Efficient Online Random Sampling via Randomness Recycling*

Thomas L. Draper† Feras A. Saad†

May 7, 2026

Abstract. This article studies the fundamental problem of using i.i.d. coin tosses from an entropy source to efficiently generate random variables $X_i \sim P_i$ $(i \geq 1)$, where $(P_1, P_2, \dots)$ is a random sequence of rational discrete probability distributions subject to an *arbitrary* stochastic process. Our method achieves an amortized expected entropy cost within $\varepsilon > 0$ bits of the information-theoretically optimal Shannon lower bound using $O(\log(1/\varepsilon))$ space. This result holds both pointwise in terms of the Shannon information content conditioned on X_i and P_i, and in expectation to obtain a rate of $\mathbb{E}[H(P_1) + \cdots + H(P_n)]/n + \varepsilon$ bits per sample as $n \to \infty$ (where H is the Shannon entropy). The combination of space, time, and entropy properties of our method improves upon the Knuth and Yao (1976) entropy-optimal algorithm and Han and Hoshi (1997) interval algorithm for online sampling, which require unbounded space. It also uses exponentially less space than the more specialized methods of Kozen and Soloviev (2022) and Shao and Wang (2025) that generate i.i.d. samples from a fixed distribution. Our online sampling algorithm rests on a powerful algorithmic technique called *randomness recycling*, which reuses a fraction of the random information consumed by a probabilistic algorithm to reduce its amortized entropy cost.

On the practical side, we develop randomness recycling techniques to accelerate a variety of prominent sampling algorithms, which include uniform sampling, inverse transform sampling, lookup-table sampling, alias sampling, and discrete distribution generating (DDG) tree sampling. We show that randomness recycling enables state-of-the-art runtime performance on the Fisher-Yates shuffle when using a cryptographically secure pseudorandom number generator, and that it reduces the entropy cost of discrete Gaussian sampling. Accompanying the manuscript is a performant software library in the C programming language.

*This manuscript appears as: Thomas L. Draper and Feras A. Saad. 2026. Efficient Online Random Sampling via Randomness Recycling. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2473–2511. doi:10.1137/1.9781611978971.89

†Computer Science Department, Carnegie Mellon University (tdraper@cmu.edu, fsaad@cmu.edu).

Contents

1 Introduction
- 1.1 Computational Model
  - 1.1.1 Entropy Cost
  - 1.1.2 Shannon's Fundamental Rate
- 1.2 Main Result
- 1.3 Existing Approaches
  - 1.3.1 Entropy-Optimal Sampling for One Distribution
  - 1.3.2 Online Entropy-Suboptimal Sampling
  - 1.3.3 Online Entropy-Optimal Sampling
  - 1.3.4 Interval Method (Arithmetic Coding)
- 1.4 Overview of Randomness Recycling

2 Random States
- 2.1 Merging and Splitting Uniform Random States
  - 2.1.1 Merging Two Uniform States into a Uniform State
  - 2.1.2 Splitting a Uniform State into Two Uniform States
- 2.2 Merging and Splitting Nonuniform Random States
  - 2.2.1 Merging a Nonuniform and Uniform State into a Uniform State
  - 2.2.2 Splitting a Uniform State into a Nonuniform and Uniform State

3 Proof of Main Theorem
- 3.1 A Recycler for Uniform Distributions
- 3.2 A Recycler for General Distributions
- 3.3 Entropy Cost of Uniform Sampling
- 3.4 Entropy Cost of General Sampling
- 3.5 Entropy Cost of Generating a Random Sequence
- 3.6 Proof of Theorem 1.5

4 Randomness Recyclers for Uniform Distributions
- 4.1 Comparison to the Lumbroso Uniform Sampler
- 4.2 Comparison to the Lemire Uniform Sampler
- 4.3 Optimized Uniform Sampling via Widening Multiplication and Batching
  - 4.3.1 Recycling with Widening Multiplication
  - 4.3.2 Uniform Sampling with Widening Multiplication and Batching
  - 4.3.3 Application to Generating Random Permutations

5 Randomness Recylers for General Distributions
- 5.1 Lookup-Table Sampling
- 5.2 Alias Sampling
- 5.3 Discrete Distribution Generating Tree Sampling
- 5.4 Benchmark Evaluations
- 5.5 Discrete Gaussian Sampling

6 Related Work

7 Remarks

A Baseline Online Random Sampling Algorithms

1 Introduction.

Let $\mathcal{X} = \{\alpha_1, \alpha_2, \ldots, \alpha_l\}$ be a finite alphabet of $l \geq 1$ distinct symbols and

$$\Delta\mathcal{X} \coloneqq \left\{ p : \mathcal{X} \to [0,1] \cap \mathbb{Q} \;\middle|\; \sum_{i=1}^{l} p(\alpha_i) = 1 \right\} \subset [0,1]^{\mathcal{X}} \tag{1.1}$$

be the set of all rational probability distributions on $\mathcal{X}$. Suppose there is a $\Delta\mathcal{X}$-valued random sequence $\boldsymbol{P} \coloneqq (P_i)_{i\geq 1}$ of probability distributions, subject to any stochastic process, that we aim to sample from. This article studies the fundamental problem of using i.i.d. unbiased coin tosses $\boldsymbol{C} \coloneqq (C_i)_{i\geq 1}$ from an entropy source to generate an output sequence $\boldsymbol{X} \coloneqq (X_i)_{i\geq 1}$ of random variables $X_i \sim P_i$, in the following setting.

Problem Statement: Online Sampling a Dynamic Sequence of Probability Distributions

Goal: Design an *online random sampling* algorithm which, given i.i.d. fair coins and access to a dynamic sequence of probability distributions, executes the following loop:

> **for** $i = 1, 2, 3, \ldots$
>
> (S1) Receive the next target distribution P_i
>
> (S2) Generate the next random variable $X_i \sim P_i$

The efficiency of online random sampling algorithm is measured in terms of its space, time, and entropy complexities, which are limited computational resources to be conserved. The random sequence $\boldsymbol{P}$ of target distributions can itself follow any law over $(\Delta\mathcal{X})^{\mathbb{N}}$, such as deterministic, i.i.d., exchangeable, stationary, Markov, etc. For example, each P_i may depend on the previous distributions $P_{<i} \coloneqq (P_j)_{j<i}$, on the previous generated variables $X_{<i}$, or on other sources of exogenous randomness or nondeterminism. The described setup is very general and covers most computational settings that involve sequentially generating discrete random variables, such as randomized algorithms, Markov chain Monte Carlo, or probabilistic programs for stochastic simulation.

1.1 Computational Model.

We formalize the space, time, and entropy complexity of online sampling algorithms that use i.i.d. coin tosses and propagate some auxiliary state that across each round of (S1) and (S2). Listing 1.1 defines an online random sampling algorithm in the special case that $\boldsymbol{P} = \boldsymbol{p}$ is a *fixed* $\Delta\mathcal{X}$-valued sequence of probability distributions. The correctness requirement (1.5) implies that when the online random sampling algorithm is instead given a *random* $\Delta\mathcal{X}$-valued sequence $\boldsymbol{P}$ over the same probability space $(\Omega, \mathcal{E}, \mathbb{P})$, its output X_i at each round $i \geq 1$ is correctly drawn from the dynamically provided target distribution

$$\mathbb{P}(X_i = x_i \mid X_{<i} = x_{<i}, P_{\leq i} = p_{\leq i}) = p_i(x_i), \tag{1.6}$$

subject to a natural independence constraint between the coin process $\boldsymbol{C}$ and distribution process $\boldsymbol{P}$:

$$P_i \perp \boldsymbol{C} \;\middle|\; X_{<i}. \tag{1.7}$$

Equation (1.7) states that P_i cannot depend on fresh coins to be consumed by the sampler in future rounds, and that it cannot depend on the internal history of the sampler other than through its generated outputs $X_{<i}$.

1.1.1 Entropy Cost.

In Listing 1.1, the counters $t_n \coloneqq v_1 + \cdots + v_n$ of consumed coin tosses define the *entropy cost* of an online sampling algorithm, i.e., the total number of tosses consumed to generate outputs $(x_1, \ldots, x_n)$. For a probabilistic run of online random sampling with random coin and distribution sequences $(\boldsymbol{C}, \boldsymbol{P})$, we notate the random entropy cost explicitly as $T_n \equiv t_n(\boldsymbol{C}, \boldsymbol{P})$. For a probabilistic run where the distribution sequence $\boldsymbol{p}$ is fixed, we write the entropy cost $t_n(\boldsymbol{C}, \boldsymbol{p})$. For a fixed distribution sequence $\boldsymbol{p}$, the *expected entropy cost* $\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{p})\right]$ is the entropy cost averaged over random coin tosses.

1.1.2 Shannon's Fundamental Rate.

Suppose momentarily that $\boldsymbol{p}$ is a fixed distribution sequence and that we aim to generate an independent output sequence $\boldsymbol{X} = (X_i \sim p_i)_{i\geq 1}$. Knuth and Yao (1976; Theorem 2.2) show that the expected number of coin tosses $t_n(\boldsymbol{C}, \boldsymbol{p})$ needed to generate $(X_1, \ldots, X_n)$, conditioned on any positive probability event $\{X_1 = x_1, \ldots, X_n = x_n\}$ is at least the Shannon *information content* (or surprisal) of

Listing 1.1: Formal specification of an online random sampling algorithm.

An *online random sampling algorithm* over a finite alphabet $\mathcal{X}$ and a countable auxiliary state space $\mathcal{S}$ with initial state $s_0 \in \mathcal{S}$ is a partial computable function

$$f : \{0,1\}^{\mathbb{N}} \times \Delta\mathcal{X} \times \mathcal{S} \rightharpoonup \mathcal{X} \times \mathcal{S}. \tag{1.2}$$

For each $(p, s) \in \Delta\mathcal{X} \times \mathcal{S}$, the set $\{\boldsymbol{c} \mid (\boldsymbol{c}, p, s) \notin \mathrm{dom}(f)\}$ of "non-halting" streams $\boldsymbol{c}$ has Lebesgue measure zero (i.e., f halts almost everywhere in its first argument). Online sampling operates as follows.

- **Per-Round Map.** Let $\boldsymbol{p} = (p_i)_{i\geq 1}$ be a sequence of target distributions and $\boldsymbol{c} = (c_i)_{i\geq 1}$ a sequence of coin tosses. For each round $i \geq 1$, the ith output symbol x_i and updated auxiliary state s_i at the end of the round i are obtained by iterating

  $$(x_i, s_i) \coloneqq f(\boldsymbol{c}_{>t_{i-1}}, p_i, s_{i-1}) \qquad (i = 1, 2, \dots). \tag{1.3}$$

  Here, $t_{i-1} \coloneqq v_1 + \cdots + v_{i-1}$ is the total number coin tosses consumed in rounds 1 through $i-1$ and $\boldsymbol{c}_{>t_{i-1}} \coloneqq (c_j)_{j>t_{i-1}}$ is the suffix of $\boldsymbol{c}$, which contains only fresh tosses. Formally, the number $v \geq 0$ of coin tosses consumed on a given input $(\boldsymbol{c}, p, s) \in \mathrm{dom}(f)$ is the smallest number such that $f(\boldsymbol{c}, p, s) = f(\boldsymbol{c}', p, s)$ for every coin toss sequence $\boldsymbol{c}'$ whose first v values agree with those of $\boldsymbol{c}$.

- **Probabilistic Run.** Let $\boldsymbol{C} \sim \mathrm{Uniform}(\{0,1\}^{\mathbb{N}})$ be an i.i.d. sequence of coin tosses which lives on a standard probability space $(\Omega, \mathcal{E}, \mathbb{P})$. Let $S_0 = s_0$ be a degenerate random variable. The random output symbol sequence $\boldsymbol{X} \coloneqq (X_i)_{i\geq 1}$ and auxiliary state sequence $\boldsymbol{S} \coloneqq (S_i)_{i\geq 1}$ are obtained by iterating

  $$(X_i, S_i) \coloneqq f(\boldsymbol{C}_{>T_{i-1}}, p_i, S_{i-1}) \qquad (i = 1, 2, \dots). \tag{1.4}$$

  where $T_{i-1} = V_0 + \cdots + V_{i-1}$ is now the random total number of coin tosses consumed so far.

- **Correctness Requirement.** For any fixed sequence of target distributions $\boldsymbol{p} \coloneqq (p_i)_{i\geq 1}$, the random output sequence $\boldsymbol{X}$ from a probabilistic run must satisfy

  $$\mathbb{P}(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^{n} p_i(x_i) \qquad (n \geq 1;\ x_1, \dots, x_n \in \mathcal{X}). \tag{1.5}$$

the observed symbols, which implies the following fundamental lower bounds for every integer $n \geq 1$:

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{p}) \mid X_1 = x_1, \dots, X_n = x_n\right] \geq \sum_{i=1}^{n} \log\left(\frac{1}{p_i(x_i)}\right), \tag{1.8}$$

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{p})\right] \geq \sum_{i=1}^{n} H(p_i) \qquad \text{where } H(p) \coloneqq \sum_{\alpha\in\mathcal{X}} \log\left(\frac{1}{p(\alpha)}\right) p(\alpha). \tag{1.9}$$

Equation (1.9) states that the expected number of coin tosses is lower bounded by the sum of *Shannon entropies* $H(p)$, which comports with the source coding theorem of Shannon (1948).

Complexity. The rates (1.8) and (1.9) capture entropy-cost lower bounds of online random sampling using i.i.d. coin tosses in terms of information content and Shannon entropy. The entropy complexity of an online sampling algorithm is defined as its entropy cost $t_n(\cdot)$. Its space complexity is captured by the auxiliary states $\boldsymbol{S} = (S_i)_{i\geq 1}$ propagated across rounds, and any temporary space used by f within each round. Its (expected) time complexity is that of the function f. We seek space- and time-efficient online sampling algorithms whose amortized entropy cost $\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P})\right]/n$ as n grows large is as close as possible to the information-theoretic lower bound. We assume each target distribution p_i is given either as a (possibly sparse) list of integer weights $(w_1, \dots, w_l)$ where $p_i(\alpha_k) = w_k/(w_1 + \cdots + w_l)$, or as an efficiently computable cumulative distribution function $k \mapsto \sum_{j\leq k} p_i(\alpha_j)$.

Remark 1.1. We will reserve the notation $H(p)$ to denote the entropy of a probability distribution $p \in \Delta\mathcal{X}$, and instead use square braces for the entropy of a random variable, i.e., $H[X] \coloneqq -\sum_{\alpha\in\mathcal{X}} \log(\mathbb{P}(X=\alpha))\mathbb{P}(X=\alpha)$.

Remark 1.2. For a random distribution P, the entropy $H(P) \coloneqq \omega \mapsto H(P(\omega))$ is itself a random variable, which is of course distinct from the entropy $H[P] = -\sum_{p\in\Delta\mathcal{X}} \log(\mathbb{P}(P=p))\mathbb{P}(P=p)$ of the random element P (an irrelevant quantity in this article).

Remark 1.3. When applying the bounds from (1.8) and (1.9) to a random distribution sequence $\boldsymbol{P}$, it is essential to consider the surprisals and entropies of the random distributions $(P_1, \dots, P_n)$, and *not* the joint entropy $H[X_1, \dots, X_n]$ of the output sequence. This distinction is subtle, but obvious from a simple example. Suppose $P_1 = \delta_{\alpha_1}$ or $P_1 = \delta_{\alpha_2}$ with equal probability, i.e., P_1 is always realized as a degenerate distribution. Any sampling algorithm can generate $X_1 \sim P_1$ using zero coin tosses for any realization of P_1, whereas $H[X_1] = 1$ bit.

Remark 1.4. A common case for online sampling is generating an output sequence $\boldsymbol{X} \coloneqq (X_i)_{i\geq 1}$ subject to an arbitrary stochastic process whose distribution is not influenced by any external randomness. In particular, each random distribution P_i is itself fully determined by $X_{<i}$, allowing us to write $\boldsymbol{P} = \{p_i(\,\cdot \mid X_1, \dots, X_{i-1})\}_{i\geq 1}$, where $p_i(\,\cdot \mid X_1, \dots, X_{i-1}) \coloneqq \mathbb{P}(X_i = \cdot \mid X_1, \dots, X_{i-1})$ is the conditional distribution of X_i given all previously generated outputs for $i > 1$. In this special case, the bound (1.9) on the expected entropy cost of generating $(X_1, \dots, X_n)$ becomes the more familiar entropy of the output sequence:

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P})\right] \geq \mathbb{E}\left[\sum_{i=1}^{n} H(p_i(\,\cdot \mid X_1, \dots, X_{i-1}))\right] = H[X_1, \dots, X_n]. \tag{1.10}$$

1.2 Main Result. The main contribution of this article is an online random sampling algorithm whose amortized entropy cost can be made arbitrarily close to the Shannon bound in the limit, using bounded auxiliary space across rounds. Recalling the lower bound (1.8) from Knuth and Yao (1976; Theorem 2.2), the precise result on entropy and space complexity is as follows, where $\Delta\mathcal{X}_d \coloneqq \{p \in \Delta\mathcal{X} \mid d \geq \min_k\{k \in \mathbb{N} \mid \forall \alpha \in \mathcal{X}.\ kp(\alpha) \in \mathbb{N}\}\}$ is the set of rational probability distributions over $\mathcal{X}$ whose probabilities have common denominator at most d.

THEOREM 1.5. *For any $\varepsilon > 0$ and $d \geq 1$, there exists an online random sampling algorithm using a sequence $\boldsymbol{C}$ of i.i.d. coin tosses such that, for every distribution sequence $\boldsymbol{p} \in (\Delta\mathcal{X}_d)^{\mathbb{N}}$, the entropy cost of generating an output sequence $\boldsymbol{X} = (X_i \sim p_i)_{i\geq 1}$ satisfies*

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{p}) \mid X_1 = x_1, \dots, X_n = x_n\right] < \sum_{i=1}^{n} \log\left(\frac{1}{p_i(x_i)}\right) + \varepsilon n + W_{d,\varepsilon} \qquad (n \geq 1), \tag{1.11}$$

where $\{X_1 = x_1, \dots, X_n = x_n\}$ is any positive probability event and $W_{d,\varepsilon} \sim \log(d/\varepsilon)$ as $d/\varepsilon \to \infty$. The algorithm uses auxiliary space of at most $2W_{d,\varepsilon}$ bits across rounds and $O(W_{d,\varepsilon})$ temporary space per round.

This strong result establishes an online sampling algorithm whose entropy cost is ε-close to the information-content lower bound (1.8) for any n, *uniformly in the distribution sequence* $\boldsymbol{p}$. In particular, Theorem 1.5 implies the expected entropy cost of the online sampling algorithm is close to the Shannon-entropy lower bound (1.9):

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{p})\right] < \sum_{i=1}^{n} H(p_i) + \varepsilon n + W_{d,\varepsilon} \qquad (n \geq 1). \tag{1.12}$$

Equations (1.11) and (1.12) are the analogues of (1.8) and (1.9) when the target sequence $\boldsymbol{p}$ is fixed. If the distribution sequence $\boldsymbol{P}$ follows an arbitrary $\Delta\mathcal{X}_d$-valued stochastic process subject to (1.7), then Theorem 1.5 also implies the following analogues of (1.8) and (1.9):

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P}) \mid X_{\leq n} = x_{\leq n}, P_{\leq n} = p_{\leq n}\right] < \sum_{i=1}^{n} \log\left(\frac{1}{p_i(x_i)}\right) + \varepsilon n + W_{d,\varepsilon} \qquad (n \geq 1), \tag{1.13}$$

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P})\right] < \sum_{i=1}^{n} \mathbb{E}\left[H(P_i)\right] + \varepsilon n + W_{d,\varepsilon} \qquad (n \geq 1). \tag{1.14}$$

We conjecture that the space-entropy combination in Theorem 1.5 is the best possible, up to constant factors and dependence on d.

CONJECTURE 1.6. *Any online random sampling algorithm that generates exact samples from a sequence of arbitrary discrete distributions, within $\varepsilon > 0$ of the information-theoretically optimal entropy rate, using a stream of i.i.d. coin tosses as the entropy source, requires $\Omega(\log(1/\varepsilon))$ bits of space for an auxiliary state that is carried over between rounds.*

For the conditional distribution sequence considered in Remark 1.4, (1.14) gives the following corollary.

COROLLARY 1.7. *Let $\boldsymbol{P} \coloneqq \{p_i(\cdot \,|\, X_1, \ldots, X_{i-1})\}_{i\geq 1}$ denote a sequence of conditional distributions for random variables $\boldsymbol{X} \coloneqq (X_i)_{i\geq 1}$, whose probabilities have common denominator $d \geq 1$. For any $\varepsilon > 0$, there exists an online random sampling algorithm that generates $\boldsymbol{X}$ whose entropy cost satisfies*

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P})\right] < H[X_1, \ldots, X_n] + \varepsilon n + W_{d,\varepsilon} \qquad (n \geq 1). \tag{1.15}$$

Remark 1.8. In the special case of an i.i.d. sequence with common distribution P, Theorem 1.5 furnishes, for any $\varepsilon > 0$, an algorithm with amortized entropy cost asymptotically less than $H(P) + \varepsilon$ tosses per output.

The algorithm witnessing Theorem 1.5 enjoys several desirable theoretical properties.

- It works in an online setting, where the distributions $\boldsymbol{P}$ are given sequentially and follow a random process.
- It guarantees an $\varepsilon > 0$ bound on the amortized entropy consumption using $O(\log(1/\varepsilon))$ space, which remains bounded as the number n of generated samples grows. The methods of Knuth and Yao (1976); Han and Hoshi (1997) achieve $\varepsilon = 0$ entropy loss; however, they require unbounded memory even for distributions in $\Delta\mathcal{X}_d$, i.e., distributions whose probabilities have common denominator bounded by $d \geq 1$.
- Compared to the finite-memory implementation of the Han and Hoshi (1997) interval (arithmetic coding) method from Uyematsu and Li (2003), our method
  - produces exact samples from the target distributions, not approximate samples;
  - uses efficient integer arithmetic, not expensive arbitrary-precision arithmetic.
- Compared to the more specialized sampling algorithms of Kozen and Soloviev (2022) and Shao and Wang (2025) that sample a *fixed* i.i.d. sequence $\boldsymbol{P}$, our method obtains an exponential improvement, using $O(\log(1/\varepsilon))$ rather than $O(1/\varepsilon)$ space to achieve a desired $\varepsilon > 0$ entropy bound.

On the practical side, we show that our method delivers the following improvements.

- It enables speedups to generating random permutations using the Fisher and Yates (1953) shuffle, compared to the state-of-the-art method of Brackett-Rozinsky and Lemire (2025) (Figure 4.4).
- It enables speedups to several prominent sampling algorithms for discrete probability distributions (Figures 4.1, 4.3 and 5.4) and reduces the entropy cost of discrete Gaussian sampling (Figure 5.5).

1.3 Existing Approaches. To give further context to our main result, we discuss the strengths and limitations of existing online random sampling algorithms. For this discussion it is sufficient to focus on the setting from Remark 1.4 and Corollary 1.7, where $\boldsymbol{P} = \{p_i(\cdot \,|\, X_1, \ldots, X_{i-1})\}_{i\geq 1}$ is a random sequence of conditional distributions that are determined solely by the previously generated outputs. Table 1.1 summarizes the entropy loss and space and time complexity of existing baseline algorithms as well as our method (refer to §A for technical details and concrete implementations of the baselines).

1.3.1 Entropy-Optimal Sampling for One Distribution. For generating a single discrete random variable $X \sim p$, Knuth and Yao (1976) describe an "entropy-optimal" algorithm with expected entropy cost

$$\mathbb{E}\left[t_1(\boldsymbol{C}, p)\right] = \sum_{\alpha\in\mathcal{X}} \sum_{i=0}^{\infty} (\lfloor 2^i p(\alpha) \rfloor \bmod 2) i 2^{-i} < H[X] + 2, \tag{1.16}$$

based on the binary expansions of the $p(\alpha)$, which is the best possible (Knuth and Yao, 1976; Theorem 2.2). This result shows that the Shannon rate of $H[X]$ tosses per output may not be achievable for a single output. (The

Table 1.1: Comparison of online random sampling algorithms for generating a sequence $(X_1, \ldots, X_n)$ of $n \geq 1$ discrete random variables, given conditional distributions $p_i(\cdot \,|\, X_1, \ldots, X_{i-1})$ over K outcomes whose probabilities have common denominator at most d. The online optimal method and interval method have an entropy loss that tends to zero as $n \to \infty$, but their worst-case space complexity is unbounded, even for fixed n; furthermore, the expected space complexity grows with n and is thus also unbounded as $n \to \infty$. The online suboptimal method uses bounded space but its amortized entropy loss is arbitrary close to 2 bits in the worst case. Randomness recycling achieves an entropy loss of $\varepsilon > 0$ bits as $n \to \infty$, using bounded space of $O(\log(d/\varepsilon))$ bits.

	Online Suboptimal Method (Knuth and Yao, 1976; Algorithm A1)
Amortized Entropy Loss	< 2
Time Complexity	$O\left(nK \log(d) \log(K)\right)$
Space Complexity	$O\left(K \log(d)\right)$
	Online Optimal Method (Knuth and Yao, 1976; Algorithm A2)
Amortized Entropy Loss	$< 2/n$
Time Complexity	$O\left(n^2 K \log(d) \log(K) + n^3 K \log^2(d) + n^3 \log(n) \log^2(d)\right)$
Space Complexity	$O\left(nK \log(d)\right)$ (Expected)
	Interval Method (Han and Hoshi, 1997; Algorithm A3)
Amortized Entropy Loss	$< 3/n$
Time Complexity	$O\left(nK \log(d) + n^2 \log^2(d) \log(K)\right)$
Space Complexity	$O\left((n + K) \log(d)\right)$ (Expected)
	Randomness Recycling (Table 1.2; Algorithm 3.3)
Amortized Entropy Loss	$< \varepsilon + O(\log(d/\varepsilon))/n$
Time Complexity	$O\left(nK \log(d) + n \log(d) \log(d/\varepsilon)\right)$
Space Complexity	$O\left(\log(d/\varepsilon)\right)$

expression $\mathbb{E}\left[t_1(\boldsymbol{C}, p)\right]$ is a minor abuse of notation: it indicates the expected entropy cost of generating X_1 in the first round of online sampling, and so only one distribution p is notated instead of a sequence $\boldsymbol{p}$.)

The usual way to implement the Knuth and Yao method is to explicitly construct an entropy-optimal "discrete distribution generating" (DDG) tree for p as a preprocessing step (e.g., Roy et al. (2013; Algorithm 1); Saad et al. (2020b; Algorithms 5 and 6)), which can require exponential space in the number of bits needed to encode p (Saad et al., 2020b; Theorem 3.6). Alternatively, using the cumulative distribution function of p, Saad and Lee (2025; Algorithm 1) give a linear-space entropy-optimal algorithm that avoids preprocessing a DDG tree. However, this method incurs a runtime overhead of computing binary expansions of target probabilities during sampling.

1.3.2 Online Entropy-Suboptimal Sampling. For the problem of generating an output sequence given by conditional distributions specified in an online fashion, the most straightforward application of §1.3.1 is to use a sequence of entropy-optimal samplers that target each conditional distribution:

$$X_1 \overset{\text{KY}}{\sim} p_1, \qquad X_2 \overset{\text{KY}}{\sim} p_2(\cdot \,|\, X_1), \qquad \ldots, \qquad X_n \overset{\text{KY}}{\sim} p_n(\cdot \,|\, X_1, \ldots, X_{n-1}). \tag{1.17}$$

The expected entropy cost of generating $(X_1, \ldots, X_n)$ using this dynamic approach satisfies

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P})\right] < \sum_{i=1}^{n} \left(H[X_i \mid X_1, \ldots, X_{i-1}] + 2\right) = H[X_1, \ldots, X_n] + 2n. \tag{1.18}$$

The space complexity remains bounded as n grows large, since at each step the conditional distribution $p_i(x_i \mid X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$ is finitely encodable using at most $d\lceil \log d \rceil$ bits. The space complexity of generating each X_i may be exponential or linear as described before, depending on the amount of preprocessing.

The term $2n$ in (1.18) indicates a significant waste of random bits, and the cost bound $H[X_1, \ldots, X_n]/n + 2$ does not tend to the optimal rate. Devroye and Gravel (2020; §2.3) describe a "randomness extraction" procedure that recycles the random bits in this sequential approach using the interval method of Han and Hoshi (1997)

(described in the next paragraph), which claims (cf. Footnote 1) to achieve

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P})\right] < H[X_1, \ldots, X_n] + 2 + o(n). \tag{1.19}$$

However, the space complexity is no longer bounded as the algorithm requires arbitrary-precision arithmetic. Randomness extraction uses operations with big integers of unbounded size as $n \to \infty$ *even if* the rational probabilities have bounded denominator d, compromising performance.

1.3.3 Online Entropy-Optimal Sampling. The entropy loss of < 2 bits per sample in (1.18) arises because each X_i is treated completely separately; batched sampling or storing more information across rounds can drive this loss to zero. Consider first an offline setting where n is a prespecified value, and each P_i is a deterministic function of the previous outputs $X_{<i}$ as in Remark 1.4. Then we can generate $(X_1, \ldots, X_n)$ by first reconstructing the joint distribution $p^{(n)}$ from the conditionals and then sampling

$$(X_1, \ldots, X_n) \overset{\text{KY}}{\sim} p^{(n)}(x_1, \ldots, x_n) \coloneqq \prod_{i=1}^{n} p_i(x_i \mid x_1, \ldots, x_{i-1}), \tag{1.20}$$

where $X \overset{\text{KY}}{\sim} p$ means that "X is generated from p using the Knuth and Yao method" as described in §1.3.1. As $p^{(n)}$ is a bona fide discrete distribution over $\mathcal{X}^n$, the expected entropy cost

$$\mathbb{E}\left[t_1(\boldsymbol{C}, p^{(n)})\right] < H[X_1, \ldots, X_n] + 2 \tag{1.21}$$

achieves the best possible rate for each batch size n. The expected amortized entropy cost $H[X_1, \ldots, X_n]/n + 2/n$ tends to the Shannon-optimal rate of $H[X_1, \ldots, X_n]/n$ tosses per output as $n \to \infty$. The disadvantage is that this method is offline. Further, constructing the joint distributions $p^{(n)}$ requires exponential space (up to $|\mathcal{X}|^n$) and its probabilities can have a large denominator (up to d^n).

Fortunately, Knuth and Yao (1976; p. 384) show that it is possible to entropy-optimally sample from $p^{(n)}$ in an online fashion, without explicitly constructing a DDG tree or even the joint distribution. Knuth and Yao show that their tree representations for entropy-optimal samplers can be dynamically refined step-by-step, as new conditional distributions are specified. In fact, only the leaves corresponding to the previously sampled symbol need to be refined, so this online refinement algorithm avoids the exponential complexity of the naïve batching method (1.21). This method achieves the absolute minimum expected entropy cost over all algorithms. However, the space and runtime complexity per sample grow unbounded with the number of samples, because the algorithm requires the binary expansions of rational numbers with denominators growing as d^n after n samples. This same issue arises with other exact online samplers that achieve an $O(1/n)$ entropy loss per sample.

1.3.4 Interval Method (Arithmetic Coding). The interval algorithm of Han and Hoshi (1997) generates samples from a discrete distribution by recursively subdividing the unit interval, analogously to arithmetic coding (MacKay, 2003; §6.3). It extends to sampling i.i.d. random variables, a stationary homogeneous Markov process, or an arbitrary process (Han and Hoshi, 1997; §V). When the entropy source emits fair coin tosses, Han and Hoshi (1997; Theorem 3) show that generating $(X_1, \ldots, X_n)$ using the interval method has

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{P})\right] < H[X_1, \ldots, X_n] + 3. \tag{1.22}$$

The interval method does not require batching and is suitable for online sampling. Its cost bound $H[X_1, \ldots, X_n]/n + 3/n$ also converges to the asymptotically optimal rate (Han and Hoshi, 1997; Eq. (5.2)). The disadvantage is that it requires arbitrary-precision arithmetic, as with the method of Devroye and Gravel (2020) and the online optimal method of Knuth and Yao (1976). Uyematsu and Li (2003) give a finite-space implementation of the interval method that uses the usual arithmetic coding technique of rounding cumulative probabilities, but this version does not return exact samples.

1.4 Overview of Randomness Recycling. The entropy bound in Theorem 1.5 is enabled by a technique called *randomness recycling*. The key idea is as follows. The auxiliary-state space $\mathcal{S} \coloneqq \{(z, m) \mid m \geq 1, 0 \leq z < m\}$ of our online sampling algorithm is comprised of pairs denoting a uniform draw z over $\{0, \ldots, m-1\}$, with initial state $s_0 = (0, 1)$. When sampling $X_i \sim P_i$ at round i, the sampler is given fresh coin tosses $\boldsymbol{C}_{\geq T_{i-1}}$ and the

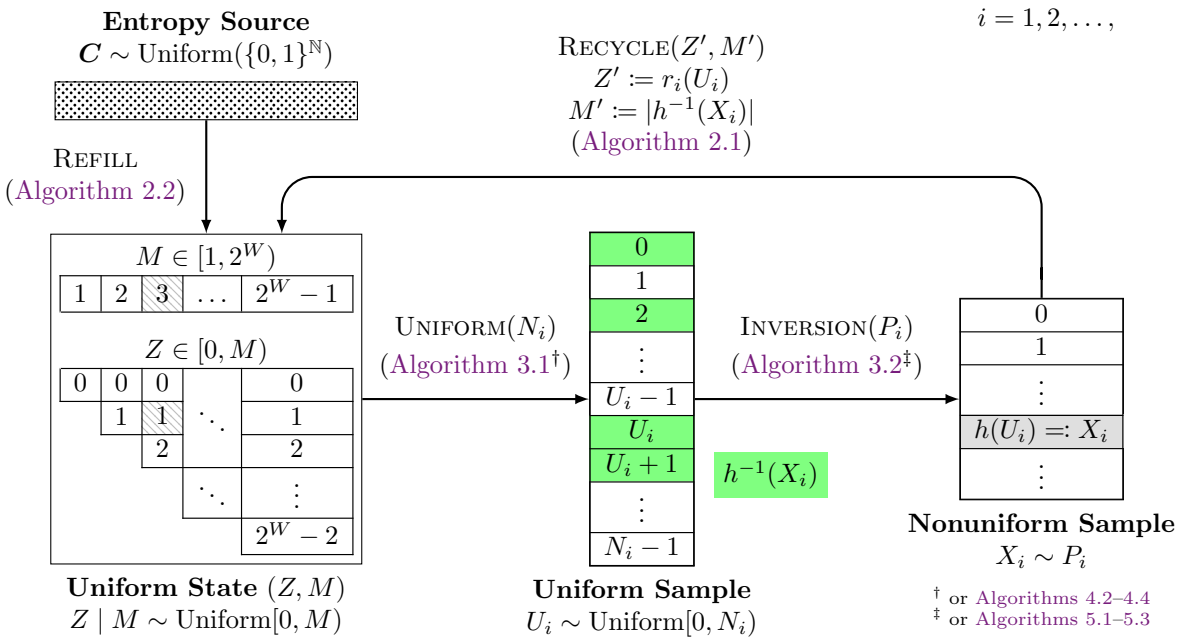


Figure 1.1: Online random sampling using randomness recycling. The sampling algorithm is dynamically given a random sequence $\boldsymbol{P} := (P_i)_{i \geq 1}$ of probability distributions and access to i.i.d. coin tosses $\boldsymbol{C} := (C_i)_{i \geq 1}$, and generates output sequence $\boldsymbol{X} = (X_i)_{i \geq 1}$ such that $X_i \sim P_i$ for $i \geq 1$.

uniform state $S_{i-1} = (Z_{i-1}, M_{i-1})$, which is a pair of random variables such that $Z_{i-1} \mid M_{i-1} \sim \text{Uniform}[0, M_{i-1})$. (We only consider discrete random variables, so we write $\text{Uniform}[0, M)$ as shorthand for $\text{Uniform}(\{0, \ldots, M-1\})$.) This uniform state incorporates fresh coin tosses drawn from the entropy source and randomness that has been recycled from generating random variables $X_1, X_2, \ldots, X_{i-1}$ in previous rounds. After X_i is generated, $S_{i-1} = (Z_{i-1}, M_{i-1})$ is updated to value $S_i = (Z_i, M_i)$ such that $Z_i \mid M_i \sim \text{Uniform}[0, M_i)$. This auxiliary state will be used to sample $X_{i+1} \sim P_{i+1}$ in the next round, in accordance with the per-round map specification (1.3).

To meet the correctness requirement (1.5), randomness recycling requires the following invariant[1] on the auxiliary state to be maintained at all rounds:

(I) For all $i \geq 1$, the global uniform state (Z_i, M_i) satisfies $Z_i \perp (X_1, \ldots, X_i) \mid M_i$.

Maintaining this global uniform state makes it possible to substantially reduce the number fresh coin tosses drawn from the entropy source, by storing unused information from previous coin tosses whenever possible. By induction, the invariant (I) is maintained as long as $(Z_0, M_0) = (0, 1)$ and the sampling function used at each step i fulfills the following contract:

(C) For all $i \geq 1$, if the initial global uniform state (Z_{i-1}, M_{i-1}) satisfies $Z_{i-1} \perp (X_1, \ldots, X_{i-1}) \mid M_{i-1}$, then the updated global uniform state (Z_i, M_i) satisfies $Z_i \perp (X_1, \ldots, X_i) \mid M_i$.

Workflow. Figure 1.1 gives an overview of the "randomness recycling" workflow for generating a random sequence $(X_i \sim P_i)_{i \geq 1}$. The entropy source provides an i.i.d. sequence $\boldsymbol{C}$ of unbiased coin tosses. These coin tosses are used to create and refill a global uniform state (Z, M) so that $Z \mid M \sim \text{Uniform}[0, M)$. To sample $X_i \sim P_i$, a discrete uniform $U_i \sim \text{Uniform}[0, N_i)$ over an appropriate range is sampled via Algorithm 3.1, which

[1] Careful reasoning about this invariant is required for generating correct samples according to (1.5). For example, the randomness extraction method of Devroye and Gravel (2020; §2.3, Algorithms 4 and 5) does not maintain an analogous invariant in its implementation of the recycling rule from the Han and Hoshi (1997) interval method. As a result, it does not produce samples from the correct target distributions. For example, given target distributions $P_1 = P_2 = \text{Bernoulli}(1/4)$ for X_1 and X_2 (i.i.d.), Devroye and Gravel (2020; Algorithm 5) can (under certain configurations) produce $X_2 \sim \text{Bernoulli}(5/16)$, because the *value* of a recycled bit is correlated with its *availability* at a point in time.

Table 1.2: Randomness recycling strategies developed in this work for uniform and general sampling algorithms.

Algorithm	Reference
Uniform Distributions	
Division Method	Algorithm 3.1
Machine Word Division Method	Algorithm 4.2
Lemire (2019) Method	Algorithm 4.3
Brackett-Rozinsky and Lemire (2025) Method	Algorithm 4.4
General Distributions	
Inversion Sampling	Algorithm 3.2
Lookup-Table Sampling (Devroye, 1986)	Algorithm 5.1
Alias Sampling (Walker, 1977)	Algorithm 5.2
DDG Sampling (Knuth and Yao, 1976; Saad et al., 2020a)	Algorithm 5.3

accesses and possibly refills the uniform state (Z, M). The output sample $X_i \leftarrow h(U_i)$ is a deterministic function of U_i, where h depends on the specific sampling algorithm used (e.g., Algorithms 3.2 and 5.1–5.3). The map h ensures that, conditioned on $\{X_i = x_i\}$, all elements of the preimage $h^{-1}(x_i) \coloneqq \{u \in [0, N_i) \mid f_i(u) = x_i\}$ are equally likely. We thus extract a uniform state (Z', M') where $M' = |h^{-1}(x_i)|$ is the cardinality of the preimage and $Z' \coloneqq r(U_i) \in [0, M')$ is obtained using a *randomness recycling rule* for h that guarantees $Z' \perp X_i \mid M'$. The final step is to call Recycle (Algorithm 2.1) to merge (Z', M') into the global uniform state (Z, M).

Organization. §2 introduces uniform and nonuniform random states and describes how they can be manipulated into new random states through information-preserving transformations. §3 proves Theorem 1.5, by demonstrating an online random sampling algorithm that leverages randomness recycling. §4 and §5 investigate further applications of randomness recycling for accelerating samplers for uniform and general discrete distributions (Table 1.2). §6 discusses related work. §7 concludes the paper with remarks and a conjecture.

2 Random States.

We first describe the auxiliary-state space $\mathcal{S} = \{(z, m) \mid m \geq 1, 0 \leq z < m\}$ used by our online sampling algorithm, as well as information-preserving operations on the elements of this space.

Definition 2.1. *A uniform random state (Z, M) is any pair of discrete random variables such that $M \geq 1$ and $Z \mid M \sim \text{Uniform}[0, M)$.*

Definition 2.2. *A nonuniform random state $(Z, \mathbf{W})$ is a pair of discrete random variables such that $\mathbf{W} \coloneqq (W_0, \ldots, W_{n-1})$ is a list of $n \geq 1$ positive random variables and $Z \mid \mathbf{W} \sim \text{Discrete}(\mathbf{W})$, i.e., $\mathbb{P}(Z = i \mid \mathbf{W}) = W_i / \sum_{j=0}^{n-1} W_j$ for $i \in [0, n)$.*

Randomness recycling leverages several properties about merging and splitting both uniform (§2.1) and nonuniform §2.2 random states. Merging and splitting are information-preserving transformations (bijections) of random variables that perfectly invert one another. The general idea is that is if X is a discrete random element with probability mass function p_X and f is a bijection, the transformation $Y = f(X)$ preserves Shannon information content (surprisals) in the sense that for all (x, y) such that $y = f(x)$, we have

$$p_Y(y) = p_X(x) \implies \log\left(\frac{1}{p_Y(y)}\right) = \log\left(\frac{1}{p_X(x)}\right). \tag{2.1}$$

2.1 Merging and Splitting Uniform Random States.

2.1.1 Merging Two Uniform States into a Uniform State.

We first describe how to *merge* two uniform states (Z, M) and (Z', M') to obtain a new uniform state over a larger range MM'.

Proposition 2.3. *For any integers $m, m' \geq 1$,*

$$Z \sim \text{Uniform}[0, m), \quad Z' \sim \text{Uniform}[0, m'), \quad Z \perp Z' \implies Z + Z'm \sim \text{Uniform}[0, mm').$$

Proof. The map $[0, m) \times [0, m') \ni (x, x') \mapsto x + x'm \in \{0, 1, \ldots, mm' - 1\}$ is a bijection, so each outcome is equally likely with probability $1/(mm')$. ∎

Algorithm 2.1 Recycling a uniform state into the global uniform state

1: ▷ *Global variables: word size W; uniform state $S = (Z, M)$*
2: **int** $W \leftarrow W_{d,\varepsilon}$ ▷ *cf.* (3.28)
3: **mutable int** $Z \leftarrow 0$ ▷ *W-bit integer*
4: **mutable int** $M \leftarrow 1$ ▷ *W-bit integer*

Require: Uniform state (Z', M') such that $Z' \mid M' \sim \mathrm{Uniform}[0, M')$ and $Z' \perp Z \mid M, M'$.
Ensure: Update the global uniform state (Z, M) such that $Z \mid M, M' \sim \mathrm{Uniform}[0, MM')$ (cf. Proposition 2.3).
5: **procedure** RECYCLE(Z', M')
6: **update** $Z \twoheadleftarrow Z + Z'M$
7: **update** $M \twoheadleftarrow MM'$

Algorithm 2.2 Refilling the global uniform state with fresh uniform bits from the entropy source

Require: Read access to global variables W and M from Algorithm 2.1.
Ensure: Update the global uniform state (Z, M) so that $M \in [2^{W-1}, 2^W)$ and $Z \mid M \sim \mathrm{Uniform}[0, M)$.
1: **procedure** REFILL()
2: $k \leftarrow W - \lceil \log_2(M+1) \rceil$ ▷ *number of fresh bits to append to Z*
3: $(c_0, \ldots, c_{k-1}) \leftarrow \mathrm{FLIP}(k)$ ▷ *obtain k i.i.d. bits from entropy source*
4: $Z' \leftarrow c_{k-1} \cdot 2^{k-1} + \cdots + c_1 \cdot 2^1 + c_0 \cdot 2^0$ ▷ *compute random k-bit integer*
5: $M' \leftarrow 2^k$ ▷ *upper bound on the value of the random integer*
6: RECYCLE(Z',M') ▷ *merge fresh uniform state into global state (Algorithm 2.1)*

Our randomness recycling algorithms will propagate a uniform state $S = (Z, M)$ in a way that maintains the invariant (I). To ensure that the space complexity remains bounded, we fix an upper bound $W \geq 1$ on the size of any uniform state, so that $\mathbb{P}(M \in [1, 2^W)) = 1$ and $\mathbb{P}(Z \in [0, 2^W - 1)) = 1$. The variables (Z, M) are each stored as one W-bit word in memory. Rather than explicitly accept and return a state in the functional style (1.3), we instead manipulate a global random state using in-place updates.

The RECYCLE procedure shown in Algorithm 2.1 uses Proposition 2.3 to recycle a new uniform state (Z', M') into the global uniform state (Z, M). The notation $Z \twoheadleftarrow \ldots$ and $M \twoheadleftarrow \ldots$ on lines 6 and 7 means that (Z, M) is being updated in place. The global uniform state is initialized as $s_0 = (Z, M) = (0, 1)$ in lines 3 and 4, which is the only valid deterministic choice.

It will also be necessary to recycle a *fresh* uniform state (Z', M') that is drawn directly from the entropy source into (Z, M). We call this operation REFILL, as shown in Algorithm 2.2. The interface to the entropy stream $\boldsymbol{C} = (C_i)_{i \geq 1}$ is by denoted FLIP(k), which returns off the next k i.i.d. fair coin tosses subject to the fresh coin guarantee (1.7). The number k of tosses computed on line 2 ensures that the updated uniform state (Z, M) does not overflow beyond W-bit words. The coin toss counters v_i and t_i from Listing 1.1 are obtained by keeping track of the calls to FLIP(k) on line 2 across the rounds of online sampling.

2.1.2 Splitting a Uniform State into Two Uniform States. The merge operation Proposition 2.3 preserves information and has the following inverse.

PROPOSITION 2.4. *For any integers $m, m' \geq 1$,*

$$Z \sim \mathrm{Uniform}[0, mm') \implies \begin{cases} \lfloor Z/m \rfloor \sim \mathrm{Uniform}[0, m') \\ (Z \bmod m) \sim \mathrm{Uniform}[0, m) \\ \lfloor Z/m \rfloor \perp (Z \bmod m). \end{cases}$$

Proof. Immediate from the arguments in §4. ∎

Proposition 2.4 shows how to split a uniform random state (over a composite integer range with known factorization) into two smaller independent uniform random states, using a single integer division.

2.2 Merging and Splitting Nonuniform Random States. We next describe generalizations of Propositions 2.3 and 2.4 for nonuinform random states, which will be used by our recycling algorithms.

2.2.1 Merging a Nonuniform and Uniform State into a Uniform State.

PROPOSITION 2.5. *For any positive integers* $\mathbf{w} \coloneqq w_0, \ldots, w_{n-1}$ *with sum* m*,*

$$Z \sim \text{Discrete}(\mathbf{w}), \quad (Z' \mid Z) \sim \text{Uniform}[0, w_Z) \implies \left(Z' + \sum_{i=0}^{Z-1} w_i\right) \sim \text{Uniform}[0, m).$$

Proof. The map $\cup_{i=0}^{n-1}(\{i\} \times [0, w_i)) \ni (x, x') \mapsto x' + \sum_{i=0}^{x-1} w_i \in [0, m)$ is a bijection, so each outcome is equally likely with probability $(w_i/m)(1/w_i) = 1/m$. ∎

Remark 2.6. Proposition 2.3 is a special case of Proposition 2.5 with $\mathbf{w} = (m', m', \ldots, m')$.

2.2.2 Splitting a Uniform State into a Nonuniform and Uniform State. The inverse of Proposition 2.5 is as follows.

PROPOSITION 2.7. *For any positive integers* $\mathbf{w} \coloneqq w_0, \ldots, w_{n-1}$ *with sum* m*,*

$$Z \sim \text{Uniform}[0, m), \quad Z \equiv \sum_{i=0}^{Y-1} w_i + X \text{ (where } X \in [0, w_Y)) \implies \begin{cases} Y \sim \text{Discrete}(\mathbf{w}) \\ (X \mid Y) \sim \text{Uniform}[0, w_Y). \end{cases}$$

Proof. Let $W_i \coloneqq w_0 + \cdots + w_i$ denote the prefix sums of $\mathbf{w}$ for $i \in [-1, n)$. The event $\{Y = i\}$ holds if and only if $Z \in [W_{i-1}, W_i)$ which has probability $(W_i - W_{i-1})/m = w_i/m$. Since $X = Z - W_{Y-1}$, conditioning on the event $\{Y = i\}$ gives $Z \mid \{Y = i\} \in [W_{i-1}, W_i)$ and $X \mid \{Y = i\} \in [0, W_i - W_{i-1} \equiv w_i)$ which are both uniformly distributed by the uniformity of Z. ∎

Proposition 2.7 says that a uniform random state can be split into (i) a nonuniform random state following *any* given finite discrete rational distribution; and (ii) a leftover uniform random state; as long as the original uniform range matches the common denominator of the target discrete distribution. It confirms the merge operation in Proposition 2.5 is reversible and justifies the recycling rule for the inversion method presented in the next section.

3 Proof of Main Theorem.

3.1 A Recycler for Uniform Distributions. We now use the ideas from the previous section to develop a randomness recycler for sampling discrete uniforms. Suppose our goal is to generate $X \sim \text{Uniform}[0, n)$ over a range $n \leq M$, where (Z, M) denotes the global uniform state maintained in Algorithm 2.1. Applying Proposition 2.7 with the weights $\mathbf{w} = (M - n, n)$ furnishes a Bernoulli sample $Y \sim \text{Discrete}(M - n, n)$ and a uniform sample $X \sim \text{Uniform}[0, w_Y)$. A straightforward accept-reject approach is as follows.

(R1) If $Y = 1$, then $X \sim \text{Uniform}[0, n)$ is accepted and returned as the desired sample.

(R2) If $Y = 0$, then $X \sim \text{Uniform}[0, M - n)$, which is not over the desired range, is rejected and $(X, M - n)$ is used as the new global uniform state.

Information Loss. In the accept and reject cases, we must discard the Bernoulli outcome Y and waste the corresponding $H_b(n/M)$ bits of information because it is in general correlated with the output sample X, which violates the contract (C). More importantly, we cannot recycle Y back into the global uniform state using Proposition 2.5 unless we recycle it in both the accept and reject cases. But recycling in the accept case requires consuming the desired $X \sim \text{Uniform}[0, n)$, which effectively undoes the sampling operation. Discarding a Bernoulli outcome Y in this way is the only operation where we lose information and is fundamentally irreversible.

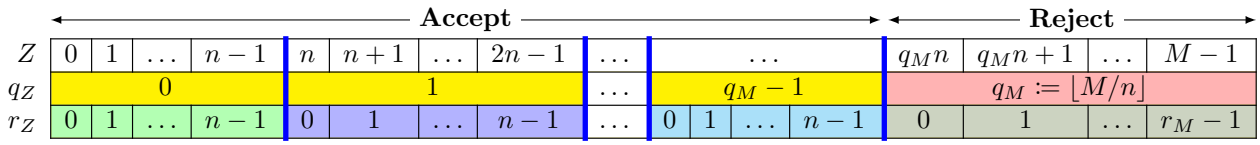
Reducing Rejection Probability. If $M \approx n$, then (R1)–(R2) gives a reasonably time- and entropy-efficient sampler for $X \sim \text{Uniform}[0, n)$ (cf. Remark 5.1). On the other hand, if $M \gg n$ then the rejection probability $(M - n)/M$ is very high. To decrease the rejection probability, we can multiply the accept weight and use Proposition 2.4 in the accept case to split the uniform state into two smaller uniform states. The resulting sampler operates as follows:

(S1) Generate $\hat{X} \sim \text{Uniform}[0, \lfloor M/n \rfloor n)$ using (R1)–(R2).

(S2) Apply Proposition 2.4 (with $Z = \hat{X}$, $m = n$, $m' = \lfloor M/n \rfloor$) to obtain $X \coloneqq (\hat{X} \bmod n) \sim \text{Uniform}[0, n)$ as the desired sample. Recycle the leftover uniform state $\lfloor \hat{X}/n \rfloor \sim \text{Uniform}[0, \lfloor M/n \rfloor)$ into the global uniform state using Algorithm 2.1.

The correctness of (S1) follows from the fact $\lfloor M/n \rfloor n \leq M$, so (R1)–(R2) can be applied.

Implementation. We now give a concrete algorithm that fuses (S1)–(S2) together to arrive at a simple and efficient algorithm for uniform sampling. The goal is to generate a discrete uniform $U \sim \text{Uniform}[0, n)$ for some $n \in [1, M]$, using the global uniform state (Z, M). Consider the quotients and remainders of $Z = q_Z n + r_Z$ and $M = q_M n + r_M$ modulo n, whose possible values as Z ranges over $[0, M)$ are shown below.



The first row shows the possible values of $Z \in [0, M)$. The second and third rows show the corresponding random quotient $q_Z = \lfloor Z/n \rfloor$ and remainder $r_Z = Z \bmod n$, respectively. Algorithm 3.1 uses these properties as follows:

- Consider the event $\{q_Z < q_M\}$. Here, Z is equally likely to be any one of the elements whose quotient is less than q_M. Further, among all elements with quotient q_Z, Z is equally likely to have any remainder $r_Z \in \{0, \ldots, n-1\}$. It follows that $q_Z \mid \{q_Z < q_M\} \sim \text{Uniform}[0, q_M)$ and $r_Z \mid \{q_Z < q_M\} \sim \text{Uniform}[0, n)$.

- Consider the event $\{q_Z = q_M\}$. In this case, Z falls in the final (pink) segment above. It follows that the remainder $r_Z \mid \{q_Z = q_M\} \sim \text{Uniform}[0, r_M)$, where $r_Z = r_M$ is impossible because $Z < M$ surely.

Algorithm 3.1 Uniform sampling with randomness recycling

Require: Read and write access to global variables (Z, M) from Algorithm 2.1;
Integer $n \in [1, 2^{W-1}]$, where W is the bit width of Z and M.
Ensure: Random sample $U \sim \text{Uniform}[0, n)$

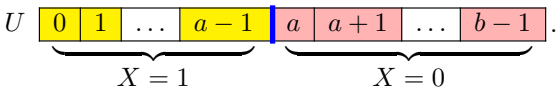
```
1:  procedure UNIFORM(n)
2:      while true do
3:          REFILL()                          ▷ refill global uniform state (Algorithm 2.2)
4:          (q_Z, r_Z) ← DIVMOD(Z, n)         ▷ Z = q_Z n + r_Z
5:          (q_M, r_M) ← DIVMOD(M, n)         ▷ M = q_M n + r_M
6:          if q_Z < q_M then                 ▷ accept case
7:              update (Z, M) ↞ (q_Z, q_M)    ▷ recycling
8:              return r_Z                    ▷ r_Z ~ Uniform[0, n)
9:          else                              ▷ reject case
10:             update (Z, M) ↞ (r_Z, r_M)    ▷ recycling
```

Remark 3.1. If n divides M, then $\mathbb{P}(q_Z = q_M) = 0$, i.e., UNIFORM$(n)$ never rejects. The if-else statement in Algorithm 3.1 can be seen as branching into either the accept case $Z \sim \text{Uniform}[0, q_M n)$, where $Z \in [0, q_M n)$ ensures termination, or the rejection case $Z - q_M n \in [0, r_M)$, which requires drawing more coin tosses from the entropy source.

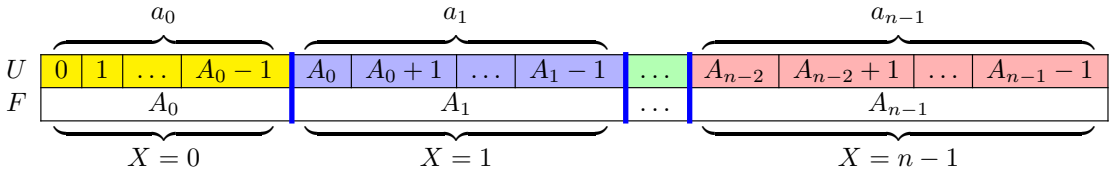
Remark 3.2. In line 3, REFILL ensures that the upper bound M of the uniform state lies in the range $[2^{W-1}, 2^W)$. While refilling the uniform state is not necessary when $n \leq M$, the algorithm is most entropy efficient when the rejection probability $r_M/M \leq (n-1)/M$ is near zero. To ensure efficient amortized entropy usage, the call to REFILL in Algorithm 3.1 minimizes the rejection probability r_M/M by enlarging M to the range $[2^{W-1}, 2^W)$, where 2^{W-1} is much larger than any integer n we typically intend to sample from. For example, if we only intend to sample from uniforms over $n \in (1, 2^{32})$, then setting $W = 64$ bits ensures a vanishingly small rejection probability of less than $2^{-31} < 5 \times 10^{-10}$. Each time an accept/reject decision is made (i.e., the if-else branch in Algorithm 3.1), exactly $H_b(r_M/M)$ bits of entropy are wasted. This accept/reject information must be discarded to maintain the independence invariant (I). The fresh coin tosses created in the call to REFILL are recovered to the maximum extent possible in lines 7 and 10 of Algorithm 3.1 while maintaining the invariants.

3.2 A Recycler for General Distributions. The uniform recycler in the previous section can be used to sample from any rational discrete distribution. Consider first generating $X \sim \text{Bernoulli}(a/b)$ with rational weight a/b. We draw $U \sim \text{Uniform}[0, b)$ and set $X \leftarrow 1$ if $U < a$, and $X \leftarrow 0$ if $a \le U$. The randomness recycling rule is based on extracting the following uniform states:

- If $\{U < a\}$, then $U \mid \{U < a\} \sim \text{Uniform}[0, a)$.
- If $\{a \le U\}$, then $U \mid \{a \le U\} \sim \text{Uniform}[a, b)$.



This method admits a direct generalization to sampling from $p \coloneqq (a_0, \ldots, a_{n-1})/A$, where $A \coloneqq a_0 + \cdots + a_{n-1} > 0$. Let $F \coloneqq (A_0, A_1, \ldots, A_{n-1} \equiv A)$ denote the cumulative probabilities, i.e., the prefix sums of the integers that define p, which are either constructed from the integers $(a_0, \ldots, a_{n-1})$ or given as a function $i \mapsto F_i$. A uniform variate $U \sim \text{Uniform}[0, A)$ is first generated and then $X \leftarrow \min\{i \in [0, n) \mid U < A_i\}$ is returned as the sample from p. In this setup, the event $\{X = i\}$ is equivalent to the event $U \in [A_{i-1}, A_i)$, where $A_{-1} \coloneqq 0$. The uniformity of U ensures that $U \mid \{X = i\} \sim \text{Uniform}[A_{i-1}, A_i)$, which gives a uniform random state to recycle.



The top array (which has A_{n-1} elements) shows all possible values of $U \sim \text{Uniform}[0, A_{n-1})$. The bottom array (which has n elements) shows the cumulative probabilities F that determine X from U. Algorithm 3.2 shows a generic inversion sampler, which recycles randomness from the leftover uniform random state in Proposition 2.7. The search on line 3 can be performed in several different ways, depending on the usage pattern of the sampler.

- Using binary search on the cumulative probabilities when they are either given as input or computed as a preprocessing step. This method is ideal for generating multiple samples from the same distribution, since its space is linear and search time is logarithmic.
- Using a linear scan over the array of (cumulative) probabilities. This method is ideal when just a single sample from the distribution is required, as it does not require preprocessing.
- Using a lookup table (§5.1). This method is ideal when A is small, and it delivers constant search time.

Algorithm 3.2 Inversion sampling with randomness recycling

Require: Positive integers $a_0, \ldots, a_{n-1}$ with prefix sums $A_i = \sum_{j=0}^{i} a_j$ (or a computable function $i \mapsto A_i$).
Ensure: Random sample $X \sim \text{Discrete}(a_0, \ldots, a_{n-1})$

1: **procedure** INVERSION($a_0, \ldots, a_{n-1}$)
2: $\quad U \sim \text{UNIFORM}(A_{n-1})$ ▷ *draw uniform variate (Algorithm 3.1)*
3: $\quad$ Let $X \leftarrow \min\{i \in [0, n) \mid U < A_i\}$ ▷ *search for sample X*
4: $\quad (Z', M') \leftarrow (U - A_{X-1}, a_X)$ ▷ *extract uniform state*
5: $\quad \text{RECYCLE}(Z', M')$ ▷ *recycle the uniform state (Algorithm 2.1)*
6: $\quad$ **return** X

3.3 Entropy Cost of Uniform Sampling. We first analyze the entropy loss that occurs in a single invocation of UNIFORM (Algorithm 3.1). For any invocation of UNIFORM(n), the call to REFILL on line 3 ensures that the global uniform state (Z, M) satisfies $M \ge M_{\min} \coloneqq 2^{W-1}$. The rejection probability is thus $(M \bmod n)/M \le (n-1)/M \le (n-1)/M_{\min}$. Recall from §4 that entropy is only ever lost when discarding the

information of a Bernoulli$((M \bmod n)/M)$ coin toss in the accept/reject decision. The total entropy loss from a given invocation is then the expected number of trials times the expected entropy of the discarded Bernoulli.

We make this intuition precise. Consider a generic invocation of $U \sim \textsc{Uniform}(n)$, given a random state (Z^-, M^-) such that $Z^- \mid M^- \sim \text{Uniform}[0, M^-)$, where $n \in [1, M_{\min}]$. Let (Z^+, M^+) be the final random state at termination of the call. Let V denote the (random) number of coin tosses requested from the entropy source, across all invocations of Flip (through line 3 of Refill) made by the call $\textsc{Uniform}(n)$. With these notations, the information loss given initial state (Z^-, M^-) is

$$(\log(M^-) + V) - (\log(n) + \log(M^+)) \tag{3.1}$$

where $\log(M^-)+V$ is the information content consumed and $\log(n)+\log(M^+)$ is the information content produced (all input and output variables in Algorithm 3.1 are uniform).

Proposition 3.3. *The entropy loss of any invocation* $\textsc{Uniform}(n)$ *of Algorithm 3.1 satisfies*

$$\mathbb{E}\left[(\log(M^-) + V) - (\log(n) + \log(M^+)) \mid M^-, U\right] \le \frac{M_{\min}}{M_{\min} - n + 1} H_{\mathrm{b}}\left(\frac{n-1}{M_{\min}}\right), \tag{3.2}$$

where the expectation is taken over the random coin tosses from the entropy source Flip.

Proof. This entropy loss bound is justified informally in §4. Here, we prove the bound using a formal analysis of Algorithm 3.1. Let $J \ge 1$ denote the random number of loops in the invocation $\textsc{Uniform}(n)$ until a sample is returned. In each loop iteration j, the global uniform state is manipulated in two places: the call to Refill on line 3 at the start of the loop; and the recycling on line 7 or line 10 at the end the of the loop. To track the state, we define for $j \in [1, J]$

- (Z_j^-, M_j^-) is the global uniform state at the start of iteration j;
- $(Z_j^\star, M_j^\star)$ is the global uniform state after the call to Refill;
- (Z_j^+, M_j^+) is the global uniform state at the end of iteration j.

These quantities satisfy

$$M_1^- = M^- \qquad M_{j+1}^- = M_j^+ \ (j \in [1, J-1]) \qquad M_J^+ = M^+. \tag{3.3}$$

Let V_j for $j \in [1, J]$ denote the number of fresh coin tosses at iteration j of Uniform, through line 2 of Refill. Proposition 2.3 implies the information is preserved:

$$\log(M_j^\star) = \log(M_j^-) + V_j \qquad (j \in [1, J]). \tag{3.4}$$

The accept-reject step at iteration j is

(Accept) If $Z_j^\star < n\lfloor M_j^\star/n \rfloor$ then $M_j^+ = \lfloor M_j^\star/n \rfloor$; and the loop terminates and a sample from Uniform$[0, n)$ is successfully returned.

(Reject) Else $M_j^+ = M_j^\star \bmod n$; and the loop continues.

It follows that the output information O_j at the end of iteration j satisfies

$$O_j = \begin{cases} \log(\lfloor M_j^\star/n \rfloor) + \log(n) & \text{with probability } \dfrac{n\lfloor M_j^\star/n \rfloor}{M_j^\star} \quad \text{(accept case)} \\ \log(M_j^\star \bmod n) & \text{with probability } 1 - \dfrac{n\lfloor M_j^\star/n \rfloor}{M_j^\star} \quad \text{(reject case).} \end{cases} \tag{3.5}$$

Applying (3.4), the expected entropy loss at iteration j is then

$$\mathbb{E}\left[(\log(M_j^-) + V_j) - O_j \mid M^-\right] = \mathbb{E}\left[(\log(M_j^\star) - O_j) \mid M^-\right]. \tag{3.6}$$

Applying (3.5) gives

$$\mathbb{E}\left[\log(M_j^\star) - O_j \mid M^-\right]$$

$$= \mathbb{E}\left[\log(M_j^\star) - \frac{n\lfloor M_j^\star/n\rfloor}{M_j^\star}\left(\log(\lfloor M_j^\star/n\rfloor) + \log(n)\right) - \frac{M_j^\star \bmod n}{M_j^\star}\left(\log(M_j^\star \bmod n)\right) \;\middle|\; M^-\right] \tag{3.7}$$

$$= \mathbb{E}\left[\frac{n\lfloor M_j^\star/n\rfloor}{M_j^\star}\left(\log\left(\frac{M_j^\star}{n\lfloor M_j^\star/n\rfloor}\right)\right) + \frac{M_j^\star \bmod n}{M_j^\star}\left(\log\left(\frac{M_j^\star}{M_j^\star \bmod n}\right)\right) \;\middle|\; M^-\right] \tag{3.8}$$

$$= \mathbb{E}\left[H_\mathrm{b}\left(\frac{M_j^\star \bmod n}{M_j^\star}\right) \;\middle|\; M^-\right] \tag{3.9}$$

$$\leq H_\mathrm{b}\left(\frac{(n-1)}{M_\mathrm{min}}\right). \tag{3.10}$$

We now argue that the sum of these entropy losses over the J random iterations precisely corresponds to the total entropy loss (3.1), as follows:

$$\sum_{j=1}^{J}\left(\left(\log(M_j^-) + V_j\right) - O_j\right)$$

$$= \log(M_1^-) + (V_1 + \ldots V_j) + \sum_{j=2}^{J}(\log(M_j^-) - \log(M_{j-1}^+)) - (\log(n) + \log(M_J^+)) \tag{3.11}$$

$$= \left(\log(M^-) + V\right) - (\log(n) + \log(M^+)), \tag{3.12}$$

where we have used the properties in (3.3) and the fact that O_j is precisely in the *reject case* of (3.5) for $j \in [1, J-1]$ (i.e., $M_j^+ = M_j^\star \bmod n = M_{j+1}^-$) by the definition of J.

Finally, since the acceptance probability in (3.5) is at least $(M_\mathrm{min} - n + 1)/M_\mathrm{min}$, the expected number of iterations is $\mathbb{E}[J] \leq M_\mathrm{min}/(M_\mathrm{min} - n + 1)$, which combined with (3.10) gives the result:

$$\mathbb{E}\left[\sum_{j=1}^{J}\left(\log(M_j^-) + V_j - O_j\right) \;\middle|\; M^-, U\right] \leq \mathbb{E}\left[\sum_{j=1}^{J} H_\mathrm{b}\left(\frac{(n-1)}{M_\mathrm{min}}\right) \;\middle|\; M^-, U\right] \tag{3.13}$$

$$= \mathbb{E}\left[J \;\middle|\; M^-, U\right] H_\mathrm{b}\left(\frac{(n-1)}{M_\mathrm{min}}\right) \tag{3.14}$$

$$\leq \frac{M_\mathrm{min}}{(M_\mathrm{min} - n + 1)} H_\mathrm{b}\left(\frac{(n-1)}{M_\mathrm{min}}\right). \tag{3.15}$$

In (3.13), conditioning on the returned uniform U is justified by the following facts:

- Conditioned on M^-, the number of loop iterations J is independent of the return value U.
- The initial state $M_1^- = M^-$ is vacuously conditionally independent of U given M^-.
- For each $j \in [1, J]$, coin toss counter $V_j = W - (\lfloor \log_2(M_j^-)\rfloor + 1)$ (cf. line 2 of Algorithm 2.2) is determined by M_j^-. For $j \in [1, J]$, the output entropy O_j and state bound M_j^+ are determined by M_j^-, V_j, and J (cf. (3.4) and (3.5)). Lastly, $M_j^+ = M_{j+1}^-$ by definition for $j \in [1, J-1]$.
- By induction, for all $j \in [1, J]$, the quantities M_j^-, V_j, and O_j, are conditionally independent of U given M^-. The second point is the base case, the third point provides the inductive step, and the first point handles the random stopping time J which is used in each inductive step. ∎

Remark 3.4. In practice, a typical setting to minimize entropy loss is letting $M_\mathrm{min} = 2^{63}$ (i.e., $W = 64$; M is guaranteed to fit in an `unsigned long long int` word) and $n < 2^{32}$ (guaranteed to fit in an `unsigned long int` word). From Proposition 3.3, these values guarantee an expected entropy loss that is less than 2×10^{-8} bits, and a similar bound parametric in n is $5 \times 10^{-10}\log(n)$ bits.

3.4 Entropy Cost of General Sampling. We now analyze the entropy loss that occurs in a single invocation of Inversion (Algorithm 3.2). Any invocation Inversion(p) uses exactly *one call* $U \leftarrow$ Uniform(n) (on line 2) It then uses Proposition 2.7 to reversibly decompose U into a sample X from the desired target distribution together with a leftover uniform state (Z', M') that is recycled into the global uniform state. Therefore, the information loss should correspond precisely to that of generating the uniform U in Proposition 3.3.

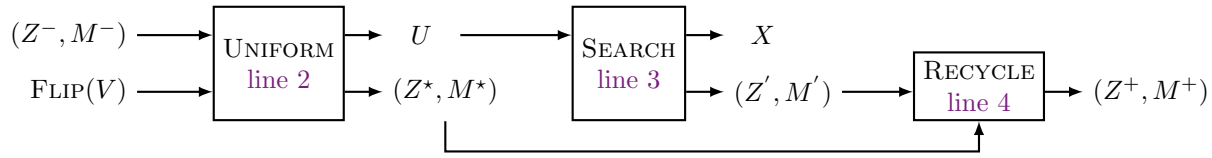


Figure 3.1: Information flow in Inversion (Algorithm 3.2) for sampling general distributions.

We make this intuition precise. Consider a generic invocation of $X \sim$ Inversion(p), given a random state (Z^-, M^-) such that $Z^- \mid M^- \sim \mathrm{Uniform}[0, M^-)$. Let $n \in [1, M_{\min}]$ denote the argument to Uniform on line 2. Write $(Z^\star, M^\star)$ to be the uniform state after line 2 and let V denote the total number of coin tosses requested by Uniform. Let (Z', M') be as defined in line 4 and write (Z^+, M^+) for the final random state after line 5, which is the result of applying Proposition 2.3 to $(Z^\star, M^\star)$ and (Z', M'). Figure 3.1 shows the information flow in Inversion and Proposition 3.5 formalizes the entropy loss bound with respect to this diagram.

Proposition 3.5. *The entropy loss of any invocation* Inversion(p) *of Algorithm 3.2 satisfies*

$$\mathbb{E}\left[(\log(M^-) + V) - \left(\log\left(\frac{1}{p(X)}\right) + \log(M^+)\right) \;\middle|\; M^-, U, X\right] \le \frac{M_{\min}}{M_{\min} - n + 1} H_{\mathrm{b}}\left(\frac{n-1}{M_{\min}}\right), \tag{3.16}$$

where the expectation is taken over the random coin tosses from the entropy source Flip.

Proof. From Proposition 2.7, the variable U is reversibly split into (X, Z'), which via (2.1) preserves the information content pointwise:

$$\log n = \log\left(\frac{1}{p(x)}\right) + \log(M'). \tag{3.17}$$

From Proposition 2.3, the variables $((Z^\star, M^\star), (Z', M'))$ are reversibly merged into (Z^+, M^+), which gives

$$\log(M^+) = \log(M^\star) + \log(M'). \tag{3.18}$$

These equalities establish that $\log n + \log M^\star = \log(1/p(x)) + \log M^+$, where the expression on the left-hand-side is precisely the output entropy in (3.2). The conclusion follows from Proposition 3.3, where the additional condition on X in (3.16) is permitted as X is a *deterministic* function of U (cf. line 3 of Algorithm 3.2). ∎

3.5 Entropy Cost of Generating a Random Sequence. Toward proving Theorem 1.5, we analyze the overall entropy cost of using Inversion to generate an output sequence $\boldsymbol{X} = (X_i)_{i \ge 1}$, given an arbitrary distribution sequence $\boldsymbol{p} = (p_i)_{i \ge 1}$.

Proposition 3.6. *The entropy cost of a sequence of calls of* Inversion(p_i) *to generate* $(X_i \sim p_i)_{i \ge 1}$ *satisfies*

$$\mathbb{E}\left[t_k(\boldsymbol{C}, \boldsymbol{p}) \mid X_1 = x_1, \ldots, X_k = x_k\right] < \sum_{i=1}^{k} \log\left(\frac{1}{p_i(x_i)}\right) + k \frac{M_{\min}}{M_{\min} - d + 1} H_{\mathrm{b}}\left(\frac{d-1}{M_{\min}}\right) + W, \tag{3.19}$$

where $\{X_1 = x_1, \ldots, X_k = x_k\}$ *is a positive probability event and* $d \in [1, M_{\min}]$ *is an upper bound on the argument* n *in any invocation of* Uniform(n) *(i.e.,* $p_{\le k} \in (\Delta \mathcal{X}_d)^k$*).*

Proof. Write $t_k(\boldsymbol{C}, \boldsymbol{p}) = V_1 + \cdots + V_k$ as the sum of coin tosses used in each round and denote the conditioning event as $E \coloneqq \{X_1 = x_1, \ldots, X_k = x_k\}$. Then

$$\mathbb{E}\left[t_k(\boldsymbol{C}, \boldsymbol{p}) \mid E\right] = \mathbb{E}\left[V_1 + \ldots V_k \mid E\right] = \sum_{i=1}^{k} \mathbb{E}\left[V_i \mid E\right] \tag{3.20}$$

$$= \sum_{i=1}^{k} \mathbb{E}\left[\mathbb{E}\left[V_i \mid M_i^-, U_i, X_i = x_i\right] \mid E\right] \tag{3.21}$$

$$\leq \sum_{i=1}^{k} \left[\log\left(\frac{1}{p(x_i)}\right) + \mathbb{E}\left[\mathbb{E}\left[\log(M_i^+) - \log(M_i^-) \mid M_i^-, X_i = x_i\right] \mid E\right] + \frac{M_{\min}}{M_{\min} - n + 1} H_{\mathrm{b}}\left(\frac{n-1}{M_{\min}}\right)\right] \tag{3.22}$$

$$= \sum_{i=1}^{k} \log\left(\frac{1}{p_i(x_i)}\right) + k\frac{M_{\min}}{M_{\min} - d + 1} H_{\mathrm{b}}\left(\frac{d-1}{M_{\min}}\right) + \sum_{i=1}^{k} \mathbb{E}\left[\mathbb{E}\left[\log(M_i^+) - \log(M_i^-) \mid X_i = x_i, M_i^-\right] \mid E\right]. \tag{3.23}$$

Equation (3.21) uses the tower property of conditional expectation, where the fresh coin guarantee guarantee (1.7) ensures that V_i is conditionally independent of all events in E except $\{X_i = x_i\}$ given the initial state M_i^-. Equation (3.22) applies the bound from (3.16) and monotonicity of conditional expectation. We next analyze the term (3.23), recalling that (by definition) $M_1^- = M_0$ and $M_i^- = M_{i-1}^+$ for $i = 2, \ldots, k$. The shared terms for $i = 1, \ldots, k-1$ in the telescoping series are

$$\mathbb{E}\left[\mathbb{E}\left[\log(M_i^+) \mid M_i^-, X_i = x_i\right] \mid E\right] = \mathbb{E}\left[\log(M_i^+) \mid E\right] = \mathbb{E}\left[\log(M_{i+1}^-) \mid E\right], \tag{3.24}$$

$$\mathbb{E}\left[\mathbb{E}\left[-\log(M_{i+1}^-) \mid M_{i+1}^-\right] \mid E\right] = \mathbb{E}\left[-\log(M_{i+1}^-) \mid E\right]. \tag{3.25}$$

where the first equality follows from the fact that M_i^+ is independent of E given M_i^- and X_i. The surviving terms from the final sum in (3.23) are

$$\mathbb{E}\left[\mathbb{E}\left[\log(M_k^+) \mid M_k^-, X_k = x_k\right] - \mathbb{E}\left[\log(M_1^-) \mid M_1^-, X_1 = x_1\right] \mid E\right] < W, \tag{3.26}$$

where use the fact that $M_1^- = M_0 = 1$ and at every step $M^+ < 2^W$ surely. The conclusion (3.19) follows. ∎

As the bound involving $\mathbb{E}\left[t_k(\boldsymbol{C}, \boldsymbol{p}) \mid X_{\leq k} = x_{\leq k}\right] - \sum_{i=1}^{k} \log(1/p_i(x_i))$ in (3.19) holds for *every* distribution sequence $\boldsymbol{p}$, an identical bound on $\mathbb{E}\left[t_k(\boldsymbol{C}, \boldsymbol{P}) \mid X_{\leq k} = x_{\leq k}, P_{\leq k} = p_{\leq k}\right] - \sum_{i=1}^{k} \log(1/p_i(x_i))$ holds for any random distribution sequence $\boldsymbol{P}$ satisfying (1.7), as noted in (1.13).

Remark 3.7. The same result holds for any randomness recycling sampler that generates $X \sim p$ by calling Uniform once, such as Algorithms 5.1–5.3 in §5.

3.6 Proof of Theorem 1.5.

This section proves the main result, which is restated below.

Theorem 1.5. *For any $\varepsilon > 0$ and $d \geq 1$, there exists an online random sampling algorithm using a sequence $\boldsymbol{C}$ of i.i.d. coin tosses such that, for every distribution sequence $\boldsymbol{p} \in (\Delta\mathcal{X}_d)^{\mathbb{N}}$, the entropy cost of generating an output sequence $\boldsymbol{X} = (X_i \sim p_i)_{i \geq 1}$ satisfies*

$$\mathbb{E}\left[t_n(\boldsymbol{C}, \boldsymbol{p}) \mid X_1 = x_1, \ldots, X_n = x_n\right] < \sum_{i=1}^{n} \log\left(\frac{1}{p_i(x_i)}\right) + \varepsilon n + W_{d,\varepsilon} \qquad (n \geq 1), \tag{1.11}$$

where $\{X_1 = x_1, \ldots, X_n = x_n\}$ is any positive probability event and $W_{d,\varepsilon} \sim \log(d/\varepsilon)$ as $d/\varepsilon \to \infty$. The algorithm uses auxiliary space of at most $2W_{d,\varepsilon}$ bits across rounds and $O(W_{d,\varepsilon})$ temporary space per round.

Proof. Algorithm 3.3 is a witness to the online algorithm from the theorem statement, making repeated use of the inversion sampler in Algorithm 3.2. The Recycle(Z', M'), Refill(), and Uniform(n) methods each use $O(W)$ space. The global uniform state (Z, M), which is the only auxiliary state carried over between rounds, uses $2W$ bits of space. Algorithms 3.1 and 3.2 will never overflow the W-bit size of the global uniform state (Z, M). After calling Uniform(n), the global uniform state has a bound of $M \leq (2^W - 1)/n$. Any corresponding

Algorithm 3.3 Online random sampling algorithm witnessing Theorem 1.5

Require: Target distributions $\boldsymbol{P} = (P_i)_{i\geq 1}$, with P_i presented at step $i \geq 1$.
Ensure: Sequence of output samples $X_i \sim P_i$, for $i \geq 1$.

```
procedure RandomSequence
    for i ← 1 to ∞ do
        (S1) Receive the next target distribution P_i
        (S2) X_i ← Inversion(P_i)                    ▷ (Algorithm 3.2)
        yield X_i
```

recycled uniform state (Z', M') always satisfies $M' \leq n$. To arrive at the entropy bound in Theorem 1.5, by Proposition 3.6 it suffices to find $W \geq 1$ such that

$$\frac{1}{1-(d-1)/2^{W-1}} H_{\mathrm{b}}\left(\frac{d-1}{2^{W-1}}\right) \leq \varepsilon. \tag{3.27}$$

Let $h(\delta) \coloneqq H_{\mathrm{b}}(\delta)/(1-\delta)$. Then $h(\delta) \sim \delta \log(1/\delta)$ as $\delta \to 0$, so

$$W_{d,\varepsilon} \coloneqq 1 + \lceil \log((d-1)/h^{-1}(\varepsilon)) \rceil \tag{3.28}$$

satisfies $W_{d,\varepsilon} \sim \log(d/\varepsilon)$ as $d/\varepsilon \to \infty$, which matches the claimed entropy-space tradeoff. □

Remark 3.8. The exact inverse of $h(\delta)$ requires solving a transcendental equation, which is generally not possible in closed form, but it can be approximated with a series expansion, which may look roughly like

$$W_{d,\varepsilon} \approx 1 + \lceil \log(d) + \sum_{m=1}^{\log^*(1/\varepsilon)} \log^{\circ m}(1/\varepsilon) \rceil. \tag{3.29}$$

Remark 3.9. We assume the rational target distribution p is given as either a list of integer weights or pointwise access to its cumulative distribution function, which enables linear or logarithmic time complexity. A detailed discussion of time complexity lower bounds for random sampling under different representations of the target distribution—e.g., as probability mass functions or cumulative probabilities, and possibly as computable reals—is available in Trevisan (2010); Yamakami (1999). Bringmann and Panagiotou (2017) discuss the time complexity of discrete sampling algorithms in the real RAM model for a static array of sorted or unsorted probabilities.

4 Randomness Recyclers for Uniform Distributions.

In this section we compare the randomness recycler Algorithm 3.1 for discrete uniforms to previous uniform samplers (§4.1–§4.2) and then present algorithmic extensions that leverage the speed of word-level operations on modern CPUs (§4.3).

4.1 Comparison to the Lumbroso Uniform Sampler.

The Fast Dice Roller (FDR) algorithm from Lumbroso (2013; p. 4) for generating a discrete uniform can be understood as a special case of Algorithm 3.1. It uses a different strategy in the call to $\textsc{Refill}(Z, m)$ on line 3 than our strategy described in Remark 3.2. The "randomness recycling" interpretation of FDR in Huber and Vargas (2024) shows that the algorithm refills the uniform random state by repeatedly drawing fresh random bits until $M \geq n$, rather than our approach of always ensuring $M \in [2^{W-1}, 2^W)$. FDR is entropy optimal in the sense of Knuth and Yao when taking a *single sample*, and coincides with the suboptimal baseline from §1.3.2 when naïvely used to generate a sequence of samples.

Lumbroso (2013; Section 3.1) observes that, for generating an i.i.d. sequence $U_0, U_1, \ldots, U_{k-1} \sim \mathrm{Uniform}[0, n)$, it is possible to first generate $Y = \mathrm{Uniform}[0, n^k)$ entropy optimally using FDR and then recover the individual U_i using the decomposition of Y in base-n:

$$Y = U_{k-1} n^{k-1} + \cdots + U_0 n^0. \tag{4.1}$$

This technique coincides with batched baseline from (1.21). Because each distribution P_i is uniform, the space complexity of storing $P_{1:k}$ does not grow exponentially. However, computing the decomposition (4.1) requires integer division operations, which becomes expensive for large n^k. The method also requires prespecifying k and the ranges of the U_i beforehand. Sequential sampling with a fixed batch size $2/\varepsilon$ allows FDR to achieve

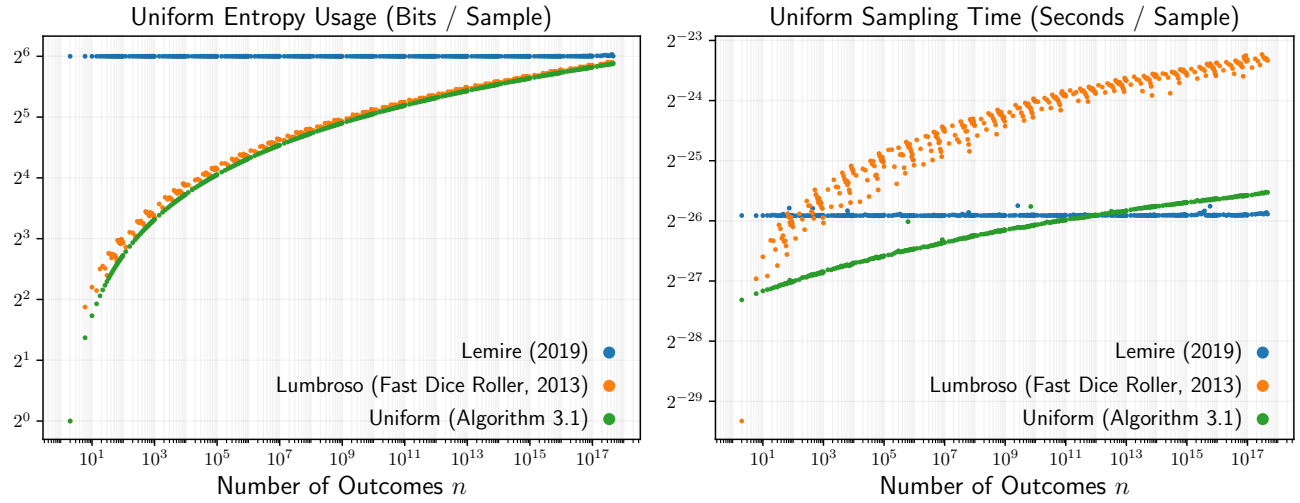


Figure 4.1: Benchmark comparison of entropy consumption and sampling time on a range of distribution sizes n, for three uniform samplers: the Fast Dice Roller of Lumbroso (2013), the method of Lemire (2019), and our uniform sampler with randomness recycling (Algorithm 3.1). Random bits are supplied by 256-byte buffered requests to `/dev/random`.

the asymptotic entropy rate of $H[U_0, \ldots, U_{k-1}]/k + \varepsilon$ bits per sample, but the $O(1/\varepsilon)$ computational overhead exceeds the $O(\log(1/\varepsilon))$ achieved by Algorithm 3.1 using randomness recycling.

The analysis of FDR reveals that the Knuth and Yao entropy toll of less than 2 bits arises from the rejection probability r_M/M when splitting the uniform state. In the worst case, this probability approaches a geometric distribution with parameter $1/2$. Using a larger value of $M \gg n$ reduces this rejection probability, and in turn the expected entropy cost of a given run. A larger uniform state requires an upfront investment of entropy, but amortized over many samples, the total cost quickly falls below that of any single-sample entropy-optimal sampler. Remark 3.2 shows that if $W = 64$ bits and $n < 2^{32}$, then the rejection probability is less than 2^{-31}. The expected amortized entropy cost of sampling each individual uniform is then bounded by

$$(1 + 10^{-9})H[U] + 64/k \text{ bits per sample.} \tag{4.2}$$

Thus, Algorithm 3.1 is a natural way to extend the spirit of batched generation à la Lumbroso in an online and space-efficient manner.

4.2 Comparison to the Lemire Uniform Sampler. Discrete uniform samplers often scale a uniform state over a range $[0, 2^w)$ (where the word size is typically $w = 64$) to a smaller range $[0, n)$ using integer division, with rejection sampling to ensure that the result is exact. Lemire (2019) describes a fast exact uniform sampler that eliminates the division in almost all cases by instead using a widening multiplication, which is efficient on modern CPUs (e.g., x86). This method is especially fast when the randomness source is inexpensive. In cases where the randomness source is expensive, the entropy inefficiency of the method becomes significant. Figure 4.1 compares the method of Lemire to Algorithm 3.1 and to the Fast Dice Roller from Lumbroso when using a cryptographically secure PRNG. The plot of sampling time shows three distinct regimes:

- For large n, near the maximum 64-bit integer, each sampler consumes approximately the same amount of entropy, roughly 64 bits per sample, so the difference in sampling time is determined by the other operations. The widening multiplication of Lemire is faster than both the divisions in Algorithm 3.1 and the loop over the bit-length of n in the FDR sampler, which requires roughly 64 iterations in this regime.

- For small n, the method of Lemire is slowest because it still consumes 64 bits of entropy per sample, which is wasteful. The FDR sampler can be fastest in this case because it is reasonably entropy-efficient and only requires a few loop iterations with efficient bitwise operations, rather than the division in Algorithm 3.1.

- For intermediate n, Algorithm 3.1 is fastest because it is more entropy efficient than the method of Lemire and even FDR, without looping over the bit-length of n as FDR does.

4.3 Optimized Uniform Sampling via Widening Multiplication and Batching. Real-world software libraries for uniform sampling leverage code optimizations that exploit the speed of word-level operations on modern CPUs. We develop additional randomness recycling techniques for uniform samplers that incorporate two optimizations: widening multiplication and batched sampling. We will show how randomness recycling can be used to speed up highly optimized uniform samplers from the literature that use widening multiplication and batching as follows.

- Algorithms 4.1 and 4.2 adapt Algorithms 2.1 and 3.1 by using specialized recycling rules that use efficient widening multiplication.
- Algorithm 4.3 augments the widening-multiplication method of Lemire (2019) discussed in §4.2 with randomness recycling.
- Algorithm 4.4 augments the widening-multiplication and batched-sampling method of Brackett-Rozinsky and Lemire (2025) with randomness recycling.

We assume for this section that W is the word size, the randomness stream is read in W-bit words, and W-bit integer operations are efficient (in particular, widening multiplication, which maps two W-bit integers to their $2W$-bit product, stored as two W-bit words that contain the high and low bits of the result, respectively).

4.3.1 Recycling with Widening Multiplication. Implementing the uniform-merging map of Proposition 2.3 in finite space as Algorithm 2.1 could result in overflow if the product MM' exceeds $2^W - 1$, so we must ensure a relationship between the global uniform state bound M and the size M' of any state which we want to recycle. This requirement restricts algorithms that recycle uniform states and can make them significantly slower. For example, the call to Refill in Algorithm 3.1 requires that the randomness stream be shifted by variable numbers of bits, instead of simply read word by word. Algorithm 4.1 shows how widening multiplication can be used to merge two word-sized uniform states into a single word-sized uniform state, with an additional independent full word of i.i.d. random bits in the case that the product would overflow.

Algorithm 4.1 Recycling a uniform state into a global uniform state using widening multiplication

Require: Read and write access to global variables (Z, M) from Algorithm 2.1;
Uniform state (Z', M') such that $Z' \mid M' \sim \text{Uniform}[0, M')$ and $Z' \perp Z \mid M, M'$.
Ensure: Update the global uniform state (Z, M) to incorporate randomness recycled from (Z', M'); possibly push an independent random uniform word to the entropy stream.

procedure RecycleWidening(Z', M')
  $(Z_{\text{hi}}, Z_{\text{lo}}) \leftarrow Z + Z' \otimes M$ ▷ *widening multiplication*
  $(M_{\text{hi}}, M_{\text{lo}}) \leftarrow M \otimes M'$ ▷ *widening multiplication*
  if $Z_{\text{hi}} = M_{\text{hi}}$ **then**
    update $(Z, M) \twoheadleftarrow (Z_{\text{lo}}, M_{\text{lo}})$
  else
    update $(Z, M) \twoheadleftarrow (Z_{\text{hi}}, M_{\text{hi}})$
    Recycle Z_{lo} into the entropy source ▷ $Z_{\text{lo}} \sim \text{Uniform}[0, 2^W)$

Algorithm 4.1 can be seen as merging two states as in Recycle (Algorithm 2.1) and then splitting the result as in Uniform(2^W) (Algorithm 3.1), so the analysis of correctness is similar. By Proposition 2.5, $2^W Z_{\text{hi}} + Z_{\text{lo}}$ is uniformly distributed over $[0, 2^W M_{\text{hi}} + M_{\text{lo}})$ Conditioned on the event $Z_{\text{hi}} = M_{\text{hi}}$, the value Z_{lo} is uniformly distributed over $[0, M_{\text{lo}})$ by Proposition 2.7. Similarly, conditioned on the event $Z_{\text{hi}} < M_{\text{hi}}$, the value $2^W Z_{\text{hi}} + Z_{\text{lo}}$ is uniformly distributed over $[0, 2^W M_{\text{hi}})$. In this case, Proposition 2.4 shows that the quotient and remainder must be distributed as $Z_{\text{hi}} \mid M_{\text{hi}} \sim \text{Uniform}[0, M_{\text{hi}})$ and $Z_{\text{lo}} \sim \text{Uniform}[0, 2^W)$, respectively, and they are independent.

Equipped with Algorithm 4.1, we can implement a simpler variant of Uniform, as shown in Algorithm 4.2. This variant uses random words directly from the source instead of calling Refill, and we note that it only accesses the global uniform state indirectly via RecycleWidening. Algorithm 4.2 and the remaining uniform

Algorithm 4.2 Uniform sampling with RecycleWidening

Require: Integer $n \in [1, 2^W)$, where W is the word size from Algorithm 2.1.
Ensure: Random sample $X \sim \text{Uniform}[0, n)$

```
procedure UniformWidening(n)
    while true do
        X ← Flip(W)                          ▷ X ~ Uniform[0, 2^W); uniform random word
        (q_X, r_X) ← DivMod(X, n)            ▷ X = q_X n + r_X
        (q_B, r_B) ← DivMod(2^W, n)          ▷ 2^W = q_B n + r_B
        if q_X < q_B then                    ▷ accept case
            RecycleWidening(q_X, q_B)        ▷ (Algorithm 4.1)
            return r_X                       ▷ r_X ~ Uniform[0, n)
        else                                 ▷ reject case
            RecycleWidening(r_X, r_B)        ▷ (Algorithm 4.1)
```

algorithms in this section only recycle randomness *into* the global uniform state, and the recycled randomness is reused only in the form of random words produced by RecycleWidening. The accept-reject branching in Algorithm 4.1 can waste up to $H_b(2^W/(2^{W+1}-1)) \approx 1$ bits of entropy in the worst case, so samplers like Algorithm 4.2 do not achieve the same entropy efficiency as Algorithm 3.1. However, in practice, the entropy waste is not too large, and samplers based on this recycling method can be significantly faster than those using Algorithm 3.1, even when the randomness source is very expensive.

4.3.2 Uniform Sampling with Widening Multiplication and Batching. Algorithm 4.2 benefits significantly from the word-level operations, but it still requires two divisions. Lemire (2019) shows how to essentially eliminate the division, but there is no apparent efficient way to implement a recycling rule; the leftover randomness from their algorithm is found in the low bits of the widening multiplication result, and a naïve approach to extract a recyclable uniform state introduces two extra divisions, entirely nullifying the benefit of fewer divisions. Instead of recycling after running the algorithm of Lemire (2019) verbatim, we can instead eliminate a division by using a generalization of this method to batched sampling. Brackett-Rozinsky and Lemire (2025) show how to generate multiple uniforms by repeated widening multiplications from a single random word, which is batched analogously to (1.21). We can apply this method to batch together uniforms over the ranges n and $\lfloor 2^W/n \rfloor$, which are the target uniform and the recyclable state, respectively. Figure 4.2 shows this method, for the example of generating a uniform of size $n = 6$ given a uniform random word of length $W = 4$ bits.

U	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$6U \div 16$	0	0	0	1	1	1	2	2	3	3	3	4	4	4	5	5
$2(6U \bmod 16) \div 16 \equiv (12U \div 16) \bmod 2$	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1
$2(6U \bmod 16) \bmod 16 \equiv 12U \bmod 16$	0	12	8	4	0	12	8	4	0	12	8	4	0	12	8	4
$U - (12U \div 16)$	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4

Figure 4.2: Illustration of Algorithm 4.3 to generate a uniform with range $n = 6$ given a uniform random word U of length $W = 4$ bits, including the operations required for randomness recycling.

The rows of this diagram are interpreted as follows:

- The first row shows the possible values of a uniform random word $U \in [0, 16)$.
- The second and third rows show the target uniform over $n = 6$ outcomes and the recyclable uniform state over $\lfloor 2^W/n \rfloor = 2$ outcomes, respectively.
- The fourth row shows the rejection condition $n\lfloor 2^W/n \rfloor U \bmod 2^W < 2^W \bmod n$.
- The fifth row shows a novel recycling rule for this method, which is justified in Proposition 4.1.

The recycling rule in the reject case is not essential, as noted by Mennucci (2010; Remark 7), because

Algorithm 4.3 Uniform sampling with widening multiplication and RecycleWidening

Require: Integer $n \in [1, 2^W)$, where W is the word size from Algorithm 2.1.
Ensure: Random sample $U \sim \text{Uniform}[0, n)$

procedure UniformLemire(n)
  $(q, t) \leftarrow \text{DivMod}(2^W, n)$ ▷ $2^W = q \cdot n + t$
  while true do
    $X \leftarrow \text{Flip}(W)$ ▷ $X \sim \text{Uniform}[0, 2^W)$; *uniform random word*
    $(U, r) \leftarrow X \otimes n$ ▷ *widening multiplication;* $U \nsim \text{Uniform}[0, n)$
    $(U', r') \leftarrow q \otimes r$ ▷ *widening multiplication;* $U' \nsim \text{Uniform}[0, q)$
    if $r' \geq t$ **then** ▷ *accept case*
      RecycleWidening(U', q) ▷ *(Algorithm 4.1)*
      return U ▷ $U \sim \text{Uniform}[0, n)$
    else ▷ *reject case*
      $(U'', r'') \leftarrow X \otimes (n \cdot q)$ ▷ *widening multiplication*
      RecycleWidening($X - U'', t$) ▷ *(Algorithm 4.1)*

Algorithm 4.4 Batched uniform sampling with widening multiplication and RecycleWidening

Require: Positive integers $n_i > 0$ with $\prod_{i=1}^{k} n_i < 2^W$, where W is the word size from Algorithm 2.1.
Ensure: Independent random samples $U_i \sim \text{Uniform}[0, n_i)$ for $i \in \{1, \ldots, k\}$

procedure UniformBrackett($n_1, \ldots, n_k$)
  $n \leftarrow \prod_{i=1}^{k} n_i$ ▷ *total target outcomes*
  $(n_{k+1}, t) \leftarrow \text{DivMod}(2^W, n)$ ▷ $2^W = t + \prod_{i=1}^{k+1} n_i$
  while true do
    $X \leftarrow \text{Flip}(W)$ ▷ $X \sim \text{Uniform}[0, 2^W)$; *uniform random word*
    $r \leftarrow X$ ▷ *copy of X to allow recycling*
    for $i \leftarrow 1, \ldots, k+1$ **do**
      $(U_i, r) \leftarrow r \otimes n_i$ ▷ *widening multiplication;* $U_i \nsim \text{Uniform}[0, n_i)$
    if $r \geq t$ **then** ▷ *accept case*
      RecycleWidening(U_{k+1}, n_{k+1}) ▷ *(Algorithm 4.1)*
      return $U_1, \ldots, U_k$ ▷ $U_i \sim \text{Uniform}[0, n_i)$
    else ▷ *reject case*
      $(U, r) \leftarrow X \otimes (n \cdot n_{k+1})$ ▷ *widening multiplication*
      RecycleWidening($X - U, t$) ▷ *(Algorithm 4.1)*

the rejection probability is vanishingly small in practice. Even more significantly, the entropy inefficiency of RecycleWidening dominates the efficiency gained by recycling the reject case in Algorithms 4.3 and 4.4.

The following result proves the correctness of the recycling rule.

Proposition 4.1. *Let $N = n\lfloor 2^W/n \rfloor \in (2^{W-1}, 2^W]$ be the target number of uniform outcomes including the recyclable state, and let $U \sim \text{Uniform}[0, 2^W)$. Let $R \coloneqq \{u \in [0, 2^W) \mid uN \bmod 2^W < 2^W - N\}$ be the set of reject outcomes and $f(u) \coloneqq u - \lfloor uN/2^W \rfloor$. Then $[f(U) \mid \{U \in R\}] \sim \text{Uniform}[0, 2^W - N)$.*

Proof. First observe that $f(0) = 0$. Next, note that $\lfloor (x+1)N/2^W \rfloor = \lfloor xN/2^W \rfloor \iff xN \bmod 2^W < 2^W - N$. Otherwise, $\lfloor (x+1)N/2^W \rfloor = \lfloor xN/2^W \rfloor + 1$. Therefore, $f(x+1) = f(x) + 1$ if and only if $x \in R$; otherwise, $f(x+1) = f(x)$. It follows that

$$f(x) = |[0, x) \cap R|. \tag{4.3}$$

Lemire (2019; Lemma 4.1) shows that $|R| = 2^W \bmod N = 2^W - N$, so $f(x)$ ranges over $[0, 2^W - N)$ as x ranges over R. The result is uniform because each value in $[0, 2^W - N)$ is attained exactly once by some $x \in R$. ∎

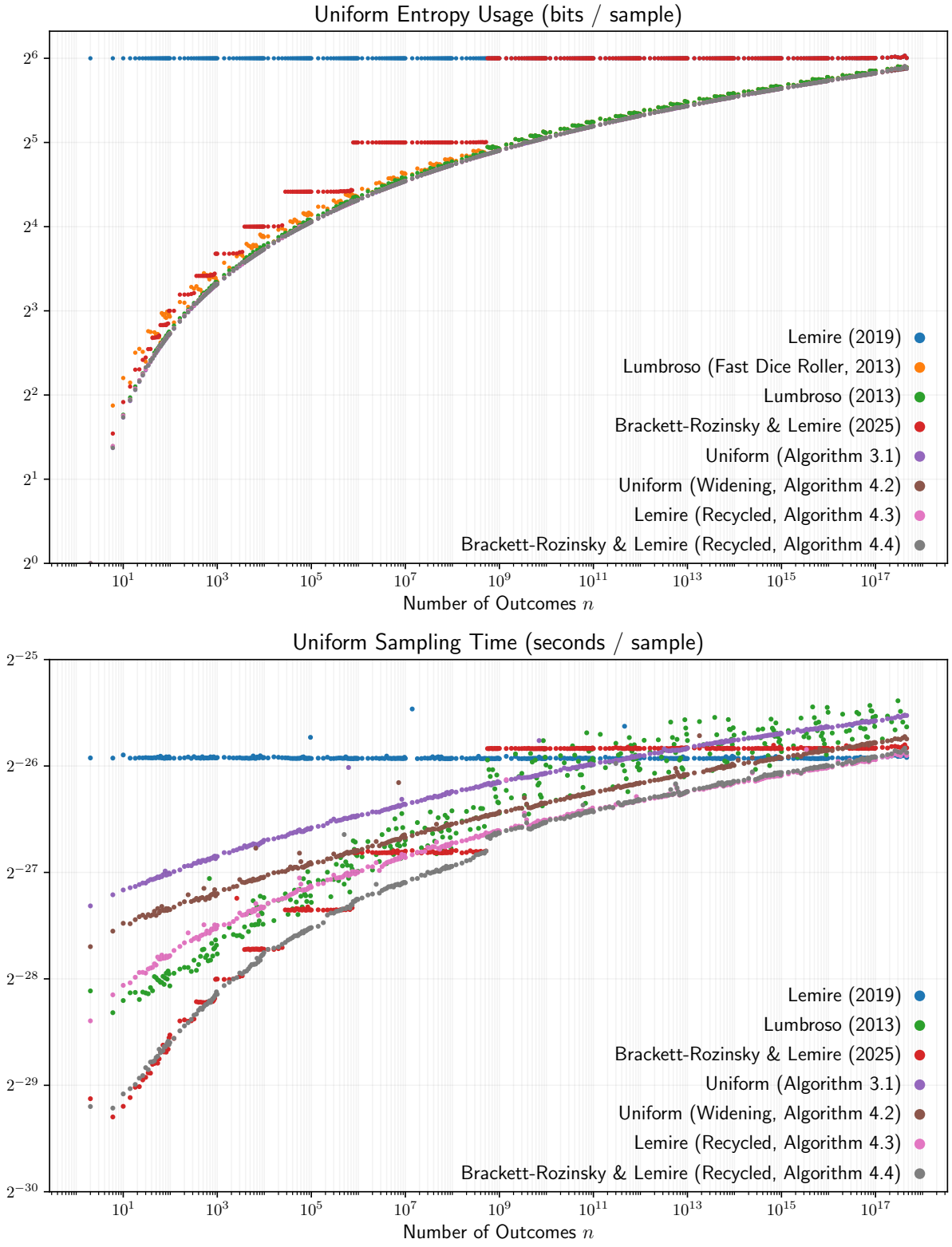


Figure 4.3: Benchmark comparison of entropy consumption and sampling time using various optimized sampling algorithms for discrete uniforms. Algorithms 4.2–4.4 are novel to this work.

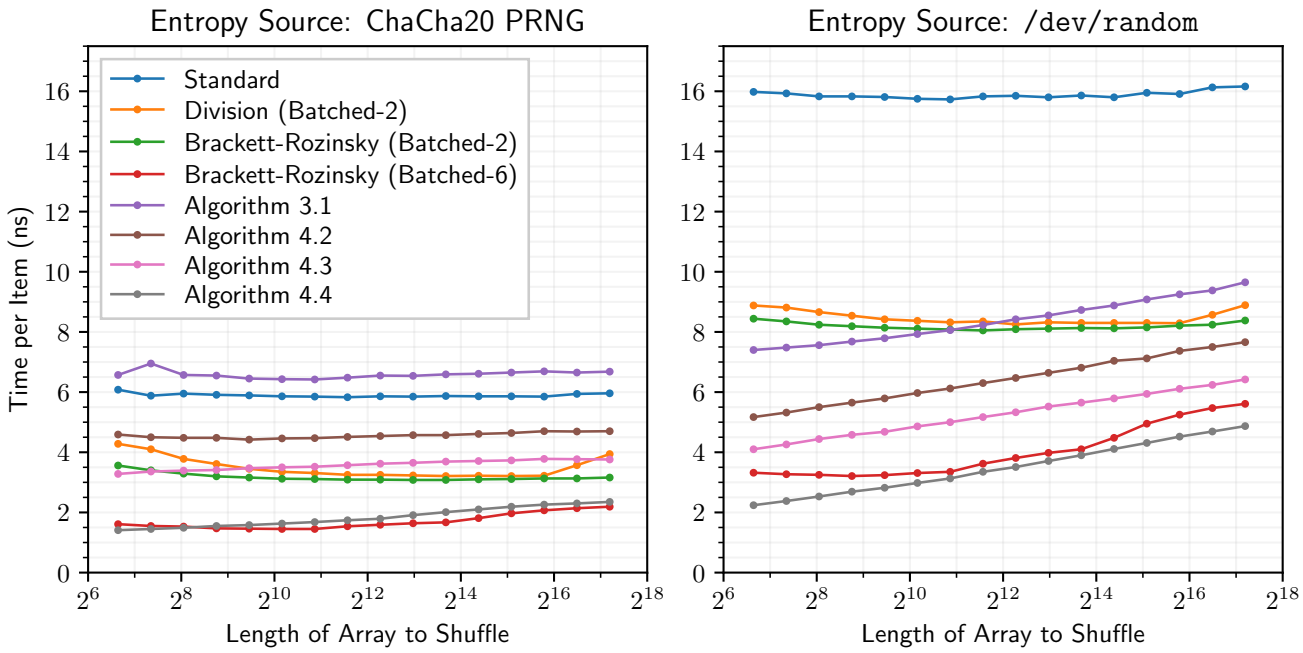


Figure 4.4: Runtime of Fisher and Yates (1953) shuffle for generating random permutations, using eight different sampling algorithms for discrete uniforms and two different entropy sources (ChaCha20 and `/dev/random`). The first four samplers in the legend are from the benchmark set in Brackett-Rozinsky and Lemire (2025). The remaining samplers are our Algorithms 3.1 and 4.2–4.4; the last three use randomness recycling techniques that are optimized for word-based operations.

Algorithm 4.3 shows the implementation of this method. The exact same method can be used to recycle leftover randomness from the batched sampler of Brackett-Rozinsky and Lemire (2025), by simply adding one uniform to the end of the batch, as shown in Algorithm 4.4. This modification of the batched sampler with randomness recycling is competitive with the original method, even though it introduces one extra division per batch and computes the product of the ranges in the batch every time.

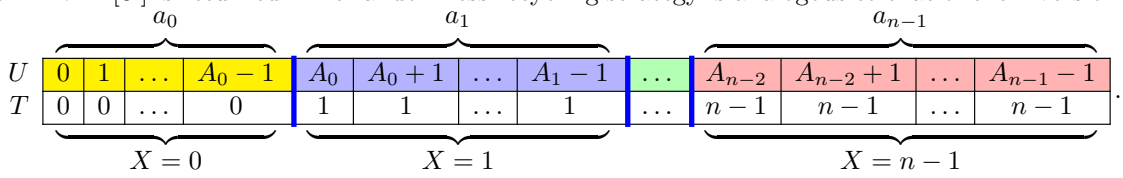
Figure 4.3 shows the improvements enabled by our randomness recycling techniques (Algorithms 4.2–4.4) in terms of the entropy cost and wall-clock sampling time, when used to repeatedly generate many uniforms over a fixed range n (for n between 2 and 10^{17}). The plot shows that Algorithms 4.2–4.4 deliver performance improvements over both Algorithm 3.1 (from §4), the prior methods of Lemire (2019); Brackett-Rozinsky and Lemire (2025), and an optimized version of the FDR (Lumbroso, 2013). Particularly notable in Figure 4.3 is the runtime performance of Algorithm 4.4, which consistently retains fast sampling time across all values of n.

4.3.3 Application to Generating Random Permutations. Brackett-Rozinsky and Lemire (2025) show that variable-sized batching and mixed-radix decomposition using the method of Lemire (2019) to eliminate divisions yields a state-of-the art sampler for multiple uniforms $(U_0, U_1, \ldots, U_{k-1})$. They apply their sampler to generate random permutations via the Fisher-Yates shuffle (Durstenfeld, 1964; Fisher and Yates, 1953), whose key subroutine is efficiently sampling $U_i \sim \text{Uniform}[0, k-i)$ for $i = 0, \ldots, k-1$.

Our methods with randomness recycling require more operations and are therefore comparatively slower when the entropy source is a fast pseudorandom number generator (PRNG) that quickly returns (pseudo)random coin tosses. However, when using a more computationally expensive entropy source such as a cryptographically secure PRNG that is continuously seeded from system entropy (e.g., `/dev/random` on Linux), the entropy consumption becomes a significant factor in the overall runtime. In this setting, our randomness recycling algorithms, which consume almost the minimum possible expected entropy, can surpass the performance of the Brackett-Rozinsky and Lemire method. Figure 4.4 shows a comparison of the runtime per shuffled item (in nanoseconds) using four baseline algorithms (Standard, Division Batched-2, BR Batched-2, BR Batched-6) for uniform sampling from Brackett-Rozinsky and Lemire with Algorithms 3.1 and 4.2–4.4 from this article.

5 Randomness Recylers for General Distributions. In §3.4 we presented a randomness recycling method for sampling general (nonuniform) distributions using inversion sampling. Randomness recycling can also be used with many other samplers for general distributions, summarized in Table 1.2. We describe these techniques here. Whereas the proof of Theorem 1.5 in §3.6 used Algorithm 3.2 in the witness, in principle any one of the forthcoming algorithms could be used instead. When used to generate a single sample, the algorithms in this section may be far from entropy optimal. When used to generate a random sequence, however, they achieve nearly optimal amortized entropy in the sense of Theorem 1.5, by virtue of the randomness recycling rules.

5.1 Lookup-Table Sampling. For a rational discrete distribution $p \coloneqq (a_0, \dots, a_{n-1})/A$ with moderately sized weight sum A, the lookup-table method (Devroye, 1986; p. 770) is a practical way to eliminate the logarithmic cost of binary search in inversion sampling. A table $T[0\,..\,A-1]$ of size A is constructed such that each integer $i \in [0, n)$ is stored exactly a_i times. To generate a sample from p, a uniform variate $U \sim \text{Uniform}[0, A)$ is generated and then $X \leftarrow T[U]$ is returned. The randomness recycling strategy is analogous to that of the inversion sampler.



The top array shows all possible values of $U \sim \text{Uniform}[0, A_{n-1})$ and the bottom array shows the lookup table T. Since T has length $A \equiv A_{n-1}$ and must be stored in memory, lookup-table sampling scales exponentially with the number of bits needed to encode p.

Algorithm 5.1 Lookup table sampling with randomness recycling

Require: Positive integers $a_0, \dots, a_{n-1}$ with sum A
Ensure: Random sample $X \sim \text{Discrete}(a_0, \dots, a_{n-1})$

procedure LOOKUP($a_0, \dots, a_{n-1}$)
  Set $A_{-1} \leftarrow 0$ and $A_i \leftarrow a_0 + \cdots + a_i$ for $i = 0, \dots, n-1$ ▷ *prefix sums (if not given)*
  Construct array $T[0\,..\,A-1]$ where $i \in [0, n)$ is stored a_i times. ▷ *lookup table*
  $U \sim$ UNIFORM(A) ▷ *draw uniform variate (Algorithm 3.1)*
  Let $X \leftarrow T[U]$ ▷ *lookup*
  $(Z', M') \leftarrow (U - A_{i-1}, a_i)$ ▷ *extract uniform state*
  RECYCLE(Z', M') ▷ *(Algorithm 2.1)*
  return X

5.2 Alias Sampling. The alias method (Walker, 1977) is a state-of-the-art sampler that avoids the exponential space complexity of the lookup-table method while retaining its extremely fast runtime. In the preprocessing phase, the target distribution $p = (a_0, \dots, a_{n-1})/b$ is used to compute (i) the alias outcomes $\mathbf{z} \coloneqq (z_0, \dots, z_{n-1})$, where $z_i \in [0, n-1] \setminus \{i\}$; and (ii) the "no alias" odds $\mathbf{w} \coloneqq (w_0, \dots, w_{n-1})$, where $w_i \in [0, b]$ (Vose, 1991). Equipped with these data structures, the generation phase is as follows:

- Draw a uniform index $I \sim \text{Uniform}[0, n)$.
- Draw $B \sim \text{Bernoulli}(w_I/b)$; if $B = 1$ then return $X \leftarrow I$, else return $X \leftarrow z_I$.

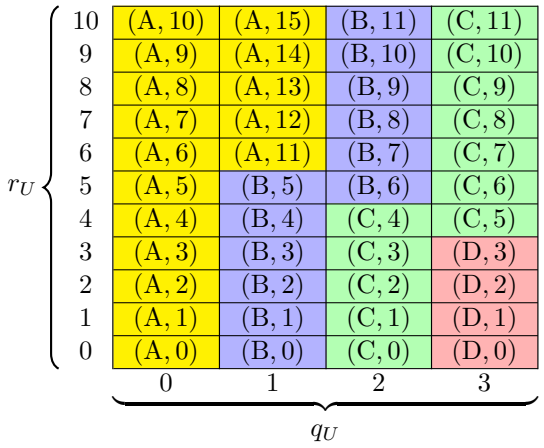
For any discrete distribution p, it is always possible to construct an alias data structure that guarantees the return value X is distributed according to p (Devroye, 1986; Theorem 4.1). That is, for a given outcome $j \in [0, n)$, let $k_1, \dots, k_m$ be the indices for which j is an alias (i.e., $z_{k_i} = j$, for $i = 1, \dots, m$). The alias method guarantees that

$$\mathbb{P}(X = i) \coloneqq 1/n \cdot w_j/b + \sum_{i=1}^{m} 1/n \cdot (b - w_{k_i})/b = a_j/b \qquad (0 \le j < n). \tag{5.1}$$

A naïve approach to randomness recycling with the alias sampler is to sample $I \leftarrow$ UNIFORM(n) using Algorithm 3.1 and $B \leftarrow$ INVERSION($(w_I, b - w_I)$) using Algorithm 3.2. However, this approach does not optimally recycle randomness. Instead, the optimal recycling rule is as follows:

- Sample $U \sim \text{Uniform}[0, nb)$ using Algorithm 3.1.
- Write $U = q_U b + r_U$, where $q_U := \lfloor U/b \rfloor$ and $r_U := U \bmod b$. Here $I \equiv q_U \sim \text{Uniform}[0, n)$ furnishes a uniformly chosen index in $[0, n)$ and $r_U \sim \text{Uniform}[0, b)$ furnishes the uniform needed to generate the Bernoulli(w_I/b) variable.
- If $r_U < w_I$, then return $X \leftarrow I$, else return $X \leftarrow z_I$; and recycle a uniform state (Z', M') with $M' = na_X$ by using the auxiliary array $\mathbf{c}$ described in the remainder of this section.

Conditional on the event $\{X = i\}$, the selected cell in the alias table is uniform over all na_i ways the label i could have been achieved. Applying Proposition 2.5 gives the recycling rule, which is visualized in the following example.



$\mathbf{z}$	$z_0 = *,$	$z_1 = \text{A},$	$z_2 = \text{B},$	$z_3 = \text{C}$
$\mathbf{w}$	$w_0 = 11,$	$w_1 = 6,$	$w_2 = 5,$	$w_3 = 4$
$\mathbf{a}$	$a_0 = 4,$	$a_1 = 3,$	$a_2 = 3,$	$a_3 = 1$
$\mathbf{c}$	$c_0 = *,$	$c_1 = 5,$	$c_2 = 1,$	$c_3 = 1$

Figure 5.1: Randomness recycling with the alias sampler for the target distribution $p := \{\text{A} \mapsto 4 =: a_0, \text{B} \mapsto 3 =: a_1, \text{C} \mapsto 3 =: a_2, \text{D} \mapsto 1 =: a_3\}/11$.

Let $p := \{\text{A} \mapsto 4 =: a_0, \text{B} \mapsto 3 =: a_1, \text{C} \mapsto 3 =: a_2, \text{D} \mapsto 1 =: a_3\}/11$ be the target distribution, where we use the symbols $\text{A} = 0, \text{B} = 1, \text{C} = 2, \text{D} = 3$ to avoid confusion between the outcome labels and other integers in the alias data structures. Figure 5.1 shows the corresponding data structures for alias sampling with randomness recycling, where $*$ denotes an arbitrary value. Note that the (exponentially sized) alias table itself is never stored in memory, but is represented compactly as $(\mathbf{z}, \mathbf{w}, \mathbf{a}, \mathbf{c})$. In this example, consider two cases:

- Suppose that $(q_U, r_U) = (1, 2)$ specifies the cell $(\text{B}, 2)$. Then conditioned on $\{X = \text{B}\}$, the value 2 is uniformly distributed over $[0, 12)$, i.e., the indices of all other cells labeled B in the table. We can directly recycle $(Z', M') = (2, 12)$ into the global uniform state, where $Z' = r_U = 2$ and $M' = na_1 = 12$.
- Suppose that $(q_U, r_U) = (1, 7)$ specifies the cell $(\text{A}, 12)$. Then conditioned on $\{X = \text{A}\}$, the value 12 is uniformly distributed over $[0, 16)$, i.e., the indices of all other cells labeled A in the table. We can thus recycle $(Z', M') = (12, 16)$ into the uniform state. To compute the value $Z' = 12$ from $(q_U, r_U) = (1, 7)$, we first add the number of A cells to the left of the current column (11), then subtract the number of B cells in the current column (6) to obtain the offset $c_1 = 11 - 6 = 5$, and finally add $r_U = 7$ to obtain $Z' = c_1 + r_U = 5 + 7 = 12$. The bound $M' = 16 = na_{z_1}$ is available as in the previous case.

To enhance the alias sampler with this randomness recycling rule, we construct an array $\mathbf{c} := (c_0, \ldots, c_{n-1})$ of the offsets needed to perform recycling as described above, which can be done in linear time. Algorithm 5.2 shows the resulting alias sampler with randomness recycling.

5.3 Discrete Distribution Generating Tree Sampling. A discrete distribution generating (DDG) tree is a universal computational model introduced by Knuth and Yao (1976) for describing any computable sampling algorithm that maps random bits to discrete outcomes. A DDG tree G is a complete, rooted binary tree where each leaf node has an outcome label $i \in \mathbb{N}$. DDG tree sampling operates as follows, starting from the root of G:

Algorithm 5.2 Alias sampling with randomness recycling

Require: Positive integers $a_0, \ldots, a_{n-1}$ with sum A
Aliases $\mathbf{z} \coloneqq (z_0, \ldots, z_{n-1})$
No alias odds $\mathbf{w} \coloneqq (w_0, \ldots, w_{n-1})$
Recycling offsets $\mathbf{c} \coloneqq (c_0, \ldots, c_{n-1})$
Ensure: Random sample $X \sim \text{Discrete}(a_0, \ldots, a_{n-1})$

procedure ALIAS($(a_0, \ldots, a_{n-1})$, $\mathbf{z}$, $\mathbf{w}$, $\mathbf{c}$)
  $U \leftarrow \text{UNIFORM}(A \cdot n)$ ▷ *draw uniform variate (Algorithm 3.1)*
  $(q, r) \leftarrow \text{DIVMOD}(U, A)$ ▷ *compute cell (q, r) in alias table*
  if $r < w_q$ **then** ▷ *no alias: select outcome q*
    $(Z', M') \leftarrow (r, n \cdot a_q)$ ▷ *extract uniform state*
    RECYCLE(Z', M') ▷ *(Algorithm 2.1)*
    return q
  else ▷ *alias: select outcome z_q*
    $(Z', M') \leftarrow (r + c_q, n \cdot a_{z_q})$ ▷ *extract uniform state*
    RECYCLE(Z', M') ▷ *(Algorithm 2.1)*
    return z_q

(D1) Obtain a fair coin toss $B \leftarrow \text{FLIP}(1)$. If $B = 0$, then visit the left child of the current node; else if $B = 1$, then visit the right child.

(D2) If the visited child node is a leaf node, then return its label; else go to (D1).

The set of leaf nodes in a DDG tree G is denoted $\mathcal{L}(G)$. The depth and label of any leaf $l \in \mathcal{L}(G)$, are denoted $d(l) \geq 0$ and $\ell(l) \in \mathbb{N}$, respectively. With these notations, a random variable $X \sim G$ obtained by DDG tree sampling (D1)–(D2) has the following distribution:

$$\mathbb{P}(X = i) = \sum_{l \in \mathcal{L}(G)} 2^{-d(l)} \cdot \mathbb{I}[\ell(l) = i]. \tag{5.2}$$

The expected number of coin tosses used to sample X is equal to the average depth of a leaf:

$$\mathbb{E}\left[T_G\right] = \sum_{l \in \mathcal{L}(G)} 2^{-d(l)} \cdot d(l). \tag{5.3}$$

As discussed in §1.3, Knuth and Yao (1976) show how to construct an entropy-optimal DDG tree G^* for any distribution distribution $p \coloneqq (a_0, \ldots, a_{n-1})/A$ whose expected number of coin tosses (5.3) is the least possible. This optimal tree G^* is constructed by placing a leaf labeled i at depth j if and only if the jth bit in the binary expansion of a_i/A is 1. Explicitly constructing G^* can require exponential space in the number of bits needed to encode p (Saad et al., 2020b; Theorem 3.6). An alternative approach is to incrementally traverse G^* without explicitly constructing the tree (e.g., Knuth and Yao (1976; p. 384), Saad and Lee (2025; Algorithm 1)), although steps (D1) and (D2) become more complicated as the leaves and labels must be created during sampling.

The Fast and Amplified Loaded Dice Roller (Saad et al., 2020a; Draper and Saad, 2025) algorithms are near entropy-optimal DDG tree samplers that use rejection sampling to reduce the space complexity of explicitly stored DDG trees. These methods can be interpreted as a compression of the lookup-table method from §5.1 using power-of-two block sizes. The key idea is to build an entropy-optimal DDG tree G' for $p' \coloneqq (a_0, \ldots, a_{n-1}, a_n)/2^k$ where 2^k is a power of two that is larger than the sum of weights A and $a_n = 2^k - A$ is a "reject" outcome. Saad et al. (2020b) and Draper and Saad (2025) show that with a linearithmic sized tree G' it is possible to achieve expected costs less than $H(p) + 6$ and $H(p) + 2$, respectively, where the latter coincides with the entropy-optimal Knuth and Yao (1976) range $[H(p), H(p) + 2)$. Figure 5.2 shows an example DDG tree G, where the colors of the leaf nodes are suggestive of the recycling strategy.

Recycling Rule. The recycling rule for DDG tree sampling extracts a uniform state (Z', M') from a nonuniform state using Proposition 2.5. We recycle a draw from the distribution of *depth* of the leaf nodes conditioned on

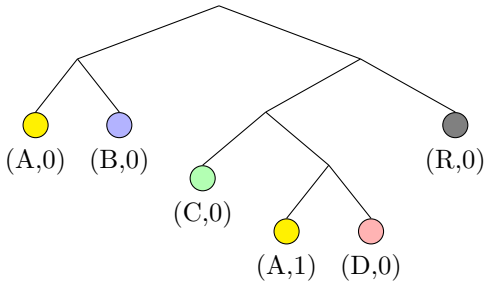


Figure 5.2: DDG tree for the target distribution $p := \{\mathrm{A} \mapsto 5, \mathrm{B} \mapsto 4, \mathrm{C} \mapsto 2, \mathrm{D} \mapsto 1\}/12$ using the rejection-based Fast Loaded Dice Roller (Saad et al., 2020a) method.

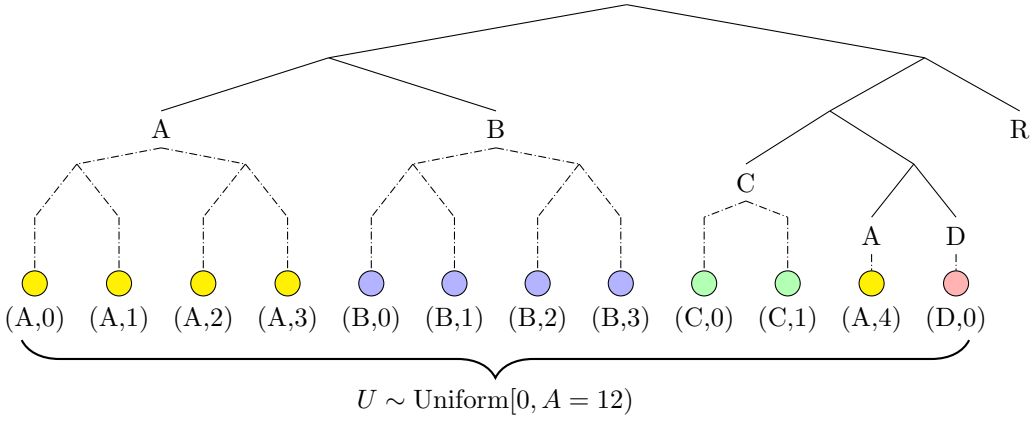


Figure 5.3: Randomness recycling for the distribution in Figure 5.2 by using a "left-packed" DDG tree whose leaves all live at the same level. This (exponentially sized) tree can be compactly represented by using the data structures listed in Algorithm 5.3.

the *label* of the visited leaf node, which gives a nonuniform distribution $\mathrm{Discrete}(\mathbf{w})$ as in Proposition 2.5 whose weights w_i are distinct powers of two corresponding to the set bits in the binary expansion of the label probability.

More specifically, consider any entropy-optimal DDG tree G whose output distribution is $p = (a_0, \ldots, a_{n-1}, a_n)/2^k$, where a_n is a "reject" outcome that implicitly denotes a back-edge to the root. Each leaf node in G has a label $i \in [0, n]$. Let $c_i \geq 1$ denote the (finite) number of leaves in G with label i. Further, let $0 \leq d_{i1} < d_{i2} < \cdots < d_{ic_i}$ denote the depths of the leaves with label i, which are distinct since G is entropy optimal. Conditioned on $\{X = i\}$, the distribution over the possible leaves with label i is

$$\mathbf{w}_i = (w_{i1}, w_{i2}, \ldots, w_{ic_i}) = (2^{d_{ic_i} - d_{i1}}, 2^{d_{ic_i} - d_{i2}} \ldots, 1) \propto (2^{-d_{i1}}, 2^{-d_{i2}} \ldots, 2^{-d_{ic_i}}) \tag{5.4}$$

Conditioned on returning $\{X = i\}$, DDG tree sampling selects one of these c_i leaves as an exact nonuniform draw $X' \sim \mathrm{Discrete}(\mathbf{w}_i)$. For example, in Figure 5.2, the (unnormalized) distribution over depths given $\{X = \mathrm{A}\}$ is given by $\mathbf{w}_\mathrm{A} = (4, 1) \propto (2^{-2}, 2^{-4})$. The randomness recycling rule is immediate from Proposition 2.5, which shows how to recycle the nonuniform state X' using a fresh draw $U \sim \mathrm{Uniform}[0, w_{X'})$. The challenge with this approach is that it is difficult to guarantee the W-bit global uniform state will not overflow, when using Algorithm 2.1 (although Algorithm 4.1 can provide an alternative solution to this concern).

Efficient Implementation. To avoid explicitly merging a nonuniform state, it is more convenient to implement randomness recycling on a (hypothetical) DDG tree G' (Figure 5.3) whose leaves live at the same level, as follows:

- Replace each leaf node in G at depth d with a subtree that terminates at the maximum depth of G; and set all the labels of the new leaf nodes to be the same label as the original leaf.
- Pack all the non-reject outcomes to the left of the resulting tree.
- Use $U \sim \mathrm{Uniform}[0, A)$ to select one of the A leaves at the final level.

Algorithm 5.3 DDG tree sampling with randomness recycling

Require: Target distribution $(a_0, \ldots, a_{n-1})$ with sum A
DDG tree depth $k \geq 0$
Leaf counts per level $\mathbf{L} \coloneqq (L_0, \ldots, L_k)$
Leaf labels $\mathbf{H} \coloneqq ((H_{0,0}, \ldots, H_{0,L_0-1}), \ldots (H_{k,0}, \ldots, H_{k,L_k-1}))$
Ensure: Random sample $X \sim \text{Discrete}(a_0, \ldots, a_{n-1})$

```
1:  procedure DDG((a_0, ..., a_{n-1}), k, L, H)
2:      U ← UNIFORM(A)                          ▷ draw uniform variate (Algorithm 3.1)
3:      (d, v) ← (0, 0)                         ▷ initialize depth and value
4:      while true do                           ▷ loop up to k times
5:          if v < L_d then                     ▷ hit leaf node
6:              X ← H_{d,v}                     ▷ label at leaf node
7:              Z' ← a_X − (a_X mod 2^{k+1−d}) + (U mod 2^{k−d})   ▷ extract uniform value
8:              M' ← a_X                        ▷ extract uniform upper bound
9:              RECYCLE(Z', M')                 ▷ (Algorithm 2.1)
10:             return X                        ▷ return the label
11:         v ← 2 · (v − L_d) + (⌊U/2^{k−1−d}⌋ mod 2)   ▷ visit random child
12:         d ← d + 1                           ▷ increment depth
```

Algorithm 5.3 shows the corresponding algorithm for sampling a "left-packed" DDG tree of this form whose leaves are all at the same level. Because the leaves in Figure 5.3 are all at the same level, the recycling rule reverts to the typical uniform case as in the inversion, lookup table, and alias sampler implementations from the previous sections, rather than the nonuniform case using Proposition 2.5 if the tree were of the form in Figure 5.2.

Algorithm 5.3 uses fast bit operations to identify the sampled leaf X and implement the recycling rule without actually constructing the hypothetical tree G'. In particular, when a leaf node with label X is encountered (line 5), a uniform $Z' \sim \text{Uniform}[0, M')$ with $M' \coloneqq a_X$ is extracted. The value of Z' is computed by first choosing one of the "hypothetical" leaf nodes below the current leaf (dashed edges in Figure 5.3) using $U \bmod 2^{k-d}$, and then computing the offset $a_X - (a_X \bmod 2^{k+1-d})$ which is the sum of all hypothetical leaves with label X to the left.

Remark 5.1. For a distribution whose sum A of weights is very close to a power of two, as for ALDR (Draper and Saad, 2025) or any dyadic distribution, the method on line 2 of Algorithm 5.3 to sample a uniform over A can be replaced by a method using rejection sampling via a uniform over $2^{\lceil \log(A) \rceil}$. Although this change slightly increases the entropy cost, it may improve the runtime, because the division operations in Algorithm 3.1 can be replaced by bitwise operations when the target uniform range is a power of two.

5.4 Benchmark Evaluations. Figure 5.4 shows how the randomness recycling strategies described in this section yield improvements in entropy cost and wall-clock runtime as compared to the baseline versions without randomness recycling. Each row in Figure 5.4 shows the results for a specific sampling algorithm. The first column shows the preprocessing time in seconds, the second column shows the entropy usage in bits/sample, and the third column shows the sampling time in seconds/sample. In each panel, every dot shows the measurements for a given probability distribution over n outcomes (x-axis) with sum of weights $m = 1000000$, amortized over an i.i.d. sequence of one million samples. In the majority of cases, randomness recycling introduces minimal runtime overhead in the preprocessing time, while enabling lower entropy consumption and sampling time. A software library in the C programming language containing the algorithms described in this paper is available at https://github.com/probsys/randomness-recycling.

5.5 Discrete Gaussian Sampling. The discrete Gaussian sampler of Canonne et al. (2020) is a prominent example of an exact sampler over the integers. The algorithm samples a discrete Gaussian by calling two primitive samplers: discrete uniform and Bernoulli. We adapted the reference Python implementation by the authors (available at https://github.com/IBM/discrete-gaussian-differential-privacy) by replacing all the calls to uniform and Bernoulli with their randomness-recycled variants from Algorithms 3.1 and 3.2. Figure 5.5 shows that randomness recycling reduces the entropy cost of the algorithm by up to 10x compared to the original version.

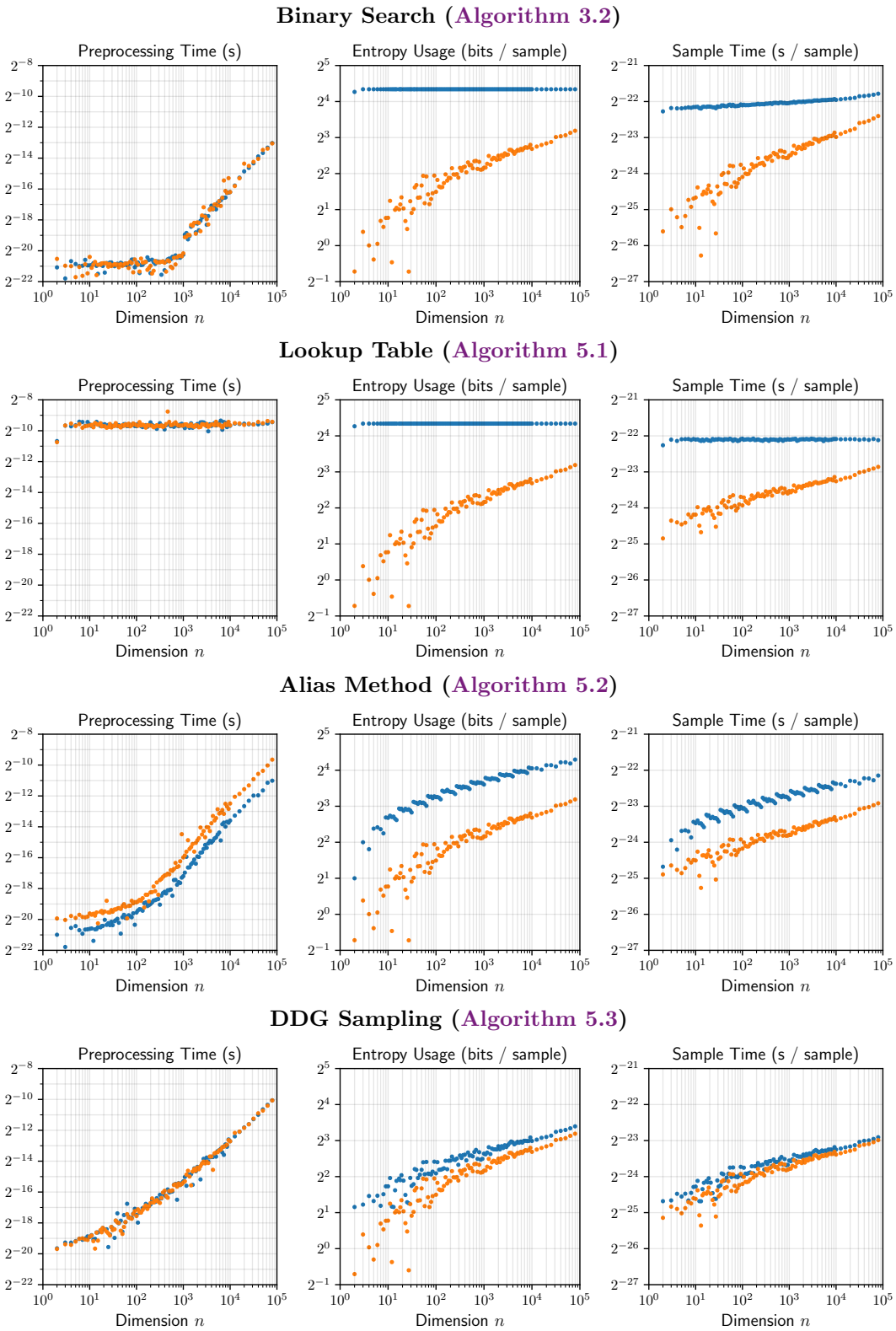


Figure 5.4: Benchmark comparison of preprocessing time, entropy consumption, and sampling time for the binary search, lookup table, alias, and DDG tree samplers; with randomness recycling (orange ●) and without (blue ●), on a range of rational discrete distributions whose probabilities have common denominator equal to 10^6.

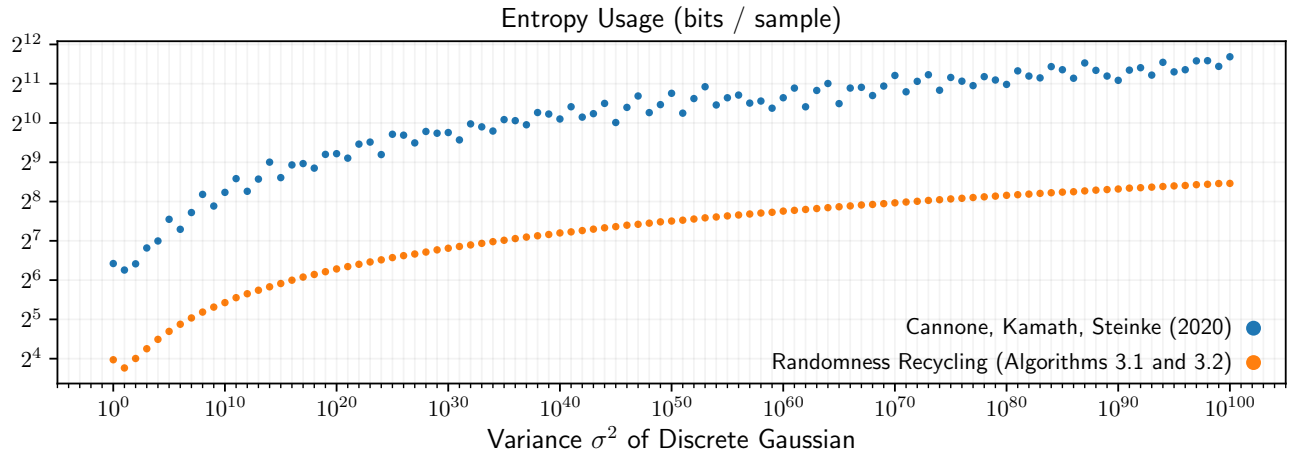


Figure 5.5: Applying randomness recycling to sample from discrete Gaussian distributions.

6 Related Work. Existing approaches to online random sampling that are most closely related to the problem setting in this article are surveyed in §1.3. Here we describe additional literature that studies variants of the sampling problem and literature on randomness recycling in other settings.

Converting Entropy Streams. Converting entropy streams from a given input distribution to a desired output distribution is widely studied in computer science. This article focuses on the case of online random sampling using an entropy source that provides an i.i.d. sequence of fair coin tosses, to generate output variables with any (nonuniform) rational distribution. Knuth and Yao (1976) introduce a complexity theory for studying nonuniform sampling algorithms given i.i.d. fair coins. Saad et al. (2020a) and Draper and Saad (2025) develop algorithms that are close to the Knuth and Yao entropy-optimal rate using linearithmic, instead of exponential, memory; but only consider the single-sample case. Saad and Lee (2025) give a deterministic implementation of the nondeterministic online sampling algorithm from Knuth and Yao (1976) (see also Algorithm A2), and efficient algorithms that guarantee zero numerical error when the probabilities are floating-point numbers. These works do not use randomness recycling and focus on achieving near-optimal entropy cost for generating a *single* random sample, whereas our method achieves a near-optimal *amortized* entropy cost for sampling an infinite sequence.

Several articles have studied variants of the entropy conversion problem under different assumptions. One popular variant is the problem of extracting unbiased coin tosses from an i.i.d. source with an arbitrary but known distribution (Elias, 1972; Abrahams, 1996; Roche, 1991; Pae, 2005; Kozen, 2014; Pae, 2015, 2020). Han and Hoshi (1997) and Kozen and Soloviev (2022) explore very general reductions for converting k-sided dice dice rolls into n-sided dice rolls; the interval method in the former article is a particularly elegant technique that allows both the input and output sequences that are be i.i.d., Markov, or arbitrary stochastic processes. Von Neumann (1951) describes a simple method that extracts i.i.d. fair coins using a source that emits i.i.d. dice rolls with unknown bias, which was explored by several additional authors (Hoeffding and Simons, 1970; Stout and Warren, 1984; Cohen, 1985; Peres, 1992; Pae and Loui, 2006). Elias (1972) and Blum (1986) consider an unknown entropy source whose stream is subject to a stationary Markov chain. Some methods (Elias, 1972; Peres, 1992; Cicalese et al., 2006) produce a variable-length output instead of a single (fixed-length) output at each invocation, depending on the specific bit pattern from the source. Han and Verdú (1993) and Vembu and Verdú (1995) allow the sampler to produce approximate samples from the target distribution up to a given statistical error tolerance, providing information-theoretic asymptotic rates. Saad et al. (2020b) provide an efficient algorithm that finds an optimal (in terms of minimizing any f-divergence) rational approximation to an arbitrary distribution, generalizing the results of Böcherer and Geiger (2016) who considered the total variation and Kullback-Leibler divergence.

Specialized Online Random Samplers. Kozen and Soloviev (2022) study the problem of exactly transforming i.i.d. random streams in an entropy-efficient manner while using limited memory, with a focus on the composition of these transformation protocols. The input and output streams may consist of discrete uniforms or arbitrary discrete distributions with finite support. For the case of transforming uniform inputs into i.i.d. uniform outputs

or to i.i.d. samples from a rational discrete distribution, they give protocols that use $O(1/\varepsilon)$ space to achieve an expected entropy inefficiency of at most $\varepsilon > 0$ bits per sample. They also note that no finite-memory algorithm can produce exact samples from an irrational target distribution, but such irrational distributions can be sampled via an infinite sequence of rational distributions; an overall entropy inefficiency of ε bits per sample can be achieved using $O(1/\varepsilon)$ space for the state of each of these rational samplers. Our results improve on these space bounds exponentially, requiring only $O(\log(1/\varepsilon))$ space to achieve the same entropy inefficiency. The difference arises in our use of a uniform random state that is maintained across rounds, whereas Kozen and Soloviev exclusively use "restart protocols" that fully reset the state after emitting samples. In an alternative setting where input entropy stream is allowed to be nonuniform, Kozen and Soloviev (2022) achieve a space bound of $O(\log(1/\varepsilon)/\varepsilon)$ in the fully arbitrary case, and $O(1/\varepsilon)$ using a cleverly optimized construction when the entropy source emits a biased coin with weight $1/r$ for some integer $r \geq 3$.

In concurrent work, Shao and Wang (2025) address the more specific problem of generating exact i.i.d. samples from a fixed discrete distribution using bounded memory and with near-optimal entropy. Whereas they consider only repeated i.i.d. samples from a prespecified distribution, we show how to sample from an arbitrary sequence of distributions specified in an online setting. They also restrict attention to the inversion method, whereas we apply randomness recycling to a variety of discrete sampling methods (Table 1.2). Their method bears some similarity to Algorithm 3.2 in our work, using amplified weights in the sense of Draper and Saad (2025) to avoid the need for division when the randomness source is a stream of bits. The recycling method used by Shao and Wang (2025) requires $O(1/\varepsilon)$ space and computation overhead to achieve a desired ε entropy bound, compared to our $O(\log(1/\varepsilon))$ from Theorem 1.5. Namely, they convert a large uniform state of size $2\log(d)/\varepsilon$ bits into i.i.d. bits (which wastes up to 2 bits) after every $2/\varepsilon$ samples, to achieve an amortized inefficiency of ε bits per sample. Notably, the $2/\varepsilon$ steps in Shao and Wang (2025) coincide exactly with the batch size n in (1.21) needed to achieve a given efficiency ε when performing batched sampling without carrying over any state between batches, this time to amortize the optimal Knuth and Yao toll of 2 bits in converting from fair coin tosses to a general random state (instead of vice versa).

Uniform Sampling. The uniform sampler in §3.3 is closely related to a method described by Willekens (2004) and rediscovered by Ömer and Pacher (2014). Mennucci (2010) empirically analyzes this method for scaling uniform random number generators, and performs extensive numerical tests with specialized optimization using bitwise operations on different machine architectures. Our contributions extend these previous works by

- using randomness recycling to efficiently generate a random sequence whose expected amortized entropy cost is arbitrarily close to the information-theoretic lower bound (Theorem 1.5);
- theoretically analyzing the entropy loss of randomness recycling samplers (§3);
- developing randomness recycling for more specialized uniform samplers that incorporate widening and batching optimizations (§4), which improve performance (Figures 4.3 and 4.4); and
- leveraging randomness recycling for sampling general nonuniform distributions (§5), which improves the runtime and entropy characteristics of diverse algorithms (Figure 5.4).

Randomness Recycling. The term "randomness recycler" appears to have originated in Fill and Huber (2000), who introduce the concept as an exact (perfect) sampling technique that stands in contrast to approximate sampling using Markov chains. Fill and Huber show how to apply randomness recycling in challenging combinatorial settings such as generating random independent sets, random graph colorings, Ising models, random cluster models, and self-organizing lists in expected linear time. These randomness recycling algorithms have better time complexity than exact rejection sampling and are more accurate than approximate sampling using Markov chains. The authors note that randomness recycling is not universally applicable in all scenarios where Markov chains are used, but it can efficiently generate perfect samples in linear time for challenging combinatorial problems. Our work develops randomness recycling techniques for sampling an online random sequence, to obtain improvements in space, time, and entropy compared to the best-known existing approaches described in §1.3.

Impagliazzo and Zuckerman (1989) describe randomness recycling techniques in the context of bounded-error probabilistic polynomial time (BPP) algorithms. The recycled random state does not satisfy the independence invariant (I) and the sampler is allowed to be approximate. The authors show how to run a BPP algorithm multiple times while using (approximately independent) recycled coin tosses from applying hash functions to more efficiently amplify the correctness probability as compared to using fresh coin tosses on each trial.

7 Remarks.

Tightness of Space-Entropy Tradeoff Bound. Algorithm 3.1 shows that it is possible to exactly sample from discrete distributions with weight sums bounded by d, in an online manner, within $\varepsilon > 0$ of the optimal entropy rate, using only $O(\log(d/\varepsilon))$ bits of space. We conjectured that this bound is tight.

Conjecture 1.6. *Any online random sampling algorithm that generates exact samples from a sequence of arbitrary discrete distributions, within $\varepsilon > 0$ of the information-theoretically optimal entropy rate, using a stream of i.i.d. coin tosses as the entropy source, requires $\Omega(\log(1/\varepsilon))$ bits of space for an auxiliary state that is carried over between rounds.*

It is difficult to reason about the space usage of all possible algorithms, but some examples can demonstrate the plausibility of this conjecture. The interval method (Han and Hoshi, 1997) achieves the entropy rate exactly, but it does not have bounded space, even after generating just a single sample (although the expected space usage can be bounded after one sample). For a DDG tree method using rejection back edges to represent the tree in finite space (Saad et al., 2020a), the recyclable state comes from the distribution of leaf given label, and if the tree has depth K, then there are at least 2^K possible denominators. Representing the recyclable state then requires at least K bits, even if it is, for example, converted to a uniform using Proposition 2.5. Further, the relationship between the DDG tree's rejection probability and its entropy inefficiency is analogous to the analysis in §3, so to achieve a rate of ε, the depth K must be roughly $\log(d/\varepsilon)$ (cf. Draper and Saad (2025; Proposition 3)).

In general, data related to the distribution, such as the weight sum, or the weight of the sampled outcome, requires roughly $\log(d)$ bits of space. However, we need not store exact data about the distribution; ε-approximate data can suffice to achieve an entropy rate within ε of optimal—an observation due to David G. Harris (pers. comm.). Generally, rejection-based methods for exact sampling with finite space seem to require $\log(1/\varepsilon)$ space to achieve a rate of ε. Our method's space usage is asymptotic to $2\log(d/\varepsilon)$ bits, and space as small as $\log(1/\varepsilon)$ is conceivable, but anything smaller seemingly could not fit the relevant recycled information.

Acknowledgements. The authors thank the anonymous referees for helpful suggestions in improving the manuscript. The authors also thank David G. Harris for observing that the space lower bound in Conjecture 1.6 need not depend on d. This material is based upon work supported by the National Science Foundation under Grant No. 2311983. Any opinions, findings, and conclusions or recommendations in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

Julia Abrahams. 1996. Generation of Discrete Distributions from Biased Coins. *IEEE Transactions on Information Theory* 42, 5 (Sept. 1996), 1541–1546. doi:10.1109/18.532895

Manuel Blum. 1986. Independent Unbiased Coin Flips from a Correlated Biased Source: A Finite State Markov Chain. *Combinatorica* 6, 2 (June 1986), 97–108. doi:10.1007/BF02579167

Nevin Brackett-Rozinsky and Daniel Lemire. 2025. Batched Ranged Random Integer Generation. *Software: Practice and Experience* 55, 1 (Jan. 2025), 155–169. doi:10.1002/spe.3369

Karl Bringmann and Konstantinos Panagiotou. 2017. Efficient Sampling Methods for Discrete Distributions. *Algorithmica* 79, 2 (Oct. 2017), 484–508. doi:10.1007/s00453-016-0205-0

Georg Böcherer and Bernhard C. Geiger. 2016. Optimal Quantization for Distribution Synthesis. *IEEE Transactions on Information Theory* 62, 11 (Nov. 2016), 6162–6172. doi:10.1109/TIT.2016.2610433

Clément L. Canonne, Gautam Kamath, and Thomas Steinke. 2020. The Discrete Gaussian for Differential Privacy. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (Advances in Neural Information Processing Systems, Vol. 33)*. Curran Associates Inc., Red Hook, NY, 15676–15688. arXiv:2004.00010

Ferdinando Cicalese, Luisa Gargano, and Ugo Vaccaro. 2006. A Note on Approximation of Uniform Distributions from Variable-to-Fixed Length Codes. *IEEE Transactions on Information Theory* 52, 8 (Aug. 2006), 3772–3777. doi:10.1109/TIT.2006.878151

Josh D. Cohen. 1985. *Fairing of Biased Coins in Bounded Time.* Technical Report YALEU/DCS/TR372. Yale University.

Luc Devroye. 1986. *Non-Uniform Random Variate Generation.* Springer-Verlag, New York. doi:10.1007/978-1-4613-8643-8

Luc Devroye and Claude Gravel. 2020. Random Variate Generation Using Only Finitely Many Unbiased, Independently and Identically Distributed Random Bits. *arXiv* 1502.02539v6 (Nov. 2020). doi:10.48550/arXiv.1502.02539

Thomas L. Draper and Feras A. Saad. 2025. Efficient Rejection Sampling in the Entropy-Optimal Range. *arXiv* 2504.04267v1 (April 2025). doi:10.48550/arXiv.2504.04267

Richard Durstenfeld. 1964. Algorithm 235: Random Permutation. *Commun. ACM* 7, 7 (July 1964), 420–421. doi:10.1145/364520.364540

Peter Elias. 1972. The Efficient Construction of an Unbiased Random Sequence. *Annals of Mathematical Statistics* 43, 3 (June 1972), 865–870. doi:10.1214/aoms/1177692552

James A. Fill and Mark Huber. 2000. The Randomness Recycler: A New Technique for Perfect Sampling. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE Press, Piscataway, NJ, 503–511. doi:10.1109/SFCS.2000.892138

Ronald Aylmer Fisher and Frank Yates. 1953. *Statistical Tables for Biological, Agricultural And Medical Research.* Hafner Publishing Company, London.

Te Sun Han and Mamoru Hoshi. 1997. Interval Algorithm for Random Number Generation. *IEEE Transactions on Information Theory* 43, 2 (March 1997), 599–611. doi:10.1109/18.556116

Te Sun Han and Sergio Verdú. 1993. Approximation Theory of Output Statistics. *IEEE Transactions on Information Theory* 39, 3 (May 1993), 752–772. doi:10.1109/18.256486

Wassily Hoeffding and Gordon Simons. 1970. Unbiased Coin Tossing with a Biased Coin. *The Annals of Mathematical Statistics* 41, 2 (April 1970), 341–352. doi:10.1214/aoms/1177697074

Mark Huber and Danny Vargas. 2024. Optimal Rolling of Fair Dice Using Fair Coins. *arXiv* 2412.20700 (Dec. 2024). doi:10.48550/arXiv.2412.20700

Russell Impagliazzo and David Zuckerman. 1989. How to Recycle Random Bits. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. IEEE Press, Piscataway, NJ, 248–253. doi:10.1109/SFCS.1989.63486

Donald E. Knuth and Andrew C. Yao. 1976. The Complexity of Nonuniform Random Number Generation. In *Algorithms and Complexity: New Directions and Recent Results*, Joseph F. Traub (Ed.). Academic Press, Inc., Orlando, FL, 357–428.

Dexter Kozen. 2014. Optimal Coin Flipping. In *Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday.* Lecture Notes in Computer Science, Vol. 8464. Springer, Cham, 407–426. doi:10.1007/978-3-319-06880-0_21

Dexter Kozen and Matvey Soloviev. 2022. Coalgebraic Tools for Randomness-Conserving Protocols. *Journal of Logical and Algebraic Methods in Programming* 125 (2022), 23 pages. doi:10.1016/j.jlamp.2021.100734

Daniel Lemire. 2019. Fast Random Integer Generation in an Interval. *ACM Transactions on Modeling and Computer Simulation* 29, 1, Article 3 (Jan. 2019), 12 pages. doi:10.1145/3230636

Jérmie Lumbroso. 2013. Optimal Discrete Uniform Generation from Coin Flips, and Applications. *arXiv* 1304.1916 (April 2013). doi:10.48550/arXiv.1304.1916

David J. C. MacKay. 2003. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, Cambridge, UK.

Andrea C. G. Mennucci. 2010. Bit Recycling for Scaling Random Number Generators. *arXiv* 1012.4290 (Dec. 2010). doi:10.48550/arXiv.1012.4290

Bernhard Ömer and Christoph Pacher. 2014. Saving Fractional Bits: A Practical Entropy Efficient Code for Fair Die Rolls. *arXiv* 1412.7407 (Dec. 2014). doi:10.48550/arXiv.1412.7407

Sung-Il Pae and Michael C Loui. 2006. Randomizing Functions: Simulation of a Discrete Probability Distribution Using a Source of Unknown Distribution. *IEEE Transactions on Information Theory* 52, 11 (Nov. 2006), 4965–4976. doi:10.1109/TIT.2006.883555

Sung-Il Pae. 2005. *Random Number Generation Using a Biased Source*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign. doi:10.5555/1123531

Sung-Il Pae. 2015. A Generalization of Peres's Algorithm for Generating Random Bits From Loaded Dice. *IEEE Transactions on Information Theory* 61, 2 (Feb. 2015), 751–757. doi:10.1109/TIT.2014.2381223

Sung-Il Pae. 2020. Binarization Trees and Random Number Generation. *IEEE Transactions on Information Theory* 66, 4 (April 2020), 2581–2587. doi:10.1109/TIT.2019.2962480

Yuval Peres. 1992. Iterating von Neumann's Procedure for Extracting Random Bits. *Annals of Statistics* 20, 1 (March 1992), 590–597. doi:10.1214/aos/1176348543

James R. Roche. 1991. Efficient Generation of Random Variables from Biased Coins. In *Proceedings of the IEEE International Symposium on Information Theory*. IEEE Press, Piscataway, NJ, 169–169. doi:10.1109/ISIT.1991.695225

Sinha S. Roy, Frederik Vercauteren, and Ingrid Verbauwhede. 2013. High Precision Discrete Gaussian Sampling on FPGAs. In *Proceedings of the 20th International Conference on Selected Areas in Cryptography (Lecture Notes in Computer Science, Vol. 8282)*. Springer, Berlin, 383–401. doi:10.1007/978-3-662-43414-7_19

Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. 2020a. The Fast Loaded Dice Roller: A Near-optimal Exact Sampler for Discrete Probability Distributions. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 108)*. PMLR, Norfolk, MA, 1036–1046.

Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. 2020b. Optimal Approximate Sampling from Discrete Probability Distributions. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 36 (Jan. 2020), 31 pages. doi:10.1145/3371104

Feras A. Saad and Wonyeol Lee. 2025. Random Variate Generation with Formal Guarantees. *Proceedings of the ACM on Programming Languages* 9, PLDI, Article 152 (June 2025), 25 pages. doi:10.1145/3729251

Claude E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27, 3 (July 1948). doi:10.1002/j.1538-7305.1948.tb01338.x

Jui-Hsiang Shao and Hsin-Po Wang. 2025. MichelangeRoll: Sculpting Rational Distributions Exactly and Efficiently. *arXiv* 2507.00915 (July 2025). doi:10.48550/arXiv.2507.00915

Quentin F. Stout and Bette Warren. 1984. Tree Algorithms for Unbiased Coin Tossing with a Biased Coin. *The Annals of Probability* 12, 1 (Feb. 1984), 212–222. doi:10.1214/aop/1176993384

Luca Trevisan. 2010. Pseudorandomness in Computer Science and in Additive Combinatorics. In *An Irregular Mind: Szemerédi is 70*, Imre Bárány, József Solymosi, and Gábor Sági (Eds.). Bolyai Society Mathematical Studies, Vol. 21. Springer, Berlin, Heidelberg, 619–650. doi:10.1007/978-3-642-14444-8_19

Tomohiko Uyematsu and Yuan Li. 2003. Two Algorithms for Random Number Generation Implemented by Using Arithmetic of Limited Precision. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A, 10 (Oct. 2003), 2542–2551.

Sridhar Vembu and Sergio Verdú. 1995. Generating Random Bits from an Arbitrary Source: Fundamental Limits. *IEEE Transactions on Information Theory* 41, 5 (Sept. 1995), 1322–1332. doi:10.1109/18.412679

John von Neumann. 1951. Various Techniques Used in Connection with Random Digits. In *Monte Carlo Method*, A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.). Number 12 in National Bureau of Standards Applied Mathematics Series. U.S. Government Printing Office, Washington, DC, Chapter 13, 36–38.

Michael D. Vose. 1991. A Linear Algorithm for Generating Random Numbers with a Given Distribution. *IEEE Transactions on Software Engineering* 17, 9 (Sept. 1991), 972–975. doi:10.1109/32.92917

Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Software* 3, 3 (Sept. 1977), 253–256. doi:10.1145/355744.355749

Jacques Willekens. 2004. Probability and Random Numbers. https://archive.is/VOK26.

Tomoyuki Yamakami. 1999. Polynomial Time Samplable Distributions. *Journal of Complexity* 15, 4 (1999), 557–574. doi:10.1006/jcom.1999.0523

A Baseline Online Random Sampling Algorithms. This appendix provides concrete implementations of several baseline online samplers discussed in §1.3, when the target distributions $(p_i)_{i\geq 1}$ are given as arrays of integer weights at each round. The common denominator of the probabilities in p_i are most $d \geq 1$. These implementations admit a precise analysis of their space and time complexities.

Entropy-Optimal Sampling for One Distribution. Algorithm A1 shows an optimized implementation of the algorithm described in §1.3.1. For a single sample from a particular target distribution, explicitly building a DDG tree as in Knuth and Yao (1976) can require exponential space in the worst case, so instead we implicitly traverse the tree level-by-level as proposed in Roy et al. (2013). Additionally, we compute the binary expansions of the weights during traversal, instead of precomputing them, to avoid preprocessing time.

Algorithm A1 Single-sample entropy-optimal sampling

Require: Positive integers $a_0, \ldots, a_{k-1}$ with sum $A = a_0 + \cdots + a_{k-1}$
Ensure: Random sample $X \sim \text{Discrete}(a_0, \ldots, a_{k-1})$

```
1:  procedure KY-DDG(a_0,...,a_{k-1})
2:      z ← 0                                  ▷ initialize uniform state (upper bound is implicit)
3:      b_0 ← a_0; ...; b_{k-1} ← a_{k-1}      ▷ copy weights for computing binary expansions
4:      while true do                          ▷ iterate over levels of tree
5:          for i ← 0 to k − 1 do              ▷ iterate over leaves at current level
6:              if b_i ≥ A then                ▷ leaf i exists at this level
7:                  if z = 0 then              ▷ hit leaf node
8:                      return i               ▷ return leaf label
9:                  else                       ▷ pass leaf node
10:                     z ← z − 1              ▷ reduce space of live nodes
11:                 b_i ← b_i − A              ▷ remove leaf weight
12:             b_i ← 2b_i                     ▷ double weight for next level
13:         z ← 2z + Flip()                    ▷ refine uniform state
```

The loop over levels in Algorithm A1 occurs less than $H(p) + 2 = O(\log(k))$ times in expectation. The **for** loop over k leaves performs $O(\log(A))$ work per leaf, for a total of $O(k \log(A))$. Thus, the overall expected time complexity is bounded as $O(k \log(k) \log(A))$. The space used is $O(k \log(A))$ bits to store the copied weights b_i used to compute the binary expansions. (Alternatively, one could recompute b_i from a_i at each level to save space at the expense of runtime, but we omit this analysis.) Over n samples with each $k \leq K$ and $A \leq d$, the expected time complexity is $O(nK \log(K) \log(d))$ and the space complexity is $O(K \log(d))$ bits. This algorithm does not maintain any auxiliary state between samples.

Online Entropy-Optimal Sampling. Algorithm A2 shows an optimized implementation of the algorithm described in §1.3.3. The online entropy-optimal algorithm of Knuth and Yao (1976) can also be implemented efficiently using a level-by-level approach as in Algorithm A1, but several details require extra care. The optimum refinement algorithms given in Knuth and Yao (1976; p. 384) and Saad and Lee (2025; Algorithm 1) are specialized to the case of binary-coded probability distributions, so we replace the two explicit leaf creation steps (for their p' and p'') by an iteration over all new refined labels. Further, the new leaf order can be fixed to match the input order for compatibility with the method of Algorithm A1, which is equivalent to replacing the set S with a queue in the algorithm of Knuth and Yao (1976). Lastly, the uniform state over the level nodes in the refined subtrees requires a more complex initialization, based on the number of live nodes at the current level as computed in line 8.

Algorithm A2 Online entropy-optimal sampling

Require: Positive integers $a_0, \ldots, a_{k-1}$ with sum $A = a_0 + \cdots + a_{k-1}$
Ensure: Random sample $X \sim \text{Discrete}(a_0, \ldots, a_{k-1})$

1: ▷ *Auxiliary State Variables D, B, C*
2: **mutable int** $D \leftarrow 0$ ▷ *depth in tree*
3: **mutable int** $C \leftarrow 1$ ▷ *numerator weight so far*
4: **mutable int** $B \leftarrow 1$ ▷ *denominator weight so far*
5: **procedure** KY-DG($a_0, \ldots, a_{k-1}$)
6: **update** $B \twoheadleftarrow AB$ ▷ *current total weight*
7: $b_0 \leftarrow 2^D C a_0 \bmod 2B; \ldots; b_{k-1} \leftarrow 2^D C a_{k-1} \bmod 2B$ ▷ *weights for computing binary expansions*
8: $z \leftarrow \lfloor (b_0 + \cdots + b_{k-1})/B \rfloor - 1$ ▷ *reinitialize uniform state (upper bound is implicit)*
9: **while true do** ▷ *iterate over levels of tree*
10: **for** $i \leftarrow 0$ **to** $k - 1$ **do** ▷ *iterate over leaves at current level*
11: **if** $b_i \geq B$ **then** ▷ *leaf i exists at this level*
12: **if** $z = 0$ **then** ▷ *hit leaf node*
13: **update** $C \twoheadleftarrow a_i C$ ▷ *update numerator weight*
14: **return** i ▷ *return leaf label*
15: **else** ▷ *pass leaf node*
16: $z \leftarrow z - 1$ ▷ *reduce space of live nodes*
17: $b_i \leftarrow b_i - B$ ▷ *remove leaf weight*
18: $b_i \leftarrow 2b_i$ ▷ *double weight for next level*
19: $z \leftarrow 2z + \text{FLIP}()$ ▷ *refine uniform state*
20: **update** $D \twoheadleftarrow D + 1$ ▷ *increment depth*

In Algorithm A2, after n samples with each $k \leq K$ and $A \leq d$, the parameters are bounded as $C \leq B \leq d^n$ and $\mathbb{E}[D] < n \log(K) + 2$, so the (expected) space complexity of the auxiliary state is $O(n \log(d))$, although it is unbounded in the worst case, even after just the first sample, because the depth D can be arbitrarily large. The computation of $2^D \bmod 2B$ in line 7 requires time $O(\log(D) \log^2(B))$ when using long multiplication, and the computation of each b_i requires time $O(\log^2(B))$ given the value $2^D \bmod 2B$. Additionally, the computation of z requires time $O(k \log(B) + \log^2(B))$, so the expected combined time complexity before the loop is $O((k + \log(n \log(K)))n^2 \log^2(d))$. The space complexity of the b_i is $O(kn \log(d))$ bits, which is the same as the overall expected space complexity. As for Algorithm A1, the expected time complexity of the loop is $O(kn \log(k) \log(d))$. Thus, the overall expected time complexity is $O(n \log(d)(k \log(k) + (k + \log(n \log(K)))n \log(d)))$. Over n samples with each $k \leq K$ and $A \leq d$, the expected time complexity is $O(n^2 \log(d)(K \log(K) + Kn \log(d) + n \log(n) \log(d)))$ and the expected space complexity is $O(Kn \log(d))$ bits. The expected size of the auxiliary state after n samples is $O(n \log(d))$ bits.

Interval Method (Arithmetic Coding). Algorithm A3 shows an optimized implementation of the algorithm described in §1.3.4. Rather than explicitly maintaining the interval state $[\alpha, \beta) \subseteq [\gamma, \delta) \subseteq [0, 1)$ described in Han and Hoshi (1997), we normalize the state to $[(\alpha - \gamma)/(\delta - \gamma), (\beta - \gamma)/(\delta - \gamma)) \subseteq [0, 1)$. This representation requires only three integers L, R, B to represent this interval $[L/B, R/B)$ by the endpoint numerators and their common denominator. Further, we compute the prefix sums using linear preprocessing, in order to allow binary search for the interval refinement step, which avoids a linear number of big integer operations per sample.

Algorithm A3 Online interval method sampling

Require: Positive integers $a_0, \ldots, a_{k-1}$ with sum $A = a_0 + \cdots + a_{k-1}$
Ensure: Random sample $X \sim \text{Discrete}(a_0, \ldots, a_{k-1})$

▷ *Auxiliary State Variables* L, R, B
mutable int $L \leftarrow 0$ ▷ *left numerator so far*
mutable int $R \leftarrow 1$ ▷ *right numerator so far*
mutable int $B \leftarrow 1$ ▷ *denominator so far*
procedure HH-INTERVAL($a_0, \ldots, a_{k-1}$)
    $A_0 \leftarrow 0; A_1 \leftarrow a_0; \ldots; A_n \leftarrow a_0 + \cdots + a_{k-1}$ ▷ *prefix sums of weights*
    $i \leftarrow 0; j \leftarrow k$ ▷ *initialize comparison interval* $[A_i/A, A_j/A)$
    while $j - i > 1$ **do** ▷ *iteratively refine* $[L/B, R/B) \subseteq [A_i/A, A_j/A)$
        $m \leftarrow \lfloor (i + j)/2 \rfloor$ ▷ *midpoint of comparison interval*
        if $A_m B \leq LA$ **then** ▷ $A_m/A \leq L/B$
            $i \leftarrow m$ ▷ *narrow comparison interval from left*
        else if $A_m B \geq RA$ **then** ▷ $A_m/A \geq R/B$
            $j \leftarrow m$ ▷ *narrow comparison interval from right*
        else ▷ $[L/B, R/B)$ *must be refined*
            $D \leftarrow R - L$ ▷ *compute interval width numerator*
            update $(L, R, B) \twoheadleftarrow (2L, 2R, 2B)$ ▷ *increase precision*
            if FLIP() $= 1$ **then** $L \twoheadleftarrow L + D$ **else** $R \twoheadleftarrow R - D$ ▷ *refine interval state*
    update $(L, R, B) \twoheadleftarrow (LA - A_i B, RA - A_i B, a_i B)$ ▷ *update interval state*
    return i ▷ *return leaf label*

Algorithm A3 requires $O(\log(k))$ interval comparisons in expectation, so the expected time complexity of the loop is $O(\log(k)\log(A)\log(B))$ when using long multiplication. The prefix sums require $O(k\log(A))$ time to compute and $O(k\log(A))$ space to store. Over n samples with each $k \leq K$ and $A \leq d$, the interval state grows as $\mathbb{E}[\log(B)] = O(n\log(d))$, so the expected total space complexity is $O((n + K)\log(d))$, and the expected total time complexity is $O(n\log(d)(n\log(K)\log(d) + K))$. The expected size of the auxiliary state after n samples is $O(n\log(d))$ bits.