# Mastermind with a linear number of queries



**Author(s):**
Martinsson, Anders; Su, Pascal

**Publication date:**
2024-03

**Permanent link:**
https://doi.org/10.3929/ethz-b-000647235



**Originally published in:**
Combinatorics, Probability & Computing 33(2), https://doi.org/10.1017/S0963548323000366

**Funding acknowledgement:**
200021_156011 - Hierarchical carbon-fiber composites with tailored interphase obtained via electrophoretic deposition of magnetized and funtionalized carbon nanotubes (SNF)
169242 - Saturation Games and Robust Random Structures (SNF)

*Combinatorics, Probability and Computing* (2024), **33**, pp. 143–156
doi:10.1017/S0963548323000366

CAMBRIDGE UNIVERSITY PRESS

ARTICLE

# Mastermind with a linear number of queries

Anders Martinsson and Pascal Su

Department of Computer Science, ETH Zurich, Zurich, Switzerland
**Corresponding author:** Anders Martinsson; Email: anders.martinsson@inf.ethz.ch

(Received 23 January 2022; revised 19 September 2023; accepted 25 September 2023;
first published online: 8 November 2023)

**Abstract**

Since the 1960s Mastermind has been studied for the combinatorial and information-theoretical interest the game has to offer. Many results have been discovered starting with Erdős and Rényi determining the optimal number of queries needed for two colours. For $k$ colours and $n$ positions, Chvátal found asymptotically optimal bounds when $k \leq n^{1-\varepsilon}$. Following a sequence of gradual improvements for $k \geq n$ colours, the central open question is to resolve the gap between $\Omega(n)$ and $\mathcal{O}(n \log\log n)$ for $k = n$. In this paper, we resolve this gap by presenting the first algorithm for solving $k = n$ Mastermind with a linear number of queries. As a consequence, we are able to determine the query complexity of Mastermind for any parameters $k$ and $n$.

**Keywords:** Mastermind; coin-weighing; information theory; combinatorial games; query complexity

## 1. Introduction

Mastermind is a famous code-breaking board game for two players. One player, codemaker, creates a hidden codeword consisting of a sequence of four colours. The goal of the second player, the codebreaker, is to determine this codeword in as few guesses as possible. After each guess, codemaker provides a certain number of black and white pegs indicating how close the guess is to the real codeword. The game is over when codebreaker has made a guess identical to the hidden string.

The board game version was first released in 1971, though the idea of the game is older, and variations of this game have been played earlier under other names, such as the pen-and-paper-based games of Bulls and Cows, and Jotto. The game has been played on TV as a game show in multiple countries under the name of Lingo. Recently, a similar web-based game has gained much attention under the name of Wordle.

Guessing games such as Mastermind have gained much attention in the scientific community. This is in part due to their popularity as recreational games, but importantly also as natural problems in the intersection of information theory and algorithms. In particular, it is not too hard to see that two-colour Mastermind is equivalent to coin-weighing with a spring scale. This problem was first introduced in 1960 by Shapiro and Fine [14]. In subsequent years, a number of different approaches have been devised which solve this problem up to a constant factor of the information-theoretic lower bound.

The general $k$ colour $n$ slot Mastermind first appeared in the scientific literature in 1983 in a paper by Chvátal [3]. By extending ideas of Erdős and Rényi [5] from coin-weighing he showed

Pascal Su was supported by grant no. 200021 169242 of the Swiss National Science Foundation.

that the information-theoretic lower bound is sharp up to a constant factor for $k \leq n^{1-\varepsilon}$ for any fixed $\varepsilon > 0$.

Surprisingly, for a larger number of colours, the number of guesses needed to reconstruct the codeword has remained unknown. In particular, for $k = n$, the best known upper bound remained for a long time $\mathcal{O}(n \log n)$ as shown by Chvátal, with only constant factor improvements given in [2], [8], and [9]. Only quite recently, this bound was improved to $\mathcal{O}(n \log \log n)$ in an article by Doerr, Doerr, Spöhel, and Thomas [4] published in Journal of the ACM in 2016. At the same time, no significant improvement on the information-theoretic lower bound of $n$ queries has been obtained.

The challenge of finding short solutions to Mastermind in the setting of $k = n$ colours and positions can be thought of as a bootstrapping paradox. On the one hand, a query could in principle return anything between 0 and $n$ black pegs, awarding codebreaker potentially with as many as $\log(n+1)$ bits of information about the codeword. As it takes $\log(n^n) = n \log n$ bits of information to encode an arbitrary codeword, it follows that any strategy needs $\Omega(n \log n / \log(n+1)) = \Omega(n)$ queries. On the other hand, given no prior information about the codeword, codebreaker can only expect to get a constant number of correct guesses, meaning that the query only awards codebreaker with $\mathcal{O}(1)$ bits of information.[1] Morally, one expects queries with higher information content to gradually become available as codebreaker gains information about the codeword, but making sense of this formally has turned out to be difficult problem. In particular, should one expect the entropy lower bound of $\Omega(n)$ to be the truth, or could it be that this bound is unattainable as it takes too long to reach the point where queries give the full $\Theta(\log n)$ bits of information?

In this paper, we resolve this problem after almost 40 years by showing how the $n$ colour $n$ slot Mastermind can be solved with $\mathcal{O}(n)$ guesses with high probability, matching the information-theoretic lower bound up to a constant factor. By combining this with a result by Doerr, Doerr, Spöhel, and Thomas [4], we determine asymptotically the optimal number of guesses for all $k$ and $n$.

### *1.1 Game of Mastermind*

We define a game of Mastermind with $k \geq 1$ *colours* and $n \geq 1$ *positions* as a two-player game played as follows. One player, the codemaker, initially chooses a hidden codeword $c = (c_1, \ldots, c_n)$ in $[k]^n$. The other player, the codebreaker, is then tasked with determining the hidden codeword by submitting a sequence of queries of the form $q = (q_1, \ldots, q_n) \in [k]^n$. For each query, the codemaker must directly respond with information on how well the query matches the codeword. The codebreaker may use this information to adapt subsequent queries. The precise information given depends on which variation of Mastermind is played, as will be specified below. The game is over as soon as codebreaker makes a query such that $q = c$. The goal of the codebreaker is to make the game ends after as few queries as possible.

For each pair of a codeword $c$ and a query $q$, we associate two integers called the number of *black pegs* and *white pegs,* respectively. The number of black pegs,

$$b(q) = b_c(q) := |\{i \in [n] : q_i = c_i\}|,$$

is the number of positions in which the codeword matches the query string. The number of white pegs is often referred to as the number of correctly guessed colours that do not have the correct position. More precisely, the number of white pegs,

$$w(q) = w_c(q) := \max_{\sigma \in S_n} |\{i \in [n] : q_{\sigma(i)} = c_i\}| - b_c(q),$$

[1] For simplicity, we here assume that codebreaker only receives the number of black pegs for each query. The situation is similar if also white pegs are provided, but it takes additional arguments to see that not too much additional information can come from the white pegs.

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

is the number of additional correct positions one can maximally obtain by permuting the entries of $q$.

In this paper, we will consider two versions of Mastermind. First, in *black-peg* Mastermind the codemaker gives only the number of black pegs as an answer to every query. Second, in *black-white-peg* Mastermind the codemaker gives both the number of black and the number of white pegs as answers to every query.

### 1.2 Our results

Our contribution lies in resolving the black-peg Mastermind game for $k = n$ where we have as many possible colours as positions.

**Theorem 1.1.** *There exists a randomised algorithm that solves black-peg Mastermind with n colours and n positions with high probability using $\mathcal{O}(n)$ queries. Moreover, the runtime of the algorithm is polynomial in n.*

The above result is best possible up to a constant factor. This can be seen by observing that there are at most $n + 1$ possible answers to any query. Hence, codebreaker gains at most $\log_2(n + 1)$ bits of information from a query. As $\log_2(n^n) = n \log_2 n$ bits are required to uniquely determine the codeword, any strategy needs at least $\Omega(n)$ queries. Moreover, if we additionally assume that the codeword can only be a permutation of the colours, that is each colour must appear exactly once, then we can solve it deterministically with $\mathcal{O}(n)$ queries.

Combining our results with earlier results by Doerr, Doerr, Spöhel, and Thomas we are able to resolve the randomised query complexity of Mastermind in the full parameter range, thus finally resolving this problem after almost 40 years. We define the randomised query complexity as the minimum (over all strategies) maximum (over all codewords) expected number of queries needed to win the game.

**Theorem 1.2.** *For k colours and n positions, the randomised query complexity of Mastermind is*

$$\Theta(n \log k / \log n + k/n),$$

*if codebreaker receives both black-peg and white-peg information for each query, and*

$$\Theta(n \log k / \log n + k),$$

*if codebreaker only receives black-peg information.*

We believe the same result holds true for the deterministic query complexity, but we will not attempt to prove it here.

### 1.3 Related work

The study of two-colour Mastermind dates back to an American Mathematical Monthly post in 1960 by Shapiro and Fine [14]: "*Counterfeit coins weigh 9 grams and genuine coins all weigh 10 grams. One is given n coins of unknown composition, and an accurate scale (not a balance). How many weighings are needed to isolate the counterfeit coins?*"

It can be observed, already for $n = 4$, that fewer than $n$ weighings are required. The authors consequently conjecture that $o(n)$ weighings suffice for large $n$. Indeed, the entropy lower bound states that at least $n/\log_2(n + 1)$ weighings are necessary. In the subsequent years, many techniques were independently discovered that attain this bound within a constant factor [1, 5, 12, 13], see also [5] for further early works. Erdős and Rényi [5] showed that a sequence of $(2 + o(1))n/\log_2 n$ random weighings would uniquely identify the counterfeit coins with high probability, and by the probabilistic method, there is a deterministic sequence of $\mathcal{O}(n/\log n)$ weighings that identify any set of counterfeit coins.

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

Cantor and Mills [1] proposed a recursive solution to this problem. Here it is natural to consider *signed* coin weighings, where, for each coin on the scale, we may choose whether it contributes with its weight or minus its weight. We call a $\{-1, 0, 1\}$ valued matrix $A$ an *identification matrix* if any binary vector $x$ of compatible length to $A$ can be uniquely determined by the values of $Ax$. So for example $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ is an identification matrix for binary vectors of length 2. It is not too hard to show that if $A$ is an identification matrix, then so is

$$\begin{pmatrix} A & A & I \\ A & -A & 0 \end{pmatrix}. \tag{1}$$

By putting $A_0 = (1)$ and recursing this formula, we obtain an identification matrix $A_k$ with $2^k$ rows and $(k+2)2^{k-1}$ columns. Thus, we can identify which out of $n = (k+2)2^{k-1}$ coins are counterfeit by using $2^k \sim 2n/\log_2 n$ *signed* weighings, or, by weighing the $+1$s and $-1$s separately, using $\sim 4n/\log_2 n$ (unsigned) weighings. Using a more careful analysis, the authors show that $\sim 2n/\log_2 n$ weighings suffice.

It was shown in [5] that $2n/\log_2 n$ is best possible, up to lower-order terms, for *non-adaptive* strategies. It is a central open problem to determine the optimal constant for general strategies, but it is currently not known whether adaptiveness can be used to get a leading term improvement.

Knuth [10] studied optimal strategies for the commercially available version of Mastermind, consisting of four positions and six colours. He showed that the optimal deterministic strategy needs 5 guesses in the worst case. In the randomised setting, it was shown by Koyama [11] that optimal strategy needs in expectation $5625/1296 = 4.34\ldots$ guesses for the worst-case distribution of codewords.

The generalisation of Mastermind to $k$ colours and $n$ positions first appeared in the scientific literature in 1983 in a paper by Chvátal [3], who attributed the idea to Pierre Duchet. Here the entropy lower bound states that $\Omega(n \log k/\log n)$ guesses are necessary. For $k \leq n^{1-\varepsilon}$, Chvátal showed that a simple random guessing strategy uniquely determines the codeword within a constant factor of the entropy bound.

For larger $k$, less has been known. For $k$ between $n$ and $n^2$, Chvátal showed that $2n \log_2 k + 4n$ guesses suffice. For any $k \geq n$, this was improved to $2n \log_2 n + 2n + \lceil k/n \rceil + 2$ by Chen, Cunha, and Homer [2], further to $n\lceil \log_2 k \rceil + \lceil (2 - 1/k)n \rceil + k$ by Goodrich [8], and again to $n\lceil \log_2 n \rceil - n + k + 1$ by Jäger and Peczarski [9]. As a comparison, we note that if $k = k(n)$ is polynomial in $n$, then the entropy lower bound is simply $\Omega(n)$. This gap is a very natural one as Doerr, Doerr, Spöhel, and Thomas [4] showed in a relatively recent paper that if one uses a non-adaptive strategy, there is in fact a lower bound of $\Omega(n \log n)$ when $k = n$. In the same paper, they also use an adaptive strategy to significantly narrow this gap, showing that $\mathcal{O}(n \log \log n)$ guesses suffice for $k = n$. Moreover, they present a randomised reduction from black-white-peg Mastermind to black-peg Mastermind which shows that

$$\mathrm{bwmm}\,(n, k) = \Theta(k/n + \mathrm{bmm}\,(n, n)),$$

for any $k \geq n$ where $\mathrm{bwmm}\,(n, k)$ denotes the randomised query complexity for black-white-peg Mastermind with $k$ colours and $n$ positions, and where $\mathrm{bmm}\,(n, n)$ denotes the randomised query complexity for black-peg Mastermind with $n$ colours and positions. As a consequence, they concluded that $\mathrm{bwmm}\,(n, k) = \mathcal{O}(n \log \log n + k/n)$ for all $k$ and $n$.

Stuckman and Zhang [15] showed that it is NP-hard to determine whether a sequence of guesses with black and white peg answers is consistent with any codeword. The analogous result was shown by Goodrich [8] assuming only black-peg answers are given. It was shown by Viglietta [16] that both of these results hold even for $k = 2$.

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

Variations of Mastermind have furthermore been proposed to model problems in security, such as revealing someone's identity by making queries to a genomic database [7], and API-based attacks to determine bank PINs [6].

## 2. Proof outline

The proof of Theorem 1.1 can be broken up in two main steps. In the first step, contained in Section 3, we reduce the traditional form of Mastermind to what we call *signed permutation Mastermind*. This can be described as the variation of Mastermind on $n$ positions and colours where

1. codemaker is restricted to choosing the codeword $c$ to be a permutation of $[n]$, and
2. codebreaker can make signed queries $q \in \{-n, \dots n\}$, where, after each query, codemaker must respond with the value

$$\hat{b}(q) = \hat{b}_c(q) := |\{i \in [n] | c_i = q_i\}| - |\{i \in [n] | c_i = -q_i\}|.$$

In other words, codebreaker can decide, for each position in a query, whether a correct guess should count as a $+1$ or as a $-1$. Note also that codebreaker can choose to leave an entry in a query "blank" by giving it the value 0, in which case it will never contribute to the returned value.

This is all done in preparation for the second step, contained in Section 4, where we take a more constructive approach and recursively as well as deterministically provide a set of adaptive queries that give the answer to the signed permutation Mastermind problem.

Our approach to determining the codeword in this version of Mastermind can be described as resolving $\mathcal{O}(n \log n)$ subtasks of the form "Given that a colour $x$ is present in an interval $I$, determine whether $x$ is present in the left or right half of $I$". In other words, we attempt a binary search for each colour. We can make such a task part of a query by putting $q_i = x$ for all indices $i$ in the left half of $I$, and putting $q_i = 0$ in all indices $i$ in the right half of $I$. Then this will contribute a with $+1$ to the answer of the query if colour $x$ is in the left half of $I$, and 0 and if $x$ is in the right half of $I$. Clearly, a single task can be resolved after a single query, however the challenge is to find a way to resolve these tasks by performing only $\mathcal{O}(n)$ queries.

Similar to the algorithm by Doerr, Doerr, Spöhel, and Thomas [4], we will base our solution on coin-weighing schemes. Here, we think of the $\mathcal{O}(n \log n)$ tasks described above as our coins, where a coin has the value 1 if the colour in the corresponding task is present in the left half of its interval, and 0 otherwise. Weighing a collection of coins together corresponds to making one query where each of the corresponding tasks is encoded as described above. Note that if there were no further restrictions on how tasks could be queried together, then any of the classical solutions to coin-weighing would let us resolve the $\mathcal{O}(n \log n)$ tasks in $\mathcal{O}(n \log n / \log (n \log n)) = \mathcal{O}(n)$ queries, which is the conclusion we want, but under far too weak assumptions. The classical coin-weighing problem assumes that *any* collection of coins can be weighed together. This is far from true in the problem at hand. First, we cannot encode two tasks into the same query if their corresponding intervals overlap. Second, tasks depend on each other in the sense that certain tasks are only available for querying after another task is resolved. For instance, we cannot query in which quarter of the codeword the colour red is present until we have first determined in which half of the codeword it is in. This is of particular concern for tasks corresponding to big intervals as, on the one hand, they, intuitively, have little potential to be resolved in parallel, and on the other hand, these are the only tasks that are available initially.

One natural approach to circumvent this is to resolve the tasks ordered by the interval size from large to small, layer by layer. That is, we first determine which half each colour is present in by querying colours one at a time. We then determine which quarter each colour is present

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

in two at a time, and so on. That is, in the $i$th layer we query $2^i$ colours at a time. Using optimal coin-weighing schemes, this can be done in $\mathcal{O}((n/2^i) \cdot 2^i / \log(2^i)) = \mathcal{O}(n/i)$ queries. Alas, this is too slow as summing this over all layers $i = 0, \dots, \log_2 n$ gives a total of $\mathcal{O}(n \log \log n)$ queries. In order to speed this up to $\mathcal{O}(n)$ queries, we need to find a novel take on coin-weighing schemes where intervals of all different sizes are queried together in one big phase, which respects the dependencies between tasks.

In fact, our solution to this problem is surprisingly elegant. We start by performing a procedure we call *preprocess*. The aim of which is to use $\mathcal{O}(n)$ queries (so far without any information-theoretic speedup) in order to determine the positions of colours sufficiently well so that, after this point, it is possible to query $n^{\Omega(1)}$ colours together. We then perform a procedure we call *solve* that employs the parallelism unlocked by *preprocess* to determine the codeword in an additional $\mathcal{O}(n)$ queries. This procedure is recursively constructed in a manner similar to divide and conquer algorithms, and using a technique similar to the coin-weighing scheme by Cantor-Mills [1] to achieve the information-theoretic speedup.

Section 5 is then dedicated to the general case where the number of colours $k$ and number of positions $n$ may differ Mastermind. We prove Theorem 1.2. This can be seen as a direct consequence of Theorem 1.1 and applying previous results by Doerr, Doerr, Spöhel, and Thomas [4].

## 3. Signed permutation Mastermind

In the following section, we show how to reduce the traditional form of Mastermind to *signed permutation Mastermind*, as defined in Section 2.

### 3.1 Finding a zero query

We first show how one can simulate "blank" guesses in Mastermind. To achieve this, it suffices to find a query $z$ such that $b(z) = 0$, which can be done as follows.

**Lemma 3.1.** *For any $k \geq 2$, it is possible to find a string $q$ with $b_c(q) = 0$ using at most $n + 1$ queries.*

**Proof.** Query the string of all 1s, $t^{(0)}$, as well as the strings $t^{(i)}$ for all $i \in \{1, \dots, n\}$ where $t^{(i)}$ is the string of all ones except at the $i$th position it has a 2. Now if $b_c(t^{(i)}) = b_c(t^{(0)}) - 1$, then $c_i \neq 2$, otherwise $c_i \neq 1$ and we have at every position a colour that is incorrect, so we have a string $z$ which satisfies $b_c(z) = 0$. □

Note that if $k = n$ and we are content with a randomised search then we can find an all-zero string by choosing queries $z \in [n]^n$ uniformly at random until a query is obtained with $b_c(z) = 0$. As the success probability of one iteration $(1 - 1/n)^n \geq 1/4$, this takes on average 4 guesses.

### 3.2 Finding pairwise elementwise distinct one queries

We say that a set of queries $f^{(1)}, f^{(2)}, \dots$ are pairwise elementwise distinct if $f_i^{(s)} \neq f_i^{(t)}$ for all $i \in [n]$ and all $s \neq t$.

The second part of the reduction is to show how codebreaker can transform the problem to the setting where the codeword is a permutation of $[n]$. It turns out codebreaker can achieve this by first finding $n$ pairwise elementwise distinct queries $f^{(1)}, \dots f^{(n)}$ such that $b_c(f^{(i)}) = 1$ for all $i$. This can be done with high probability using $\mathcal{O}(n)$ random queries in the following fashion.

This argument is due to Angelika Steger (from personal communication).

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

**Algorithm 1:** Permutation

**Output:** $n$ pairwise elementwise distinct strings $f^{(1)}, \ldots, f^{(n)}$ such that each string contains exactly one correct position

```
for i ∈ [n] do
    S_i^(1) = [n];
end
t = 1;
while t ≤ n do
    X ← Choose a random color for each position i uniformly from S_i^(t) for each
      i ∈ [n];
    if b(X) = 1 then
        f^(t) ← X;
        for i ∈ [n] do
            S_i^(t+1) = S_i^(t) \ {X_i};
        end
        t = t + 1;
    end
end
return f^(1), ..., f^(n)
```

**Lemma 3.2.** *Algorithm 1 will, with high probability, need at most $\mathcal{O}(n)$ many queries to identify pairwise elementwise distinct query strings $f^{(1)}, \ldots, f^{(n)}$ of length $n$ such that $b_c(f^{(i)}) = 1$ for all $i \in [n]$.*

**Proof.** The finding of these strings is done by random queries. Let $S^{(1)} = [n]^n$ start out to be the entire space of possible queries. Sample uniform random queries $X$ from $S^{(1)}$ until one of them gives $b(X) = 1$. Set this query to be $f^{(1)}$. Now set aside all colours at the corresponding positions and keep querying. That is, $S^{(2)} = [n]\backslash\{f_1^{(1)}\} \times \ldots \times [n]\backslash\{f_n^{(1)}\}$ and sample again random queries $X$ from $S^{(2)}$ until one of them gives $b(X) = 1$. In this way set aside $f^{(i)}$ which was received by querying randomly from $S^{(i)} = [n]\backslash\{f_1^{(1)}, \ldots, f_1^{(i-1)}\} \times \ldots \times [n]\backslash\{f_n^{(1)}, \ldots, f_n^{(i-1)}\}$.

We analyse how many queries this takes. The set $S^{(i)}$ has $n - i + 1$ many possible colours at every position and also $n - i + 1$ many positions at which there is still a correct colour available. So for each position where there still is an available correct colour, there is a chance of $1/(n - i + 1)$ that this is guessed correctly in $X$, independently of every other position. So the probability that $b(X) = 1$ for a random query is

$$\mathbb{P}[b(X) = 1] = (n - i + 1) \cdot \frac{1}{n - i + 1} \left(1 - \frac{1}{n - i + 1}\right)^{n-i} \geq e^{-1}. \tag{2}$$

Let $Y_t$ be the number of queries it takes to find the $t$th string to set aside. Then $Y_t$ is geometrically distributed and the total time is the sum of all $Y_t$, $t \in [n]$, which is an independent sum of geometrically distributed random variables with success probabilities as in (2), so expected at most $e^{-1}$. Applying the Chebychev inequality gives us that, w.h.p. we can find $f^{(1)}$ to $f^{(n)}$ with $\mathcal{O}(n)$ many queries. □

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

### 3.3 Reduction to signed permutation Mastermind

The previous subsections set the stage for the following lemma, which shows the dependency between the signed permutation Mastermind and the black-peg Mastermind Problem. With only $\mathcal{O}(n)$ additional queries that result from Lemma 3.2 it is possible to solve black-peg Mastermind assuming we can solve signed permutation Mastermind with the same number of queries up to a constant factor. We denote by spmm($n$) the minimum number of guesses needed by any deterministic strategy to solve signed permutation Mastermind.

**Lemma 3.3.** *Black-peg Mastermind with $n$ colours and slots can be solved in $\mathcal{O}(n + spmm(n))$ guesses with high probability.*

**Proof.** First apply the algorithms from Lemmas 3.1 and 3.2 to obtain the corresponding queries $z$ and $f^{(1)}, \dots, f^{(n)}$. Given this, run any optimal solution to signed permutation Mastermind. Whenever this solution wants to perform a query $q \in \{-n, \dots, n\}^n$, we construct queries $q^+, q^- \in [n]^n$ according to

$$q_i^+ = \begin{cases} f_i^{(q_i)} & \text{if } q_i > 0 \\ z_i & \text{otherwise,} \end{cases}$$

and

$$q_i^- = \begin{cases} f_i^{(-q_i)} & \text{if } q_i < 0 \\ z_i & \text{otherwise.} \end{cases}$$

We consequently query $b_c(q^+)$ and $b_c(q^-)$ and return the value $b_c(q^+) - b_c(q^-)$ as the answer to query $q$.

We want to show that the answers given by this procedure are equal to $\hat{b}_{c'}(q)$ for some permutation $c'$ of $[n]$ not depending on $q$. Let $\varphi : [n]^n \to [n]^n$ be the map given by $\varphi(x)_i = f_i^{(x_i)}$. As $f_i^{(1)}, \dots, f_i^{(n)}$ are all distinct values between 1 and $n$, by construction, it follows that $\varphi$ acts as a bijection on each position of the input. In fact, we claim that

$$\hat{b}_{\varphi^{-1}(c)}(q) = b_c(q^+) - b_c(q^-) \qquad \forall q \in \{-n, \dots, n\}^n. \tag{3}$$

In other words, any signed query $q$ made in this setting will be answered as if the codeword is $c' := \varphi^{-1}(c)$. Observe that (3) implies that $c'$ is a permutation as $\hat{b}_{c'}(i \dots i)$ is, by the definition of $\hat{b}_{c'}(\cdot)$, equal to the number of occurrences of colour $i$ in $c'$, and by definitions of $b_c(\cdot)$, $q^+$ and $q-$ equal to $b_c(f^{(i)}) - b_c(z) = 1 - 0 = 1$, so each colour appears exactly once in $c'$.

To see that (3) holds, let $q \in \{-n, \dots, n\}^n$ let $q^+, q^-$ be as above, and consider the contributions to $\hat{b}_{c'}(q)$ and $b_c(q^+) - b_c(q^-)$ respectively from the $i$th position. If $q_i = 0$, then $q_i^+ = q_i^- = z_i$ where, by choice of $z$, $c_i \neq z_i$, so the contribution to both expressions is 0. If $q_i > 0$, then the contribution from the $i$th position to $\hat{b}_{c'}(q)$ is 1 if $c'_i = \varphi^{-1}(c)_i = q_i$, and 0 otherwise. Similarly, the contribution to $b_c(q^+) - b_c(q^-)$ is 1 if $c_i = q_i^+ = f_i^{(q_i)} = \varphi(q)_i$, and 0 otherwise. One immediately checks because $\varphi$ is a bijection at any position that the two conditions are equivalent. This works analogously if $q_i < 0$.

As the answers given to the signed permutation Mastermind solution are consistent with $\hat{b}_{c'}(q)$, the solution will terminate by outputting the string $c'$. We can consequently compute the codeword $c$ to the original game according to $c = \varphi(c')$.

With high probability, this process uses only $\mathcal{O}(n)$ queries to obtain $z, f^{(1)}, \dots, f^{(n)}$. Additionally, it uses two times the number of queries used by the signed permutation Mastermind strategy in order to solve the corresponding signed permutation Mastermind instance, which then obtains the codeword for the original game. □

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

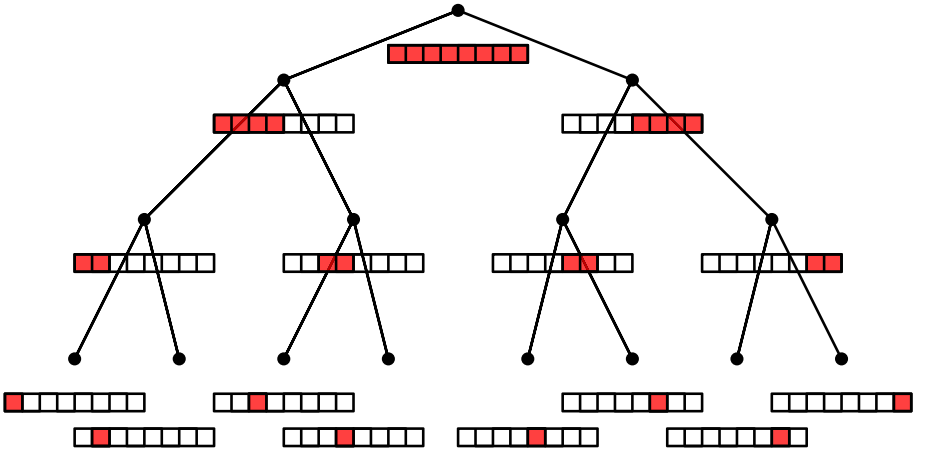

**Figure 1.** Information Tree for $n = 8$. Every node corresponds to an interval, here marked in red.

## 4. Proof of Theorem 1.1

With the reduction from the previous section at hand, we may assume that each colour appears in the hidden codeword exactly once. It remains to show that we can determine the position of every colour by using $\mathcal{O}(n)$ signed queries. Before presenting our algorithm, we will first present a token sliding game that will be used to housekeep, at any point while playing signed permutation Mastermind, the information we currently have for each colour.

**Definition 4.1.** *Given an instance of signed permutation Mastermind, we define the corresponding information tree $T$ as a rooted complete balanced binary tree of depth $\lceil \log_2 n \rceil$. We denote by $n_T$ the number of leaves of the tree. In other words, $n_T$ is the smallest power of two bigger than or equal to $n$. For each vertex in $T$, we associate a (sub-)interval of $[n_T]$ as follows. We order the vertices at depth $d$ in the canonical way from left to right and associate the $j$th such vertex with the interval $[(n_T/2^d)(j-1)+1, (n_T/2^d)\,j]$.*

Note that if $n$ is not a power of two, some vertices will be associated with intervals that go outside $[n]$. An example of an information tree for $n = 8$ is illustrated in Figure 1.

We introduce handy notation for the complete binary tree $T$. The root of $T$ is denoted by $r$. For any vertex, we denote by $v_L$ and $v_R$ its left and right child respectively if they exist and if we descend multiple vertices we write $v_{LR}$ for $(v_L)_R$. Further $T_L$ denotes the induced subtree rooted at $r_L$, similarly $T_\star$ is the the induced subtree rooted at $r_\star$ for $\star$ being any combination of $R$ and $L$ such that $r_\star$ exists.

For any instance of signed permutation Mastermind, we perform the following *token game* on the information tree as follows. We initially place $n$ coloured tokens at the root $r$, one for each possible colour in our Mastermind instance. At any point, we may take a token at position $v$ and slide it to either $v_L$ or $v_R$, if we can prove that the position of its colour in the hidden codeword lies in the corresponding sub-interval. See Figure 2 for an example.

**Observation 4.2.** *When all tokens are positioned on leaves of $T$, we know the complete codeword.* □

The simplest way to move a token is by performing a query that equals that colour on, say, the left half of its current position, and zero everywhere else. We call this step querying a token.

**Definition 4.3.** *For a colour $f$ with its token at non-leaf node $v$, we say we query the colour $f$ if we make a query of the colour $f$ only in the left half of the interval corresponding to $v$ (zero everywhere else).*

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

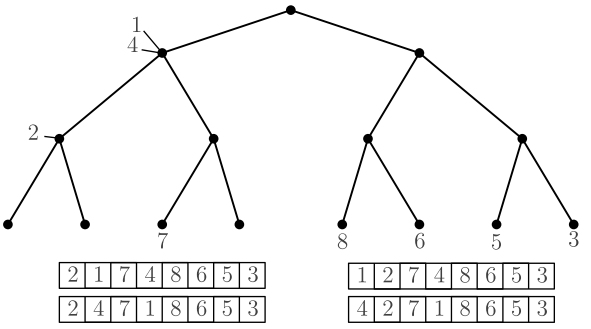


**Figure 2.** Example configuration for $n = 8$. Tokens of colours 1 through 8 are at different positions in the information tree and there are four remaining possibilities for the codeword.

**Algorithm 2:** `Preprocess`$(T)$

**Input:** Tree $T$
**Output:** Preprocessed tree T
**if** $n_T \leq 2$ **then**
  Use 1 query to move all tokens to leafs of $T$;
  **return** T;
**end**
**for each** token $t$ at $r$ **do**
  Query $t$ and slide token to either $r_L$ or $r_R$;
**end**
**for each** token $t$ at $r_L$ **do**
  Query $t$ and slide token to either $r_{LL}$ or $r_{LR}$;
**end**
Run `Preprocess` $(T_{LL})$;
Run `Preprocess` $(T_{LR})$;
**return** T;

We note that any query of this form can only give 0 or 1 as output. We will refer to any such queries as *zero-one queries*.

### 4.1 Solving signed permutation Mastermind

We are now ready to present our main strategy to solve signed permutation Mastermind by constructing a sequence of entropy dense queries that allows us to slide all tokens on $T$ from the root to their respective leaves. This will be done in two steps, which we call `Preprocess(T)` and `Solve(T)`.

As intervals corresponding to vertices close to the root of $T$ are so large, there is initially not much room to include many colours in the same query. Thus the idea of the preprocessing step is to perform $\mathcal{O}(n)$ *zero-one queries*, in order to move some of the tokens down the left side of the tree.

`Preprocess(T)`, see Algorithm 2, takes the tree, queries (Definition 4.3) all colours whose tokens are at the root $r$ and slides the tokens accordingly. Then repeats for the left child of the

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

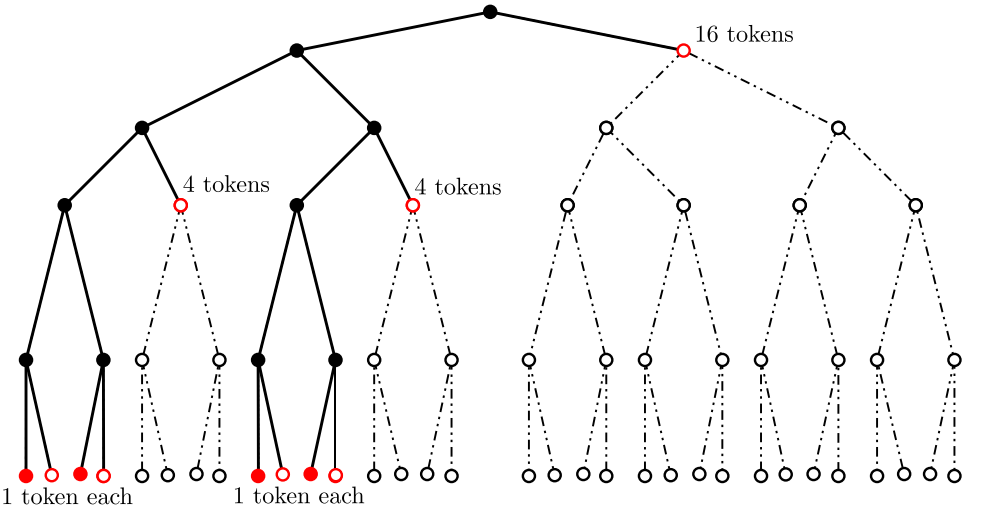


**Figure 3.** Tree preprocessed for $n = 32$, all black vertices have been emptied, tokens are at red vertices.

root $r_L$. Then we recursively apply the algorithm to the subtrees of the left two grandchildren of the root $T_{LL}$ and $T_{LR}$. If the tree has a depth of 2 or less, we skip all the steps on vertices that do not exist.

**Proposition 4.4.** *The Algorithm 2* `Preprocess(T)` *requires at most* $3n_T$ *zero-one queries and runs in polynomial time.*

**Proof.** Clearly, if the depth of the tree is $\leq 2$ this holds, as we make only a single zero-one query. Then the rest follows by induction, if we analyse the number of queries needed we see, at the root $r$ we need to query at most $n_T$ tokens, and at the left child $r_L$ we query at most $n_T/2$. In the left grandchildren, we recurse. So if $a(n_T)$ is the total number of queries we need for `Preprocess(T)` for a tree with $n_T$ leafs, then it holds that

$$a(n_T) \leq n_T + n_T/2 + a(n_{T_{LL}}) + a(n_{T_{LR}}) = n_T + n_T/2 + a(n_T/4) + a(n_T/4)$$

From which follows that $a(n_T) \leq 3n_T$. Since we only query tokens all queries we do are zero-one queries and this can be done in polynomial time concluding the proof. □

The result of running `Preprocess(T)` is illustrated in Figure 3. All tokens have either been moved to leaves, or to a vertex of the form $r_R, r_{*R}, r_{**R}, \ldots$ where each '$*$' denotes either $LL$ or $LR$. This we call a preprocessed tree. Another way of viewing this is that a tree $T$ is preprocessed if there are no tokens at $r$ or $r_L$, and the subtrees $T_{LL}$ and $T_{LR}$ are preprocessed.

Once $T$ is preprocessed we can run the second algorithm `Solve(T)`, see Algorithm 3. This is constructed recursively as follows. First note that when $n_T = 1$ or 2, then the preprocessing has already put all tokens at leaves, so the problem is already solved. Now suppose we already know how to run `Solve(T')` for all preprocessed trees $T'$ with $n_{T'} < n$, and let $T$ be a preprocessed tree with $n_T = n$. Observe that preprocessing $T$ means that $T_{LL}$ and $T_{LR}$ are both preprocessed. Hence, we already know procedures `Solve(`$T_{LL}$`)` and `Solve(`$T_{LR}$`)` that move all tokens in the respective subtrees to leaves by making a sequence of queries whose support are restricted to the first and second quarter of the available positions respectively. Similarly, as the preprocessing has determined all colours that belong in the right half of the codeword, we know how to move all tokens in $T_R$ to leaves by first performing `Preprocess(`$T_R$`)` and then `Solve(`$T_R$`)`, both of which make queries whose support are restricted to the second half of the available positions.

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

**Algorithm 3:** `Solve`($T$)

**Input:** Preprocessed Tree $T$
**Output:** Tree $T$ where all tokens have been queried down to the leaves
**if** $n_T \leq 2$ **then**
  **return** T;
**end**
Run the algorithms `Solve` ($T_{LL}$), `Solve` ($T_{LR}$) and `Preprocess` ($T_R$) in parallel, and note every time they try to make a query
**while** at least one algorithm still has queries **do**
  Get the next queries $q^{(1)}$, $q^{(2)}$, and $s$ requested by the algorithms;
  Compute the answers using two queries, as in Lemma 4.5, and return to the respective algorithm;
**end**
`Solve` ($T_R$);
**return** T;

In order to achieve the information-theoretic speedup of queries, the idea is to perform the queries of `Solve`($T_{LL}$), `Solve`($T_{LR}$) and `Preprocess`($T_R$) in parallel. This can be thought of as follows. Codebreaker runs the respective algorithms until each of them attempts to make a query. Instead of actually asking codemaker about these queries, codebreaker simply notes the query and pauses the respective algorithm. This continues until all three have proposed queries. Suppose the queries given are $q^{(1)}, q^{(2)}$ and $s$ respectively. Codebreaker combines these into two queries, determined by Lemma 4.5 explained later, that are asked to codemaker, and given the answers, codebreaker computes $\hat{b}(q^{(1)})$, $\hat{b}(q^{(2)})$, and $\hat{b}(s)$ that are then presented to the respective algorithms *as if* they were given from codemaker as answers to the respective queries, after which the algorithms are allowed to continue. This is repeated until all three algorithms have terminated. (In case one of the algorithms terminates while another still makes queries, we can simply treat the terminated algorithm as if it is making the all zeros query until the other ones finish.) Finally, this leaves $T_{LL}$ and $T_{LR}$ solved and $T_R$ preprocessed. Hence we can solve $T$ by running `Solve`($T_R$) (without any parallelism).

In order to combine queries as mentioned above, we employ an idea similar to the Cantor-Mills construction (1) for coin-weighing.

**Lemma 4.5.** *Define the support of a query $q$ as the set of indices $i \in [n]$ such that $q_i \neq 0$. For any three queries $q^{(1)}, q^{(2)}$, and $s$ with disjoint supports and such that $s$ is a zero-one query, we can determine $\hat{b}(q^{(1)})$, $\hat{b}(q^{(2)})$, and $\hat{b}(s)$ by making only two queries.*

**Proof.** We query $w^{(1)} = q^{(1)} + q^{(2)} + s$ and $w^{(2)} = q^{(1)} - q^{(2)}$, where $+$ and $-$ denote element-wise addition subtraction respectively. Then we can retrieve the answers from just the information of $\hat{b}(w^{(1)})$ and $\hat{b}(w^{(2)})$. If we combine the queries, $\hat{b}(w^{(2)}) + \hat{b}(w^{(2)}) = 2\hat{b}(q^{(1)}) + \hat{b}(s)$. So we can retrieve $\hat{b}(s) = \hat{b}(w^{(2)}) + \hat{b}(w^{(2)}) \mod 2$. And then also the queries, $\hat{b}(q^{(2)}) = (\hat{b}(w^{(1)}) + \hat{b}(w^{(2)}) - \hat{b}(s))/2$ and $\hat{b}(q^{(2)}) = (\hat{b}(w^{(1)}) - \hat{b}(w^{(2)}) - \hat{b}(s))/2$ are recoverable. □

**Proposition 4.6.** *Calling Algorithm 3, `Solve()`, for a preprocessed tree will move all the tokens to leaves. Moreover, the algorithm will use at most $6n_T$ queries and runs in polynomial time.*

**Proof.** The first statement follows by induction. If $n_T \leq 2$ a preprocessed tree already has all its tokens at leaves, nothing needs to be done. The induction step follows by the correctness of Lemma 4.5.

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

For the runtime analysis, we also use induction. If the depth $n_T \leq 2$ then clearly the statement holds. If $n_T > 2$ the procedure first runs `Solve(`$T_{LL}$`)`, `Solve(`$T_{LR}$`)` and `Preprocess(`$T_R$`)` in parallel. In each iteration of the main loop, we resolve one query from each of the subprocesses by making two actual queries. This continues until all three processes have terminated, meaning that the number of iterations is the maximum of the number of queries made by the respective subprocesses. After which, we run `Solve(`$T_R$`)`. In total, we get the following recursion where $c(n_T)$ is the total number of queries we must make during `Solve(`$T$`)` and $a(n_T)$ the total number of queries that `Preprocess(`$T$`)` must make in a tree with $n_T$ leaves.

$$\begin{aligned} c(n_T) &\leq 2 \cdot \max\left(c(n_{T_{LL}}), c(n_{T_{LR}}), a(n_{T_R})\right) + c(n_{T_R}) \\ &= 2 \cdot \max\left(c(n_T/4), a(n_T/2)\right) + c(n_T/2) \end{aligned}$$

By Proposition 4.4 we know that $a(n_T/2) \leq 3n_T/2$ so by induction we get that $c(n_T) \leq 6n_T$. All the operations as well as the computing and decoding of the combined queries in Lemma 4.5 are clearly in polynomial time so this concludes the proof. □

Now we have all the tools at hand to prove our main result.

**Proof of Theorem** 1.1. We have Lemma 3.3 which reduces the problem of solving black-peg Mastermind to solving signed permutation Mastermind. For the new instance of signed permutation Mastermind that results from this, we consider the information tree. We move the tokens of this tree to the leaves by first running the algorithm `Preprocess(`$T$`)` and then applying the algorithm `Solve(`$T$`)` on the preprocessed tree. By Propositions 4.4 and 4.6, this takes at most $9n_T$ signed queries and is done in polynomial time. By Observation 4.2 we have found the hidden codeword of the signed permutation Mastermind, and therefore also the hidden codeword of the black-peg Mastermind game. The transformation can be done in polynomial time and by Lemma 3.3 we need at most $\mathcal{O}(n + 9n_T) = \mathcal{O}(n)$ queries to solve black-peg Mastermind. □

## 5. Playing Mastermind with arbitrarily many colours

We briefly make some remarks on other ranges of $k$ and $n$ for Mastermind. By combining Theorem 1.1 with results of Chvátal [3] and Doerr, Doerr, Spöhel, and Thomas [4], we determine up to a constant factor the smallest expected number of queries needed to solve black-peg and black-white-peg Mastermind for any $n$ and $k$.

For any $n$ and $k$, let $\mathrm{bmm}(n, k)$ denote the minimum (over all strategies) worst-case (over all codewords) expected number of guesses to solve Mastermind if only black peg information can be used. Similarly, denote $\mathrm{bwmm}(n, k)$ the smallest expected number of queries needed if both black and white peg information can be used. The following relation between black-peg and black-white-peg Mastermind was shown by Doerr, Doerr, Spöhel, and Thomas.

**Theorem 5.1** (Theorem 4, [4]). *For all* $k \geq n \geq 1$,

$$bwmm(n, k) = \Theta\left(bmm(n, n) + k/n\right).$$

**Proof of Theorem** 1.2. For small $k$, say $k \leq \sqrt{n}$, the result follows by the results of Chvátal [3]. Moreover, for $k \geq n$ the white peg statement follows directly by combining Theorems 1.1 and 5.1. Thus it remains to consider the case of $\sqrt{n} \leq k \leq n$, and the case of $k \geq n$ for black-peg Mastermind.

For $\sqrt{n} \leq k \leq n$, the leading terms in both bounds in Theorem 1.2 are of order $n$, which matches the entropy lower bound. On the other hand, using Lemma 3.1 we can find a query such that $b(z) = 0$ in $\mathcal{O}(n)$ queries. Having found this, we simply follow the same strategy as for $n$ colour black-peg Mastermind by replacing any colour $> k$ in a query by the corresponding entry of $z$. Thus finishing in $\mathcal{O}(n)$ queries.

https://doi.org/10.1017/S0963548323000366 Published online by Cambridge University Press

Finally, for black-peg Mastermind with $k \geq n$, the leading term in Theorem 1.2 is of order $k$. This can be attained by using $k$ guesses to determine which colours appear in the codeword and then reduce to the case of $n$ colours. On the other hand, $\Omega(k)$ is clearly necessary as this is the expected number of queries needed to guess the correct colour in a single position, provided the codeword is chosen uniformly at random. □

## References


[1] Cantor, D. G. and Mills, W. (1966) Determination of a subset from certain combinatorial properties. *Can. J. Math.* **18** 42–48.
[2] Chen, Z., Cunha, C. and Homer, S. (1996) Finding a hidden code by asking questions. In *Computing and Combinatorics* (J.-Y. Cai and C. K. Wong, eds), Springer, pp. 50–55.
[3] Chvátal, V. (1983) Mastermind. *Combinatorica* **3**(3-4) 325–329.
[4] Doerr, B., Doerr, C., Spöhel, R. and Thomas, H. (2016) Playing Mastermind with many colors. *J. ACM* **63**(5) ArticleNo:42.
[5] Erdős, P. and Rényi, A. (1963) On two problems of information theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.* **8** 229–243.
[6] Focardi, R. and Luccio, F. L. (2010) Cracking bank pins by playing Mastermind. In *Fun with Algorithms* (P. Boldi and L. Gargano, eds), Springer, pp. 202–213.
[7] Goodrich, M. (2009) The Mastermind attack on genomic data. In *2009 30th IEEE Symposium on Security and Privacy*, pp. 204–218.
[8] Goodrich, M. (2009) On the algorithmic complexity of the Mastermind game with black-peg results. *Inform. Process. Lett.* **109**(13) 675–678.
[9] Jäger, G. and Peczarski, M. (2011) The number of pessimistic guesses in Generalized Black-peg Mastermind. *Inform. Process. Lett.* **111**(19) 933–940.
[10] Knuth, D. E. (1976/77) The computer as master mind. *J. Recreational Math.* **9**(1) 1–6.
[11] Koyama, K. (1993) An optimal Mastermind strategy. *J. Recreational Math.* **25** 251–256.
[12] Lindström, B. (1964) On a combinatory detection problem I. *I. Magyar Tud. Akad. Mat. Kutató Int. Közl.* **9** 195–207.
[13] Lindström, B. (1965) On a combinatorial problem in number theory. *Can. Math. Bull.* **8**(4) 477–490.
[14] Shapiro, H. S. and Fine, N. J. (1960) E1399. *Am. Math. Mon.* **67**(7) 697–698.
[15] Stuckman, J. and Zhang, G.-Q. (2006) Mastermind is NP-complete. *INFOCOMP J. Comput. Sci.* **5** 25–28.
[16] Viglietta, G. (2012) Hardness of Mastermind. In *Proceedings of the 6th International Conference on Fun with Algorithms, FUN'12*, Springer, pp. 368–378.


**Cite this article:** Martinsson A and Su P (2024). Mastermind with a linear number of queries. *Combinatorics, Probability and Computing* **33**, 143–156. https://doi.org/10.1017/S0963548323000366