Analysis of Algorithms

Original MOOC title: ANALYTIC COMBINATORICS, PART ONE

http://aofa.cs.princeton.edu
http://ac.cs.princeton.edu
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
Dedicated to the memory of Philippe Flajolet

Philippe Flajolet 1948–2011

Fundamental Study

A calculus for the random generation of labelled combinatorial structures

Philippe Flajolet

Paul Zimmermann

Bernard Hanen

Commemoration by L. Devroye

Bernard Hanen

Philippe Flajolet 1948–2011

Boltzmann Samplers

for the Random Generation

of Combinatorial Structures

Philippe Flajolet

Guy Lyons

Philippe Flajolet

EPFL

1 Laboratoire de Bioinformatique, EPFL, Lausanne, Switzerland.
2 EPFL, Lausanne, Switzerland.
3 INRIA Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay cedex, France.
4 EPFL, Lausanne, Switzerland.

Boltzmann samplers are introduced as a framework for the random generation of labelled combinatorial structures such as trees and permutations, permutations, labelled graphs, and others. This framework is used to express the generating function algorithm for a class of combinatorial structures, and to derive an algorithm for the random generation of such structures. The algorithm is based on the classical Boltzmann formalism, and it is shown that it produces uniformly distributed random samples.

1 Introduction

In the study, Boltzmann samplers are introduced as a framework for the random generation of labelled combinatorial structures such as trees and permutations, permutations, labelled graphs, and others. This framework is used to express the generating function algorithm for a class of combinatorial structures, and to derive an algorithm for the random generation of such structures. The algorithm is based on the classical Boltzmann formalism, and it is shown that it produces uniformly distributed random samples.
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.

“\textit{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.}”

– Alan Turing

\textbf{Combinatorial classes} are often the basis for such models, with \textit{random sampling} critical for validation.

Pioneering work, complete with FORTRAN code

Nijenhuis and Wilf, \textit{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**

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

return each with probability 1/8

**random permutation**

<table>
<thead>
<tr>
<th>012</th>
<th>021</th>
<th>102</th>
<th>120</th>
<th>201</th>
<th>210</th>
</tr>
</thead>
<tbody>
<tr>
<td>012</td>
<td>021</td>
<td>102</td>
<td>120</td>
<td>201</td>
<td>210</td>
</tr>
</tbody>
</table>

return each with probability 1/6

**random mapping**

<table>
<thead>
<tr>
<th>000</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>100</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>001</th>
<th>101</th>
<th>201</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>101</td>
<td>201</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>002</th>
<th>102</th>
<th>202</th>
</tr>
</thead>
<tbody>
<tr>
<td>002</td>
<td>102</td>
<td>202</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>010</th>
<th>110</th>
<th>210</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>110</td>
<td>210</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>011</th>
<th>111</th>
<th>211</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>111</td>
<td>211</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>012</th>
<th>112</th>
<th>212</th>
</tr>
</thead>
<tbody>
<tr>
<td>012</td>
<td>112</td>
<td>212</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>020</th>
<th>120</th>
<th>220</th>
</tr>
</thead>
<tbody>
<tr>
<td>020</td>
<td>120</td>
<td>220</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>021</th>
<th>121</th>
<th>221</th>
</tr>
</thead>
<tbody>
<tr>
<td>021</td>
<td>121</td>
<td>221</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>022</th>
<th>122</th>
<th>222</th>
</tr>
</thead>
<tbody>
<tr>
<td>022</td>
<td>122</td>
<td>222</td>
</tr>
</tbody>
</table>

return each with probability 1/27

random binary tree

return each with probability 1/5
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

\[
\begin{align*}
\text{(6 * (8 - 2))} + \left(\frac{0}{4}\right)
\end{align*}
\]

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 \( \tilde{f}(x) = (x^2 + c) \mod N \)
- 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>example</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>

**AofA lecture 9**

- Rho length
  - Def. The rho-length of a function at a given point is the number of iterates until it repeats.

**Application: Pollard's rho-method for factoring**

Factors an integer \( N \) by iterating a random quadratic function to find a cycle.

1. How does it work?
   - A. Iterate \( f(x) = x^2 + c \) until finding a cycle using Floyd’s algorithm. Use a random value of \( c \) and start at a random point.

2. Pollard’s algorithm
   - long \( a = (\text{long}) (\text{Rand}() \times c) \mod N \); \( b = a; \)
   - long \( c = (\text{long}) (\text{Rand}() \times c) \mod N \); \( d = 1; \)
   - while \( (d == 1) \) { 
     a = \((a^2 + c) \mod N \);
     b = \((b^2 + c) \mod N \);
     if \( (a == b) \) \( d = \gcd(a - b, N, \text{N}) \); 
     else \( d = \gcd(b - a, N, \text{N}) \); 
   }
   // \( d \) is a factor of \( N \).

**pp. 534–536**
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

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()

```
% 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
```

```
uniform(100)
gaussian(9.0, .2)
uniform(10.0, 99.0) bernoulli(.5)
```

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

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

– John von Neumann

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

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

A solution. “Knuth-Yates shuffle”.

```java
public 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 *visualizing* combinatorial structures
Properties of random mappings (for validation)

[See AofA lecture 9 and Section 7 in Analysis of Algorithms]

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 *randomly permute* browsers.
- But IE was still favored.
- Why? They used a "random method".

---

**good method**

*assign random keys, then sort*

**good method**

*Knuth-Yates permutation*

**random method**

*sort with comparator that returns a random value*

makes no sense

---

“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><strong>5034</strong></td>
<td>1248</td>
<td>2542</td>
<td>571</td>
<td>605</td>
</tr>
</tbody>
</table>

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

**Counting sequence**

$$T_n = \frac{1}{n+1} \binom{2n}{n} = \frac{1}{2n} (1 - \sqrt{1 - 4n})$$

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.
Remy’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).

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)
Rémy’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);
    Node 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.
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
Rejection

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*

001111000110110001
111011101110110010
100011111111001101
10001111010001001110
0110001010101101100
0101001111110000110
101000011011101101
10110110010110101
001001110010110111
111100000011111100
010000011110001001
111111110101101111

*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
00110001001110110001010110011110010011110010011110010111

% java RandomBitsReject 50 3
11111110111111110110010111110111111110110111

% java RandomBitsReject 50 2
01101110110111101010101101011011011111101111101

1 trial
89 trials (?)
50490 trials (!!!)
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 $1.0917 \times 0.96328^N \approx 0.000000346$ for $N = 400$
Anticipated rejection

Generally not necessary to generate the whole object.

Ex: random 30-bit string with no 000

```
100101111101100010010000001010
10100010100111011001011110011
1101000101111000110010110011
001011110110110101010111100100
10111110100110101111111111101000
101001011001010110000100010010
00011011110100010010101100100
01111000100100010101010011001
111000100110001111001000010001
1011110000101010010110011110010
11101010011010110010010011001
01101010001001011111111010110
01001001000111101000000011001
10110010000011010110010001001
01111011111000011100111010000
0110000010010101110001110110110
0011101010010000100000001010111
010000001111011010000101010011
101111111110101100110100110011
```  

only need 14 bits on average (see AoA Lecture 8)

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**

**Recursive sampler**

**Precomputed probabilities**

**Ex: AoF lecture 6 (details revisited soon)**

- **Catlan distribution**: Probability that the root is of rank $k$ in a randomly chosen binary tree with $N$ nodes.

- Aside: Generating random binary trees

```java
class RandomBST {
    private Node root;
    private int n;
    private int d;
    private class Node {
        private Node left, right;
        private int n, d;
    }
    public RandomBST(int N) {
        root = generate(N, 0);
    }
    private Node generate(int N, int d) {
        // See code at right...
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        RandomBST t = new RandomBST(N);
        t.show();
    }
}
```
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

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

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 $1/\phi$

```java
private String B00(int N)
{
    if (N == 0) return "";
    if (N == 1) return "0";
    if (StdRandom.bernoulli(1.0/phi))
        return B00(N-1) + "1";
    else
        return B00(N-2) + "10";
}
```
Example 2: Recursive method for random binary trees

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

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

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

Precomputed probabilities

Recursive sampler

Specification

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;
}
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 (AoF lecture 6):

```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-1; i++)
        T[i] = T[i-1] * (4*i-2)/(i+1);
    cat[0] = new double[1];
    cat[0][0] = 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;
}
```

**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).
"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.
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

[GenRGens](https://www.lri.fr/genrgens)

Lumbroso, *to appear.*

- Free publicly available modern implementation

[Python](https://www.python.org) [Sage](https://www.sagemath.org)

**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><em>not always</em></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
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 $x < x_0$

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

**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.

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

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

$$= x \frac{A'(x)}{A(x)}$$
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: | 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 $z$ |
**Dijoint union and Cartesian product construction for analytic samplers**

### Disjoint union

**Analytic sampler for** $A = B + C$

```java
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** $A = B \times C$

```java
return compose(B, C)
```

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

$$
\frac{x^{|a|}}{A(x)} = \frac{x^{|b|+|c|}}{A(x)} = \frac{x^{|b|+|c|}}{B(x)C(x)} = \frac{x^{|b|}}{B(x)} \frac{x^{|c|}}{C(x)}
$$

combines B and C into a single object
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 $\frac{|a|}{A(x)}$ |
|--------------|---------|--------------------------------------------------------------------------------|
| 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 $\text{compose}(B, C)$ | $\frac{x^{|a|}}{A(x)} = \frac{x^{|b|+|c|}}{A(x)} = \frac{x^{|b|+|c|}}{B(x)C(x)} = \frac{x^{|b|} x^{|c|}}{B(x) C(x)}$ |

<table>
<thead>
<tr>
<th>notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$a$</td>
</tr>
<tr>
<td>$</td>
</tr>
<tr>
<td>$A(z)$</td>
</tr>
<tr>
<td>$x_0$</td>
</tr>
<tr>
<td>$x$</td>
</tr>
</tbody>
</table>
Example 1: Analytic sampler for random bitstrings without long runs

**Specification**

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

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

**Symbolic transfer**

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

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

**GF equation**

**Analytic sampler**

```java
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); }
```

- **E**
  - return \( \varepsilon \)

- **Z**
  - return \( \cdot \)

- **A = B + C**
  - \( u = \text{StdRandom.bernoulli}(B(x)/A(x)) \)
  - if (u) return B else return C

- **A = B \times C**
  - return \( \text{compose}(B, C) \)

**concatentation**

returns "0" "00" "000" with equal probability

see Lecture 1
Critical question about an analytic sampler

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

```java
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);
}
```

\( B_4, \) the class of all bitstrings with no 0

\[
B_4 = \mathbb{Z}_{<4} (E + Z_1 B_4)
\]

\[
B_4(z) = (1 + z + z^2 + z^3)(1 + zB_4(z)) = \frac{1 + z + z^2 + z^3}{1 - z - z^2 - z^3 - z^4}
\]

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.
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 0^4

\[
B(r) = \frac{1 + r + r^2 + r^3}{1 - r - r^2 - r^3 - r^4} \sim \frac{C}{1 - \beta r} \quad \text{with} \quad \beta = 1.9276
\]

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

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

Ex. random string with no 0^4

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

\[
r \sim \frac{N}{(N + 1)\beta}
\]
Practical consideration: variance

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

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

Important note. **Variance is not small**

Exercise: compute it!

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".
Summary

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

Ex. random string with no $0^4$

```java
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);
    }
}
```

% java RandomStringNo4 50
001000100010001000110011001100011100101110
% java RandomStringNo4 50
0010010001010001000101001011001011001110
% java RandomStringNo4 50
0101010110010101000110010100101001001001
Analytic sampler for random binary trees

Specification

Symbolic transfer

GF equation

Analytic sampler

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

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

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;
}

<table>
<thead>
<tr>
<th>E</th>
<th>return □</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>return ●</td>
</tr>
<tr>
<td>A = B + C</td>
<td>if ( u &lt; \frac{B(x)}{A(x)} ) return B else return C</td>
</tr>
<tr>
<td>A = B \times C</td>
<td>return \text{compose}(B, C)</td>
</tr>
</tbody>
</table>
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 \( about \) \( N \) 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

$p_0 = p_2 = 1/2$

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

ternary

$p_0 = p_1 = p_2 = 1.0/3.0$

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

binary

0–1–2 (Motzkin)
Aside: iterative (breadth-first) singular analytic samplers for trees

Idea. Implement a "Galton-Watson process".

- Use parent-link representation
- ith entry on queue is parent of i
- Generate k children for each node w.p. pk
  (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 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>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>return $\epsilon$</td>
</tr>
<tr>
<td>$Z$</td>
<td>return $\cdot$</td>
</tr>
<tr>
<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>$A = B \star C$</td>
<td>return $\text{compose}(B, C)$</td>
</tr>
</tbody>
</table>
| $A = \text{SEQ}(B)$ | $k = \text{geometric}(B(x))$
return $\text{compose}(B, B, \ldots, B)$ |
| $A = \text{SET}(B)$ | $k = \text{poisson}(B(x))$
return $\text{compose}(B, B, \ldots, B)$ |
| $A = \text{CYC}(B)$ | $k = \text{logseries}(B(x))$
return $\text{compose}(B, B, \ldots, B)$ |

**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 $< x_0$

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

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

```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]*(k-1)/(1.0*k);
```
Analytic sampler for sets of cycles (permutations)

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

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

\[
\begin{align*}
\text{private static Queue generate(double r)} & \quad \text{construction} \\
\{ & \\
\text{int lambda = Math.log(1.0/(1.0 - r));} & \\
\text{int k = StdRandom.poisson(lambda);} & \\
\text{Queue<Integer> q = new Queue<Integer>();} & \\
\text{for (int i = 0; i < k; i++)} & \\
\text{q.enqueue(logseries(r));} & \\
\text{return q;} & \\
\} & \quad \text{sampler}
\end{align*}
\]

| \( A = \text{SET}(B) \) | k = poisson(B(x)) \quad \text{return compose}(B, B, \ldots, B) |
| \( A = \text{CYC}(B) \) | k = logseries(B(x)) \quad \text{return compose}(B, B, \ldots, B) |
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

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;
}

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

### Specification

- **Symbolic transfer**

### GF equation

### Analytic sampler

#### construction

\[ Z \]

- return \( \bullet \)

#### sampler

\[ A = \text{SET}(B) \]

- \( k = \text{poisson}(B(x)) \)
- return \( \text{compose}(B, B, ..., B) \)

\[ A = \text{CYC}(B) \]

- \( k = \text{logseries}(B(x)) \)
- return \( \text{compose}(B, B, ..., B) \)

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)

Another approach: iterative (breadth-first) singular analytic samplers for trees

```java
private static Queue generate(double[] p) {
    Queue<Integer> tree = new Queue<Integer>();
    int root = 0;
    tree.enQueue(0);
    while (root < tree.size()) {
        int k = StdRandom.discrete();
        for (int j = 3; j <= k; j++)
            tree.enQueue(root);
        root++;
    }
    return tree;
}
```
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).

```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;
}
```

Analytic sampling

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

```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;
}
```

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).
Various random objects produced by analytic samplers

- Skew plane partition
  Bodini, Fusy, and Pivoteau, 2006

- Cactus graph, Bahrani and Lumbroso, 2016

- Polynomial tilings
  Bendkowski, Bodini, and Dovgal, 2018

- Reluctant random walk, Lumbroso, Mishna, and Ponty, 2016

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.