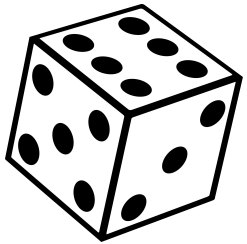


The Fast Loaded Dice Roller

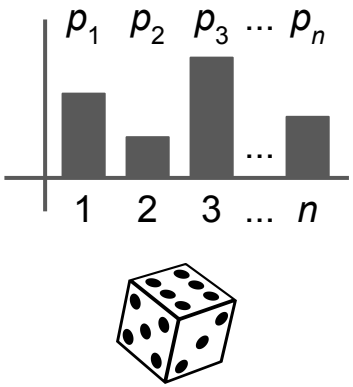
A Near-Optimal Exact Sampler for Discrete Probability Distributions

Feras Saad, Cameron Freer,
Martin Rinard, Vikash Mansinghka

Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics, PMLR 108:1036-1046, 2020. http://proceedings.mlr.press/v108/saad20a.html
Github: https://github.com/probcomp/fast-loaded-dice-roller/

What is a random sampling algorithm?

Suppose **p** := (p_1, ..., p_n) is a list of n probabilities between 0 and 1.



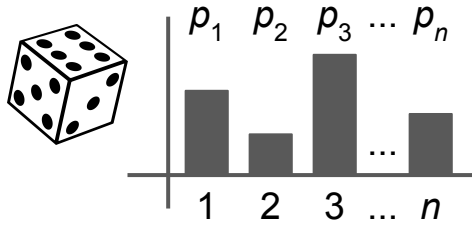
A **sampling algorithm** (sampler) for **p** is a randomized algorithm *A such that*:

Pr[*A* returns integer i] = p_i (i = 1, ..., n)

Sampling is a fundamental operation in many fields

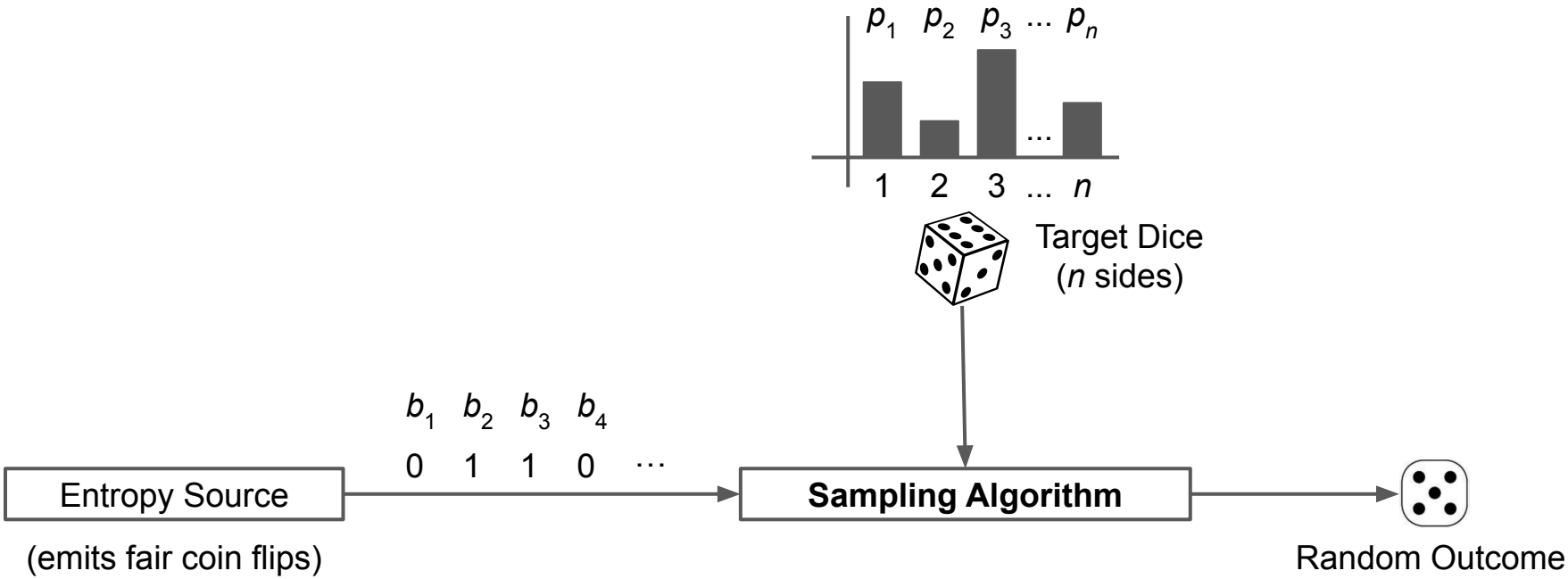
Efficient dice rolling is a classic problem in CS theory

It is also a foundational operation in many fields



Robotics	*Probabilistic Robotics*, Thrun et al. 2005
Artificial Intelligence	*Artificial Intelligence: A Modern Approach*, Russell & Norvig 1994
Computational Statistics	*Random Variate Generation*, Devroye 1986
Operations Research	*Simulation Techniques in Operations Research*, Harling 1958
Statistical Physics	*Monte Carlo Methods in Statistical Physics*, Binder 1986
Financial Engineering	*Monte Carlo Methods in Financial Engineering*, Glasserman 2003
Machine Learning	*An Introduction to MCMC for Machine Learning*, Andrieu et al. 2003
Systems Biology	*Randomization and Monte Carlo Methods in Biology*, Manly 1991
Scientific Computing	*Monte Carlo Strategies in Scientific Computing*, Liu 2001

Key ingredients of a sampling algorithm

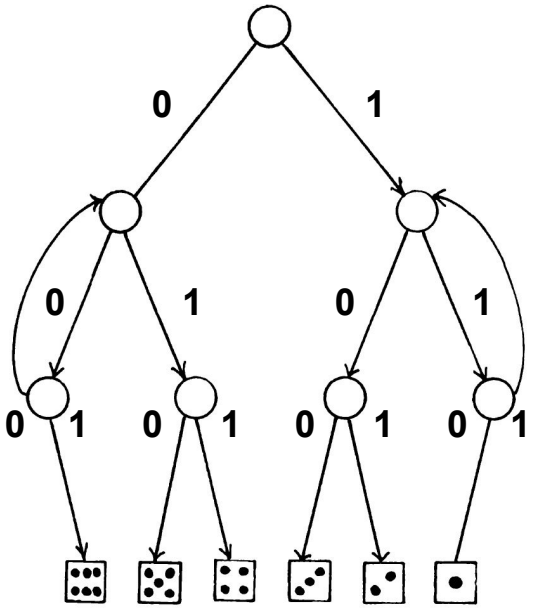


Every sampling algorithm is a tree

A sampler *A* for distribution **p** is a complete binary tree *T*

1. Start at root
2. Call *flip*, if 0 go to left child, if 1 go to right child
3. If child is leaf, return the label of that node, else goto 2.

T is called a **discrete distribution-generating** (**DDG**) tree.



Example DDG tree for a fair dice, **p** = (⅙,⅙,⅙,⅙,⅙,⅙)

111→repeat	011→4
110→1	010→5
101→2	001→6
110→3	000→repeat

Key objectives for designing efficient samplers

We assess the “efficiency” of a sampler for **p** according to two criteria:

1. **Runtime**

Minimize average number of flips
needed to generate a sample
(i.e., average depth of leaf in the tree)

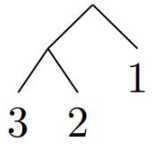
2. **Memory**

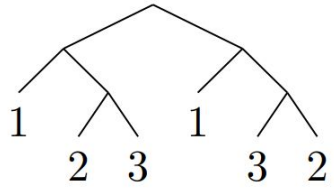
Minimize number of nodes in the tree

Key objectives for designing efficient samplers

We assess the “efficiency” of a sampler for **p** according to two criteria:

Example: 3-sided dice, **p** = (½, ¼, ¼)





1. Runtime

Minimize average number of flips needed to generate a sample (i.e., average depth of leaf in the tree)

1.5 bits / sample

2.25 bits / sample

2. Memory

Minimize number of nodes in the tree

5 nodes

11 nodes

Knuth & Yao solved the runtime problem in 1976

Knuth & Yao (1976) solve the problem for building the fastest sampler for any loaded dice by giving an explicit construction

1. Runtime

Minimize average number of flips needed to generate a sample
(i.e., average depth of leaf in the tree)

THE COMPLEXITY OF NONUNIFORM RANDOM NUMBER GENERATION */

by Donald E. Knuth and Andrew C. Yao +/
Computer Science Department, Stanford University
Stanford, California 94305

The purpose of this paper is to introduce a type of complexity theory which is relevant to the problem of generating random numbers with nonuniform distributions, given a source of uniform random bits. We shall examine procedures which minimize the average number of bits required to generate random numbers from arbitrary numeric distributions in arbitrary systems of notation.

D. Knuth, A. Yao. The complexity of nonuniform random number generation. In Algorithms and Complexity: New Directions and Recent Results. 1976.

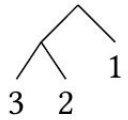
Entropy-optimal sampling (Knuth-Yao 1976)

Theorem Entropy-optimal tree has leaf *i* at level *j* **iff** *j*th bit in binary expansion of p_i = 1.

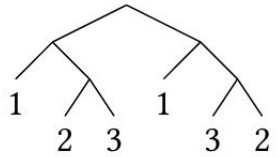
Example 1

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} .1\,0 \\ .0\,1 \\ .0\,1 \end{bmatrix}$$

Binary probability matrix



Entropy-optimal DDG tree



Entropy-suboptimal DDG tree

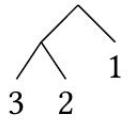
Entropy-optimal sampling (Knuth-Yao 1976)

Theorem Entropy-optimal tree has leaf *i* at level *j* **iff** *j*th bit in binary expansion of $p_i = 1$.

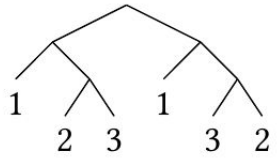
Example 1

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} .1\,0 \\ .0\,1 \\ .0\,1 \end{bmatrix}$$

Binary probability matrix



Entropy-optimal DDG tree

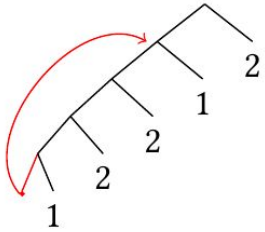


Entropy-suboptimal DDG tree

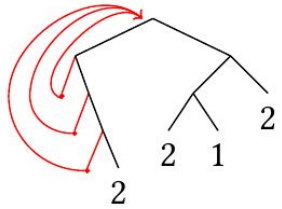
Example 2

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 3/10 \\ 7/10 \end{bmatrix} = \begin{bmatrix} .0\,\overline{1\,0\,0\,1} \\ .1\,0\,1\,1\,0 \end{bmatrix}$$

Binary probability matrix



Entropy-optimal DDG tree

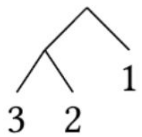


Entropy-suboptimal DDG tree

Main challenge with Knuth-Yao optimal sampler

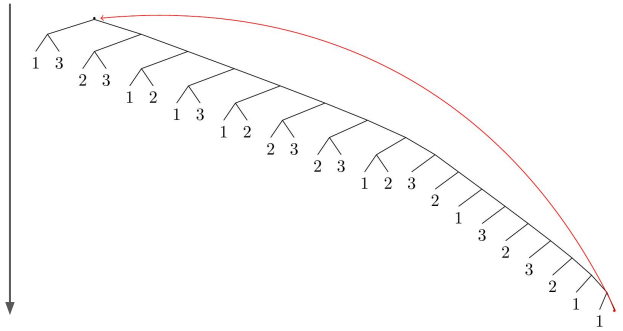
Theorem The Knuth-Yao sampler is **runtime** optimal: H(**p**)≤ E[# flips] < H(**p**) + **2**

However, the **memory** (number of nodes) can scale exponentially



height = 2
= $\log_2(4)$

height = 18
≫ $\log_2(19)$



<u>Good Case</u>

p = (2/4, 1/4, 1/4)

(sum of weights is power of two)

<u>Bad Case</u>

p = (7/19, 4/19, 8/19)

Exact entropy-optimal samplers = exponential size

Example

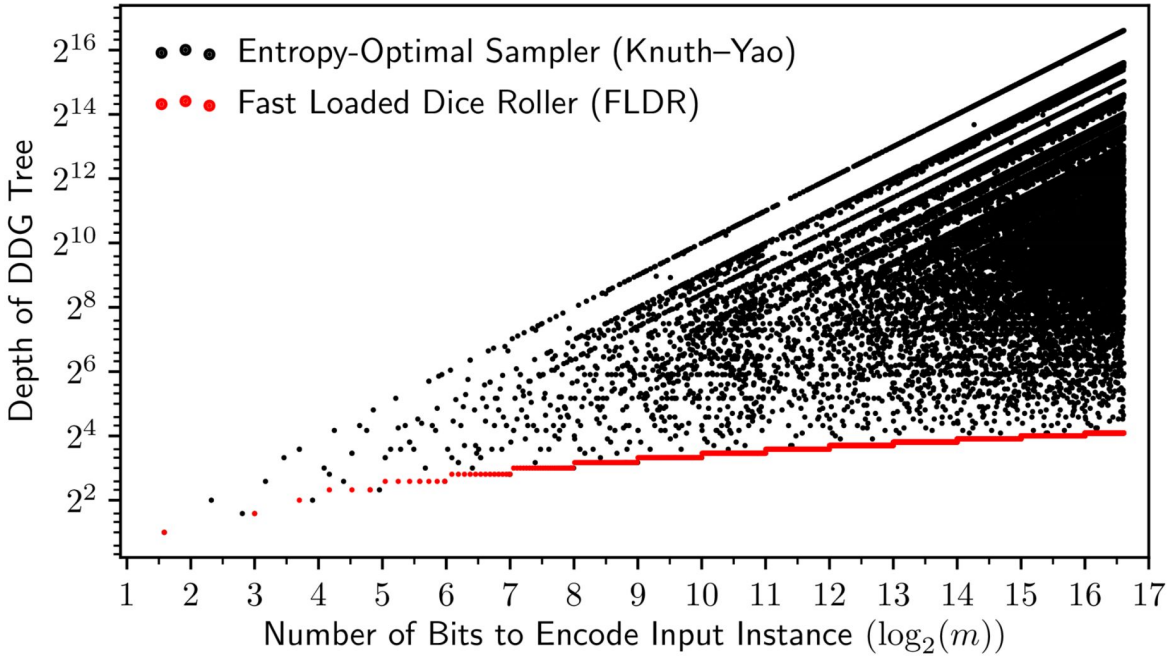
Knuth-Yao tree for Binomial(n=50, p=61/500) has 10^{104} levels (i.e., ~10^{91} terabytes)

> "*Most of the algorithms which achieve these optimum bounds are **very complex**, requiring a **tremendous amount of space***". [KY76]

We will develop an algorithm where the memory scales **linearly** in the size of the dice.

This work: The Fast Loaded Dice Roller

We present an algorithm (FLDR) whose memory scales linearly in the input size



Main intuition of FLDR

The Knuth & Yao sampler is **fast** and **small** when sum of weights m = power of 2.

Main intuition of FLDR

The Knuth & Yao sampler is **fast** and **small** when sum of weights *m* = power of 2.

For *n*-sided dice with sum of weights *m*
we create an (n+1)th side to ensure the new weights m' = 2^k, where $2^{k-1} < m < 2^k$.

original dice
(*n* sides, weight sum *m*)

p ∝ (2, 5, 3)
m = 10

new dice
(*n*+1 sides, weight sum 2^k)

p' ∝ (2, 5, 3, 6)
m = 16

Main intuition of FLDR

The Knuth & Yao sampler is **fast** and **small** when sum of weights m = power of 2.

For n-sided dice with sum of weights m
we create an (n+1)th side to ensure the new weights m' = 2^k, where $2^{k-1} < m < 2^k$.

original dice
(n sides, weight sum m)

new dice
(n+1 sides, weight sum 2^k)

Build Knuth & Yao sampler for the new dice.

Roll the new dice, if  shows up then reject the outcome and roll again.

Main theorems of FLDR

FLDR has near-optimal **runtime**

Exponential decrease in **memory** as compared to optimal KY sampler.

	Average # of bits / sample	**Tree Size**
Knuth & Yao (1976)	$H(\mathbf{p}) \leq E[\#\text{flips}] \leq H(\mathbf{p}) + \mathbf{2}$	$\Theta(n\ \boldsymbol{m})$
FLDR (this work)	$H(\mathbf{p}) \leq E[\#\text{flips}] \leq H(\mathbf{p}) + \mathbf{6}$	$\Theta(n\ \mathbf{\log_2(\boldsymbol{m})})$

n = number of faces in the dice
m = sum of weights of the dice faces

Implementing FLDR using fast integer arithmetic


+  "reject side"

Create Knuth & Yao sampler
for the new dice.

Roll the new dice.
If  shows up, then reject & roll again.

Algorithm 5 Implementation of the Fast Loaded Dice Roller using unsigned integer arithmetic

Input: Positive integers $(a_1, \ldots, a_n)$, $m \coloneqq \sum_{i=1}^{n} a_i$.
Output: Random integer i with probability a_i/m.

```
    // PREPROCESS
 1: k ← ⌈log(m)⌉;
 2: a_{n+1} ← 2^k − m;
 3: initialize h int[k];
 4: initialize H int[n + 1][k];
 5: for j = 0, ..., k − 1 do
 6:     d ← 0;
 7:     for i = 1, ..., n + 1 do
 8:         bool w ← (a_i >> (k − 1) − j)) & 1;
 9:         h[j] ← h[j] + w;
10:         if w then
11:             H[d, j] ← i;
12:             d ← d + 1;
    // SAMPLE
13: d ← 0, c ← 0;
14: while true do
15:     b ∼ FLIP();
16:     d ← 2 · d + (1 − b);
17:     if d < h[c] then
18:         if H[d, c] ≤ n then
19:             return H[d, c];
20:         else {d ← 0; c ← 0;}
21:     else {d ← d − h[c]; c ← c + 1;}
```

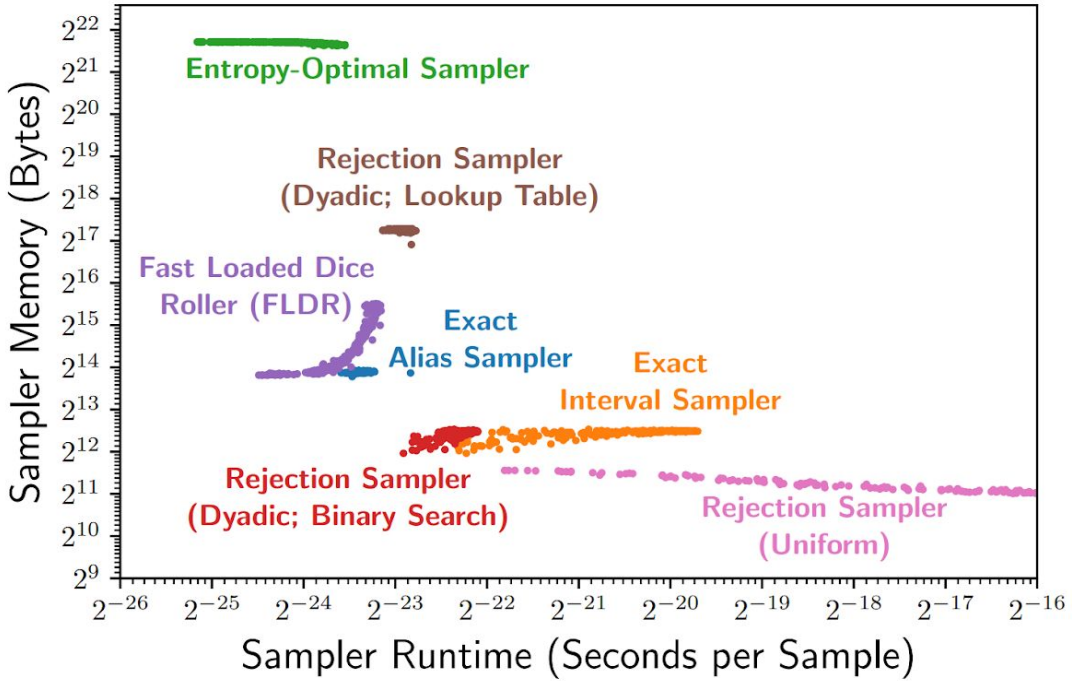
Comparing FLDR to baseline exact samplers

We benchmark FDLR against six baselines for exact sampling:

1. Entropy-optimal sampler [KY76]
2. Rejection sampler with uniform proposal
3. Rejection sampler with dyadic proposal + lookup table [Dev86]
4. Rejection sampler with dyadic proposal + binary search [Dev86]
5. Interval sampler [HH97], using exact implementation of [DG15]
6. Exact alias sampler [Wal77], using one-table implementation [Vos91]

https://github.com/probcomp/fast-loaded-dice-roller-experiments

Comparing memory versus runtime



https://github.com/probcomp/fast-loaded-dice-roller-experiments

Software libraries are starting to incorporate FLDR

Reference implementation (C) https://github.com/probcomp/fast-loaded-dice-roller/tree/master/src/c

Reference implementation (Python) https://github.com/probcomp/fast-loaded-dice-roller/tree/master/src/python

randomgen library (Python) https://github.com/peteroupc/peteroupc.github.io/blob/master/randomgen.py

rust-random library (Rust) https://github.com/vks/rand/blob/fldr/rand_distr/src/weighted_fldr.rs

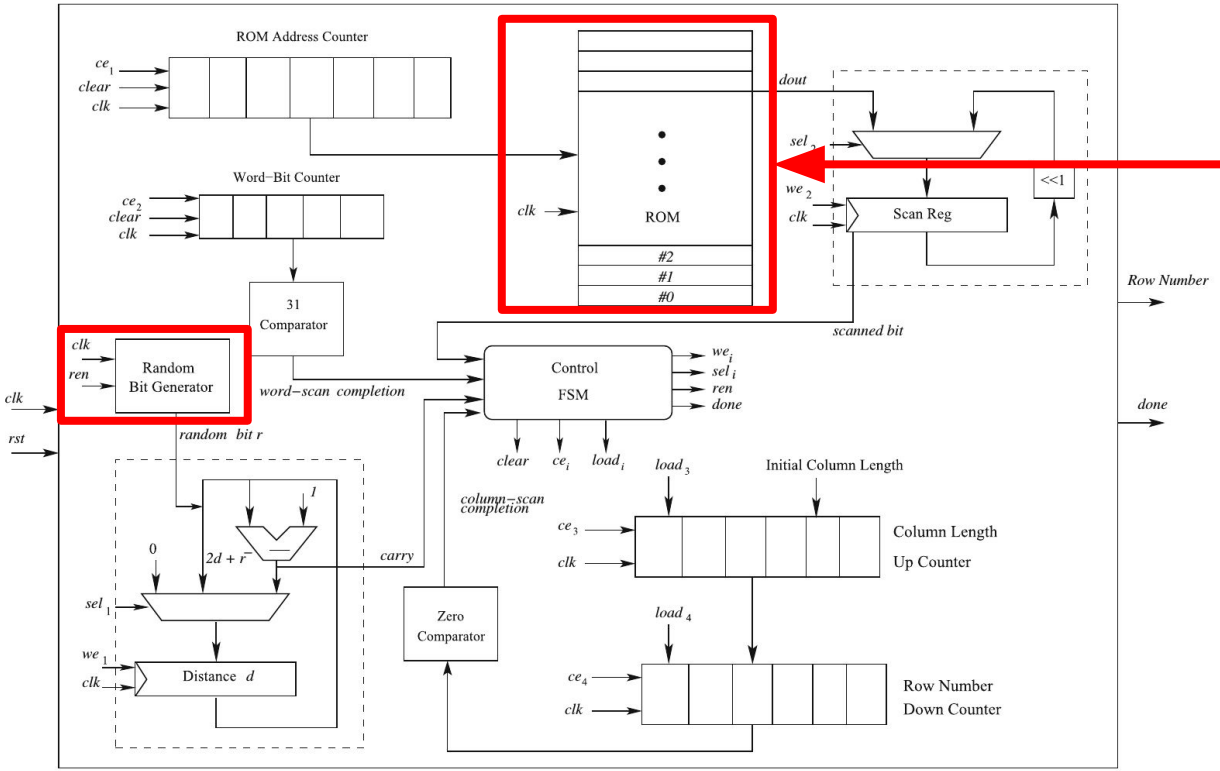
Further directions

1. Speeding up LDA samplers using fast primitives
   - Li, et. al. Reducing the Sampling Complexity of Topic Models. *KDD* 2014
   - Magnusson et. al. Sparse Partially Collapsed MCMC for Parallel Inference in Topic Models. *J. Comp. Graph Stat.* 2018

2. Batch generation by recycling random bits
   - Devroye and Gravel 2019. arXiv:1502.02539 [cs.IT]

3. Probabilistic hardware circuits for fast sampling
   - Mansinghka, Jonas, & Tenenbaum. Stochastic Digital Circuits for Probabilistic Inference, MIT-CSAIL-TR-2008-069, 2008
   - Roy, et al. High Precision Discrete Gaussian Sampling on FPGAs. *SAC* 2013
   - Mansinghka & Jonas. Building Fast Bayesian Computing Machines out of Intentionally Stochastic, Digital Parts. 2014. arXiv:1402.4914 [cs.AI]
   - Du and Bai. Towards Efficient Discrete Gaussian Sampling for Lattice-based Cryptography. *FPL* 2015

Applications to probabilistic hardware



The ($N \times k$) binary probability matrix **P** is encoded into ROM and sampled as follows:

Algorithm	Knuth-Yao Sampling
Input	Probability Matrix **P**
Output	Sample in [0, ..., N-1]

```
d = 0
col = 0
while True:
        r = flip()
        d = 2*d + (1-r)
        for row in [N-1, ..., 0]:
                d -= P[row][col]
                if d == -1:
                        return row
        col = col + 1
```

Roy, et al. High Precision Discrete Gaussian Sampling on FPGAs. SAC 2013.

Related work

1. Optimal sampling with a predefined precision (memory) budget.
   Saad, Freer, Rinard, & Mansinghka [2020], POPL

2. Coalgebraic framework for implementing and composing entropy-preserving reductions between arbitrary input sources to output distributions
   Kozen & Soloviev [2018]; see also Pae & Loui [2006] for asymptotically-optimal variable-length conversions using coins of unknown bias

3. Limited-precision samplers for discrete distributions
   random graph [Blanca & Mihail 2012], geometric [Bringmann & Friedrich 2013], uniform [Lumbroso 2013], discrete Gaussian [Folláth 2014], general [Uyematsu & Li 2003]

4. Variants of random bit model (biased/unknown/non-i.i.d. sources, variable precision)
   [von Neumann 1951; Elias 1972; Stout & Warren 1984; Blum 1986; Roche 1991; Peres 1992; Han & Verdú 1993; Vembu & Verdú 1995; Abrahams 1996; Cicalese et al. 2006; Kozen 2014]

Appendix

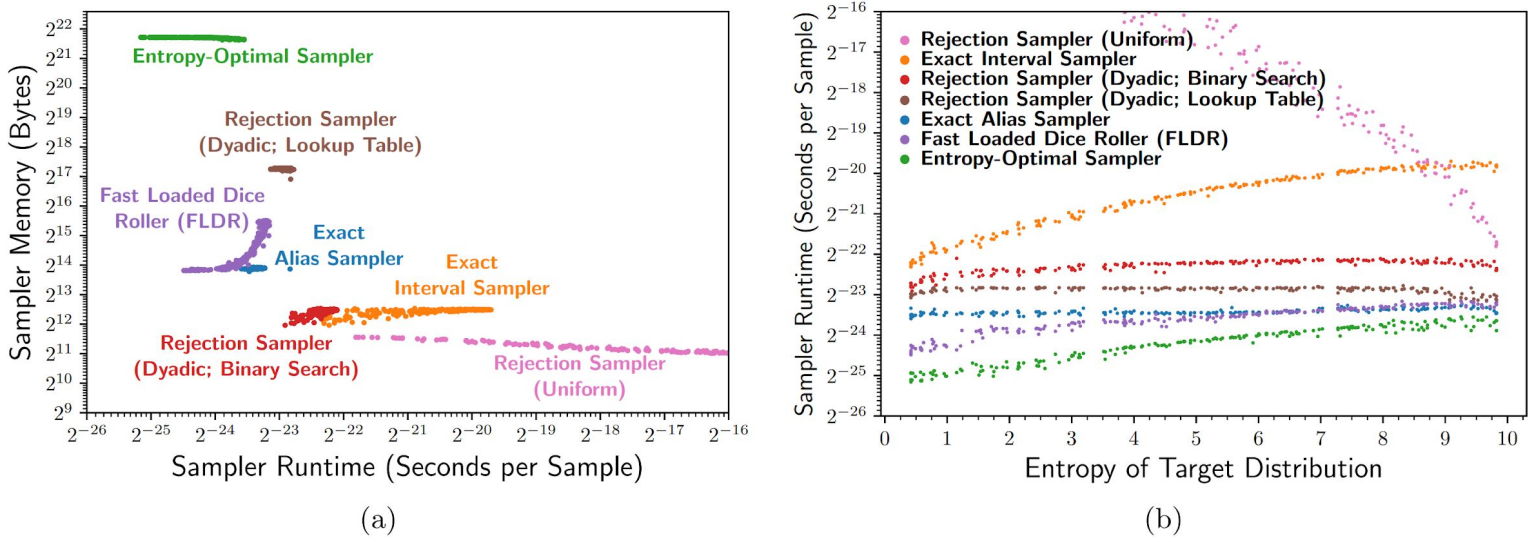
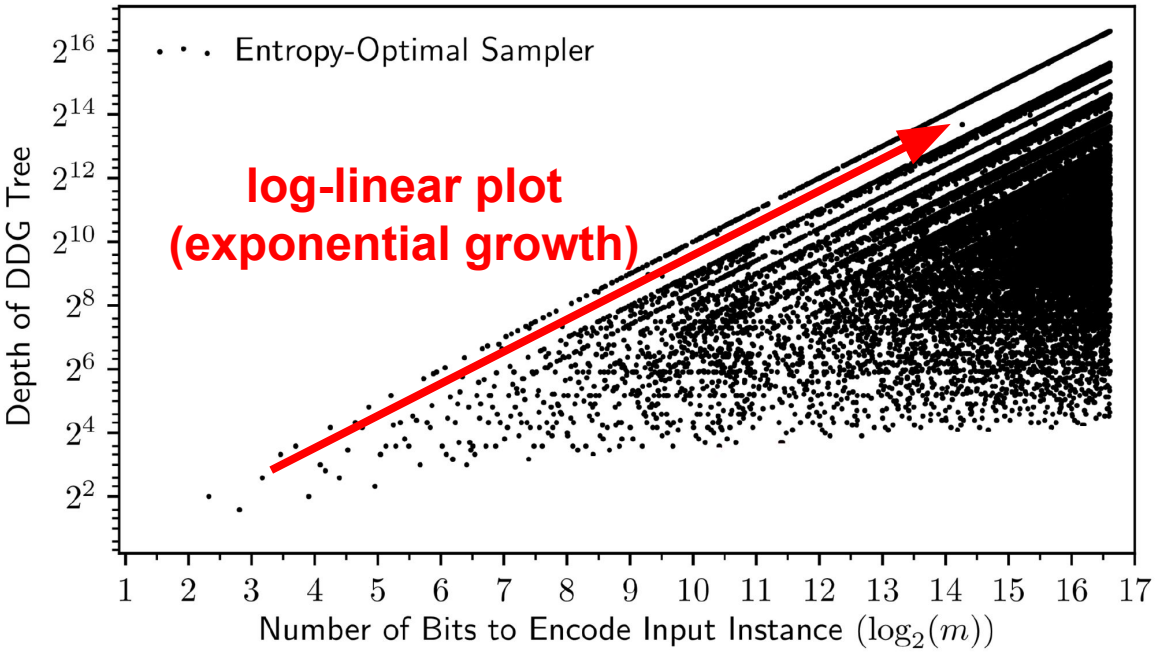


Figure 4: Comparison of memory and runtime performance for sampling 500 random frequency distributions over $n = 1000$ dimensions with sum $m = 40000$, using FLDR and six baseline exact samplers. (a) shows a scatter plot of the sampler runtime (x-axis; seconds per sample) versus sampler memory (y-axis; bytes); and (b) shows how the sampler runtime varies with the entropy of the target distribution, for each method and each of the 500 distributions.

Exact entropy-optimal samplers = exponential size



Example

Entropy-optimal tree for Binomial(n=50, p=61/500) has 10^{104} levels (i.e., ~10^{91} terabytes)

"*Most of the algorithms which achieve these optimum bounds are **very complex**, requiring a **tremendous amount of space***". [KY76]

Comparing preprocessing time vs Alias

To measure preprocessing time of alias we used C GNU Scientific Library (GSL) *gsl_ran_discrete_preproc* function, which implements the alias preprocessing algorithm from [Vos91]

https://github.com/ampl/gsl/blob/master/randist/discrete.c

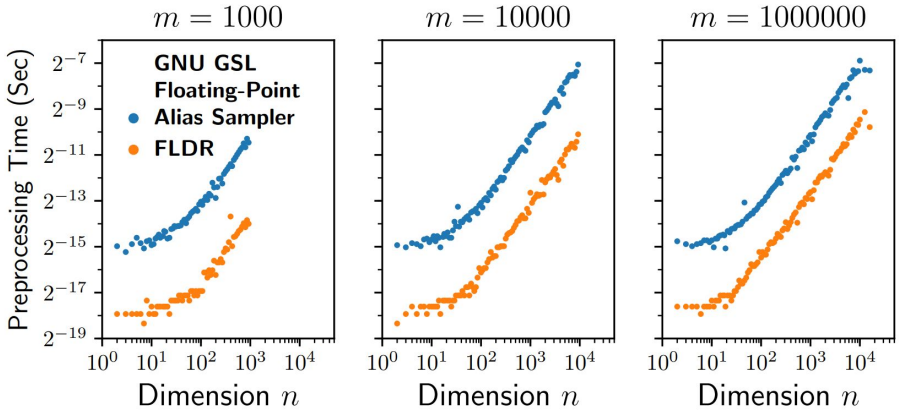


Figure 5: Comparison of the preprocessing times (y-axes; wall-clock seconds) of FLDR with those of the alias sampler, for distributions with dimension ranging from $n = 10^0, \ldots 2 \times 10^4$ (x-axes) and normalizers $m = 1000$, 10000, and 1000000 (left, center, and right panels, respectively).

https://github.com/probcomp/fast-loaded-dice-roller-experiments

Comparing runtime and calls to the PRNG

Table 1: Number of PRNG calls and wall-clock time when drawing 10^6 samples from $n = 1000$ dimensional distributions, using FLDR & approximate floating-point samplers.

Method	Entropy (bits)	Number of PRNG Calls	PRNG Wall Time (ms)
FLDR	1	123,607	3.69
	3	182,839	4.27
	5	258,786	5.66
	7	325,781	7.90
	9	383,138	8.68
Floating Point	all	1,000,000	21.51

https://github.com/probcomp/fast-loaded-dice-roller-experiments