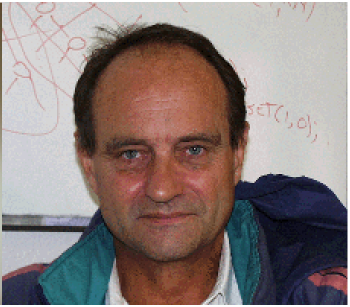
Random Generation with Philippe Flajolet

Philippe Flajolet and Analytic Combinatorics

Paul Zimmermann

December 15, 2011

I had Philippe as teacher of “petites classes d’informatique” at École Polytechnique in 1985-1987.



At that time Patrick Cousot gave the main lectures (in Pascal) and in 2nd year Jean Vuillemin (Le Lisp)

At that time we worked with Macintosh'es

Archivio Composizione Esposizione Strumenti
The Macintosh™ Finder
Bruce Horn and Steve Capps
512K
Version 4.1 ©1985 Apple Computer

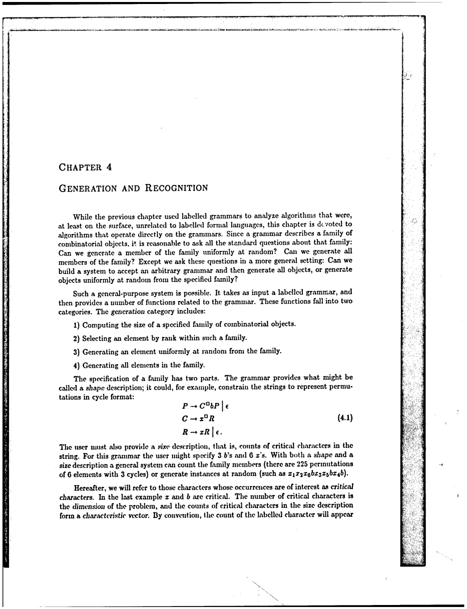
which enabled us to connect to a Vax 8600:

and learn a few Unix commands:

```
% ls -lR /
...
% man yacc
...
% cd .
...
% ftp prep.ai.mit.edu
...
% pc toto.p; ./a.out
...
% cat > a.out << EOF; ./a.out; /bin/rm a.out
...
```

I did a PhD under the supervision of Philippe in 1988-1991 (*Séries génératrices et analyse automatique d'algorithmes*).

During my PhD, Philippe showed me the thesis of Daniel H. Greene (*Labelled formal languages and their uses*)



CHAPTER 4

GENERATION AND RECOGNITION

While the previous chapter used labelled grammars to analyze algorithms that were, at least on the surface, unrelated to labelled formal languages, this chapter is devoted to algorithms that operate directly on the grammars. Since a grammar describes a family of combinatorial objects, it is reasonable to ask all the standard questions about that family: Can we generate a member of the family uniformly at random? Can we generate all members of the family? Except we ask these questions in a more general setting: Can we build a system to accept an arbitrary grammar and then generate all objects, or generate objects uniformly at random from the specified family?

Such a general-purpose system is possible. It takes as input a labelled grammar, and then provides a number of functions related to the grammar. These functions fall into two categories. The *generation* category includes:

1) Computing the size of a specified family of combinatorial objects.

2) Selecting an element by rank within such a family.

3) Generating an element uniformly at random from the family.

4) Generating all elements in the family.

The specification of a family has two parts. The grammar provides what might be called a *shape* description; it could, for example, constrain the strings to represent permutations in cycle format:

$$\begin{aligned} P &\to C^{\square} b P \mid \epsilon \\ C &\to x^{\square} R \\ R &\to x R \mid \epsilon . \end{aligned} \qquad (4.1)$$

The user must also provide a *size* description, that is, counts of critical characters in the string. For this grammar the user might specify 3 b's and 6 x's. With both a *shape* and a *size* description a general system can count the family members (there are 225 permutations of 6 elements with 3 cycles) or generate instances at random (such as $x_1x_2x_6bx_3x_5bx_4b$).

Hereafter, we will refer to those characters whose occurrences are of interest as *critical characters*. In the last example x and b are critical. The number of critical characters is the *dimension* of the problem, and the counts of critical characters in the size description form a *characteristic vector*. By convention, the count of the labelled character will appear

Appendix C of Greene's thesis was particularly interesting: *A General-Purpose Generator of Combinatorial Elements*.

One day, Bernard van Cutsem (who was doing research in statistics) asked Philippe how to generate so-called *hierarchies* at random. This led to the *recursive method* which I will try to describe in detail, and to the *boustrophedon* algorithm.

The Recursive Method

[113] *A Calculus of Random Generation*, Ph. Flajolet, P. Zimmermann and B. Van Cutsem, European Symposium on Algorithms, 1993.

[119] *A Calculus for the Random Generation of Labelled Combinatorial Structures*, Ph. Flajolet, P. Zimmermann and B. Van Cutsem, Theoretical Computer Science, 1994.

Later Philippe was interested in faster random generation of objects of size *near* n [cf MS talk].

Combstruct: an example

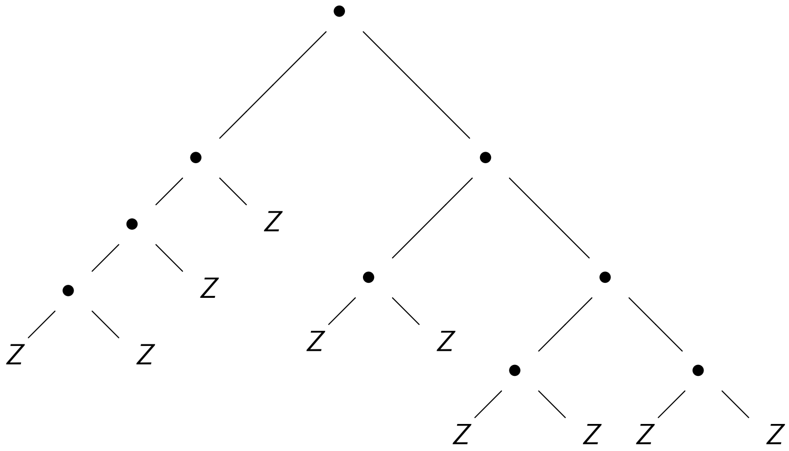
```
> with(combstruct):
> bin := {B=Union(Z, Prod(B,B))}:
> seq(count([B, bin, unlabelled], size=n), n=1..19);
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,

    742900, 2674440, 9694845, 35357670, 129644790, 477638700

> draw([B, bin, unlabelled], size=10);
Prod(Prod(Prod(Prod(Z, Z), Z), Z),

    Prod(Prod(Z, Z), Prod(Prod(Z, Z), Prod(Z, Z))))
```

Of course you have recognized



The Recursive Method in a few slides

Applies to *decomposable* data structures (idem $\Lambda\Upsilon\Omega$):

Atomic objects: `Epsilon` (size 0) and `Z` (size 1).

Constructors: `Union`, `Product`, `Sequence`, `Set`, `Cycle`, ...

`A = Prod(Z, Set(A))`	non plane trees
`B = Union(Z, Prod(B,B))`	plane binary trees
`C = Prod(Z, Seq(C))`	plane general trees
`D = Set(Cycle(Z))`	permutations
`E = Set(Cycle(A))`	functional graphs
`F = Set(Set(Z,card>=1))`	set partitions
`H = Union(Z,Set(H,card>=2))`	hierarchies
`M = Seq(Set(Z,card>=1))`	surjections

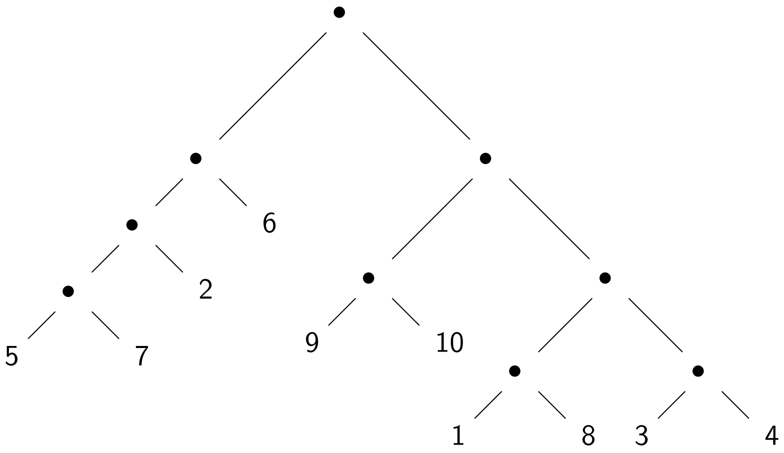
Folk theorem of combinatorial analysis: the specification can be turned out into a set of equations for the generating functions.

Hierarchies: `H = Union(Z,Set(H,card>=2))`

$$H(z) = z + \exp(H(z)) - 1 - H(z)$$

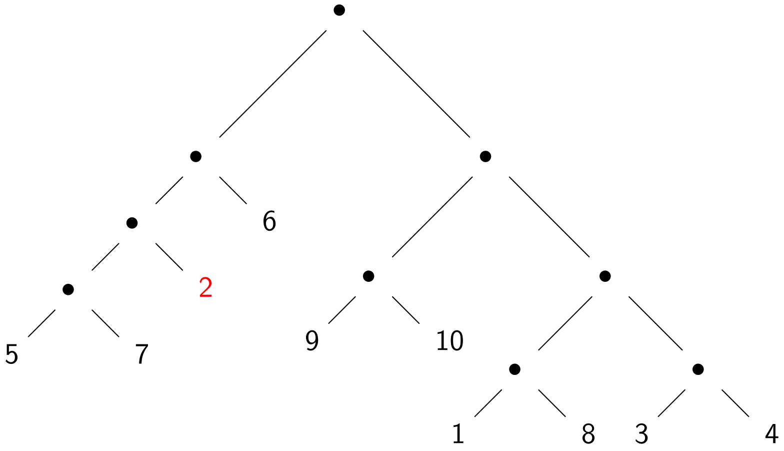
Corollary: the counting sequences up to size n can be computed in $O(n^2)$ arithmetic operations.

There are $4862 \cdot 10!$ labelled plane binary trees:



The pointing operator Θ [cf Basile Morcrette's talk]

There are $4862 \cdot 10! \cdot 10$ *pointed* labelled plane binary trees:



Footnote 2 of [119]: *An interesting outcome of this idea [the pointing operator] is the combinatorial differential calculus of Leroux and Viennot, see for instance [26].*

Standard Specifications

$$C = \Theta A \qquad \Longrightarrow \qquad C(z) = z\frac{d}{dz}A(z)$$

Example: `A = Prod(Z, Set(A))` (non plane trees)

$$A(z) = z \cdot B(z), \quad \Theta B(z) = B(z) \cdot \Theta A(z)$$

From standard specifications to counting

$C = Z$:

```
gC := proc(n) if n=1 then Z else error() fi end
```

$C = A + B$:

```
gC := proc(n) u := rand(1..C(n)); if u <= A(n) then
gA(n) else gB(n) fi end
```

$C = A \cdot B$:

```
gC := proc(n) u := rand(1..C(n)); for k from 0 to n
do t := A(k)*B(n-k); if u <= t then [gA(k),gB(n-k)]
else u := u - t fi od end
```

$C = \Theta A$:

```
gC := proc(n) point(gA(n), rand(1..n)) end
```

$\Theta C = A$:

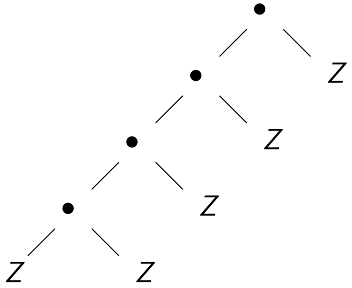
```
gC := proc(n) unpoint(gA(n)) end
```

The Boustrophedon Algorithm

The main cost comes from the product operation, when k goes up to n (or $n-1$, $n-2$, ...):

$C = A \cdot B$:

```
gC := proc(n) u:=rand(1..C(n)); for k in [0...n] do
u:=u-A(k)*B(n-k); if u<=0 then
return([gA(k),gB(n-k)]) fi od end
```

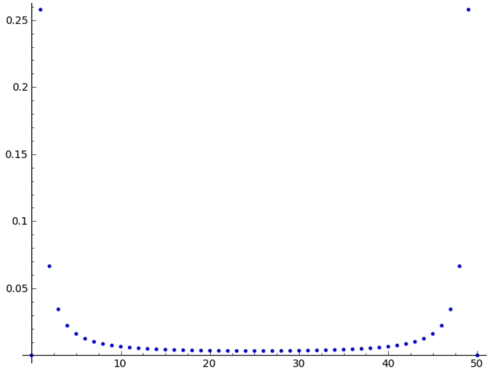


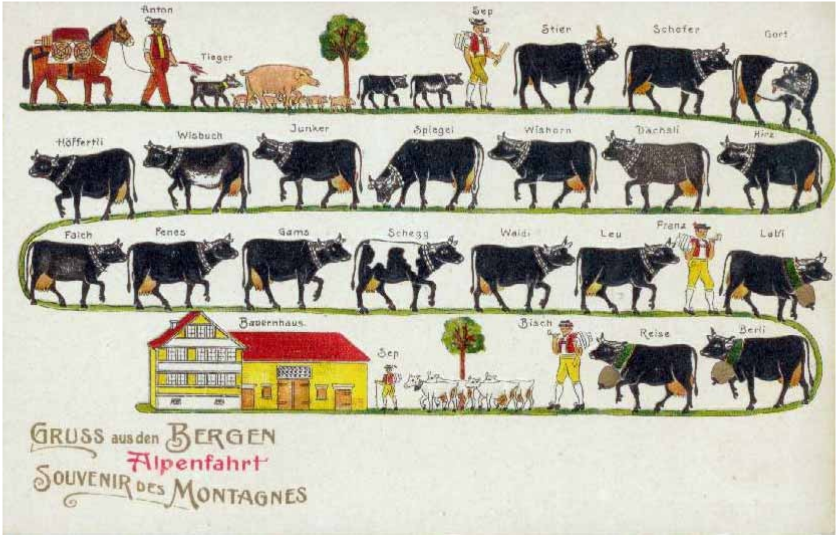
$$f(n) = \max_k [k + f(k) + f(n-k)] \quad \Longrightarrow \quad f(n) = \Theta(n^2)$$

```
@CachedFunction
def B(n):
   if n==1: return 1
   else: return sum([B(k)*B(n-k) for k in [1..n-1]])

def Pr(n,k):
   return 1.0*B(k)*B(n-k)/B(n)

sage: list_plot([Pr(50,k) for k in [0..50]])
```




Anton
Tieger
Sep
Stier
Schefer
Gort
Höffertli
Wisbuch
Junker
Spiegel
Wishorn
Dachsli
Hirz
Falch
Fenes
Gams
Schegg
Waldi
Leu
Franz
Lobi
Bauernhaus
Sep
Bisch
Reise
Berli
GRUSS aus den BERGEN
Alpenfahrt
SOUVENIR DES MONTAGNES

The Boustrophedon Idea

$C = A \cdot B$:

```
gC:=proc(n) u:=rand(1..C(n)); for k in [0,n,1,n-1,...]
do u:=u-A(k)*B(n-k); if u<=0 then
return([gA(k),gB(n-k)]) fi od end
```

$$f(n) = \max_k [\min(k, n-k) + f(k) + f(n-k)]$$

$$f(n) = \frac{1}{2\log 2} n \log n + O(n)$$

Philippe's influence on my research

- doing experiments to confirm or discover theories
- using and extending computer algebra: Maple, MuPAD, Sage, ...
- rigorous floating-point computations: IEEE 754, GNU MPFR, ...

A few more stories about Philippe

- Philippe before a talk
- Philippe's `mbox`
- Philippe telling me to visit G. Kahn
- Philippe drinking a beer after the Algo seminar