorithm 2 to implement a stable version of the insertion sort using comparison queries. Let Π be a distribution over the set of permutations on n elements. Complementing with prior algorithms achieving a complexity of $H(\Pi) + O(n)$ in the randomized setting with no lies [17], and $n \log n + O(nk + n)$ in the deterministic setting with k lies [3, 16], we present an algorithm with a complexity of $H(\Pi) + O(nk + n + k\log k)$ in the distributed setting with k lies. Note that $k \log k = O(nk)$ unless unless the unlikely case that $k = e^{w(n)}$, hence the $k\log k$ term can be ignored. Therefore, the guarantee of our algorithm matches the guarantees of the prior algorithms substituting either $k = 0$ or $\Pi =$ Uniform.

▶ **Theorem 10.** *Assume a distribution Π over the set of all permutations on n elements. There exists a sorting algorithm which is resistant to k lies and sorts the elements using $H(\Pi) + O(nk + n + k\log k)$ comparisons on expectation.*

The randomized algorithms benefit from prior knowledge, namely, when one has information about the correct ordering. This is especially useful for maintaining a sorted list of elements, a procedure common in many sequential algorithms. In these settings, the values of the elements can change in time, hence, the elements have to be re-sorted regularly, however, their locations are not expected to change drastically.

The suggested sorting algorithm performs n iterations of insertion sort. By the end of each iteration i, $x_1, \dots, x_i$ are successfully sorted. Then, on iteration $i+1$, one performs a binary search to find the location where x_{i+1} should be inserted, using conditional probabilities.

The guarantee of the algorithm is asymptotically tight: a lower bound of $H(\Pi)$ follows from information theoretic reasons, and a lower bound of $\Omega(nk)$ follows as well: the bound of Lakshmanan et al. [15] can be adjusted to the randomized setting.

References

1 Martin Aigner. Finding the maximum and minimum. *Discrete Applied Mathematics*, 74:1–12, 1997.

2 Javed A. Aslam and Aditi Dhagat. Searching in the presence of linearly bounded errors. In *Proceedings of the 23rd annual Symposium on Theory of Computing (STOC'91)*, pages 486–493, 1991.

3 A. Bagchi. On sorting in the presence of erroneous information. *Information Processing Letters*, 43:213–215, 1992.

4 Elwyn R. Berlekamp. *Block coding for the binary symmetric channel with noiseless, delayless feedback*, pages 61–85. Wiley, New York, 1968.

5 R. Sean Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. In *Proceedings of the 25th annual symposium on theory of computing (STOC'93)*, pages 130–136, 1993.

6 Mark Braverman and Elchanan 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.

7 Yuval Dagan, Yuval Filmus, Ariel Gabizon, and Shay Moran. Twenty (simple) questions. In *49th ACM Symposium on Theory of Computing (STOC 2017)*, 2017.

8 Yuval Dagan, Yuval Filmus, Daniel Kane, and Shay Moran. The entropy of lies: playing twenty questions with a liar. *arXiv preprint*, 2018. `arXiv:1811.02177`.

9 Robert Mario Fano. The transmission of information. Technical Report 65, Research Laboratory of Electronics at MIT, Cambridge (Mass.), USA, 1949.

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

11 Ran Gelles et al. Coding for interactive communication: A survey. *Foundations and Trends® in Theoretical Computer Science*, 13(1–2):1–157, 2017.

12 E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38:933–967, 1959.

13 Bernhard Haeupler. Interactive channel capacity revisited. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 226–235. IEEE, 2014.

14 Gillat Kol and Ran Raz. Interactive channel capacity. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 715–724. ACM, 2013.

15 K.B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Transactions on Computers*, 40:1081–1084, 1991.

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

17 Shay Moran and Amir Yehudayoff. A note on average-case sorting. *Order*, 33(1):23–28, 2016.

18 Andrzej Pelc. Coding with bounded error fraction. *Ars Combinatorica*, 42:17–22, 1987.

19 Andrzej Pelc. Searching games with errors—fifty years of coping with liars. *Theoretical Computer Science*, 270:71–109, 2002.

20 Alfréd Rényi. On a problem of information theory. *MTA Mat. Kut. Int. Kozl.*, 6B:505–516, 1961.

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

22 Leonard J Schulman. Coding for interactive communication. *IEEE transactions on information theory*, 42(6):1745–1756, 1996.

23 Claude Elwood Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948.

24 Joel Spencer and Peter Winkler. Three thresholds for a liar. *Combin. Probab. Comput.*, 1(1):81–93, 1992. doi:10.1017/S0963548300000080.

25 Stanislav M. Ulam. *Adventures of a mathematician.* Scribner's, New York, 1976.