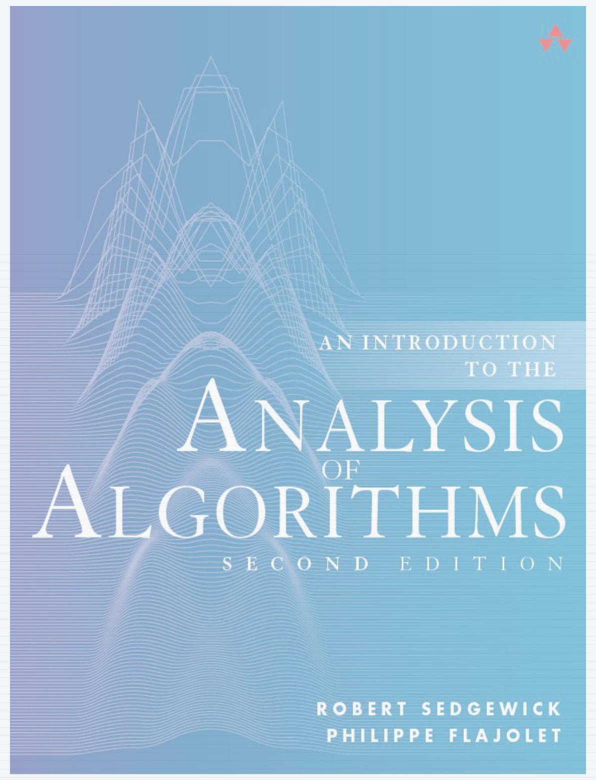
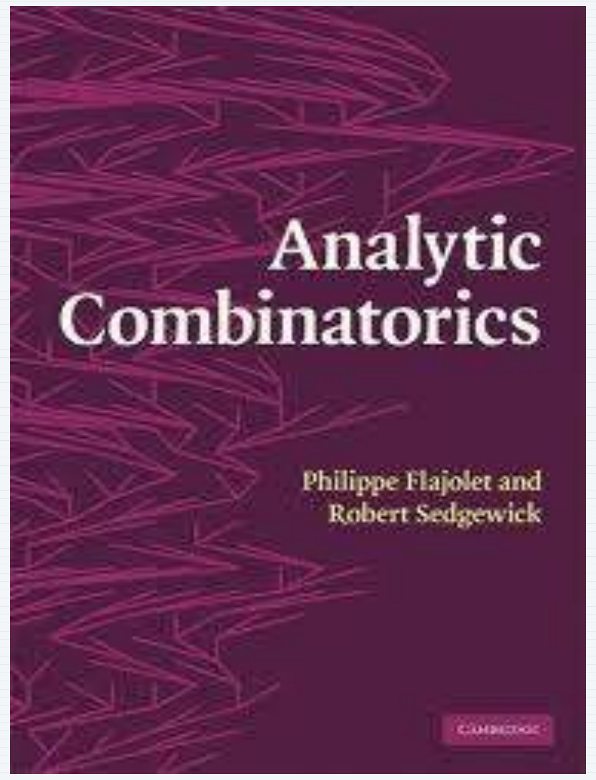


Analysis of Algorithms

Original MOOC title: ANALYTIC COMBINATORICS, PART ONE



Analytic Combinatorics

Original MOOC title: ANALYTIC COMBINATORICS, PART TWO

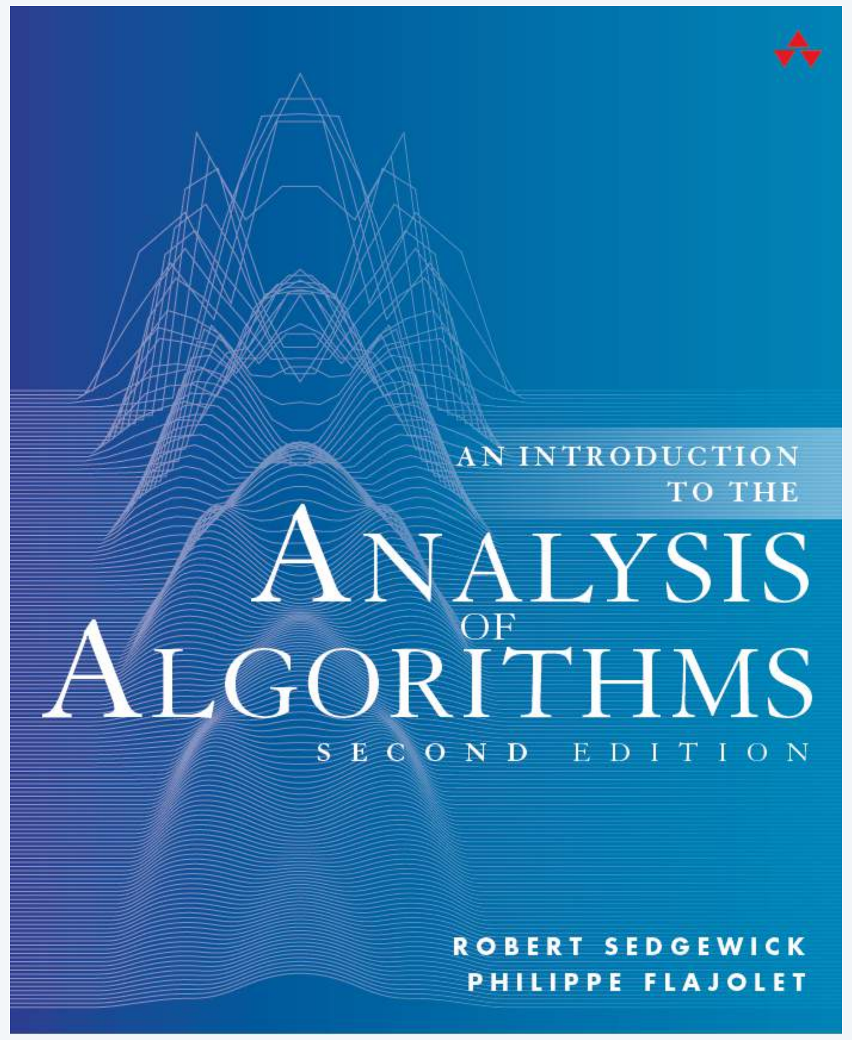
http://aofa.cs.princeton.edu

http://ac.cs.princeton.edu

Overview

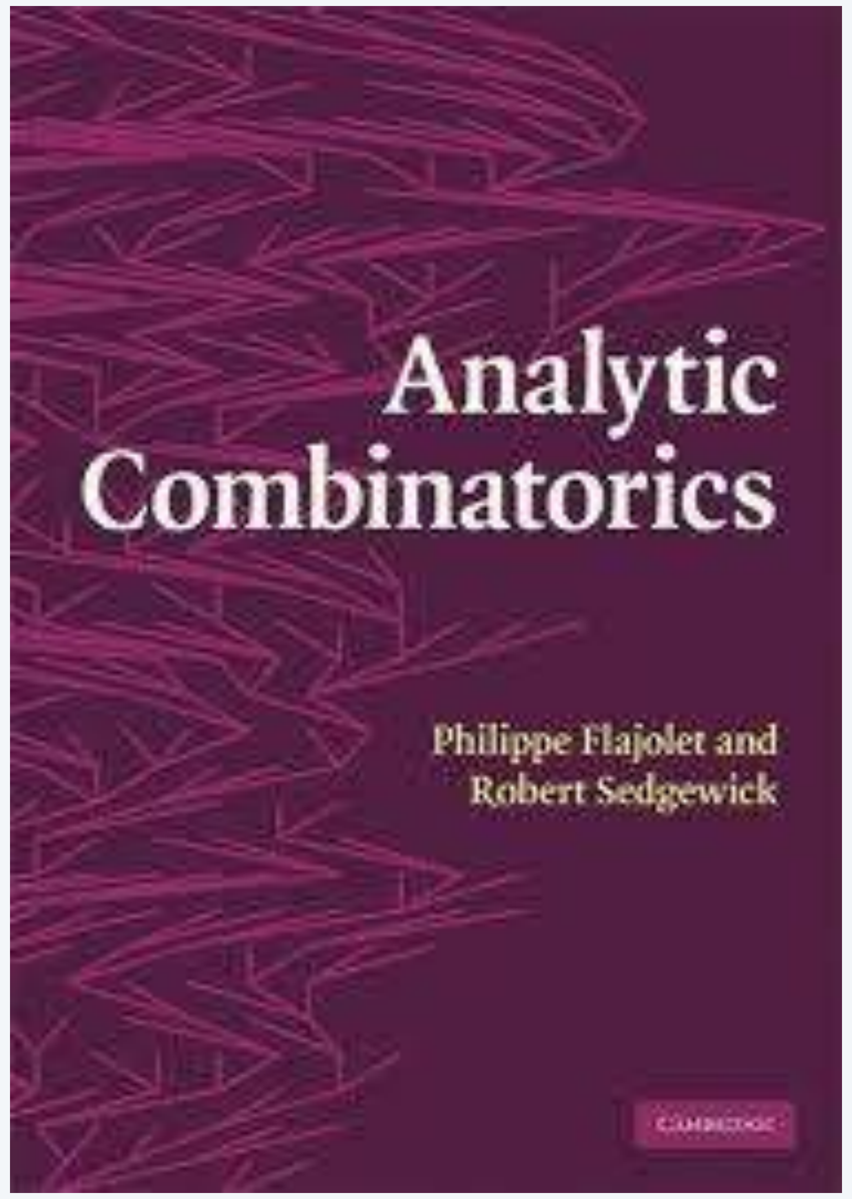
Analysis of algorithms

- 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.

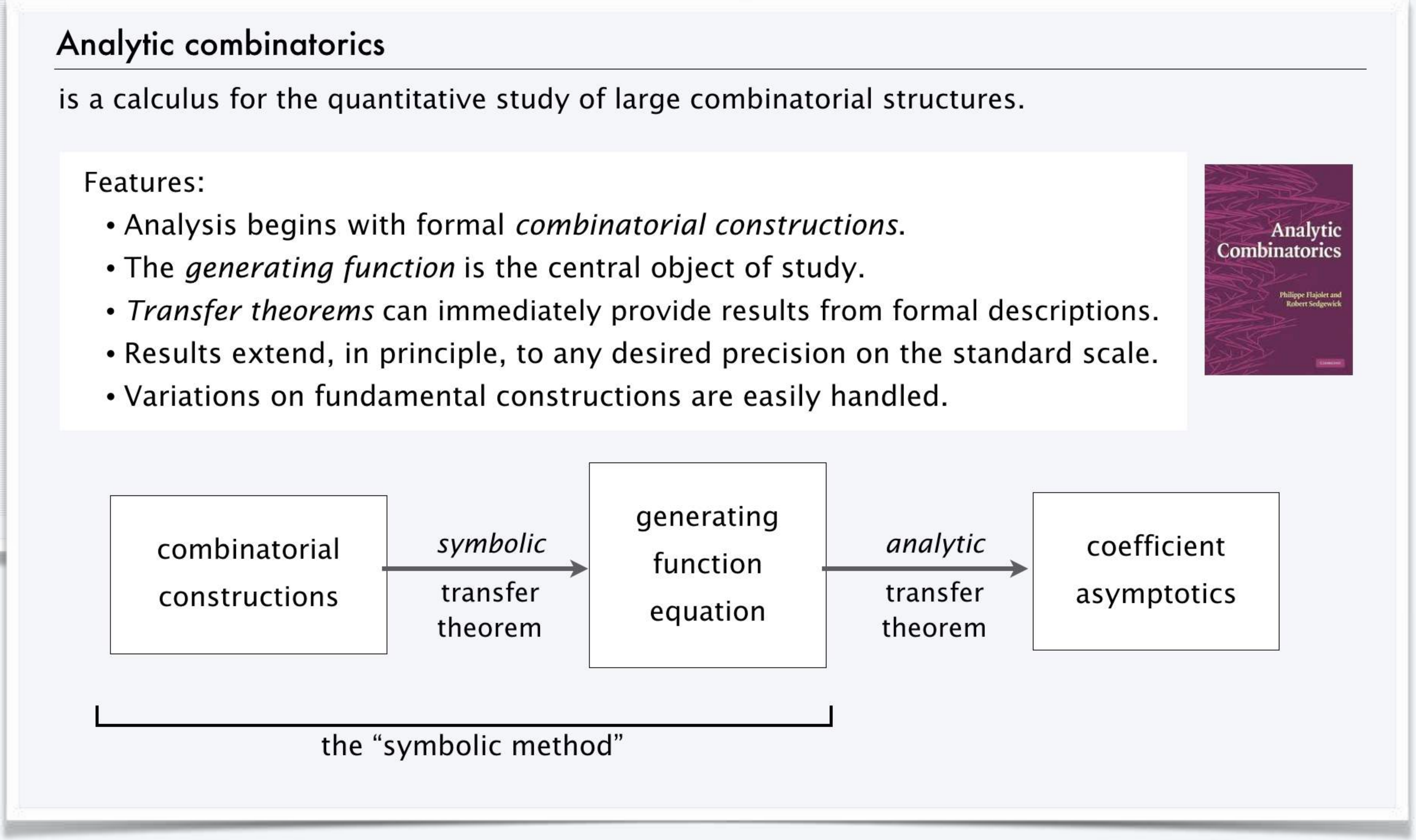


Context for this lecture

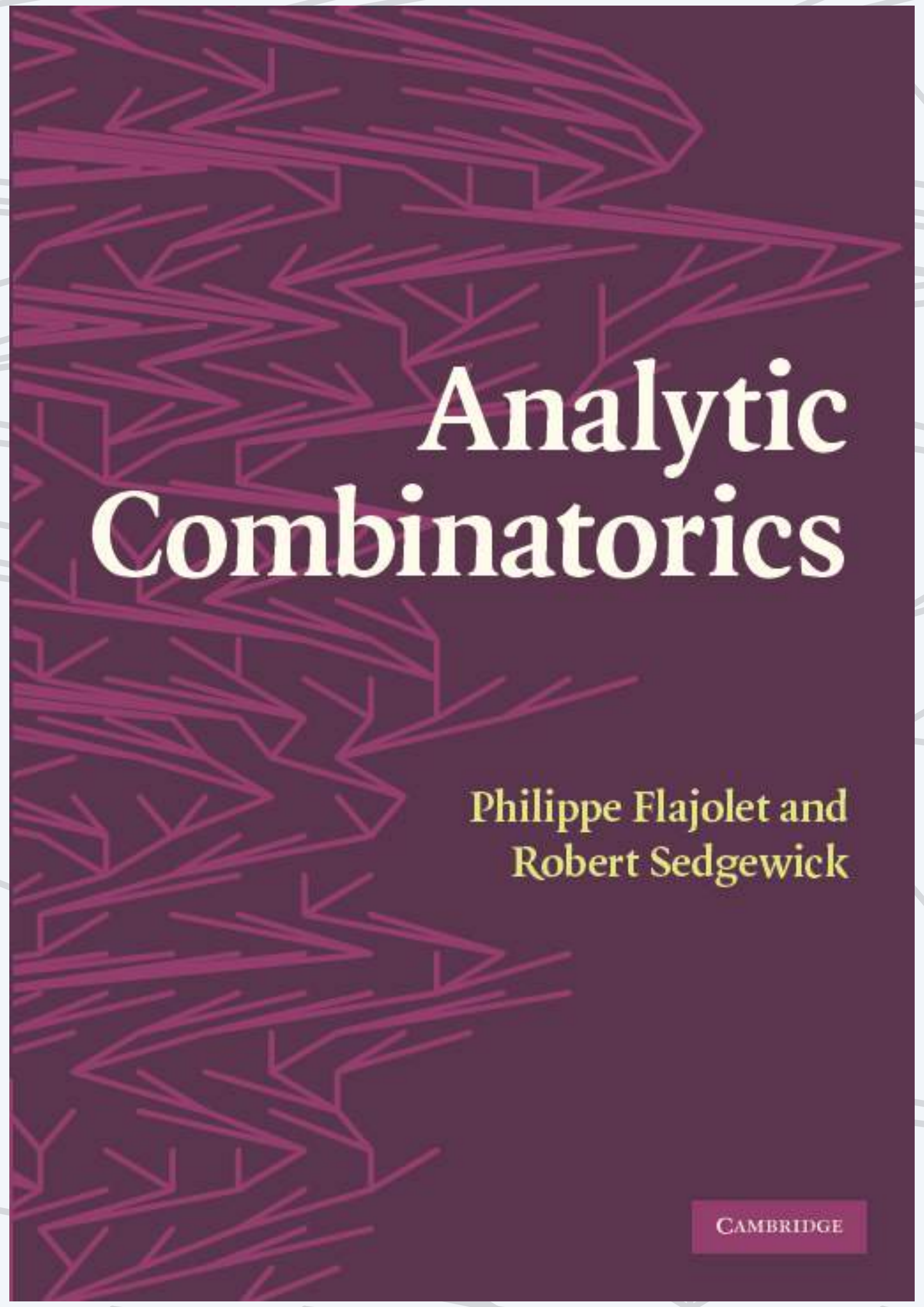
Purpose. Prepare for the study of analytic combinatorics *in the context of an important application.*

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

AofA lecture 5



Random Sampling of Combinatorial Objects



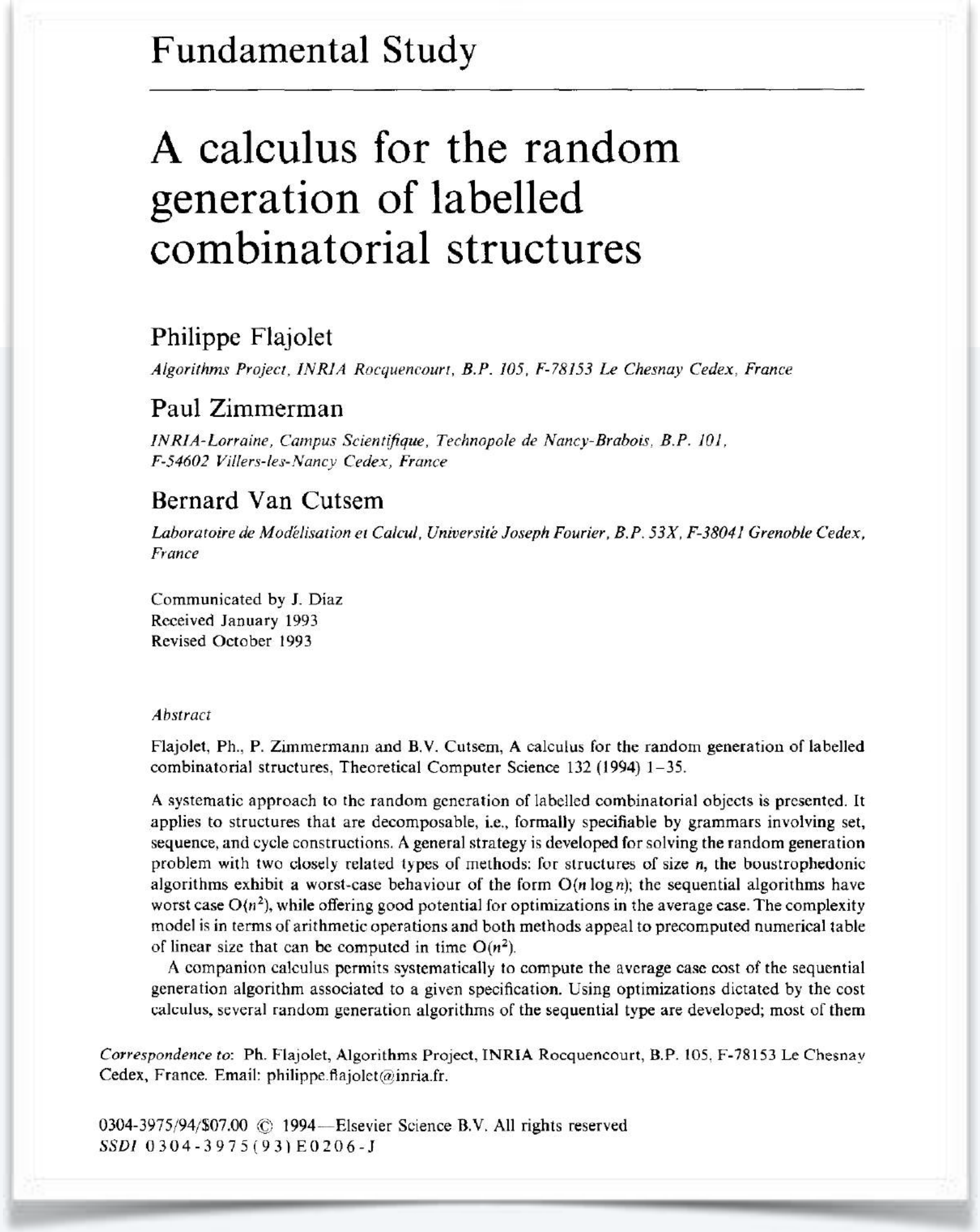
http://ac.cs.princeton.edu

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

Algorithms Project, INRIA Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France

Paul Zimmerman

INRIA-Lorraine, Campus Scientifique, Technopole de Nancy-Brabois, B.P. 101, F-54602 Villers-les-Nancy Cedex, France

Bernard Van Cutsem

Laboratoire de Modélisation et Calcul, Université Joseph Fourier, B.P. 53X, F-38041 Grenoble Cedex, France

Communicated by J. Diaz
Received January 1993
Revised October 1993

Abstract

Flajolet, Ph., P. Zimmermann and B.V. Cutsem, A calculus for the random generation of labelled combinatorial structures, Theoretical Computer Science 132 (1994) 1–35.

A systematic approach to the random generation of labelled combinatorial objects is presented. It applies to structures that are decomposable, i.e., formally specifiable 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 methods: for structures of size n, the boustrophedonic algorithms exhibit a worst-case behaviour of the form $O(n \log n)$; the sequential algorithms have worst case $O(n^2)$, while offering good potential for optimizations in the average case. The complexity model is in terms of arithmetic operations and both methods appeal to precomputed numerical table of linear size that can be computed in time $O(n^2)$.

A companion calculus permits systematically to compute the average case cost of the sequential generation algorithm associated to a given specification. Using optimizations dictated by the cost calculus, several random generation algorithms of the sequential type are developed; most of them

Correspondence to: Ph. Flajolet, Algorithms Project, INRIA Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France. Email: philippe.flajolet@inria.fr.

0304-3975/94/$07.00 © 1994—Elsevier Science B.V. All rights reserved
SSDI 0304-3975(93)E0206-J

Combinatorics, Probability and Computing (2004) 13, 577–625. © 2004 Cambridge University Press
DOI: 10.1017/S0963548304006315 Printed in the United Kingdom

Boltzmann Samplers for the Random Generation of Combinatorial Structures

PHILIPPE DUCHON,[1] PHILIPPE FLAJOLET,[2] GUY LOUCHARD[3] and GILLES SCHAEFFER[4]

[1] LaBRI, Université de Bordeaux I, 351 Cours de la Libération, F-33405 Talence Cedex, France (e-mail: duchon@labri.fr)

[2] Algorithms Project, INRIA-Rocquencourt, F-78153 Le Chesnay, France (e-mail: Philippe.Flajolet@inria.fr)

[3] Université Libre de Bruxelles, Département d'informatique, Boulevard du Triomphe, B-1050 Bruxelles, Belgique (e-mail: louchard@ulb.ac.be)

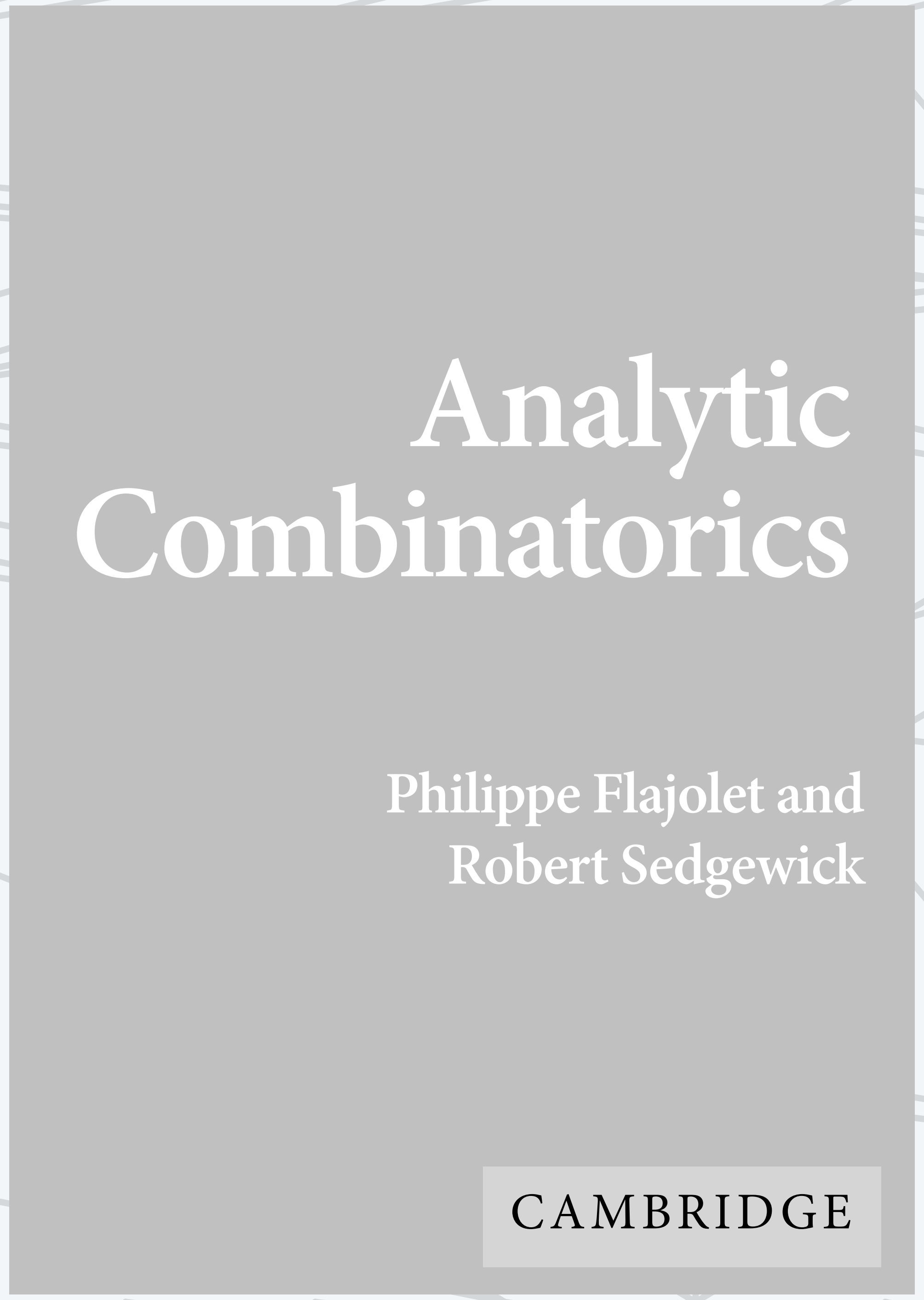
[4] Laboratoire d'Informatique (LIX), École Polytechnique, 91128 Palaiseau Cedex, France (e-mail: Gilles.Schaeffer@lix.polytechnique.fr)

Received 1 January 2003; revised 31 December 2003

This article proposes a surprisingly simple framework for the random generation of combinatorial configurations based on what we call *Boltzmann models*. The idea is to perform random generation of possibly complex structured objects by placing an appropriate measure spread over the whole of a combinatorial class – an object receives a probability essentially proportional to an exponential of its size. As demonstrated here, the resulting algorithms based on real-arithmetic operations often operate in linear time. They can be implemented easily, be analysed mathematically with great precision, and, when suitably tuned, tend to be very efficient in practice.

1. Introduction

In this study, *Boltzmann models* are introduced as a framework for the random generation of structured combinatorial configurations, such as words, trees, permutations, constrained graphs, and so on. A Boltzmann model relative to a combinatorial class $\mathscr{C}$ depends on a *real-valued* (continuous) control parameter $x > 0$ and places an appropriate measure that is spread over the whole of $\mathscr{C}$. This measure is essentially proportional to $x^{|\omega|}$ for an object $\omega \in \mathscr{C}$ of size $|\omega|$. Random objects under a Boltzmann model then have a fluctuating size, but objects with the same size invariably occur with the same probability. In particular, a *Boltzmann sampler* (*i.e.*, a random generator that produces objects distributed according



http://ac.cs.princeton.edu

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."

– Alan Turing

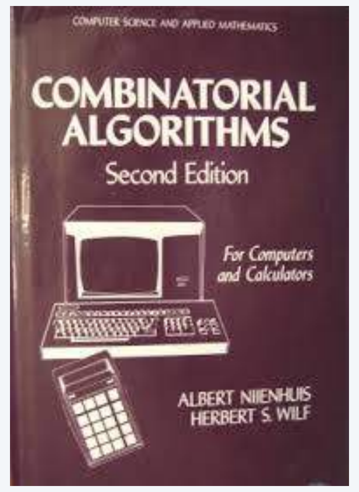
A. Turing. *The Chemical Basis of Morphogenesis*, Phil. Trans. of the Royal Society of London, 1952.

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

Devroye, *Non-Uniform Random Variate Generation*, Springer, 1986

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

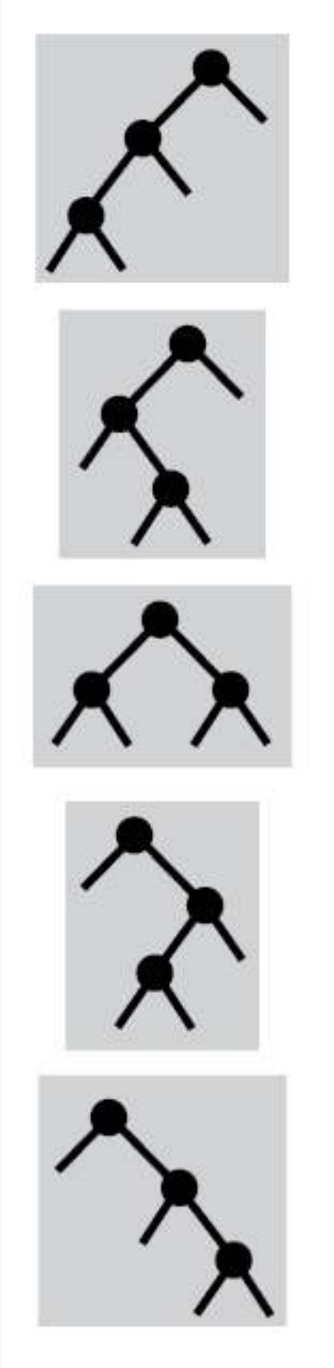
return each with probability 1/6

random mapping

0 0 0	1 0 0	2 0 0
0 0 1	1 0 1	2 0 1
0 0 2	1 0 2	2 0 2
0 1 0	1 1 0	2 1 0
0 1 1	1 1 1	2 1 1
0 1 2	1 1 2	2 1 2
0 2 0	1 2 0	2 2 0
0 2 1	1 2 1	2 2 1
0 2 2	1 2 2	2 2 2

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***.

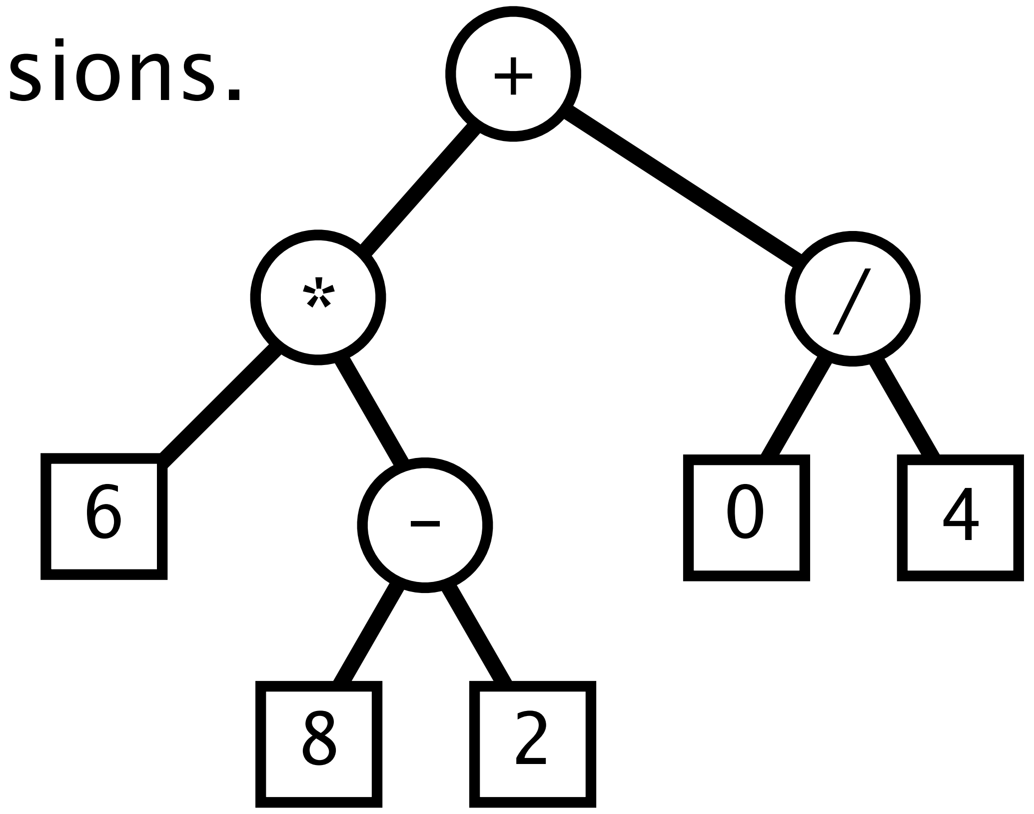
a 6,343 char RE for validating e-mail addresses, found on the web

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.



(6 * (8 - 2)) + (0 / 4)

binary operators	binary tree
unary-binary operators	Motzkin (0-1-2) tree
multiway operators	tree

Result. A realistic benchmark for program testing

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

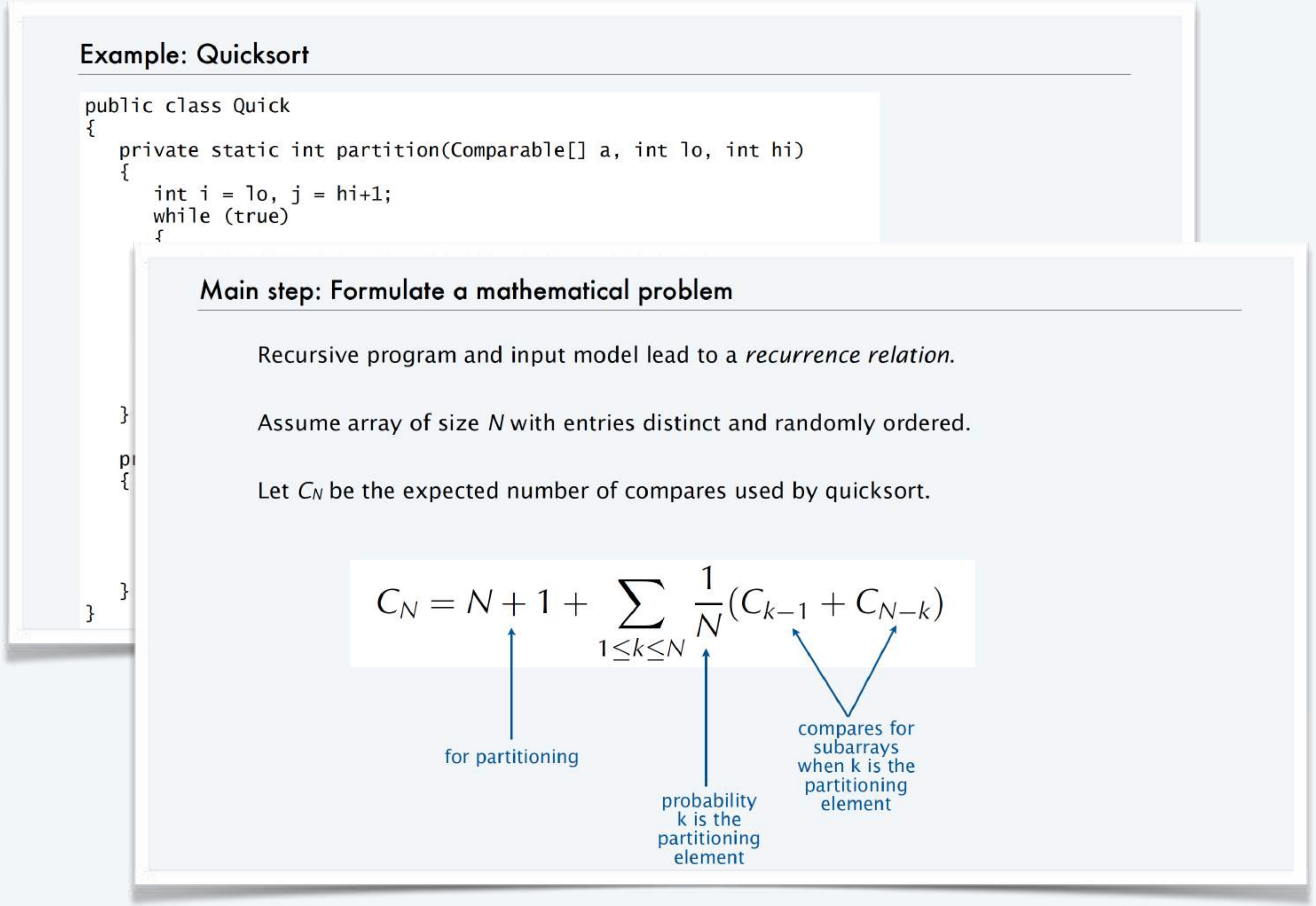
Problem. Improve a program with bad worst-case performance.

Classic example: Quicksort

Approach. Randomize the input.

- Start with a ***random permutation*** of the input
- Makes worst case *negligable*
- Enables mathematical analysis
- Makes performance *predictable* in practice

AofA lecture 5



Method of choice for a broad variety of applications.

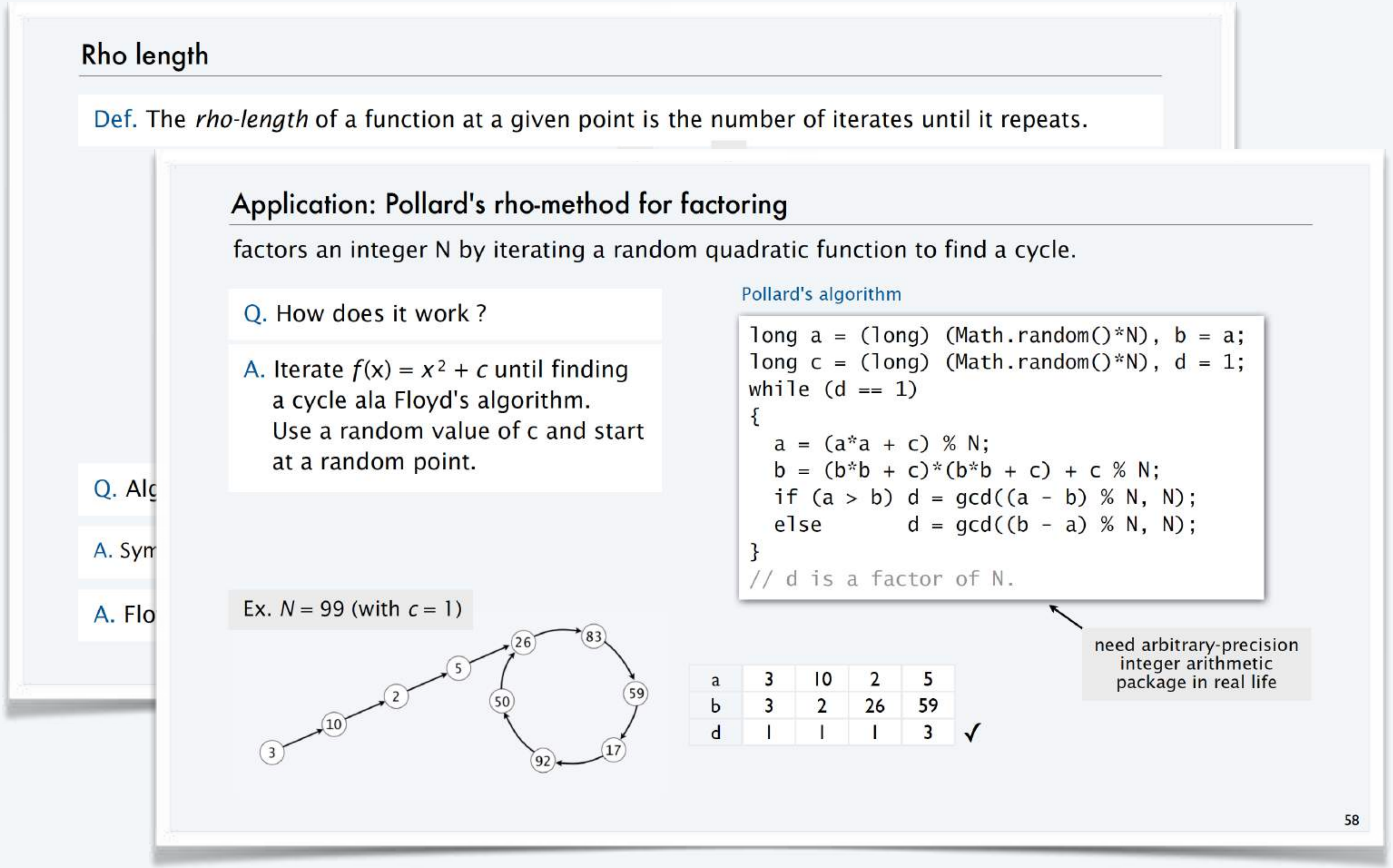
Application example III: Factoring

Problem. ***Factor*** a large integer N

Approach ("Pollard's rho method").

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

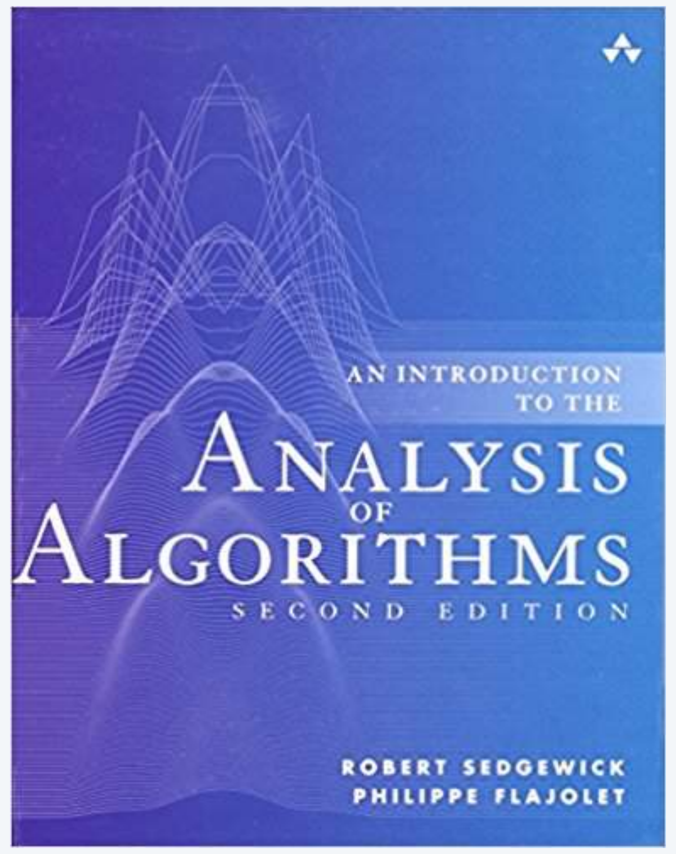
AofA lecture 9



Factors N in $N^{1/4}$ steps

example	*steps*
1237 · 4327	21
123457 · 654323	243
1234577 · 7654337	1478
12345701 · 87654337	3939
123456791 · 987654323	11225
1234567901 · 10987654367	23932

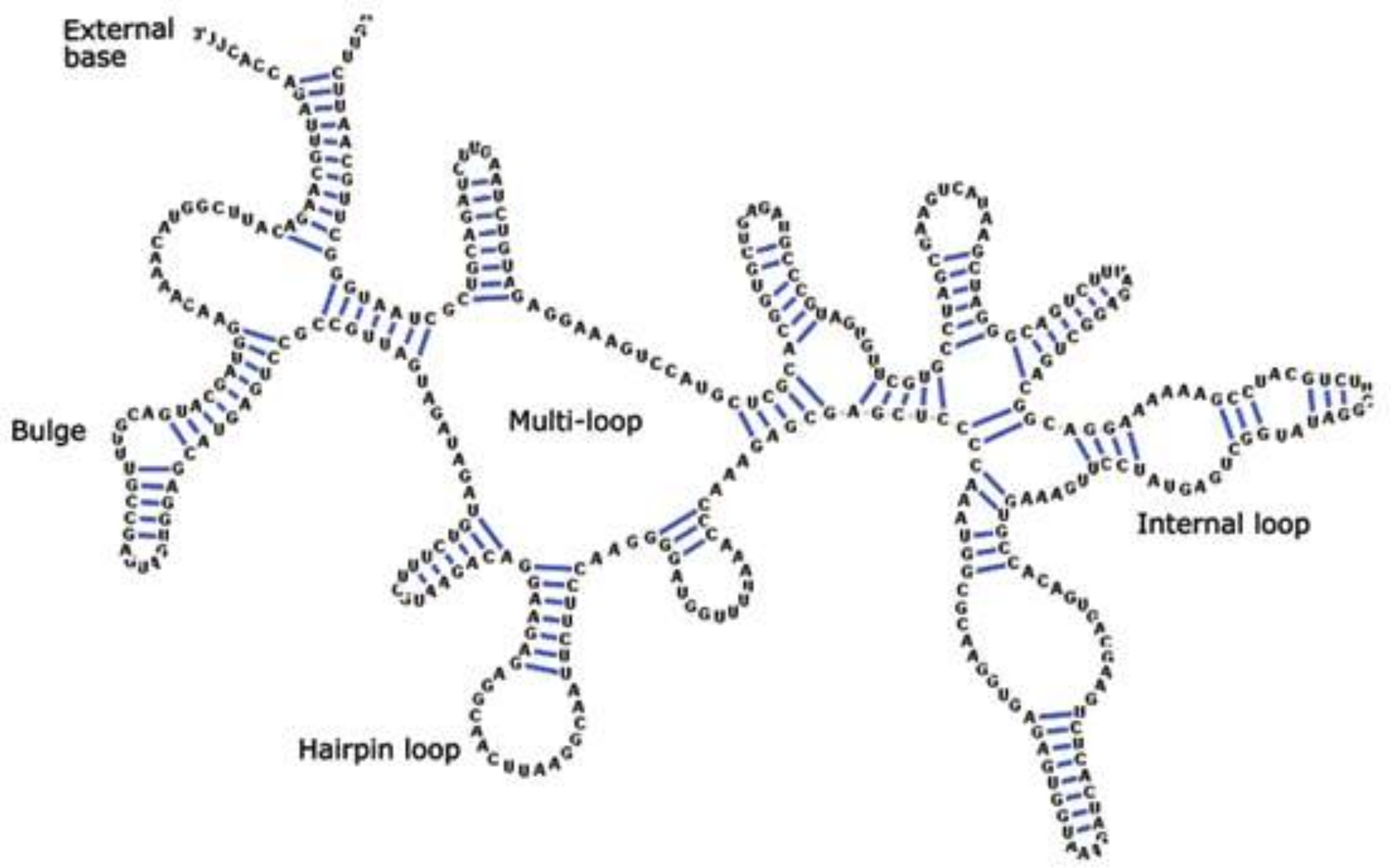
pp. 534–536



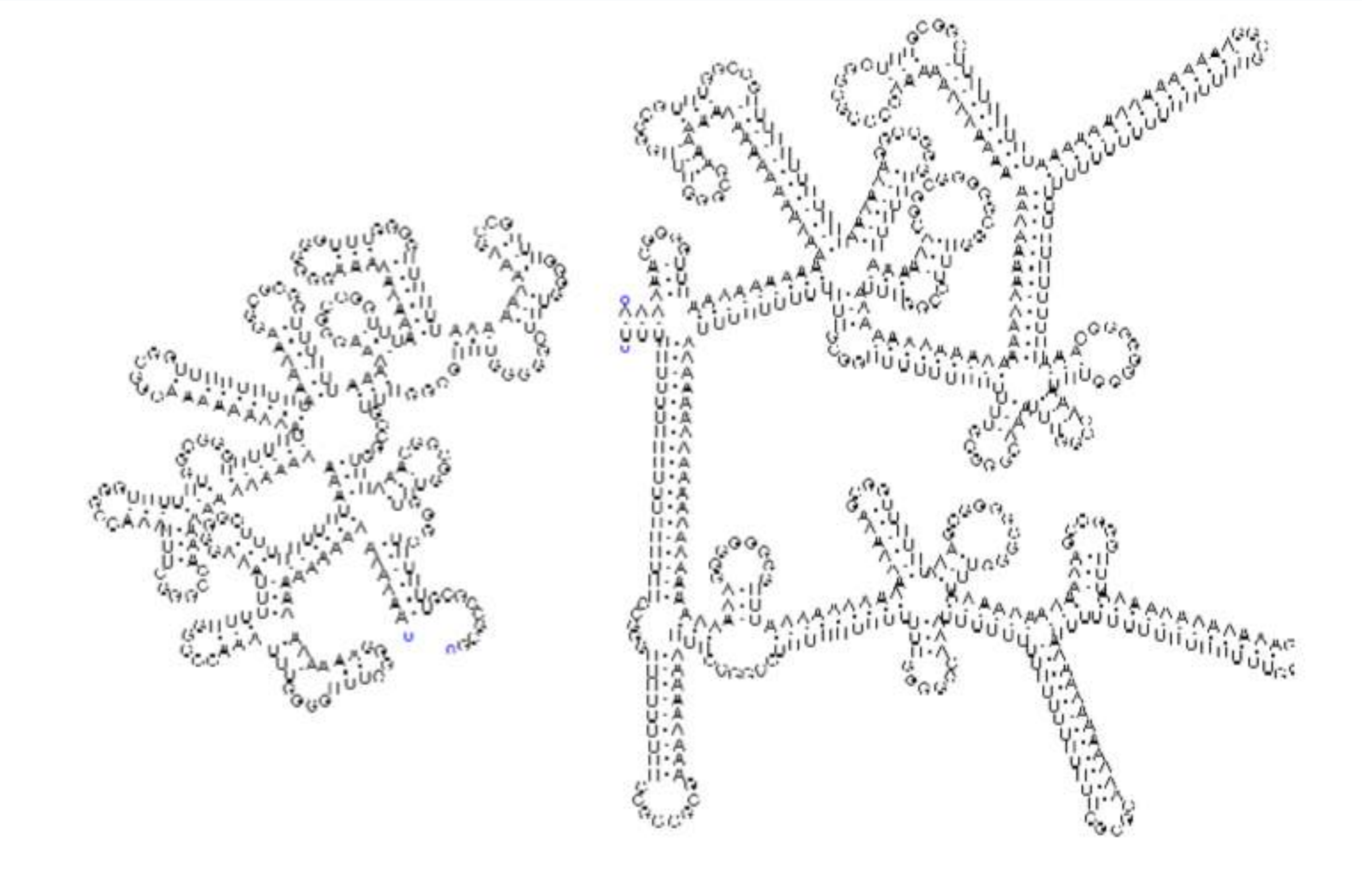
Application example IV: Bioinformatics

Problem. Develop a model for ***RNA secondary structures***.

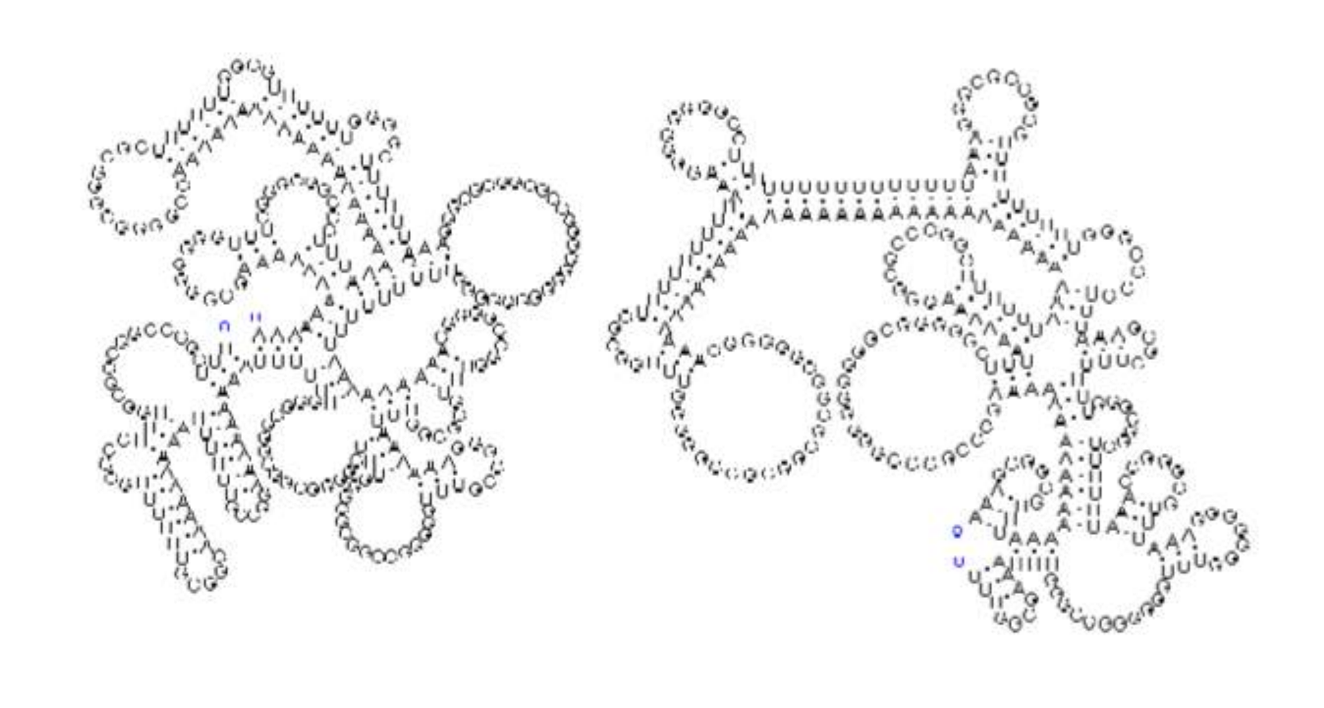
The secondary structure elements of Bacillus subtilis (M13175)



Randomly generated from a simple specification



with constraints

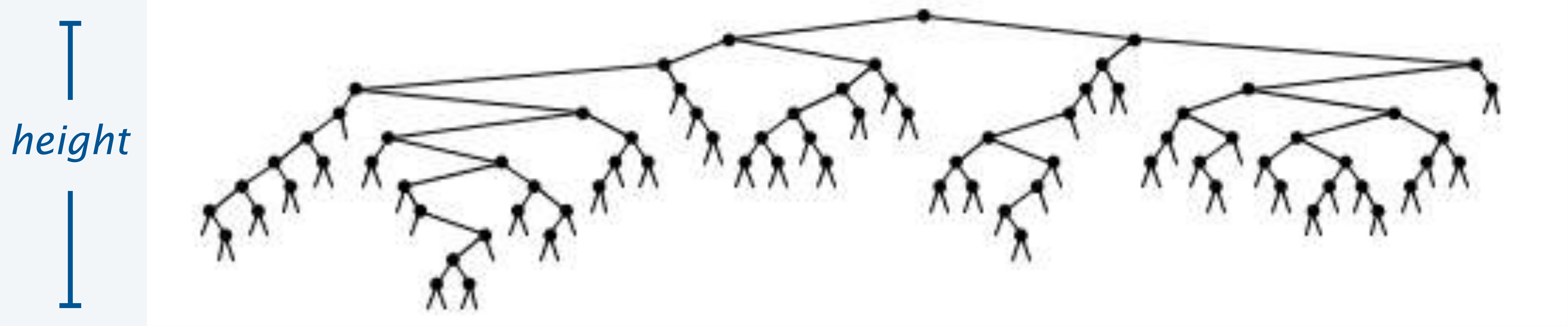


Many, many other applications in bioinformatics

A. Denise, Y. Ponty and M. Termier. *Random Generation of structured genomic sequences*, RECOMB'03 (poster session).

Application example V: Experimental mathematics

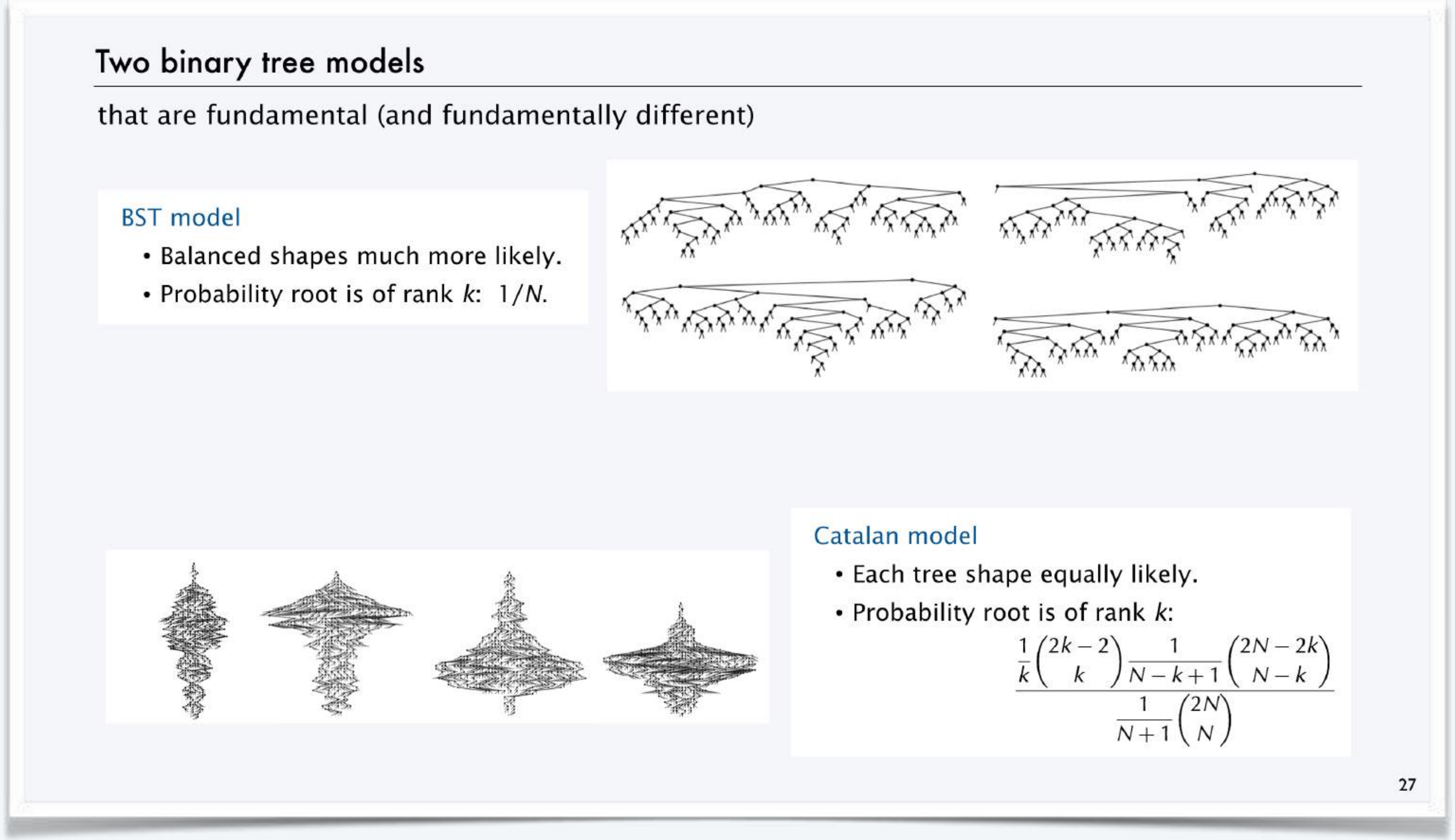
Problem. What is the average ***height*** of a ***binary search tree*** with N nodes?



Approach.

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

AofA lecture 6



History.

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

L. Devroye. *A note on the height of binary search trees*, JACM 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

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

– John von Neumann

discrete({ .5, .3, .1, .1 })

```
% 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
```

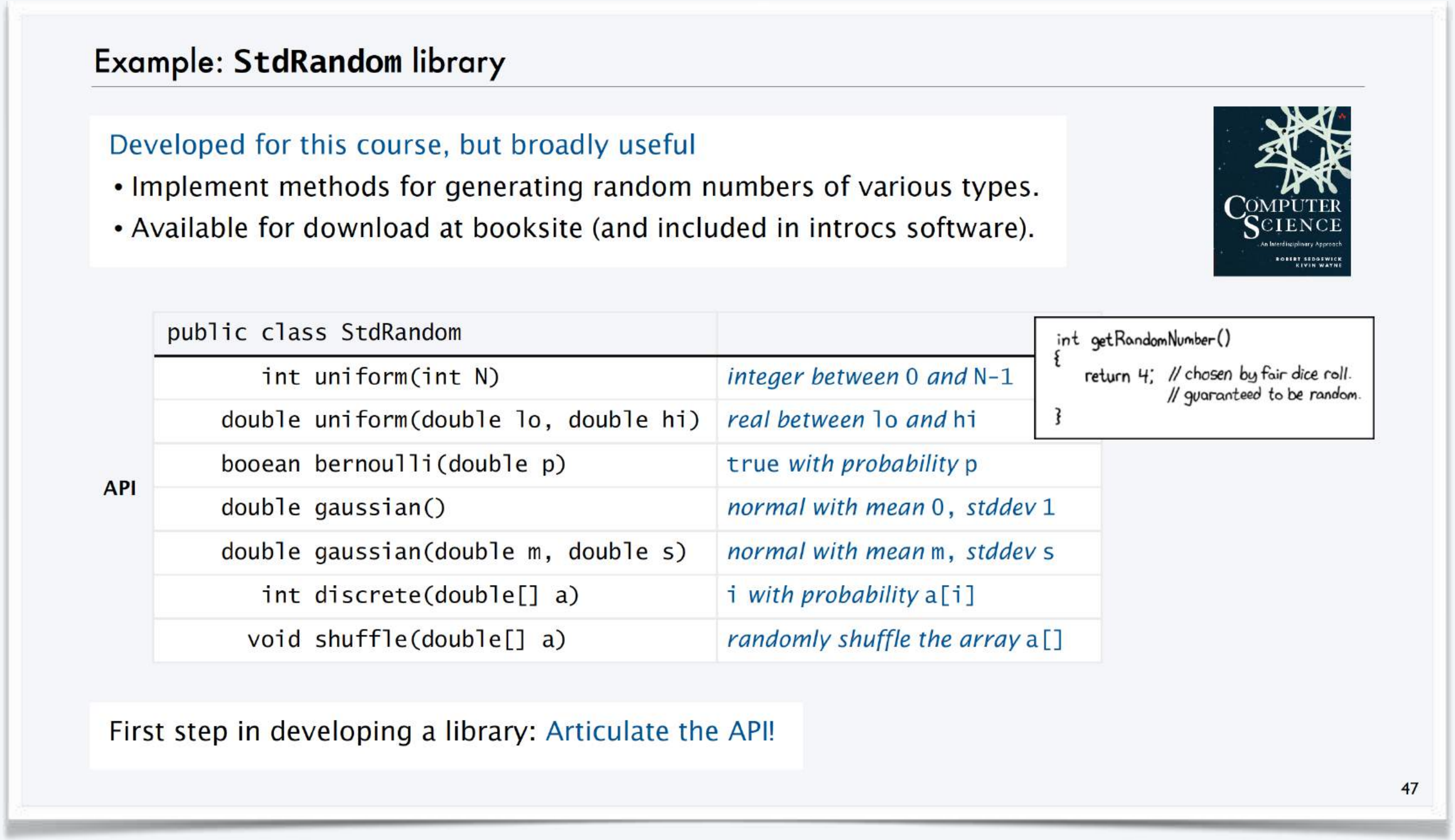
uniform(100)

uniform(10.0, 99.0)

bernoulli(.5)

gaussian(9.0, .2)

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



Example: **StdRandom** library

Developed for this course, but broadly useful

- Implement methods for generating random numbers of various types.
- Available for download at booksite (and included in introcs software).

API

public class StdRandom	
int uniform(int N)	*integer between* 0 *and* N-1
double uniform(double lo, double hi)	*real between* lo *and* hi
booean bernoulli(double p)	*true with probability* p
double gaussian()	*normal with mean* 0, *stddev* 1
double gaussian(double m, double s)	*normal with mean* m, *stddev* s
int discrete(double[] a)	i *with probability* a[i]
void shuffle(double[] a)	*randomly shuffle the array* a[]

```
int getRandomNumber()
{
   return 4;  // chosen by fair dice roll.
              // guaranteed to be random.
}
```

First step in developing a library: Articulate the API!

47

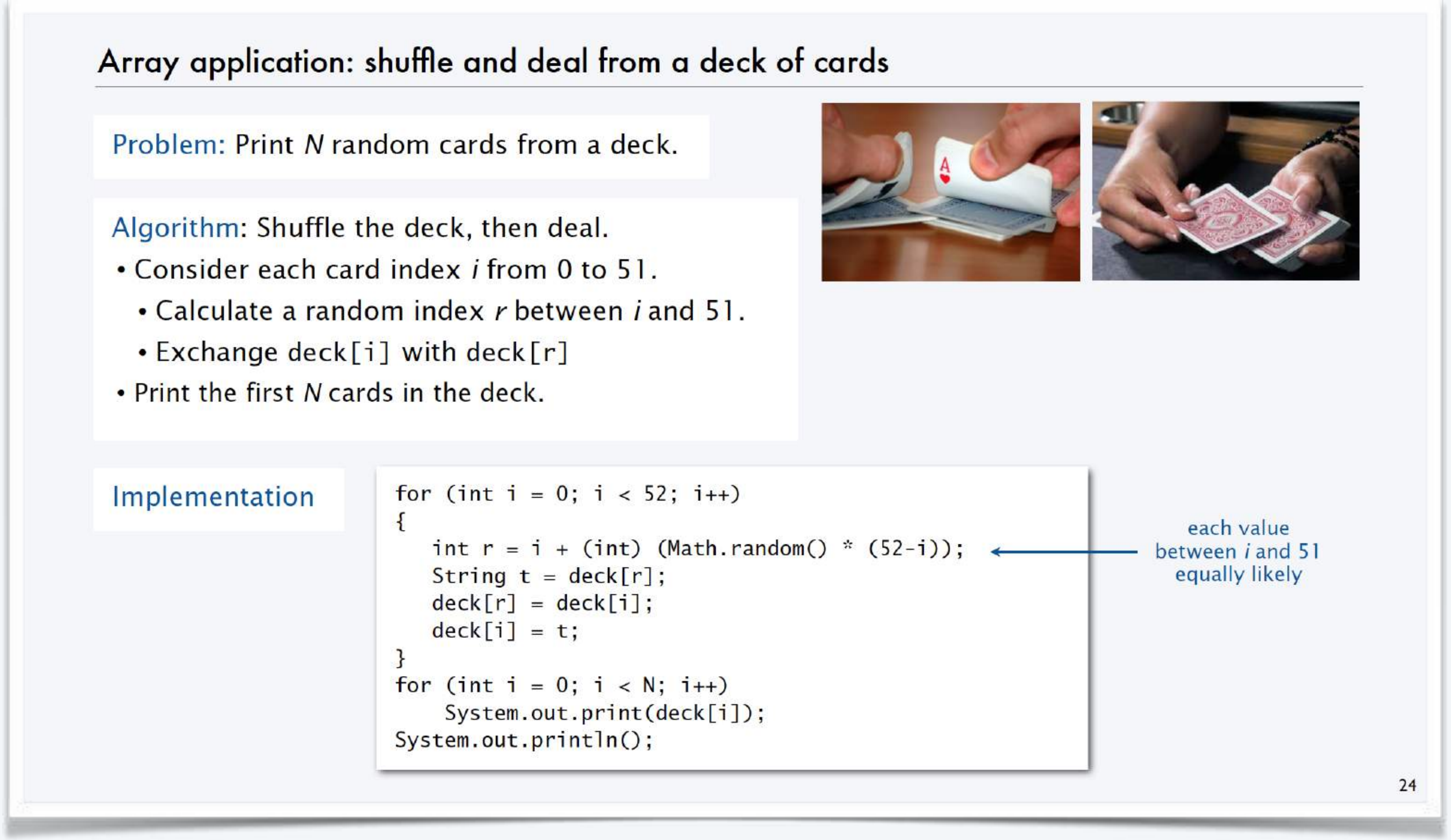
For a modern treatment in the context of this lecture, see Flajolet, Pelletier, and Soria, *On Buffon Machines and Numbers, SODA 2011.*

Random permutations

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

A solution. “Knuth-Yates shuffle”.

```
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();
   }
}
```

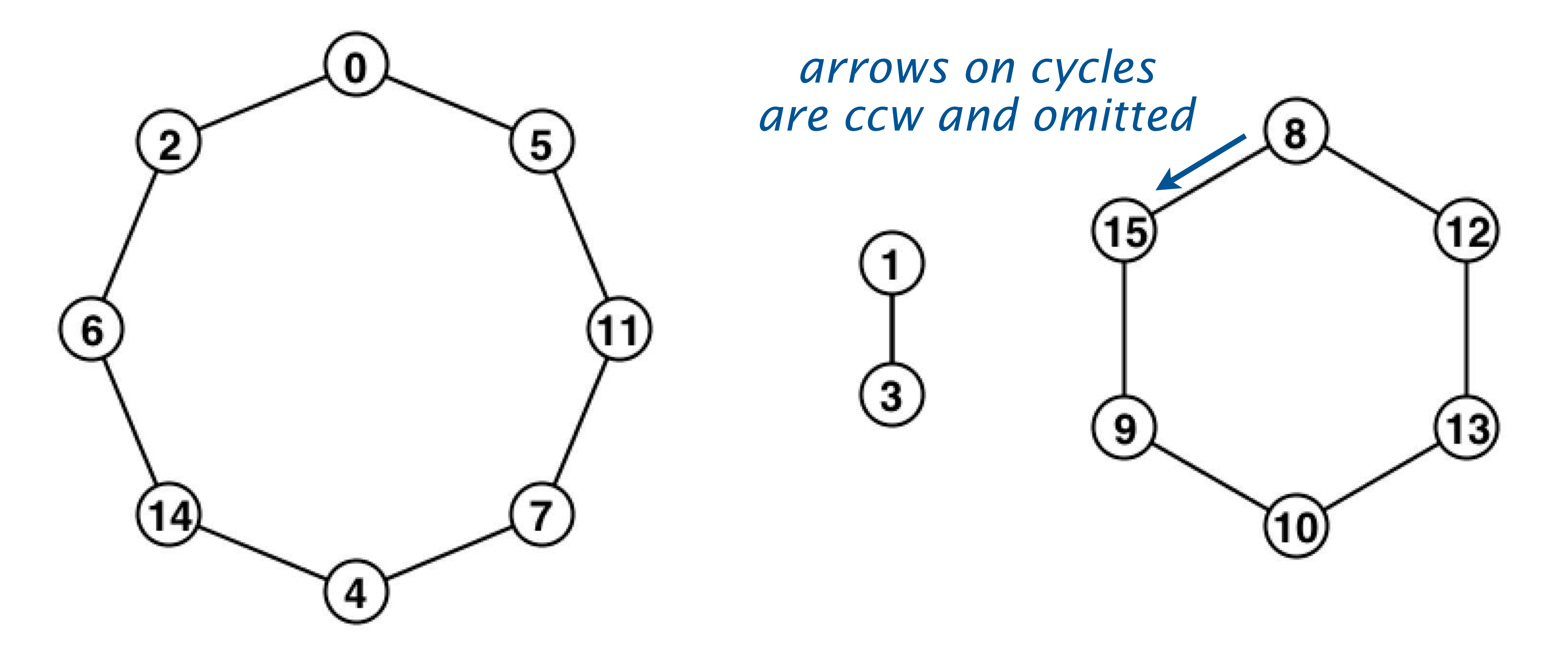


A poster child for the utility of arrays (CS lecture 3)

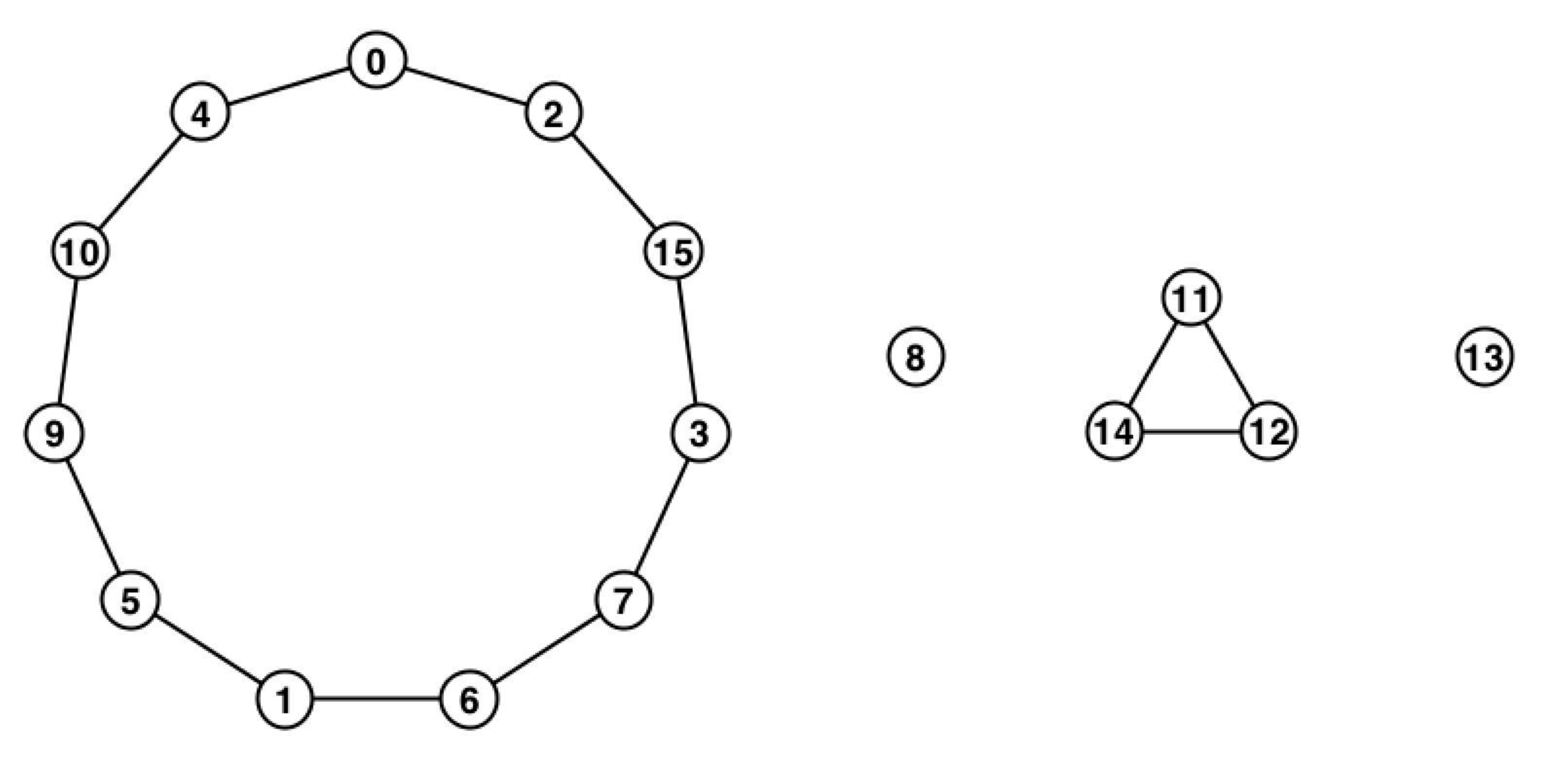
Proof of uniformity. *N* ! different permutations possible, all equally likely.

Three random permutations of size 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	6	1	7	0	14	11	15	10	13	5	8	12	4	9



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	6	0	15	10	1	7	3	8	5	9	14	11	13	12	2

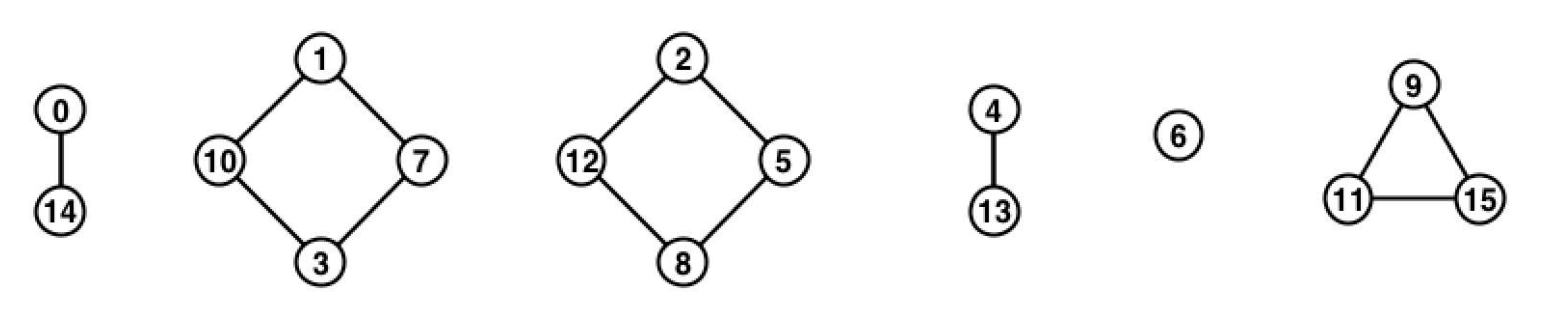


Q. How do we know they're random?

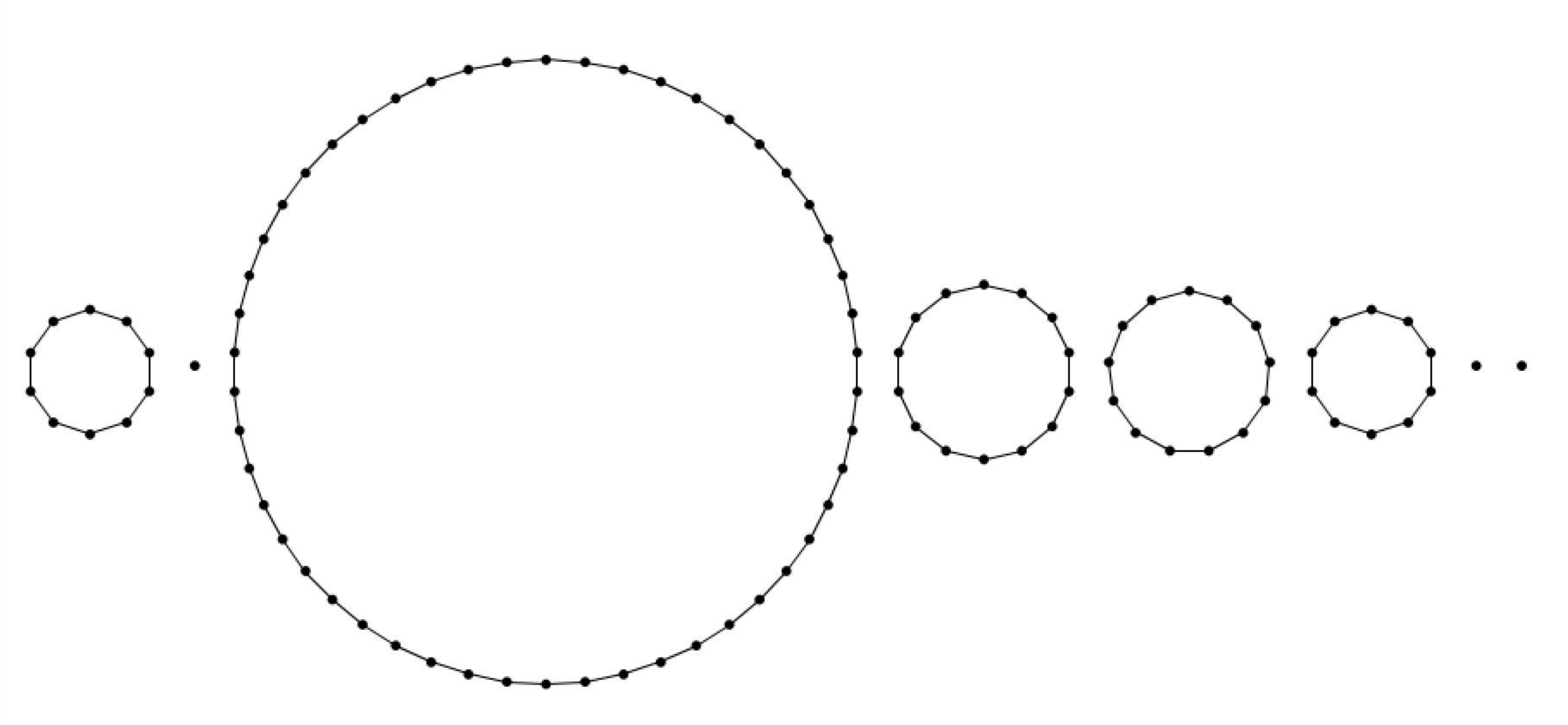
A. They're *not* random (only appear to be)!

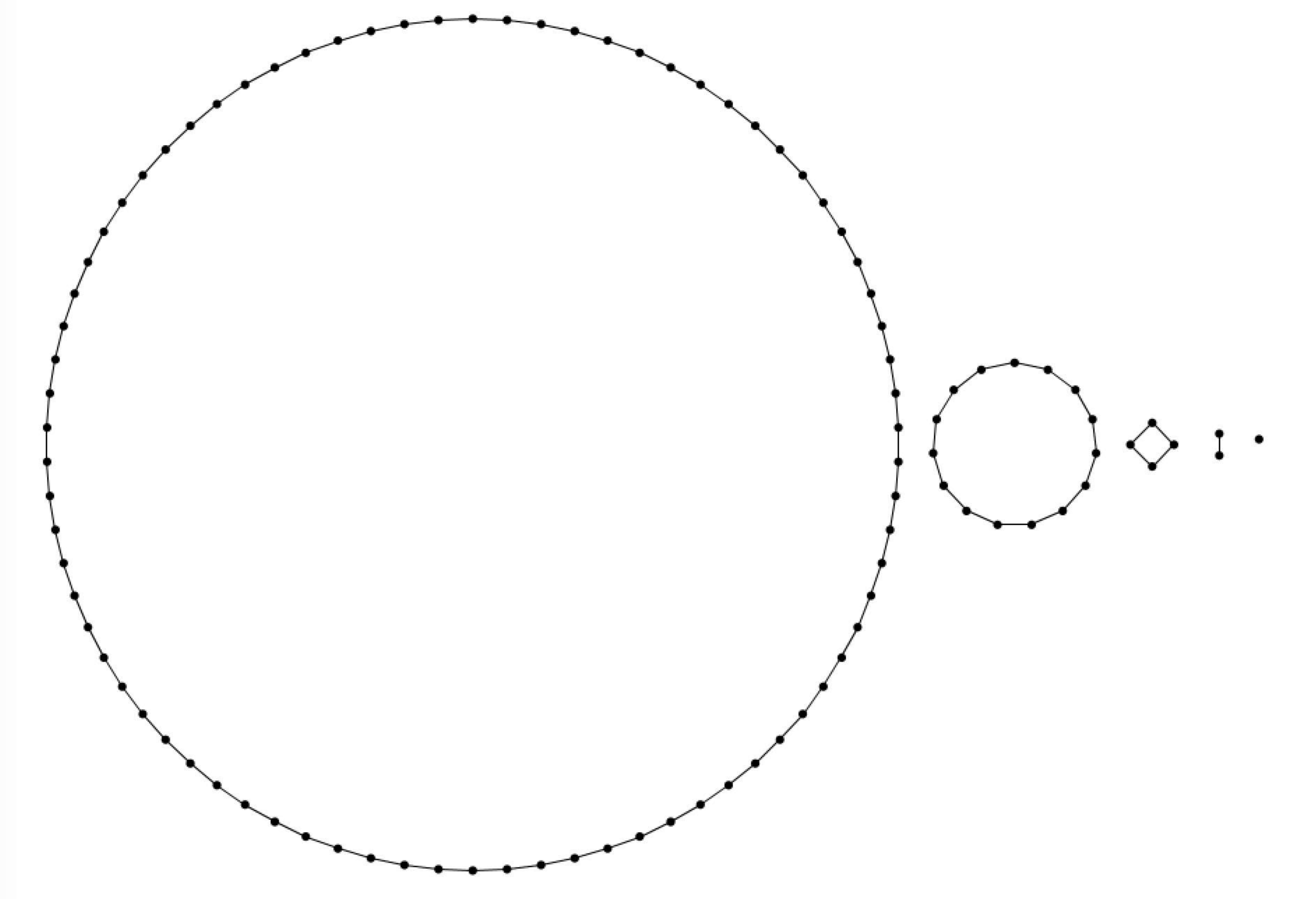
A. Need to test to see if they have the same properties as random ones.

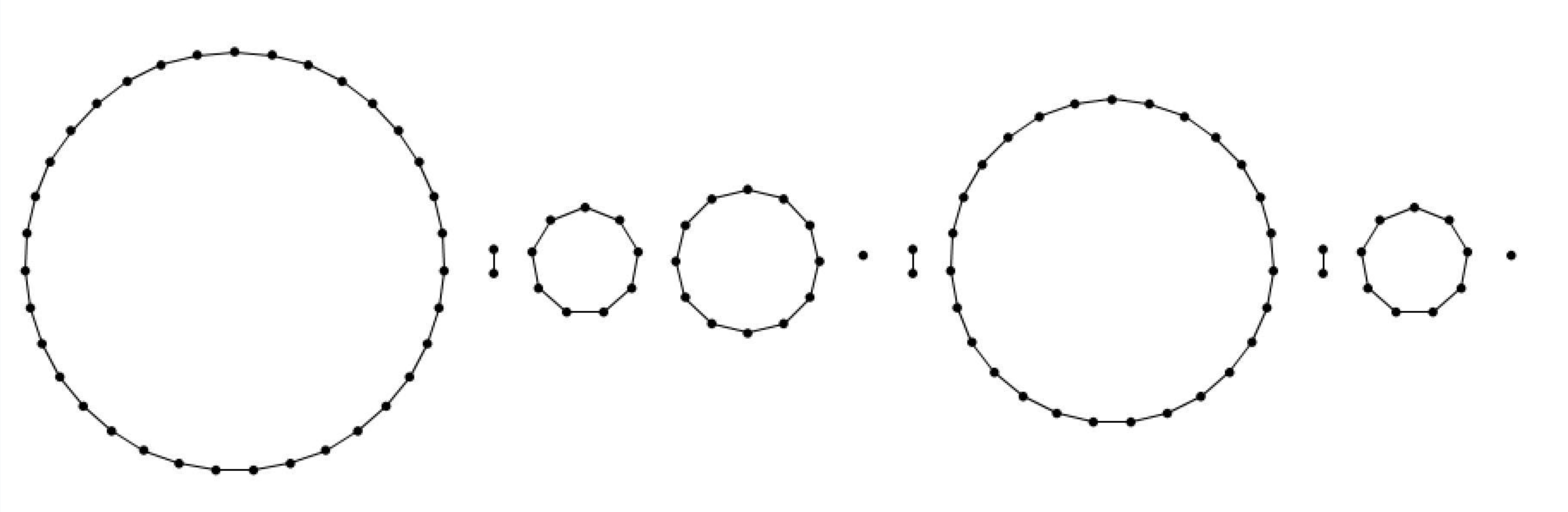
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	10	12	7	13	2	6	1	5	11	3	15	8	4	0	9



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.

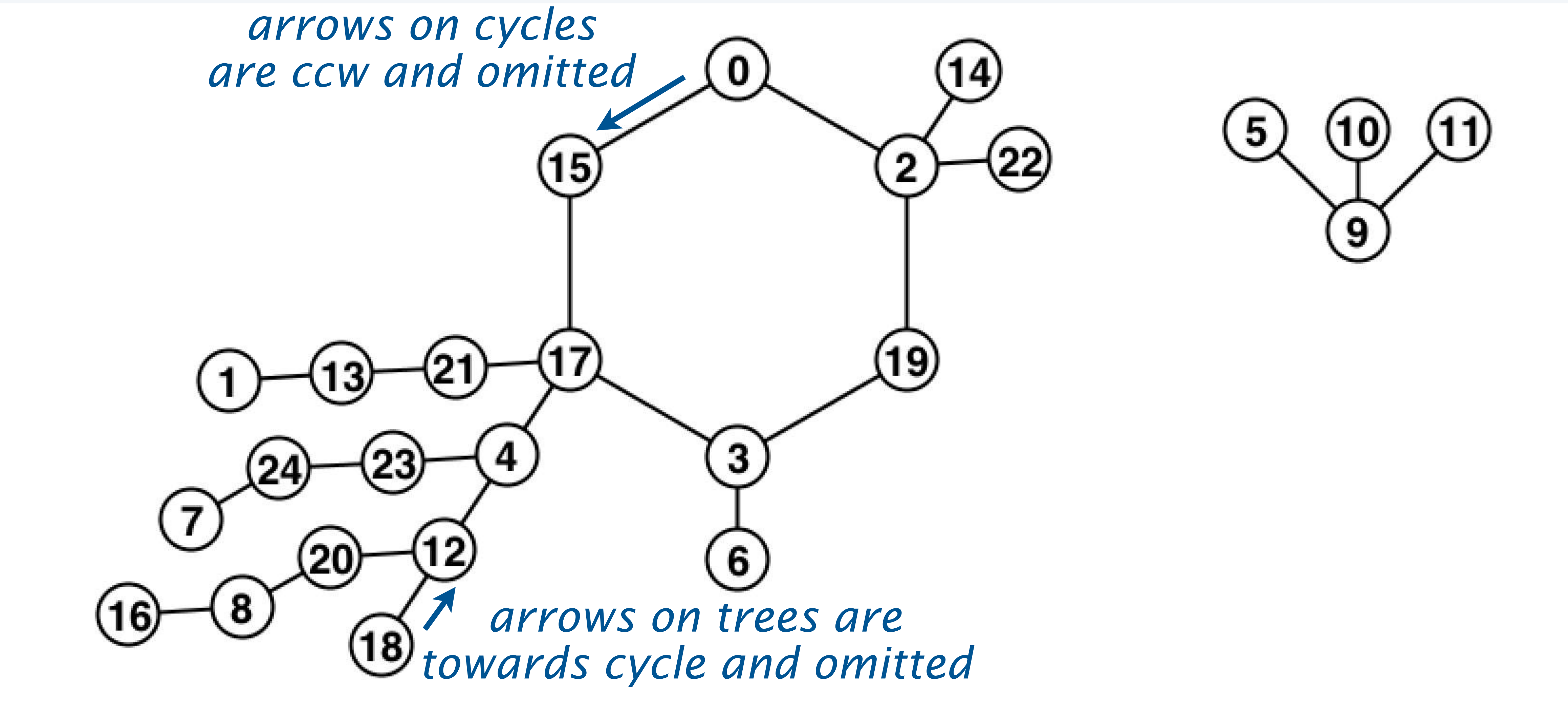
```
public class RandomMapping
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      int[] a = new int[N];
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform(N);
      for (int i = 0; i < N; i++)
          StdOut.print(a[i] + " ");
      StdOut.println();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

```
% java RandomMapping 25
15 13 0 19 17 9 3 24 20 9 9 9 4 21 2 17 8 3 12 2 12 17 2 4 23

% java RandomMapping 25
8 9 5 18 6 11 23 7 22 14 12 22 18 17 13 24 12 14 19 8 5 19 20 6 5

% java RandomMapping 25
17 14 23 9 5 21 19 23 14 10 22 2 4 8 2 15 11 4 1 4 20 17 19 19 11
```

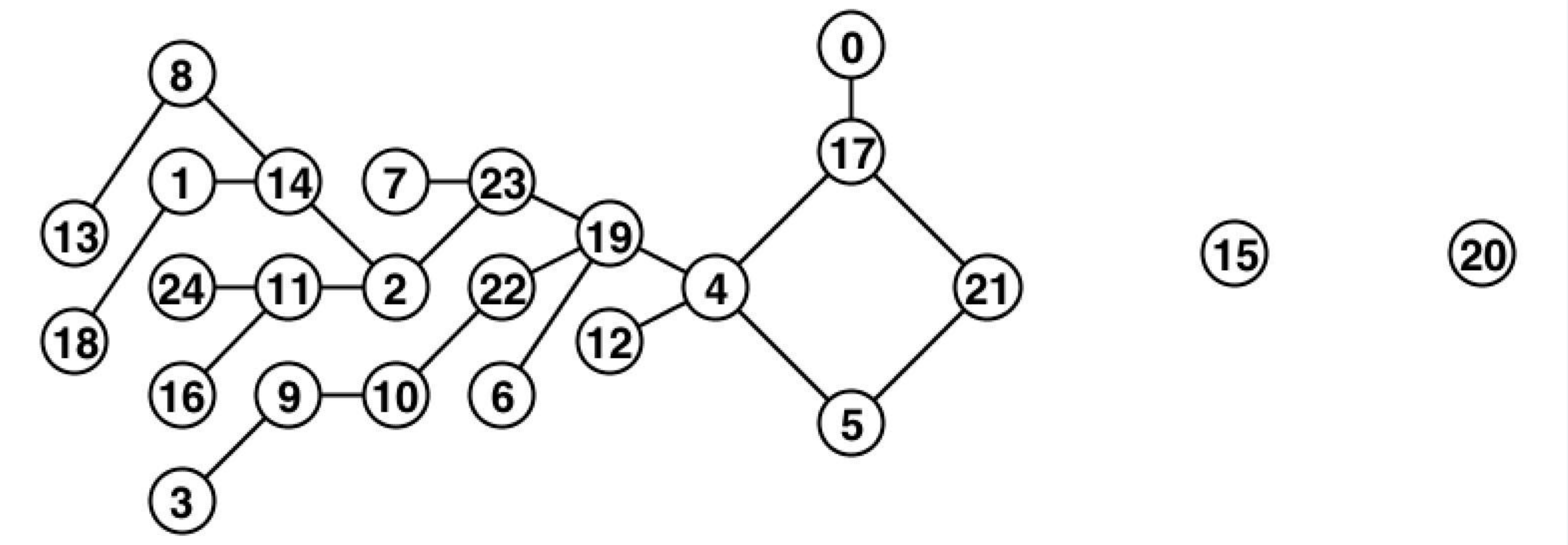
Proof of uniformity. N^N different mappings possible, all equally likely.

Three random mappings of size 25

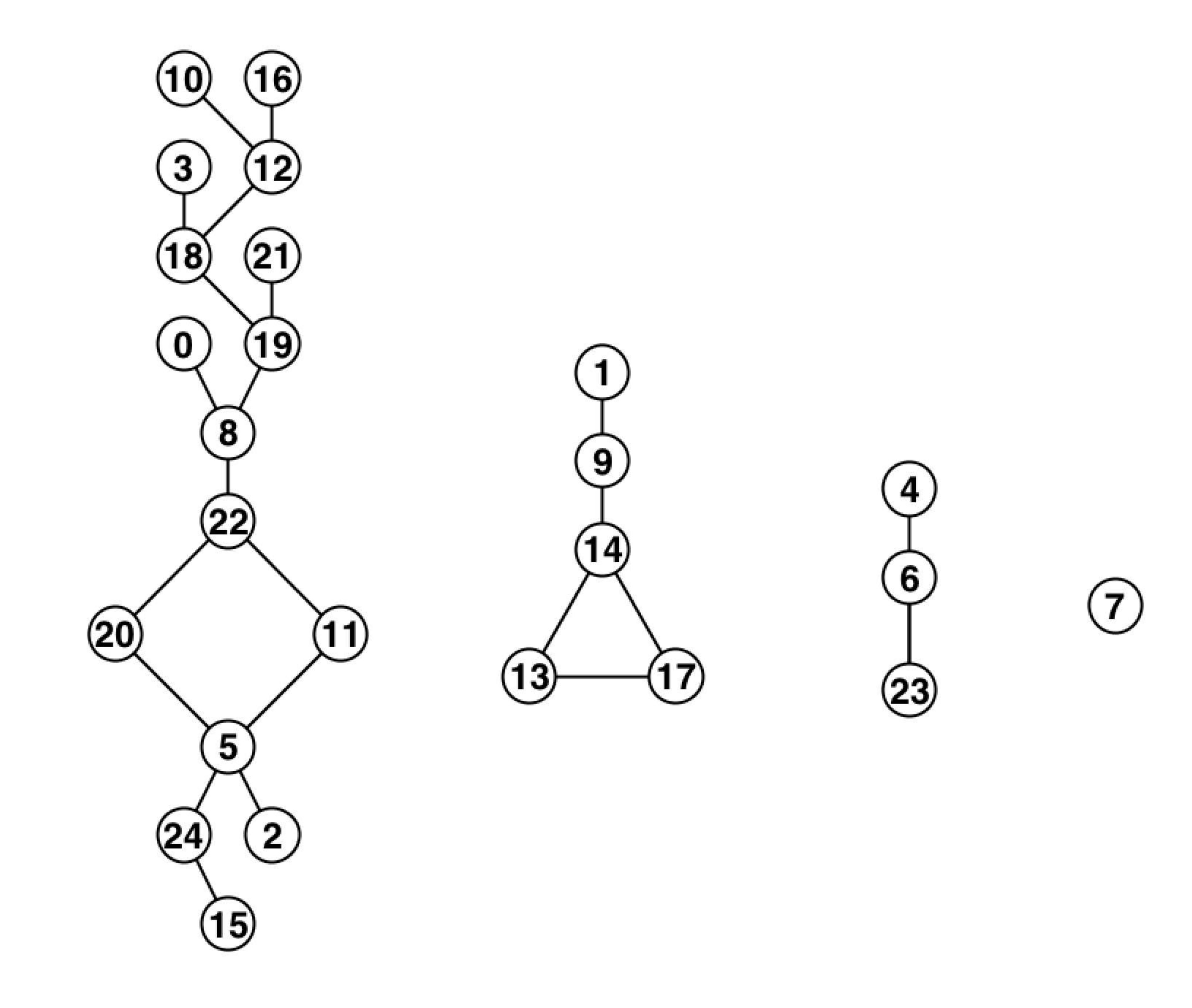
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
15	13	0	19	17	9	3	24	20	9	9	9	4	21	2	17	8	3	12	2	12	17	2	4	23



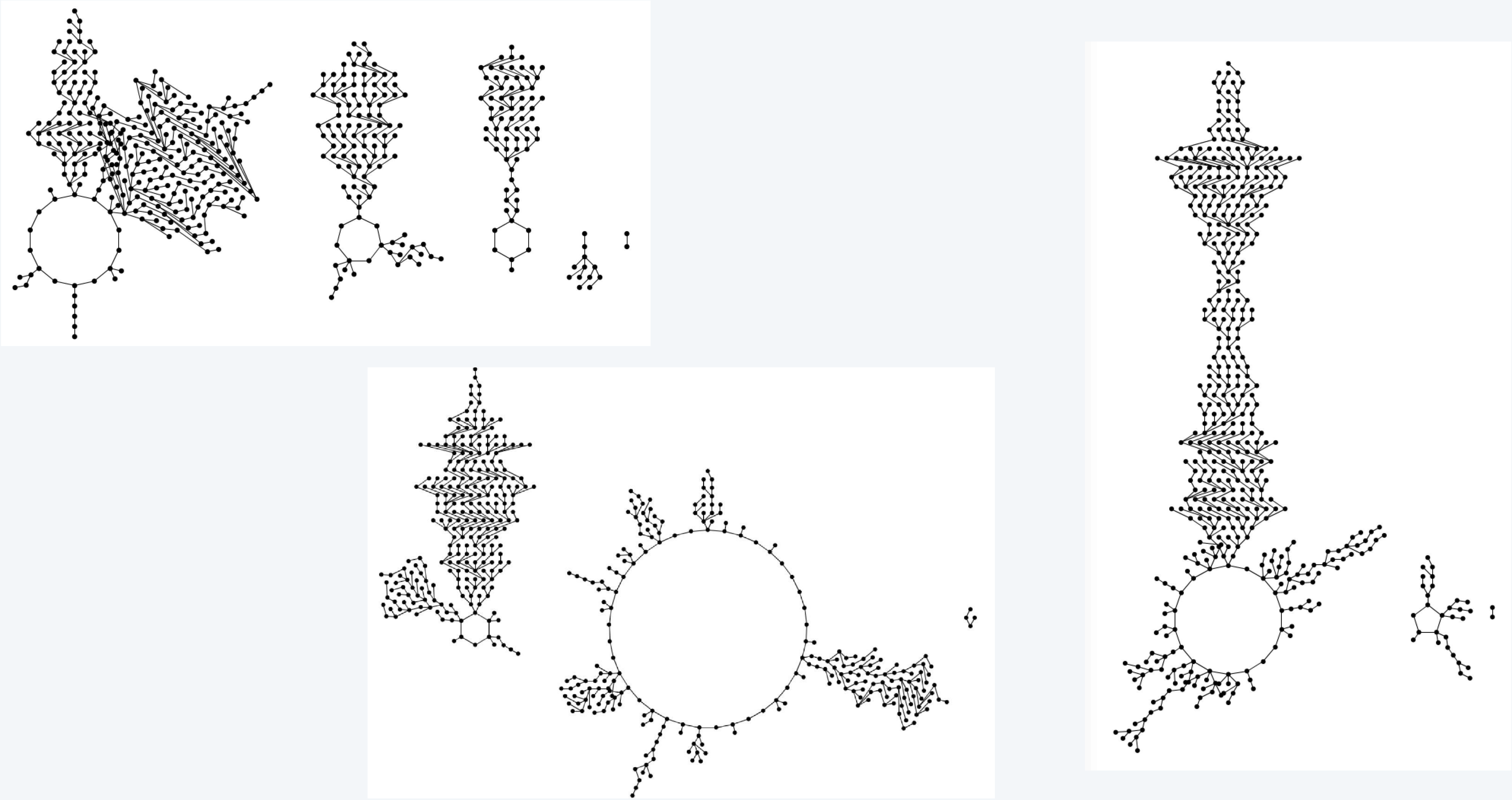
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
17	14	23	9	5	21	19	23	14	10	22	2	4	8	2	15	11	4	1	4	20	17	19	19	11



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
8	9	5	18	6	11	6	7	22	14	12	22	18	17	13	24	12	14	19	8	5	19	20	6	5



Three random mappings of size 500



Another interesting topic. Approaches to *visualizing* combinatorial structures

Properties of random mappings (for validation)

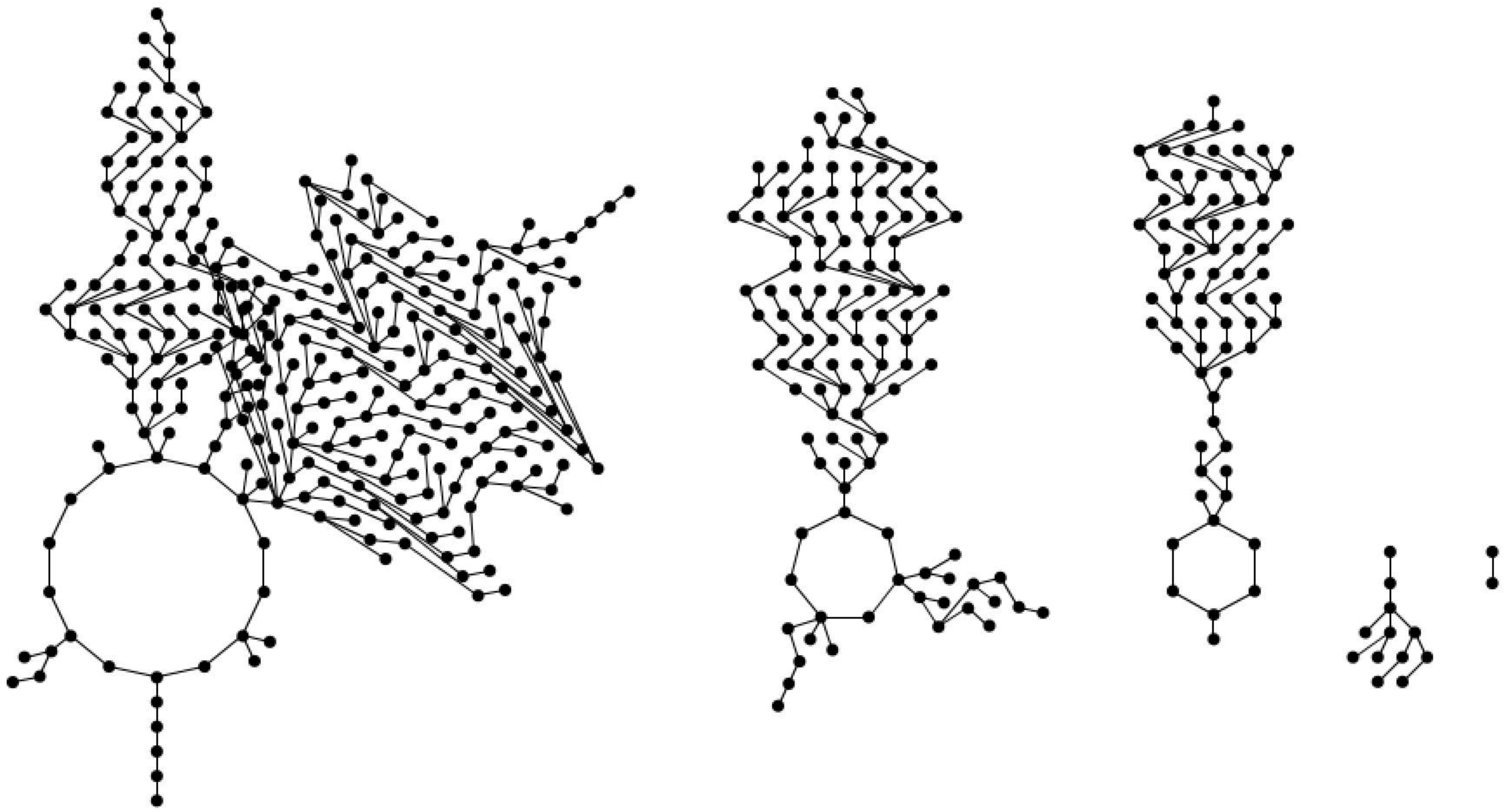
[See AofA lecture 9 and Section 7 in *Analysis of Algorithms*]

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

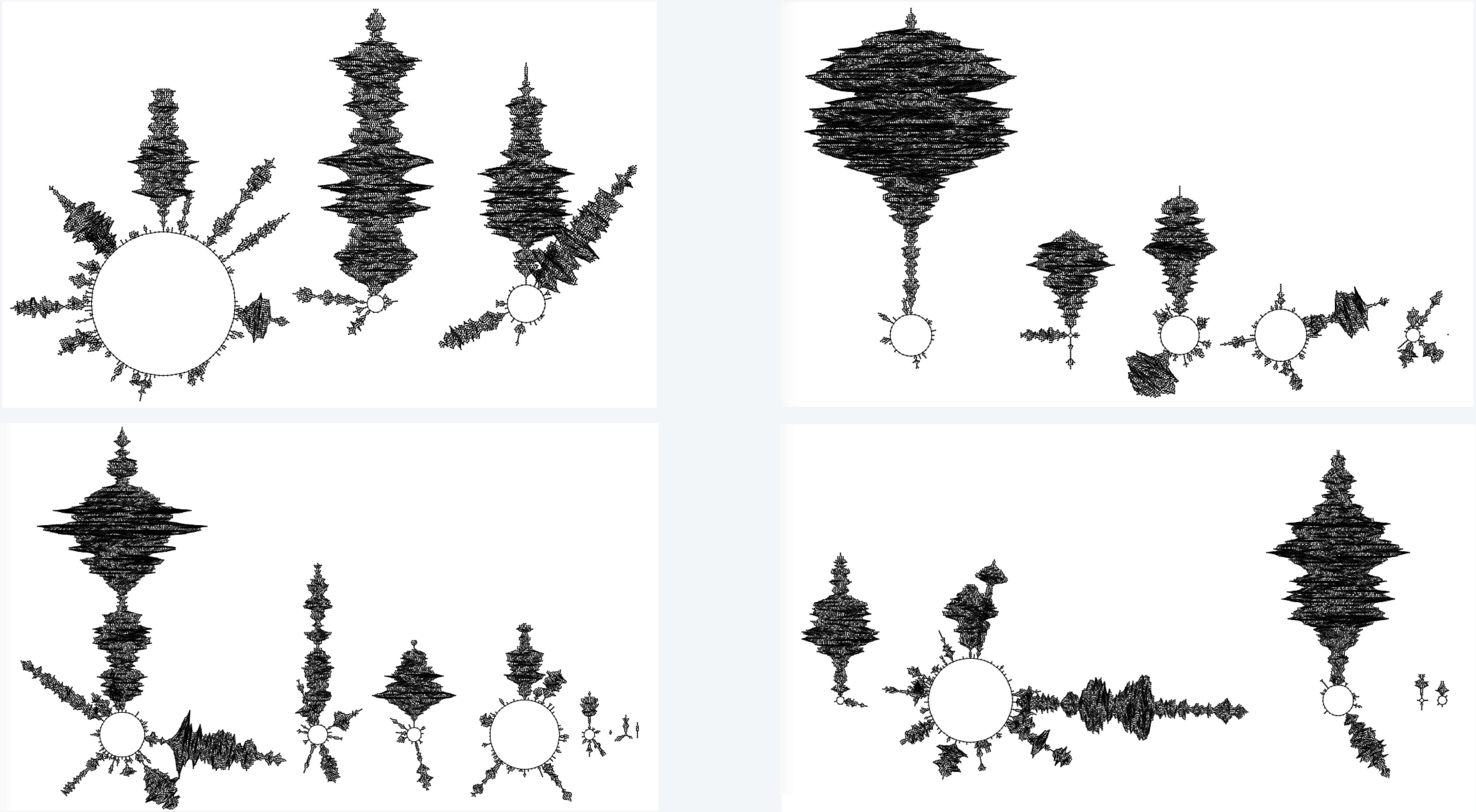
- *longest cycle* is about $0.78\sqrt{N}$
- *longest tail* is about $1.74\sqrt{N}$
- *longest rho-path* is about $2.41\sqrt{N}$
- *largest tree* is about $0.48\,N$
- *largest component* is about $0.76\,N$

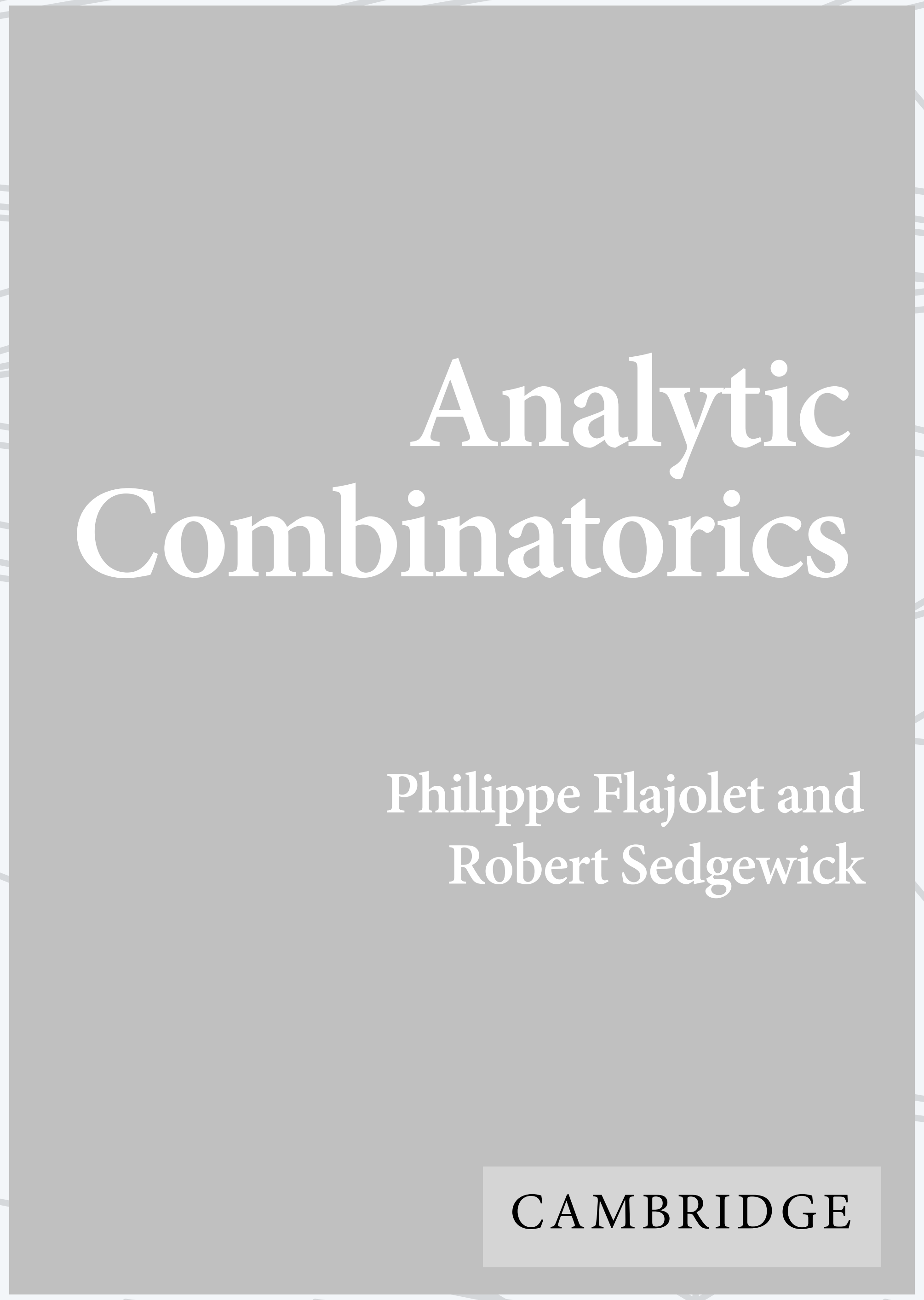


Exercise. Generate 10^6 random mappings to validate (both the analysis and the sampler!)

Note. Depends on fast generation

Four random mappings of size 10000





http://ac.cs.princeton.edu

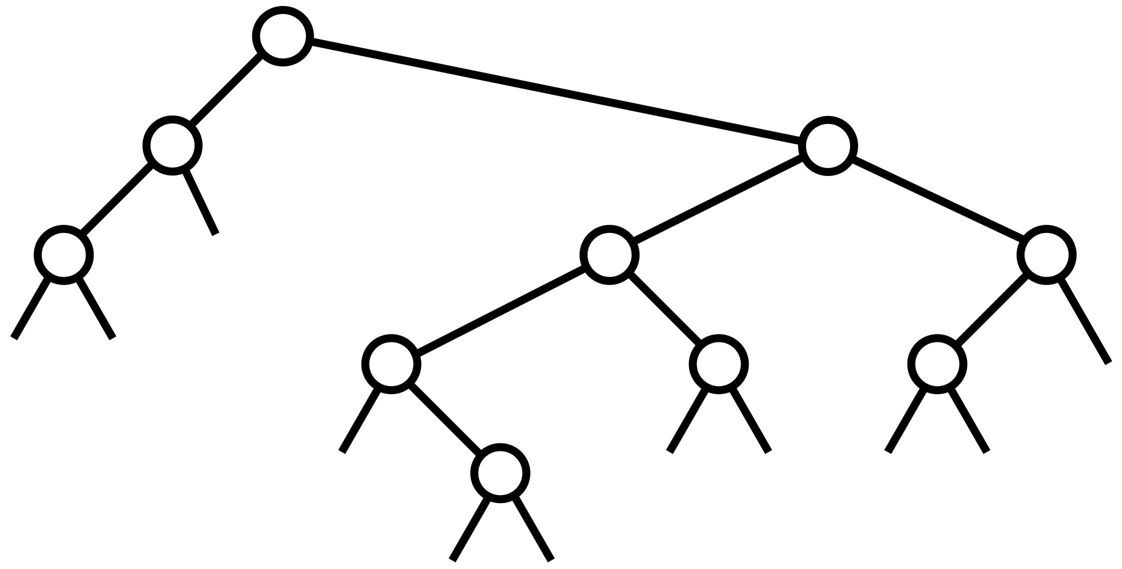
Uniform Sampling of Combinatorial Objects

- 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*

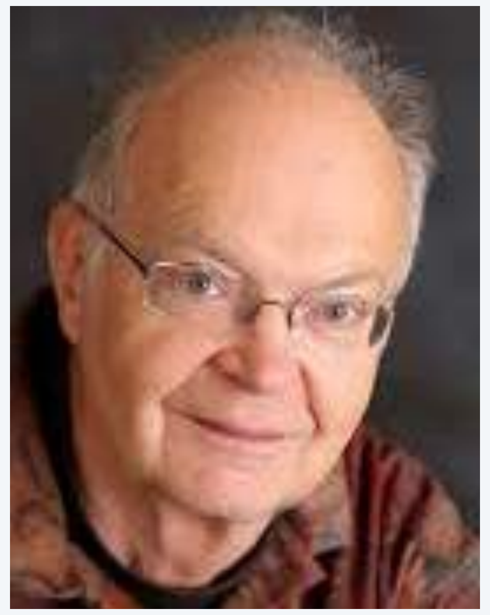
class	*typical random object* ($N = 10$)	*probability*
bitstring	1100101101	$1/2^N$
permutation	9572301486	$1/N!$
mapping	4938375038	$1/N^N$
binary tree		$\dfrac{N+1}{\binom{2N}{N}}$

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

Achieving uniformity is *not* to be taken for granted

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

– Donald E. Knuth



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".

favored position

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

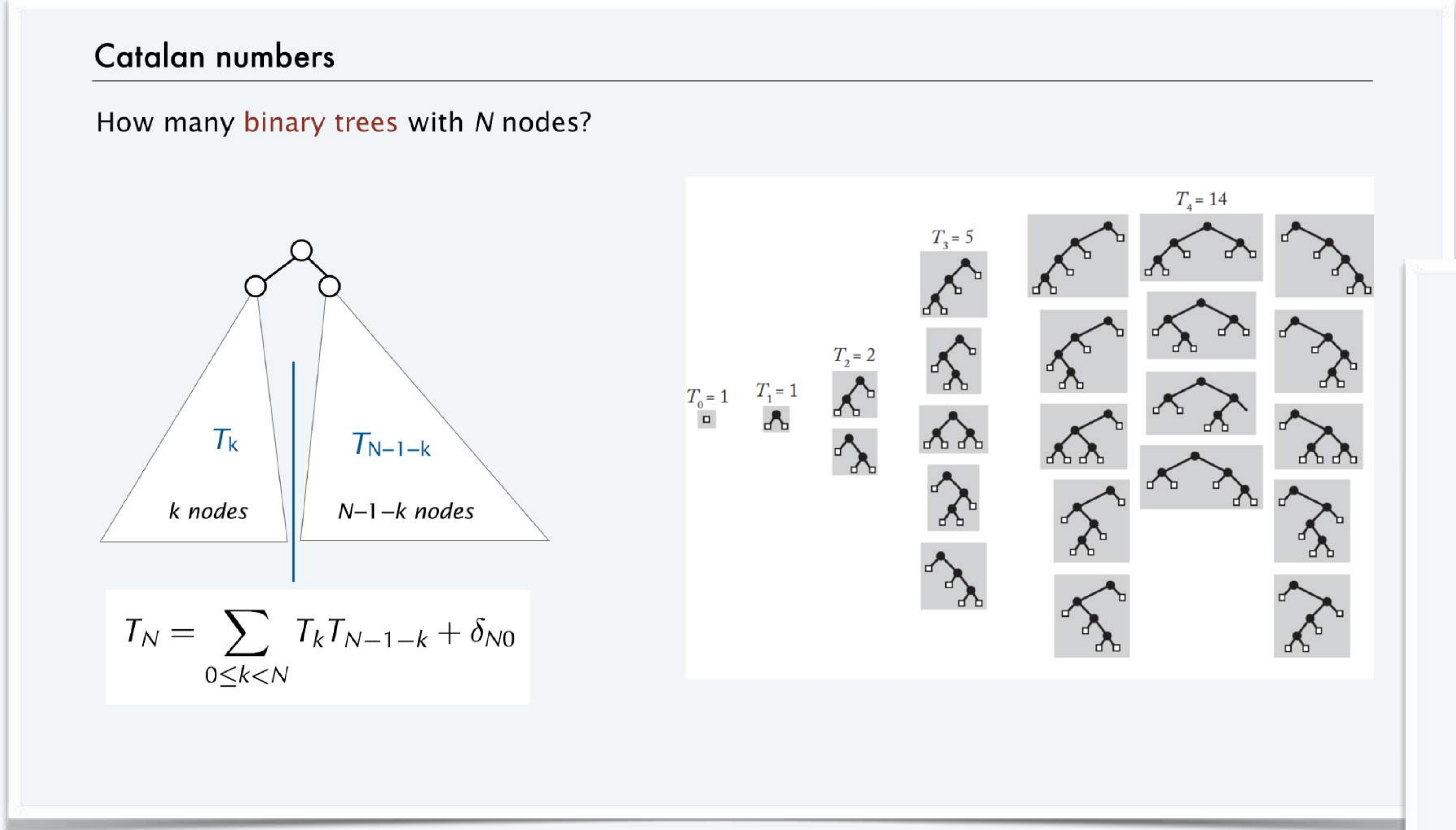
position	*IE*	*Firefox*	*Opera*	*Chrome*	*Safari*
1	1304	2099	2132	2595	1870
2	1325	2161	2036	2565	1913
3	1105	2244	1374	3679	1598
4	1232	2248	1916	590	4014
5	**5034**	1248	2542	571	605

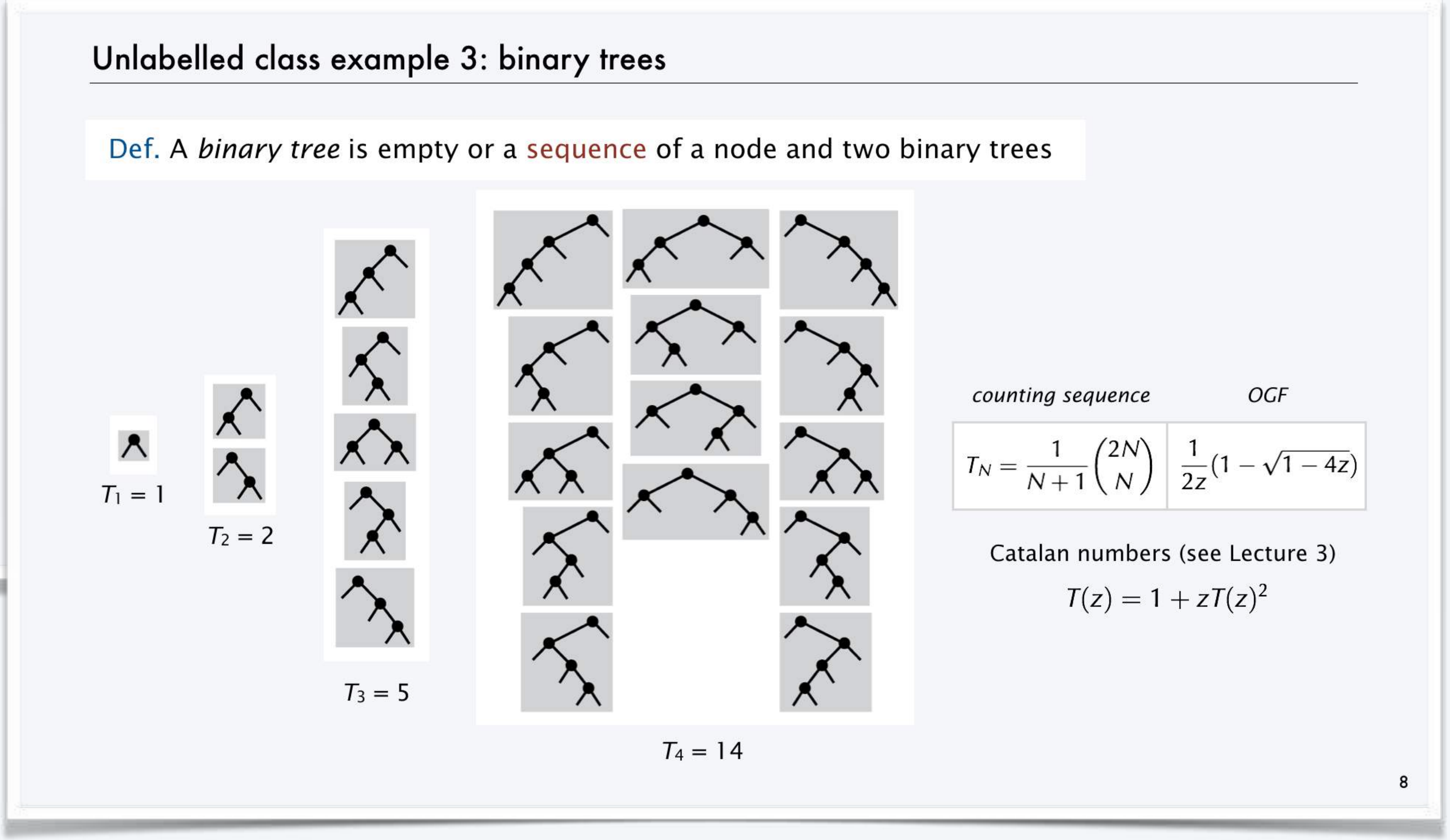
https://www.robweir.com/blog/2010/02

Binary trees

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

AofA lecture 3

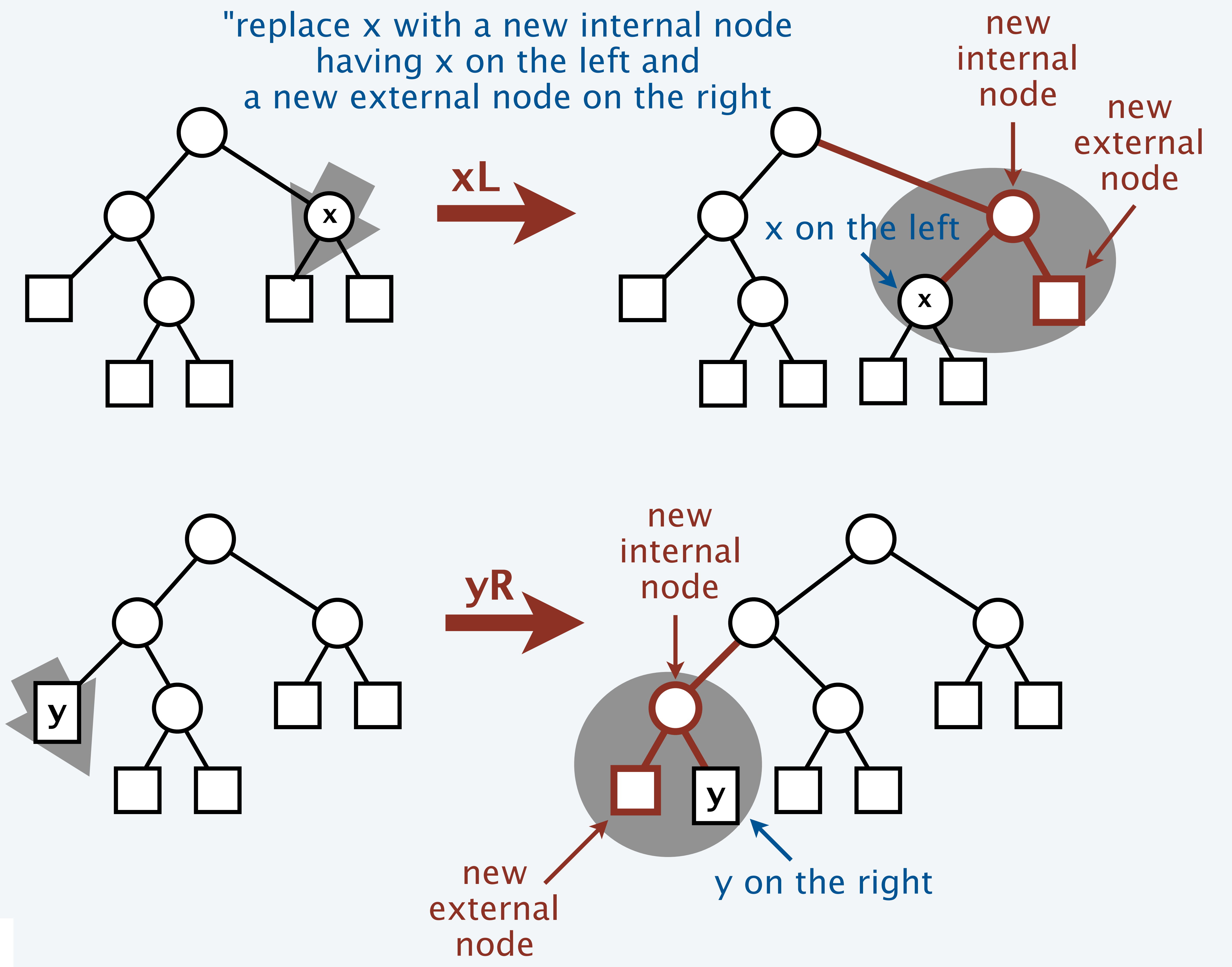




AofA lecture 5

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

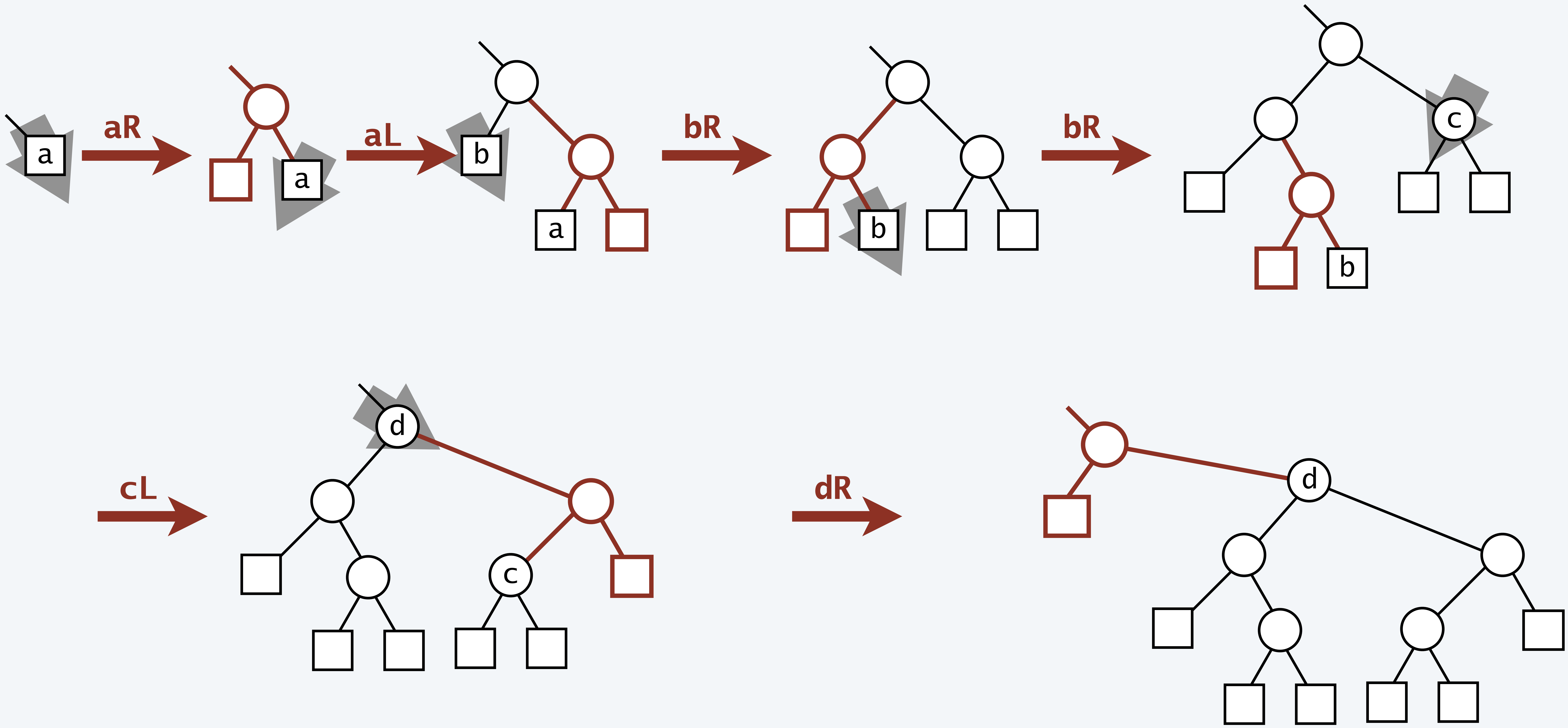
- Choose a node x (internal or external) at random.
- Choose an orientation (L or R) at random.
- Replace x with a new internal node having x as one child (as per orientation) and a new external node as the other.

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

$$\underbrace{(N+1)}_{\text{each appears } N+1 \text{ times}} \times \overbrace{T_N}^{\text{total number of trees with } N \text{ internal nodes}} = \overbrace{2}^{\text{L or R}} \times \overbrace{(2N-1)}^{N-1 \text{ internal, } N \text{ external}} \times \overbrace{T_{N-1}}^{\text{total number of trees with } N-1 \text{ internal nodes}}$$

- Therefore

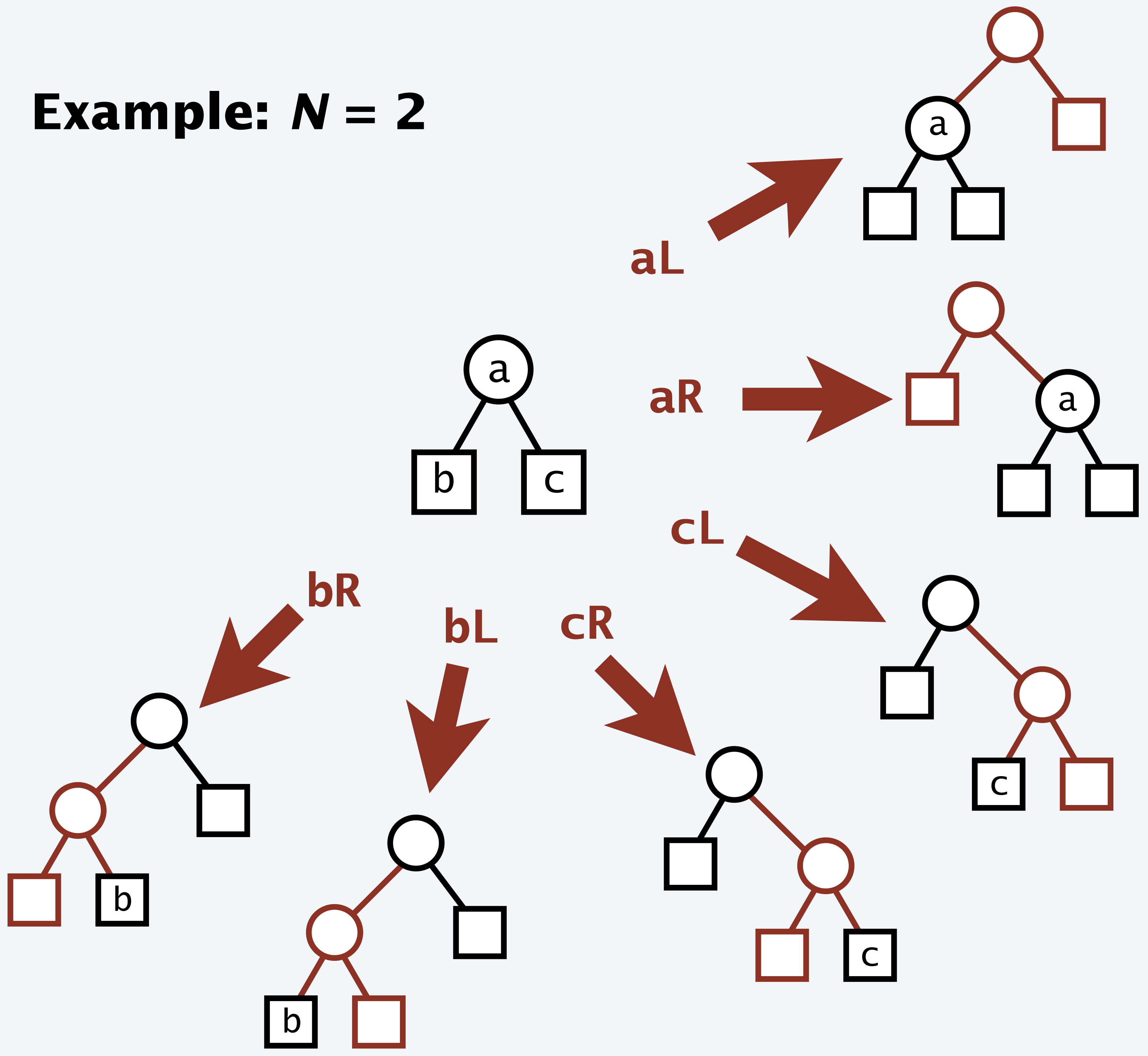
$$T_N = \frac{(2N)(2N-1)}{(N+1)N} \times T_{N-1}$$

$$= \frac{(2N)!}{(N+1)!N!} \quad \longleftarrow \text{telescope the recurrence}$$

$$= \frac{1}{N+1}\binom{2N}{N}$$

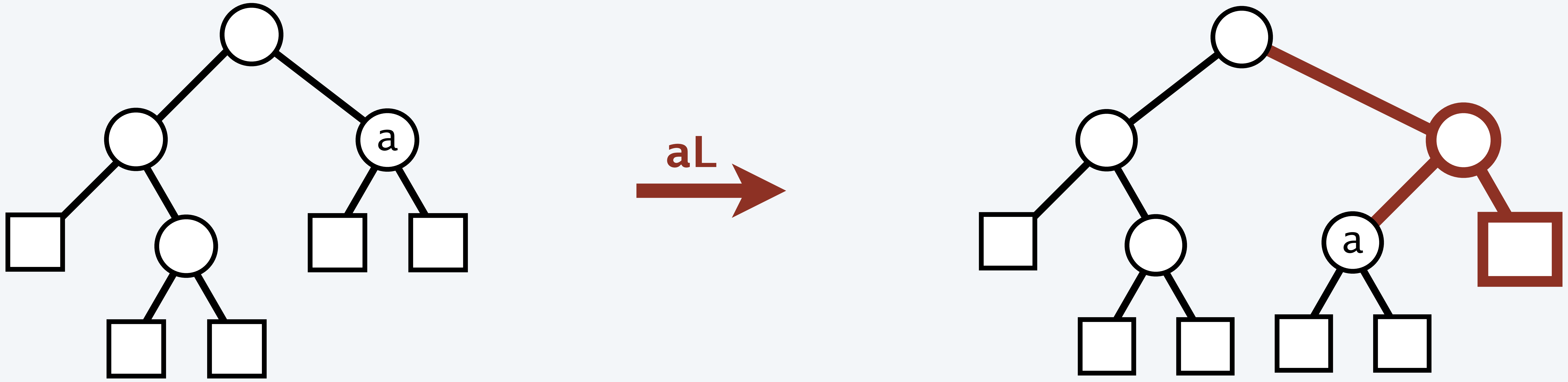
- Which implies that each binary tree of size N is equally likely.

Example: N = 2



Rémy's algorithm: implementation

Straightforward implementation can be complicated (try it!)



Complications

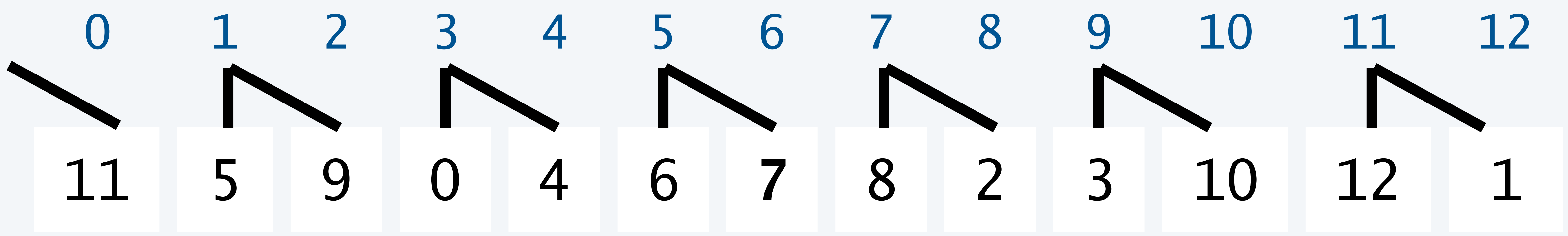
- Need explicit external nodes.
- Need array of node pointers to choose random node.
- Need "parent" links, which are notoriously complicated to maintain.
- Each iteration creates two nodes and changes three links (not counting parent links).

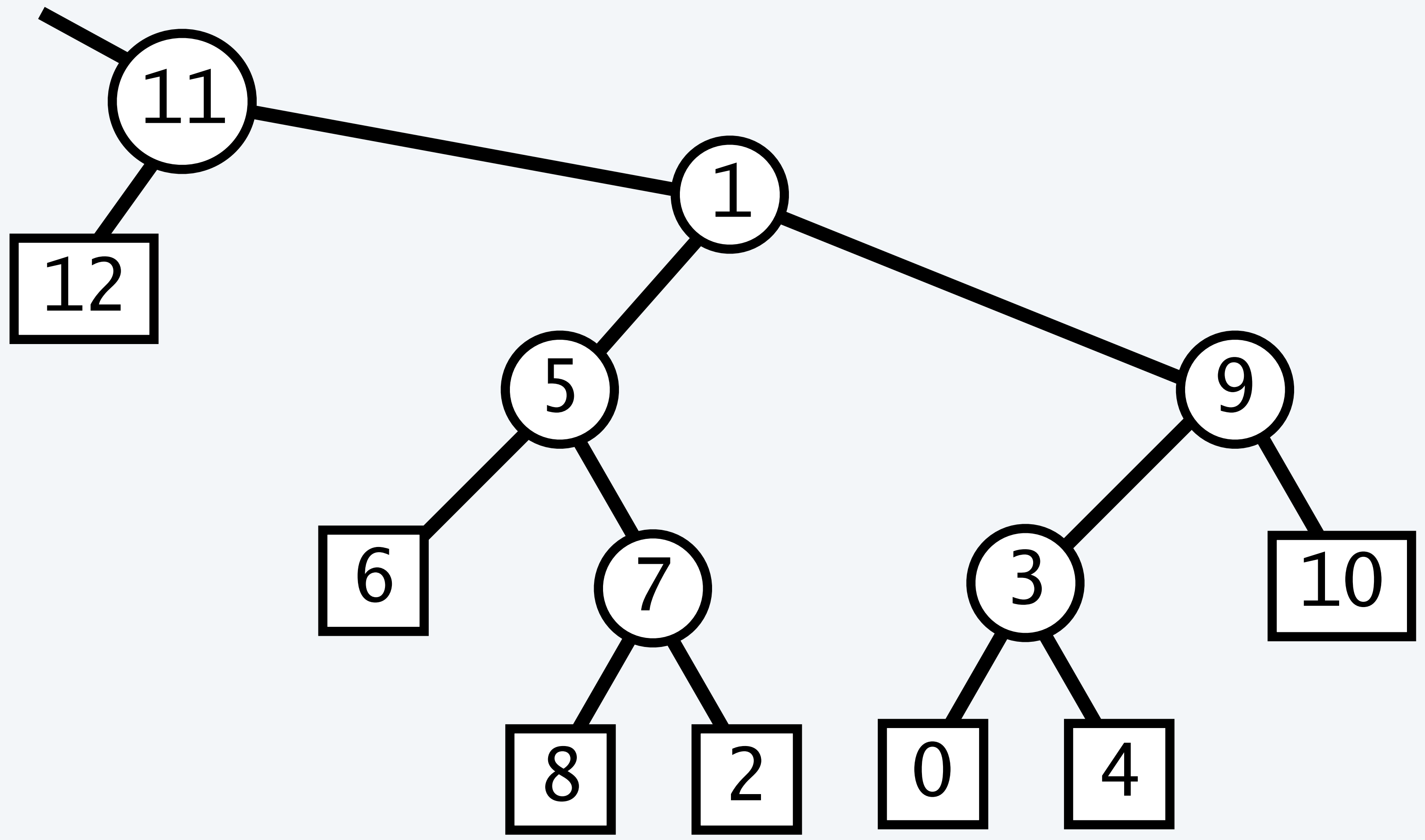
bottom line: you can find some ugly code in the literature

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

```
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]];
}
```

create external nodes

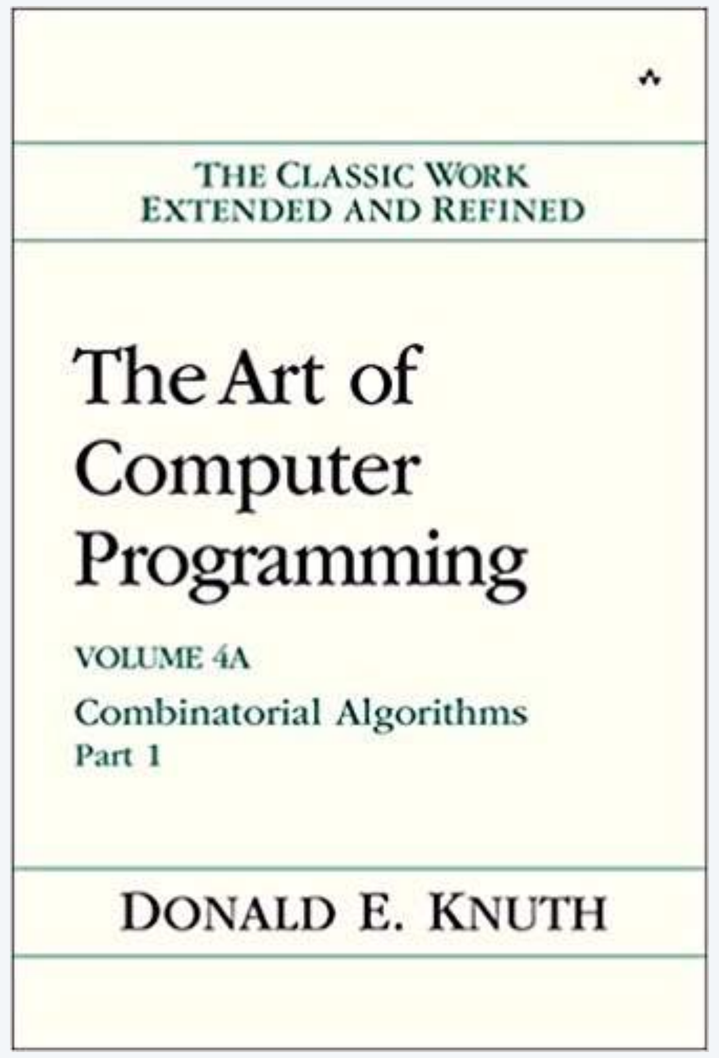
create internal nodes

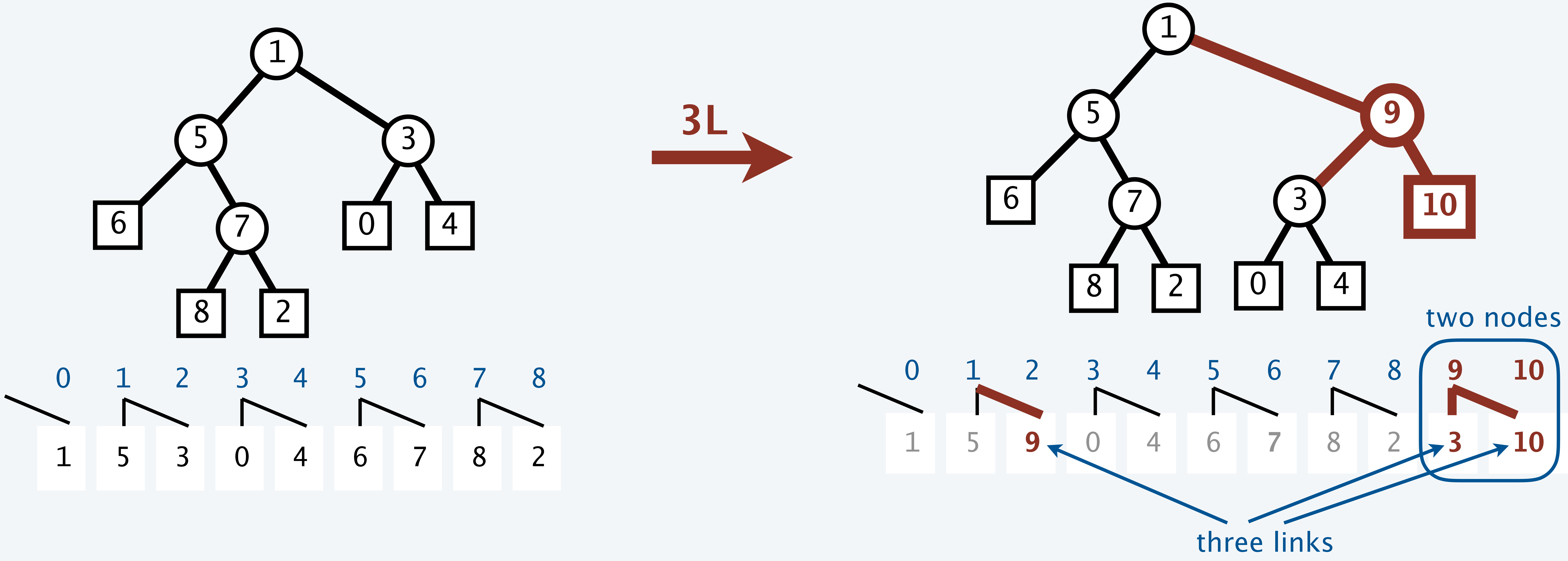
fill in links in internal nodes

Knuth's implementation of Rémy's algorithm

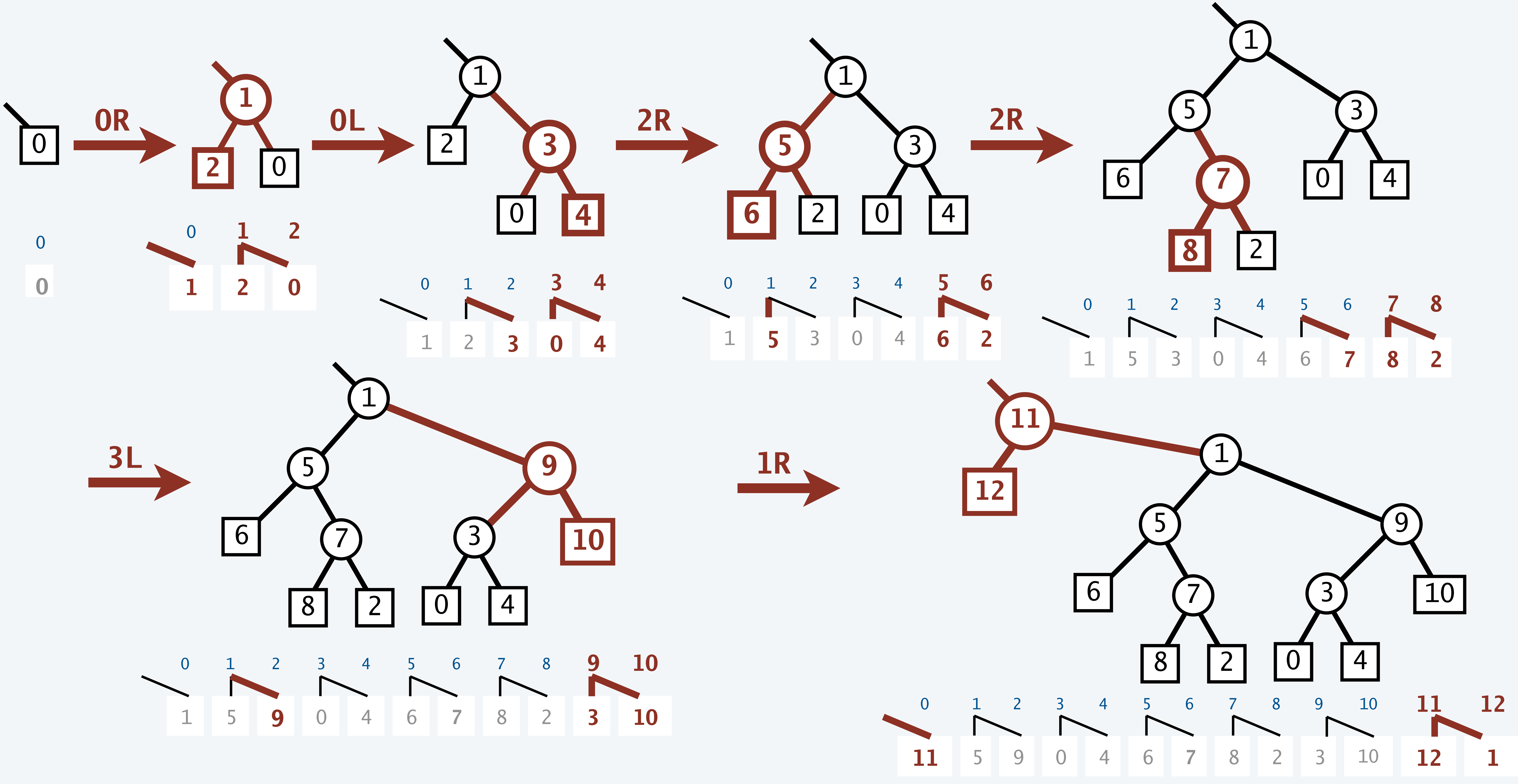
```
int[] links = new int[2*N + 1];
for (int k = 1; k < 2*N; k+=2)
{
   int x = StdRandom.uniform(k);
   if (StdRandom.bernoulli(.5))
      { links[k] = k+1; links[k+1] = links[x]; }
   else
      { links[k] = links[x]; links[k+1] = k+1; }
   links[x] = k;
}
```

"Then the program is short and sweet"





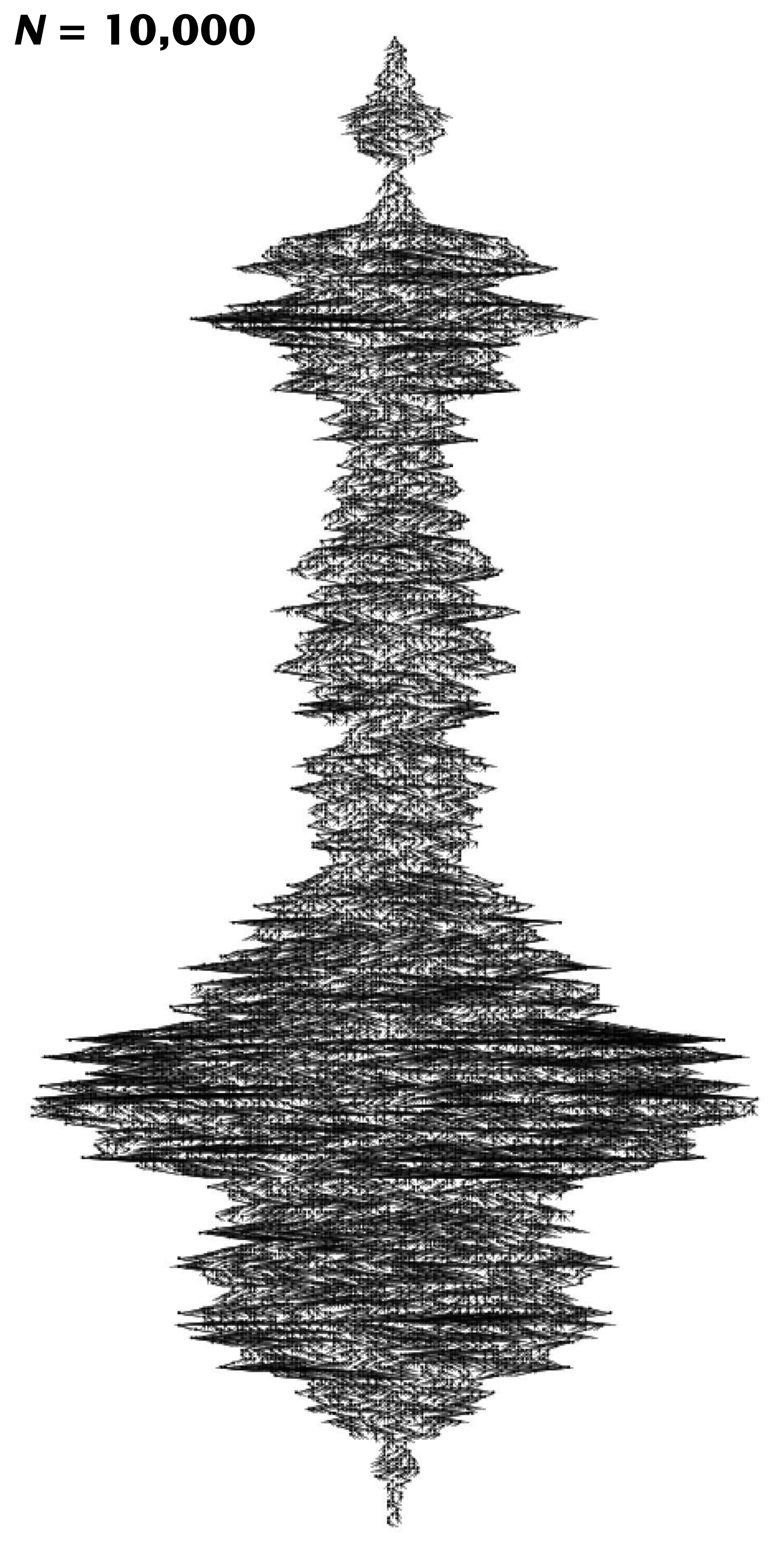
Knuth's implementation of Rémy's algorithm (example)



Rémy's algorithm

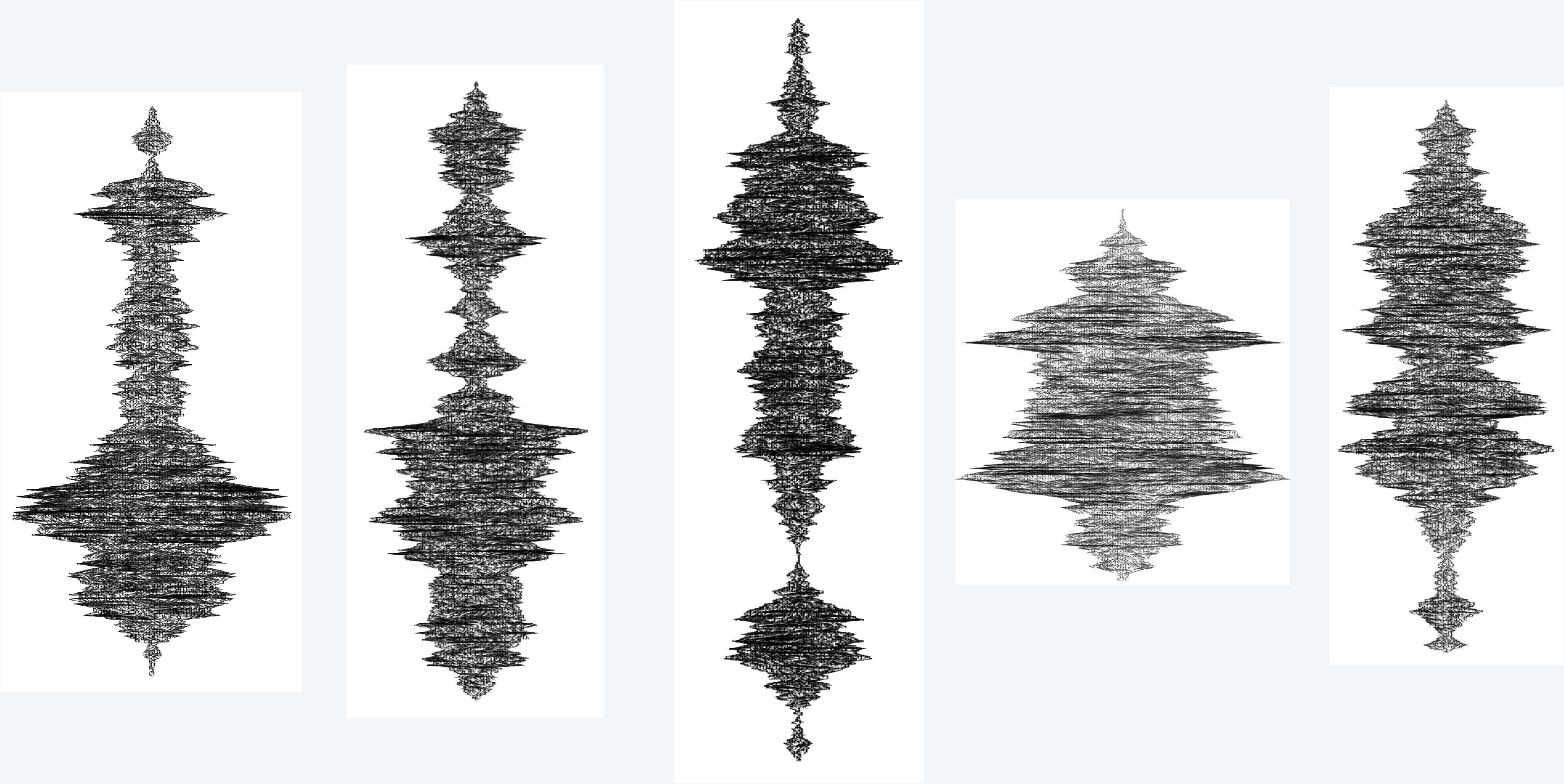
Generate a random binary tree with *N* nodes.

```
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.

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

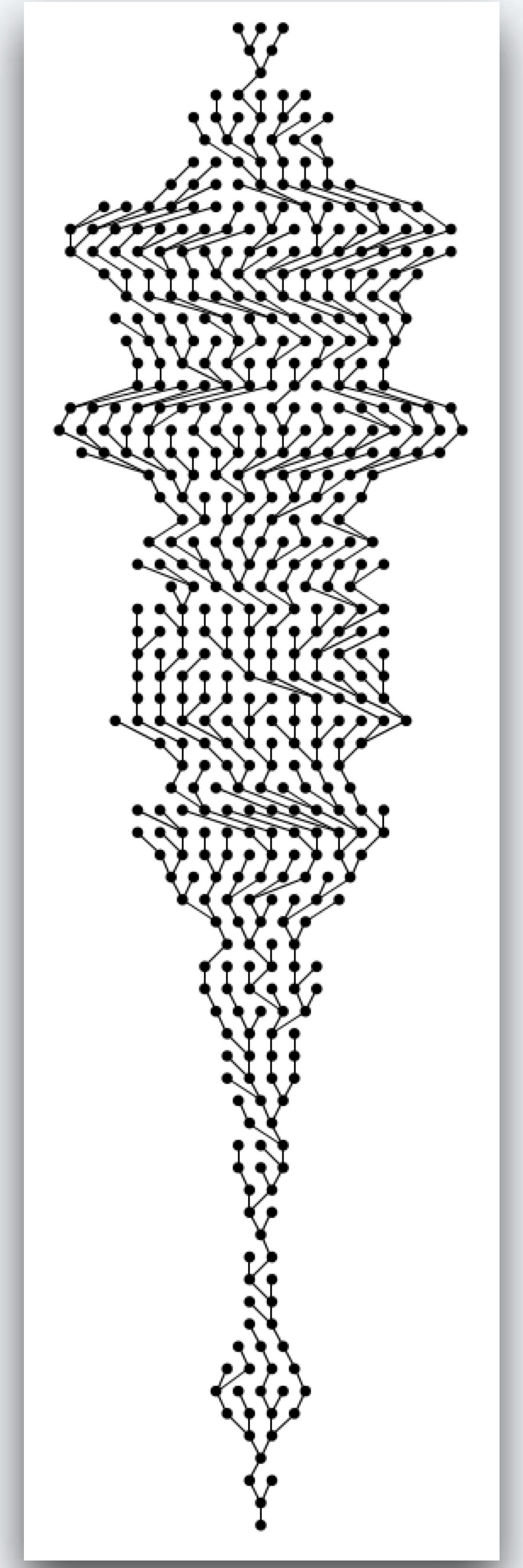
Problem. Our samplers so far are *specialized* and do not extend to more complicated situations.

Examples.

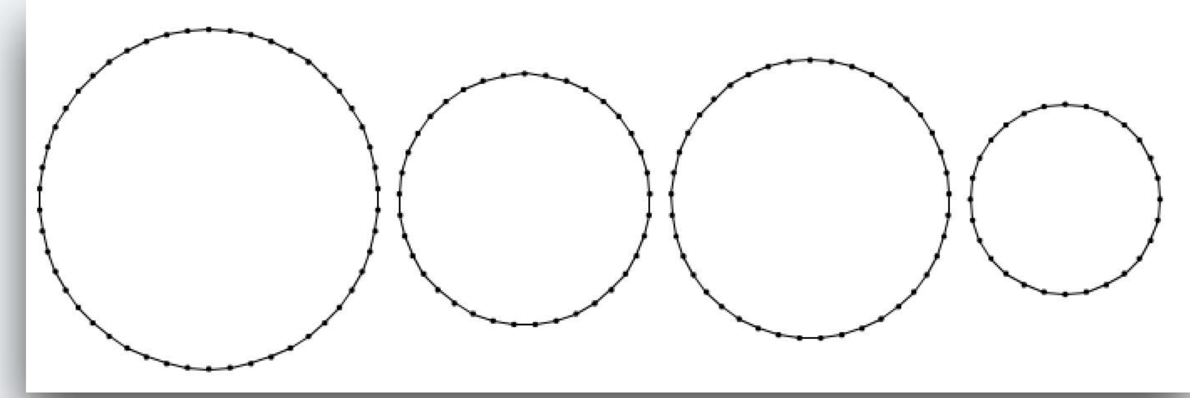
- bitstrings with forbidden patterns
- generalized derangements and involutions
- trees of all sorts
- restricted mappings
- ...

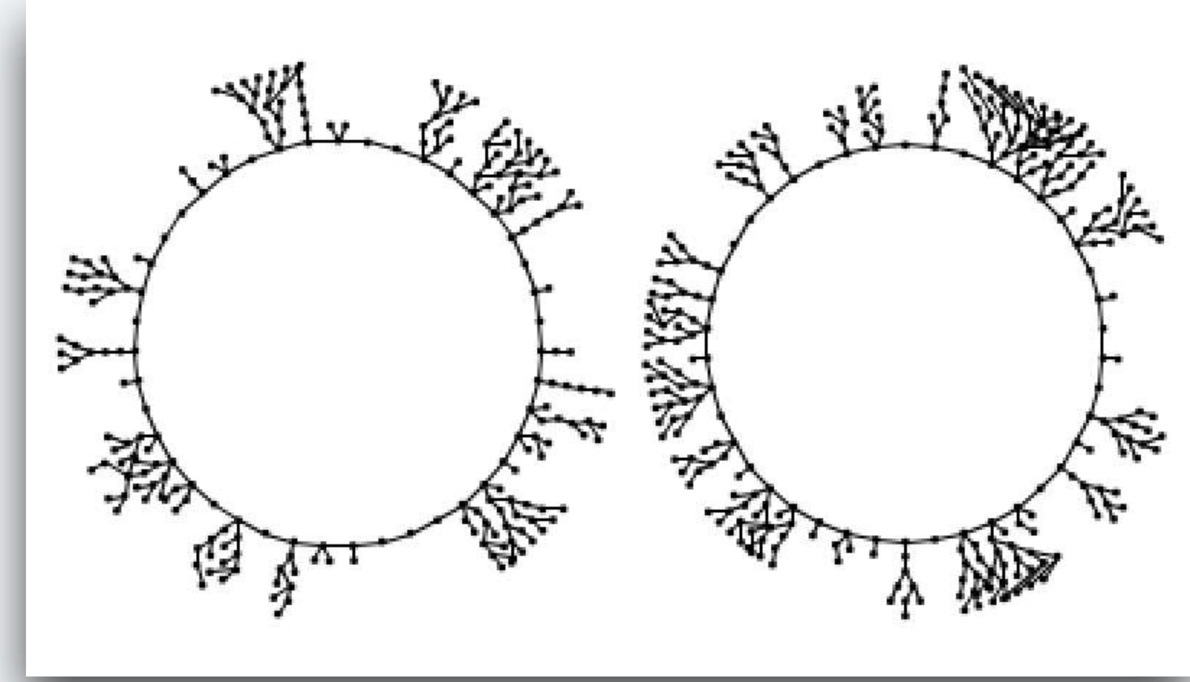
Fundamental challenge. Develop methods that

- apply to a broad variety of classes *and*
- are provably uniform *and*
- admit efficient implementations

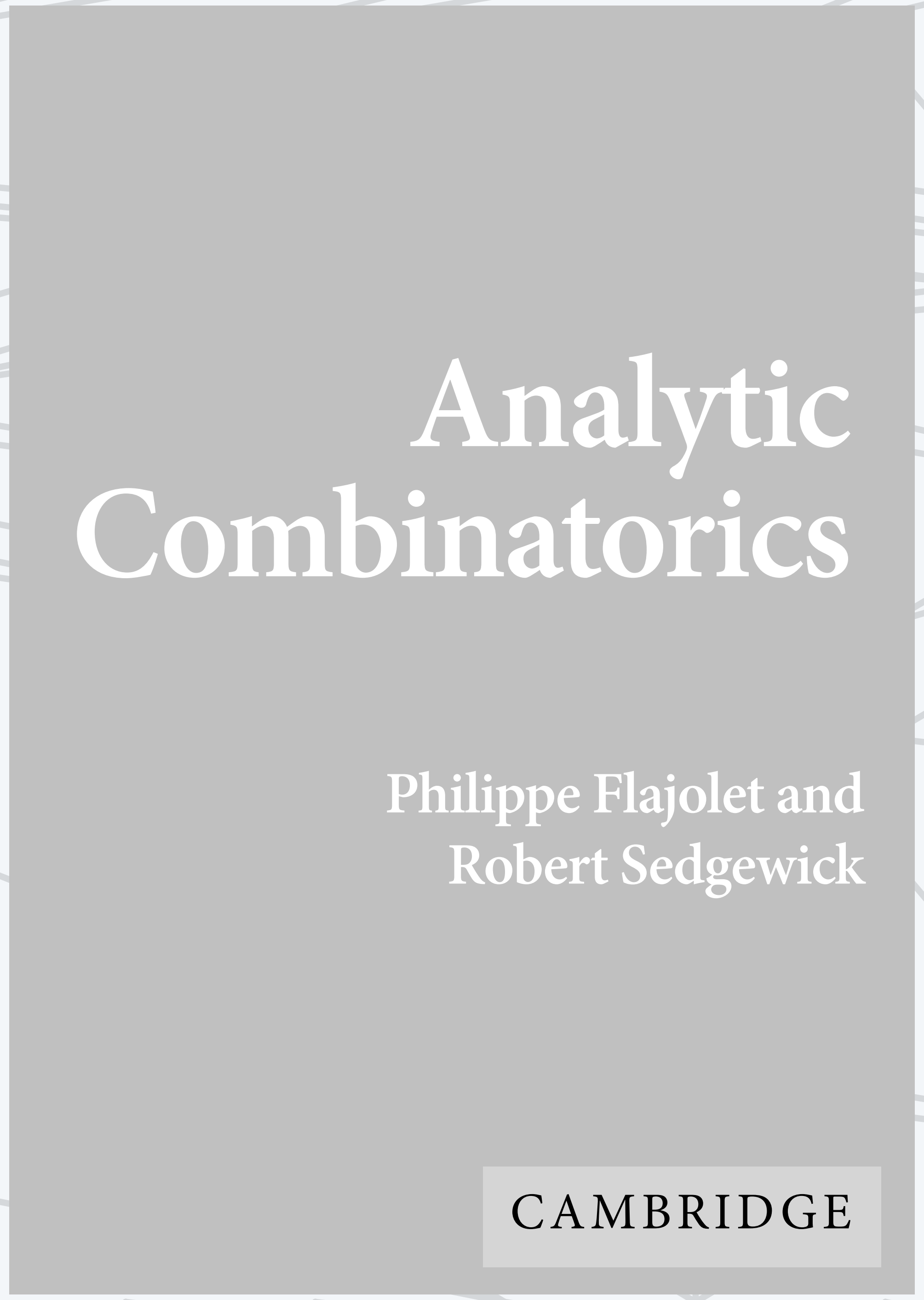


```
1001101000100010111110100111001 0
1011111001100011101110110111101 1
0011010001110111010101010110110 1
1101110100101100011111111011111 0
1001100100111010111000100100100 1
0011001100011000111111000110100 1
```





Ultimate goal. Generate a sampler for a combinatorial class *automatically* from its specification.



http://ac.cs.princeton.edu

Random Generation of Combinatorial Objects

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

Rejection

First technique to consider: *rejection*

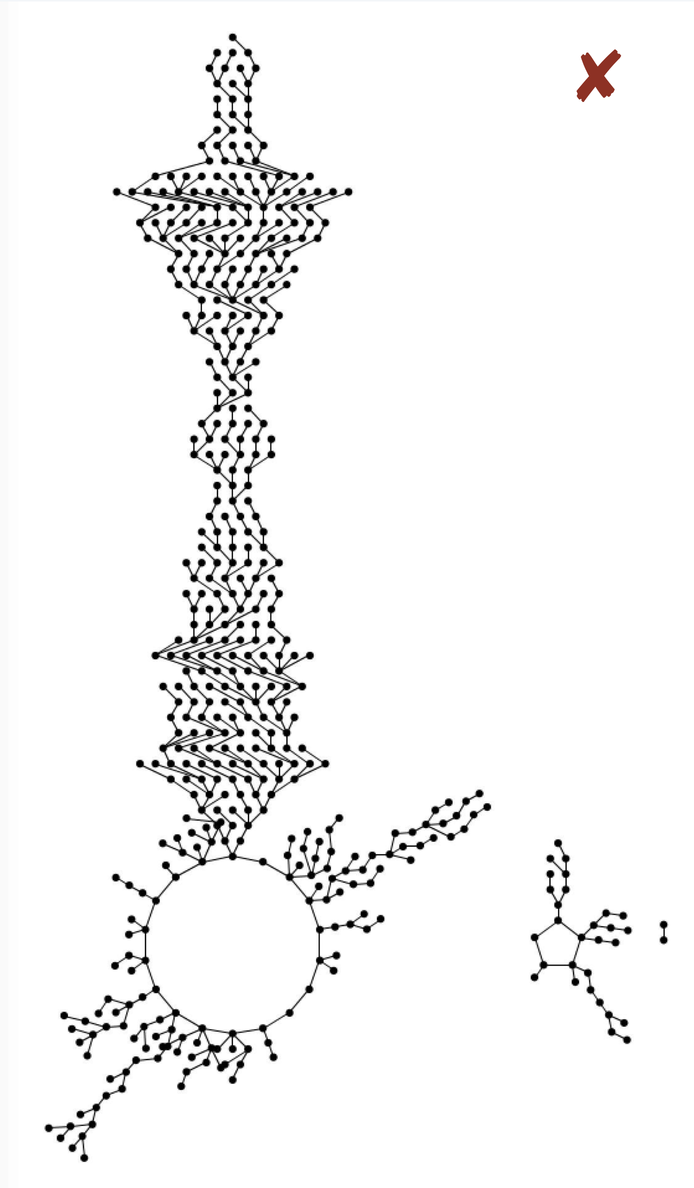
- Generate a random object
- Reject it if it does not have a specified property
- Continue until finding one that *does* have the property

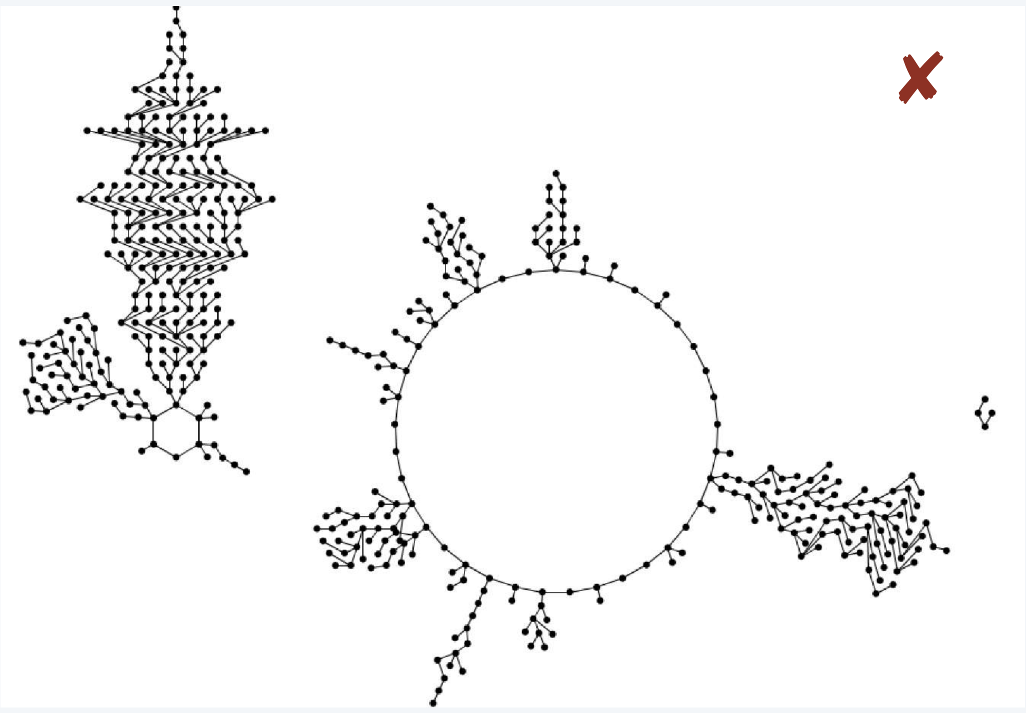
see Section II.3 for mathematical foundations

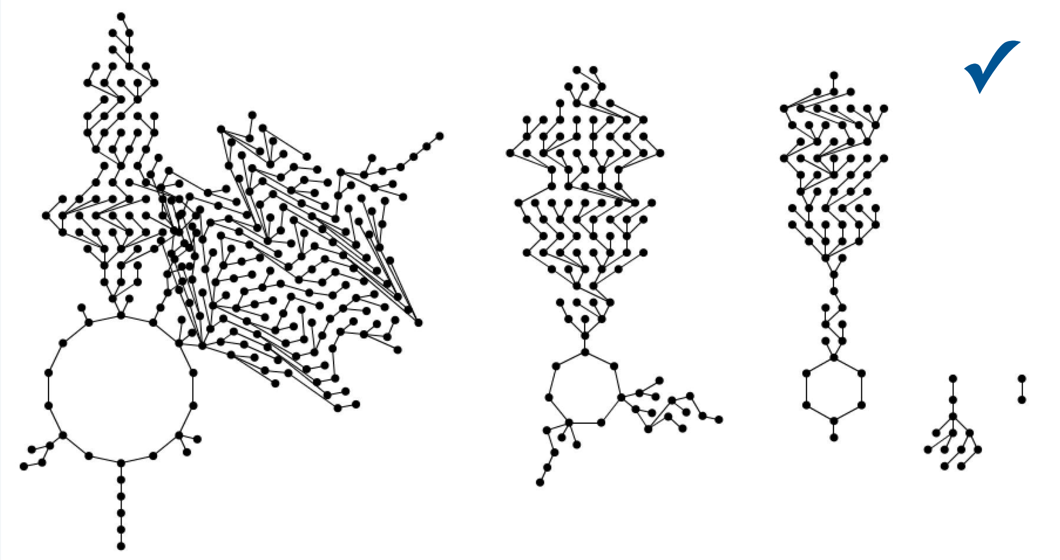
Ex: random 20-bit string with no 00

00011110001101100010 ✗
11101101111001100010 ✗
10000111110111001101 ✗
10001111010001001110 ✗
01100001010101101100 ✗
01010001111110000110 ✗
10100000110110110110 ✗
10110111001011010101 ✗
00100111000101011011 ✗
11111000000111111000 ✗
01000000111100010001 ✗
11111111011011011111 ✓

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.

```
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
00110001001110110001010110100111100100111100010111

% java RandomBitsReject 50 3
11111110111101111111010010010111101111110110111011

% java RandomBitsReject 50 2
01101110110111110101010110101101011111111011111101
```

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.

```
% java RandomBitsReject 100 4
1011110010100011011101010011101000101110010111110011100011100111101110110100111110001010110101011101
```

69 trials

```
% java RandomBitsReject 200 4
1000100111001111011110010100010101001010111011011000110111111001101110101010001001011001011101101000110100101
0001100101010011011001100110011111011011111001101100101100111101001110010010001100110111110
```

655 trials

```
% java RandomBitsReject 300 4
0111100110011110111110011111000111011010101001010011100111101101111010001011000110101110110100101010011110010
1001000100110100101001001000101011011101101110110010101110011110111110001110001111110011101011110100100110110
0011000110101010111111010011110101001010100100010101011000111011101110111101010101
```

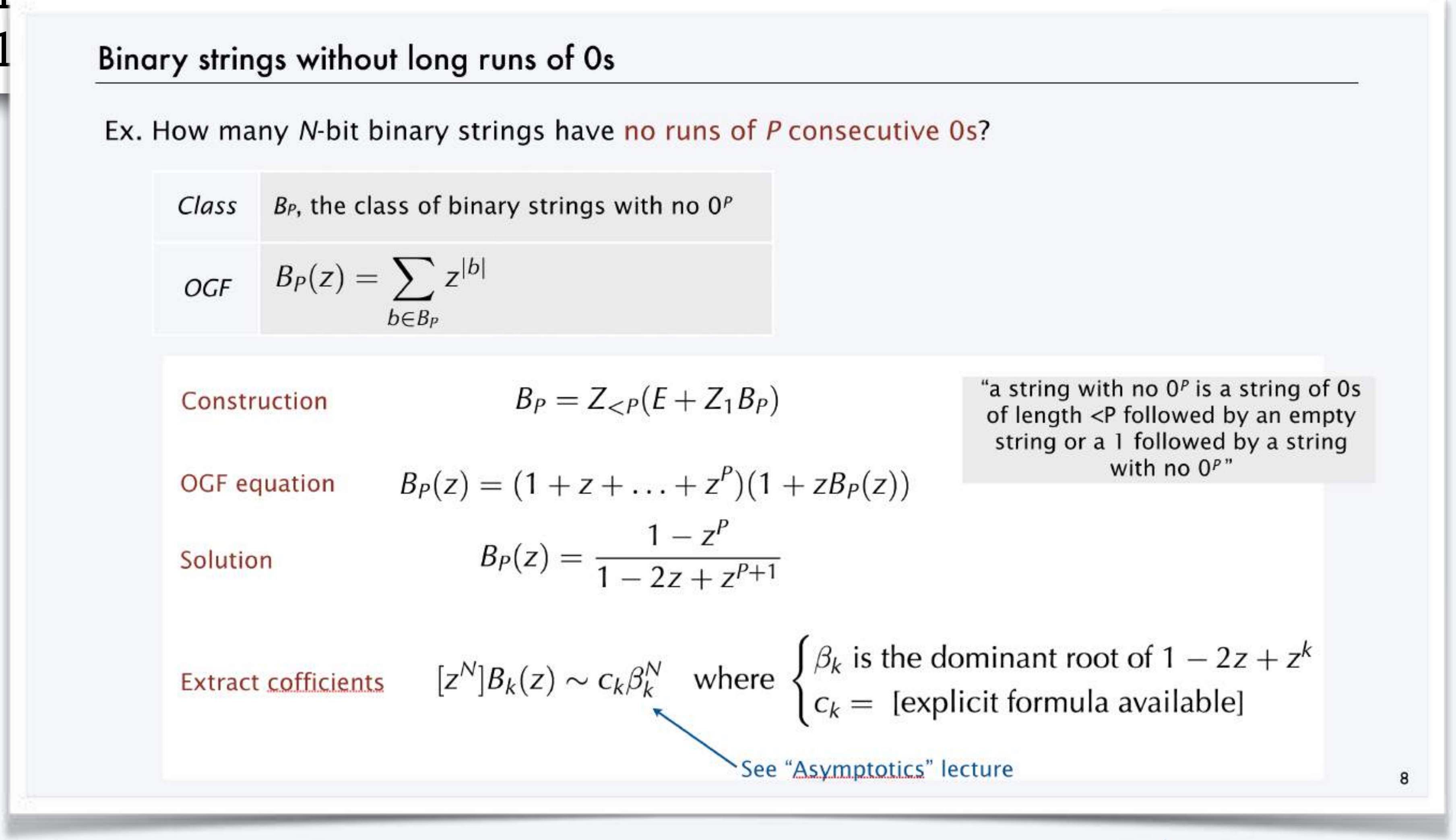
1269 trials

```
% java RandomBitsReject 400 4
1001101000100010111111010011100101011111100110001110111011011110110011010001110111010101010110110111011110100101
1000111111110111110100110010011101011100010010010010011001100011000111110001101001101111001111110100111010 01
0011111001110111111000100101111111001010001111001101010101010111110010111010011100111111011000100011001100010 1
00011101111001111111110011000111000111011100110011110011000110110111111
```

4952205 trials

Analysis clearly exposes the problem

- Probability that an N-bit string has no run of 4 0s is about $1.0917 \times .96328^N \doteq .000000346$ for $N = 400$



AofA lecture 8

Anticipated rejection

Generally not necessary to generate the whole object.

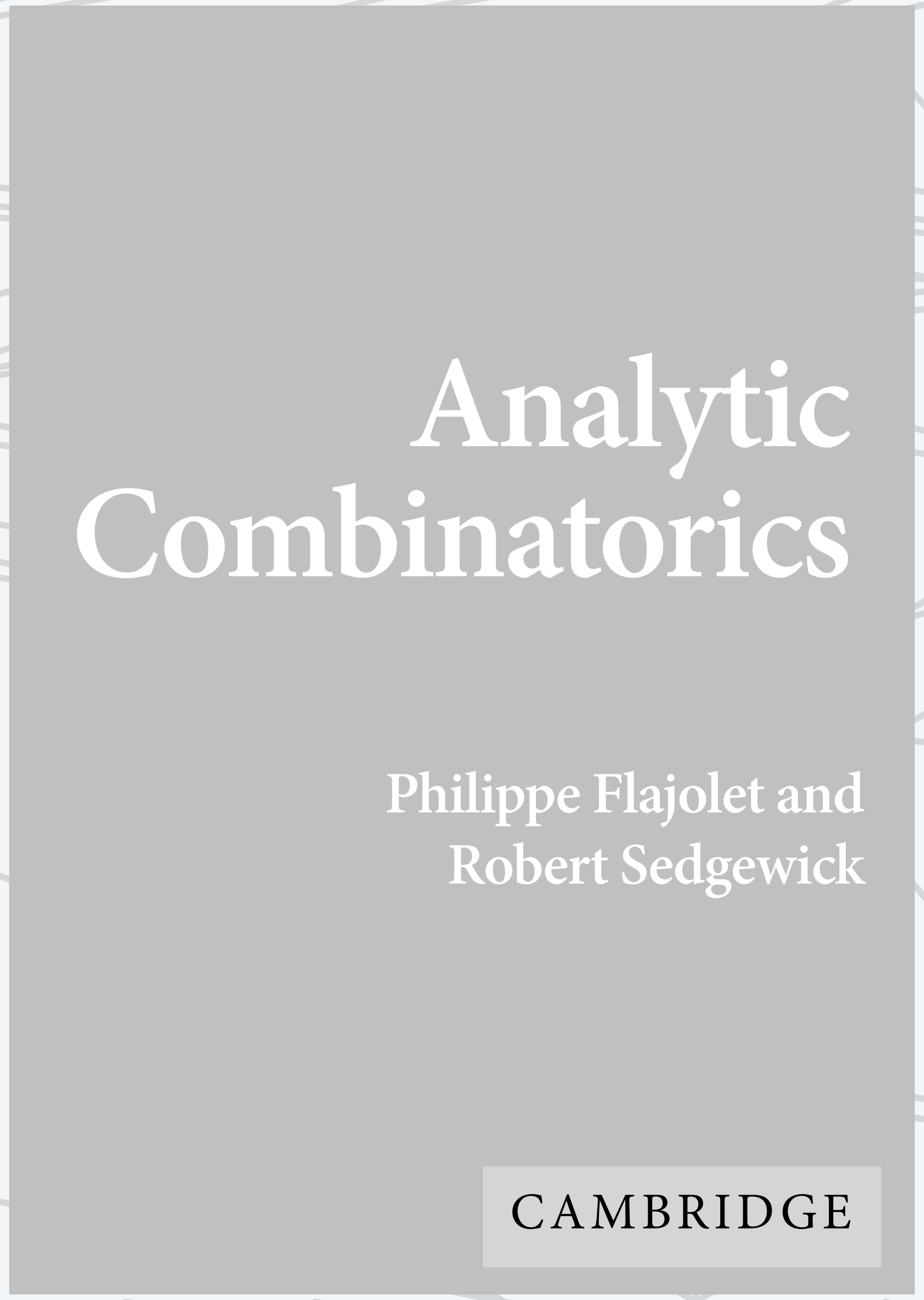
Ex: random 30-bit string with no 000

```
100101111101100010010000001010 ✗
101000101001110110010111101111 ✗
110100010111100110010101110001 ✗
001011101011011010101111001000 ✗
101111010010110111111110100000 ✗
101001011001010110000100010010 ✗
000110111100101000100101100100 ✗
011110001001000010101010011001 ✗
111000100110010111100100010001 ✗
101110000101010010110011110010 ✗
111010100110110101100100010101 ✗
011010100010010011111101010110 ✗
010010010001111010000000110101 ✗
101100100000010110100100010001 ✗
011101011100000111001110100000 ✗
011000001001010110000111001101 ✗
001110101001000010000010110111 ✗
010000001111011010000101010011 ✗
101111111111101011010011010011 ✓
```

only need 14 bits on average (see AofA 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).



http://ac.cs.princeton.edu

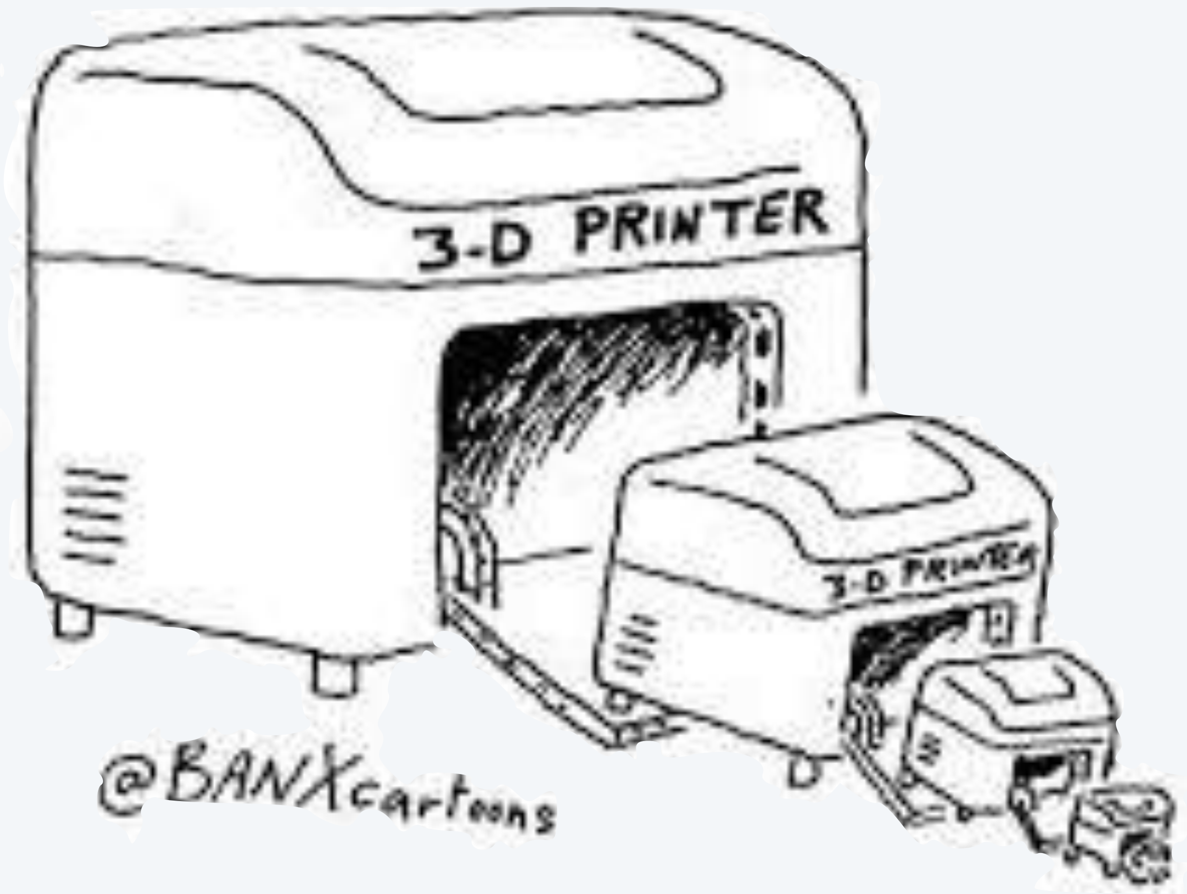
Random Generation of Combinatorial Objects

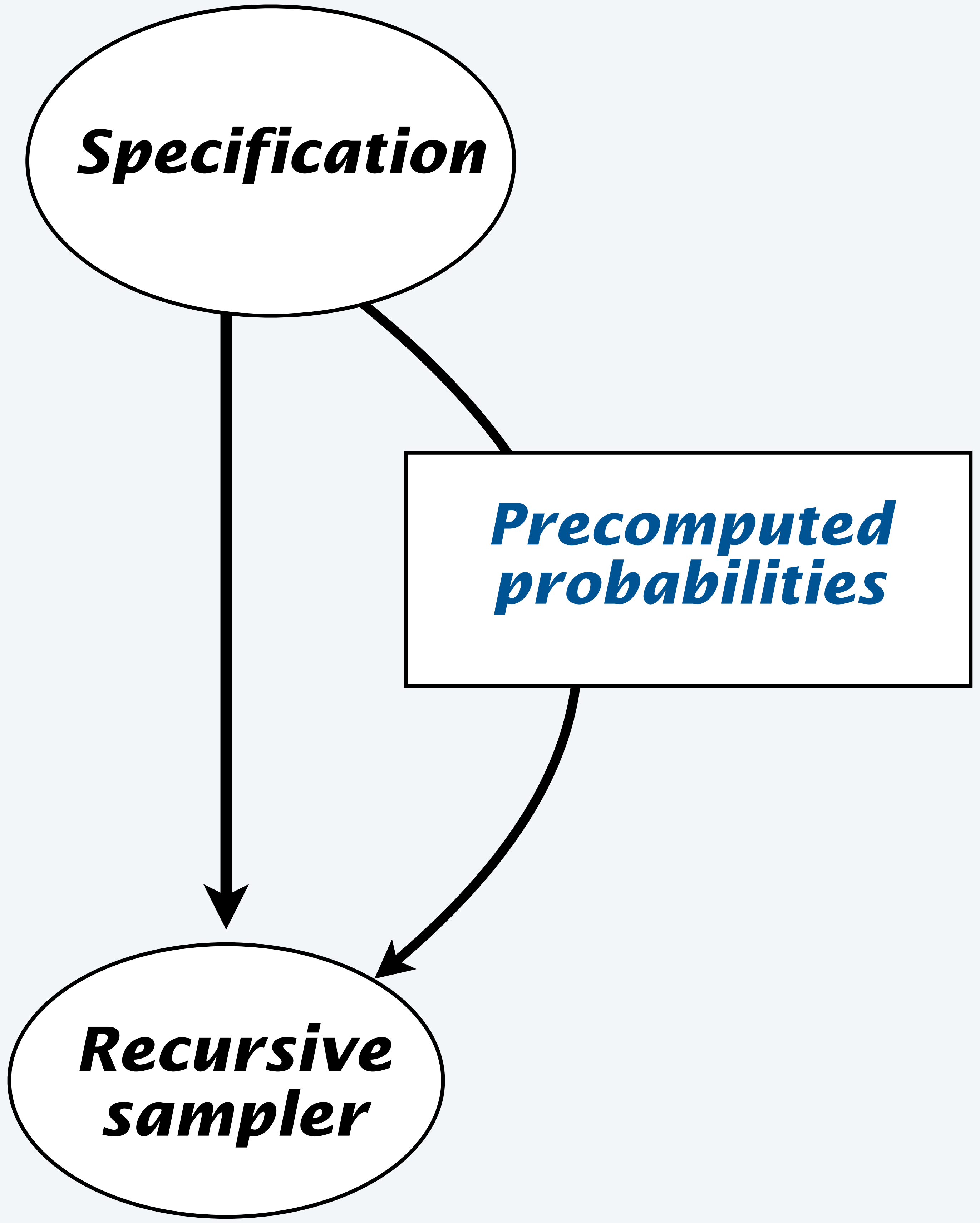
- Basics
- Achieving uniformity
- Rejection
- **Recursive method**
- Analytic samplers

Recursive method

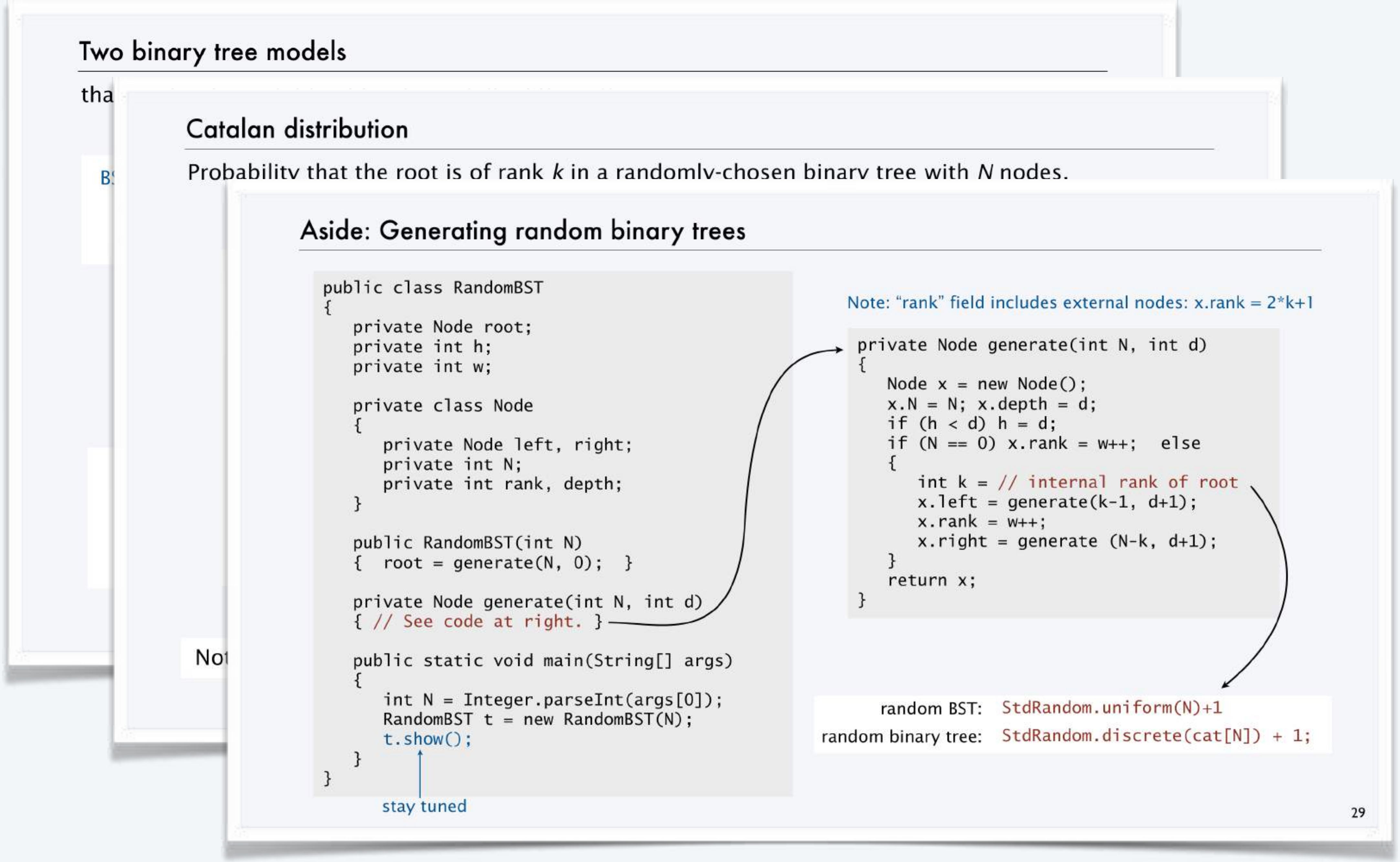
Second technique to consider: *the "recursive method"*

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





Ex: AofA lecture 6 (details revisited soon)


Two binary tree models

Catalan distribution

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

Aside: Generating random binary trees

```
public class RandomBST
{
   private Node root;
   private int h;
   private int w;

   private class Node
   {
      private Node left, right;
      private int N;
      private int rank, depth;
   }

   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();
   }
}
```

stay tuned

Note: "rank" field includes external nodes: x.rank = 2*k+1

```
private Node generate(int N, int d)
{
   Node x = new Node();
   x.N = N; x.depth = d;
   if (h < d) h = d;
   if (N == 0) x.rank = w++;   else
   {
      int k = // internal rank of root
      x.left = generate(k-1, d+1);
      x.rank = w++;
      x.right = generate (N-k, d+1);
   }
   return x;
}
```

random BST:	StdRandom.uniform(N)+1
random binary tree:	StdRandom.discrete(cat[N]) + 1;

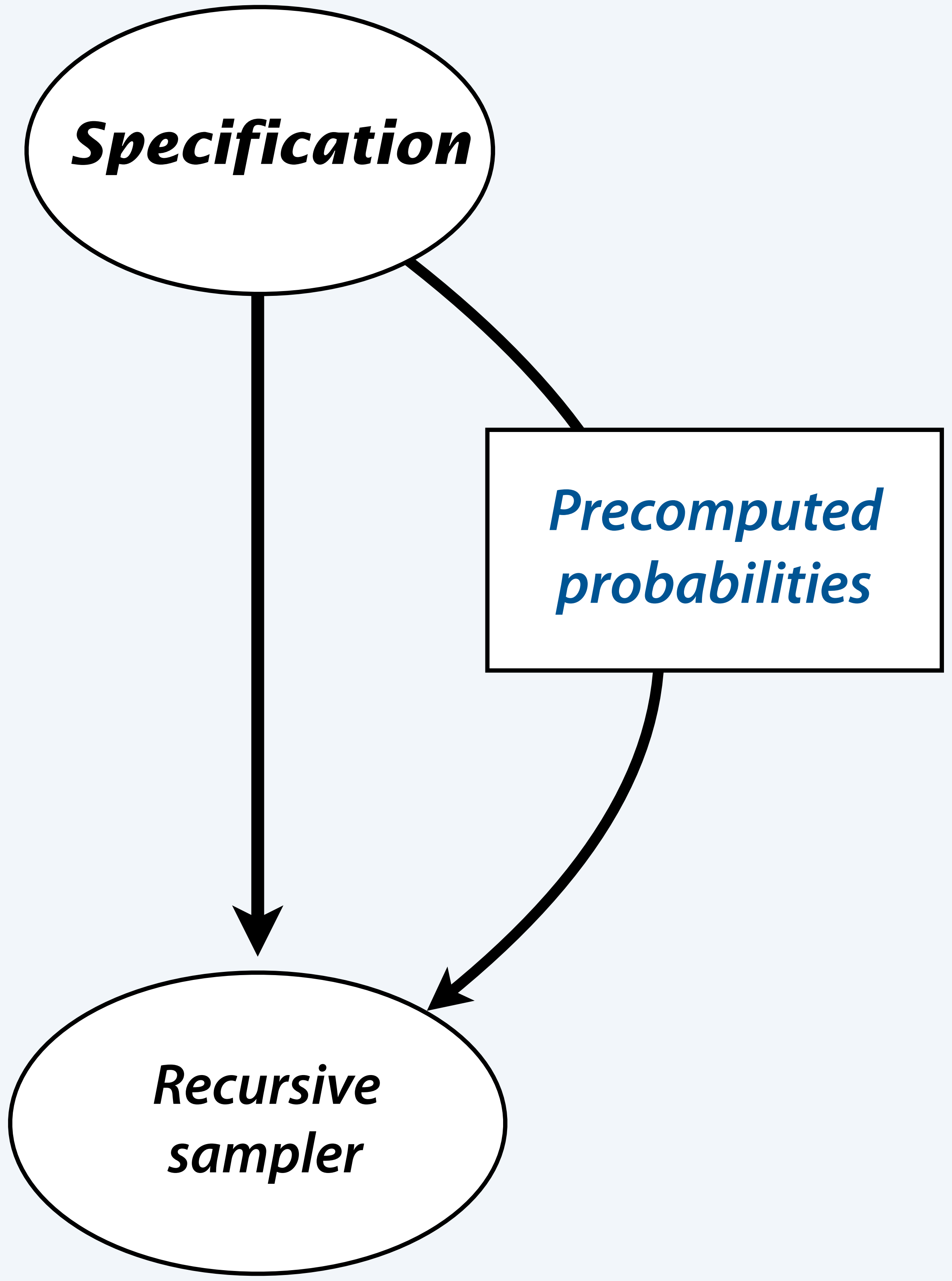
29

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

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



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 ~ ϕ^N
- no 00, ending in 1 is ~ ϕ^{N-1}
- no 00, ending in 10 is ~ ϕ^{N-2}

∴ *Probability of ending in 1 is 1/ϕ*

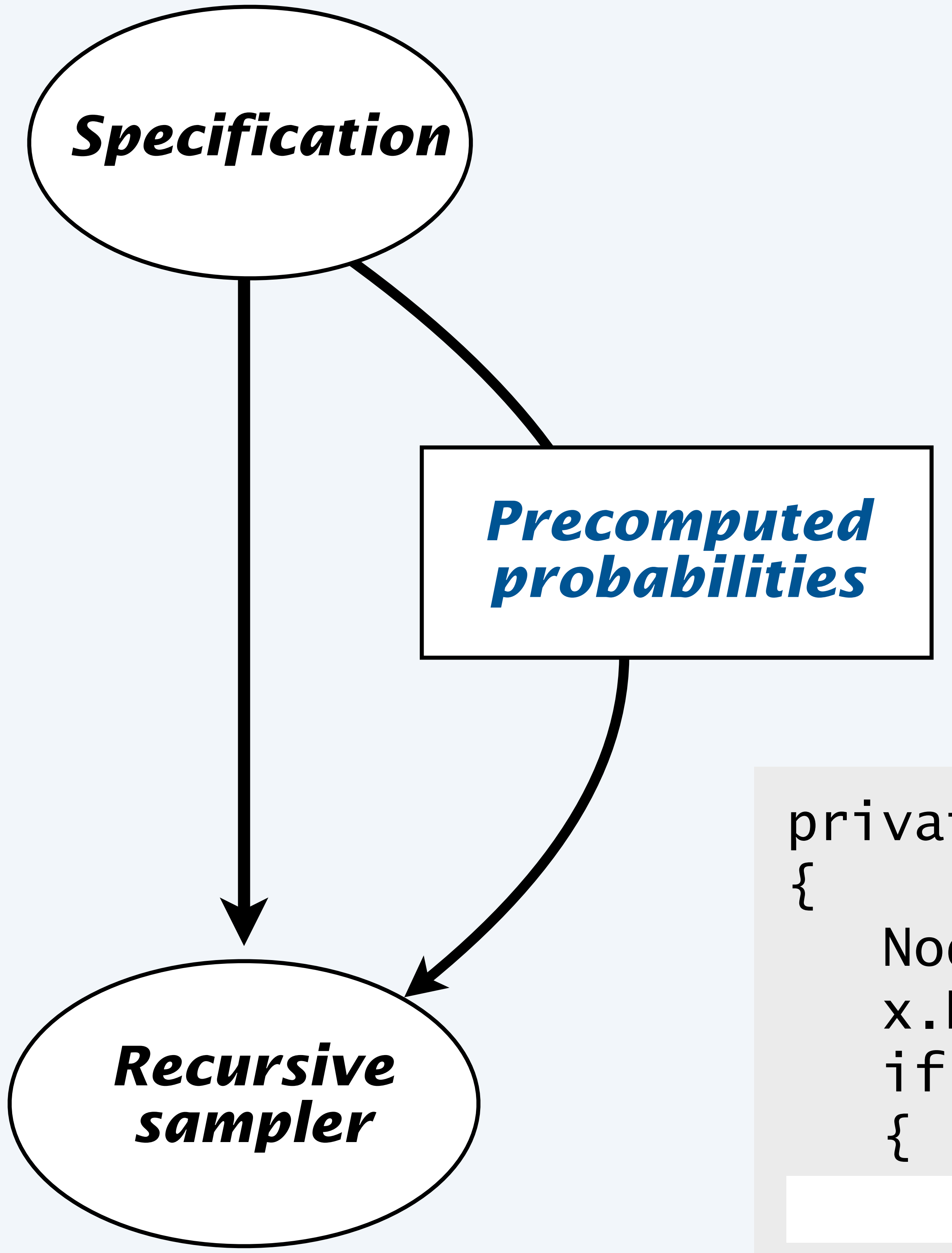
```
private String B00(int N)
{
   if (N == 0) return "";
   if (N == 1) return "0";
   if (StdRandom.bernoulli(1.0/phi))
        return(B00(N-1) + "1";
   else
        return(B00(N-2) + "10";
}
```

```
% java RandomBitsRecursive 50
11111110101110110111110111011011011110101101011111
```

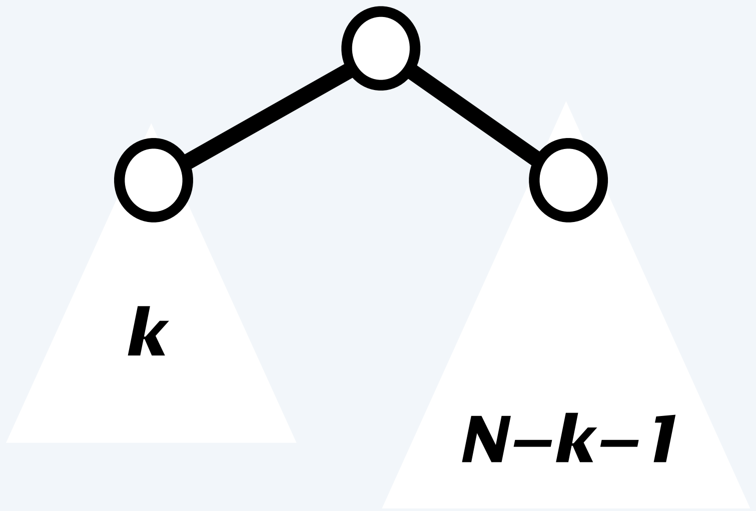
Example 2: Recursive method for random binary trees

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

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



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

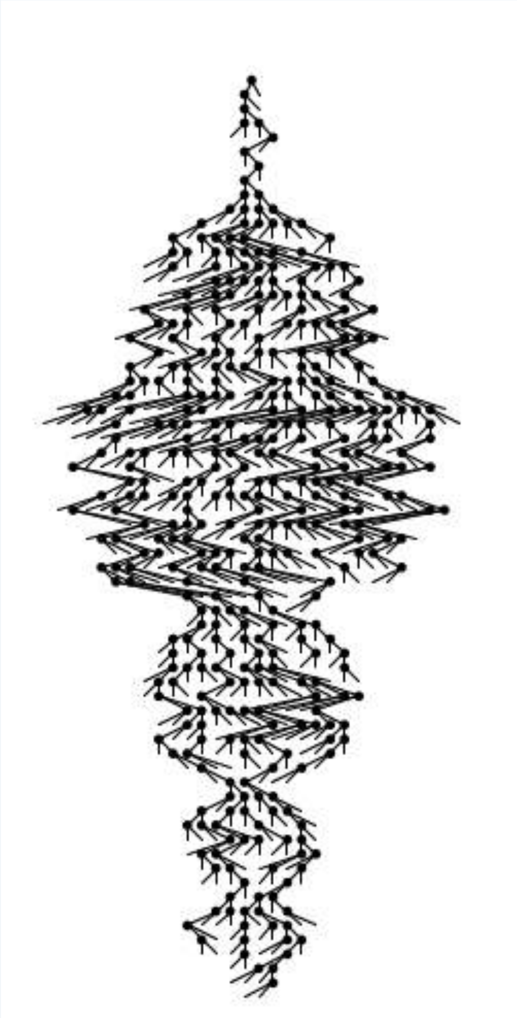


Probability subtree sizes are k and N–k–1

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

```
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;
}
```

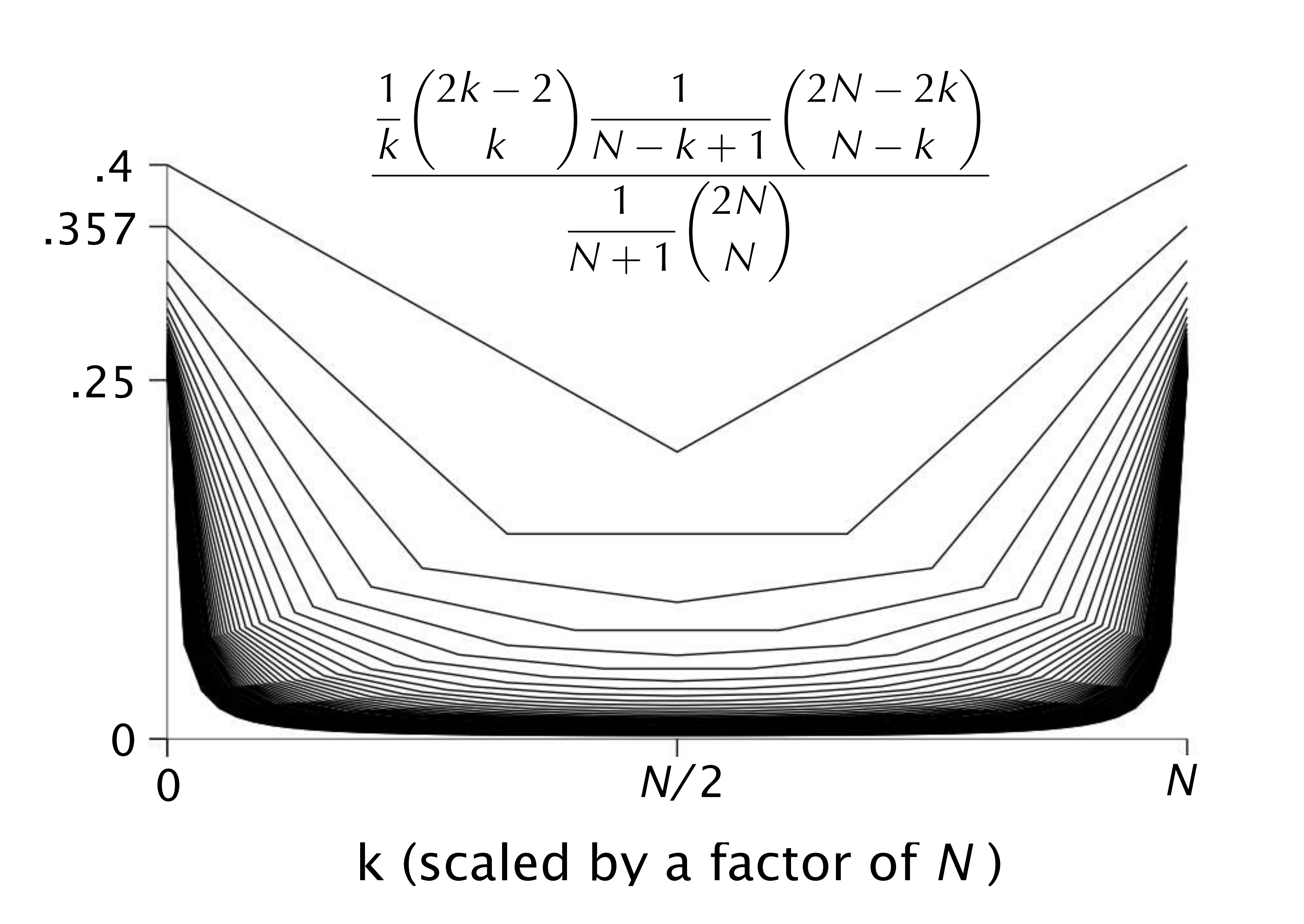
```
% java RandomBTree 100
```



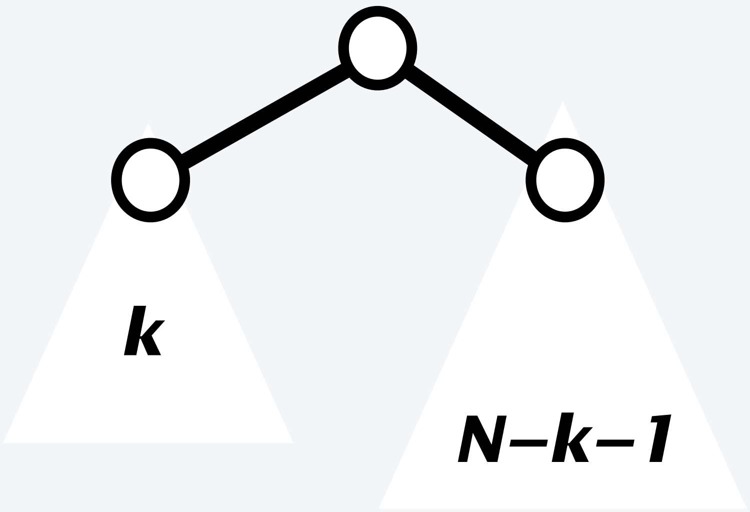
Basis for the recursive method

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):




```
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];
   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*

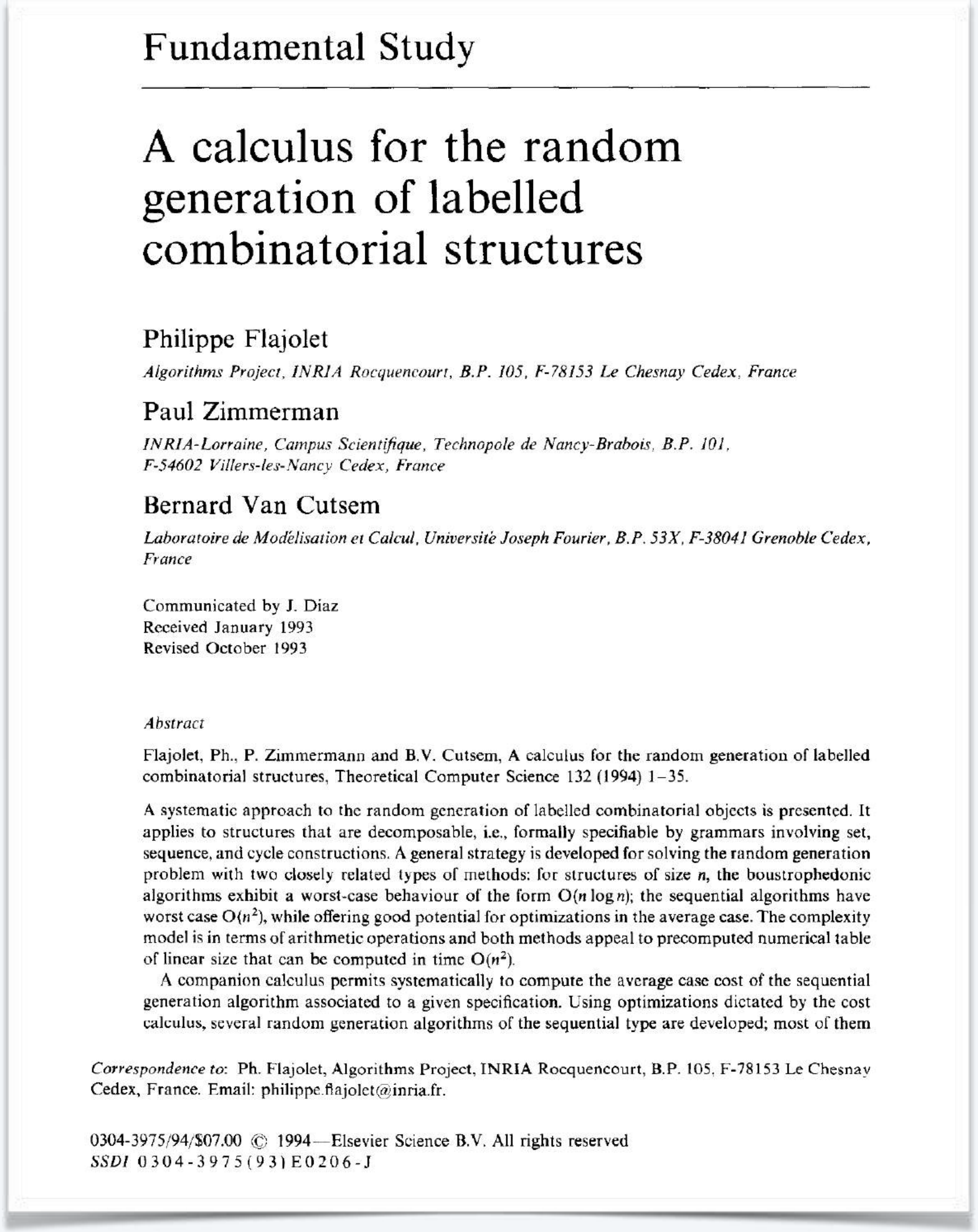
Caveat. Requires excessive time and space, in general (quadratic, in this case).

“If you can specify it, you can generate a random one.”

Flajolet, Zimmerman, and Van Cutsem, *A calculus for the random generation of labelled combinatorial structures, Theoretical Computer Science, 1994.*

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.



Fundamental Study

A calculus for the random generation of labelled combinatorial structures

Philippe Flajolet
Algorithms Project, INRIA Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France

Paul Zimmerman
INRIA-Lorraine, Campus Scientifique, Technopole de Nancy-Brabois, B.P. 101, F-54602 Villers-les-Nancy Cedex, France

Bernard Van Cutsem
Laboratoire de Modélisation et Calcul, Université Joseph Fourier, B.P. 53X, F-38041 Grenoble Cedex, France

Communicated by J. Diaz
Received January 1993
Revised October 1993

Abstract

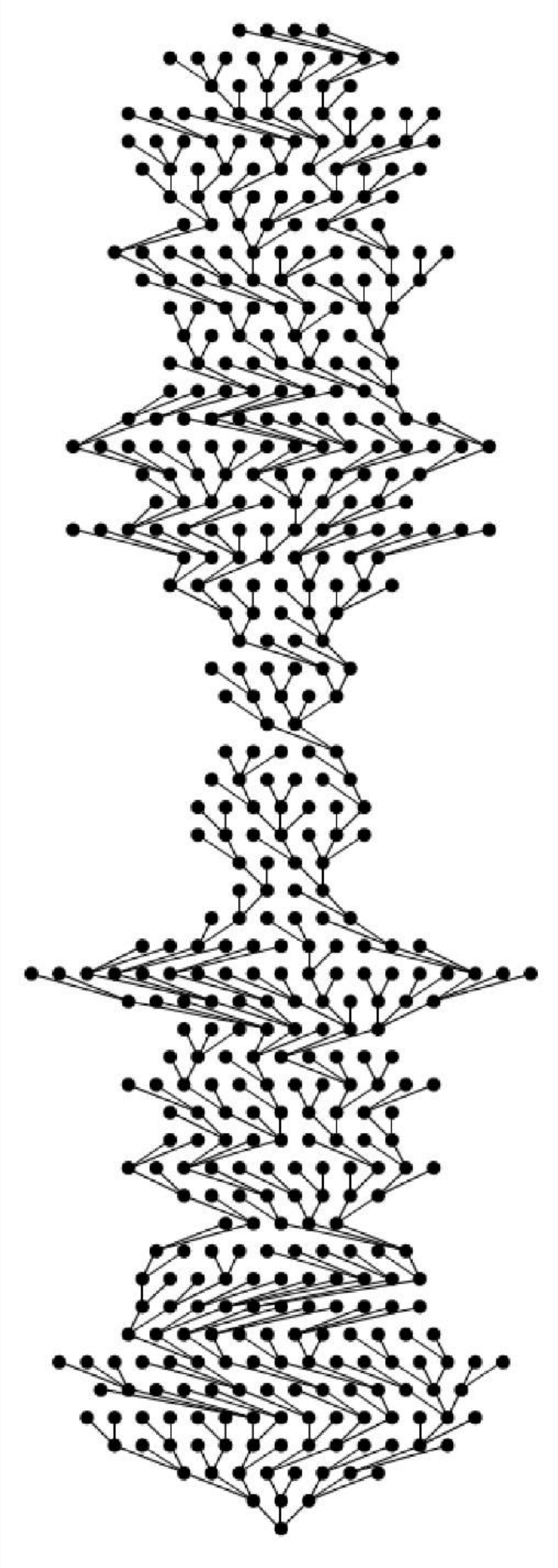
Flajolet, Ph., P. Zimmermann and B.V. Cutsem, A calculus for the random generation of labelled combinatorial structures, Theoretical Computer Science 132 (1994) 1–35.

A systematic approach to the random generation of labelled combinatorial objects is presented. It applies to structures that are decomposable, i.e., formally specifiable 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 methods: for structures of size n, the boustrophedonic algorithms exhibit a worst-case behaviour of the form $O(n \log n)$; the sequential algorithms have worst case $O(n^2)$, while offering good potential for optimizations in the average case. The complexity model is in terms of arithmetic operations and both methods appeal to precomputed numerical table of linear size that can be computed in time $O(n^2)$.

A companion calculus permits systematically to compute the average case cost of the sequential generation algorithm associated to a given specification. Using optimizations dictated by the cost calculus, several random generation algorithms of the sequential type are developed; most of them

Correspondence to: Ph. Flajolet, Algorithms Project, INRIA Rocquencourt, B.P. 105, F-78153 Le Chesnay Cedex, France. Email: philippe.flajolet@inria.fr.

SSDI 0304-3975(93)E0206-J



a random 0-2-3 tree

Industrial-strength random sampling

Flajolet and Salvy, *Computer Algebra Libraries for Combinatorial Structures, J. of Symbolic Computation, 1995.*

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

https://www.maplesoft.com/support/help/maple/view.aspx?path=combstruct

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).

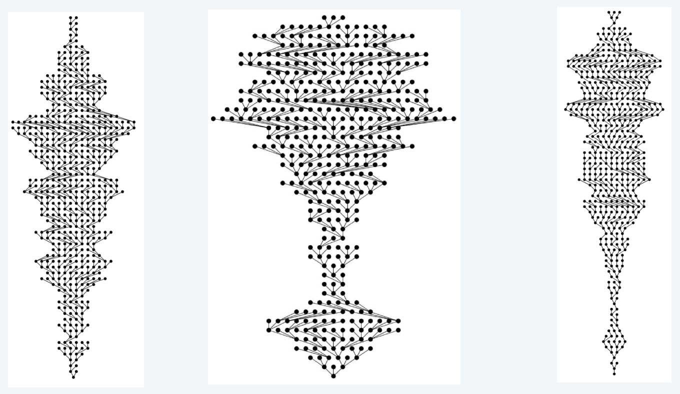
Context

Recursive method leads to *automatic* uniform sampler for any constructible class,

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

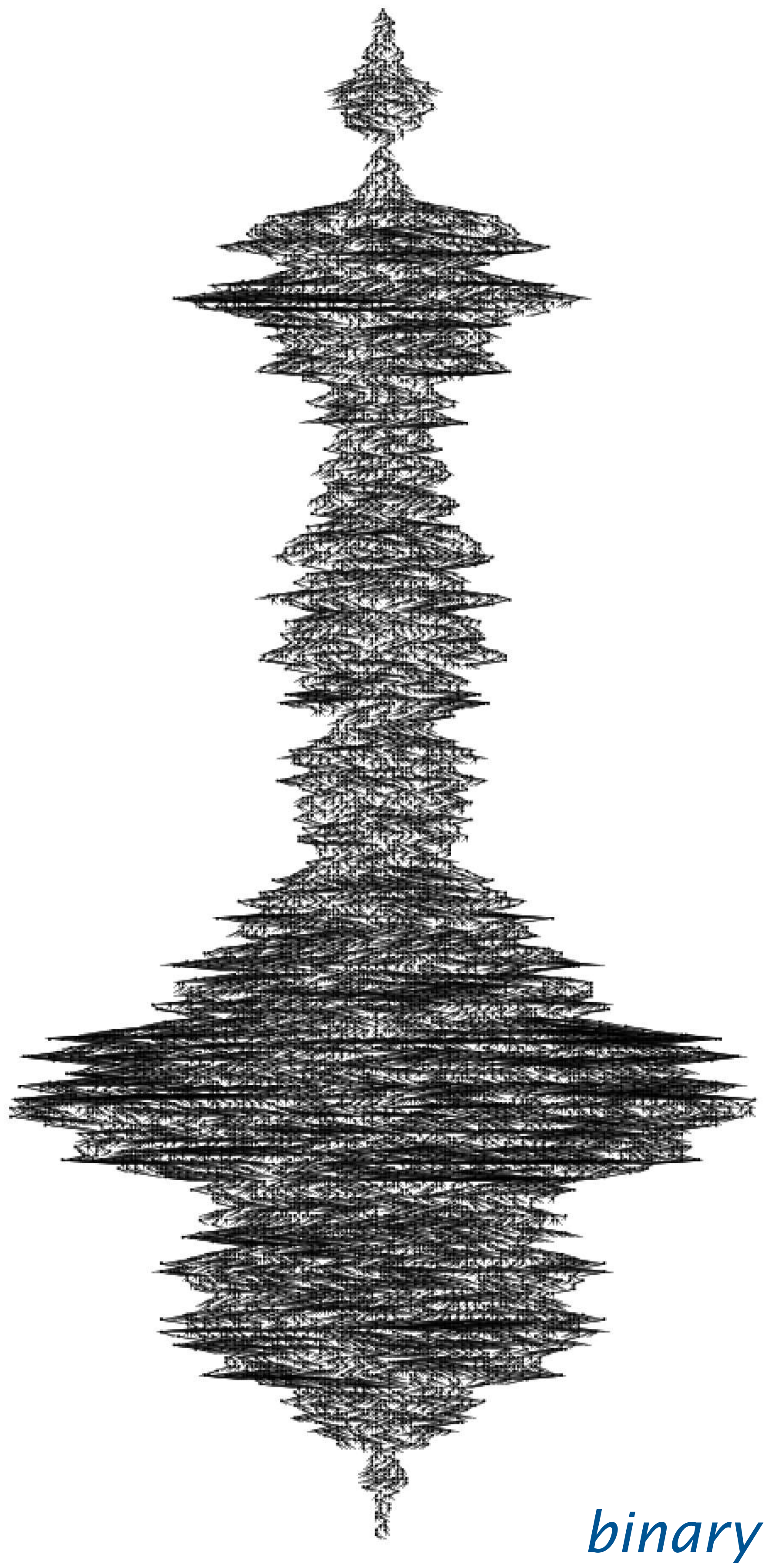
	scalable	*extensible*
recursive method	*not always*	✓
Remy's algorithm	✓	✗
next challenge	✓	✓

recursive method



binary *ternary* *Motzkin*

Remy's algorithm

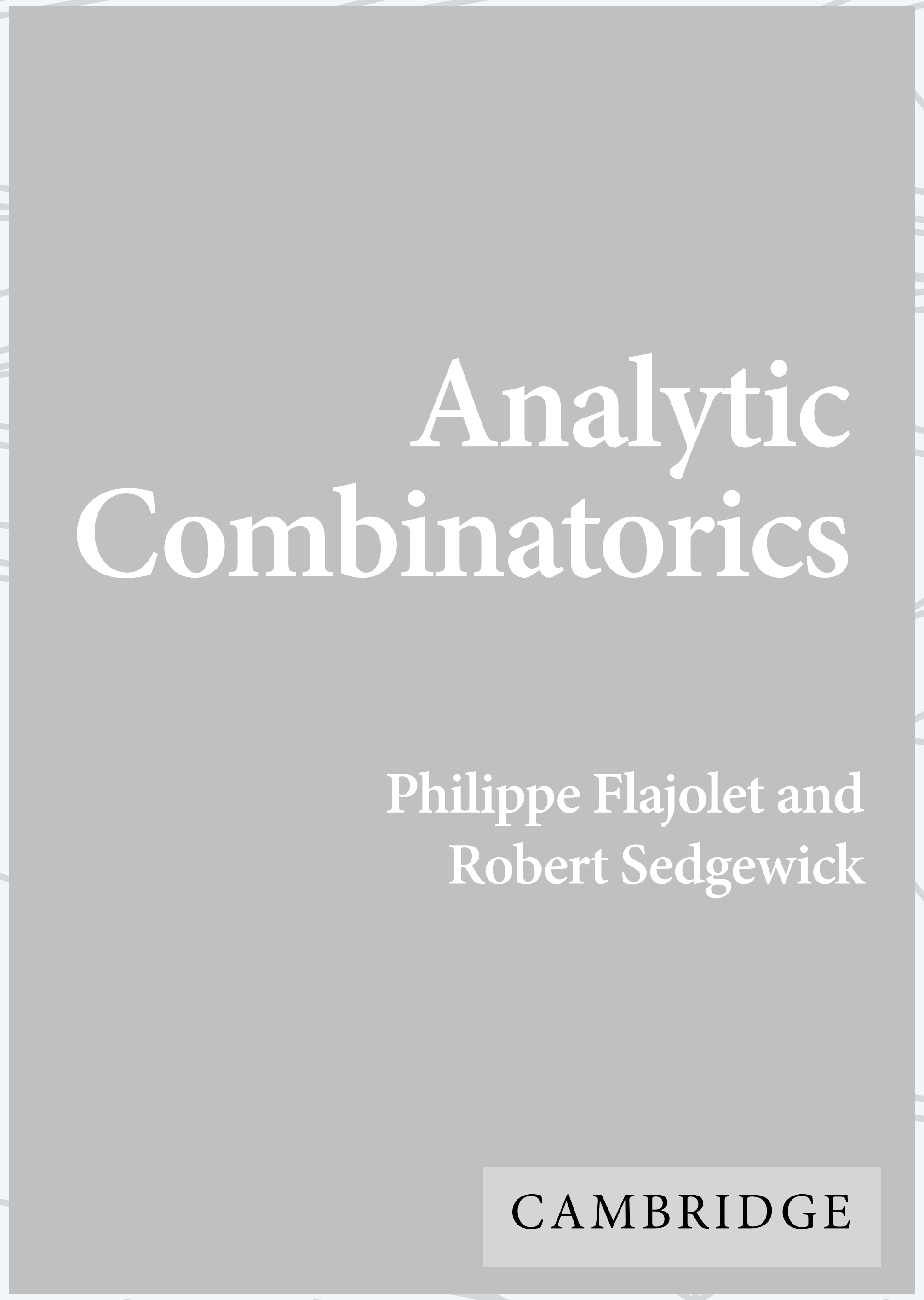


binary

Next challenge

ternary?
Motzkin?
...?

Next. Scalable *and* extensible uniform samplers



http://ac.cs.princeton.edu

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 $\dfrac{x^{|a|}}{A(x)}$

A. Noack. *A class of random variables with discrete distributions*, Annals of Mathematical Statistics, 1950.

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.

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

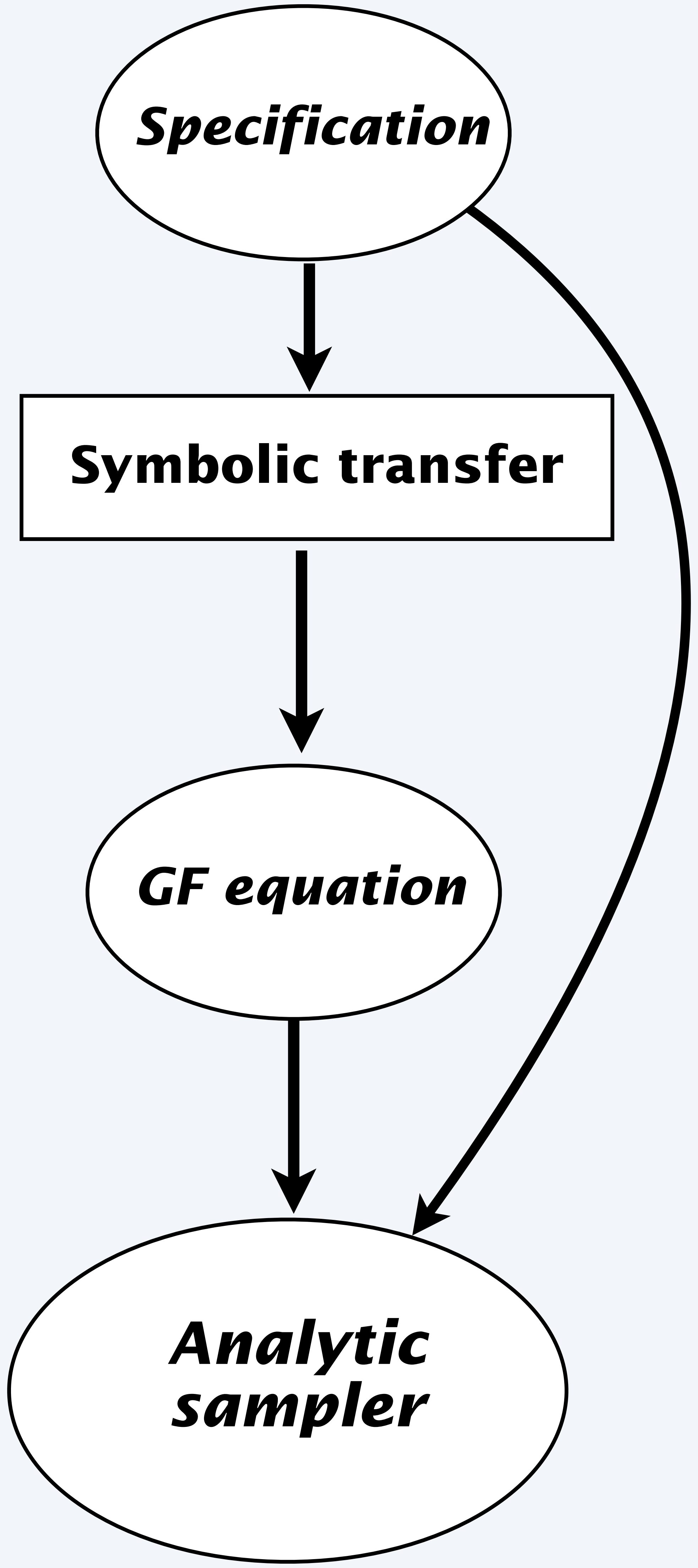
$$E(N_x) = \sum_{a \in A} |a| \frac{x^{|a|}}{A(x)} = \sum_{n \geq 0} n A_n \frac{x^n}{A(x)}$$

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

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 ε`	1 object of size 0, OGF is 1
atomic class	***Z***	`return •`	1 object of size 1, OGF is z

Dijoint union and Cartesian product construction for analytic samplers

Disjoint union

Analytic sampler for $\boldsymbol{A} = \boldsymbol{B} + \boldsymbol{C}$

```
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 $\boldsymbol{A} = \boldsymbol{B} \times \boldsymbol{C}$

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

combines B and C into a single object

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 ε`	1 object of size 0, OGF is 1
atomic class	***Z***	`return •`	1 object of size 1, OGF is z
disjoint union	***A = B + C***	`u = 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 × C***	`return` *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 → Symbolic transfer → GF equation → Analytic sampler

B_4, the class of all bitstrings with no 0^4 ← see Lecture 1

$$B_4 = Z_{<4}\,(E + Z_1 B_4)$$

E	`return ε`
Z	`return •`
A = B + C	`u = StdRandom.bernoulli(B(x)/A(x))` `if (u) return B else return C`
A = B × C	`return compose(B, C)`

concatentation

$$B_4(z) = (1 + z + z^2 + z^3)(1 + zB_4(z))$$

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

```
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 "" "0" "00" "000" with equal probability

Critical question about an analytic sampler

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

```
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 0^4

$$B_4 = Z_{<4}\,(E + Z_1 B_4)$$

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

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:

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

Therefore, to generate a sample of expected size N

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

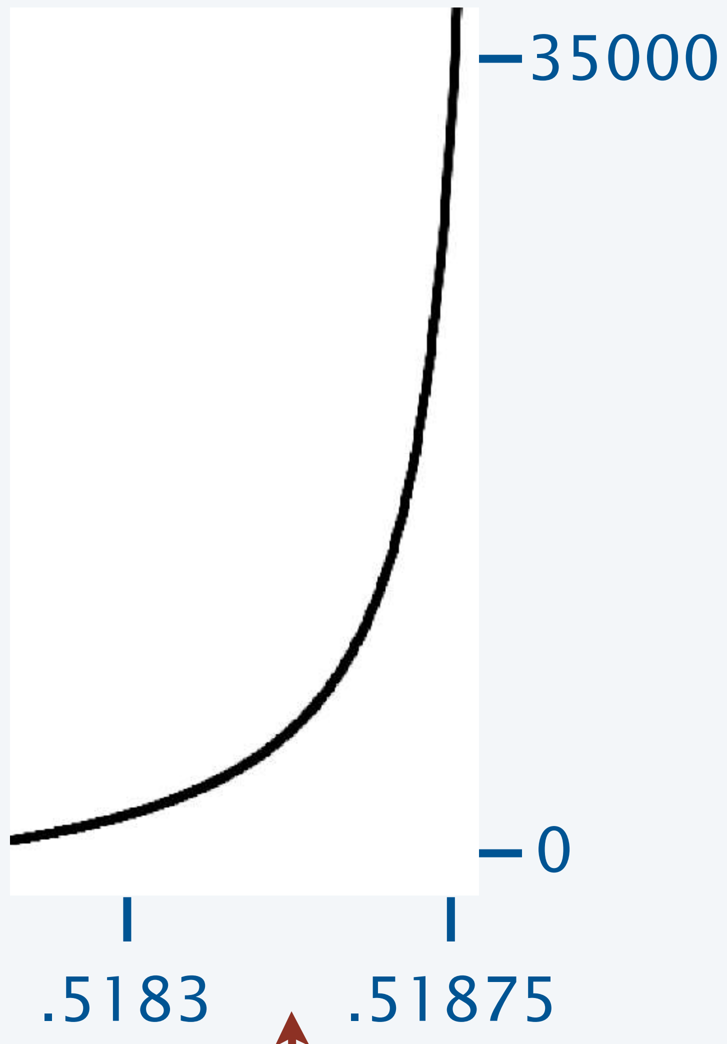
Ex. random string with no 0⁴

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

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

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

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



the action
is very close
to the singularity

Ex. random string with no 0⁴

$$N \sim \frac{\beta}{1 - \beta r}$$

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

Practical consideration: variance

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

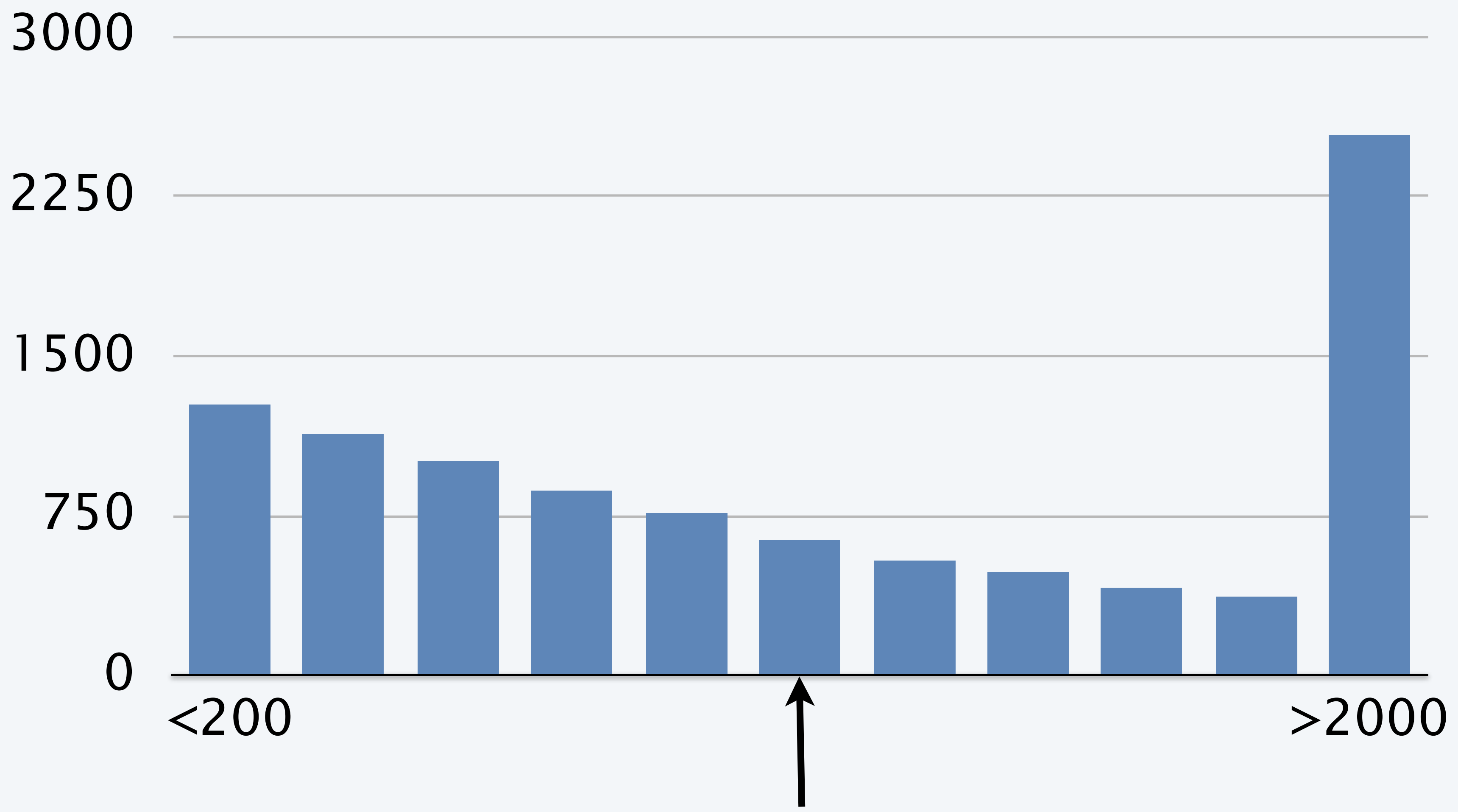
```
double beta = 1.9276;
double r = (1.0*N)/(beta*(1.0 + N);
StdOut.println(generate(r));
```

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

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

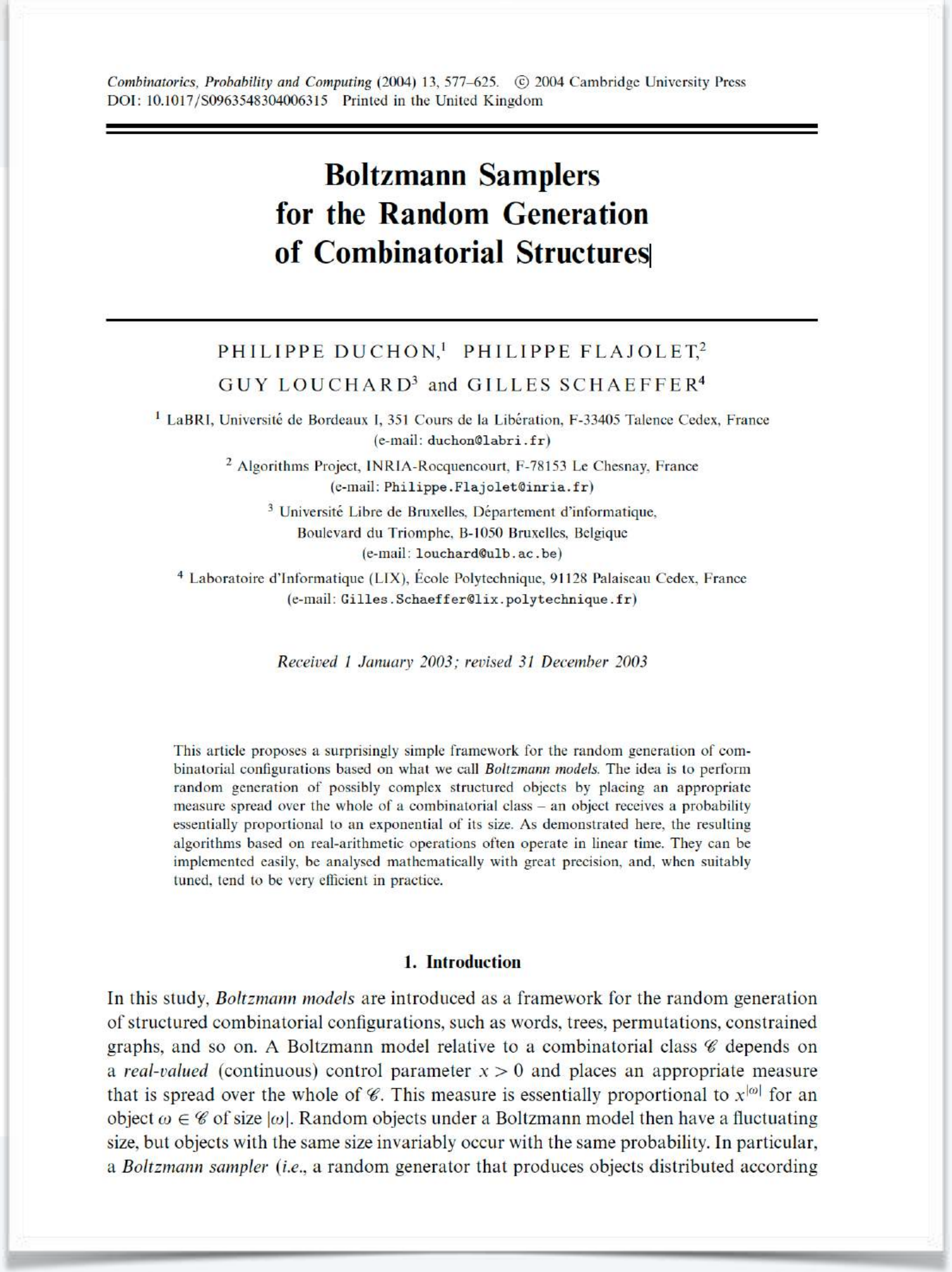
Bad news. Some of the strings are very long

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*.

“If you can specify it, you can generate a ***HUGE*** random one.”

Duchon, Flajolet, Louchard, and Schaeffer, *Boltzmann Samplers for the Random Generation of Combinatorial Structures, Combinatorics, Probability, and Computing, 2004.*


Combinatorics, Probability and Computing (2004) 13, 577–625. © 2004 Cambridge University Press
DOI: 10.1017/S0963548304006315 Printed in the United Kingdom

Boltzmann Samplers for the Random Generation of Combinatorial Structures

PHILIPPE DUCHON,[1] PHILIPPE FLAJOLET,[2] GUY LOUCHARD[3] and GILLES SCHAEFFER[4]

[1] LaBRI, Université de Bordeaux I, 351 Cours de la Libération, F-33405 Talence Cedex, France (e-mail: duchon@labri.fr)
[2] Algorithms Project, INRIA-Rocquencourt, F-78153 Le Chesnay, France (e-mail: Philippe.Flajolet@inria.fr)
[3] Université Libre de Bruxelles, Département d’informatique, Boulevard du Triomphe, B-1050 Bruxelles, Belgique (e-mail: louchard@ulb.ac.be)
[4] Laboratoire d’Informatique (LIX), École Polytechnique, 91128 Palaiseau Cedex, France (e-mail: Gilles.Schaeffer@lix.polytechnique.fr)

Received 1 January 2003; revised 31 December 2003

This article proposes a surprisingly simple framework for the random generation of combinatorial configurations based on what we call *Boltzmann models*. The idea is to perform random generation of possibly complex structured objects by placing an appropriate measure spread over the whole of a combinatorial class – an object receives a probability essentially proportional to an exponential of its size. As demonstrated here, the resulting algorithms based on real-arithmetic operations often operate in linear time. They can be implemented easily, be analysed mathematically with great precision, and, when suitably tuned, tend to be very efficient in practice.

1. Introduction

In this study, *Boltzmann models* are introduced as a framework for the random generation of structured combinatorial configurations, such as words, trees, permutations, constrained graphs, and so on. A Boltzmann model relative to a combinatorial class $\mathcal{C}$ depends on a *real-valued* (continuous) control parameter $x > 0$ and places an appropriate measure that is spread over the whole of $\mathcal{C}$. This measure is essentially proportional to $x^{|\omega|}$ for an object $\omega \in \mathcal{C}$ of size $|\omega|$. Random objects under a Boltzmann model then have a fluctuating size, but objects with the same size invariably occur with the same probability. In particular, a *Boltzmann sampler* (i.e., a random generator that produces objects distributed according

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.*

assumes an oracle exists that can evaluate generating functions efficiently

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

Summary

Ex. random string with no 0^4

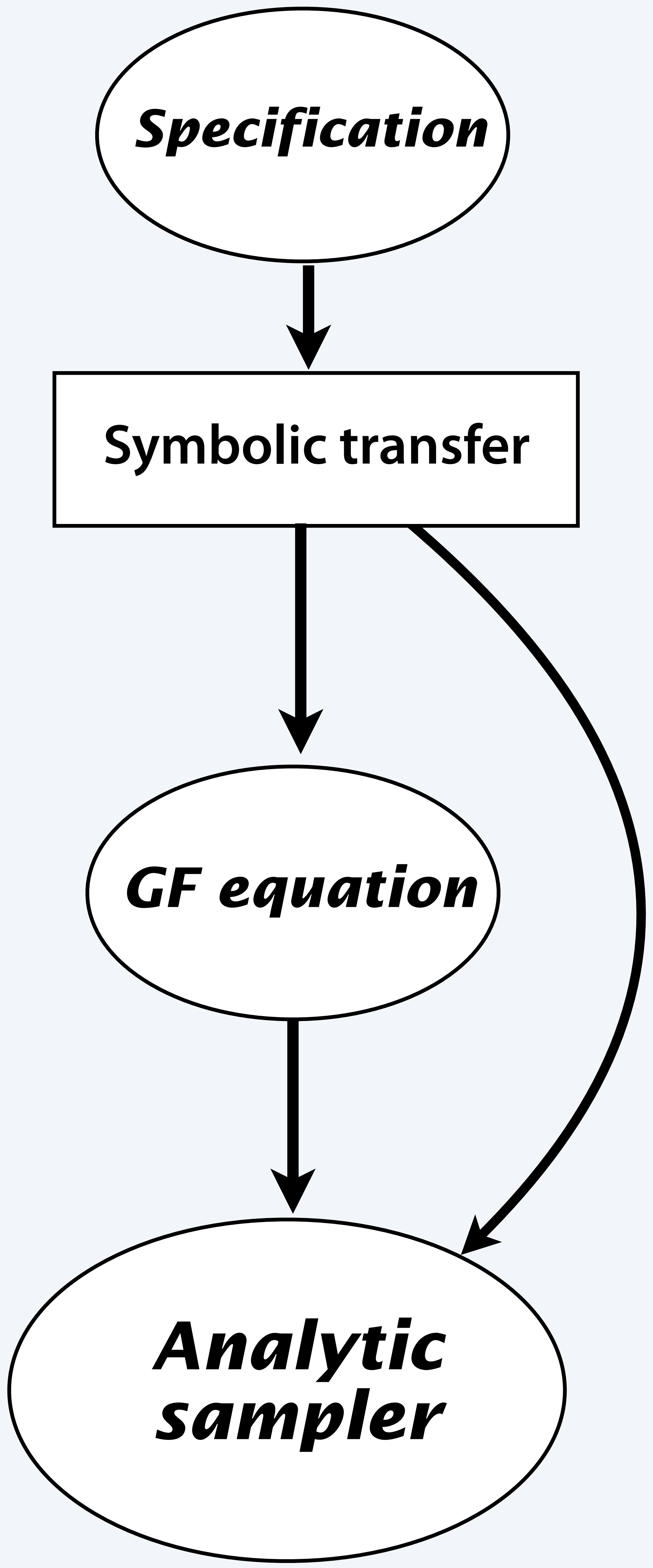
```
public class RandomStringNo4
{
  static int N;
  static double r;
  static double p;

  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);
  }
}
```

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 RandomStringNo4 50
001000100010001000100011001000100011100101110

% java RandomStringNo4 50
0001001000101000100101010100010001010001101010110001110

% java RandomStringNo4 50
01010101001001010100010010001100010010100010010001
```

Analytic sampler for random binary trees



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

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

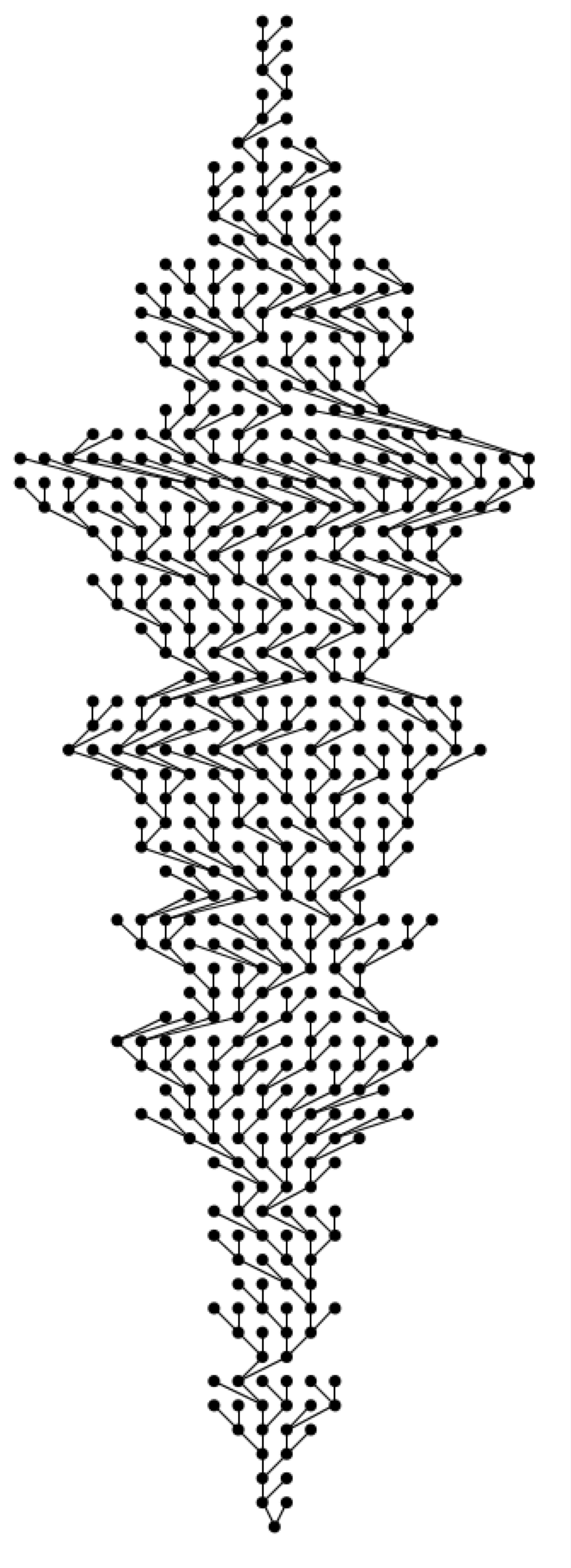
```
private double T(double r)
{ return (1.0 - Math.sqrt(1.0 - 4.0*r))/2.0; }

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;
}
```

E	return □
Z	return ●
A = B + C	if $u < B(x)/A(x)$ return B else return C
A = B × 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 $N = r\dfrac{A'(r)}{A(r)}$

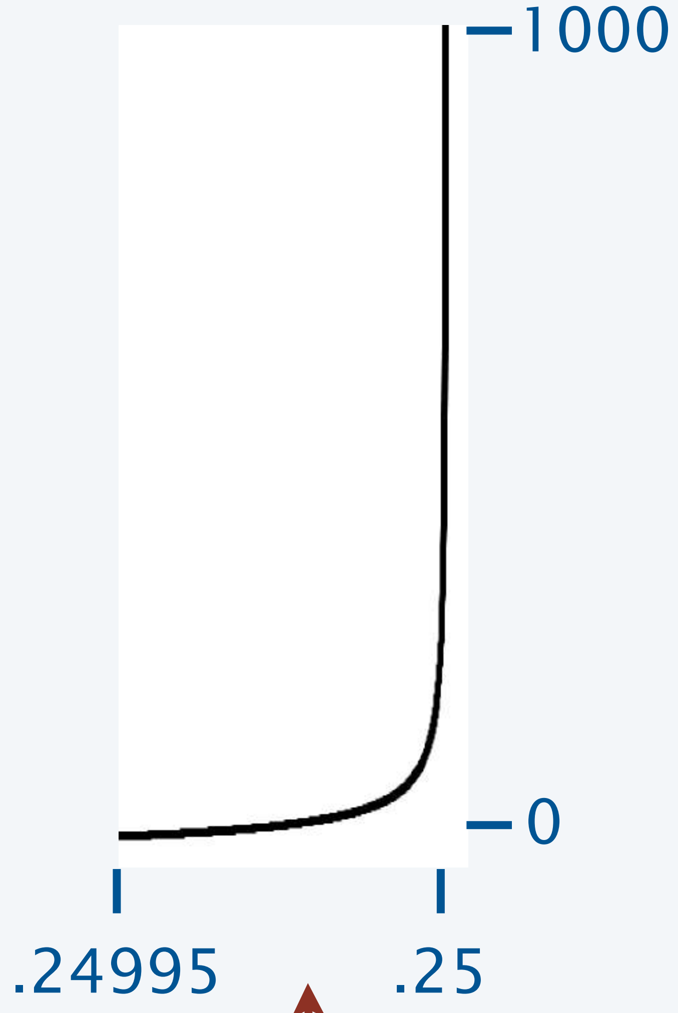


Expected size of a random binary tree

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

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

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



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)

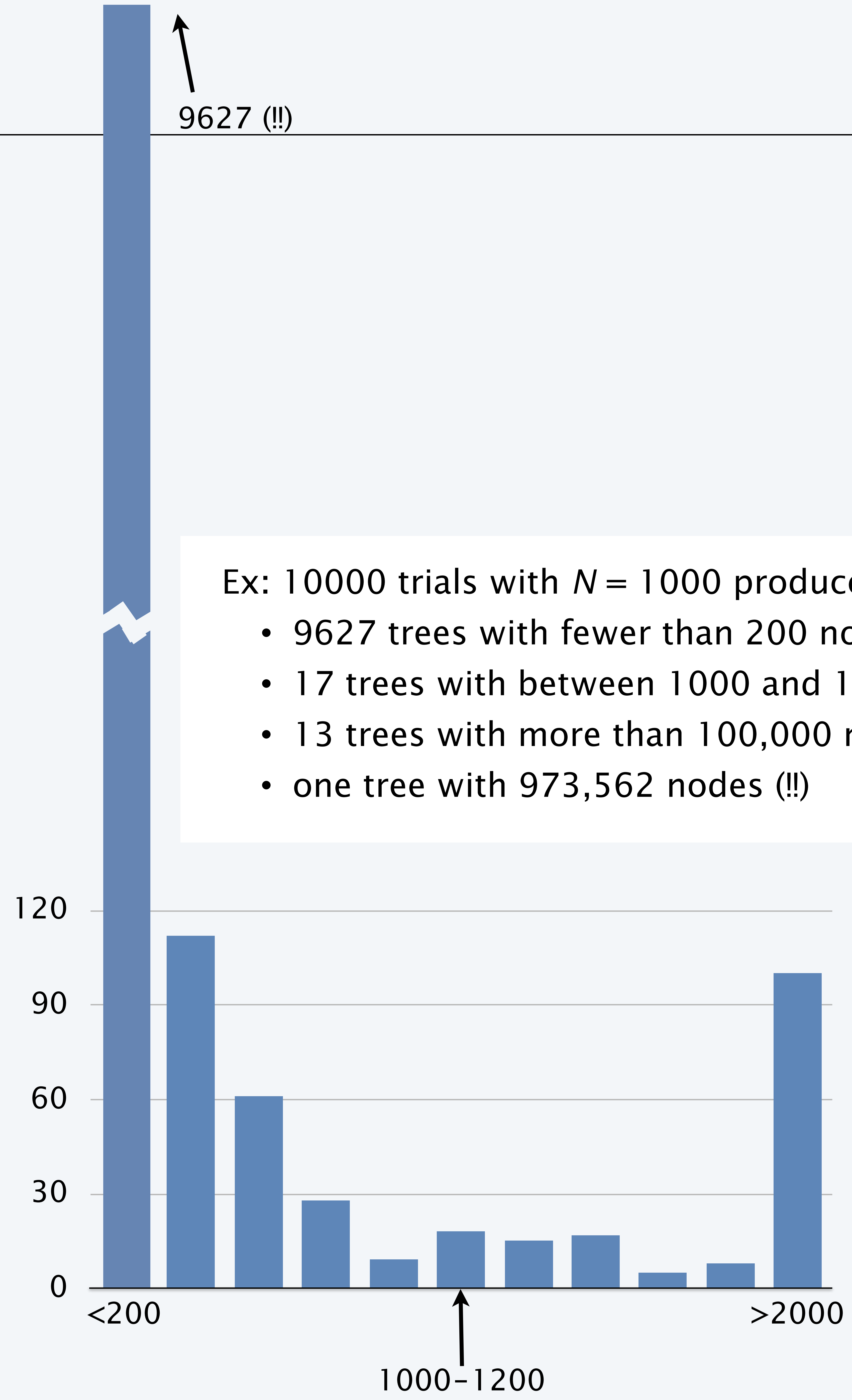
$$\frac{r}{T(r)} = \frac{N}{T'(r)} = N\sqrt{1-4r} = \frac{N}{2N-1} = \frac{1}{2-\frac{1}{N}}$$

Analytic sampler for random binary trees

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

```
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;
}
```

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 (!!)



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.

Singular analytic sampler for trees with anticipated rejection

Idea. Just use the singular value.

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

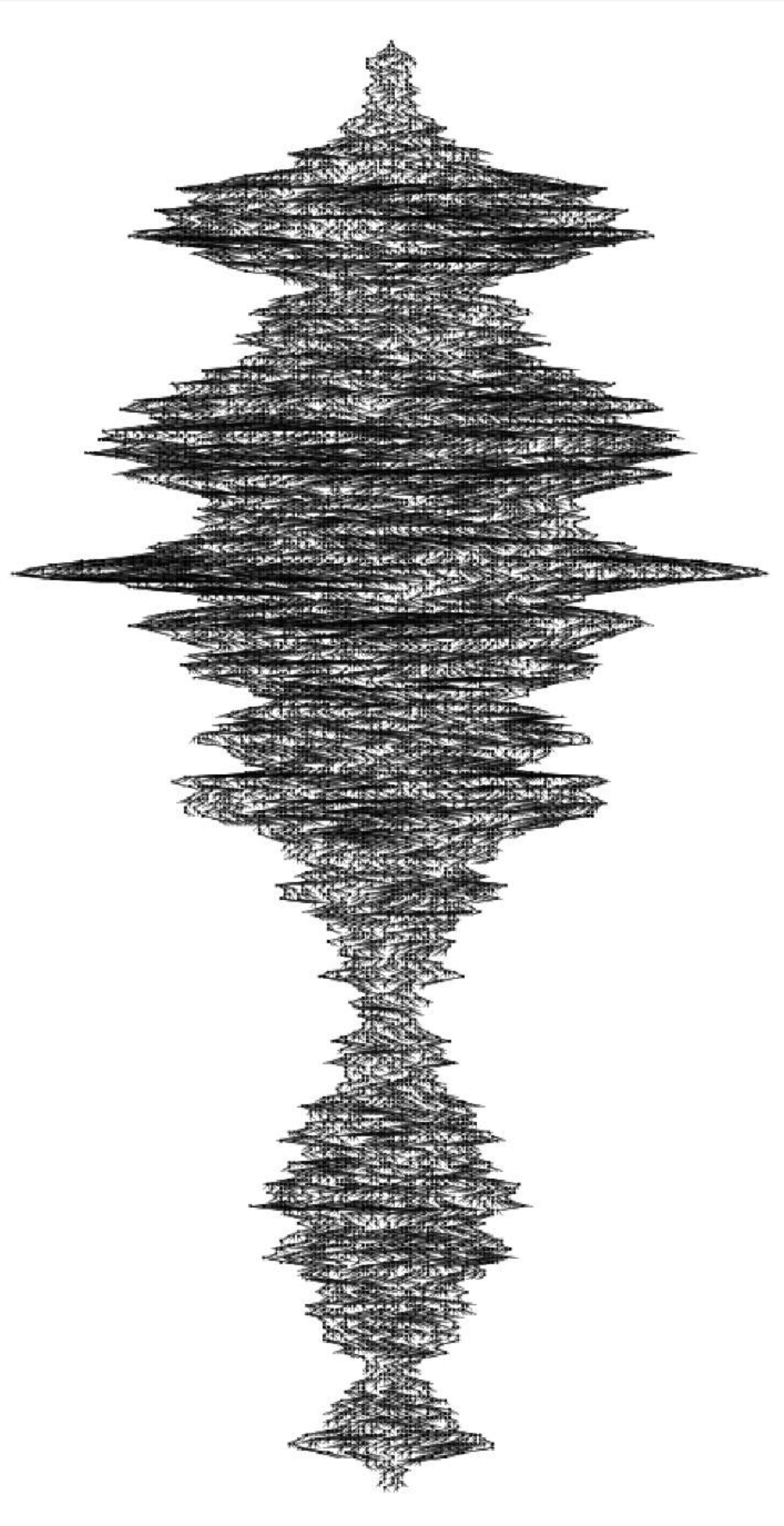
may as well just use 1/4
which gives $r/T(r) = 1/2$

Example. Sampler for a binary tree with *about* N nodes.

```
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;
}
```

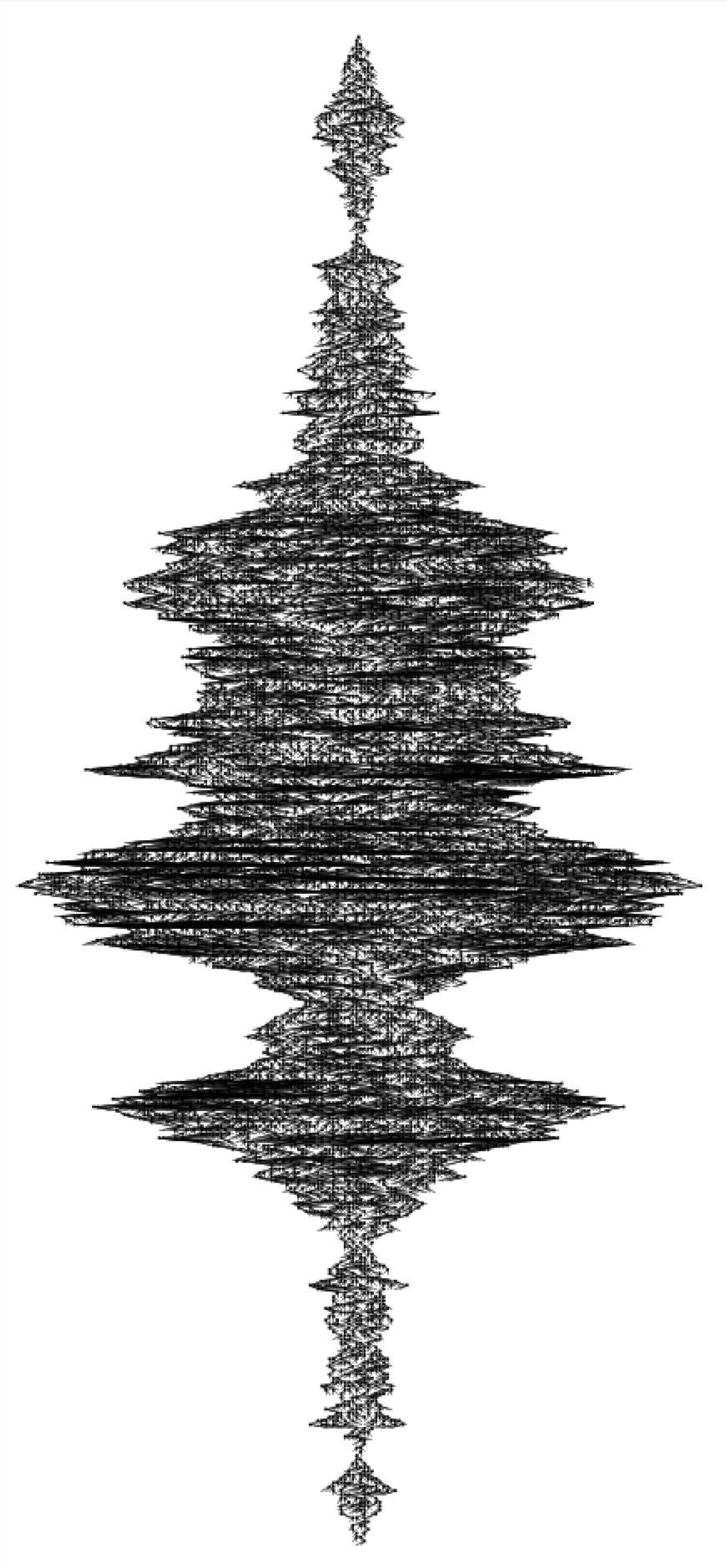
```
while ( CNT < 0.95*N || CNT > 1.05*N )
   { CNT = 0; t = generate(N); }
```

9972 nodes



25 trials

10182 nodes

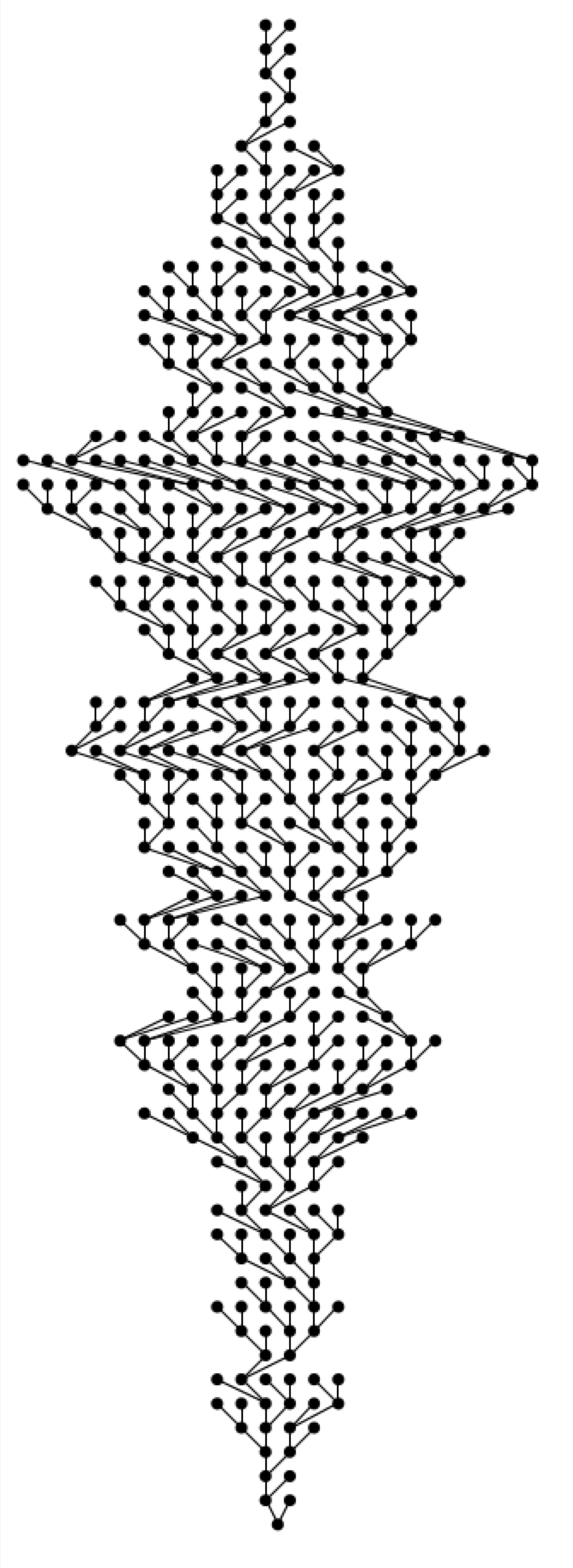


1239 trials

Important point. Easily extends to other types of trees and other classes.

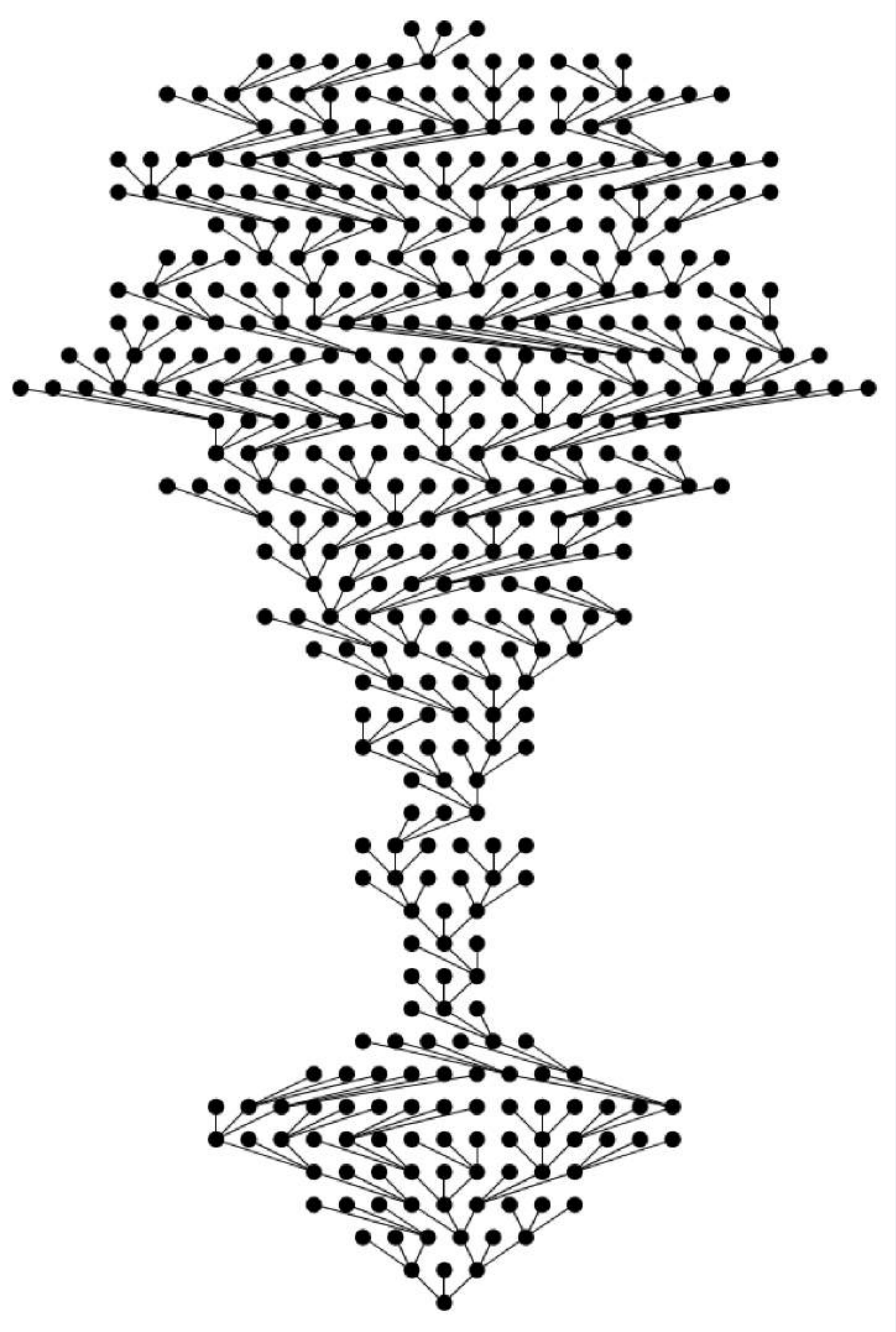
Four random trees with about 500 nodes

$p_0 = p_2 = 1/2$



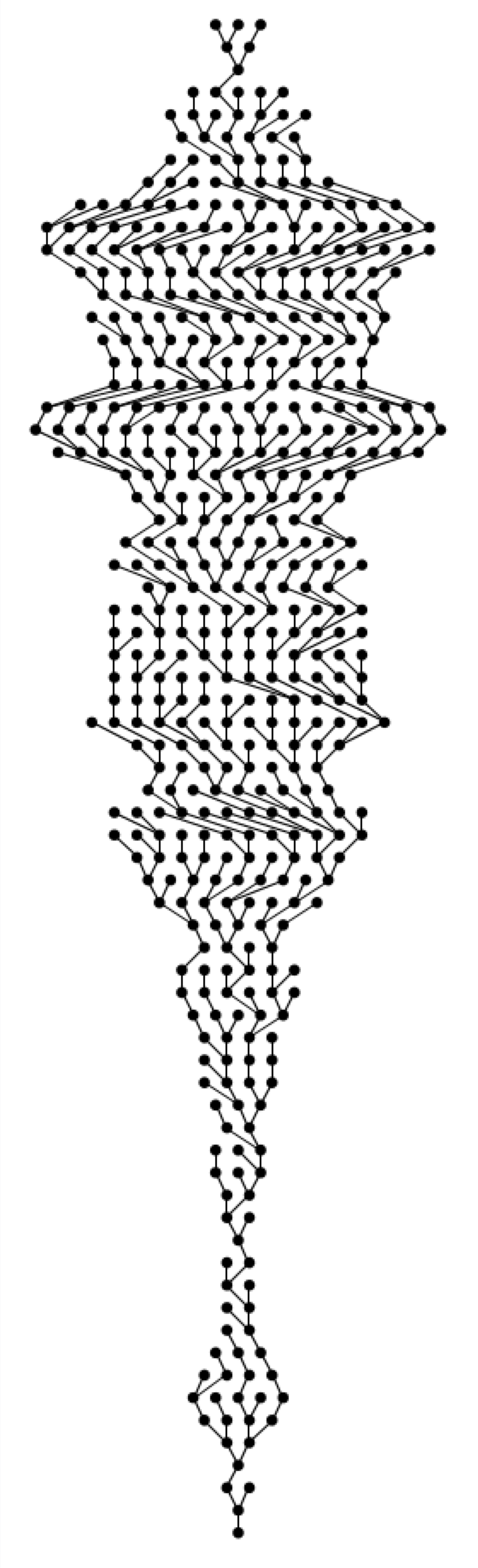
binary

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



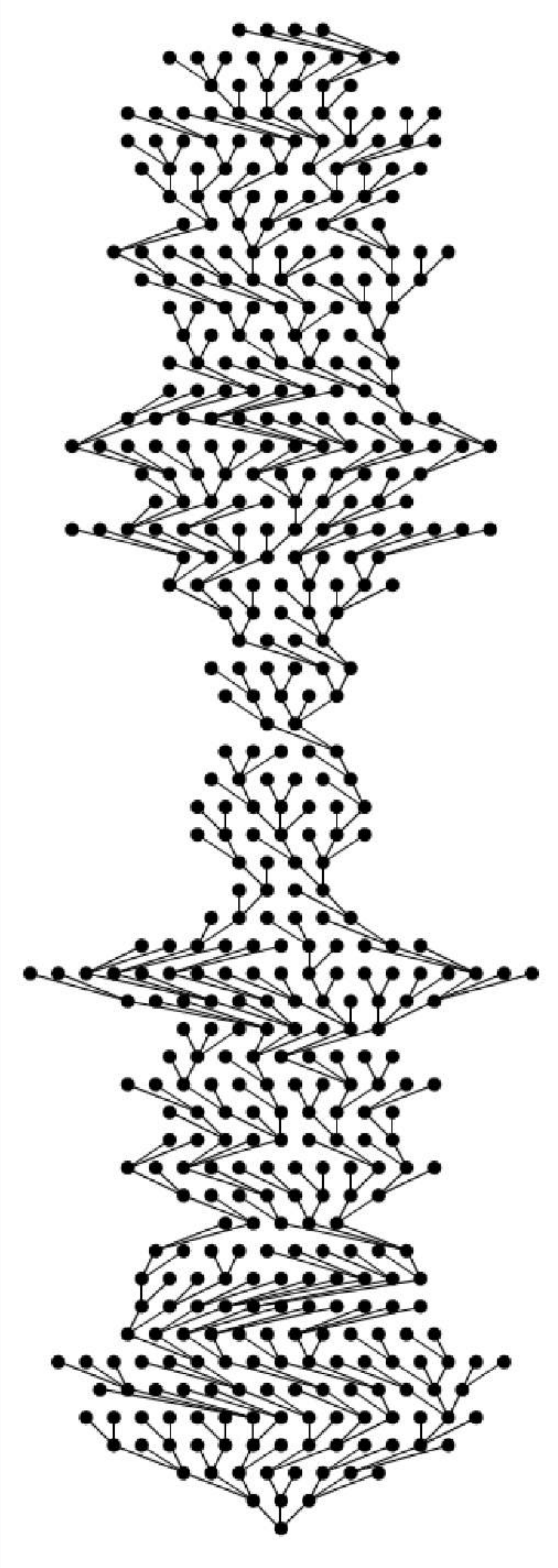
ternary

$p_0 = p_1 = p_2 = 1.0/3.0$



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

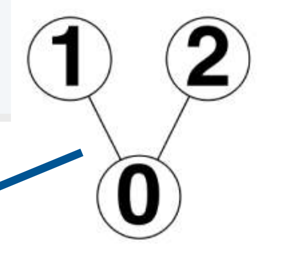
Idea. Implement a "*Galton-Watson process*".

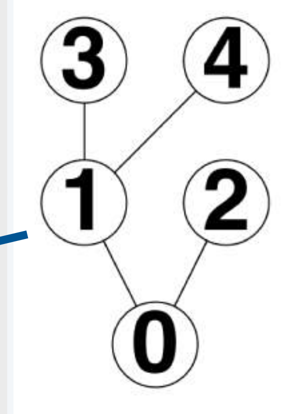
- Use parent-link representation
- i th entry on queue is parent of i
- Generate k children for each node w.p. p_k (need analytic combinatorics, in general)
- Use upper bounds and tolerance to terminate
- Example: $p_0 = p_2 = .5$ gives binary trees

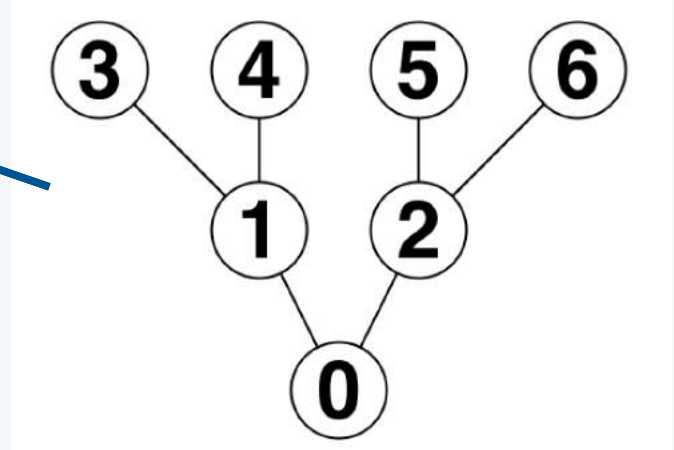
```
private static Queue generate(double[] p)
{
   Queue<Integer> tree = new Queue<Integer>();
   int root = 0;
   tree.enqueue(0);
   while (root < tree.size())
   {
      int k = StdRandom.discrete(p);
      for (int j = 1; j <= k; j++)
         tree.enqueue(root);
      root++;
   }
   return tree;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

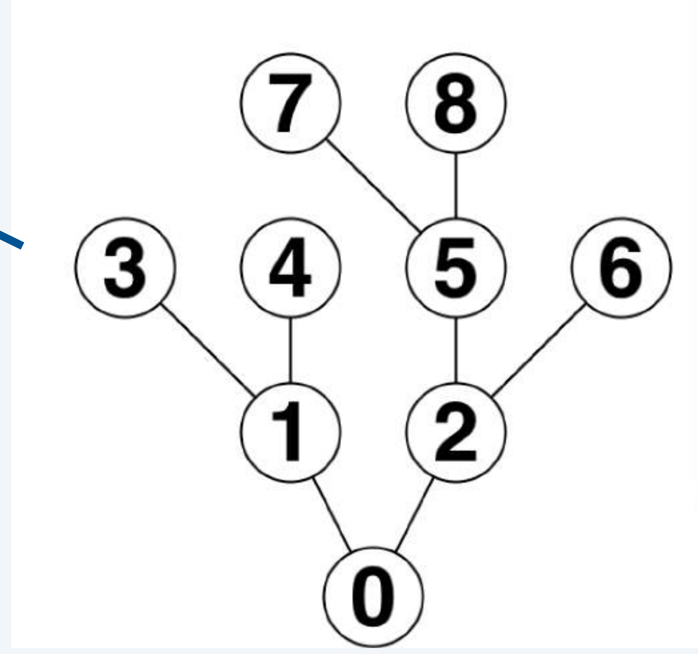
Confession: trees on previous slide generated with this code!

root	k	tree
0	2	0 1 2 0 0 0
1	2	0 1 2 3 4 0 0 0 1 1
2	2	0 1 2 3 4 5 6 0 0 0 1 1 2 2
3	0	0 1 2 3 4 5 6 0 0 0 1 1 2 2
4	0	0 1 2 3 4 5 6 0 0 0 1 1 2 2
5	2	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5
6	0	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5
7	0	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5
8	0	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5









Analytic samplers for labeled classes

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

	construction	*sampler*
neutral class	$\boldsymbol{E}$	`return ε`
atomic class	$\boldsymbol{Z}$	`return •`
disjoint union	$\boldsymbol{A = B + C}$	`u = StdRandom.bernoulli(B(x)/A(x))` `if (u) return B else return C`
labeled product	$\boldsymbol{A = B \star C}$	`return` *compose*`(B, C)`

sequence	$\boldsymbol{A = SEQ\,(B)}$	`k = geometric(B(x))`
		`return compose(B, B,…, B)`
		k independent instances

set	$\boldsymbol{A = SET\,(B)}$	`k = poisson(B(x))`
		`return compose(B, B,…, B)`
		k independent instances

cycle	$\boldsymbol{A = CYC\,(B)}$	`k = logseries(B(x))`
		`return compose(B, B,…, B)`
		k independent instances

notation

A	combinatorial class
a	object in A
$\lvert a \rvert$	size of a
$A(z)$	OGF for A
x_0	radius of convergence of $A(z)$
x	positive real $< x_0$

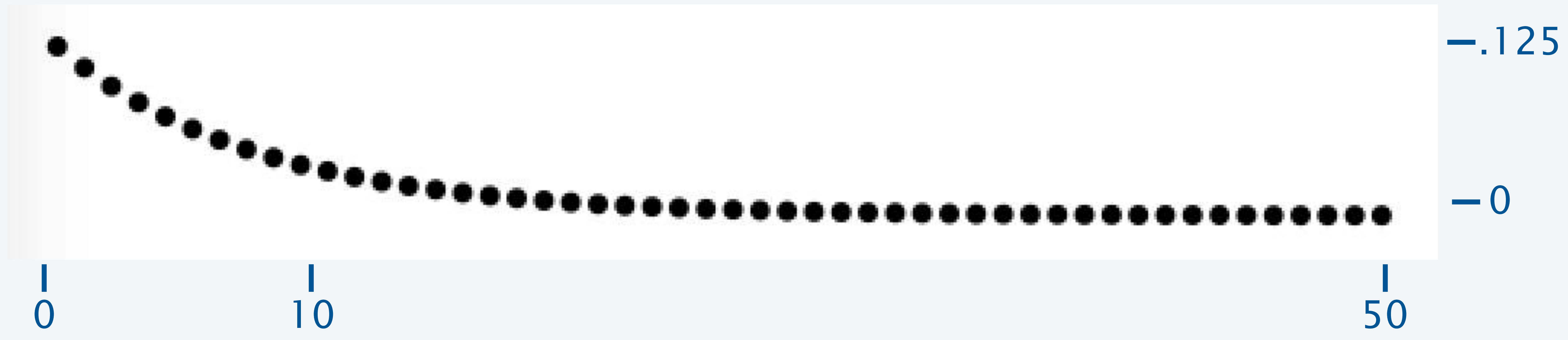
Note: Apply actual labels to the sampled structure (if needed) using a random permutation.

Distributions for labeled classes

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

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

λ = 0.875



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

```
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);
```

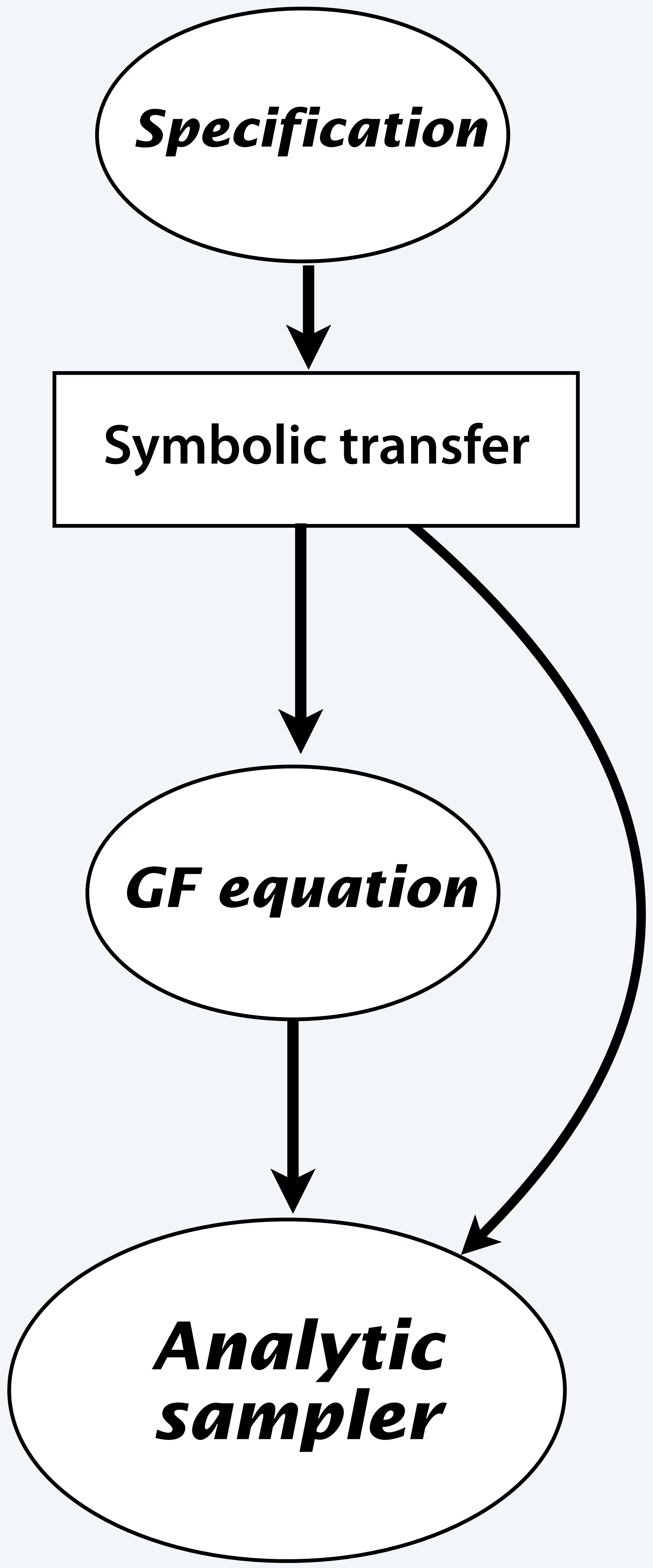
λ = 10

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

```
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);
```

λ = 0.999

Analytic sampler for sets of cycles (permutations)



***P* = *SET* (*CYC* (*Z*)**

$$P(z) = \exp\Big(\ln \frac{1}{1-z}\Big) = \frac{1}{1-z}$$

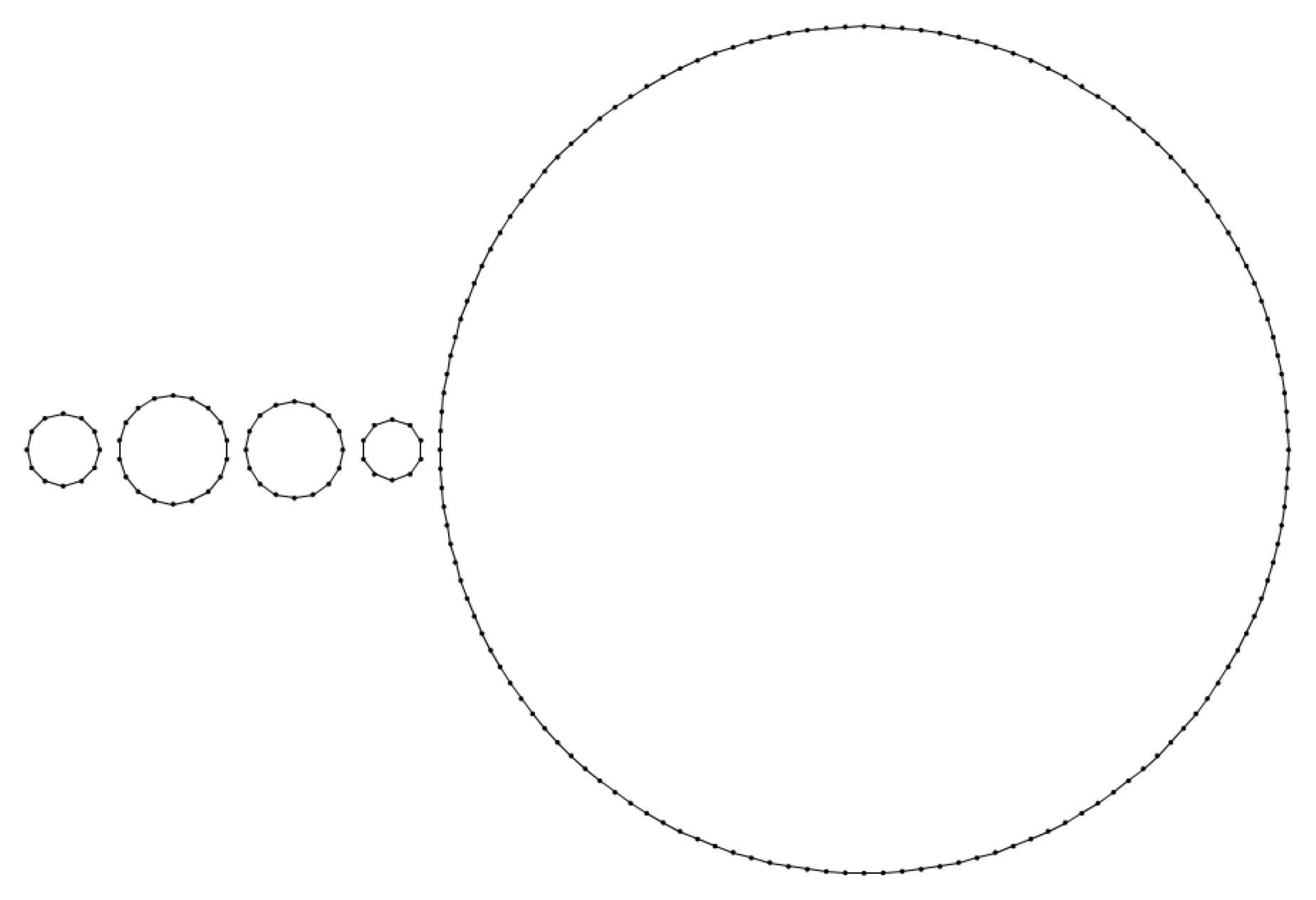
```
private static Queue generate(double r)
{
   int lambda = Math.log(1.0/(1.0 - r));
   int k = StdRandom.poisson(lambda);
   Queue<Integer> q = new Queue<Integer>();
   for (int i = 0; i < k; i++)
      q.enqueue(logseries(r));
   return q;
}
```

construction	*sampler*
Z	`return •`
***A* = *SET* (*B*)**	`k = poisson(B(x))` `return compose(B, B,…, B)`
***A* = *CYC* (*B*)**	`k = logseries(B(x))` `return compose(B, B,…, B)`

Next step for sets of cycles (permutations)

To generate a sample of expected size N

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



Expected size of a permutation

$$P(r) = \frac{1}{1-r} \qquad P'(r) = \frac{1}{(1-r)^2}$$

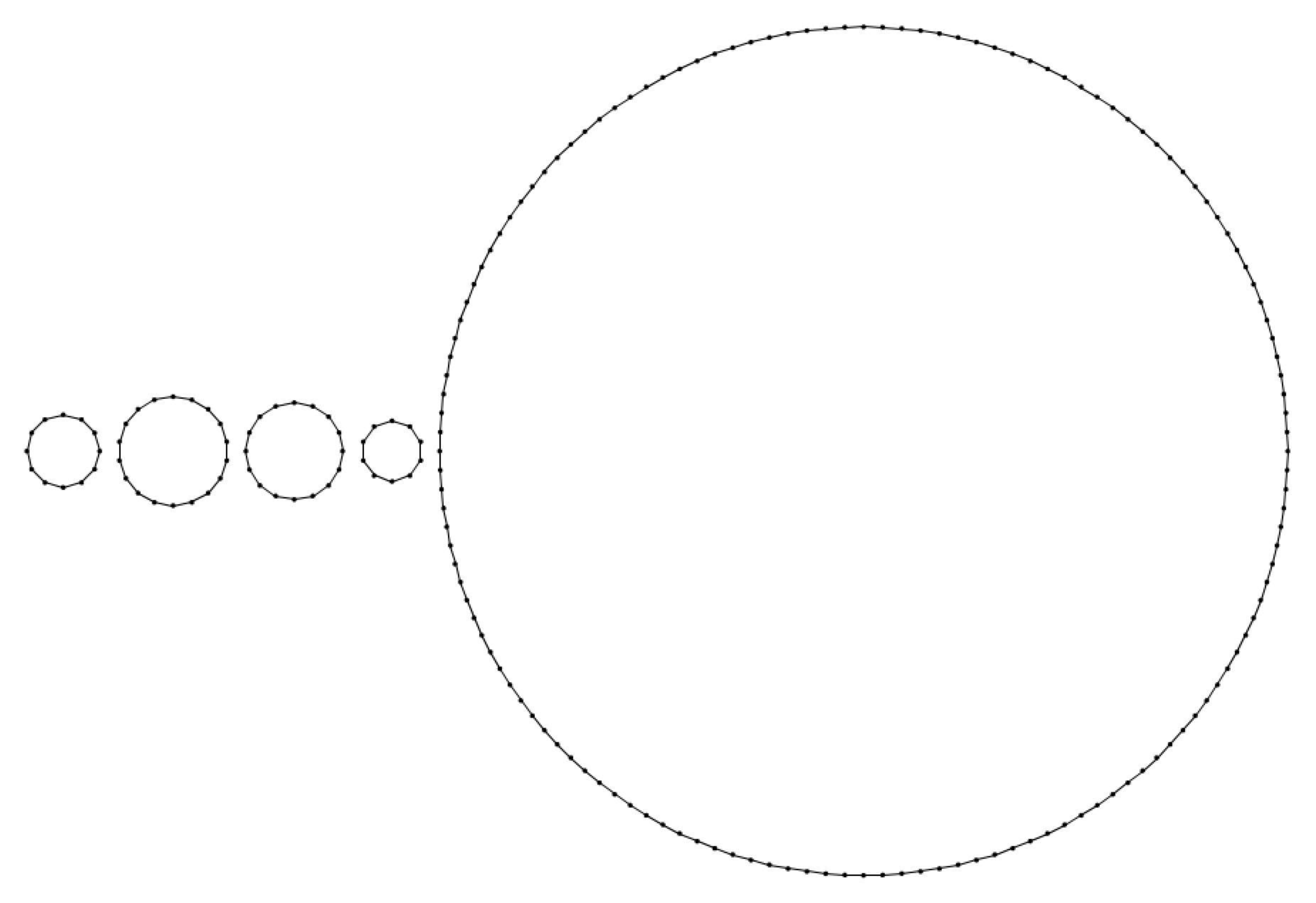
$$r\frac{P'(r)}{P(r)} = \frac{r}{1-r}$$

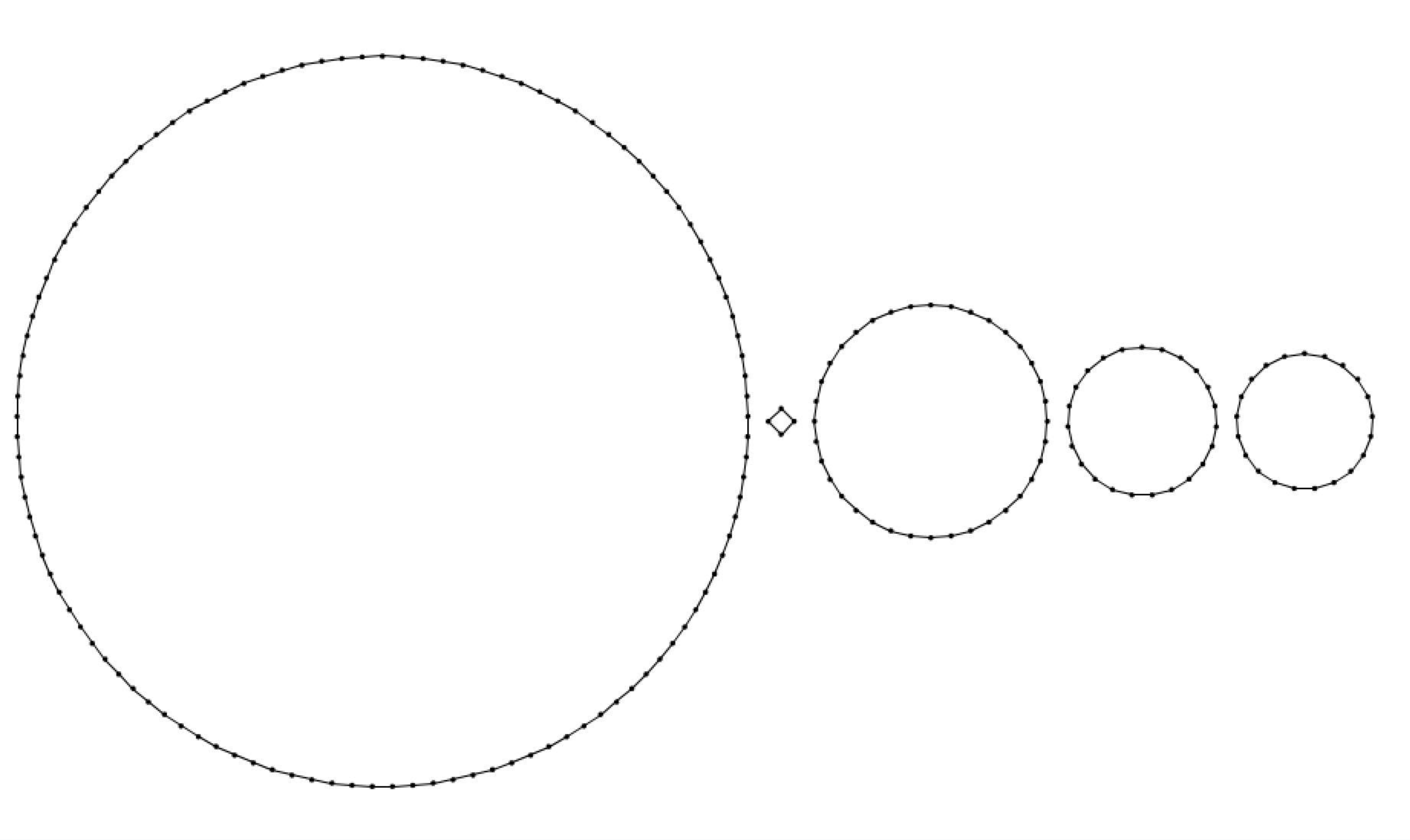
Value of r to expect a permutation of size *N*

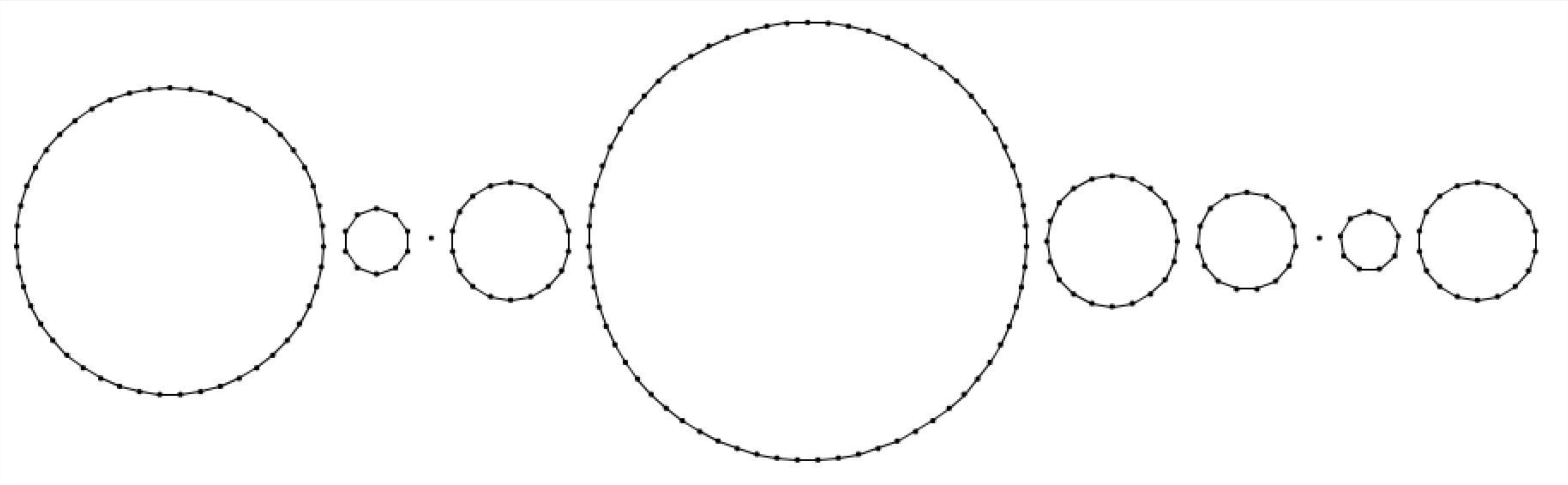
$$N = \frac{r}{1-r}$$

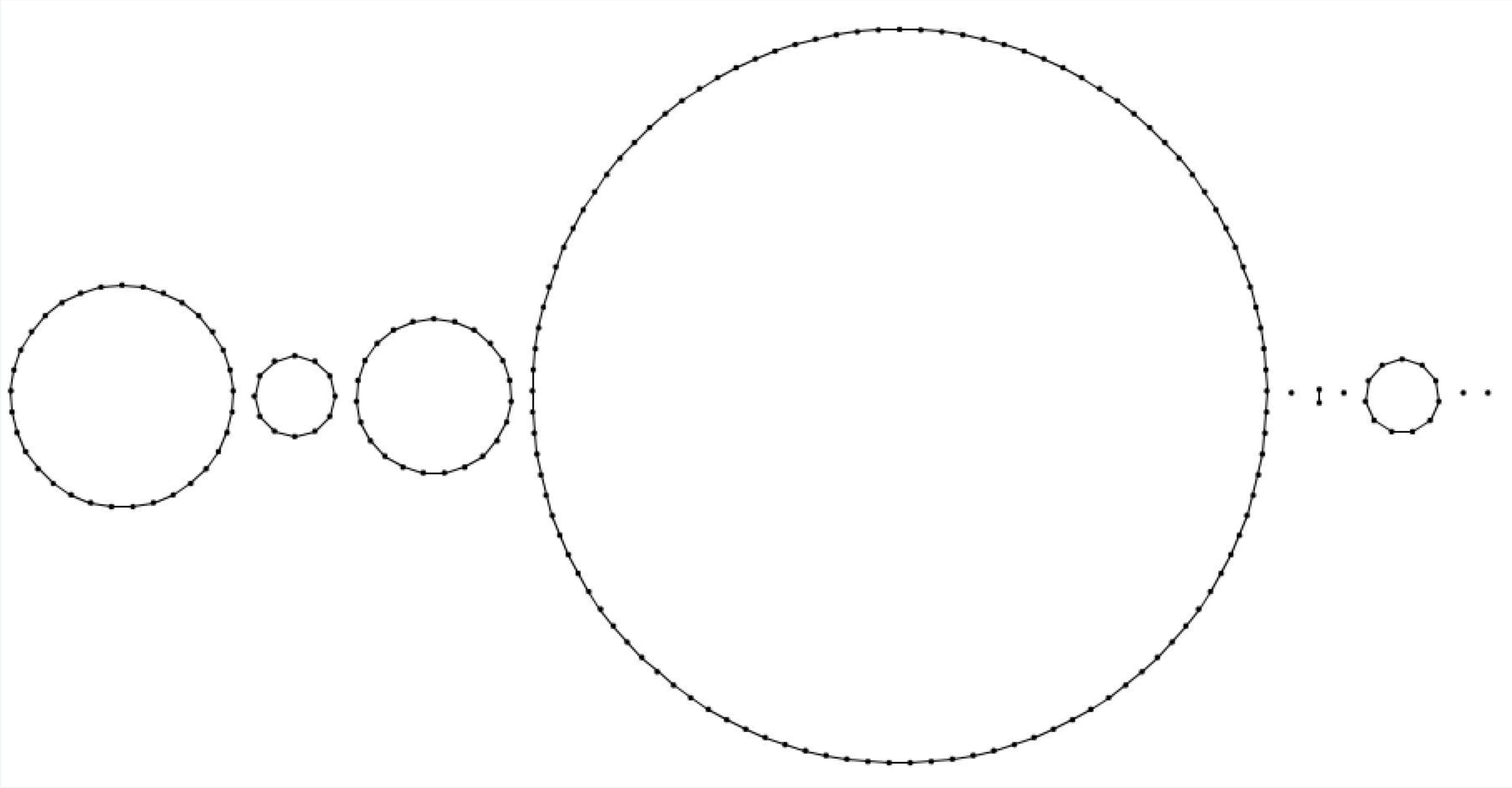
$$r = \frac{N}{N+1}$$

Four random sets of cycles (permutations) with about 200 nodes

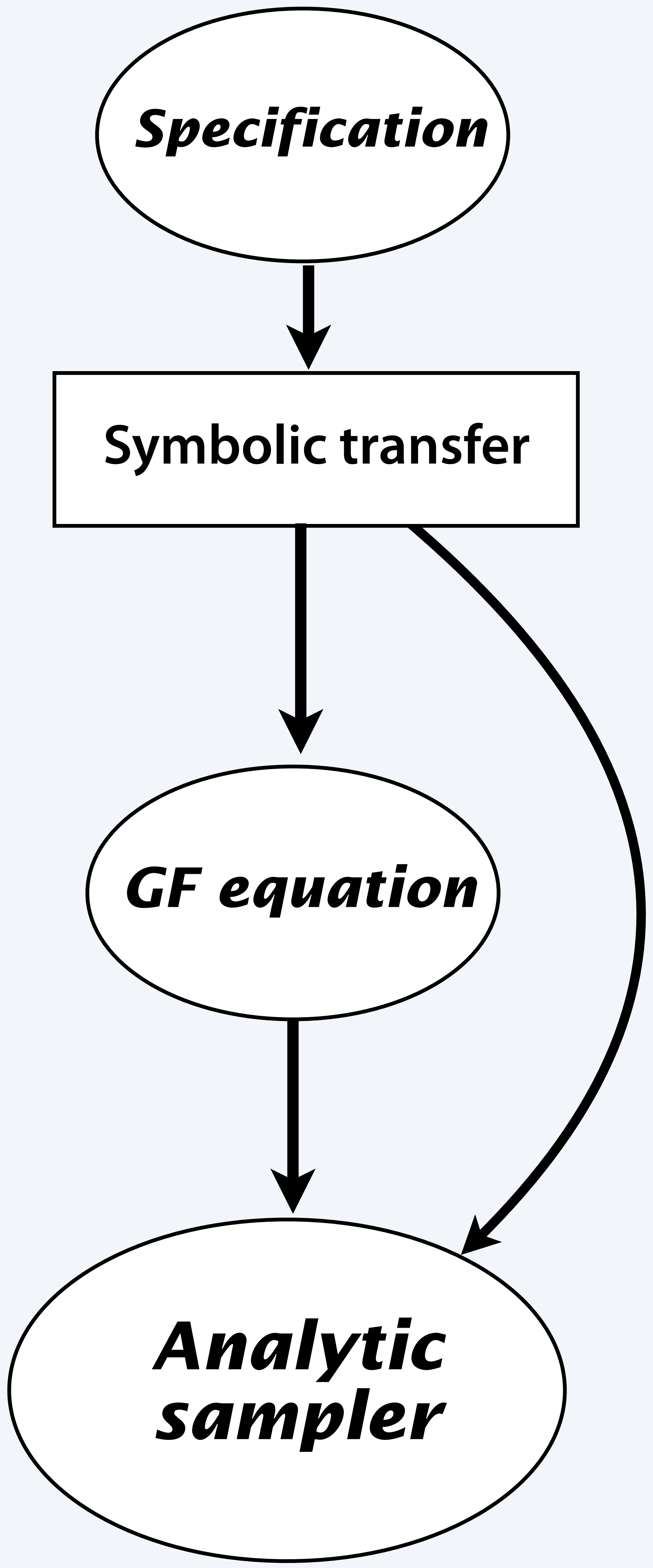








Analytic sampler for sets of cycles (permutations) with size restrictions



$$P = SET\,(CYC_{\Omega}\,(Z\,))$$

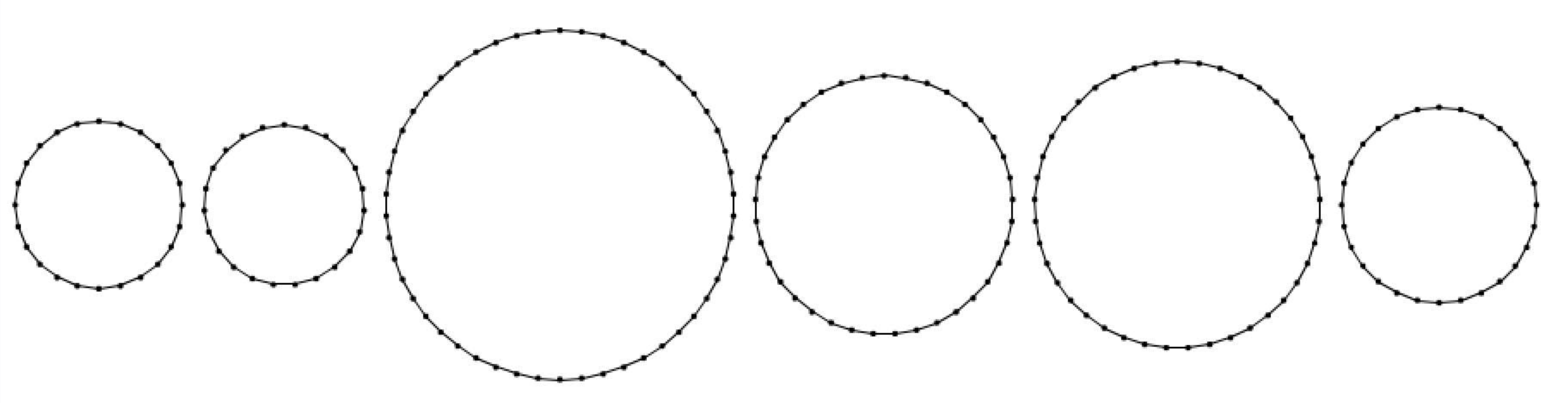
construction	sampler
Z	`return •`
$A = SET\,(B\,)$	`k = poisson(B(x))` `return compose(B, B,…, B)`
$A = CYC\,(B\,)$	`k = logseries(B(x))` `return compose(B, B,…, B)`

```
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;
}
```

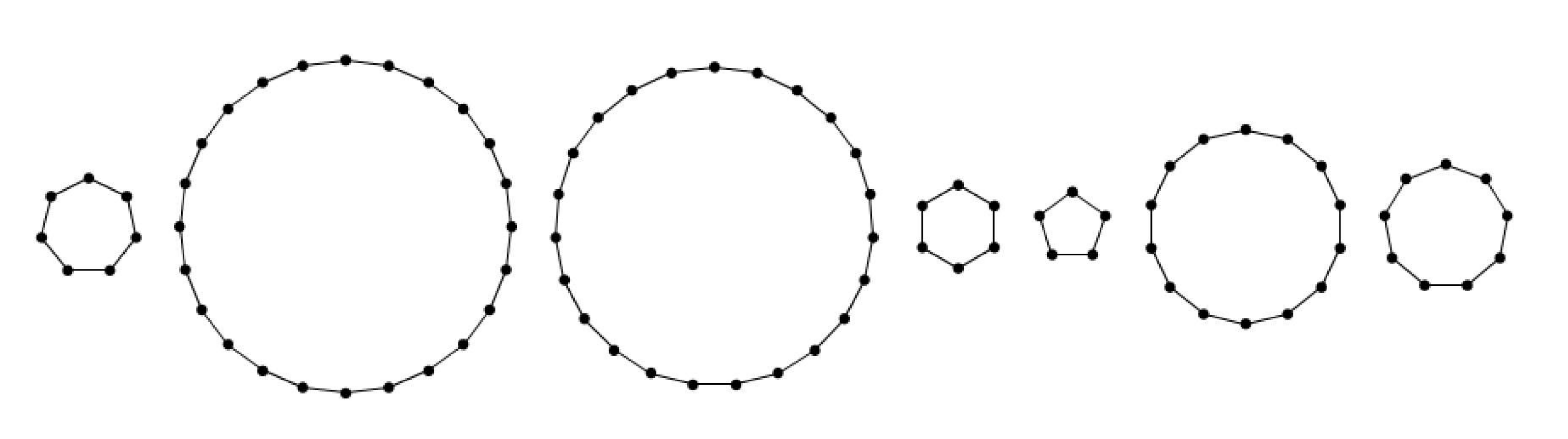
Note: Difficult to compute optimal value for r (works to use same value as for unrestricted case)

Four random sets of cycles with size restrictions

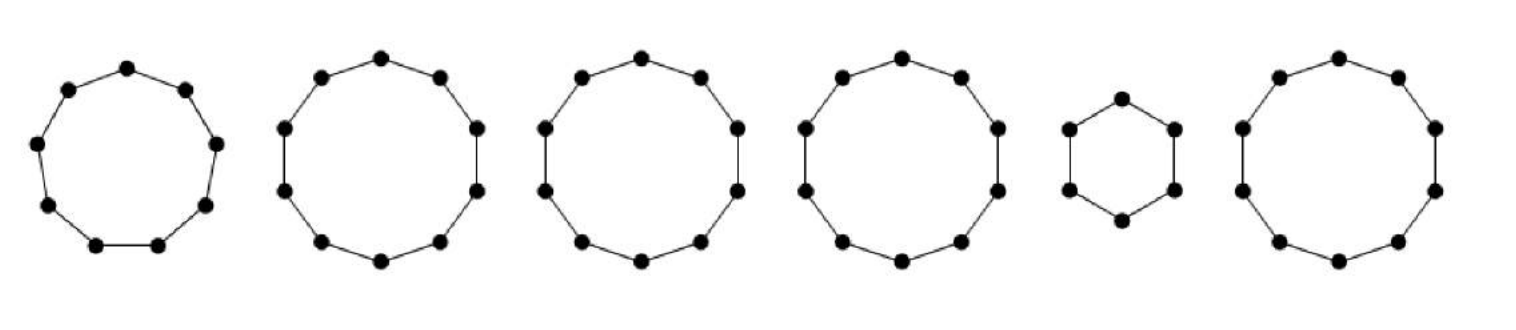
About 200 nodes, cycle lengths between 20 and 50



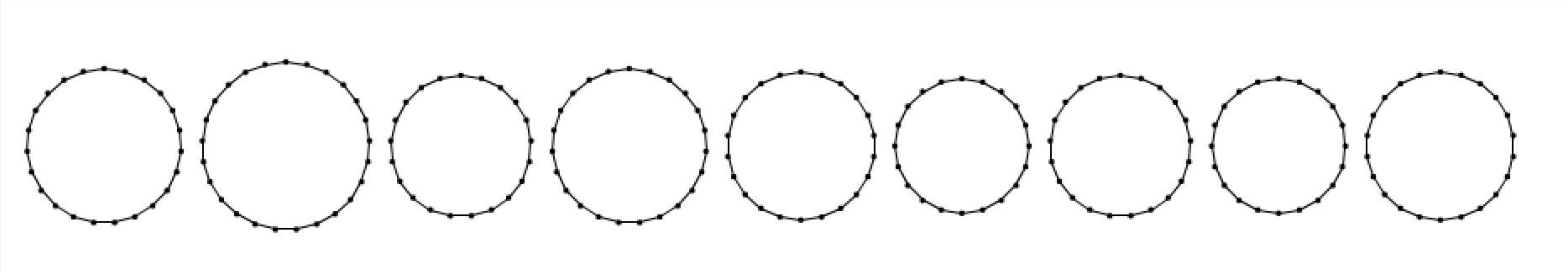
About 100 nodes, cycle lengths between 5 and 25



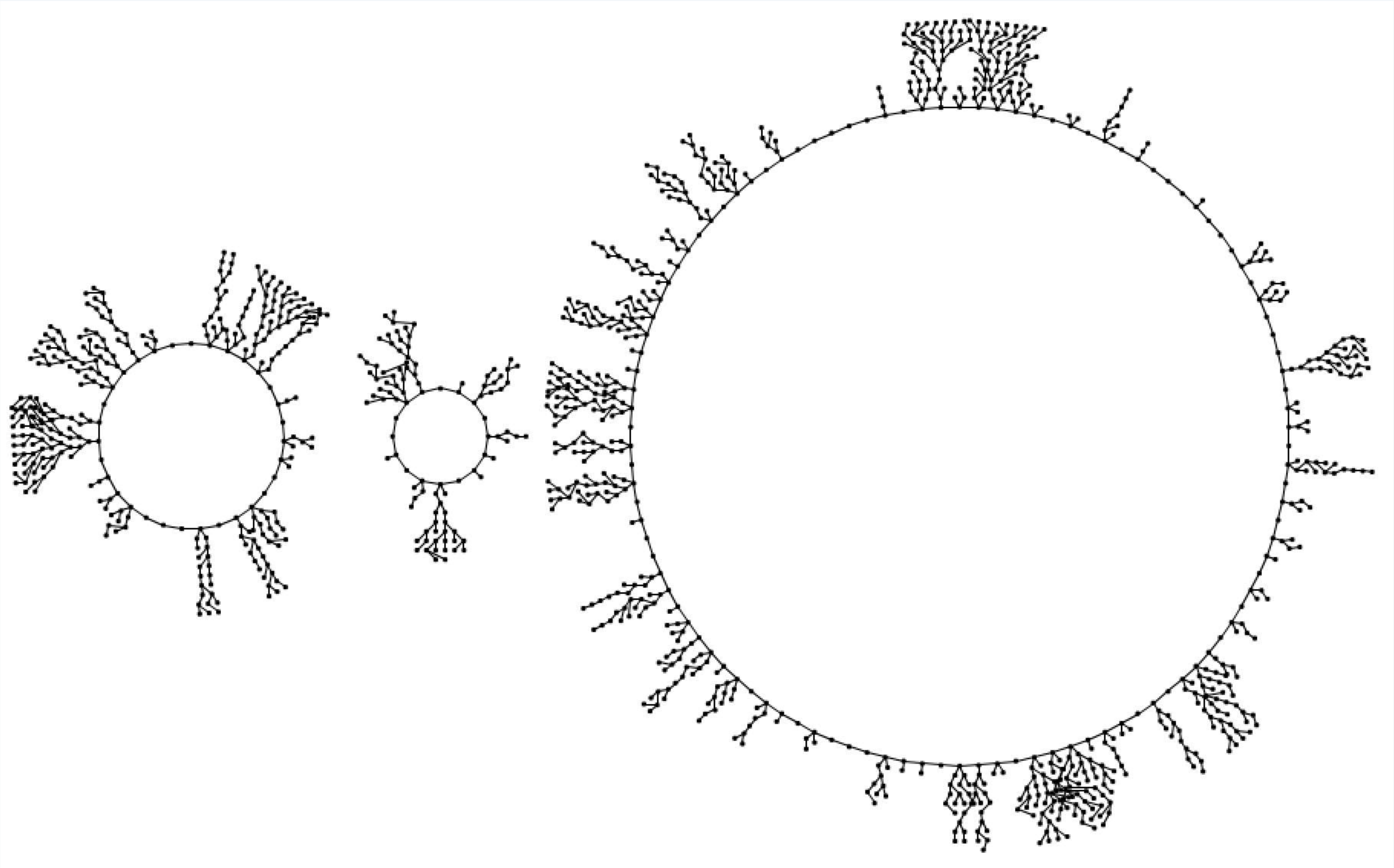
About 50 nodes, cycle lengths between 5 and 10

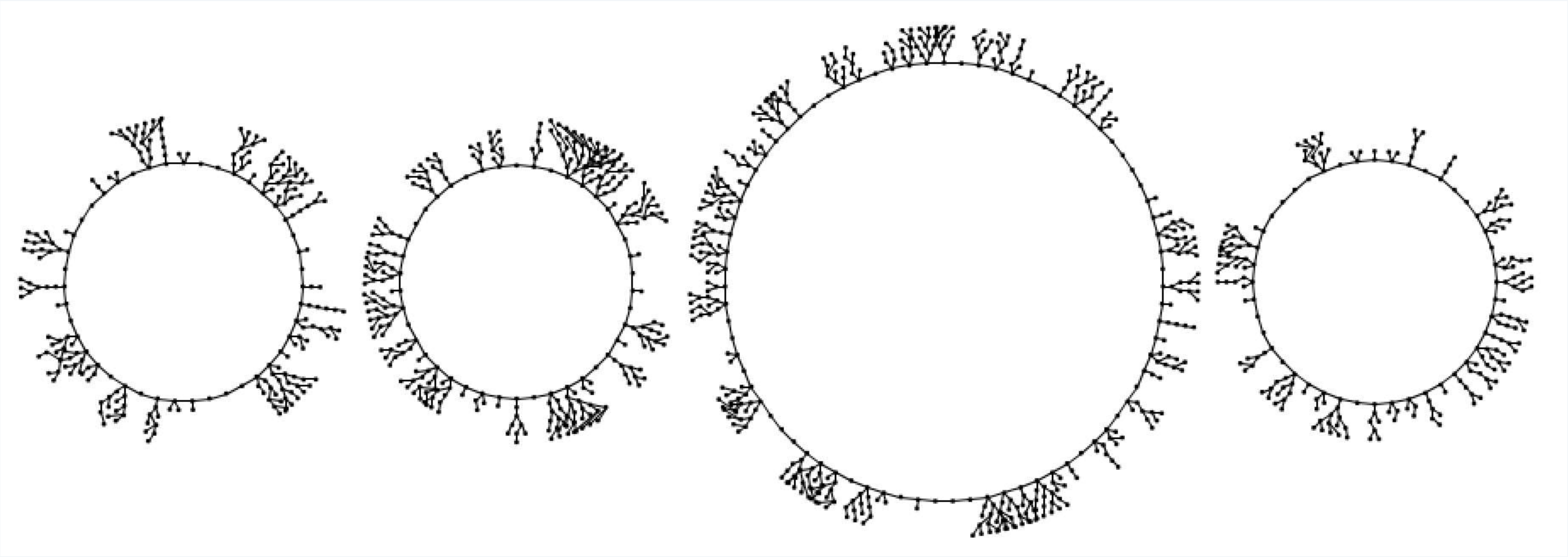


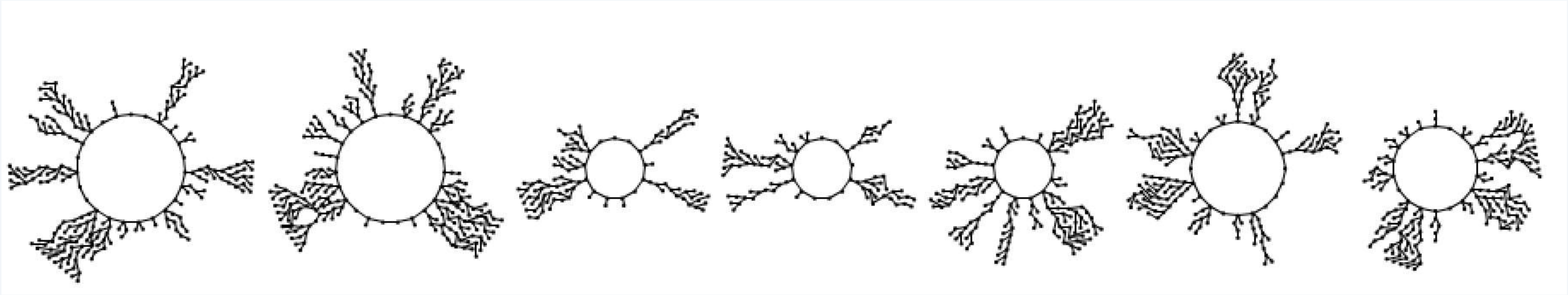
About 200 nodes, cycle lengths between 20 and 25



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)

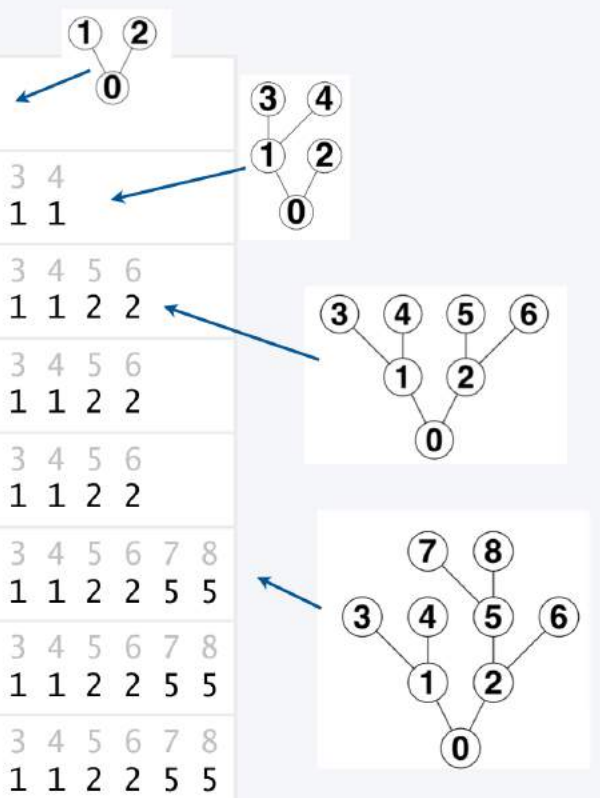
Another approach: iterative (breadth-first) singular analytic samplers for trees

Idea. Implement a *branching 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
- Ex. $p_0 = p_2 = .5$ gives binary trees

```
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;
}
```

root	k	tree
0	2	0 1 2 0 0 0
1	2	0 1 2 3 4 0 0 0 1 1
2	2	0 1 2 3 4 5 6 0 0 0 1 1 2 2
3	0	0 1 2 3 4 5 6 0 0 0 1 1 2 2
4	0	0 1 2 3 4 5 6 0 0 0 1 1 2 2
5	2	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5
6	0	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5
7	0	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5
8	0	0 1 2 3 4 5 6 7 8 0 0 0 1 1 2 2 5 5



48

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*).

```
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;
}
```

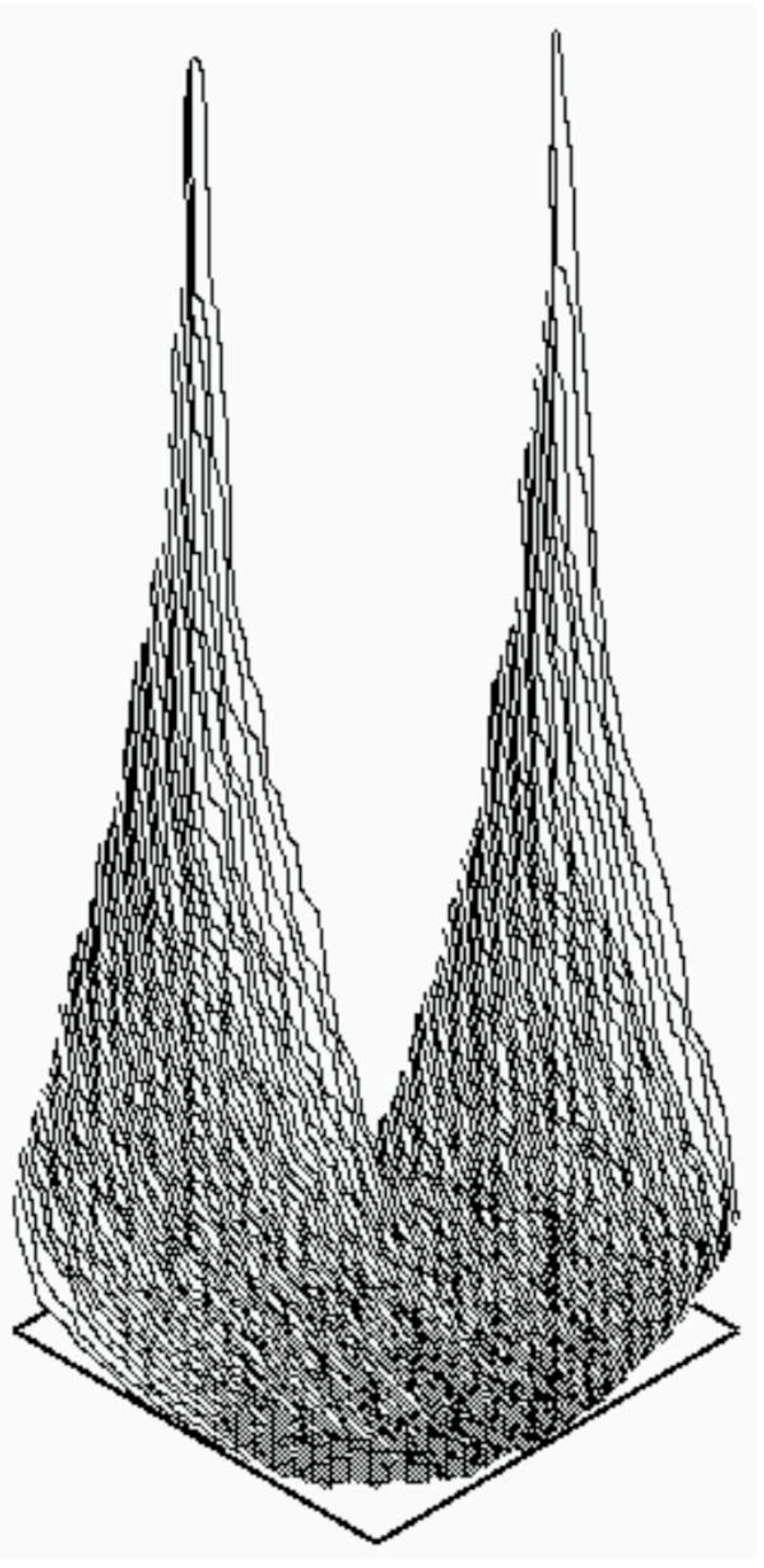
```
private Node generate(int N)
{
   double u = StdRandom.uniform();
   if (u < 1.0/(2.0 - 1.0/N))
      return new Node(0);

   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).

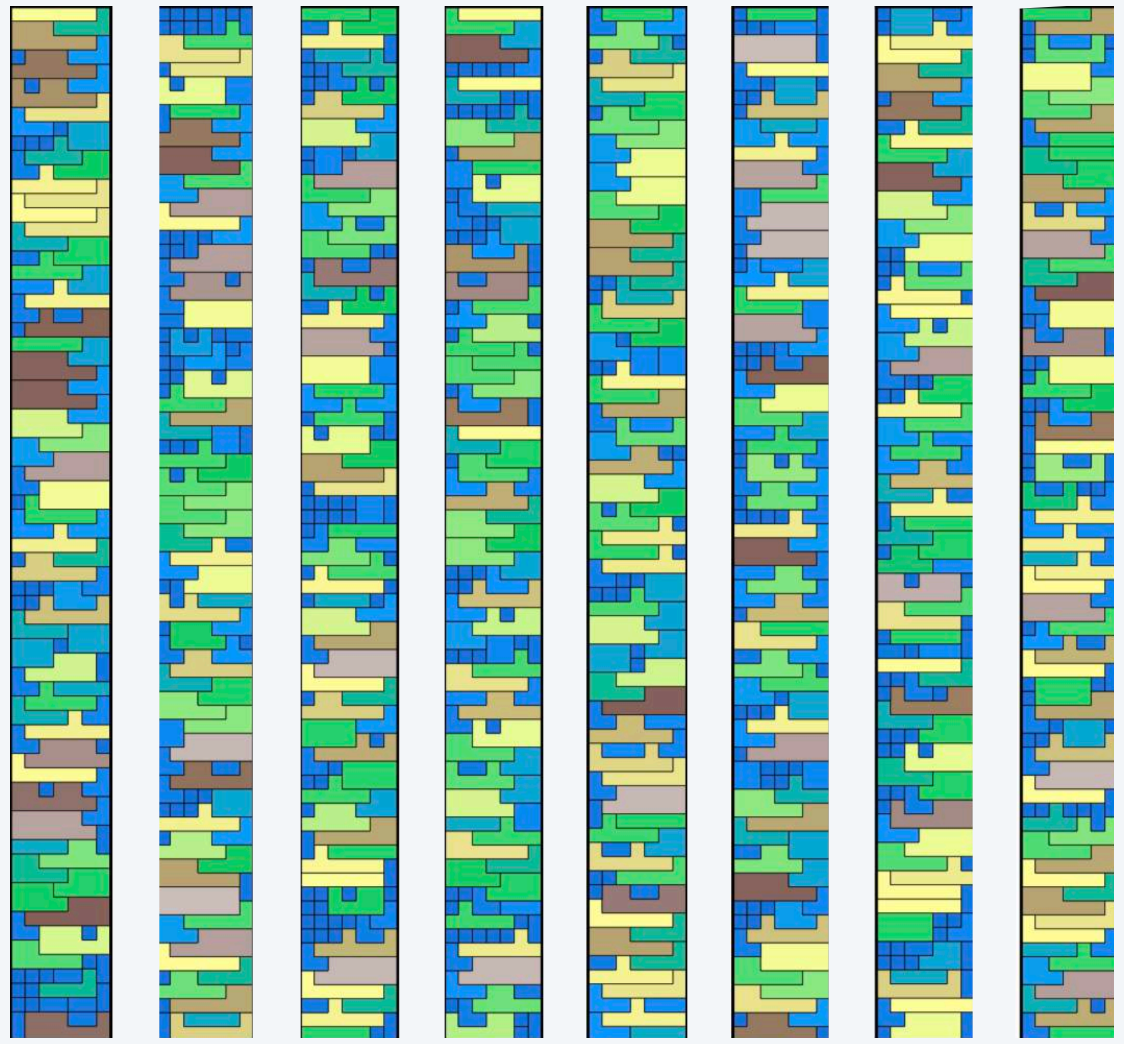
Various random objects produced by analytic samplers

Skew plane partition
Bodini, Fusy, and Pivoteau, 2006

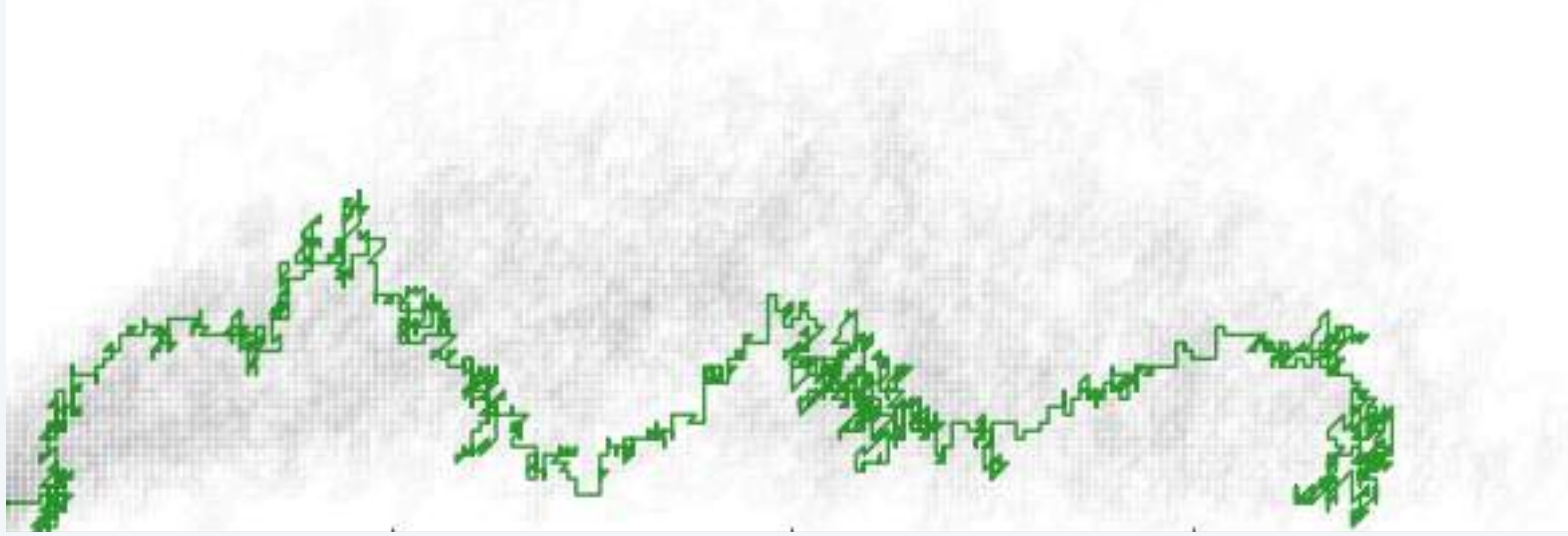


Cactus graph, Bahrani and Lumbroso, 2016

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.

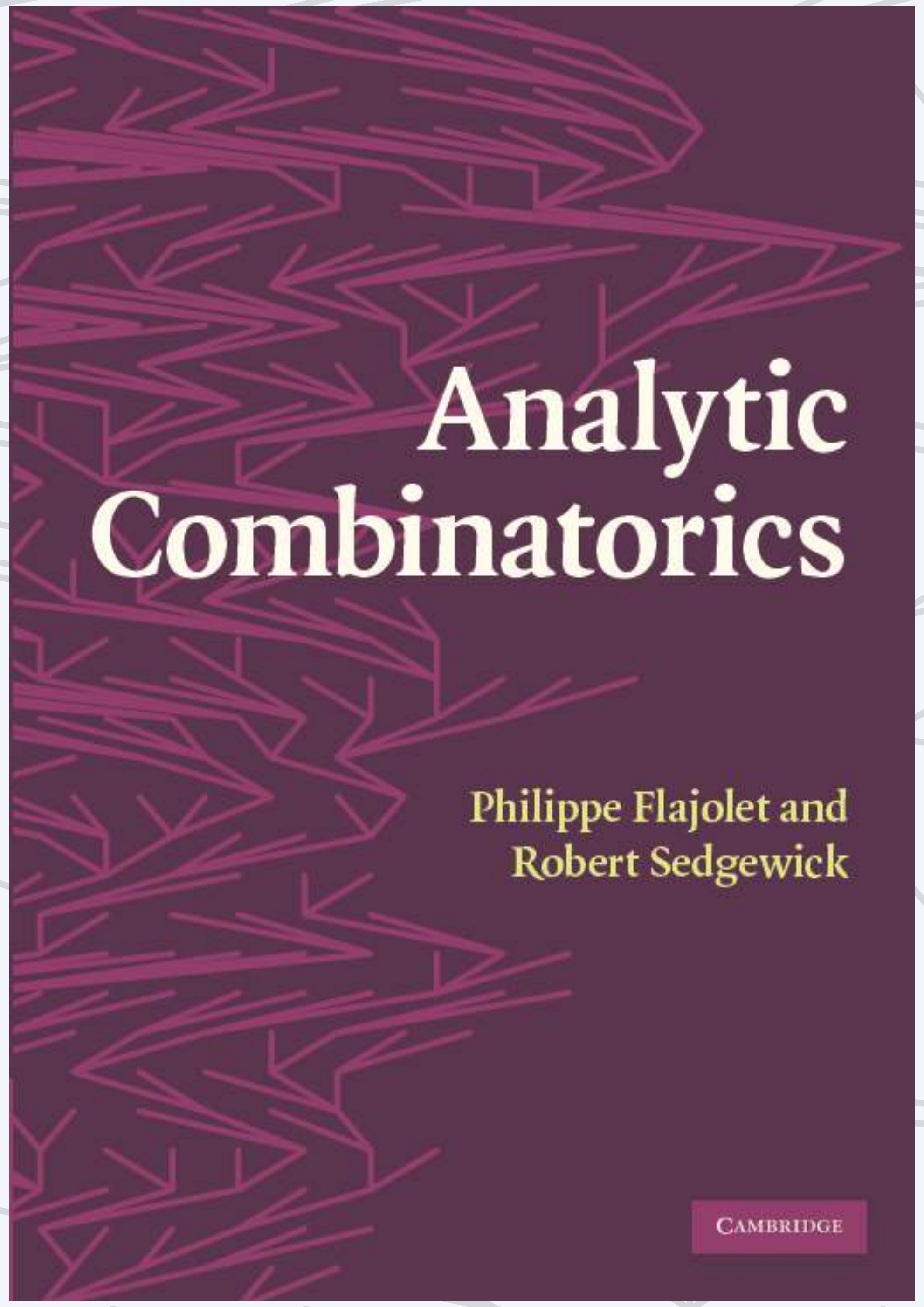
A scientific approach to discrete models

- 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.

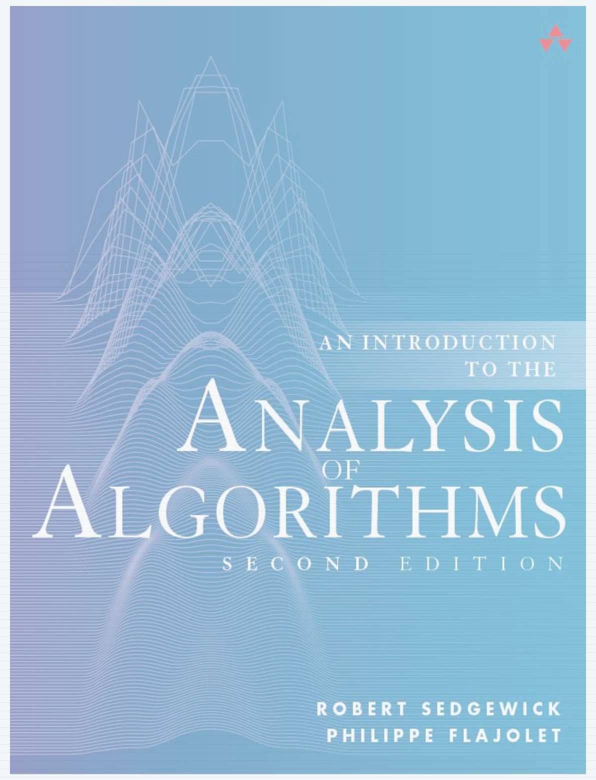
Random Sampling of Combinatorial Objects



http://ac.cs.princeton.edu

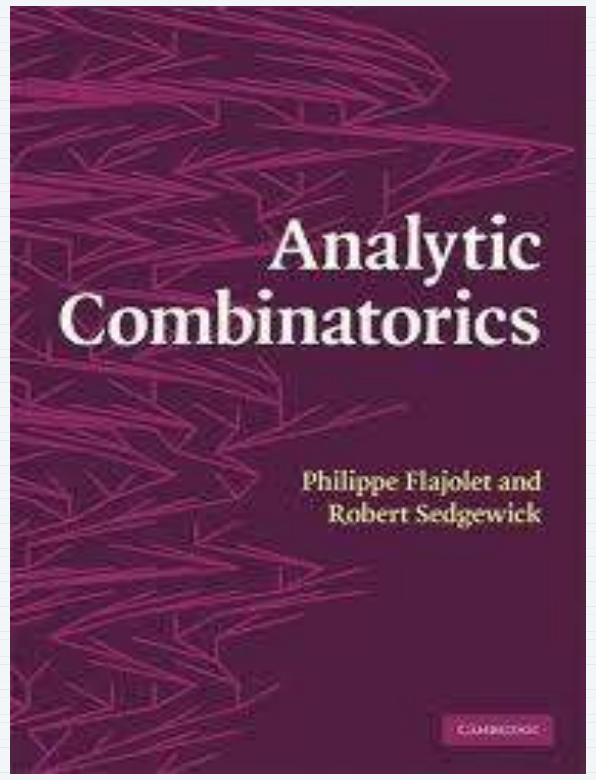
Robert Sedgewick
Princeton University

with special thanks to Jérémie Lumbroso



Analysis of Algorithms

Original MOOC title: ANALYTIC COMBINATORICS, PART ONE



Analytic Combinatorics

Original MOOC title: ANALYTIC COMBINATORICS, PART TWO

http://aofa.cs.princeton.edu

http://ac.cs.princeton.edu