Recursive Random Generation

Contents

1 Lexicographic isn't everything 1
1.1 Recall: Uniform generation . . . 1
1.2 Binary strings with no 00 substring . . . 1

2 Recursive generation 2
2.1 Combinatorial sum . . . 2
2.2 Product . . . 3
2.3 Binary Trees . . . 3

1 Lexicographic isn't everything

1.1 Recall: Uniform generation

A uniform generation scheme for combinatorial class $\mathcal{C}$ outputs an element of $\mathcal{C}$ such that all elements of size n are generated with equal probability. In the simplest scenario it takes as input n and outputs an element of $\mathcal{C}_n$ with probability $\frac{1}{C_n}$. In all of our analysis we assume that we have a constant time, perfect random number generator rnd that generates some element of $(0, x)$. [1] Given this we can draw a random integer in the range $[1..n]$ by

$$\mathrm{rnd}[1..n] : \lfloor \mathrm{rnd}(0,1) * n \rfloor + 1.$$

In general, when we discuss the complexity of an algorithm we will make clear the sort of operations that we "count" towards the complexity. We say that an algorithm has runtime $O(f(n))$ if the actual run-time on input n, denoted $g(n)$, satisfies the limit

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$$

for some constant c. To be $O(1)$ implies something constant with respect to n, like an evaluation of rnd or a swap.

1.2 Binary strings with no 00 substring

Lexicographic unranking can be used for uniform random generation, but its not always convenient or efficient. Consider binary strings with no 00 substring. We could view these as subsets and then order the subsets lexicographically, but how can we get a handle on what lists get skipped. Can we do something more systematic, something from the combinatorial specification.

Alternately, to generate binary strings with no 00 substring we could do a rejection algorithm. However, it is a small proportion of binary strings that are in this class, so this would actually waste a lot of time in the verification, and in the excessive generation.

For now let us treat the `draw` routine like a black box– we will see how to build it in the next section. First, let us see how we could play with it.

Maple
```
> with(combstruct):
# Encode 1* (011*)* (ep+0)
> Sys:= {W= Prod(Sequence(Z1), Sequence(Prod(Z0, Z1, Sequence(Z1))),
  Union(E, Z0)), Z0= Atom, Z1=Atom, E=Epsilon}:

# Verify that the OGF is Fibonacci
> gfseries(Sys, unlabelled, z)[W(z)];
    series(1+2*z+3*z^2+5*z^3+8*z^4+13*z^5+O(z^6),z,6)

# Make a procedure to tidy the output
> clean_string:= proc(w)
>   eval(subs( Prod=(()-> args), Sequence=(()->args), E=NULL, Epsilon=NULL, Z0=0, Z1=1, w))
```

[1] Why is this at all a reasonable assumption? Good question, and a topic for another day.

```
> end proc:
>clean_string(draw([W, Sys, unlabelled], size = 5));
     0, 1, 0, 1, 1
```

Hmmm. Now we are motivated to be even more efficient. It should be sufficient to generate (randomly) the number of 0s, and the sizes of the blocks between them. But how to do this so that uniform generation is preserved..?

We could start to answer this question by generating lots of long strings to guess the distribution of number of $0s$, and the lenghts of the blocks of 1s.

Here is the first part.

Maple

```
> Sys:= {W= Prod(Sequence(Z0), Sequence(Prod(Z0, Z1, Sequence(Z1))),
  Union(E, Z0)),  Z0= Atom, Z1=Atom, E=Epsilon}:
> N:= 200:
> for i from 1 to N do
>  w:= combstruct[draw]([W, Sys, unlabelled], size = 200):
>  Total[i]:=nops([eval(subs( Prod=(()-> args), Sequence=(()->args),
                             E=NULL, Epsilon=NULL, Z0=0, Z1=NULL, w))]);
>end do:
>average:= evalf(add(Total[i], i=1..N)/N);
                          57.46000000

# to get a full histogram try:    Statistics[Histogram]([seq(Total[i], i=1..N)]);
```

The result of this maple is appended at the end of this document.

Exercise. One of the Andrews had two ideas. Determine if they are true uniform random generation schemes.

Idea one: For each step generate a 1 or a 01 with equal probability.

Idea two: For each step flip a coin. After a zero always place a one.

2 Recursive generation

Next we describe a recursive generation scheme for structures using combinatorial sum and product operators. Let us recall the notation that A_n is the number of objects of size n in class $\mathcal{A}$.

For each class we want to describe a procedure $genA$ which takes as argument n and returns a random element of $\mathcal{A}$ such that each element of size n has probability $\frac{1}{A_n}$ of being generated. If there is no element of that size, the procedure returns NULL.

We will assume, in the algorithm descriptions below that the enumerative information is available, say in a table or a sub-routine. The Maple implementation builds a table as a pre-processing step.

We start with descriptions of very trivial generator for the atom and neutral classes.

Recursive Generator: Atom

```
// input n, a positive integer
// generates an Atom (if n=1)
genZ:= proc(n)
      if n=1 then Z else NULL
 end proc;
```

Recursive Generator: Epsilon

```
// input: n, a positive integer
// ouput: an Epsilon if n=0.
  genE:= proc(n)
       if n=0 then E else NULL
  end proc;
```

Given these foundations, we can build up generators for the larger classes.

2.1 Combinatorial sum

This operator is quite straightforward. Let us suppose that $\mathcal{A} = \mathcal{B} + \mathcal{C}$. Imagine that we have a method for generate elements of $\mathcal{B}$ and $\mathcal{C}$. How do we describe a method to generate elements from $\mathcal{A}$, again ensuring that the

generation is uniform? We need a mechanism to decide whether or not to generate an element from $\mathcal{B}$ or $\mathcal{C}$, and then we call those subroutines.

Consider the probabilities: The probability that element of $\mathcal{A}_n$ "came from" $\mathcal{B}_n$ is $\frac{B_n}{A_n}$.

Recursive Generator: Combinatorial Sum $\mathcal{A} = \mathcal{B} + \mathcal{C}$

```
// input: n a positive integer
// output: a uniformly generated element of A of size n
// remark: A= B+C
genA:= proc(n)
        let  x = rnd(0,1):
        if x< Bn/(Bn+Cn) then Return genB(n) else Return genC(n);
    od:
  end proc;
```

2.2 Product

If $\mathcal{A} = \mathcal{B} \times \mathcal{C}$ then the probability that a $\mathcal{A}$ structure of size n has a $\mathcal{B}$-component of size k and an $\mathcal{C}$-component of size $n-k$ is

$$\frac{B_k C_{n-k}}{A_n}.$$

Thus, we first generate a random number between 0 and 1 and then we find the smallest k such that the probability of generating a $\mathcal{B}$-component of size k or smaller is less than this number, and iterate on k

Recursive Generator: Product $\mathcal{A} = \mathcal{B} \times \mathcal{C}$

```
// input: n a positive integer
// output: a uniformly generated element of A of size n
// A= B*C
genA:= proc(n)
        let  x = rnd(0,1)
        k:= 0; s:= (b(0)*c(n))/a(n)
        while x > s do
              k:= k+1
              s:= s+ (b(k)*c(n-k))/an
           od
        Return  (genB(k), genC(n-k))
  end proc
```

Note that the cost of drawing k is equal to the number of iterative steps, i.e., is equal to k

2.3 Binary Trees

Let us try this out with binary trees. Now, we are lucky in this case, because we know the generating function, and an explicit expression for the n-th coefficient.

$$\mathcal{B} = \mathcal{Z} + \mathcal{Z} \times \mathcal{B} \times \mathcal{B}$$

$B_{2n+1} = \binom{2n}{n}\frac{1}{n+1}$.

Recursive Generator: Binary trees $\mathcal{B} = \mathcal{Z} + \mathcal{Z} \times \mathcal{B} \times \mathcal{B}$

```
// input: n a positive integer
// output: a uniformly generated binary tree with n vertices

genA:= proc(n)
        if n is even then NULL
        // Handle the union
        let  x = rnd(0,1)
        if x <= Z(n)/B(n)  then genZ(n) //Z(n)=1 if n=1 and 0 otherwise
        else // Handle the product
           k:= 0; s:= (B(1)*B(n-2))/B(n)
           while x > s do
              k:= k+1
              s:= s+(B(k)*B(n-k-1))/B(n)
```

```
        od
        Return  (genZ(1), genB(k), genB(n-k-1))
end proc
```

We could do a precise analysis to show that this method for generating a tree is $O(n^{3/2})$ and this is drieclty related to the asymptotic form of Catalan numbers. This is not the most efficient method.

The problem of scanning k from 0 to n is that, for large binary trees (as revealed by some simple asymptotic analysis) most of the size n is concentrated either on the left subtree or on the right subtree. Hence it is better to test for the values of k in the following order: $k = 0, n-1, 1, n-2, 2, n-3, \ldots$ (instead of $k = 0, 1, 2, \ldots$). Using this rule, one obtains a worst- case complexity of order $n \log(n)$.

We will look at sequences later.

```
> restart:
  with(combstruct):
```
[$agfeqns$, $agfmomentsolve$, $agfseries$, $allstructs$, $count$, $draw$, $finished$, $gfeqns$, $gfseries$, $gfsolve$, $iterstructs$, $nextstruct$] **(1)**

Binary strings with no "00" substring

We define a grammar for all binary strings with no 00 subsequence. We decompose the string by occurrences of 0.

```
> Sys:= {W= Prod(Sequence(Z0), Sequence(Prod(Z0, Z1, Sequence(Z1)
  )), Union(E, Z0)), Z0= Atom, Z1=Atom, E=Epsilon};
```
$Sys := \{E = \mathrm{E}, W = Prod(Sequence(Z0), Sequence(Prod(Z0, Z1, Sequence(Z1))), Union(E, Z0)), Z0 = Atom, Z1 = Atom\}$ **(1.1)**

These are counted by Fibonacci numbers, which we see in the ogf.

```
> gfseries(Sys, unlabelled, z)[W(z)];
```
$$1 + 2z + 3z^2 + 5z^3 + 8z^4 + 13z^5 + \mathrm{O}(z^6) \quad \textbf{(1.2)}$$

We can draw a random string of a given length. The output is awkward looking.

```
> draw([W, Sys, unlabelled], size = 5);
```
$$Prod(Sequence(Z0, Z0), Sequence(Prod(Z0, Z1, Sequence(Z1))), E) \quad \textbf{(1.3)}$$

So, let us clean it up. We remove the Epsilons, and the labels indicating a sequence or product construction.

```
> clean_string:= proc(w)
      eval(subs( Prod=(()-> args), Sequence=(()->args), E=NULL,
  Epsilon=NULL, Z0=0, Z1=1, w))
  end proc;
```
$clean_string := \textbf{proc}(w)$ **(1.4)**

$eval(subs(Prod = (() \to args), Sequence = (() \to args), E = NULL, \mathrm{Epsilon} = NULL, Z0 = 0, Z1 = 1, w))$

end proc

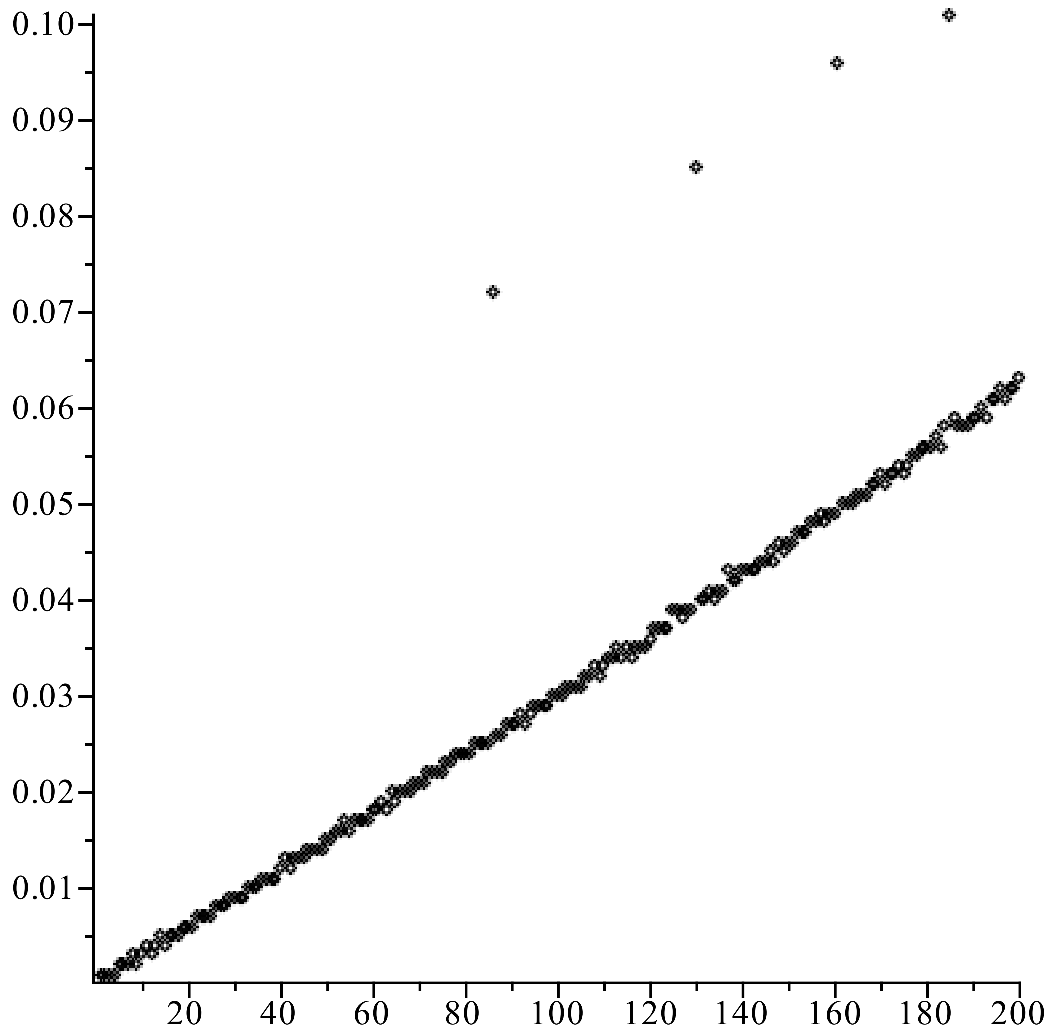
```
> clean_string(draw([W, Sys, unlabelled], size = 5));
```
$$0, 0, 0, 0, 0 \quad \textbf{(1.5)}$$

Efficiency

The program sets up an inital counting table to compute probabilities. It saves this in its memory the table so subsequent calls are faster. This explains the outlier points. Otherwise, the algorithm appears linear in n.

```
> Digits:= 10:

  for i from 1 to 200 do
    Sys:= {W= Prod(Sequence(Z0), Sequence(Prod(Z0, Z1, Sequence
  (Z1))), Union(E, Z0)), Z0= Atom, Z1=Atom, E=Epsilon}:
    T1:= time():
    draw([W, Sys, unlabelled], size = 10*i):
    TIME[i]:= time()-T1:
  end do:
  plots[listplot]([seq(TIME[i], i=1..200)], style=point);
```



Compare this linear algorithm to an algorithm which generates all strings, and then discards the strings with 00 substrings. How could we be even more efficient? We really just want to decide the number of 0s, and then the length of 1 strings between them. But, how to do this with the right probability????

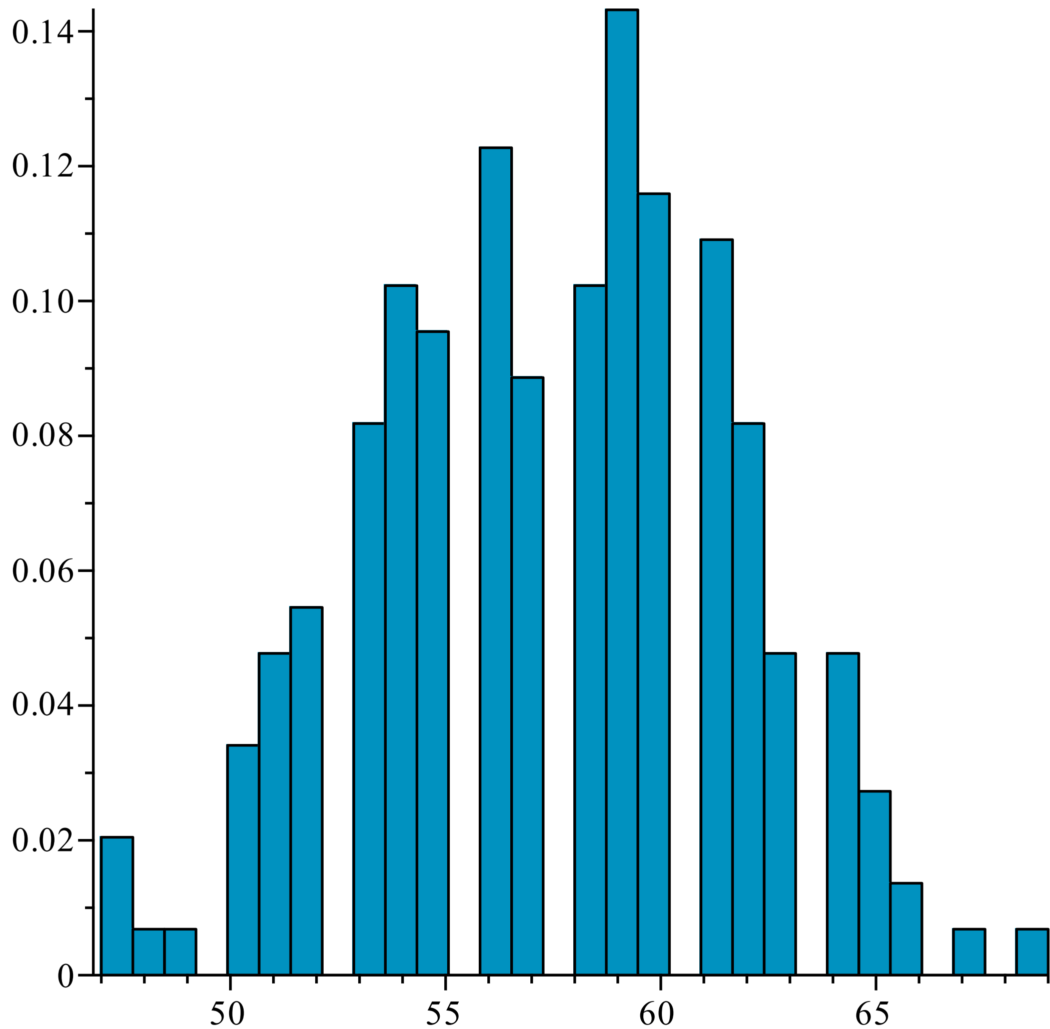
Properties of a random string

The expected number of 0s in a binary string with no 00 substring

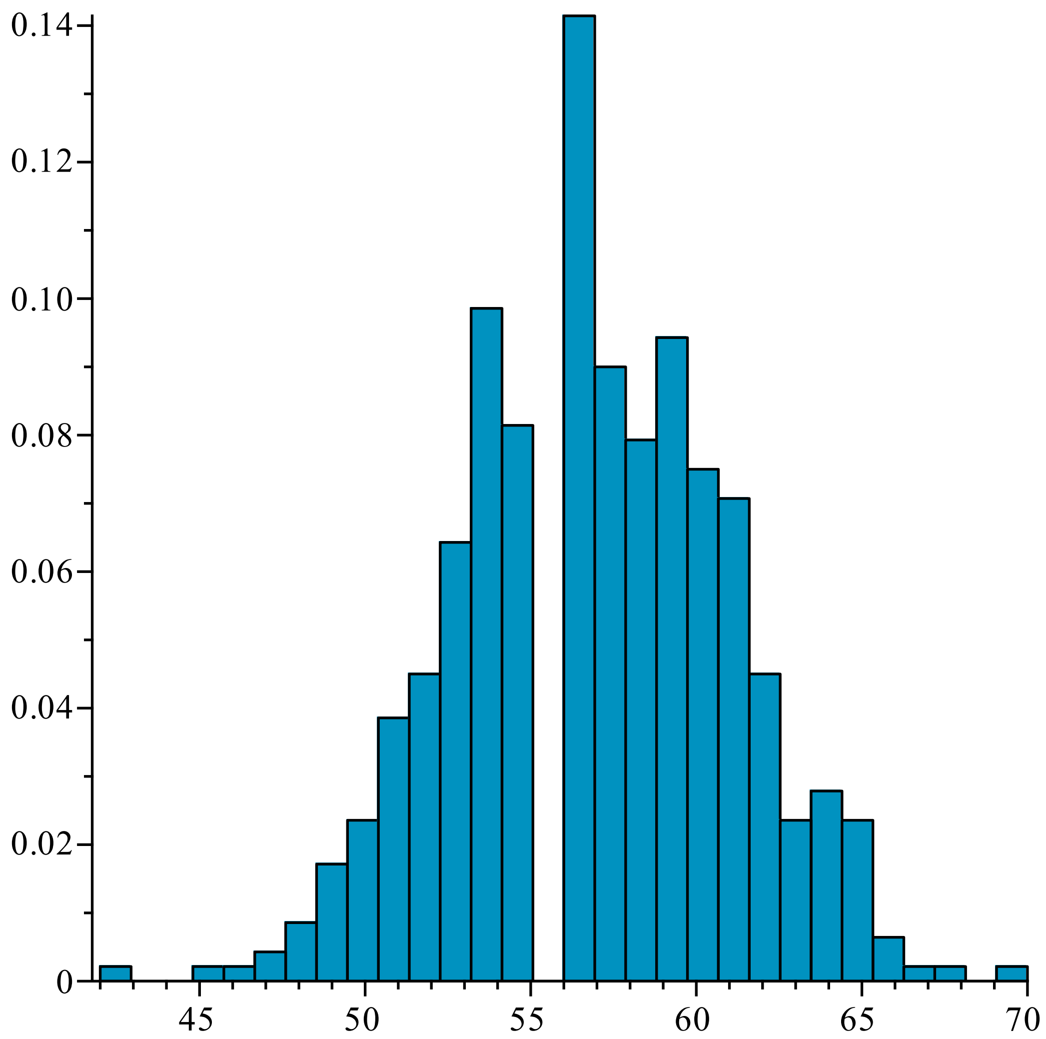
```
> Sys:= {W= Prod(Sequence(Z0), Sequence(Prod(Z0, Z1, Sequence
  (Z1))), Union(E, Z0)), Z0= Atom, Z1=Atom, E=Epsilon}:
  N:= 200:
  for i from 1 to N do
    w:= draw([W, Sys, unlabelled], size = 200):
    Total[i]:=nops([eval(subs( Prod=(()-> args), Sequence=(()
  ->args), E=NULL, Epsilon=NULL, Z0=0, Z1=NULL, w))]);
  end do:
> average:= evalf(add(Total[i], i=1..N)/N);
```

$$average := 57.46000000 \qquad (1.2.1)$$

```
> Statistics[Histogram]([seq(Total[i], i=1..N)]);
```




```
> Sys:= {W= Prod(Sequence(Z0), Sequence(Prod(Z0, Z1, Sequence
(Z1))), Union(E, Z0)), Z0= Atom, Z1=Atom, E=Epsilon}:
N:= 500:
for i from 1 to N do
  w:= draw([W, Sys, unlabelled], size = 200):
  Total[i]:=nops([eval(subs( Prod=(()-> args), Sequence=(()
->args), E=NULL, Epsilon=NULL, Z0=0, Z1=NULL, w))]);
end do:
Statistics[Histogram]([seq(Total[i], i=1..N)]);
```



The average is about 57, and it appears to be a binomial distribution, but it is tough to say.