Buffon machines:

the Von Neumann/Flajolet scheme, and a fast Poisson simulator

(work in progress)

Jérémie Lumbroso & Michèle Soria

30/05/2013 — AofA 2013, Minorca (Spain)

0. Introduction

Version **RECURSIVE** [Flajolet *et al.* 1994]

RTree(n) := {
  if $n = 1$ **then** return Leaf
  else
    k from distr. $\mathbb{P}[K = k] = (b_k \cdot b_{n-k})/b_n$
    return Node(RTree(k), RTree($n - k$))
}

Vers. **"BOLTZMANN"** [Duchon *et al.* 02]

ATree(z) := {
  if $\mathrm{Ber}(z/B(z)) = 1$ **then** return Leaf
  else
    return Node(ATree(z), ATree(z))
}

How to simulate the probability distributions in blue?

1. efficiently? simply?
2. using only random bits and counters?

This talk: the Poisson distribution.

building blocks of randomness

1) BERNOULLI LAW: discrete law, noted $\mathrm{Ber}(p)$, $p \in [0, 1]$, and

- coin flip (of bias p)
- random bit
- Bernoulli law of parameter p

designate the same thing, a random process X such that

$$\mathbb{P}[X = 1] = p \qquad \mathbb{P}[X = 0] = 1 - p$$

2) UNIFORM LAW: continuous law, noted $\mathcal{U}(0, 1)$, is random process X which produces a **real** uniformly in the unit interval $[0, 1]$

$$\mathbb{P}[X \in [x, x + \mathrm{d}x]] = \mathrm{d}x$$

as a real	as an (infinite) sequence of random bits
0.7139282598...	0.101101101100010000...
	[= développement dyadique]
	$x = \sum_{k=1}^{\infty} \frac{b_i}{2^k}$

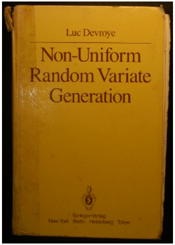
We know:

- go from Bernoulli 1/2 to uniform: dyadic development
- go from uniform to Bernoulli: `if X <= p then 1 else 0`

global problem

most work on random variate generation use the continuous uniform law as a unit of randomness

Bib.: Devroye 86 (seminal reference), etc.



- in theory: uniform variables with infinite precision
- in practice: computers, floating points with fixed 32/64/128 bit precision

Thus waste/inefficiency [too precise] or bias [not precise enough].

AIM: to obtain a set of algorithms to simulate probability distributions (discrete + continuous) using a discrete unit of randomness, the Bernoulli law of 1/2, with constraints

- **exact** simulations
- only a finite number of counters (incr./decr.), of stacks, and strings
- algorithms must be conceptually **simple**, and efficient (= exponential tails of the number of bits)

Bib.: Flajolet, Pelletier & Soria 2009 (Buffon machine).

1. Von Neumann's exponential (1951)

Generate a variate with unit exponential distribution:

(1) $D \leftarrow 0$ (integer part)

(2) generate $Y_0 > Y_1 > Y_2 > \ldots$, until first $n \leqslant 1$ such that $Y_{n-1} \leqslant Y_n$

(3) if

- n even **then** $D \leftarrow D + 1$ and go to (1)
- n odd **then** return $D + Y_0$

Let event G_n : "$Y_0 > Y_1 > Y_2 > \cdots > Y_{n-1} \leqslant Y_n$".

$$\mathbb{P}[G_n \textbf{ and } x < Y_0 < x + \mathrm{d}x] = \left[\frac{x^{n-1}}{(n-1)!} - \frac{x^n}{n!}\right] \mathrm{d}x.$$

By summing over all possible n, get exponential probability

[+ independence of integer part distributed as geometric].

2. Von Neumann/Flajolet scheme

Idea: use the enumeration of permutations to simulate laws

$\mathcal{P}$: some class of perm. with generating function $P(x)$
$\mathcal{P}_n$: subset of perms of size n and P_n : nb perms of size n

function $\Gamma\text{VNF}[\mathcal{P}](x)$
  loop
    $N \leftarrow \mathrm{Geo}(x)$
    draw σ random permutation of size n
    if $\sigma \in \mathcal{P}_N$ **then return** N
  end loop
end function

$$\mathbb{P}_x[N = n] = \frac{(1-x)x^n \cdot P_n/n!}{\sum_k (1-x)x^k \cdot P_k/k!} = \frac{1}{P(x)} \frac{P_n x^n}{n!}$$

- based off of **random permutations** because
  1. very simple to randomly generate
  2. many well-known classes are enumerated
- $1/(x \cdot P(x))$ iterations on average (geometric distribution)

[from "*On Buffon Machines and Numbers*", Flajolet, Pelletier, Soria, SODA 2011]

permutation classes and corresponding distribution

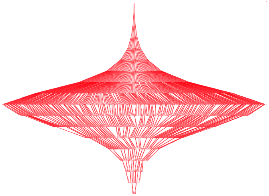
$$\mathbb{P}_x[N = n] = \frac{1}{P(x)} \frac{P_n x^n}{n!}$$

class $\mathcal{P}$	count P_n	EGF	probability	distribution
all	$n!$	$1/(1-x)$	$(1-x)x^n$	geometric
sorted	1	$\exp(x)$	$\mathrm{e}^{-x}x^n/n!$	Poisson
cyclic	$(n-1)!$	$\log(1/(1-x))$	$1/L \cdot x^n/n$	log-series
alternating even	A_{2n}	$\sec(x)$	–	–
alternating odd	A_{2n+1}	$\tan(x)$	–	–

how to generate random permutations?

1. Fisher-Yates random shuffle [$n \log n + o(1)$ bits, L. 2013] $\rightarrow$ no control
2. by drawing n numbered uniform variables (as a sequence of random bits), inserting them in a trie, and looking at order (leaves)

function RandomPermutation(N)
  $\mathbf{U} \leftarrow (U_1, \ldots, U_N), \quad U_i \sim \mathcal{U}(0,1)$
  $\tau \leftarrow \text{trie}(\mathbf{U})$ [insert each U_i in the trie]
  $\sigma \leftarrow \text{order-type}(\tau)$
end function



bits needed: $n \log_2 n + O(n)$ (avg path length of trie)

EXCEPT since we just want to test membership of a random permutation to a class, no need to generate the entire permutation to realize it is wrong

optimizing tests of random permutations

function RandomPermutation(N)
  $\mathbf{U} \leftarrow (U_1, \ldots, U_N), \quad U_i \sim \mathcal{U}(0,1)$
  $\tau \leftarrow$ trie($\mathbf{U}$) [insert each U_i in the trie]
  $\sigma \leftarrow$ order-type(τ)
end function

1. (as said before) since we just want to check "$\sigma \in P_N$?", do **not need** to draw **all bits** of the U_i
2. each **random bit** of the U_i is **indep.** and identically distributed
   $\Rightarrow$ the **order** in which these bits are drawn **does not matter**

exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

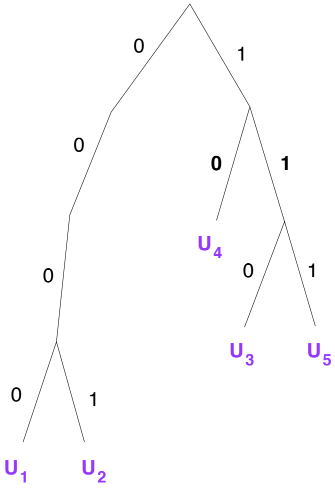
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1
U_2
U_3
U_4
U_5

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

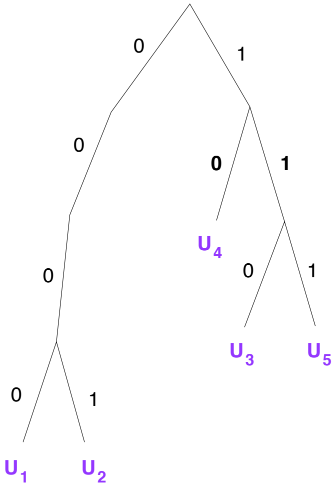
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0
U_2	
U_3	
U_4	
U_5	

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

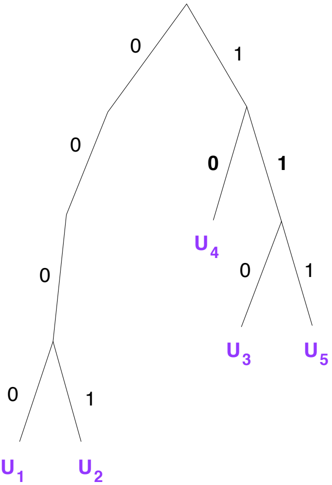
U_1	0	0	0	0	0	0	···
U_2	0	0	0	1	1	0	···
U_3	1	**1**	1	0	1	1	···
U_4	1	**0**	1	0	0	0	···
U_5	1	1	1	0	1	0	···

... with **vertical slices**

U_1	0
U_2	0
U_3	
U_4	
U_5	

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

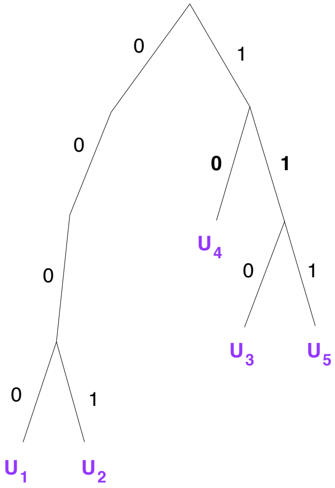
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0
U_2	0	
U_3		
U_4		
U_5		

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

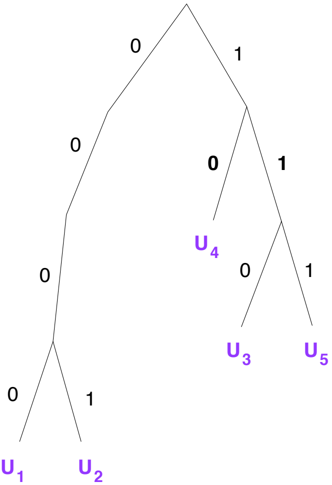
U_1	0	0	0	0	0	0	⋯
U_2	0	0	0	1	1	0	⋯
U_3	1	**1**	1	0	1	1	⋯
U_4	1	**0**	1	0	0	0	⋯
U_5	1	1	1	0	1	0	⋯

... with **vertical slices**

U_1	0	0
U_2	0	0
U_3		
U_4		
U_5		

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

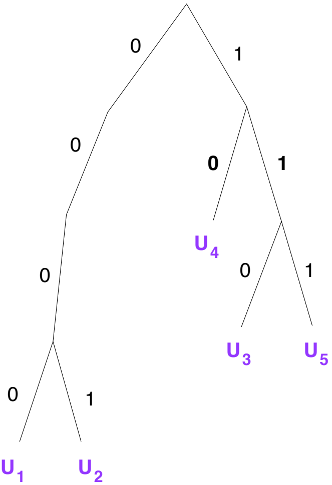
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0
U_2	0	0	
U_3			
U_4			
U_5			

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

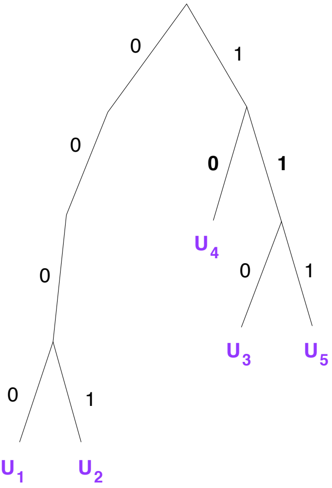
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0
U_2	0	0	0
U_3			
U_4			
U_5			

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

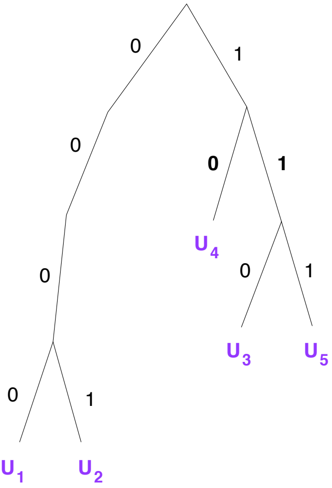
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	
U_3				
U_4				
U_5				

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

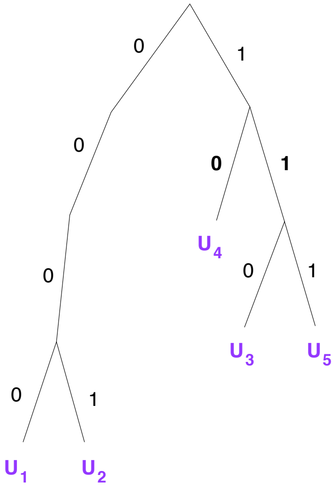
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3				
U_4				
U_5				

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

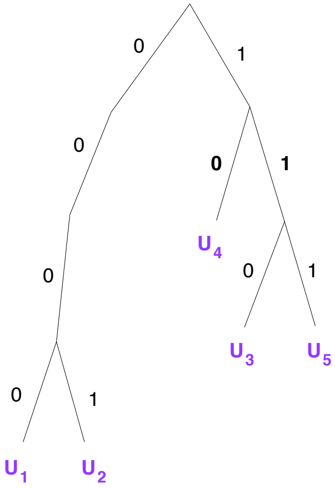
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1			
U_4				
U_5				

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

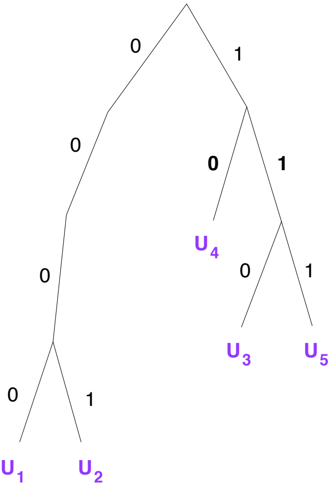
U_1	0	0	0	0	0	0	⋯
U_2	0	0	0	1	1	0	⋯
U_3	1	**1**	1	0	1	1	⋯
U_4	1	**0**	1	0	0	0	⋯
U_5	1	1	1	0	1	0	⋯

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1			
U_4	1			
U_5				

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

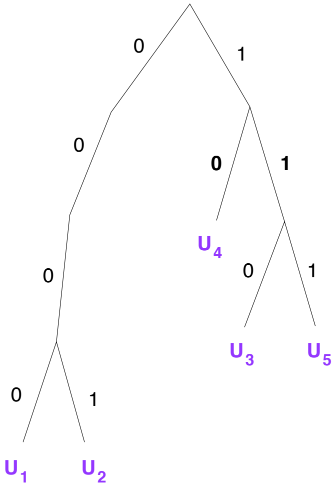
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	1		
U_4	1			
U_5				

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

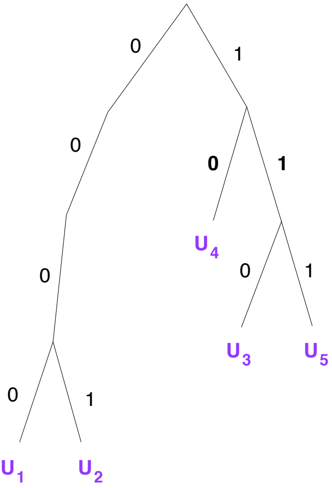
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1
U_2
U_3
U_4
U_5



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

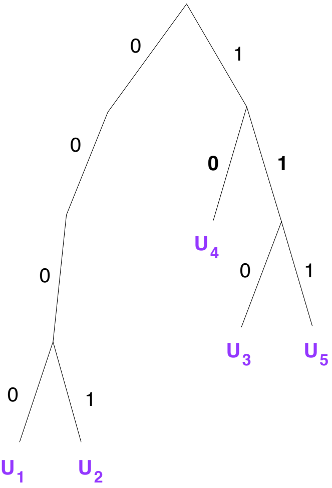
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0
U_2	
U_3	
U_4	
U_5	



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

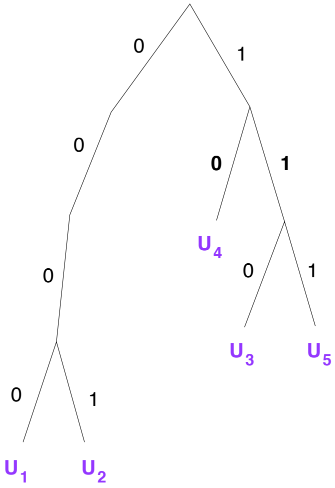
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0
U_2	0
U_3	
U_4	
U_5	



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

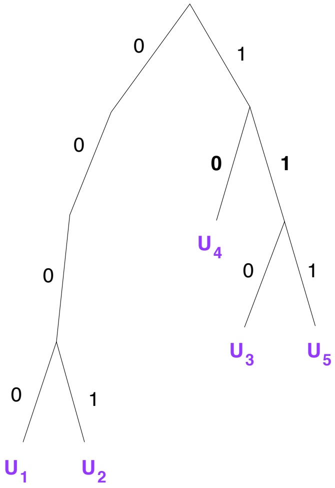
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0
U_2	0
U_3	1
U_4	
U_5	



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

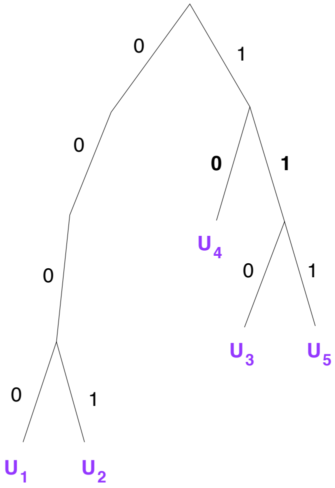
U_1	0	0	0	0	0	0	⋯
U_2	0	0	0	1	1	0	⋯
U_3	1	**1**	1	0	1	1	⋯
U_4	1	**0**	1	0	0	0	⋯
U_5	1	1	1	0	1	0	⋯

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0
U_2	0
U_3	1
U_4	1
U_5	



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

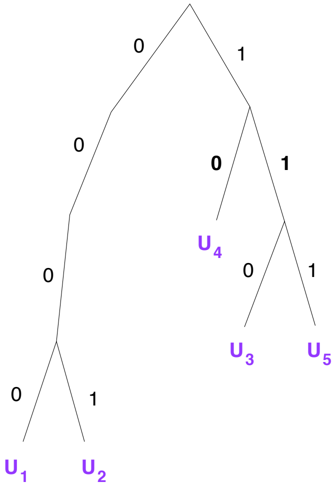
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0
U_2	0
U_3	1
U_4	1
U_5	1



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

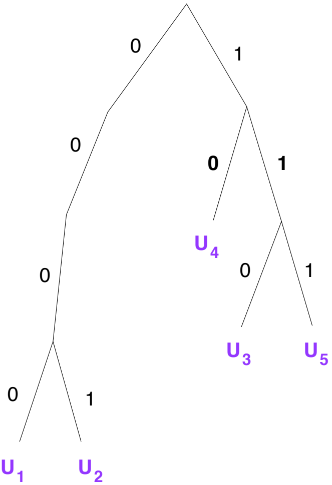
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0	0
U_2	0	
U_3	1	
U_4	1	
U_5	1	



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

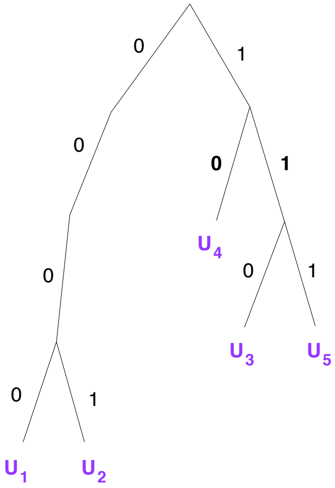
U_1	0	0	0	0	0	0	···
U_2	0	0	0	1	1	0	···
U_3	1	**1**	1	0	1	1	···
U_4	1	**0**	1	0	0	0	···
U_5	1	1	1	0	1	0	···

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0	0
U_2	0	0
U_3	1	
U_4	1	
U_5	1	



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

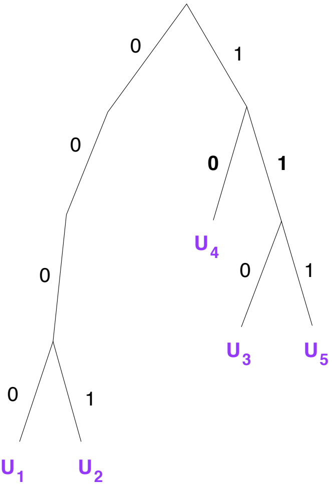
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0	0
U_2	0	0
U_3	1	1
U_4	1	
U_5	1	



exemple: vertical or horizontal slices

suppose the bits were going to come out this way (with $U_4 < U_3$)

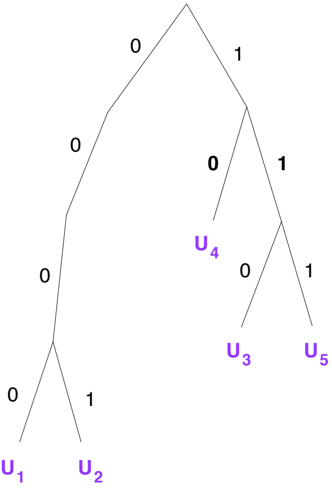
U_1	0	0	0	0	0	0	$\cdots$
U_2	0	0	0	1	1	0	$\cdots$
U_3	1	**1**	1	0	1	1	$\cdots$
U_4	1	**0**	1	0	0	0	$\cdots$
U_5	1	1	1	0	1	0	$\cdots$

... with **vertical slices**

U_1	0	0	0	0
U_2	0	0	0	1
U_3	1	**1**		
U_4	1	**0**		
U_5				

... with **horizontal slices**

U_1	0	0
U_2	0	0
U_3	1	**1**
U_4	1	**0**
U_5	1	



3. Efficient Poisson law from bits

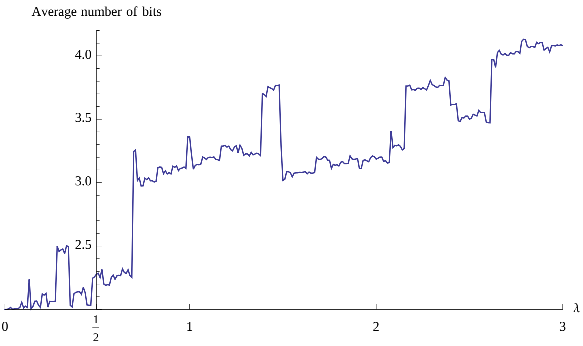
Variable N has Poisson distribution means

$$\mathbb{P}[N = n] = \mathrm{e}^{-\lambda} \frac{\lambda^n}{n!}$$

This can be generated with the VNF scheme with **sorted permutations**.

Optimal average number of bits [Knuth & Yao 83]:

$$\sum_{n=0}^{\infty} \sum_{k=0}^{\infty} \left\{ 2^k \cdot \mathrm{e}^{-\lambda} \frac{\lambda^n}{n!} \right\} \frac{1}{2^k}$$

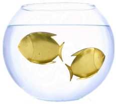


Sorted permutations by horizontal slices

Idea: look horizontal slices and check "seq. of size n with pattern $\mathbf{0}^\star\mathbf{1}^\star$" + recursion

```
VNSorted[n] := if n <= 1 then return true
  else
  {
    k := 0
    while k < n and flip() == 0 { k = k + 1 }
    cut = k ; k = k + 1  /* count the non-0 flip */
    while k < n and flip() == 1 { k = k + 1 }

    if k >= n then
      return VNSorted[k] and VNSorted[n - k]
  }
```



0	0	0	0	0	1	1	1	1	1	1	1

0	0	1	1	1

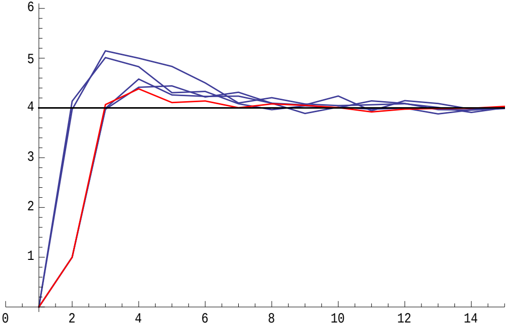
0	0	0	0	0	1	1

... ...

By observing the rejected patterns, establish recurrence of average cost:

$$c_n := t_n + \frac{1}{2^n} \sum_{k=0}^{n} (c_k + c_{n-k}) \qquad\qquad t_n := 4 - (2n+4)/2^n$$

Average cost for algorithm: tends to **4 flips** (the toll)
Intuition: cost of two geometrics of $1/2$ (run of sequence of 0s + of 1s)



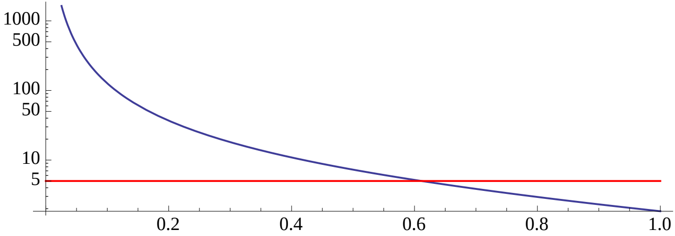
back to the VNF scheme

draw random permutation of size N + check if sorted $\equiv$ draw $\mathrm{Ber}(1/N!)$

function ΓVNF-Poisson(λ)
  loop
    $N \leftarrow \mathrm{Geo}(\lambda)$
    if $\mathrm{Ber}(1/N!) = 1$ **then return** N
  end loop
end function

$\dfrac{1}{\lambda \mathrm{e}^{\lambda}}$ iterations and each iter. (= geometric, $1/\lambda$, + $\mathrm{Ber}(1/N!)$, 4) thus:

$$\text{total avg cost} \approx \frac{\mathrm{e}^{-\lambda}(1+4\lambda)}{\lambda^2}$$



inefficiency when $\lambda < 1/2$

Explanation: when $\lambda < 1/2$, geometric N of λ tends to be large, and it becomes improbable to successfully draw of a sorted permutation of size N, i.e., $\mathrm{Ber}(1/N!)$

Other algorithm (Pelletier/Soria): same as Von Neumann but draw λ-bounded sequences, i.e.,

$$U_0 < U_1 < U_2 < \dots U_{n-1} < \lambda < U_n$$

has dual problem, efficient when $\lambda < 1/2$