

Shuffling Cards When You Are of Very Little Brain: Low Memory Generation of Permutations*

Boaz Menuhin[†] Moni Naor[‡]

June 13, 2025

Abstract

How can we generate a permutation of the numbers 1 through n so that it is hard to guess the next element given the history so far? The twist is that the generator of the permutation (the “Dealer”) has limited memory, while the “Guesser” has unlimited memory. With unbounded memory (actually n bits suffice), the Dealer can generate a truly random permutation where $\ln n$ is the expected number of correct guesses.

Our main results establish tight bounds for the relationship between the guessing probability and the memory m required to generate the permutation. We suggest a method for an m -bit Dealer that operates in constant time per turn, and any Guesser can pick correctly only $O(n/m + \log m)$ cards in expectation. The method is fully transparent, requiring no hidden information from the Dealer (i.e., it is “open book” or “whitebox”).

We show that this bound is the best possible, even with secret memory. Specifically, for any m -bit Dealer, there is a (computationally powerful) guesser that achieves $\Omega(n/m + \log m)$ correct guesses in expectation. We point out that the assumption that the Guesser is computationally powerful is necessary: under cryptographic assumptions, there exists a low-memory Dealer that can fool any computationally bounded guesser.

We also give an $O(n)$ bit memory Dealer that generates perfectly random permutations and operates in constant time per turn.

1 Introduction

The question of generating random permutations has received significant attention, dating back to at least the work of Fisher and Yates in the 1930s (see Section 3.4.2 in Knuth’s [Knu98]). In this work we concentrate on generating permutations using a small amount of memory while ensuring that it is difficult to predict the next value based on the previously generated values.

The following game was considered by Menuhin and Naor [MN22b]: A card guessing game is played between two players, “Guesser” and “Dealer”. At the beginning of the game, the Dealer holds a deck of n distinct cards (labeled $1, \dots, n$). For n turns, the Dealer draws a card from the deck, the Guesser guesses which card was drawn, and then the card is discarded from the deck. The Guesser receives a point for each correctly guessed card.

*Research supported in part by grants from the Israel Science Foundation (no. 2686/20), by the Simons Foundation Collaboration on the Theory of Algorithmic Fairness and by the Israeli Council for Higher Education (CHE) via the Weizmann Data Science Research Center.

[†]Some of this work was accomplished when the author was at the Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. Email: boaz.menuhin@gmail.com.

[‡]Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. Incumbent of the Judith Kleeman Professorial Chair. Email: moni.naor@weizmann.ac.il.

Menuhin and Naor considered the asymmetric case where the Dealer remembers everything, and the Guesser has limited memory and derived tight bounds in this case. In this work we consider what happens when the shoe is on the other foot and the Dealer is limited in space, while the Guesser has plenty of memory. Say that the Dealer has only m bits of memory, and wants to pick a permutation that is unpredictable by any Guesser, that has no limitation on the number of bits it can store or on its computational power? We know that when both players have plenty of memory, then the expected number of correct guesses is $H_n \approx \ln n$.

In our model, the Dealer has a limited number of long-lived random bits, in the sense that storing them is part of its memory, as well as “on-the-fly” randomness that is available at every round. Note that with $O(n \log n)$ long-lived bits, the Dealer can generate a truly random permutation. As Knuth showed (see Algorithm P in Section 3.4.2 in [Knu98]), we can drop the longevity requirement by maintaining an ordered list of available cards, pick one at random and discard. This algorithm, which is attributed to Fisher and Yates requires $O(n \log n)$ bits of memory. It is also possible to get a *perfectly random permutation* with only n bits of memory, using a bitmap Dealer (see below), where what the generator maintains is the subset of cards that was used so far.

If the Dealer has only m bits of memory, then if we partition the deck into n/m parts and for each one generate a perfect permutation using the same storage of m bits (that is n/m perfect permutations on sets of size m). For each such small permutation, the expected number of guesses a perfect Guesser makes is $\ln m$, and thus $(n/m) \ln m$ altogether.

The natural question is whether this algorithm is the best possible in terms of its unpredictability or whether it can be improved. After some pondering the reader may conclude that the best strategy such a Dealer can have is pick the next card from a set of m cards at random (in the above description, towards the end of each epoch the number of possibilities diminishes), and the question is how to ensure that there is such a set available for as many rounds as possible especially given the tight memory requirements. This is indeed true, and our paper proves it and, furthermore, shows that this is the best possible.

Our Contributions: Our main results are tight bounds for the relationship between the guessing probability and the memory required to generate the permutation.

For the case where the Dealer is frugal and is willing to allocate only $m = o(n)$ bits of memory, we found a method that makes any Guesser pick correctly only $O(n/m + \log m)$ cards in expectation. The method does not require any secrecy from the Dealer, i.e. it is “open book” in the language of Magen and Naor [MN22a] or “whitebox” in terms of Ajtai et al. [ABJ⁺22]). Furthermore, this method is computationally efficient, where each card draw takes a constant time in the worst case and overall $O(n)$ time.

On the other hand, we show that this bound is the best possible, even for Dealers with secret memory. For any m bit Dealer there is a (computationally powerful) Guesser, which we call the *myopic optimizing Guesser*, that makes $\Omega(n/m + \log m)$ correct guesses in expectation.

Note that the Guesser from the lower bound is *not* efficient, and this is not a coincidence: we show that assuming that the Guesser is polynomial time and that one-way functions exist, then it is possible to generate a permutation that is indistinguishable from random (and hence any computationally efficient Dealer can guess at most $\ln n$ cards correctly in expectation) using an amount of memory needed to store a key to a pseudo-random function (PRF); See Section 6.1.

For the high memory regime, we introduce a Dealer that produces a perfectly random permutation, that is, one against which any Guesser scores at most $\ln n$ points in expectation, runs in constant time, and requires a linear amount of bits. This ultimately closes the gap between the Bitmap Dealer that uses n bits of memory and runs in super-constant time (even amortized),

and Knuth’s implementation of Fisher and Yates’s algorithm that runs in constant time and requires $\Omega(n \log n)$ bits of memory. To the best of the authors’ knowledge, this is the first suggestion to achieve this.

1.1 Related Work

Generating Permutations and Cards Shuffling Due to its fundamental nature, the problem of generating a random permutation has been widely treated and analyzed from different angles and has a relatively long history. The computational aspect of generating a permutation got a dedicated treatment in Knuth’s [Knu98], where he analyzed a method by Fisher and Yates from the 1930s. In an iconic paper among many, Bayer and Diaconis [BD92] analyzed the amount of Rifle-Shuffles required to achieve a close to uniform distribution of permutations. Shuffles are studied cryptographically, as in Morris and Rogaway’s [MR14] construction (see Section 6.1). Permutations are essential, and thus, so is the study of the computational resources required to produce them.

The Missing Item Problem: In the Missing Item Finding problem (MIF), a stream of r arbitrary elements $(e_1, \dots, e_r) \in [n]^r$ is presented to an observer, which has to come up with some element $x \in [n]$, that has not appeared yet in the stream so far. Stoekl [Sto23] and Magen [Mag24] thoroughly analyzed upper and lower bounds for various computational variants of the MIF problem, and in particular the one in which the generator’s internal are “open-book”. In each turn, our Dealer faces a similar challenge, where it must produce a never-seen-before element.

Mirror Games: Mirror games were invented by Garg and Schnieder [GS19]. In this game, Alice and Bob take turns (with Alice playing first) in declaring numbers from the set $\{1, 2, \dots, 2n\}$. If a player picks a number that was previously played, that player loses and the other player wins. If all numbers are declared without repetition, the result is a draw. Bob has a simple mirror strategy that assures he won’t lose and requires no memory. On the other hand, Garg and Schnieder showed that every deterministic Alice needs memory of size linear in n in order to secure a draw. Regarding probabilistic strategies, with sufficiently many secret bits or using cryptographic assumptions, there is a strategy for Alice that enables her to achieve a draw in the game w.h.p. using only polylog bits of memory [GS19, Fei19, MN22b]. The requirement for secret bits is crucial [MN22a].

Dynamically Changing Distributions The task of sampling a random variable from a given discrete distribution has been ultimately solved by the famous Alias method. The case where the distribution changes across time has been studied by Rajasekaran and Ross [RR93], who suggested several rejection-based algorithms that work well for some families of dynamic distributions. Their work followed a fruitful line of work for handling more classes of dynamically changing distributions, where Matias, Vitter and Ni [MVN93] suggested a general solution that works for any dynamic distribution, albeit one that runs at a super constant time. Inspired by their work, Hagerup, Mehlhorn and Munro [HMM93] introduced two algorithms for different classes of dynamic distributions. While many of these works reached expected constant time (or near it), none reached worst case constant time, and they have neglected the space aspect; thus, to the best of our knowledge, none of our results immediately follow from this branch.

Sampling marbles from urns, with and without replacements, is a dynamically changing distributions of special interest in simulation of interactions between agents. Berenbrink, Hammer, Kaaser, Meyer, Penschuck and Tran [BHK⁺20] presented a memory efficient rejection-based algorithm for this task, especially designed for bulk operations. One of our machineries addresses this

distribution, as well as a broader family of distributions, and runs faster. However, it also consumes much more memory, and is not intended for bulk operation.

Generating and Storing Permutations The randomness of a Guesser-agnostic Dealer dictates the drawing order of the cards and, thus, describes a permutation. In the other direction, given a succinct permutation data structure that can be initialized with random bits, we can generate a random drawing order.

Storing a permutation succinctly aims at representing a permutation using space as close to the information-theoretic lower bound as possible while enabling a quick (adaptive) evaluation of arbitrary elements. Such schemes are significant for Proof-of-Space schemes [Bon23] and database optimization [MRRS12].

Circuit Depth of Generating Permutations: It is possible to generate random permutations in parallel. In particular, it can be done in the class AC^0 , polynomial size, constant depth unbounded fan-in AND/OR gates, but not in the class NC^0 , circuits where every output bit depends on a constant number of input bits [MV91, Hag91, Vio12, Vio20].

1.2 Results and Structure

As mentioned, our primary results are smooth bounds relating the amount of memory that the Dealer poses and the predictability of the permutations they can produce.

Low Memory Dealer: With m bits of memory, we aim to have $\Omega(m)$ cards to choose from in every turn. Our Dealer is based on the idea that it is easier to track the progress of many small mini-decks than the deck in its entirety. After splitting the deck, in each turn, the Dealer chooses a mini-deck to draw a card from and draws the top card of that mini-deck. This already results in a huge reduction in the required memory. If the mini-decks are close to each other, then we can store their distances from some central point and thus achieve additional savings. So we want the mini-decks to progress somewhat evenly without giving too much advantage to the Guesser; this is exactly the kind of problem addressed by allocation schemes in the balls-into-bins model (See Section 2.3 and a survey by Wieder [Wie17]).

Our Dealer utilizes the Adaptive-Threshold allocation scheme introduced by Berenbrink, Khodamoradi, Sauerwald, and Stauffer [BKSS13], which can be roughly described as “sample mini-decks until finding one that is not much ahead of the average”. As we will show (Lemma 3.2), this scheme can be implemented by a Dealer with m bits of memory can track $\Theta(m)$ mini-decks. But how well does the Dealer play? It turns out that it is pretty good in the sense that there are $\Theta(m)$ cards that can be played at every round. Therefore, our Dealer scores $O(n/m + \ln m)$ points in expectation against any Guesser (Section 3.3.) Furthermore, this Dealer is efficient and runs in constant time in expectation.

We prove that the Dealer has these properties by walking the path paved by Berenbrink et al. in [BKSS13]. As in other proofs on allocation schemes, our main tool is a potential function. We relate the value of the potential function to the set of mini-decks that we can draw from and show that we expect to have sufficiently many such mini-decks. This would imply that the draws are sufficiently unpredictable and that each turn runs in constant time in expectation. We present our low memory Dealer in Section 3, and analyze its behavior and implementation for the first $n - m$ turns. We treat the last m turns in Section 4.3, where we also improve the Dealer’s runtime.

The analysis of the first $n - m$ turns (Section 3), and that of the last m turns (Section 4.3) are of different nature and rely on different building blocks. The separation to two sections enables us to treat each mechanism in a self-contained and focused manner.

Constant-Time Sampling from a Dynamic Subset: Given a fixed domain B , we construct a data structure for maintaining a dynamic subset $A \subseteq B$. That data structure supports membership queries, addition and removal of a single element, as well as sampling of a member uniformly at random, all running in worst-case constant time, and using $O(|B|)$ bits of memory (see Theorem 4.1).

We use this data structure to complete the analysis of the low memory Dealer from Section 3 and improve its runtime. Namely, we show that the low-memory Dealer runs in worst-case constant time per turn, and that any Guesser scores $O(n/m + \ln m)$ points in expectation against our Dealer. In addition, we show that there exists a Dealer with $O(n)$ bits of memory, that runs in worst case constant time per draw, against which any Guesser is expected to score at most $\ln n$ points.

The data structure is constructed in three layers. Going bottom-up, we first split the domain B into $|B|/\log |B|$ cells, and allocate a bit for each element in the domain, to determine its membership. This already allows us to sample, update and query members of A within a specific cell. In the second layer, we group cells together by the number of elements from A that they track, that is, by their population size. This helps us to manage cells efficiently, as well to sample a random member of a given population size. The third layer handles the task of sampling a population size. Even though there are only $\log |B|$ possible outcomes, we are unaware of a dynamic distribution data structure that yields the desired result. Thus, we present one of our own, which may be of independent interest. We present this data structure in Section 4.

Lower Bound: In Section 5 we show that for any Dealer with m bits of memory, there exists a Guesser that scores $\Omega(n/m)$ correct guesses in expectation. We show this by establishing a connection between the size of the Dealer’s memory and the Dealer’s card sampling entropy. That is, we show that less memory implies more predictable draws. Our central tool is a compression argument; namely, we show an efficient, prefix-free encoding scheme that allows us to encode and recover the course of a random game. Since no prefix-free encoding can surpass the entropy, we get an upper bound on the Dealer’s entropy. We present a simple Guesser called the Myopic Optimizing Guesser, which simply guesses the most probable card in each turn. We draw a relation between the Dealer’s entropy and the success probability of the Myopic Guesser, and by doing so, we prove that our low-memory Dealer is optimal.

Connections with Cryptography: In Section 6 we discuss the (tight) relationship between bypassing the lower bound and cryptography. The Guesser from the lower bound is not computationally efficient, and there is a good reason for that: if certain computational assumptions are true, then there is a low memory Dealer that produces a random-looking permutation where no computationally efficient Guesser can get non negligibly better than $\ln n$ correct guesses in expectation. This is when the Dealer can keep a secret.

But in the open book case, where the Dealer has no secrets, the Myopic Guesser from Section 5 can still operate efficiently. We present a few research questions about the exact necessity of the existence of one-way functions to obtain our result.

2 Preliminaries

Throughout this paper we use $[n]$ to denote the set of integers $\{1, \dots, n\}$ and $[a - b]$ to denote the set of integers $\{a, a + 1, \dots, b\}$. All logs are base 2 unless explicitly stated otherwise, \ln is the natural logarithm (base e). For binary strings $s_1, s_2 \in \{0, 1\}^*$, $|s_1|$ denotes the length in bits of s_1 , and $s_1 \circ s_2$ denotes the concatenation of s_1 and s_2 . $s[i..j]$ refers to the substring of s that spans between the i th and j th bits. As a convention, we will denote random variables by bold capital letters. We use $\text{Poi}(x)$ for the Poisson distribution with expected value x .

2.1 Word RAM Generative Streaming Model

In the generative streaming model, an algorithm produces a sequence of elements, one at a time. The span of time it takes to produce an element is referred to as a *turn* or a *step*. There may be an observer responding to the elements, and the algorithm may or may not take their responses into account when producing the next elements in the sequence.

We work under the assumptions of the Word RAM model in a streaming fashion. This model is considered realistic for capturing multiple aspects of modern computers; namely, processors, operating systems, and memory. The main standard assumptions of this model are that an algorithm has a finite memory of m bits and that this memory is made of cells (or words) where each cell consists of $w \geq \log m$ bits. The algorithm can access any cell in constant time.

As for randomness, the model allows access to random bits, where the bits are produced and read “on the fly”, so one cannot access previously read bits without storing them in memory. We assume that we can ask for biased random bits, so that we can sample a random number $r \sim [x]$ for any $x \leq 2^w$ in worst-case constant time. We discuss this in more detail in Section 4.3.

In terms of computation, we assume that integer and bit-wise operations on w bits, occur in constant time, as is the case in modern processors. In this paper, we assume that the processor exposes the operations `popCount`, `bitSelect`, and `bitRank`, and that these operations run in constant time. Given a binary string s of w bits $s[1..w]$ the `popCount(s)` operation returns the number of 1-bits in the string, that is `popCount(s) = |\{i \in [w] : s_i = 1\}|`. The `bitRank(s, j)` returns the number of 1 bits in the substring $s[1..j]$, that is `bitRank(s, j) = |\{i \in [j] : s_i = 1\}|`. The `bitSelect(s, i)` operation returns the index of the i th 1-bit, or in other words, the least index j such that $s[1..j]$ has exactly i 1-bits, that is `bitSelect(s, i) = \text{argmin}_j \text{bitRank}(s, j) = i`.

CPUs implement `popCount` since the 60s [Wik25]. And clearly, `popCount(s)` is just `bitRank(s, w)`, and conversely `bitRank` can be implemented by masking the higher $w - j$ bits and calling `popCount`. If the processor does not expose any of these operations, we can preprocess a data structure that does during the initialization phase, which will use only a linear amount of bits (See Section 2 in [BB08]).

2.2 Introduction to Card Guessing

Card Guessing is a game played by two players, a Guesser and a Dealer. At the beginning of the game, the Dealer holds a deck consisting of n distinct cards (labeled $1, 2, \dots, n$ for simplicity). In each turn, the Dealer picks a card from the deck, placing it face down, and the Guesser attempts to guess the identity of that card. Once the Guesser declares its prediction, the card is revealed and discarded, and can no longer be played during this game. The Guesser receives a point for each correct guess, and attempts to maximize the number of correct guesses throughout the game, while the Dealer attempts to be as unpredictable as possible, and reduce the amount of predicted card draws.

Say that the Dealer shuffles the deck properly at the beginning of the game, and draws cards one by one. The capable Guesser who remembers previous draws can simply guess a card that still resides in the deck. When the deck contains i cards, any such guess is correct with probability $1/i$. Therefore, by linearity of expectation, the expected number of correct guesses throughout the game is ¹

$$\frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{3} + \frac{1}{2} + 1 \approx \ln(n).$$

In the setting considered in this paper, we assume that the Guesser has sufficient memory to remember all previously drawn cards, so $\ln n$ is the best that a Dealer can aim for. We study the Dealer’s predictability as a function of the computational resources it possesses, namely, the size of the Dealer’s memory, the time spent during each turn, and its access to cryptographic primitives.

A transcript of a turn t between a Guesser and a Dealer is a pair of (i) a guess $g_t \in [n]$ (made by the Guesser) and (ii) a draw of a card $d_t \in [n]$ (declared by the Dealer). The draw d_t should not have appeared before. A transcript of a game is a sequence of guesses and draws $\{(g_t, d_t)\}_{t=1}^n$, such that at turn t the Guesser guessed the card g_t while the Dealer drew the card d_t . Whenever the Guesser and the Dealer declared the same card, i.e. whenever $g_t = d_t$, a single point is attributed to the Guesser. So the total *score* of the game is the total number of turns at which a card was *predicted correctly*. We are interested in the *expected number of correct guesses* during a game.

At turn t , we say that a card is *available* for drawing if it has not been drawn yet. That is, a card is available if it belongs to the set $[n] \setminus \{d_1, \dots, d_{t-1}\}$. A crucial requirement of the game is that the Dealer must, under all circumstances, declare an available card in each turn. That is, the sequence d_1, \dots, d_n has no repetitions and thus, forms a permutation of $[n]$.

We think of the Dealer as being made up of two functions:

- **DRAW**(turn, memory_state, randomness): produces the Dealer’s next card draw. It receives the current turn, the Dealer’s memory state (m bits), and on-the-fly random bits.
- **UPDATE**(turn, memory_state, last_draw, randomness): produces the Dealer’s next memory state. It receives the current turn, the memory state at the beginning of the turn, on-the-fly random bits, the last card drawn (the result of **DRAW**) and *optionally* the Guesser’s guess.

If the Guesser can observe or deduce the Dealer’s memory state, between turns, then we say that the Dealer is “open book”. A Dealer that does not take the Guesser’s guesses into account is said to be *Guesser-agnostic*.

As a warm-up, consider a Dealer with n bits of memory that maintains a bitmap of available cards. In each turn, the Dealer samples a card to pick, and if that card is available, then the Dealer draws it. Otherwise, the Dealer keeps sampling until an available card is found. Tracking the cards using a bitmap takes n bits of memory. From the coupon collector’s problem, we know that the total number of sampling attempts is expected to be $n \ln n$; thus, it is far from constant time per draw, even amortized.

Construction 2.1 (Bitmap Dealer). *In each turn, the Dealer samples a card $c \in_R [n]$ uniformly at random. If x is available for drawing, then the Dealer declares c as its draw. Otherwise, the dealer will sample again until an available card is found.*

An algorithmic description of this Dealer is presented in Algorithm 2.1

Consider also a Dealer that implements Knuth’s algorithm [Knu98] (Section 3.4.2, Algorithm P). That Dealer explicitly stores all available cards (n cards, $\log n$ bits each), samples one of them

¹Taken from textbook on algorithms by Kleinberg and Tardos [KT06], Chapter 13, Pages 721-722.

Algorithm 2.1 Bitmap Dealer

Let $A \leftarrow 1^n$ ▷ Cards availability bitmap
for turn $t \in [n]$ **do**
 Sample $c \sim [n]$ uniformly at random
 while $A[c] = 0$ **do** ▷ Sample until an available card is found
 Sample $c \sim [n]$ uniformly at random
 Draw card c
 $A[c] \leftarrow 0$

in each turn, and pops from memory. More formally, that Dealer initiates their memory with an array A of size n (total of $n \log n$ bits), such that the j th cell contains j , i.e. $A[j] \leftarrow j$. In turn t , the Dealer samples an index i uniformly at random from $[n - t + 1]$, draws the card stored in $A[i]$, and assigns the card that resides in $A[n - t + 1]$ to cell $A[i]$. So, assuming that index i was chosen at the first turn, then the memory state switches from

$$1, 2, \dots, i - 1, i, i + 1, \dots, n - 1, n$$

to

$$1, 2, \dots, i - 1, n, i + 1, \dots, n - 1.$$

Construction 2.2 (Knuth’s Dealer). *At the beginning of the game, the Dealer initiates an array A of n cells such that, for every $i \in [n] : A[i] \leftarrow i$. At turn t , the Dealer samples an index $i \sim [n - t + 1]$ uniformly at random and draws the card that resides at cell $A[i]$. Then $A[i] \leftarrow A[n - t + 1]$.*

An algorithmic description is provided in Algorithm 2.2. And we note that this Dealer takes $O(1)$ time per draw, and $O(n \log n)$ bits of memory.

Algorithm 2.2 Knuth’s Dealer

Initiate an array of n cells such that $A[i] \leftarrow i$
for turn $t \in [n]$ **do**
 Sample index $i \sim [n - t + 1]$ uniformly at random
 Draw card $A[i]$
 $A[i] \leftarrow A[n - t + 1]$

2.3 Balls-into-Bins

The Balls-into-Bins model we consider is a stochastic process where m balls are thrown into n bins². The balls are thrown in some random manner, one after another, and we are interested in the question of *how many balls are there in the most (or least) loaded bin*, often with respect to the average load.

In the setting of interest to this work, m will be larger than n . This case is known as the “heavily loaded” case. A probabilistic algorithm for placing balls into bins is often called an *allocation process* or a *sampling scheme*. There has been much work on methods for achieving balanced allocations via simple rules of assigning the balls into the bins (e.g. the famous “two-choice paradigm”, where balls are thrown into two bins and are kept in the less loaded one). See [Wie17] for a survey on the topic.

We will actually consider two simple methods: the one choice scheme and the Adaptive-Threshold. Let $\ell_i(m)$ denote the load of bin i after placing m balls.

²In this section, we follow the common naming convention from the literature. However, in the context of our work, m will be related to the number of bins, and n to the number of balls.

Algorithm 2.3 One-choice scheme

```
 $\ell_i = 0 : \forall i \in n$   
for ball  $v \in [m]$  do  
   $i \sim [n]$  ▷ Sample a random bin  $i$   
  Assign ball  $v$  to bin  $i$   
   $\ell_i += 1$ 
```

Proposition 2.3 (Theorem 2.2 in [Wie17]). *After throwing m balls into n bins using the one-choice scheme, with probability at least $1 - 1/n$ the maximum load is $\frac{m}{n} + O\left(\sqrt{\frac{m \ln n}{n}}\right)$ when $m > n \ln n$.*

An allocation scheme that achieves a more balanced allocation is called Adaptive-Threshold and was introduced in [BKSS13]. The idea here is to make sure that no bin is too advanced. This is achieved by placing the v th ball into a bin with load at most $v/n + 1$, and by doing so, preventing bins from diverging far ahead of the average. We will study this scheme extensively in Section 3.

Algorithm 2.4 Adaptive-Threshold

```
 $\ell_i = 0 : \forall i \in n$   
for ball  $v \in [m]$  do  
  repeat  
     $i \sim [n]$  ▷ sample a bin uniformly at random  
    if  $\ell_i < \lceil \frac{v}{n} \rceil + 1$  then ▷ If the selected bin is loaded below the threshold  
      Place ball  $v$  into bin  $i$   
       $\ell_i += 1$   
  until ball is placed
```

2.4 Data Structures

2.4.1 Variable bit-length arrays

An array of n cells of k bits each can easily store an ordered set of objects of size at most k while allowing fast access and update time. If many of the stored objects require significantly less than k bits, then this scheme is wasteful. How can we dynamically adjust object lengths while maintaining efficiency?

The problem of maintaining a compact data structure for variable bit-length arrays while allowing fast look-up and update times has several solutions. Two that match our needs are those by Blandford and Blelloch [BB08], who showed a worst-case constant time compact solution for objects of known size bound, and that of PacHash [KLS23], who lifted the known size bound at the cost of having the lookup and update time constant in expectation. Both can be implemented in our computational model.

Let S_w be the set of binary objects of size at most w .

Proposition 2.4 (Theorem 3.2 in [BB08]). *An array $A[1, n]$ storing elements from S_w with m total bits can be represented using $O(n + m)$ bits while supporting lookups and updates in $O(1)$ worst-case time.*

2.5 Dynamically Changing Distributions

Consider the task of generating the outcome of a discrete random variable X that has n possible outcomes, such that X equals i with probability a_i . We can describe such a random variable as

a vector (a_1, \dots, a_n) where $\sum a_i = 1$. The famous Alias method solves this problem; it uses at most $O(n \log n)$ bits and runs in constant time.

What happens when the random variable X changes over time? We now want a data structure that, in addition to initialization and generation, also supports updating the weight of some of its values, and efficiently. There are multiple solutions to this problem, addressing different families of dynamic distributions.

If the sum of weights is not equal to 1, then we say that this is a pseudo distribution, and we would like to sample the value j with probability $a_j / \sum_{i=1}^n a_i$, where we usually consider the case where $a_j \in \mathbb{N}$. Call a pseudo-distribution polynomially-bounded, if $\sum_{i=1}^n a_i$ is at most poly(n) for some polynomial of degree at least 2. A construction by Hagerup et. al. [HMM93] utilized multiple ideas by [MVN93] and [RR93], and introduced a rejection-based algorithm to maintain and sample from polynomially bounded pseudo distributions in expected constant time.

Proposition 2.5 (Theorem 2 in [HMM93]). *Polynomially bounded pseudo-distribution on $[n]$ can be maintained with constant expected generation time, constant update time, $O(n \log n)$ bits of space, and $O(n)$ initialization time.*

2.6 Information Theory and Probability

Definition 2.6 (Entropy). *Let X be a discrete random variable that takes values from domain \mathcal{X} with probability mass function $p(x) = \Pr[X = x]$. The Binary Entropy (abbreviated Entropy) of X , denoted $H(X)$ is*

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \cdot \log p(x).$$

Lemma 2.7. *Let X be drawn according to the probability mass function p . If $H[X] \leq k$ then there exists $x \in \mathcal{X}$ such that*

$$\Pr[X = x] \geq 2^{-k}.$$

Proof. Assume the converse, then for every $x \in \mathcal{X} : p(x) < 2^{-k}$. In that case,

$$\begin{aligned} H[X] &= \sum_{x \in \mathcal{X}} p(x) \cdot \log \left(\frac{1}{p(x)} \right) \\ &> \sum_{x \in \mathcal{X}} p(x) \cdot \log \left(2^k \right) = k. \end{aligned}$$

In contradiction to the assumption. □

Proposition 2.8 (Jensen's Inequality - Theorem 2.6.2 in [CT06]). *If f is a convex function and X is a random variable,*

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X]).$$

3 Low-Memory Dealer

We present our low-memory Dealer. Recall that our goal is to have about m choices at each turn and pick one of them at random. We can think of several strategies for doing so, and the one we present is based on the “mini-deck” approach. At the beginning of the game, the Dealer splits the deck into d mini-decks of equal size (i.e. n/d cards in each mini-deck). In each turn, the Dealer chooses a mini-deck and draws the card at the top of it. So if there are d mini-decks, the Dealer will

draw one of the d top cards of the different mini-decks. The advantage of picking the top card from each deck is that we simply need to recover the number of cards drawn from it, and this determines the identity of the chosen card. In terms of guessing probability, as long as all the d mini-decks have cards, the probability of guessing is $1/d$.

Several complications arise: first, we need to record the number of cards drawn from each mini-deck. Naively, this takes $\log n$ bits. But even if we try to be a bit more sophisticated and record the difference between the expected number of cards drawn and the actual one, then this difference is going to be something like $\sqrt{n/d}$ (See Proposition 2.3). This means that it takes $\log(n/d)$ bits per mini-deck, resulting in roughly $m \approx d \cdot \log n$ total number of bits. Another issue is that some mini-decks may run empty before others, resulting in fewer than m choices pretty early on (which means better probability for the adversary in guessing the next card).

Instead, we aim to get a much more balanced number of cards drawn from each mini-deck, concentrated around the expected number of drawn cards. This way achieves both a compression of the bit-representation of the numbers of cards drawn and addresses the mini-deck depletion problem. The idea is to apply the Adaptive-Threshold paradigm (Adaptive-Threshold allocation scheme, as described in Algorithm 2.4), that is, to select at each turn a mini-deck that is below the average. The fact that we choose from a (known) subset of all available mini-decks gives some advantage to the Guesser, but not a huge one: as we will show, its probability of guessing correctly is multiplied only by a constant.

3.1 Adaptive-Threshold Dealer

At the beginning of the game, the Dealer splits the cards into d mini-decks of equal size and tracks their progress. In each turn, the Dealer selects a mini-deck and draws the card that resides at the top of this mini-deck. To keep the mini-decks balanced, the Dealer draws only from mini-decks whose progress is below a certain threshold, which is the average $+1$. Let $\ell_{i,t}$ be the number of cards drawn from the i th mini-deck by turn t . At turn t the Dealer may draw from the i th mini-deck if $\ell_{i,t} < \lceil \frac{t}{d} \rceil + 1$. The Dealer performs rejection sampling to select a mini-deck, i.e., the Dealer samples repeatedly until a suitable mini-deck is found (we will see how to do this more efficiently). The Dealer plays this way until $2d$ cards are left, at which point some mini-decks might run empty. We refer to the first $n - 2d$ turns as the Adaptive-Threshold phase. When $2d$ turns are left, the Dealer shuffles all the remaining cards together and draws them at random, one after another. We refer to the last $2d$ turns as the Final phase. An algorithmic description of our Dealer is described in Algorithm 3.1.

We now describe the Dealer’s algorithm. It consists of the the Adaptive-Threshold phase for the first $n - 2d$ turns, and the Final phase for the last $2d$ turns.

Implementing Algorithm 3.1: In terms of space, our Dealer tracks the progress of the various mini-decks by their distance from the threshold. Let $x_{i,t}$ denote the distance of the i th mini-deck from the threshold in turn t , that is $x_{i,t} = \lceil \frac{t}{d} \rceil + 1 - \ell_{i,t}$. We encode the distances using Elias encoding³, and store the distances in a Variable bit-Length Array, as in Section 2.4.1. This allows us to update and track the state of a mini-deck in constant time. But how many mini-decks can we track this way, so that the total memory used is less than m ?

We treat the implementation of the Final phase in Section 4.3, since it overlaps with the implementation of the fast linear memory perfect permutation generator (note that the number of

³Any other variable-length encoding for positive integers that encodes a number $x \in \mathbb{N}$ using $O(\log(x))$ bits would work for our needs. In fact, in terms of space, we can allow ourselves to squander and encode the distances in unary, but that would come at the run time’s expense.

Algorithm 3.1 Adaptive-Threshold Dealer

 $\ell_i = 0 : \forall i \in d$

▷ **Adaptive-Threshold phase**

for turn $t \in [n - 2d]$ **do**

▷ draw a card from a suitable mini-deck

repeat

$i \sim [d]$

▷ sample a mini-deck uniformly at random

if $\ell_i < \lceil \frac{t}{d} \rceil + 1$ **then**

▷ If the sampled mini-deck is less advanced than the threshold

Draw the top card from the i th mini-deck

$\ell_i \leftarrow \ell_i + 1$

until a card is drawn

▷ **Final phase**

$A \leftarrow$ all remaining cards

for turn $t \in \{n - 2d, \dots, n\}$ **do**

Draw and discard a random card from A

cards left is $O(d)$ and the memory is $O(d)$ as well).

Observe that the value of the threshold increases every d turns. We refer to the span of turns during which the threshold is the same as a *stage*. So there are n/d stages, and the threshold is $\tau + 1$ during the τ th stage. We refer to the distance between the progress of the mini-decks and the threshold by *holes*. So when it is possible to draw k more cards from a mini-deck before reaching the threshold, we say that the mini-deck has k holes⁴.

Claim 3.1. *For any turn t , the total number of holes is between d and $2d$:*

$$d \leq \sum_{i \in [d]} x_{i,t} \leq 2d.$$

Proof. At the first turn of the first stage, the threshold is 2, and all mini-decks are full, so the Dealer can draw 2 cards from each mini-deck; thus, the number of holes is $2d$. In each turn the Dealer draws a card, so the number of holes decreases by 1. After d turns, the stage ends and the threshold grows by one, so the number of holes increases by d . Overall, the number of holes is at most $2d$ during all turns, exactly $2d$ at the beginning of every stage, and at least d . \square

Lemma 3.2. *The Adaptive-Threshold Dealer with m bits of memory tracks $d = \Theta(m)$ mini-decks.*

Proof. The Dealer tracks the progress of the mini-decks by their distance from the threshold and encodes the distances using an encoding function f_{encode} , such that $f_{\text{encode}}(x) \leq O(\log x)$. We get that the Dealer stores d distances and that the total number of bits required to store them is:

$$\sum_{i \in [d]} f_{\text{encode}}(x_{i,t}) = \sum_{i \in [d]} O(\log(x_{i,t})) = \sum_{i \in [d]} O(x_{i,t}) = O(d).$$

By using Variable bit-Length Array, and from Proposition 2.4, we know that the Dealer can do so by using $O(d)$ bits. \square

⁴Holes is a burrowed term from the balls-into-bins literature, in which the distance between the progress of less-advanced bins and a target load (in our case, the threshold) is sometimes referred to by the term *holes*; this captures the volume absent of balls that should be filled.

Roadmap for the rest of the proof: Having settled the number of mini-decks that we can track, we would like to show that our Dealer has a sufficient number of mini-decks that it can draw cards from in every turn in the Adaptive-Threshold phase. This would imply that there are sufficiently many possible cards to draw in each turn, so the Dealer is as unpredictable as we want throughout the game, and that the Dealer quickly finds a suitable mini-deck.

For turn t , we consider the vector of holes $x_t = (x_{1,t}, \dots, x_{d,t})$. Many mini-decks with holes implies that there are many options for the next card, so our goal is to show that the mini-decks progress somewhat evenly throughout the game, and that the holes are spread across the mini-decks. We apply the exponential potential function to the vector of holes and analyze the game through the lens of this function.

Set $\epsilon = 1/200$, and define the *potential contribution* of the i th mini-deck at the turn t to be $\Phi_i(x_t) = (1 + \epsilon)^{x_{i,t}}$, and let the *potential* of the holes vector x_t at turn t be

$$\Phi(x_t) = \sum_{i \in [d]} \Phi_i(x_t) = \sum_{i \in [d]} (1 + \epsilon)^{x_{i,t}}.$$

As an intuition, since there are at least d holes (Claim 3.1), in the extreme case where all holes are in one mini-deck, then the potential is $2^{\Omega(d)}$. Contrary to this example, we will show that the expected value of the potential function is $O(d)$ (in Section 3.2). We then use it to show that our Dealer is sufficiently unpredictable and that each turn takes a constant time in expectation (Section 3.3).

3.2 Bounding the Potential Function

As our Dealer relies heavily on the Adaptive-Threshold allocation scheme (Algorithm 2.4), we will walk the path paved by Berenbrink et al.[BKSS13] to show our results.

Our main machinery is the exponential potential function that we apply to the vector of holes. This potential function is monotone in the sense that mini-decks that lag behind, that is, have more holes, contribute more to the value of the potential function. We start by showing that there is some constant α such that, if a mini-deck has at least α holes at the end of a stage, then we expect to draw strictly more than one card from it during the following stage, thus, it progresses faster and has fewer holes than at the beginning of the stage. As a result, we expect its potential contribution to decrease. We then show that the potential decrease caused by lagging mini-decks suffices to bound the expected potential value of the vector of holes throughout the entire game.

As a convention, we will use Greek letters for constants, Capital letters for sets, **bold** ones for random variables, and lower letters for specific values of the corresponding random variables. We will often change focus between turns and stages (recall that stages are made of d consecutive turns). Therefore, for clarity, we will use subscripted t to refer to the state of objects during turn t and superscripted τ when referring to stages.

Let \mathbf{L}_i^τ be the random variable measuring the number of cards drawn from the i th mini-deck by the end of stage τ , and consider the “global” random variable vector that represents the progress of all mini-decks $\mathbf{L}^\tau = (\mathbf{L}_1^\tau, \dots, \mathbf{L}_d^\tau)$. Consider the equivalent random variable \mathbf{X}^τ that describes the number of holes in the i th mini-deck by the end of stage τ and is defined as $\mathbf{X}_i^\tau = \tau + 2 - \mathbf{L}_i^\tau$, and the random vector of deviation at the end of the stage $\mathbf{X}^\tau = (\mathbf{X}_1^\tau, \dots, \mathbf{X}_d^\tau)$. While we are at it, let $\mathbf{L}_{i,t}, \mathbf{L}_t, \mathbf{X}_{i,t}, \mathbf{X}_t$ be the corresponding random variables at a specific turn t . Let \mathbf{Y}_i^τ be the random variable measuring the number of cards drawn from the i th mini-deck at stage τ .

We start by showing that mini-decks that are too far from the threshold at some stage are likely to get closer to the threshold in the following stage; moreover, we expect to draw slightly more

than 1 card from such mini-decks. Call a mini-deck α -lagging at the end of stage τ if it has more than α holes, for some constant α .

Proposition 3.3 (Lemma 3.3 in [BKSS13]). *There exists a constant α , such that for any stage τ and any $0 \leq k \leq \alpha$, if mini-deck i is α -lagging at the end of stage τ , then the probability that the Adaptive-Threshold Dealer draws at least k cards from the i th mini-deck at stage $\tau + 1$ is at least*

$$\Pr[\mathbf{Y}_i^{\tau+1} \geq k] \geq \Pr[\text{Poi}(199/198) \geq k] - 2 \cdot 10^{-10}.$$

We use the exponential potential function to analyze the Dealer's progress throughout the game. Consider some deviation vector $x^\tau = (x_1^\tau, \dots, x_d^\tau)$ that captures the mini-decks holes by the end of stage τ . And recall that the further a mini-deck is from the threshold, the larger is its potential contribution. At the same time, observe that the potential contribution cannot grow much between stages.

Claim 3.4. *For any mini-deck $i \in [d]$ and any stage τ , the potential contribution of the i th mini-deck grows by a factor of at most $(1 + \epsilon)$ by the end of stage $\tau + 1$:*

$$\Phi_i(x^{\tau+1}) \leq (1 + \epsilon) \cdot \Phi_i(x^\tau).$$

Proof. It could be the case that the Dealer doesn't sample the i th mini-deck during a stage, resulting in the mini-deck i getting farther from the threshold by 1, and that's the worst that can happen. \square

The following proposition shows that there exists a constant $0 < \beta < 1$ that depends on α , such that the potential contribution of α -lagging mini-decks decreases in expectation by a factor of $(1 - \beta)$.

Proposition 3.5 (Lemma 3.4 in [BKSS13]). *If mini-deck i is α -lagging at the end of stage τ , then its potential contribution is expected to decrease at least by a factor of $(1 - \beta)$ on stage $\tau + 1$*

$$\mathbb{E} [\Phi_i(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] \leq (1 - \beta) \cdot \Phi_i(x^\tau).$$

Let $\gamma = (\beta + \epsilon) \cdot (1 + \epsilon)^\alpha \cdot \left(\frac{2}{\beta}\right)$ be a potential threshold constant. We use Proposition 3.5 to show that if the potential is too big, then the potential contribution decrease of lagging mini-decks is expected to reduce the potential of the entire system by the end of the following stage.

Lemma 3.6. *For any stage τ , and any holes vector x^τ such that $\mathbf{X}^\tau = x^\tau$, if the potential $\Phi(x^\tau)$ is larger than $\gamma \cdot d$, then the potential of $\mathbf{X}^{\tau+1}$ is expected to decrease by a factor of $\left(1 - \frac{\beta}{2}\right)$ by the end of stage $\tau + 1$,*

$$\mathbb{E}[\Phi(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] \leq \left(1 - \frac{\beta}{2}\right) \cdot \Phi(x^\tau).$$

Proof. Consider the subset of mini-decks $D_{\leq \alpha}^\tau \subseteq [d]$ such that, at the end of stage τ , are not α -lagging, i.e. mini-deck $i \in D_{\leq \alpha}^\tau$ if it has at most α holes. We get that:

$$\begin{aligned} \mathbb{E} [\Phi(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] &= \sum_{i \in [d]} \mathbb{E} [\Phi_i(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] \\ &= \sum_{i \notin D_{\leq \alpha}^\tau} \mathbb{E} [\Phi_i(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] + \sum_{i \in D_{\leq \alpha}^\tau} \mathbb{E} [\Phi_i(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] = (*). \end{aligned}$$

For the second summand, we use the general potential growth upper bound from Claim 3.4 to get that $\sum_{i \in D_{\leq \alpha}^\tau} \mathbb{E} [\Phi_i(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] \leq (1 + \epsilon) \cdot \sum_{i \in D_{\leq \alpha}^\tau} \Phi_i(x^\tau)$. And for the first summand, we apply Proposition 3.5 to conclude that

$$\begin{aligned} \sum_{i \notin D_{\leq \alpha}^\tau} \mathbb{E} [\Phi(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] &\leq (1 - \beta) \cdot \sum_{i \notin D_{\leq \alpha}^\tau} \Phi_i(x^\tau) \\ &= (1 - \beta) \cdot \left(\sum_{i \in [d]} \Phi_i(x^\tau) - \sum_{i \in D_{\leq \alpha}^\tau} \Phi_i(x^\tau) \right) \\ &= (1 - \beta) \cdot \Phi(x^\tau) + (\beta - 1) \cdot \sum_{i \in D_{\leq \alpha}^\tau} \Phi_i(x^\tau). \end{aligned}$$

Plugging these together into (*), we get that

$$\begin{aligned} \mathbb{E} [\Phi(\mathbf{X}^{\tau+1}) | \mathbf{X}^\tau = x^\tau] &\leq \left[(1 - \beta) \cdot \Phi(x^\tau) + (\beta - 1) \cdot \sum_{i \in D_{\leq \alpha}^\tau} \Phi_i(x^\tau) \right] + \left[(1 + \epsilon) \cdot \sum_{i \in D_{\leq \alpha}^\tau} \Phi_i(x^\tau) \right] \\ &= (1 - \beta) \cdot \Phi(x^\tau) + (\beta + \epsilon) \cdot \sum_{i \in D_{\leq \alpha}^\tau} \Phi_i(x^\tau) \\ &\leq (1 - \beta) \cdot \Phi(x^\tau) + (\beta + \epsilon) \cdot \sum_{i \in D_{\leq \alpha}^\tau} (1 + \epsilon)^\alpha \end{aligned} \quad (3.1)$$

$$\leq (1 - \beta) \cdot \Phi(x^\tau) + (\beta + \epsilon) \cdot d \cdot (1 + \epsilon)^\alpha \quad (3.2)$$

$$\leq (1 - \beta) \cdot \Phi(x^\tau) + \frac{\beta}{2} \cdot \Phi(x^\tau) \quad (3.3)$$

$$= \left(1 - \frac{\beta}{2} \right) \cdot \Phi(x^\tau).$$

Where Inequality (3.1) is true because for every $i \in D_{\leq \alpha}^\tau$ it holds that $\Phi_i(x^\tau) \leq (1 + \epsilon)^\alpha$, Inequality (3.2) is true because $|D_{\leq \alpha}^\tau| \leq d$, and Inequality (3.3) follows from our assumption that the potential is larger than $\Phi(x^\tau) \geq (\beta + \epsilon) \cdot (1 + \epsilon)^\alpha \cdot \left(\frac{2}{\beta}\right) \cdot d$. \square

Corollary 3.7. *For every stage τ , the expected potential is at most*

$$\mathbb{E} [\Phi(\mathbf{X}^\tau)] \leq d \cdot \left((1 + \epsilon)^2 \cdot \gamma \cdot \left(\frac{2}{\beta}\right) \right) = O(d).$$

Proof. We first observe that at the beginning of the very first stage, there are two holes in each mini-deck, and therefore $\Phi(x^0) = d \cdot (1 + \epsilon)^2 = O(d)$. Assume by induction that the statement is true for all stages up until $\tau - 1$, and we will show that it is also true for stage τ .

First, consider the case where $\mathbb{E}[\Phi(\mathbf{X}^{\tau-1})] \leq (1 + \epsilon) \cdot \gamma \cdot d \cdot \left(\frac{2}{\beta}\right)$:

$$\mathbb{E} [\Phi(\mathbf{X}^\tau)] \leq (1 + \epsilon) \cdot \mathbb{E} [\Phi(\mathbf{X}^{\tau-1})] \quad (3.4)$$

$$\leq (1 + \epsilon)^2 \cdot \gamma \cdot d \cdot \left(\frac{2}{\beta}\right). \quad (3.5)$$

Where Inequality (3.4) follows from Claim 3.4 and monotonicity of expectation, and Inequality (3.5) is true by the assumption that $\mathbb{E} [\Phi(\mathbf{X}^{\tau-1})] \leq (1 + \epsilon) \cdot \gamma \cdot d \cdot \left(\frac{2}{\beta}\right)$.

Otherwise, consider the case where $\mathbb{E} [\Phi(\mathbf{X}^{\tau-1})] \geq (1 + \epsilon) \cdot \gamma \cdot d \cdot \left(\frac{2}{\beta}\right)$. There are two options now, either the potential at the previous stage is above $\gamma \cdot d$ or below it. From the law of total expectation, we get that

$$\begin{aligned} \mathbb{E} [\Phi(\mathbf{X}^\tau)] &= \Pr [\Phi(\mathbf{X}^{\tau-1}) \leq \gamma \cdot d] \cdot \underbrace{\mathbb{E} [\Phi(\mathbf{X}^\tau) | \Phi(\mathbf{X}^{\tau-1}) \leq \gamma \cdot d]}_{(*)} \\ &\quad + \Pr [\Phi(\mathbf{X}^{\tau-1}) > \gamma \cdot d] \cdot \underbrace{\mathbb{E} [\Phi(\mathbf{X}^\tau) | \Phi(\mathbf{X}^{\tau-1}) > \gamma \cdot d]}_{(**)}. \end{aligned} \quad (3.6)$$

We handle the two terms separately.

$$(*) \leq (1 + \epsilon) \cdot \gamma \cdot d \quad (3.7)$$

$$\leq \left(\frac{\beta}{2}\right) \cdot \mathbb{E} [\Phi(\mathbf{X}^{\tau-1})]. \quad (3.8)$$

Where Inequality (3.7) follows from the general potential increase upper bound described in Claim 3.4, and Inequality (3.8) follows from the assumption that $\mathbb{E} [\Phi(\mathbf{X}^{\tau-1})] \geq (1 + \epsilon) \cdot \gamma \cdot d \cdot \left(\frac{2}{\beta}\right)$.

As for the second term, we apply Lemma 3.6 to conclude that

$$(**) \leq \left(1 - \frac{\beta}{2}\right) \cdot \mathbb{E} [\Phi(\mathbf{X}^{\tau-1})]. \quad (3.9)$$

We observe that the right-hand side of Equality (3.6) is a convex combination of (*) and (**), thus it is clearly smaller than their sum. Placing Inequality (3.8) and Inequality (3.9) into Equality (3.6) we get that

$$\begin{aligned} \mathbb{E} [\Phi(\mathbf{X}^\tau)] &< \left(\frac{\beta}{2}\right) \cdot \mathbb{E} [\Phi(\mathbf{X}^{\tau-1})] + \left(1 - \frac{\beta}{2}\right) \cdot \mathbb{E} [\Phi(\mathbf{X}^{\tau-1})] \\ &= \mathbb{E} [\Phi(\mathbf{X}^{\tau-1})]. \end{aligned}$$

Which, by our inductive assumption, satisfies the desired bound. \square

3.3 Predictability and Run Time

We now show that our Dealer is sufficiently unpredictable and that each turn takes a constant time in expectation. To do so, we focus on the set of mini-decks whose potential contribution is relatively small (compared to the average contribution), and we show that they form a constant fraction of *all* mini-decks, in every turn. This implies that (i) the rejection-sampling process is expected to take only a constant time. Since all other parts run in worst-case constant time, we get that the Dealer runs in constant time in expectation. And (ii) that the Dealer is expected to have many mini-decks to draw from, thus, the Guesser's expected benefit is small. We show that this is the case for all turns, and so, by linearity of expectation, we conclude the expected score of our Dealer.

For a stage τ , and a vector of holes \mathbf{X}^τ at the end of stage τ , we introduce a new random variable $\mathbf{B}^\tau = \max \left\{ \frac{\Phi(\mathbf{X}^\tau)}{d}, \frac{16}{\epsilon^2} \right\}$, which can be roughly thought of as the average potential contribution of the various mini-decks.

Claim 3.8. *For every stage τ , $\mathbb{E}[\mathbf{B}^\tau] = O(1)$.*

Proof. Consider the constants $c_1 = (1 + \epsilon)^2 \cdot \gamma \cdot \left(\frac{2}{\beta}\right)$ and $c_2 = 16/\epsilon^2$. From Corollary 3.7 we know that

$$\mathbb{E}[\Phi(\mathbf{X}^\tau)] \leq c_1 \cdot d,$$

so we can conclude that $\mathbb{E}\left[\frac{\Phi(\mathbf{X}^\tau)}{d}\right] \leq c_1$. Considering that \mathbf{B}^τ is either $\frac{\Phi(\mathbf{X}^\tau)}{d}$ or c_2 , and using the law of total expectation, we get that $\mathbb{E}[\mathbf{B}^\tau]$ is a convex combination of two constants, thus upper bounded by a constant. \square

Observe that for any turn there is a finite set of possible values for the holes vector, and thus, for the possible potential values. Let \mathcal{B}^τ be the set of possible values for \mathbf{B}^τ . Let D_k^τ be the set of mini-decks that have exactly k holes by the end of the τ th stage; that is, for every mini-deck $i \in D_k^\tau$ it holds that $\Phi_i(x^\tau) = (1 + \epsilon)^k$.

Claim 3.9. *For every stage τ , every $b^\tau \in \mathcal{B}^\tau$, and every $k \in [\tau + 1]$, if the average potential contribution at the end of stage τ is b^τ , then, the number of mini-decks with k holes is at most*

$$|D_k^\tau| \leq b^\tau \cdot d \cdot 2^{-k\epsilon}.$$

Proof. Let x^τ be any hole vector at the end of stage τ such that the average potential contribution is b^τ , then

$$b^\tau \cdot d \geq \Phi(x^\tau) := \sum_{i \in [d]} \Phi_i(x^\tau) \geq \sum_{i \in D_k^\tau} (1 + \epsilon)^k = |D_k^\tau| \cdot (1 + \epsilon)^k.$$

The first inequality is true by the definition of \mathbf{B}^τ , and the second inequality is true by the definition of D_k^τ and since $D_k^\tau \subseteq [d]$.

We get that

$$|D_k^\tau| \leq b^\tau \cdot d \cdot (1 + \epsilon)^{-k} \leq b^\tau \cdot d \cdot 2^{-k\epsilon}$$

where the last inequality is true because $(1 + \epsilon)^{-1} \leq 2^{-\epsilon}$ for $0 \leq \epsilon \leq 1$. \square

Observe that a mini-deck can have at most $\tau + 2$ holes during stage τ . We use this fact to conclude an upper bound on the number of holes in mini-decks that are relatively far behind.

Claim 3.10. *For every stage τ and every $b^\tau \in \mathcal{B}^\tau$, if the average potential contribution at the end of stage τ is b^τ , then the total number of holes in mini-decks having at least $\frac{4 \ln(b^\tau)}{\epsilon}$ holes is at most*

$$\sum_{k=4 \ln(b^\tau)/\epsilon}^{\tau+2} |D_k^\tau| \cdot k \leq d \cdot \frac{8}{\epsilon^2} \cdot \frac{1}{b^\tau}.$$

Proof.

$$\sum_{k=4 \ln(b^\tau)/\epsilon}^{\tau+2} |D_k^\tau| \cdot k \leq \sum_{k=4 \ln(b^\tau)/\epsilon}^{\tau+2} b^\tau \cdot d \cdot 2^{-k\epsilon} \cdot k \quad (3.10)$$

$$\begin{aligned} &= \frac{b^\tau \cdot d}{\epsilon} \cdot \sum_{k=4 \ln(b^\tau)/\epsilon}^{\tau+2} 2^{-k\epsilon} \cdot k \cdot \epsilon \\ &= \frac{b^\tau \cdot d}{\epsilon} \cdot \sum_{k=4 \ln(b^\tau)/\epsilon}^{\tau+2} e^{-k\epsilon/2} \cdot \left(\frac{4}{e}\right)^{-k\epsilon/2} \cdot k \cdot \epsilon \\ &< \frac{b^\tau \cdot d}{\epsilon} \cdot \sum_{k=4 \ln(b^\tau)/\epsilon}^{\tau+2} e^{-k\epsilon/2} \cdot 2 \end{aligned} \quad (3.11)$$

$$< 2 \cdot \frac{b^\tau \cdot d}{\epsilon} \cdot \frac{e^{-2 \cdot \ln(b^\tau)}}{1 - e^{\epsilon/2}} \quad (3.12)$$

$$< 2 \cdot \frac{b^\tau \cdot d}{\epsilon} \cdot \frac{(b^\tau)^{-2}}{\epsilon/4} = d \cdot \frac{8}{\epsilon^2} \cdot \frac{1}{b^\tau}. \quad (3.13)$$

Where Inequality (3.10) follows from Claim 3.9, Inequality (3.11) is true since $(4/e)^{x/2} \cdot x < 2$ for every x . We deduce Inequality (3.12) from the closed form of the sum of the corresponding infinite geometric series, and Inequality (3.13) is true because $x/2 \leq 1 - e^{-x}$ for $0 \leq x \leq 1$. \square

For a turn t in stage τ , such that $\mathbf{B}^\tau = b^\tau$ for some value b^τ , let D'_t be the set of mini-decks that have at least 1 hole and at most $4 \ln(b^\tau)/\epsilon$ holes at turn t .

Claim 3.11. *For every stage τ , and every value $b^\tau \in \mathcal{B}^\tau$, if the average potential contribution at the end of stage τ is b^τ , then for every turn t in stage $\tau + 1$, the set of mini-decks with at least 1 hole and at most $4 \ln(b^\tau)/\epsilon$ holes at turn t is at least*

$$|D'_t| = \frac{\epsilon}{8} \cdot \frac{d}{\ln(b^\tau)} = \Omega\left(\frac{d}{\ln(b^\tau)}\right)$$

Proof. Recall (as appears in the proof of Claim 3.1) that the number of holes is exactly $2d$ at the beginning of each stage. Let t' be the first turn at stage $\tau + 1$. From Claim 3.10 we conclude that the number of holes over mini-decks in $D'_{t'}$ is at least $2d - d \cdot \frac{8}{\epsilon^2} \cdot \frac{1}{b^\tau} \geq 2d - d \cdot \frac{8}{\epsilon^2} \cdot \frac{\epsilon^2}{16} = \frac{3}{2}d$. It follows that even if the Dealer draws only from mini-decks in $D'_{t'}$, then after $d - 1$ draws, there are at least $d/2$ holes in mini-decks from $D'_{t'}$. Thus, this is true for every turn t in the stage.

Because D'_t contains mini-decks with at most $4 \ln(b^\tau)/\epsilon$, we get that

$$|D'_t| \geq \frac{d}{2} \cdot \frac{\epsilon}{4 \ln(b^\tau)} = \Omega\left(\frac{d}{\ln(b^\tau)}\right).$$

\square

The following lemmata combine the lower bound on the size of relatively advanced mini-decks D'_t (Claim 3.11) and the upper bound on the expectation of the potential (Claim 3.8), to conclude the Dealer's run time and predictability. Their proofs share a similar structure and analysis.

Recall that the Dealer samples mini-decks until a drawable one is found. Let \mathbf{T}_t be the random variable measuring the time it takes the Dealer to draw a card at turn t .

Lemma 3.12. *For every turn t , the Dealer's expected run time at turn t is $\mathbb{E}[\mathbf{T}_t] = O(1)$.*

Proof. Let τ be the stage prior to turn t . Let $D_t \subseteq [d]$ be the set of all drawable mini-decks at turn t . Clearly, $D'_t \subseteq D_t$, and therefore, the expected number of trials to find a mini-deck from D_t is upper bounded by the expected number of trials to find a mini-deck from D'_t , the later is a geometric random variable with probability $|D'_t|/d$. Using the lower bound on D'_t from Claim 3.11 we get that if $\mathbf{B}^\tau = b^\tau$ then expected time for success is $O(\ln(b^\tau))$, where the expectation is taken solely over the randomness of the dealer at turn t .

Let \mathcal{B}^τ be the set of possible values for \mathbf{B}^τ . We get that

$$\mathbb{E}[\mathbf{T}_t] = \sum_{b^\tau \in \mathcal{B}^\tau} \mathbb{E}[\mathbf{T}_t | \mathbf{B}^\tau = b^\tau] \cdot \Pr[\mathbf{B}^\tau = b^\tau] \quad (3.14)$$

$$\leq \sum_{b^\tau \in \mathcal{B}^\tau} O(\ln(b^\tau)) \cdot \Pr[\mathbf{B}^\tau = b^\tau] \quad (3.15)$$

$$= \mathbb{E}[O(\ln(\mathbf{B}^\tau))] \quad (3.16)$$

$$\leq O(\ln(\mathbb{E}[\mathbf{B}^\tau])) \quad (3.17)$$

$$\leq O(1). \quad (3.18)$$

Where Equality (3.14) follows from the law of total expectation, Inequality (3.15) is true by the discussion above, Equality (3.16) is true by the definition of expectation, Inequality (3.17) follows from the linearity of expectation and from Jensen inequality Proposition 2.8, and Inequality (3.18) follows from Claim 3.8. □

Observe that the Guesser's expected benefit is bounded from above by the number of drawable mini-decks. Let the indicator random variable \mathbf{C}_t be the event that the Guesser predicted the Dealer's draw correctly at turn t , and thus scored a point.

Lemma 3.13. *For every turn t , any Guesser that plays against the Adaptive-Threshold Dealer is expected to score at most*

$$\mathbb{E}[\mathbf{C}_t] \leq O\left(\frac{1}{d}\right).$$

Proof. Let τ be the stage prior to turn t . Assume that $\mathbf{B}^\tau = b^\tau$ for some $b^\tau \in \mathcal{B}^\tau$, and recall that $D_t \subseteq [d]$ is the set of all drawable mini-decks at turn t . Clearly, $D'_t \subseteq D_t$, and therefore, the probability (over the randomness at turn t) that a Guesser's guess is correct is at most:

$$\Pr[\mathbf{C}_t | \mathbf{B}^\tau = b^\tau] \leq \frac{1}{|D_t|} \leq \frac{1}{|D'_t|} \leq \frac{8}{\epsilon} \cdot \frac{\ln(b^\tau)}{d}. \quad (3.19)$$

Where the last inequality follows from Claim 3.11.

Thus, we expect turn t to yield at most $\mathbb{E}[\mathbf{C}_t | \mathbf{B}^\tau = b^\tau] \leq \frac{8}{\epsilon} \cdot \frac{\ln(b^\tau)}{d}$ correct guesses. We get that

$$\mathbb{E}[\mathbf{C}_t] = \sum_{b^\tau \in \mathcal{B}^\tau} \mathbb{E}[\mathbf{C}_t | \mathbf{B}^\tau = b^\tau] \cdot \Pr[\mathbf{B}^\tau = b^\tau] \quad (3.20)$$

$$\leq \sum_{b^\tau \in \mathcal{B}^\tau} \left[\frac{8}{\epsilon} \cdot \frac{\ln(b^\tau)}{d} \right] \cdot \Pr[\mathbf{B}^\tau = b^\tau] \quad (3.21)$$

$$= \frac{1}{d} \cdot \frac{8}{\epsilon} \cdot \mathbb{E}[\ln(\mathbf{B}^\tau)] \quad (3.22)$$

$$\leq \frac{1}{d} \cdot \frac{8}{\epsilon} \cdot \ln(\mathbb{E}[\mathbf{B}^\tau]) \quad (3.23)$$

$$\leq \frac{1}{d} \cdot \frac{8}{\epsilon} \cdot \ln(O(1)) = O\left(\frac{1}{d}\right). \quad (3.24)$$

Where Equality (3.20) follows from the law of total expectation, Inequality (3.21) is true by the discussion above, Equality (3.22) is true by definition of expectation and its linearity, Inequality (3.23) follows from Jensen Inequality Proposition 2.8, and Inequality (3.24) follows from Claim 3.8. \square

Summing up: In Lemma 3.2 we showed that an Adaptive-Threshold Dealer with m bits of memory tracks $d = \Theta(m)$ mini-decks. Thus, in Lemma 3.13 we showed that during the Adaptive-Threshold phase, the expected score in each turn is at most $O(1/m)$, and, by linearity of expectation, any Guesser is expected to score at most $O(n/m)$ during the Adaptive-Threshold phase. By Lemma 3.12, each of these turns is expected to take a constant time.

The main result for this section follows:

Theorem 3.14. *An Adaptive-Threshold Dealer with m bits of memory scores $O(n/m)$ points in expectation against any Guesser during the Adaptive-Threshold phase, and each draw takes $O(1)$ time in expectation.*

In Section 4.3 we improve upon this result and expand the analysis for all turns during the game. We show that any Guesser scores $O(n/m + \ln m)$ points in expectation, and that each turn runs in worst case constant time. We do so by replacing the rejection sampling mechanism and addressing the Final phase.

4 Sampling from a Dynamic Subset in Constant Time

In this section we present a data structure for maintaining and sampling from a dynamic subset in worst case constant time. We use this data structure to complete the analysis of the Adaptive-Threshold Dealer (Algorithm 3.1) during the Final phase, and to improve upon its runtime during the Adaptive-Threshold phase. We also use it to construct an efficient Dealer that generates all permutations with equal probability; that is, one that draws available cards uniformly at random in each turn. As a result, any Guesser that plays against this Dealer scores at most $\ln n$ correct guesses in expectation. The main point of this Dealer is that it requires $O(n)$ bits of memory and runs in worst-case constant time.

Our constructions revolves around the ability to maintain and sample from the set of available cards. Generally speaking, for a given domain B , we present a data structure for maintaining and sampling from a dynamic subset $A \subseteq B$.

Theorem 4.1. *There exists a data structure S such that for any domain B , S maintains a dynamic subset $A \subseteq B$. The data structure S supports membership queries, addition and removal of an*

element to and from A , and sampling of a random member of A uniformly at random. All operations run in worst-case constant time, while using $O(|B|)$ bits of memory.

Given such a data structure, implementing our linear-space and constant time Dealer is straightforward: use S to maintain the set of available cards. Begin by initializing S to track the entire deck of n cards. In each turn, the Dealer samples uniformly at random an available card to draw, and removes it from the set. Since both operations run in constant time, so does the Dealer. All available cards are equiprobable, so the expected score is $\ln n$. The Dealer solely uses S , so it consumes $O(n)$ bits of memory.

Theorem 4.2. *There exists a Dealer with $O(n)$ bits of memory that generates a permutation uniformly at random. Any Guesser is expected to score at most $\ln n$ correct guesses against this Dealer. Furthermore, this Dealer is efficient, and each turn takes worst-case constant time.*

We describe our data structure across Sections 4.1 and 4.2. We then show how to use it in order to implement the Final phase of the Adaptive-Threshold Dealer and how to make it run in worst case constant time, these are discussed in Section 4.3.

4.1 Subset Data Structure

It seems natural to allocate a bit for every element in the domain B to indicate its membership to A , and store these bits in small cells that allow fast operations. With simple bit-wise operations on words, we can mark a cell's bit as a member of A , discard it from A , and query its membership. Fixing the cells in an array forms a bitmap that supports addition and removal of a single element, as well as set membership queries, all in constant time and within the memory requirements.

What about sampling? We think of a cell as inducing a restricted subset of A . The cells are small enough, so we can sample a random element from it in constant time. But how do we sample a cell? Choosing one at random implies that elements from less populated cells are drawn with a higher probability than elements from less populated ones. It follows that to draw an element uniformly at random, we need to sample a cell proportionally to its population size, that is, to the number of elements from A that it tracks.

Observe that cells that hold the same number of elements must be equally probable. This hints that our data structure requires the ability to uniformly sample a cell of a given population size, and also requires the ability to sample a population size in proportion to the number of elements from that size. So on a high-level, the data structure first samples a population size, then picks a cell with this many elements, and finally, chooses a random element from that cell.

For brevity, we denote the size of the domain B by $n = |B|$, and assume that n is a power of 2. We also assume that there is a natural mapping between $[n]$ and B .

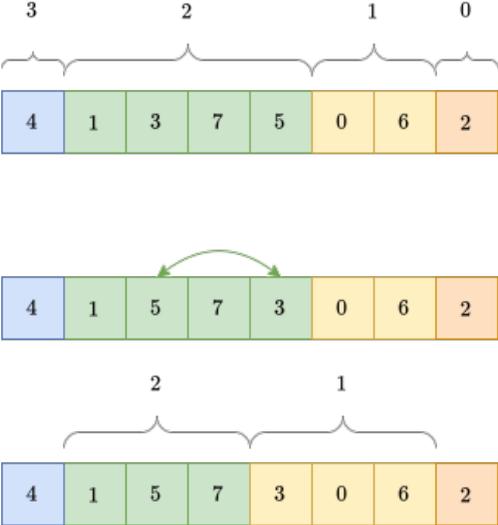
Cells: We split the domain B to cells, where each cell tracks $\log n$ elements. If a cell contains p elements from A , then we say that the cell is of population p , and similarly, that the elements in it are of population p .

In terms of space, each cell stores $\log n$ bits to track the membership of $\log n$ elements. We would like to move cells around, so each cell also contains an index j so the i th bit of that cell indicates the membership of the element numbered $(j \cdot \log n) + i$. Overall, each cell requires $O(\log n)$ bits, and there are $n/\log n$ cells, so we are good. Simple bit manipulation on words allows us to sample a random element and update its membership in constant time, as can be seen in Algorithm A.1. But how do we sample a cell?

Intervals: After removing an element from a cell of population size p it becomes a cell of population size $p - 1$. In other words, the set of cells of population $p - 1$ grows by 1, at the expense of the set of cells of population p , while the union of both sets remains in the same size. This works the same also for adding an element to a cell, and hints that updates, as well as sampling, can be done in a local manner.

We store the cells of the same population size sequentially and in an ordered manner, so that the cells of population p are stored before those of population $p - 1$. We refer to a sequence of cells of the same population size by *interval*. For each interval, we store the number of cells of that interval, as well as a pointer to its first cell. This allows us to maintain the ordered structure, as well as to sample a random cell of a given population size, at worst case constant time, while using only $O(\log^2 n)$ additional space. An algorithmic description is provided in Algorithm A.2. A visual example of interval maintenance is provided in Figure 1.

Figure 1: Cells ordered in intervals. Top to bottom: (i) there are 8 cells mentioned by their index (j). Cell 4 holds 3 elements (the blue interval), cells 1, 3, 7 and 5 contain 2 elements (the green interval), cells 0 and 6 track a single element (the yellow interval), and cell 2 is empty (red). (ii) We remove an element from cell 3, thus, it switches places with the last cell in the interval, which is cell 5. (iii) Cell 3 holds a single element now, and therefore it is marked as the beginning of the yellow interval (cells of population 1).



Population: Since each cell contains at most $\log n$ elements, then there are $\log n$ possible population sizes (empty cells are not interesting). When we remove an element from a cell of population p , that cell becomes one of population $p - 1$. In other words, let a_p be the number of elements that reside in cells that contain exactly p elements. Observe that after removing an element of population p , a_p is reduced by p , and a_{p-1} grows by $p - 1$. We would like to sample a population size according to the dynamic pseudo distribution induced by

$$(a_1, \dots, a_{\log n}).$$

We claim this is possible with the given time and memory requirements, and show the proof in Section 4.2.

Lemma 4.3. *It is possible to maintain the dynamic pseudo distribution induced by $(a_1, \dots, a_{\log n})$ while using $O(n)$ bits, where updates and sampling run in worst-case constant time.*

Having settled the space and run time requirement, it is left to show that we actually sample each element of A uniformly at random. This completes the proof of Theorem 4.1.

Claim 4.4. *All elements of A are equiprobable.*

Proof. Fix some element $b \in A$ that resides in a cell of population p , whose interval contains ℓ cells, and assume that $|A| = \alpha$. Let the indicator random variable \mathbf{Y}_b be the event that the sampling process generated b . The element b is sampled in probability $1/p$ if its cell was selected (and 0 otherwise), which happens in probability $1/\ell$ conditioned on the event that population p was sampled (and 0 otherwise). The latter happens with probability a_p/α . Since $a_p = \ell \cdot p$, we get that

$$\Pr[\mathbf{Y}_b] = \frac{1}{p} \cdot \frac{1}{\ell} \cdot \frac{\ell \cdot p}{\alpha} = \frac{1}{\alpha}.$$

□

4.2 Pseudo Distribution of Populations

And now for something completely different.

We now present a data structure for maintaining and sampling from the dynamically changing distribution of populations⁵, and thus prove Lemma 4.3.

To recap, and as a prelude to what's to come, we would like to sample a population with respect to the number of elements of that population. This distribution is induced by the vector $(a_1, \dots, a_{\log n}) \in [n]^{\log n}$, where a_p is the number of elements that reside in cells that contain p elements. As mentioned, observe that the distribution of population changes in a specific way: removing an element from a cell of population p , implies that a_p decreases by p , and a_{p-1} increases by $p-1$, and vice versa when adding an element. So for every p , a_p changes by at most $\log n$.

Consider the presentation of a number $a \in \mathbb{N}$ as being made of an integer multiplication of $\log n$ and some leftover; this results in a quotient and a residue.

$$a = \underbrace{\left\lfloor \frac{a}{\log n} \right\rfloor}_{\text{quotient}} \cdot \log n + \underbrace{(a \bmod \log n)}_{\text{residue}}.$$

Observe now that the addition and subtraction of any quantity that is at most $\log n$, changes the quotient at most by one. This fact will be of great use to us.

Given a pseudo probability mass vector $(a_1, \dots, a_{\log n})$ we split the masses into quotients and residues and manage them in two different data structures. Let $q_i = \lfloor a_i / \log n \rfloor$, and $r_i = (a_i \bmod \log n)$ be the quotient and residue (resp.) of the mass of outcome i . Let $t_q = \log n \cdot \sum_{i=1}^{\log n} q_i$ be the total mass of quotients, and let $t_r = \sum_{i=1}^{\log n} r_i$ be the total mass of residues.

The pseudo distribution of quotients $(q_1 \cdot \log n, \dots, q_{\log n} \cdot \log n)$ is maintained in one data structure, and the pseudo distribution of residues $(r_1, \dots, r_{\log n})$ in another. Flip a biased coin with probability proportional to the ratio between the total mass of quotients and the total mass of residues $t_q : t_r$, and sample from the resulting data structure. This process clearly samples from the distribution $(a_1, \dots, a_{\log n})$. So, it is left to show how to maintain the two distributions.

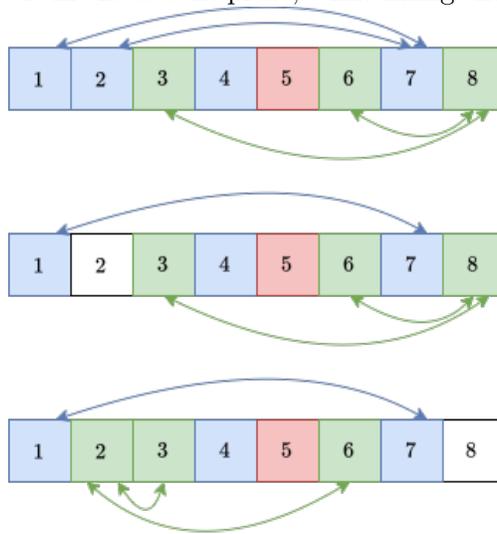
⁵See Section 2.5 for some background about dynamically changing distributions.

Quotients: Observe that the quotients pseudo distribution induced by $(q_1 \cdot \log n, \dots, q_{\log n} \cdot \log n)$ is exactly the pseudo distribution induced by $(q_1, \dots, q_{\log n})$. And as discussed, each update adds or removes at most a single point of mass there. This is a special case of a dynamically changing distribution that is used for simulating sampling of colored marbles from an urn, with and without replacement.

This can be done in a simple and explicit way⁶. To simulate an urn that contains m marbles of k colors, we represent each color as a doubly-linked list, where each node in the list represents a single marble. By storing an anchor to each linked list, we get that addition and removal can be implemented in worst-case constant time. As for sampling, the nodes of all linked lists are stored in an array of size m . We make sure that there are no empty entries, so when a marble is removed, we fill its position in the array with another marble and fix its pointers. So sampling a marble is done by sampling a random index in the array and checking its color. Sampling without replacement is a combination of sampling and removal.

An algorithmic description of the Urn data structure is provided in Algorithm A.3 and Algorithm A.4. A visual representation of the removal process is provided in Figure 2.

Figure 2: Urn data structure - marble removal process, top to bottom: (i) Initially, there are 8 marbles of 3 colors. We would like to remove a blue marble. (ii) The node in index 2 is removed, and the pointers are updated accordingly. (iii) To preserve the array's continuity, we move the last marble in the array and store it in the absent place, while fixing the pointers.



Claim 4.5. *There exists a data structure U that stores an urn of m marbles, where each marble is colored in one of k colors, such that U requires $O(m \log m + m \log k)$ bits of memory, and supports sampling, addition and removal of a marble in worst-case constant time.*

It follows that we can maintain the quotients distribution by using the Urn data structure with $m = \frac{n}{\log n}$ and $k = \log n$, for which updates and samples will run in worst-case constant time, and within the given space requirements.

Residues: We observe that the total mass of residues t_r is less than $\log^2 n$, and that there are $\log n$ possible outcomes. It is possible to maintain and sample from such distributions by using

⁶That might entertain some job interviewers.

a construction by Hagerup, Mehlhorn and Munro [HMM93] as described in Proposition 2.5. This would give an expected constant sampling run time.

Another option is to observe that while there are more than $\log^3 n$ elements, we sample from the distribution of residues with probability $O(1/\log n)$. So we can allow ourselves a relatively long sampling time. One way to do so is to store the masses explicitly, and when asked to generate an outcome, we first sample an integer $u \sim [\sum_{i=1}^{\log n} r_i]$ uniformly at random, and compare it against the corresponding prefix vector of the pseudo distribution, that is $(0, r_1, r_1 + r_2, \dots, \sum_{i=1}^{\log n-1} r_i)$, and return the index of the largest entry that is smaller than u . Since this can be done in $O(\log n)$ time, we get constant time in expectation. But we aim for worst-case constant time, so we will do better.

Recall that we have $O(n)$ bits of memory, and can perform a set of operations on words of size $\log n$ (See Section 2.1). As a result, we can explicitly store a small function in memory, so that evaluating it takes only a constant time, say, by storing an array A such that $A[x]$ holds the evaluation of the function on input x .

Call a pseudo distribution *b-mass bounded* if the mass of every outcome is at most b . Consider a function f that receives a vector $(a_1, \dots, a_k) \in [b]^k$ and a number $u \in [b \cdot k]$ and outputs a member of $[k]$ as described in the prefix-sum scheme above, that is $f : [b]^k \times [b \cdot k] \rightarrow k$. By choosing a random u uniformly at random from $[\sum_{i \in [k]} a_i]$ we, in fact, sample according to the given pseudo distribution vector. What does it take to store f in memory?

Claim 4.6. *For every $b, k \in \mathbb{N}$, if $b^k \cdot (b \cdot k) \cdot \log k \leq n$, then it is possible to maintain, and sample from, any dynamically changing b -mass bounded pseudo distribution with support of size k in worst case constant time.*

Proof. Store the vector of masses explicitly and the sum of masses; this takes $O(k \cdot \log b)$ bits and allows updates in constant time. There are b^k possible b -mass bounded distribution vectors. For each one of them, we sample a random u such that $u \leq \sum_{i \in [k]} a_i \leq k \cdot b$. So overall, we can store the sampling function f as an array of $b^k \cdot (b \cdot k)$ entries (one for every possible input), where each entry contains outcome, which is of size $\log k$. Now sampling takes a constant time, and our only wish is that this array would fit into memory, which is given by the assumption. \square

Back to the distribution of residues, while being $\log n$ -mass bounded, the support is too large for the sampling function to fit into memory. We overcome this obstacle by partitioning the support of the distribution to smaller sets of equal size, and maintaining the pseudo distribution of each one of them separately. Now we are left with the task of maintaining and sampling from the distribution of support sets. This can be visualized as a depth-2 tree, where each leaf is responsible for maintaining a distribution of a certain partition, and has a total mass, where the root maintains the distribution of the leaves with respect to their total mass. So, generating an outcome is done by first sampling a leaf according to the distribution at the root, and then sampling an outcome according to the distribution at the selected leaf. Updates involve only the relevant leaf and the root, so they are both performed at constant run time. In terms of space, this requires $O(\log n \cdot \log \log n)$ bits of memory, in addition to storing the sampling functions in memory. Since the distributions in the leaves are similarly mass bounded and are over the same number of outcomes, it suffices to store a single sampling function for the leaves (call it f_0), and another one for the distribution at the root (call it f_1).

In more detail, we partition $[\log n]$ to $k_1 = 2 \log \log n$ disjoint sets, each of size $k_0 = \frac{\log n}{2 \log \log n}$, and we consider the pseudo distributions restricted to each support set. Observe that the distribution of any particular support set is b_0 -mass bounded where $b_0 = \log n$. As a result, the total mass of any leaf is upper bounded by $b_1 = b_0 \cdot k_0 = \frac{\log^2 n}{2 \log \log n}$. It follows that the distribution at the root

is b_1 -mass bounded, and has k_1 possible outcomes. Similarly, the distributions at the leaves are all b_0 -mass bounded, and have support of size k_0 . Let f_0 (f_1) be the function for sampling from any b_0 -mass bounded (resp. b_1 -mass bounded) distributions over support of size k_0 (resp. k_1).

Claim 4.7. *It is possible to store f_0 and f_1 in memory.*

Proof. To use Claim 4.6 we would like to make sure that the inequality

$$\underbrace{b^k}_{(*)} \cdot \underbrace{(b \cdot k) \cdot \log k}_{(**)} \leq n$$

holds for $(b, k) = (b_0, k_0) = (\log n, \frac{\log n}{2 \log \log n})$ and for $(b, k) = (b_1, k_1) = (\frac{\log^2 n}{2 \log \log n}, 2 \log \log n)$. We observe that for both cases $(**)$ is upper bounded by poly $\log n$. We show that $(*) \leq \sqrt{n}$, and the result follows.

And indeed, for b_0 and k_0 :

$$k_0 \cdot \log b_0 = \frac{\log n}{2 \log \log n} \cdot \log \log n = \frac{1}{2} \log n.$$

Taking exponents on both sides implies that $(*) = b_0^{k_0} = \sqrt{n}$.

As for b_1 and k_1 :

$$k_1 = 2 \log \log n \leq \frac{\log n}{4 \log \log n} = \frac{\log n}{2 \log (\log^2 n)} \leq \frac{\log n}{2 \log \left(\frac{\log^2 n}{\log \log n} \right)} = \frac{\log n}{2 \log b_1}.$$

$$k_1 \cdot \log b_1 \leq \frac{\log n}{2}.$$

$$b_1^{k_1} \leq \sqrt{n}.$$

□

As a corollary, updating the distribution of residues, as well as sampling from it takes only a constant time in the worst case, and requires $O(\log n \cdot \log \log n)$ bits in addition to two preprocessed functions that takes $O(n)$ bits.

Since it is possible to maintain and sample in constant time, both from the distribution of quotients and the distribution of residues, both within the given space requirements, then it is possible to maintain and sample from the distribution of populations in worst-case constant time. Lemma 4.3 follows.

4.3 Back to the Adaptive-Threshold Dealer

Looking back at our Adaptive-Threshold Dealer from Algorithm 3.1, we would like to resolve two issues: (i) the time per draw during the Adaptive-Threshold phase, making it constant time in the worst case rather than expected case and (ii) to describe the Dealer's internals for the Final phase.

Starting with the Adaptive-Threshold phase, in each turn, Adaptive-Threshold Dealer samples a mini-deck repeatedly until a drawable one is found. We showed (in Lemma 3.12) that the process of finding a suitable mini-deck is expected to terminate after a constant time. We claim that there is a better way. Consider the set $D_t \subseteq [d]$ of all mini-decks that contain at least one hole at turn t , thus, we can draw a card from them. At the beginning of each stage, all mini-decks are such, and when a mini-deck reaches the threshold, we remove it from this set. We can use the data structure from Theorem 4.1 to maintain and sample from the set of drawable mini-decks in

worst-case constant time, and within the required space. We keep two copies of this data structure: one for the odd stages and one for the even stages. In each stage, we maintain and sample from one of them as described above, while we replenish the other to be ready for the beginning of the next stage. This implies that the Adaptive-Threshold Dealer runs in worst-case constant time per turn during the Adaptive-Threshold phase. Since each drawable mini-deck is equiprobable, the analysis of the Adaptive-Threshold phase holds, and we get the following improvement upon Theorem 3.14.

Lemma 4.8. *An Adaptive-Threshold Dealer with m bits of memory scores $O(n/m)$ points in expectation against any Guesser during the Adaptive-Threshold phase, and each draw runs in worst case constant time.*

As for the Final phase, recall that in Algorithm 3.1, the Adaptive-Threshold Dealer draws the last $2d$ cards uniformly at random. That is, we would like to generate a permutation of the remaining cards. Observe that generating a permutation of $2d$ cards is a similar task to what we did in this section. On the other hand, since a mini-deck might hold all remaining cards, then the initial super-set of possible cards is now $d \cdot 2d$, which requires more bits than we are willing to spend. So we need to come up with an efficient way to enumerate the remaining available cards, that is, to easily associate the set $[2d]$ with the set of available cards.

We introduce a small enhancement to the Cells data structure (Section 4.1) that allows us to draw the last $2d$ remaining cards in worst-case constant time, and in a perfectly random way. This implies that any Guesser is expected to score at most $\ln(2d)$ points during this phase.

Originally, we split the domain into consecutive sets of size $\log n$ and tracks each set in a cell, where a cell maintains a bitmap, as well as some meta-data to recover the original identity of each bit. The idea now is that we use the data structure to track the set of available cards. Each cell tracks cards of (possibly) multiple mini-decks, in a way that allows us to associate each availability bit with a mini-deck. And complementary, if a mini-deck has more available cards than a cell can handle, then we can track it across multiple cells.

In the enhanced version, each cell tracks $\log d$ cards. We iteratively assign mini-decks to cells. If a cell can track what's left of a mini-deck entirely, then we just add it to the cell. Otherwise, we assign what portion of the mini-deck that does fit into the cell and continue to the next cell. We begin initializing this data structure ahead of the Final phase while ensuring worst-case constant time (say, start in stage $n - 4$ and inspect 4 mini-decks, or portions of mini-decks, in each turn). By starting early, we ensure that no mini-deck is empty, which simplifies this construction.

In terms of metadata and space, each cell now contains:

- Cards availability bitmap ($\log d$ bits) - that tracks the availability of $\log d$ cards.
- First card bitmap ($\log d$ bits) - refers to the Cards availability bitmap, and indicates whether a card is the first card of a mini-deck.
- The first mini-deck assigned to this cell ($\log d$ bits).
- The first mini-deck's offset ($O(\log d)$ bits) - for the case that some of the cards of the first mini-deck are tracked in previous cell(s).

See Figure 3 for a visual representation.

How many cells do we need? Since there are $O(d)$ available cards to track, we get that $O(d/\log d)$ cells suffice. Since each cell consumes $O(\log d)$ bits, then we are good.

Observe that during initialization, we maintain two data structures for the same purpose, so there is some bookkeeping to perform. We add mini-decks to cells sequentially, so we make sure

Figure 3: Enhanced cells for the Adaptive-Threshold Dealer. The left cell tracks mini-decks 14, 15 and the first 4 cards of mini-deck 16. It is of population 4 since it tracks a total of 4 available cards. Mini-deck 14 is the first mini-deck that begins in this cell, thus, it is a leader. The right cell tracks the 6 cards left of mini-deck 16 and the first two cards of mini-deck 17. It is of population 6. Mini-deck 17 is a leader because the first card of mini-deck 16 is tracked in the left cell.

Cards availability:	0	1	1	0	1	0	0	1	1	1	0	0	1	1	1	1
First card of the deck:	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0
First mini-deck in the cell:	14								16							
First mini-deck Offset:	0								4							

to know what is the last fully added mini-deck. When a drawable mini-deck is chosen, if it is fully tracked, we sample from it using the cells data structure. And if not, then we draw its top card and update its hole counter. Once a mini-deck is fully added, we freeze its hole counter, and use it later for decoding. We need to pay extra attention to the currently added mini-deck.

To draw a card from a tracked mini-deck during initialization, we need to recover its cell. We introduce yet another data structure for this task. We say that a mini-deck is *leading* if it is the first mini-deck that begins in a cell. So in Figure 3, mini-decks 14 and 17 are leading. We allocate a bit for each mini-deck to indicate whether it is leading or not (d bits). Additionally, for leading mini-decks, we store a pointer to their cell. So drawing a card from a leading mini-deck is easy.

Looking at the leaders bitmap, observe that every sequence of $\log d$ bits indicates at least one leading mini-deck. This is true because, during initialization, each mini-deck contains at least one available card. So we find the cell of a non-leading mini-deck by finding its leader, which requires reading $\log d$ bits from the leader’s bitmap. Since there are $O(d/\log d)$ cells, then there are at most that many leaders, so storing their pointers also fits into memory.

After initialization, that is, during the Final phase, we use the same procedure of sampling a population size, then a cell of this size, and then an available card. The “decoding” of each card’s meaning is done, as previously, by simple bit-wise and arithmetic operations.

Lemma 4.9. *An Adaptive-Threshold Dealer with m bits of memory scores $O(\ln m)$ points in expectation against any Guesser during the Final phase, and each draw runs in worst-case constant time.*

Together with Lemma 4.8, we finish the analysis of the Adaptive-Threshold Dealer.

Theorem 4.10. *An Adaptive-Threshold Dealer with m bits of memory scores $O(n/m + \ln(m))$ points in expectation against any Guesser, and each draw takes worst-case constant time.*

Time Complexity of Sampling at Random: In the literature of dynamically changing distributions, several solutions achieve expected constant time for certain families of distributions. This is usually achieved by using some form of rejection sampling, that is, sampling repeatedly until reaching some desired outcome. While none of our machinery was exceptionally complicated, it took some work to get to worst-case constant time, let alone do so in a space-efficient manner.

We do, however, rely on the ability to sample in constant time a random number from a set $[x]$ for any $x \leq 2^w$, where w is the word size in bits. This is a non-trivial ability, and without it, the expected constant time sneaks back in.

When x is a power of two, say, $x = 2^k$, this is easy, just read k random bits and the outcome is the result. But what happens when this is not the case? One common solution is to repeatedly sample the required number of bits, until we get a result within the desired range. Since the sampled range is at most twice as large as the desired range, then the stopping time is distributed geometrically, and we get that this process runs in constant time *in expectation*. Another option is to approximate the distribution by reading a large number of random bits, say $z \lceil 2^y \rceil$, and return $\lfloor z/x \rfloor$. This process yields an additive approximation error of at most 2^{-y} .

Note that Feldman et al. [FIN⁺93] studied this problem and showed that there exists a set of $\Theta(w)$ values of predefined biases, such that for any $x \leq 2^w$, it is possible to sample uniformly at random from $[x]$ given biased coin flips from this set. Despite the small number of random bits and the small number of predefined biases, their construction requires arithmetic computations that have no known efficient algorithm.

5 Tight Lower Bound on the Predictability for a Memory Size

We now provide a lower bound on the predictability of the Dealer’s permutation as a function of the memory it has. Specifically, we show that for any Dealer with m bits of memory, there is a Guesser that makes at least $\Omega(n/m + \ln(m))$ correct predictions in expectation when playing against that Dealer. We call this Guesser the *Myopic Optimizing Guesser*.

We wish to show that the entropy of the generated permutation is relatively small for any low-memory Dealer. Observe that our low-memory (m bits) Dealer (the Adaptive-Threshold Dealer described in Section 3) could draw from a set of possibilities of at most m cards in each turn. We show that this is not coincidental. We do so by presenting an encoding scheme for the series of choices made by the Dealer. In each turn, the Dealer has a set of choices from which a card is chosen, according to some probability. We utilize the fact that if the entropy of a random choice is low, then it is predictable, and a simple Guesser can guess correctly with some probability.

To provide a bound on the expected number of correct guesses, we introduce an encoding scheme for the sequence of choices that the Dealer made during the game. Our encoding scheme is made of two ingredients: We first observe that given two memory states at different turns, we can deduce which cards were drawn between them: this is an inherent property of any Dealer that draws each card exactly once. By specifying the card drawing order, we reconstruct the exact choices made by the Dealer at each turn. This allows us to fully recover the course of the game from the Dealer’s point of view.

Consider the memory state s at some turn, and assume that the Dealer can draw from a set of k cards at that turn, such that making the i th choice would lead the Dealer to state s^i at the following turn. Needless to say, $\{s^i\}_{i=1}^k$ must all be distinct, since otherwise there are two choices that lead to the same memory state, and the Dealer would not be able to tell which choice was made, thus would not know which cards still resides in the deck. It follows that if we know s , and we are given s^i , then we can tell exactly which card was drawn by the Dealer.

This argument generalizes to multiple turns. Given two memory states, s_1, s_2 at turns t_1 and t_2 (respectively), such that there is a set of choices that leads the Dealer from s_1 to s_2 , the set of cards drawn by the Dealer between t_1 and t_2 is determined. The definition of range below captures this notion.

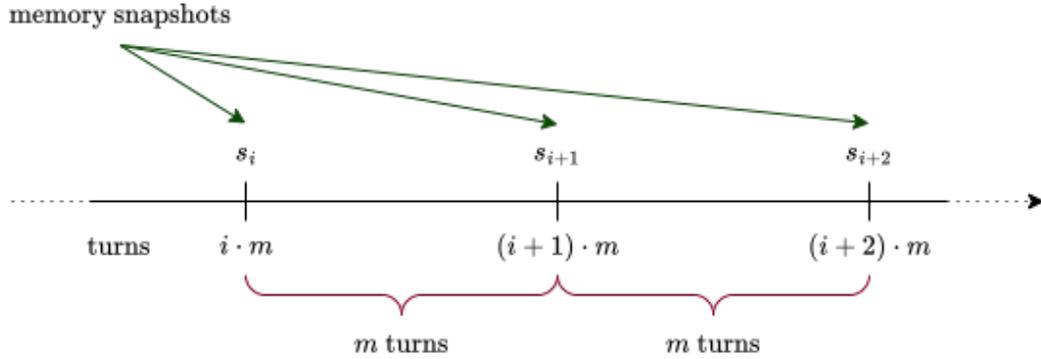
Definition 5.1 (Range). *The Range of two memory states s_1, s_2 between turns t_1 and t_2 , denoted by $\text{Range}_{t_1, t_2}(s_1, s_2)$, is the set of cards that the Dealer draws between turns t_1 and t_2 if the Dealer's memory state was s_1 at turn t_1 and s_2 at turn t_2 .*

By the discussion above, we conclude that the Range is well defined. However, while the set of cards generated between two memory states is unique, there may be multiple choice sequences that lead from one memory state to another. By specifying the drawing order, we can reconstruct the exact choices made by the Dealer at each turn.

We are interested in bounding the (un)predictability of the choices made by the Dealer. To do so, we present an encoding scheme that utilizes the above properties. The encoding scheme simulates and records a game between the Guesser and the Dealer, and stores sufficient information to recover the entire course of the game. In particular, we encode:

- Snapshots of the Dealer's memory, taken at equal intervals: to recover the Range.
- Permutations, for recovering the order of the elements in each Range.

Figure 4: Snapshots taken every m turns



Fix an encoding scheme for permutations where every permutation is represented by the same length. (see Section 1.1 or [Bon23] for a discussion).

Construction 5.2. *To encode an ordered sequence of n choices $\{k_t\}_{t=1}^n$ taken by the Dealer throughout the game, the function f_{encode} simulates and records a Dealer \mathcal{D} playing a game while making the choices $\{k_t\}_{t=1}^n$.*

For every $i \in [n/m]$, let s_i be the Dealer's memory state at turn $m \cdot i$. Let

$$R_i = \text{Range}_{m \cdot (i-1), m \cdot i}(s_{i-1}, s_i)$$

be the Range of the cards drawn during the turns $\{m \cdot (i-1), \dots, (m \cdot i) - 1\}$. Enumerate the cards of $R_i = \{c_{i,1}, \dots, c_{i,m}\}$ such that $c_{i,j} < c_{i,j+1}$. Let $\pi_i \in \mathcal{S}_m$ be the permutation describing the order in which cards from R_i are drawn; that is, the card $c_{i,j}$ is drawn at turn $m \cdot (i-1) + \pi_i(j)$.

The encoding function stores:

- n/m memory states $\{s_i\}_{i=1}^{n/m}$ ($m \cdot (n/m)$ bits),
- n/m corresponding permutations $\{\pi_i\}_{i=1}^{n/m}$ ($(m \log m) \cdot (n/m)$ bits).

Overall, the output's length is $n + n \log m$ bits.

A visual description of the snapshots taken in this construction is provided in Figure 4.

Consider the decoding process that gradually recovers choices one after the other while taking the information decoded so far into account. Clearly, this process cannot produce more information than is poured into it. Since the decoder reads $n + n \log m$ bits, we get that the amount of information gained, for any sequence, is at most $n + n \log m$ bits.

Let random variable \mathbf{K}_t be a random choice taken by the Dealer at turn t , and let $\{\mathbf{K}_t\}_{t=1}^n$ be the sequence of random choices taken by the Dealer throughout the game.

Claim 5.3. *For every Dealer with m bits of memory and every sequence $\{k_t\}_{t=1}^n$, the sum of choice entropy conditioned on past choices is at most*

$$\sum_{t=1}^n H[\mathbf{K}_t | \mathbf{K}_1 = k_1, \dots, \mathbf{K}_{t-1} = k_{t-1}] \leq n + n \log m.$$

The Myopic Optimizing Guesser: The Guesser we suggest is simply the one that at every point maximizes the probability of guessing correctly the next move by the Dealer, given the Dealer and the history so far. If there is a tie (two values that have the same probability), then the card with the smaller face value is chosen. Note that this Guesser is deterministic. Also note that this is not necessarily the best Guesser against the given Dealer since the Dealer may be influenced by the guesses it sees, so optimizing the next step does not necessarily mean optimizing the overall rate of prediction.

Lemma 2.7 states that an upper bound on the entropy of a random variable (for example, a choice) implies a lower bound on the probability mass of the heaviest element in the support. That is, whenever the entropy of a random choice is upperbounded, there is a guess that is guaranteed to be correct with probability inversely proportional to the entropy. This sets a lower bound on the expected benefit of our Myopic Guesser.

Let $\mathbf{C}_{\text{myopic},t}$ be an indicator random variable for the event that the Myopic Guesser guessed correctly at turn t . Let \mathcal{R} be the distribution over the Dealer's randomness.

Claim 5.4. *For every turn t , for every Dealer and every sequence of $t - 1$ choices $\{k_i\}_{i=1}^{t-1}$, and any number $y \geq 0$, if the entropy of the t -th choice, conditioned on past choices, is at most y , then the expected benefit of the Myopic Guesser at turn t is at least*

$$\mathbb{E}_{\mathcal{R}}[\mathbf{C}_{\text{myopic},t} | \mathbf{K}_1 = k_1, \dots, \mathbf{K}_{t-1} = k_{t-1}] \geq 2^{-y}.$$

Proof. If $H[\mathbf{K}_t | \mathbf{K}_1 = k_1, \dots, \mathbf{K}_{t-1} = k_{t-1}] \leq y$ then, from Lemma 2.7, we get that there is a choice that the Dealer takes with probability at least 2^{-y} . Thus, the Myopic Dealer predicts correctly with probability at least 2^{-y} and the result follows. \square

Theorem 5.5. *For every Dealer with m bits of memory, the Myopic Guesser scores at least $n/2m$ points in expectation.*

Proof. For any sequence of choices $\{k_t\}_{t=1}^n$, and for every turn t , denote the entropy of the t -th choice conditioned on the first $t - 1$ turns by

$$x_t = H[\mathbf{K}_t | \mathbf{K}_1 = k_1, \dots, \mathbf{K}_{t-1} = k_{t-1}],$$

and let the random variable \mathbf{X} be the entropy of a choice in a random turn conditioned on the past choices. Claim 5.3 implies that $\mathbb{E}_{\mathcal{T}}[\mathbf{X}] \leq 1 + \log m$, where \mathcal{T} is the uniform distribution over n turns.

Now consider a game played by the Myopic Guesser, and recall that in turn t , the Guesser makes a prediction based on the $t-1$ prior card draws, so if the first $t-1$ choices were k_1, \dots, k_{t-1} , then expected benefit at turn t , is $\mathbb{E}_{\mathcal{R}}[\mathbf{C}_{\text{myopic},t} | \mathbf{K}_1 = k_1, \dots, \mathbf{K}_{t-1} = k_{t-1}]$.

Fix the function $g(x) = 2^{-x}$. By linearity of expectation, we get that the expected score is at least

$$\sum_{t \in [n]} \mathbb{E}_{\mathcal{R}}[\mathbf{C}_{\text{myopic},t} | \mathbf{K}_1 = k_1, \dots, \mathbf{K}_{t-1} = k_{t-1}] \geq \sum_{t \in [n]} g(x_t) \quad (5.1)$$

$$= n \cdot \mathbb{E}_{\mathcal{F}}[g(\mathbf{X})] \quad (5.2)$$

$$\geq n \cdot g\left(\mathbb{E}_{\mathcal{F}}[\mathbf{X}]\right) \quad (5.3)$$

$$\geq n \cdot 2^{-1-\log(m)} = \frac{n}{2m}. \quad (5.4)$$

Where Inequality (5.1) follows from Claim 5.4, Equality (5.2) is true by the definition of \mathbf{X} , Inequality (5.3) follows from Jensen Inequality (see Proposition 2.8), and Inequality (5.4) follows from Claim 5.3 and the discussion above. □

6 Computationally Efficient Guessers: Open Book Dealers and Crypto to the Rescue?

The Myopic optimizing Guesser we saw in Section 5 is *not* necessarily computationally efficient *even in the case where the Dealer is efficient*: the Guesser has to find the most probable next move, and this is a computationally non-trivial task. Furthermore, if one-way functions exist, then it is a hard problem. We first point out that for any open-book Dealer there is an efficient Guesser.

Open Book Dealer: When the Dealer has no secrets (as is the case for both of our suggested Dealers), then we can make the Myopic Optimizing Guesser of the previous section efficient. Recall that the Guesser has to figure out the most probable move by the dealer, which may require a long computation time. On the other hand, finding a move whose probability is an additive approximation of the most probable move, say within $1/n$, is good enough: with this approximation, at every turn, instead of the probability of the original algorithm, we need to subtract $1/n$, and this only modifies the expected result by $n \cdot 1/n = 1$. Such a task can be performed efficiently: simply simulate the Dealer on the public state a few times ($O(n^2)$), figure out the most common response, guess that it is the most probable one, and act accordingly. Therefore, this Guesser is as efficient as the Dealer (times a polynomial in n factor).

Corollary 6.1. *For every open-book Dealer using m bits of memory, there exists a Guesser that makes at least $\frac{n}{2m} - 1$ correct guesses in expectation and operates in the same amount of time as the Dealer times a polynomial in n factor.*

6.1 Space Efficient Generation Under Computational Assumptions

Suppose that one-way functions exist and the Guesser is computationally limited (i.e. cannot break them); see Goldreich [Gol01] for background on the notions. We claim that there is a method using a small amount of secret memory to generate a hard-to-guess permutation. We will describe two methods, one simpler, but computationally more expensive, and the other one more efficient.

It is well known that the existence of one-way functions is equivalent to the existence of pseudorandom generators (PRGs), i.e. a function that maps a short seed s into a longer string where the output is indistinguishable from a truly random string function provided the seed is chosen at random. (see Goldreich [Gol01], Haitner et al. [HRV13] or Mador-Pass [MP23]). Suppose that we have a PRG $G : \{0, 1\}^k \mapsto \{0, 1\}^{2k}$ and that G does not require more than $O(k)$ bits of memory to evaluate the output. Then, we can construct from G a PRG $G' : \{0, 1\}^k \mapsto \{0, 1\}^w$ mapping a short seed s of length k into a string of length w which is polynomial in k (the value of w will be determined later) and where the amount of memory required of G' to perform the mapping is $O(k)$ bits. To see how this could be done, think of a lopsided tree of depth w where the output is in the leaves. Now, as in the famed GGM construction, we assign labels to the nodes. The root is labeled with the seed, and the left child of each internal node is labeled with the left part of the application of G to the parent label, and the right child is labeled with the right part of the application of G . We will interpret the generated string as a sequence of values in $[n]$ denoted with X_1, X_2, \dots . The permutation generated is going to be a subsequence of these values.

The output in the i th round is the next value X_j that appears for the first time. In other words, if the values so far are $X_{j_1}, X_{j_2} \dots X_{j_{i-1}}$, then the value of the i th element in the permutation is the first X_j not in the set $\{X_{j_1}, X_{j_2} \dots X_{j_{i-1}}\}$. This value can be found by regenerating the labels on the tree (from the seed) and seeing whether X_j appeared before or not. This requires just $O(k)$ bits of memory.

From the coupon collector's problem, the expected amount of work we need to do is $O(n^2 \log^2 n)$ applications of the PRG (assuming each application gives us one block in $[n]$), since after $n \ln n$ turns we expect *all* values to appear and checking whether a value appeared does not take more than the length of the sequence. This means that we can set w (the size of the tree) to be $O(n \log^2 n)$, and with a very high probability, we will not need more values than this to generate the permutation. A more careful analysis shows that the amount of work required to generate the permutation is only $O(n^2 \log n)$, since if X_j appeared before, we need, in expectation, to check only at most n previous elements.

A more efficient solution: It is well known that the existence of one-way functions is equivalent to the existence of pseudorandom functions (PRFs), i.e. collections of functions that are indistinguishable from a truly random function (see Goldreich [Gol01]). Given a PRF $F_k(x)$, it can be used to construct pseudorandom permutations (PRP) (See Luby and Rackoff [LR88]) $P_k(x) : [n] \mapsto [n]$.

The natural algorithm is then: the Dealer generated a random key k and stored it secretly. At step i it outputs $P_k(i)$. The storage requirements are the key k and the turn number, which should be much smaller than n . The result is a permutation, but is it hard to guess? If one looks at the Luby-Rackoff construction [LR88] or later ones [NR99], then the distinguishing probability is of the form $\ell^2/2^n$ (or even \sqrt{n} in the denominator), where ℓ is the number of queries the adversary has. In our case $\ell = n - 1$, so such constructions (or at least their analysis) are useless, especially towards the end of the sequence.

Instead, we need to apply the construction of pseudo-random permutations on small domains (related to format-preserving encryption) that are resilient to *any* number of queries. Such constructions are known and can be based on PRFs. The most efficient construction for small domain PRP is the ‘‘Sometimes Recurse’’ (SR) shuffle by Morris and Rogaway [MR14] (a variant of the earlier swap-or-not and mix-and-cut shuffles), which for a domain of size n runs in *expected* time of $O(\log n)$ (this is the number of applications of the PRF per call to the PRP) and worst case $O(\log^2 n)$. It is secure *even when the adversary queries the whole domain* (as will be the case in our setting).

Therefore the size of memory needed to implement the scheme is the length of the key to a PRF and the total time needed to generate a permutation on n elements is $O(n \log n)$ applications

of the PRF, with a maximum of $O(\log^2 n)$ applications of the PRF per element.

Theorem 6.2. *If one-way functions exist, then there is a closed-book Dealer that can generate a permutation where any polynomial in the security parameter time Guesser can succeed in guessing $\ln n$ plus a negligible in the security parameter many values in expectation. The amount of storage the Dealer needs is proportional to the security parameter.*

One remaining question is whether we can improve on the efficiency of the Dealer and get it down to $O(1)$ applications of the PRF per turn:

Question: Is it possible to generate a random-looking permutation with low memory and $O(1)$ calls to a PRF per element?

Another question is whether we can prove that one-way functions are necessary for such a statement, which we discuss next.

One-way functions and pseudorandomness: What are the consequences of assuming that one-way functions do not exist? Can we say that for any efficient (poly-time) Dealer there is an efficient Guesser that wins in expectation more than $\Omega(n/m + \log m)$? If we think of the myopic optimizing Guesser, then it may seem that we can approximate it by coming up with a *random inverse of the Guesser* and act similarly to the open-book case. That is given the Dealer's choices so far, find a random set of coins δ that yields its choices (i.e. choose uniformly from the set collections of coins that is consistent with its choices) and see what the next choice is; repeat this several times to get a good approximation for the most likely move.

As we know from Impagliazzo and Luby [IL89], if one-way functions do not exist, then it is possible to find a random inverse of a function. But the problem with this approach is that it works for one turn. If we want to do it again and again for each turn, then we must take into account the *complexity of the inverter*. These costs keep increasing as we compose the inverter more and more times. So it seems that this approach does not yield the desired result. This problem arises, for instance, also in the context of repeated games with limited randomness where the issue is where there is a computationally efficient way to exploit a player with limited randomness if one-way functions do not exist (see Section 5 of Hubáček, Naor and Ullman [HNU16]). We conclude with the following open problem:

Question: Is it true that that breaking the $O(n/m + \log n)$ bound efficiently is possible iff one-way functions exist?

Other Distributions

We point out that in some cases there are methods with small space for the generation of a large object. Consider, for example, the generation of a $2n$ bit string with exactly n ones and n zeroes. How much memory is needed to output such a string on-the-fly? We claim that $O(\log n)$ bits of memory suffice. At any point t in time ($1 \leq t \leq 2n$), simply store the number k_t of bits up to point t that were one. The distribution of the next bit x_i is one with probability $\frac{n-k_t}{2n-t+1}$ and zero with probability $\frac{n+k_t-t+1}{2n-t+1}$.

Question: Characterize the distributions where this low memory generation is possible.

Acknowledgments

We thank Udi Wieder and Yotam Dikstein for meaningful discussions and advice.

References

- [ABJ⁺22] Miklós Ajtai, Vladimir Braverman, T. S. Jayram, Sandeep Silwal, Alec Sun, David P. Woodruff, and Samson Zhou. The white-box adversarial data stream model. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 15–27. ACM, 2022.
- [BB08] Daniel K. Blandford and Guy E. Blelloch. Compact dictionaries for variable-length keys and data with applications. *ACM Transactions on Algorithms*, 4(2):1–25, May 2008.
- [BD92] Dave Bayer and Persi Diaconis. Trailing the dovetail shuffle to its lair. *The Annals of Applied Probability*, 2, 05 1992.
- [BHK⁺20] Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Simulating Population Protocols in Sub-Constant Time per Interaction. In *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [BKSS13] Petra Berenbrink, Kamyar Khodamoradi, Thomas Sauerwald, and Alexandre Stauffer. Balls-into-bins with nearly optimal load distribution. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '13)*, page 326–335, Montréal Québec Canada, July 2013. ACM.
- [Bon23] Boneh Dan and Cohen Bram. How to Store a Permutation Compactly <https://hackmd.io/@dabo/rkp8pcf9t>, August 2023.
- [CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory 2nd Edition*. Wiley, 2006.
- [Fei19] Uriel Feige. A randomized strategy in the mirror game. *arXiv preprint arXiv:1901.07809*, 2019.
- [FIN⁺93] D. Feldman, R. Impagliazzo, M. Naor, N. Nisan, S. Rudich, and A. Shamir. On dice and coins: Models of computation for random generation. *Information and Computation*, 104(2):159–174, 1993.
- [Gol01] O. Goldreich. *Foundations of Cryptography, Volume I – Basic Tools*. Cambridge University Press, 2001.
- [GS19] Sumegha Garg and Jon Schneider. The space complexity of mirror games. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, volume 124 of *LIPIcs*, pages 36:1–36:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [Hag91] Torben Hagerup. Fast parallel generation of random permutations. In *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, volume 510 of *Lecture Notes in Computer Science*, pages 405–416. Springer, 1991.
- [HMM93] Torben Hagerup, Kurt Mehlhorn, and James Ian Munro. Optimal algorithms for generating discrete random variables with changing distributions. *Lecture Notes in Computer Science*, 700:253–264, 1993.

- [HNU16] Pavel Hubáček, Moni Naor, and Jonathan R. Ullman. When can limited randomness be used in repeated games? *Theory Comput. Syst.*, 59(4):722–746, 2016.
- [HRV13] Iftach Haitner, Omer Reingold, and Salil P. Vadhan. Efficiency improvements in constructing pseudorandom generators from one-way functions. *SIAM J. Comput.*, 42(3):1405–1430, 2013.
- [IL89] Russell Impagliazzo and Michael Luby. One-way functions are essential for complexity based cryptography (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 230–235. IEEE Computer Society, 1989.
- [KLS23] Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. Pachash: Packed and compressed hash tables. In Gonzalo Navarro and Julian Shun, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2023, Florence, Italy, January 22-23, 2023*, pages 162–175. SIAM, 2023.
- [Knu98] Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998.
- [KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2006.
- [LR88] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988.
- [Mag24] Roey Magen. Are we still missing an item?, 2024.
- [MN22a] Roey Magen and Moni Naor. Mirror games against an open book player. In *11th International Conference on Fun with Algorithms, FUN 2022, May 30 to June 3, 2022, Island of Favignana, Sicily, Italy*, volume 226 of *LIPICs*, pages 20:1–20:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [MN22b] Boaz Menuhin and Moni Naor. Keep that card in mind: Card guessing with limited memory. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022, Berkeley, CA, USA*, volume 215 of *LIPICs*, pages 107:1–107:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [MP23] Noam Mazor and Rafael Pass. Counting unpredictable bits: A simple PRG from one-way functions. In *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part I*, volume 14369 of *Lecture Notes in Computer Science*, pages 191–218. Springer, 2023.
- [MR14] Ben Morris and Phillip Rogaway. Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In *Advances in Cryptology - EUROCRYPT 2014, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 2014.
- [MRRS12] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao S. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, June 2012.

- [MV91] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time-with applications to parallel hashing (extended abstract). In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 307–316. ACM, 1991.
- [MVN93] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 361–370. ACM/SIAM, 1993.
- [NR99] Moni Naor and Omer Reingold. On the construction of pseudorandom permutations: Luby-rackoff revisited. *J. Cryptol.*, 12(1):29–66, 1999.
- [RR93] Sanguthevar Rajasekaran and Keith W. Ross. Fast algorithms for generating discrete random variates with changing distributions. *ACM Transactions on Modeling and Computer Simulation*, 3(1):1–19, January 1993.
- [Sto23] Manuel Stoeckl. Streaming algorithms for the missing item finding problem. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 793–818. SIAM, 2023.
- [Vio12] Emanuele Viola. The complexity of distributions. *SIAM J. Comput.*, 41(1):191–218, 2012.
- [Vio20] Emanuele Viola. Sampling lower bounds: Boolean average-case and permutations. *SIAM J. Comput.*, 49(1):119–137, 2020.
- [Wie17] Udi Wieder. Hashing, Load Balancing and Multiple Choice. *Foundations and Trends® in Theoretical Computer Science*, 12(3-4):275–379, 2017.
- [Wik25] Wikipedia. Hamming weight — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Hamming%20weight&oldid=1279290156>, 2025. [Online; accessed 08-March-2025].

Appendix A Algorithmic Description of the Perfect Dealer

Algorithm A.1 To draw an element from a cell, sample a random 1-bit and toggle it off.

```

procedure DRAWELEMENTFROMCELL(cell)
  c = popCount(cell.elements)                                ▷ Count the number of elements in the cell
  r ~ {0, ..., c - 1}
  i = bitSelect(cell.elements, r)                            ▷ Get the index of the rth 1-bit
  element = (cell.j · log(n)) + i
  return element

```

Algorithm A.2 Intervals

```
procedure SAMPLEACELLOFPOPULATIONSIZE(population_size)
  ▷ Sample a random cell
   $s \leftarrow \text{interval\_size}[\text{population\_size}]$ 
   $r \sim \{0, \dots, s - 1\}$ 
   $l \leftarrow \text{interval\_beginnings}[\text{population\_size}] + r$            ▷ location of a random cell in the interval
  DRAWELEMENTFROMCELL(cells[l])

procedure DECREMENTCELLPOPULATIONSIZE( $l$ , population_size)
  ▷ Update intervals
  ▷ 1. Swap the location of  $l$  with the last index in the interval
   $b = \text{interval\_beginnings}[\text{population\_size}] + \text{interval\_size}[\text{population\_size}] - 1$ 
  cells[l]  $\leftrightarrow$  cells[b]

  ▷ 2. Update interval boundaries
  interval_size[population_size] -= 1
  interval_size[population_size-1] += 1
  interval_beginnings[population_size-1] =
    interval_beginnings[population_size] + interval_size[population_size]
```

Algorithm A.3 Urn Data Structure

```
▷ A single node in the linked list
struct Node {
  color: int           ▷ log  $k$  bits
  prev: int           ▷ log  $m$  bits
  next: int           ▷ log  $m$  bits
}
▷ The Urn data structure
struct Urn {
  nodes []Node           ▷ An array of  $m$  nodes
  size int               ▷ Number of total marbles currently in the Urn
  anchors [color]       ▷  $k$  anchors to different linked lists
}
```

Algorithm A.4 Urn Methods

```
procedure ADDMARBLE(color)
  nodes[size].color  $\leftarrow$  color
  nodes[size].prev  $\leftarrow$  Nil ▷ set the color of the last node
  head  $\leftarrow$  anchors[color]
  if head  $\neq$  Nil then ▷ if there is already a linked list for this color
    anchors[color].prev  $\leftarrow$  size
    nodes[size].next  $\leftarrow$  head
  anchors[color]  $\leftarrow$  size ▷ Set the new anchor

procedure REMOVMARBLE(color)
  prev_head  $\leftarrow$  anchors[color]
  new_head  $\leftarrow$  nodes[prev_head].next
  size  $\leftarrow$  size - 1
  nodes[prev_head]  $\leftarrow$  nodes[size] ▷ Copy the content of the last node
  ▷ Update neighbors of moved element
  if nodes[size].next  $\neq$  Nil then
    nodes[nodes[prev_head].next].prev  $\leftarrow$  prev_head
  if nodes[size].prev  $\neq$  Nil then
    nodes[nodes[prev_head].prev].next  $\leftarrow$  prev_head
  ▷ If we moved the anchor of some linked list, update the anchor table
  if nodes[prev_head].prev == Nil then
    anchors[nodes[prev_head].color]  $\leftarrow$  prev_head
  anchors[color]  $\leftarrow$  new_head
  if new_head  $\neq$  Nil then
    nodes[new_head].prev  $\leftarrow$  Nil

procedure SAMPLEMARBLE
  i  $\sim$  [size] ▷ sample a Marble uniformly at random
  return nodes[i].color
```
