Overview

Analysis of algorithms
- Methods and models for the analysis of algorithms.
- Basis for a scientific approach.
- Mathematical methods from classical analysis.
- Combinatorial structures and associated algorithms.

Analytic combinatorics
- Study of properties of large combinatorial structures.
- A foundation for analysis of algorithms, but widely applicable.
- Symbolic method for encapsulating precise description.
- Complex analysis to extract useful information.
**Context for this lecture**

**Purpose.** Prepare for the study of analytic combinatorics *in the context of an important application.*

**Assumed.** Familiarity with analytic combinatorics at the level of *Analysis of Algorithms* Lecture 5.

---

5. Analytic Combinatorics

AofA lecture 5

---

**Analytic combinatorics**

is a calculus for the quantitative study of large combinatorial structures.

**Features:**
- Analysis begins with formal *combinatorial constructions*.
- The *generating function* is the central object of study.
- *Transfer theorems* can immediately provide results from formal descriptions.
- Results extend, in principle, to any desired precision on the standard scale.
- Variations on fundamental constructions are easily handled.

---

the "symbolic method"
Random Sampling of Combinatorial Objects

Robert Sedgewick
Princeton University

with special thanks to Jérémie Lumbroso
Fundamental Study

A calculus for the random generation of labelled combinatorial structures

Philippe Flajolet

Abstract: A calculus for the random generation of labelled combinatorial structures is presented. It models structures through functional equations which are solved using generating functions. The calculus involves key techniques such as generating functions, decomposition techniques, and Mellin transforms. A complete set of examples is given to illustrate various aspects of the calculus.

Boltzmann Samplers

for the Random Generation of Combinatorial Structures

Philippe Flajolet

© 2011 Cambridge University Press

Cover design: Joffrey Dutton

Mathematical Reviews MR2831032

ISBN 978-1-107-00923-4

Hardcover: £65

Paperback: £24.99

Online: £31.99

Bibliographic Information


Random Sampling of Combinatorial

- Basics
- Achieving uniformity
- Rejection
- Recursive method
- Analytic samplers
Introduction

Computer scientists have been fascinated by simple models of natural phenomena since the beginning.

“It might be possible, however, to treat a few particular cases in detail with the aid of a digital computer. This method has the advantage that it is not so necessary to make simplifying assumptions as it is when doing a more theoretical type of analysis.”


Combinatorial classes are often the basis for such models, with *random sampling* critical for validation.

Pioneering work, complete with FORTRAN code

Nijenhuis and Wilf, *Combinatorial Algorithms*, 1975
https://www.math.upenn.edu/~wilf/website/CombinatorialAlgorithms.pdf

Classic reference, still authoritative and worthy of careful study

http://www.nrbook.com/devroye/
Uniformity

Goal for this lecture. **Given a combinatorial class and a size \( N \), return a random object of size \( N \).**

Q. Random object?  
A. Sampling process obeys a **uniform distribution**.

Q. Uniform distribution?  
A. Each object of size \( N \) equally likely to be returned.

Examples (\( N = 3 \))

- **random bitstring**
  - 000
  - 001
  - 010
  - 011
  - 100
  - 101
  - 110
  - 111
  
  Return each with probability 1/8

- **random permutation**
  - 0 1 2
  - 0 2 1
  - 1 0 2
  - 1 2 0
  - 2 0 1
  - 2 1 0
  
  Return each with probability 1/6

- **random mapping**
  - 0 0 0 1 0 0 2 0 0
  - 0 0 1 1 0 2 0 2
  - 0 0 2 1 0 1 2 0
  - 0 1 0 1 1 2 1 1
  - 0 1 1 1 1 2 1 2
  - 0 1 2 1 2 2 2 2
  - 1 0 0 1 0 1 2 0 1
  - 1 1 0 2 0 2 2 0
  - 1 1 1 2 1 1 2 2 0
  - 1 1 2 2 2 2 2
  - 1 2 0 2 0 2 2 2
  - 1 2 1 2 1 2 2 1
  - 1 2 2 2 2 2

  Return each with probability 1/27
Application example I: Program testing and analysis

Problem. Debug a program that processes expressions.

Classic examples

• Regular expression (RE) pattern matching (grep).
• Dijkstra's algorithm for evaluating arithmetic expressions.

Approach. Test implementation on large random expressions.

• Generate a random tree.
• Fill internal nodes with random operators.
• Fill external nodes with values.
• Traverse in inorder.

Result. A realistic benchmark for program testing

\[(6 \times (8 - 2)) + (0/4)\]

For a more complex example in a practical setting, see Canou, Benjamin, and Darrasse, Fast and sound random generation for automated testing and benchmarking in objective Caml, SIGPLAN, 2009.
Application example II: Randomized algorithms

Problem. Improve a program with bad worst-case performance.

Classic example: Quicksort

Approach. Randomize the input.
- Start with a random permutation of the input
- Makes worst case negligible
- Enables mathematical analysis
- Makes performance predictable in practice

Method of choice for a broad variety of applications.
Application example III: Factoring

Problem. **Factor** a large integer \( N \)

Approach ("Pollard's rho method").

- Choose random values \( c \) and \( x < N \)
- Iterate the function \( f(x) = (x^2 + c) \mod N \)
- Stop when a cycle is found
- Analyze by modeling as a **random mapping** (stay tuned)

Factors \( N \) in \( N^{1/4} \) steps

<table>
<thead>
<tr>
<th>Factors of ( N )</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1237 \cdot 4327</td>
<td>21</td>
</tr>
<tr>
<td>123457 \cdot 654323</td>
<td>243</td>
</tr>
<tr>
<td>1234577 \cdot 7654337</td>
<td>1478</td>
</tr>
<tr>
<td>12345701 \cdot 87654337</td>
<td>3939</td>
</tr>
<tr>
<td>123456791 \cdot 987654323</td>
<td>11225</td>
</tr>
<tr>
<td>1234567901 \cdot 10987654367</td>
<td>23932</td>
</tr>
</tbody>
</table>
Problem. Develop a model for RNA secondary structures.

The secondary structure elements of Bacillus subtilis (M13175)

Randomly generated from a simple specification with constraints

Many, many other applications in bioinformatics

Application example V: Experimental mathematics

**Problem.** What is the average *height* of a *binary search tree* with N nodes?

**Approach.**
- Generate a random *permutation*
- Build the BST
- Calculate the height
- Iterate as many times as possible
- Keep track of the average height

**History.**
- Shown to be *about* 4.31 \( \ln N \) by the 1970s
- *Proven* to converge to 4.31107... \( \ln N \) in 1986


Method of choice in studies of discrete structures, ever since computers have been available!
**Random numbers**

**Task.** Return a *random number*.

**Approach.** Use our "StdRandom" library.
- Self-documenting API
- Built on Java's standard Math.random()
- Available at https://introcs.cs.princeton.edu/java/stdlib/javadoc/StdRandom.html

% java StdRandom 3
seed = 1316600616575
31 59.49065 false 9.10423 1
96 51.65818 true 9.02102 0
99 17.55771 true 8.99762 0

For a modern treatment in the context of this lecture, see Flajolet, Pelletier, and Soria, *On Buffon Machines and Numbers*, SODA 2011.

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

– John von Neumann

A poster child for the utility of libraries (CS lecture 3)
Random permutations

**Task.** Return a random permutation of size $N$.

**A solution.** “Knuth-Yates shuffle”.

```java
class RandomPerm {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int[] a = new int[N];
        for (int i = 0; i < N; i++)
            a[i] = i;
        for (int i = 0; i < N; i++) {
            int r = i + StdRandom.uniform(N-i);
            int t = a[i]; a[i] = a[r]; a[r] = t;
        }
        for (int i = 0; i < N; i++)
            StdOut.print(a[i] + " ");
        StdOut.println();
    }
}
```

**Proof of uniformity.** $N!$ different permutations possible, all equally likely.
Three random permutations of size 16

Q. How do we know they're random?

A. They're not random (only appear to be)!

A. Need to test to see if they have the same properties as random ones.
Three random permutations of size 100

Expected number of cycles: $H_{100} \sim 5.2$

Expected number of singleton cycles: 1

Exercise. Generate $10^6$ random perms to validate.

Note. Depends on fast generation!
Random mappings

Task. Return a random mapping of size $N$.

Solution. Trivial.

```java
public class RandomMapping {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int[] a = new int[N];
        for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform(N);
        for (int i = 0; i < N; i++)
            StdOut.print(a[i] + " ");
        StdOut.println();
    }
}
```

Proof of uniformity. $N^N$ different mappings possible, all equally likely.
Three random mappings of size 25

arrows on cycles are ccw and omitted

arrows on trees are towards cycle and omitted
Three random mappings of size 500

Another interesting topic. Approaches to \textit{visualizing} combinatorial structures
A random mapping of size \( N \) has

- \( \sim (\ln N)/2 \) components
- \( \sim \sqrt{\pi N} \) nodes on cycles

The expected number of nodes in the
- longest cycle is about \( 0.78 \sqrt{N} \)
- longest tail is about \( 1.74 \sqrt{N} \)
- longest rho-path is about \( 2.41 \sqrt{N} \)
- largest tree is about \( 0.48 N \)
- largest component is about \( 0.76 N \)

**Exercise.** Generate \( 10^6 \) random mappings to validate (both the analysis and the sampler!)

**Note.** Depends on fast generation
Four random mappings of size 10000
Uniform Sampling of Combinatorial Objects

- Basics
- Achieving uniformity
- Rejection
- Recursive method
- Analytic samplers
Achieving uniformity

**Goal for this lecture.** Given a combinatorial class and a size $N$, return a *random* object of size $N$.

Easily arranged, so far *but not necessarily so easy in many cases*

<table>
<thead>
<tr>
<th>class</th>
<th>typical random object ($N = 10$)</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitstring</td>
<td>1100101101</td>
<td>$1/2^N$</td>
</tr>
<tr>
<td>permutation</td>
<td>9572301486</td>
<td>$1/N!$</td>
</tr>
<tr>
<td>mapping</td>
<td>4938375038</td>
<td>$1/N^N$</td>
</tr>
<tr>
<td>binary tree</td>
<td>![Binary tree diagram]</td>
<td>$\frac{N + 1}{\binom{2N}{N}}$</td>
</tr>
</tbody>
</table>

**Example.** Given $N$, return a *random binary tree* having $N$ nodes.
Achieving uniformity is not to be taken for granted

Case in point. Microsoft antitrust probe by EU
- Accused of favoring the IE browser,
- Microsoft agreed to \textit{randomly permute} browsers.
- But IE was still favored.
- Why? They used a "random method".

"Random numbers should not be generated with a method chosen at random."

– Donald E. Knuth

<table>
<thead>
<tr>
<th>position</th>
<th>IE</th>
<th>Firefox</th>
<th>Opera</th>
<th>Chrome</th>
<th>Safari</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1304</td>
<td>2099</td>
<td>2132</td>
<td>2595</td>
<td>1870</td>
</tr>
<tr>
<td>2</td>
<td>1325</td>
<td>2161</td>
<td>2036</td>
<td>2565</td>
<td>1913</td>
</tr>
<tr>
<td>3</td>
<td>1105</td>
<td>2244</td>
<td>1374</td>
<td>3679</td>
<td>1598</td>
</tr>
<tr>
<td>4</td>
<td>1232</td>
<td>2248</td>
<td>1916</td>
<td>590</td>
<td>4014</td>
</tr>
<tr>
<td>5</td>
<td>5034</td>
<td>1248</td>
<td>2542</td>
<td>571</td>
<td>605</td>
</tr>
</tbody>
</table>

https://www.robweir.com/blog/2010/02

Favored position makes no sense
**Binary trees**

**Task.** Given $N$, return a random binary tree having $N$ nodes.

---

**AofA lecture 3**

**Catalan numbers**

How many binary trees with $N$ nodes?

\[
T_N = \sum_{0 \leq k < N} T_k T_{N-1-k} + \delta_{N0}
\]

---

**Unlabelled class example 3: binary trees**

**Def.** A binary tree is empty or a sequence of a node and two binary trees.

**Unlabelled class example 3: binary trees**

\[
T_N = \frac{1}{N+1} \binom{2N}{N} \frac{1}{2^N (1 + \sqrt{1-4z})}
\]

Catalan numbers (see Lecture 3)

\[
T(z) = 1 + zT(z)^2
\]

---

**AofA lecture 5**
Rémy’s algorithm

A classic and clever algorithm for generating a random binary tree with $N$ nodes.

Given a binary tree with $N-1$ internal nodes

- Choose a node $x$ (internal or external) at random.
- Choose an orientation (L or R) at random.
- Replace $x$ with a new internal node having $x$ as one child (as per orientation) and a new external node as the other.

Result: A binary tree with $N$ internal nodes.

Rémy’s algorithm.
Start with a single external node and iterate $N$ times.
Rémy’s algorithm (examples)
Rémy’s algorithm (uniformity)

**Theorem.** Rémy’s algorithm produces each binary tree of a given size with equal likelihood.

**Proof.**

- Consider all possibilities for adding an internal node to all trees with \( N-1 \) internal nodes.
- Each tree with \( N \) internal nodes appears \( N+1 \) times, *once for each external node* (see example).
- If \( T_N \) is the number of trees produced with \( N \) internal nodes (all equally likely) then

\[
(N + 1) \times T_N = 2 \times (2N - 1) \times T_{N-1}
\]

- Therefore

\[
T_N = \frac{(2N)(2N-1)}{(N+1)N} \times T_{N-1}
\]

\[
= \frac{(2N)!}{(N+1)!N!}
\]

\[
= \frac{1}{N+1} \binom{2N}{N}
\]

- Which implies that each binary tree of size \( N \) is equally likely.
Rémy’s algorithm: implementation

Straightforward implementation can be complicated (try it!)

Complications

- Need explicit external nodes.
- Need array of node pointers to choose random node.
- Need "parent" links, which are notoriously complicated to maintain.
- Each iteration creates two nodes and changes three links (not counting parent links).

\[ \text{bottom line: you can find some ugly code in the literature} \]
Knuth’s implementation of Rémy’s algorithm (representation)

Use an array `links[]` of indices
- Root is `links[0]`
- Even indices represent external nodes
- Odd indices represent internal nodes
- For odd \( k \), children of internal node \( k \) are `links[k]` and `links[k+1]`

Code to build a linked tree from `links[]` representation

```java
Node[] nodes = new Node[2*N + 1];
for (int k = 0; k < 2*N + 1; k+=2)
    nodes[k] = new Node(0);
for (int k = 1; k < 2*N + 1; k+=2)
    nodes[k] = new Node(1);
root = nodes[links[0]];
for (int k = 1; k < 2*N; k+=2)
{
    nodes[k].left = nodes[links[k]];
    nodes[k].right = nodes[links[k+1]];
}
```
Knuth’s implementation of Rémy’s algorithm

```java
int[] links = new int[2*N + 1];
for (int k = 1; k < 2*N; k+=2) {
    int x = StdRandom.uniform(k);
    if (StdRandom.bernoulli(.5)) {
        links[k] = k+1; links[k+1] = links[x];
    } else {
        links[k] = links[x]; links[k+1] = k+1;
    }
    links[x] = k;
}
```

"Then the program is short and sweet"
Knuth’s implementation of Rémy’s algorithm (example)
Remy’s algorithm

Generate a random binary tree with $N$ nodes.

```java
private void generate(int N) {
    int[] links = new int[2*N + 1];
    for (int k = 1; k < 2*N; k+=2) {
        int x = StdRandom.uniform(k);
        if (StdRandom.bernoulli(.5)) {
            links[k] = k+1; links[k+1] = links[x];
        } else {
            links[k] = links[x]; links[k+1] = k+1;
        }
        links[x] = k;
    }
    Node[] nodes = new Node[2*N + 1];
    for (int k = 0; k < 2*N + 1; k+=2) nodes[k] = new Node(0);
    for (int k = 1; k < 2*N + 1; k+=2) nodes[k] = new Node(1);
    root = nodes[links[0]];
    for (int k = 1; k < 2*N; k+=2) {
        nodes[k].left = nodes[links[k]];
        nodes[k].right = nodes[links[k+1]];
    }
}
```

Short and sweet, but . . . no extension to other types of trees is known.

$N = 10,000$
Five random binary trees with 10,000 nodes

Challenge. Develop uniform samplers for other types of trees and other combinatorial classes.
Challenge for this lecture

**Problem.** Our samplers so far are *specialized* and do not extend to more complicated situations.

**Examples.**
- bitstrings with forbidden patterns
- generalized derangements and involutions
- trees of all sorts
- restricted mappings
- ...

**Fundamental challenge.** Develop methods that
- apply to a broad variety of classes *and*
- are provably uniform *and*
- admit efficient implementations

**Ultimate goal.** Generate a sampler for a combinatorial class *automatically* from its specification.
Random Generation of Combinatorial Objects

- Basics
- Achieving uniformity
- Rejection
- Recursive method
- Analytic samplers
First technique to consider: rejection

- Generate a random object
- Reject it if it does not have a specified property
- Continue until finding one that does have the property

Ex: random 20-bit string with no 00

Ex: random mapping of 500 nodes with at least 4 cycles
Random bit strings without long runs

**Task.** Generate a random bitstring of length $N$ with no occurrence of $P$ consecutive 0s.

**Approach.**
- Generate a random $N$-bit string.
- Reject and try again if it has $P$ consecutive 0s.

```java
private void generate(int N, int P) {
    String s;
    boolean rejected = true;
    while (rejected) {
        s = "";
        for (int i = 0; i < N; i++)
            if (StdRandom.bernoulli(.5))
                s += "1" else s += "0";
        int run = 0;
        for (int i = 0; ((i < N) && run != P); i++)
            if (s.charAt(i) == '1') run = 0; else run++;
        if (run < P) rejected = false;
    }
}
```

% java RandomBitsReject 50 4
00110001001110110010110001001111000100111100010111

89 trials (?)

% java RandomBitsReject 50 3
1111111101111011111111010010010111110110111011

50490 trials (!!!)

% java RandomBitsReject 50 2
01101110110111101011011011010111111011111101111101
Primary problem with the rejection method

May have to reject a very large number of attempts before finding a desired object.

Analysis clearly exposes the problem

• Probability that an \( N \)-bit string has no run of 4 0s is about \( 1.0917 \times .96328^N \approx .000000346 \) for \( N = 400 \)
Anticipated rejection

Generally not necessary to generate the whole object.

Many other ways to cope have been studied.

Full details omitted in this lecture so that we can cover more powerful ideas (next two sections).
Random Generation of Combinatorial Objects

- Basics
- Achieving uniformity
- Rejection
- Recursive method
- Analytic samplers
Second technique to consider: the "recursive method"

- Start with a recursive definition of a class
- Compute probabilities of sizes of subobjects
- Use recursive program to create sample

**Specification**

**Precomputed probabilities**

**Recursive sampler**

**Ex: AofA lecture 6 (details revisited soon)**
Example 1: random bitstrings with no 00

For $N > 1$, an $N$-bit string with no 00 is either

- empty or 0 or
- An $(N-1)$-bit string with no 00, followed by 1 or
- An $(N-2)$-bit string with no 00, followed by 10

$B_{00}$, the class of all bitstrings with no 00

$$B_{00} = E + Z_0 + B_{00} Z_1 + B_{00} Z_1 Z_0$$

$$
\begin{align*}
\text{private String } & B00(\text{int } N) \\
\{ & \quad \text{if (N == 0) return "";} \\
\quad & \quad \text{if (N == 1) return "0";} \\
\quad & \quad \text{if (StdRandom.bernoulli(1.0/phi))} \\
\quad & \qquad \text{return}(B00(N-1) + "1"); \\
\quad & \quad \text{else} \\
\quad & \qquad \text{return}(B00(N-2) + "10"); \\
\} 
\end{align*}
$$

For $N > 1$, # $N$-bit strings with

- no 00 is $\sim \phi^N$
- no 00, ending in 1 is $\sim \phi^{N-1}$
- no 00, ending in 10 is $\sim \phi^{N-2}$

∴ Probability of ending in 1 is $1/\phi$
Example 2: Recursive method for random binary trees

For $N > 0$, a binary tree is a node and two binary trees

$$T = E + Z \times T \times T$$

Recursive sampler

Precomputed probabilities

```
private Node T(int N)
{
    Node x = new Node();
    x.N = N;
    if (N > 0)
    {
        int k = StdRandom.discrete(cat[N]);
        x.left = T(k);
        x.right = T(N-k-1);
    }
    return x;
}
```

Probability subtree sizes are $k$ and $N-k-1$

$$\frac{1}{k} \binom{2k-2}{k} \frac{1}{N-k+1} \binom{2N-2k}{N-k}$$

Precomputed probabilities

Another poster child for the symbolic method (AofA lecture 5)
Basis for the recursive method

**Precomputed probabilities.** Need probability that subtree size is \( k \) in a binary tree with \( N \) nodes

"Dynamic programming" solution for binary trees (AofA lecture 6):

\[
\frac{1}{k} \binom{2k - 2}{k} \cdot \frac{1}{N - k + 1} \binom{2N - 2k}{N - k} \cdot \frac{1}{N + 1} \binom{2N}{N}
\]

**Important note.** Extends to trees of all types *and to any constructible combinatorial class*

**Caveat.** Requires excessive time and space, in general (quadratic, in this case).

```java
public static double[][] catalan(int N) {
    double[] T = new double[N];
    double[][] cat = new double[N-1][];
    T[0] = 1;
    for (int i = 1; i < N; i++)
        T[i] = T[i-1]*(4*i-2)/(i+1);
    cat[0] = new double[1];
    for (int i = 1; i < N-1; i++)
        { 
            cat[i] = new double[i];
            for (int j = 0; j < i; j++)
                cat[i][j] = T[j]*T[i-j-1]/T[i];
        }
    return cat;
}
```
“If you can specify it, you can generate a random one.”


Contributions.

• Systematizes earlier ideas by Wilf and Nijenhuis.

• Based on “folk theorem” equivalent to modern combinatorial constructions.

• **Theorem.** Any decomposable structure has a random generation routine that uses precomputed tables of size $O(n)$ and achieves $O(n \log n)$ worst-case time complexity.

• Basis for full implementation, now in Maple.

---


Abstract

A systematic approach to the random generation of labelled combinatorial objects is presented. It applies to structures that are decomposable, i.e., formally specified by grammars involving set, sequence, and cycle constructions. A general strategy is developed for solving the random generation problem with two closely related types of algorithms. For structures of size $n$, the branch-and-bound algorithm exhibits a worst-case behavior of the form $O(n \log n)$, whereas sequential algorithms have worst case $O(n^2)$, while offering good potential for optimization in the average case. The complexity model is based on arithmetic constructions and has been applied to precomputed random table of linear size that can be computed in time $O(n)$.

A computer algebra system has been developed to compute the average case of the sequential generation algorithm associated to a given specification. Using optimizations dictated by this cost calculus, several random generation algorithms of the sequential type are developed, most of them.
Industrial-strength random sampling


- Automatically compiles random generation methods from specifications
- In widespread use for decades, most recently "combstruct" in Maple


Ponty, Termier and Denise, *GenRGens: Software for generating random genomic sequences and structures, Bioinformatics, 2006.*

- Dedicated to randomly generating genomic sequences and structures

https://www.lri.fr/genrgens

Lumbroso, *to appear.*

- Free publicly available modern implementation

Bottom line. Recursive method can *automatically* handle *any* constructible combinatorial class.

Full details omitted in this lecture so that we can cover an even more powerful idea (next section).
Recursive method leads to *automatic* uniform sampler for any constructible class,

**BUT** preprocessing can require excessive time and space in general (does not scale).

<table>
<thead>
<tr>
<th>scalable</th>
<th>extensible</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive method</td>
<td>not always</td>
</tr>
<tr>
<td>Remy's algorithm</td>
<td>✓</td>
</tr>
<tr>
<td>next challenge</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Next.** Scalable *and* extensible uniform samplers.

**Remy's algorithm**

**Next challenge**

- binary
- ternary
- Motzkin

...?
Random Generation of Combinatorial Objects

- Basics
- Achieving uniformity
- Rejection
- Recursive method
- Analytic samplers
Power series distributions

Starting point.
• A combinatorial class $A$ with OGF $A(z)$ having radius of convergence $x_0$
• a positive number $x < x_0$

Definition. A power series distribution at $x$ for $A$ assigns to each object $a$ the probability $\frac{x^{|a|}}{A(x)}$.


Properties of power series distributions
• Distribution is spread over all objects in the class.
• All objects of each size have the same probability.
• Expected size $N_x$ of an object drawn uniformly from such a distribution is easily calculated.

$A(z) = \sum_{a \in A} z^{|a|}$

$\sum_{a \in A} \frac{x^{|a|}}{A(x)} = \frac{A(x)}{A(x)} = 1 = \sum_{n \geq 0} A_n \frac{x^n}{A(x)}$

$E(N_x) = \sum_{a \in A} |a| \frac{x^{|a|}}{A(x)} = \sum_{n \geq 0} n A_n \frac{x^n}{A(x)}$

$= x \frac{A'(x)}{A(x)}$
Analytic samplers

Starting point.
- A constructable combinatorial class \( A \)
- Use symbolic method to find OGF \( A(z) \)
- Find radius of convergence \( x_0 \)

**Definition.** An *analytic sampler* is a program that returns objects drawn from a power series distribution for \( A \)

returns each object \( a \) with probability \( x^{|a|}/A(x) \) for some \( x < x_0 \)

**Idea.** Derive the sampler directly from the specification and the OGF.

**Easy cases:**

| Class     | sampler | proof that each object \( a \) is sampled with probability \( x^{|a|}/A(x) \) |
|-----------|---------|---------------------------------------------------|
| Neutral   | return \( \epsilon \) | 1 object of size 0, OGF is 1 |
| Atomic    | return \( \bullet \) | 1 object of size 1, OGF is \( z \) |
Dijoint union and Cartesian product construction for analytic samplers

**Disjoint union**

Analytic sampler for \( A = B + C \)

if (StdRandom.bernoulli(B(x)/A(x)) return B
else return C

Proof that each object \( a \) is sampled with probability \( x^{|a|}/A(x) \)

\[
Pr\{a \in B\} = \sum_{b \in B} \frac{x^{|b|}}{A(x)} = \frac{B(x)}{A(x)}
\]

**Cartesian product**

Analytic sampler for \( A = B \times C \)

return compose(B, C)

Proof that each object \( a \) is sampled with probability \( x^{|a|}/A(x) \)

\[
\frac{x^{|a|}}{A(x)} = \frac{x^{|b|+|c|}}{A(x)} = \frac{x^{|b|+|c|}}{B(x)C(x)} = \frac{x^{|b|}}{B(x)} \frac{x^{|c|}}{C(x)}
\]
Analytic samplers for unlabeled classes (summary)

Use combinatorial constructions to build a *sampler* that produces random objects.

| construction   | sampler                                      | proof that each object $a$ is sampled with probability $x^{|a|}/A(x)$                        |
|----------------|----------------------------------------------|---------------------------------------------------------------------------------------------|
| neutral class  | $E$                                          | return $\varepsilon$                                                                         | 1 object of size 0, OGF is 1                                                                   |
| atomic class   | $Z$                                          | return $\bullet$                                                                             | 1 object of size 1, OGF is $z$                                                                |
| disjoint union | $A = B + C$                                   | $u = \text{StdRandom.bernoulli}(B(x)/A(x))$ if $(u)$ return $B$ else return $C$            | $Pr\{a \in B\} = \sum_{b \in B} \frac{x^{|b|}}{A(x)} = \frac{B(x)}{A(x)}$                   |
| Cartesian product | $A = B \times C$                           | return $\text{compose}(B, C)$                                                               | $\frac{x^{|a|}}{A(x)} = \frac{x^{|b|} + |c|}{A(x)} = \frac{x^{|b|} + |c|}{B(x)C(x)} = \frac{x^{|b|}}{B(x)} \frac{x^{|c|}}{C(x)}$ |

**notation**

- $A$: combinatorial class
- $a$: object in $A$
- $|a|$: size of $a$
- $A(z)$: OGF for $A$
- $x_0$: radius of convergence of $A(z)$
- $x$: positive real $< x_0$
Example 1: Analytic sampler for random bitstrings without long runs

**Specification**

**Analytic sampler**

**Symbolic transfer**

**GF equation**

\[ B_4(z) = (1 + z + z^2 + z^3)(1 + zB_4(z)) \]

\[ = \frac{1 + z + z^2 + z^3}{1 - z - z^2 - z^3 - z^4} \]

\[ B_4, \text{ the class of all bitstrings with no } 0^4 \]

\[ B_4 = Z_{<4}(E + Z_1 B_4) \]

\[ B_4(z) = \left(1 + z + z^2 + z^3\right)\left(1 + zB_4(z)\right) \]

\[ = \frac{1 + z + z^2 + z^3}{1 - z - z^2 - z^3 - z^4} \]

private static double B4(double r)
{
    return 1.0/(1.0 - r - r*r - r*r*r - r*r*r*r);
}

private String generate(double r)
{
    if (StdRandom.bernoulli(1.0/B4(r)) return zeros();
    return zeros() + "1" + generate(r);
}

**Returns**

- Returns "" ""0" ""00" ""000" with equal probability

- **concatenation**

- **see Lecture 1**
Critical question about an analytic sampler

Q. What is the size of the object that it generates?

A. It is a random variable that depends on the value of $r$.

A. Whatever length string is returned, each string of that length is equally likely.

Next step. Choosing a value of $r$ to achieve a given expected length.

private static double B4(double r)  
{  return 1.0/(1.0 - r - r*r - r*r*r - r*r*r*r); }

private String generate(double r)  
{  if (StdRandom.bernoulli(1.0/B4(r)) return zeros();  
    return zeros() + "1" + generate(r);  
}

\[ B_4, \text{ the class of all bitstrings with no 0}^4 \]

\[ B_4 = Z_{\lt 4} (E + Z_1 B_4) \]

\[ B_4(z) = (1 + z + z^2 + z^3)(1 + zB_4(z)) = \frac{1 + z + z^2 + z^3}{1 - z - z^2 - z^3 - z^4} \]
Next step in building an analytic sampler

Q. What is the expected size of the generated sample?

A. It is drawn uniformly from a power-series distribution. Recall this calculation for the expected size:

\[ E(N_r) = \sum_{a \in A} |a| \frac{r^{|a|}}{A(r)} = \sum_{n \geq 0} n A_n \frac{r^n}{A(r)} = r A'(r) A(r) \]

Therefore, to generate a sample of expected size \( N \), choose the value of \( r \) that satisfies \( N = r \frac{A'(r)}{A(r)} \)

Ex. random string with no 0

\[ B(r) = \frac{1 + r + r^2 + r^3}{1 - r - r^2 - r^3 - r^4} \]

\[ \sim \frac{C}{1 - \beta r} \text{ with } \beta = 1.9276 \]

\[ B'(r) \sim \frac{C \beta}{(1 - \beta r)^2} \]

\[ \frac{B'(r)}{B(r)} \sim \frac{\beta r}{1 - \beta r} \]

Ex. random string with no 0

\[ N \sim \frac{\beta}{1 - \beta r} \]

\[ r \sim \frac{N}{(N + 1) \beta} \]

the action is very close to the singularity
Practical consideration: variance

To generate a bitstring with no 0^4 of expected length N

```java
double beta = 1.9276;
double r = (1.0*N)/(beta*(1.0 + N));
StdOut.println(generate(r));
```

Important note. **Variance is not small**

**Bad news.** Many of the strings are very short

**Good news.** Not such a problem because they are so small

**Bad news.** Some of the strings are very long

**Good news.** Not such a problem because there are few of them, and we can use rejection to limit the cost

**Bottom line.** Total cost is linear.

Ex: 10000 trials with N = 1000 produced
- 1273 strings with fewer than 200 bits
- 1389 strings with between 800 and 1200 bits
- 2533 strings with more than 2000 bits
“If you can specify it, you can generate a **HUGE** random one.”


**Contributions.**

- **Scalable** and **automatic** generation.
- Use rejection to wait for an object of a desired size.
- Use anticipated rejection to avoid excessively large objects.
- Full analysis with complex-analytic methods of analytic combinatorics.
- Full characterization of three types of size distributions.

**Theorem.** Any decomposable structure has an efficient sampler that produces objects close to a desired size with each object produced equally likely among all objects of the same size.

Note. In this lecture, we use the term "**Analytic Sampler**" as equivalent to "Boltzmann Sampler".

assumes an oracle exists that can evaluate generating functions efficiently
To build an analytic sampler

- Derive Java code from construction
- Compute value of $r$ that gives target size
- Use global variables to avoid recomputation
- Use anticipated rejection to avoid large sizes

```java
public class RandomStringNo4 {
    static int N;
    static double r;
    static double p;

    private static String zeros() { /* omitted */}

    private static String generate() {
        if (StdRandom.bernoulli(p)) return zeros();
        String s = generate();
        if (s.length() > 1.1*N) return s;
        return zeros() + "1" + s;
    }

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        r = 1/1.9276 - 1.0/N;
        p = 1.0/B(r);
        String s = "";
        while ((s.length() < 0.9*N) || (s.length() > 1.1*N))
            s = generate();
        StdOut.println(s);
    }
}
```

Ex. random string with no 0

```
% java RandomStringNo4 50
001000100010001000100011001000100010001000101110

% java RandomStringNo4 50
0001001000101000100101010100010001010001101010110001110

% java RandomStringNo4 50
0101010101001010100010001100010010001000100010001
```
Analytic sampler for random binary trees

Specification

Symbolic transfer

GF equation

Analytic sampler

\[ T(z) = \frac{1 - \sqrt{1 - 4z}}{2} \]

private double T(double r)
{
    return (1.0 - Math.sqrt(1.0 - 4.0*r))/2.0;
}

private Node generate(double r)
{
    if (StdRandom.bernoulli(r/T(r)))
        return new Node(0);
    Node x = new Node(1);
    x.left = generate(r);
    x.right = generate(r);
    return x;
}

\[ T = E + Z \times T \times T \]

E
return □

Z
return ●

A = B + C
if \( u < B(x)/A(x) \) return B
else return C

A = B \times C
return compose(B, C)
**Next step for binary trees**

To generate a sample of expected size $N$

**choose the value of $r$** that satisfies

$$N = r \frac{A'(r)}{A(r)}$$

### Expected size of a random binary tree

$$T(r) = \frac{1 - \sqrt{1 - 4r}}{2}$$

$$T'(r) = \frac{1}{\sqrt{1 - 4r}}$$

$$\frac{rT'(r)}{T(r)} = \frac{2r}{(1 - \sqrt{1 - 4r})\sqrt{1 - 4r}}$$

$$= \frac{1}{2} + \frac{1}{2\sqrt{1 - 4r}}$$

### Value of $r$ to expect a tree of size $N$

$$N = \frac{1}{2} + \frac{1}{2\sqrt{1 - 4r}}$$

$$\sqrt{1 - 4r} = \frac{1}{2N - 1}$$

$$r = \frac{1}{4} \left(1 - \frac{1}{(2N - 1)^2}\right)$$

### Note: value of $r/T(r)$ (all we need)

$$\frac{r}{T(r)} = \frac{N}{T'(r)} = N\sqrt{1 - 4r} = \frac{N}{2N - 1} = \frac{1}{2 - \frac{1}{N}}$$

all the action is very close to the singularity
Analytic sampler for random binary trees

Java code to generate a tree with $N$ nodes, on average

```java
private Node generate(int N) {
    double u = Math.random();
    if (u < 1.0/(2.0 - 1.0/N))
        return new Node(0);
    Node x = new Node(1);
    x.left = generate(r);
    x.right = generate(r);
    return x;
}
```

Important notes.
- Need to use rejection to wait for tree of specified size.
- Need to use anticipated rejection to avoid huge trees.
- Then, total cost is linear.

Ex: 10000 trials with $N = 1000$ produced
- 9627 trees with fewer than 200 nodes
- 17 trees with between 1000 and 1200 nodes
- 13 trees with more than 100,000 nodes
- one tree with 973,562 nodes (!!)
Singular analytic sampler for trees with anticipated rejection

Idea. Just use the singular value.

\[ r = \frac{1}{4} \left( 1 - \frac{1}{(2N - 1)^2} \right) \]

may as well just use 1/4 which gives \( r/T(r) = 1/2 \)

Example. Sampler for a binary tree with \( N \) nodes.

```java
private Node generate()
{
    double u = Math.random();
    if (u < 1.0/2.0)
        return new Node(0);
    if (CNT++ > 1.05*N)
        return new Node(0);
    Node x = new Node(1);
    x.left = generate(r);
    x.right = generate(r);
    return x;
}

while ( CNT < 0.95*N || CNT > 1.05*N )
{ CNT = 0; t = generate(N); }
```

Important point. Easily extends to other types of trees and other classes.
Four random trees with about 500 nodes

$p_0 = p_2 = 1/2$

binary

$p_0 = 2/3, \ p_3 = 1/3$

ternary

$p_0 = p_1 = p_2 = 1.0/3.0$

$0–1–2$ (Motzkin)

$p_0 = 5.0/9.0, \ p_2 = 1.0/3.0, \ p_3 = 1.0/9.0$

$0–2–3$
Aside: iterative (breadth-first) singular analytic samplers for trees

Idea. Implement a "Galton-Watson process".
- Use parent-link representation
- $i$th entry on queue is parent of $i$
- Generate $k$ children for each node w.p. $p_k$
  (need analytic combinatorics, in general)
- Use upper bounds and tolerance to terminate
- Example: $p_0 = p_2 = .5$ gives binary trees

```java
private static Queue<Integer> generate(double[] p) {
    Queue<Integer> tree = new Queue<Integer>();
    int root = 0;
    tree.enqueue(0);
    while (root < tree.size()) {
        int k = StdRandom.discrete(p);
        for (int j = 1; j <= k; j++)
            tree.enqueue(root);
        root++;
    }
    return tree;
}
```

Confession: trees on previous slide generated with this code!
Analytic samplers for labeled classes

Use combinatorial constructions to build a sampler that produces random objects (proofs omitted).

<table>
<thead>
<tr>
<th>construction</th>
<th>sampler</th>
<th>notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>neutral class</td>
<td>$E$</td>
<td>return $\epsilon$</td>
</tr>
<tr>
<td>atomic class</td>
<td>$Z$</td>
<td>return $\bullet$</td>
</tr>
<tr>
<td>disjoint union</td>
<td>$A = B + C$</td>
<td>$u = \text{StdRandom.bernoulli}(B(x)/A(x))$ \if(u) return $B$ else return $C$</td>
</tr>
<tr>
<td>labeled product</td>
<td>$A = B \star C$</td>
<td>return $\text{compose}(B, C)$</td>
</tr>
<tr>
<td>sequence</td>
<td>$A = \text{SEQ}(B)$</td>
<td>$k = \text{geometric}(B(x))$ \return $\text{compose}(B, B, \ldots, B)$</td>
</tr>
<tr>
<td>set</td>
<td>$A = \text{SET}(B)$</td>
<td>$k = \text{poisson}(B(x))$ \return $\text{compose}(B, B, \ldots, B)$</td>
</tr>
<tr>
<td>cycle</td>
<td>$A = \text{CYC}(B)$</td>
<td>$k = \text{logseries}(B(x))$ \return $\text{compose}(B, B, \ldots, B)$</td>
</tr>
</tbody>
</table>

$k$ independent instances

Note: Apply actual labels to the sampled structure (if needed) using a random permutation.
Distributions for labeled classes

**Geometric.** $p_k = (1 - \lambda)^k$

```java
double[] p = new double[MAX];
p[0] = 1.0-lambda;
for (int k = 1; k < MAX; k++)
    p[k] = lambda*p[k-1];
```

**Poisson.** $p_k = e^{-\lambda \frac{\lambda^k}{k!}}$

```java
double[] p = new double[MAX];
p[0] = Math.exp(-lambda);
for (int k = 1; k < MAX; k++)
    p[k] = lambda*p[k-1]/(1.0*k);
```

**Log-series.** $p_k = \left( \ln \frac{1}{1 - \lambda} \right)^{-1} \frac{\lambda^k}{k}$

```java
double[] p = new double[MAX];
p[1] = 1.0/Math.log(1.0/(1.0 - lambda));
for (int k = 1; k < MAX; k++)
    p[k] = lambda*p[k-1]/(1.0*k);
```
Analytic sampler for sets of cycles (permutations)

\[ P = \text{SET (CYC (Z))} \]

\[ P(z) = \exp(\ln \frac{1}{1 - z}) = \frac{1}{1 - z} \]

```
private static Queue generate(double r)
{
    int lambda = Math.log(1.0/(1.0 - r));
    int k = StdRandom.poisson(lambda);
    Queue<Integer> q = new Queue<Integer>();
    for (int i = 0; i < k; i++)
        q.enqueue(logseries(r));
    return q;
}
```
Next step for sets of cycles (permutations)

To generate a sample of expected size $N$

**choose the value of $r$** that satisfies $N = r \frac{A'(r)}{A(r)}$

**Expected size of a permutation**

\[ P(r) = \frac{1}{1 - r} \quad P'(r) = \frac{1}{(1 - r)^2} \]

\[ r \frac{P'(r)}{P(r)} = \frac{r}{1 - r} \]

**Value of $r$ to expect a permutation of size $N$**

\[ N = \frac{r}{1 - r} \]

\[ r = \frac{N}{N + 1} \]
Four random sets of cycles (permutations) with about 200 nodes
Analytic sampler for sets of cycles (permutations) with size restrictions

```java
private static Queue generate(double r) {
    int lambda = Math.log(1.0/(1.0 - r));
    int k = StdRandom.poisson(lambda);
    Queue<Integer> q = new Queue<Integer>();
    for (int i = 0; i < k; i++)
        if (k is in omega)
            q.enqueue(logseries(r));
    return q;
}
```

Note: Difficult to compute optimal value for r
(works to use same value as for unrestricted case)
Four random sets of cycles with size restrictions

About 200 nodes, cycle lengths between 20 and 50

About 100 nodes, cycle lengths between 5 and 25

About 50 nodes, cycle lengths between 5 and 10

About 200 nodes, cycle lengths between 20 and 25
Mappings with 1000 nodes of indegree 1 or 2 and no cycle lengths less than 10

Breadth-first approach works for mappings
(start with set of cycles on the queue)
Recursive method vs. analytic sampling

Recursive method
• Gives an object of the specified size.
• Excessive preprocessing time and space (that depends on the size of the object).

Analytic sampling
• Gives an object of *about* the specified size.
• Minimal preprocessing time and space (that depends on the size of the specification).

Three key ideas
• Both *immediately* extend to handle variations and restrictions.
• Both can *automatically* be built from specifications (in principle).
• **Analytic samplers are scalable** (with slight relaxation of size constraint).

```java
private Node generate(int N) {
    if (N == 0) return new Node(0);
    int k = StdRandom.discrete(cat[N]);
    Node x = new Node(N);
    x.left = generate(k);
    x.right = generate(N-k-1);
    return x;
}
```

```java
private Node generate(int N) {
    double u = StdRandom.uniform();
    if (u < 1.0/(2.0 - 1.0/N))
        return new Node(0);
    Node x = new Node(1);
    x.left = generate(N);
    x.right = generate(N);
    return x;
}
```
With analytic samplers, we can study *anything* that can be modeled as a constructible combinatorial class.
Summary

Analytic samplers based on power series distributions are effective, extensible, and scalable.

Rigorous analysis (omitted here) proves lack of bias and scalability in many, many situations.

Ability to generate huge random instances opens new areas of scientific inquiry.

A scientific approach to discrete models

• Formulate the model (develop a specification)
• Collect instances from the real world.
• Develop scalable sampler and generate random instances.
• Test model by validating that they are similar to real ones.

Fully automating the process remains an ongoing research goal.

Also on the horizon: non-uniform samplers based on multivariate analytic combinatorics.
Random Sampling of Combinatorial Objects

Robert Sedgewick
Princeton University

with special thanks to Jérémie Lumbroso