Recursive Random Generation

Contents

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 the interval $[0, 1)$. [1] Given this we can draw a random integer in the range $[1..n]$ by

$$\mathrm{rnd}[1..n] := \lceil \mathrm{rnd}(0, 1) * n \rceil.$$

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

Consider the class $\mathcal{W}$ of binary strings which have no two consecutive zeros. How could we generate a uniformly random element of $\mathcal{W}_n$? Using the techniques of Lecture 4, we find that the counting sequence of $\mathcal{W}$ is (essentially) the Fibonacci sequence.

$$\begin{aligned}
\mathcal{W} &= 1^*(011^*)^*(\epsilon + 0)\\
W(z) &= \frac{1}{1-z}\frac{1}{1-\frac{z^2}{1-z}}(1+z) = \frac{1+z}{1-z-z^2} = \frac{A}{1-\frac{1+\sqrt{5}}{2}z} + \frac{B}{1-\frac{1-\sqrt{5}}{2}z}\\
A &= \frac{5+3\sqrt{5}}{10}, \quad B = \frac{5-3\sqrt{5}}{10}\\
W_n &= F_{n+1} \approx \frac{5+3\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{2}\right)^n.
\end{aligned}$$

We could use lexicographic unranking, for example. We could generate a random integer $x \in \mathrm{rnd}[1..F_{n+1}]$, and output the string $\textsc{Unrank}(x)$. The the problem here is that we do not know how to efficiently unrank W_n in lexicographic order.

Another way to generate a random string in $\mathcal{W}_m$ is to generate a random string binary string w of lengtih n (by converting $x = \mathrm{rnd}[0..2^n - 1]$ to binary). If w has two consecutive zeros, then we discard w and try again. This method is practical only for small values of n. The probability that a random n-bit binary string belongs to W_n is

$$\frac{W_n}{2^n} \approx \frac{5+3\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{2}\right)^n\left(\frac{1}{2}\right)^n \approx 1.17\,(0.81)^n.$$

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

Already for $n = 21$ we will be discarding more than 99% of the generated binary strings, since $W_{21}/2^{21} \approx 0.0081$.

The Maple `combstruct` package implements a procedure called `draw` that can generate a uniformly random element of a class, given its combinatorial specification. Here we encode the specification $\mathcal{W} = 1^*(011^*)^*(\epsilon + 0)$, which has two atoms and one neutral element. We verify that (W_n) is the Fibonacci series, make a tool to make the output readable, and sample $\mathcal{W}_{21}$.

Maple
```
> with(combstruct):
  Sys:={Z0 = Atom, Z1 = Atom, E = Epsilon,
         W = Prod( Sequence(Z1), Sequence(Prod(Z0,Z1,Sequence(Z1))), Union(E,Z0) )}:
  gfseries(Sys, unlabelled, z)[W(z)];
                   1+2*z+3*z^2+5*z^3+8*z^4+13*z^5+O(z^6)

> clean_word := w -> eval( w, [Prod=(()->args), Sequence=(()->args),
                               Z0=0, Z1=1, E=NULL, Epsilon=NULL] ):

> clean_word( draw([W, Sys, unlabeled], size = 21) );
                   1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1
```

How does Maple do this? Perhaps it is 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 try to answer this question by generating lots of long strings to guess the distribution of number of $0s$, and the lengths of the blocks of 1s. Here is the first part.

Maple
```
> N:=200:
  for i to N do
    w := draw([W, Sys, unlabeled], size=N) ;
    tot[i] := nops([eval(w, [Prod=(()->args),Sequence=(()->args),
                               Z0=0, Z1=NULL, E=NULL, Epsilon=NULL])])
  end do:
  average_zeros := evalf(add(tot[i], i=1..N)/N);
                   average_zeros := 55.1500000
# to get a full histogram try:    Statistics:-Histogram([seq(tot[i], i=1..N)]);
```

This is not going to help us much; even after many tests, it will be impossible to know these distributions

Exercise. Here are two more 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 defined by combinatorial sum and product operators. Recall that $\mathcal{A}_n$ is the set of objects of size n in class $\mathcal{A}$, and that $A_n = |\mathcal{A}|$. For each class we want to describe two procedures, `countA` and `genA`, Both procedures take a non-negative integer n as input. The counting procedure `countA` returns the number A_n. The generating procedure `genA` returns a random element of $\mathcal{A}_n$ such that each element has probability $\frac{1}{A_n}$ of being generated. If there is no element of size n, then the procedure returns NULL.

We start with the (almost trivial) descriptions of the counters and generators for for the atomic class $\mathcal{Z}$ and the neutral class $\mathcal{E}$.

Recursive Counter and Generator: Atom
```
// Input:  n (a non-negative integer)
countZ := proc(n)
            if n=1
               return 1
            else
               return 0
genZ := proc(n)
          if n=1
             return Z
          else
             return NULL
```

Recursive Counter ane Generator: Epsilon

```
// Input: n (a non-negative integer)
countE := proc(n)
          if n=0
             return 1
          else
             return 0
genE := proc(n)
          if n=0
             return E
          else
             return NULL
```

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

2.1 Combinatorial sum

Let us suppose that $\mathcal{A} = \mathcal{B} + \mathcal{C}$, and that we already have procedures, `genB` and `genC`, for generating random elements of both $\mathcal{B}$ and $\mathcal{C}$ of a given size (respectively). We would like to generate a random element of $\mathcal{A}_n$.

This is straight forward, provided that we have access to two tables or procedures, `countB` and `countC`, which return B_n and C_n (respectively) when n is input. Every random member of A_n comes from either $\mathcal{B}_n$ or $\mathcal{C}_n$ (not both). The probability that it comes from $\mathcal{B}_n$ is $\frac{B_n}{A_n}$. We assume the ability to generate a uniformly random integer from the set $\{0, 1, \ldots, A_n\}$, where $A_n = B_n + C_n$.

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

```
// Inputs: n (a nonnegative integer),
//         countB, genB, countC, genC (procedures for counting & generating in B and C)
// Output: a uniformly generated element of size n from A = B + C
// Global:  rnd (a random integer generator)
genSum := proc( n, countB, genB, countC, genC )
            x := rnd( 1 .. countB(n)+countC(n) ):
            if x <= countB(n)
                return genB(n)
            else
                return genC(n)
```

It would be more useful if, instead of returning a random element of $\mathcal{A}_n$, the program `genSum` returned *two procedures*. The first procedure, referred to as `countA`, inputs n and outputs A_n. The second procedure, referred to as `genA`, inputs n and outputs a random member of $\mathcal{A}_n$. Here is how to modify the above program to do this.

Make Recursive Generator: CombinatorialSum $\mathcal{A} = \mathcal{B} + \mathcal{C}$

```
// Inputs: countB, genB, countC, genC (procedures for counting & generating in B and C)
// Output: a pair ( countA, genA ) of procedures, both accepting an integer n as input
//         where - countA(n) is the number of objects in A_n,
//               - genA(n) is a randomly generated object from A_n
// Global:  rnd (a random integer generator)
makeGenSum := proc( countB, genB, countC, genC )

                countA := proc(n)
                            return countB(n)+countC(n)

                genA := proc(n)
                          x := rnd( 1 .. countA(n) ):
                          if x <= countB(n)
                              return genB(n)
                          else
                              return genC(n)

                return ( countA, genA )
```

2.2 Product

If $\mathcal{A} = \mathcal{B} \times \mathcal{C}$, then for $n \geq 0$ we have

$$A_n = \sum_{k=0}^{n} B_k C_{n-k} \, .$$

For $k \in \{0, 1, \ldots, n\}$, the probability that an random member of $\mathcal{A}_n$ has a $\mathcal{B}$-component of size k and an $\mathcal{C}$-component of size $n-k$ is

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

To find which value of k to use, we generate a random integer $x \in \{1, 2, \ldots, A_n\}$, and then we find the smallest integer k such that the probability of generating a pair in $\mathcal{B} \times \mathcal{C}$ whose $\mathcal{B}$-component has size less than k is less than $\frac{x}{A_n}$. For example, if x satisfies

$$B_0C_n + B_1C_{n-1} < x \leq B_0C_n + B_1C_{n-1} + B_2C_{n-2}$$

then we should take $k = 2$ and generate a random pair (b, c) where $b \in B_k$ and $c \in C_{n-k}$.

$$\begin{array}{c} \hspace{2em} x \\ \hspace{2em} \downarrow \\ \begin{array}{|c|c|c|c|c|c|} \hline B_0C_n & B_1C_{n-1} & B_2C_{n-2} & B_3C_{n-3} & \ldots & B_nC_0 \\ \hline \end{array} \end{array}$$

As above, we would like our program to return two procedures, `countA` and `genA`, where `countA`$(n) = A_n$ and `genA`(n) is a random element of $\mathcal{A}_n$.

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

```
// Input: countB, genB, countC, genC (procedures for counting & generating in B and C
// Output: a pair ( countA, genA ) of procedures, both accepting an integer n as input
//         where - countA(n) is the number of objects in A_n,
//               - genA(n) is a randomly generated object from A_n
// Global:  rnd (a random integer generator)
makeGenProd := proc( countB, genB, countC, genC )

                 countA := proc(n)
                           return add( countB(k)*countC(n-k), k=0..n )

                 genA := proc(n)
                           x := rnd( 1 .. countA(n) )
                           k :=  0
                           s := countB(0) * countC(n)
                           while s < x do
                               k := k+1
                               s := s + countB(k) * countC(n-k)
                           return ( genB(k), genC(n-k) )

                 return ( countA, genA )
```

2.3 Sequences

Recall that if $\mathcal{A} = \textsc{Seq}(\mathcal{B})$, then

$$\mathcal{A} \cong \mathcal{E} + \mathcal{B} \times \mathcal{A}.$$

We can use this recursive specification to construct counter and generator procedures (`countA`, `genA`) for $\mathcal{A}$ in a recursive implementation relying on `makeGenSum` and `makeGenProd`. The resulting algorithm is not particularly efficient. First we check that $B_0 = 0$, for otherwise $\textsc{Seq}(\mathcal{B})$ is not defined.

Make Recursive Generator: Sequence $\mathcal{A} = \textsc{Seq}(B)$

```
// Input: countB, genB (counter and generator procedures for combinatorial class B)
// Output: a pair ( countA, genA ) of procedures, both accepting an integer n as input
//         where - countA(n) is the number of objects in A_n,
//               - genA(n) is a randomly generated object from A_n

makeGenSeq := proc( countB, genB )
                if countB(0)>0
                  return "Error: The input class has a neutral element"
                else
                  (countBxA, genBxA) := makeGenProd( countB, genB, countA, genA)
                  (countA, genA) := makeGenSum( countE, genE, countBxA, genBxA )
                  return ( countA, genA )
```

When implementing this, care must be taken to to prevent infinite recursion. In particular, in the procedure `makeGenSum` we must ensure that the function calls `countC`(n) and `genC`(n) are never evaluated if `countB`$(0) = 0$, for this would result in an infinite recursion in the evaluation of either of the procedures output by `makeGenSeq`.

Exercise. Explain why the following procedure counts the number of objects of size n in $\mathcal{A} = \textsc{Seq}(\mathcal{B})$, where $\mathcal{B}$ has counting sequence $0, B_1, B_2, \ldots$.

```
> with(combinat):
  countSeqB := proc(n)
    if n=0
    then return 1
    else return add( convert( map(t->B[t],comp), `*` ), comp in `union`(seq(composition(n,i), i=1..n)) )
    end if
  end proc:
> for n from 0 to 5 do A[n]=countSeqB(n) end do;
```

$$A_0 = 1$$
$$A_1 = B_1$$
$$A_2 = B_1^2 + B_2$$
$$A_3 = B_1^3 + 2\,B_1\,B_2 + B_3$$
$$A_4 = B_1^4 + 3\,B_1^2\,B_2 + 2\,B_1\,B_3 + B_2^2 + B_4$$
$$A_5 = B_1^5 + 4\,B_1^3\,B_2 + 3\,B_1^2\,B_3 + 3\,B_1\,B_2^2 + 2\,B_1\,B_4 + 2\,B_2\,B_3 + B_5$$

2.4 Binary Trees

Let us make a random generator for the class $\mathcal{B}$ of plane binary trees. We recall that each member of $\mathcal{B}$ is a non-empty tree which is either a single atomic root node $\mathcal{Z}$ or is specified by a root node and an ordered pair of members of $\mathcal{B}$ (the left and right subtrees). So $\mathcal{B}$ has the recursive specification

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

It is straight forward to make a counter and generator for $\mathcal{B}$.

Make Recursive Generator: Plane binary trees $\mathcal{B} = \mathcal{Z} + \mathcal{Z} \times \mathcal{B} \times \mathcal{B}$

```
// Input: none
// Output: a pair ( countB, genB ) of procedures, both accepting an integer n as input
//         where - countB(n) is the number of plane binary trees with n nodes,
//               - genB(n) is a randomly generated plane binary tree with n nodes

makeGenBinaryTree := proc()
                      ( countZxB, genZxB ) := makeGenProd( countZ, genZ, countB, genB )
                      ( countB, genB ) := makeGenProd( countZxB, genZxB, countB, genB )
                      return ( countB, genB )
```

Let us write an explicit version of this. We know the counting sequence of $\mathcal{B}$ in terms of the Catalan numbers C_m.

$$B_n = \begin{cases} 0 & \text{if } n \text{ is even} \\ C_m = \frac{1}{m+1}\binom{2m}{m} & \text{if } n = 2m+1 \end{cases}.$$

If a tree has n nodes where $n > 1$, and its left subtree has k nodes, then both n and k are odd numbers, and the right subtree has $n - 1 - k$ nodes.

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 plane binary tree with n vertices
makeGenBinaryTree := proc(n)
   countB := proc(n)
                   if n is even
                     return 0
                   else
                      return 2/(n+1) * (n-1)! / (((n-1)/2)!)^2

   genB := proc(n)
             if n is even
                return NULL
             else
               x = rnd( 1 .. countB(n) )
               if x = 1                       // Handle the union
                 return genZ(n)
               else                           // Handle the product
                 k = 1
                 s =  countB(k)*countB(n-1-k)
                 while s < x do
                     k = k+2
                     s = s + countB(k)*countB(n-1-k)
                 return ( genZ(1), genB(k), genB(n-1-k) )   //This specifies the tree

             return ( countB, genB )
```

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

2.5 Boustrophedon

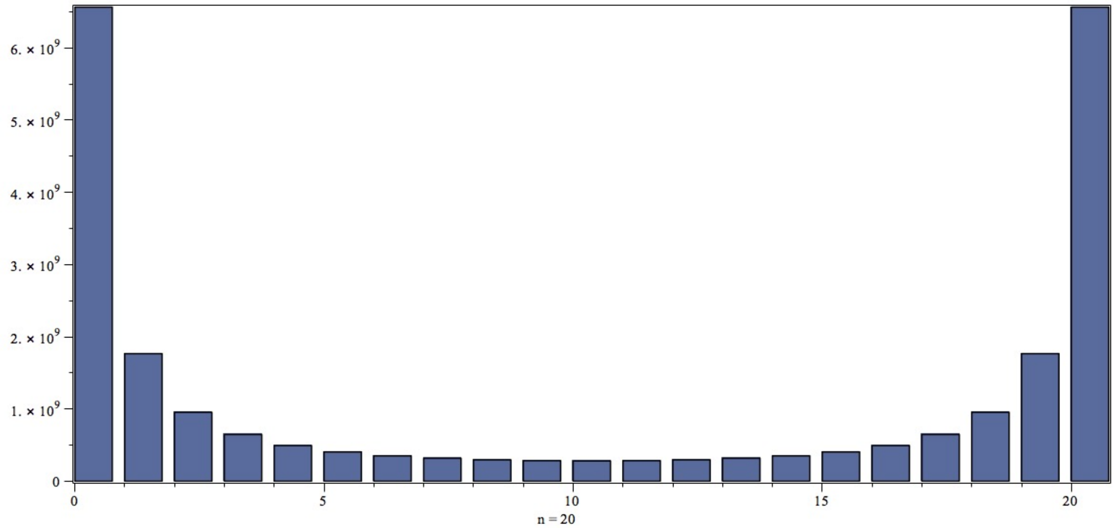
There is an inefficiency in the above random tree generator GENB that arises from the fact that most binary trees of size n is have almost all of their nodes concentrated either on the left subtree or on the right subtree. We saw above that, for integers t and m with $0 \leq t \leq m - 1$, the number of binary trees of size $n = 2m + 1$ which have $2t + 1$ nodes in its left subtree and $(n - 1) - (2t + 1) = 2(m - 1 - t) + 1$ nodes in its right subtree equals

$$B_{2t+1}B_{2(m-1-t)+1} = C_t C_{m-1-t}.$$

The Catalan numbers grow so quickly, that the numbers near the start and end of the following sequence are much larger than those in the middle.

$$C_0C_{m-1},\ C_1C_{m-2},\ \ldots\ ,\ C_{m-1}C_0$$

Here is a plot of C_tC_{m-1-t} versus t when $m = 21$.



In fact Catalan numbers satisfy the inequality

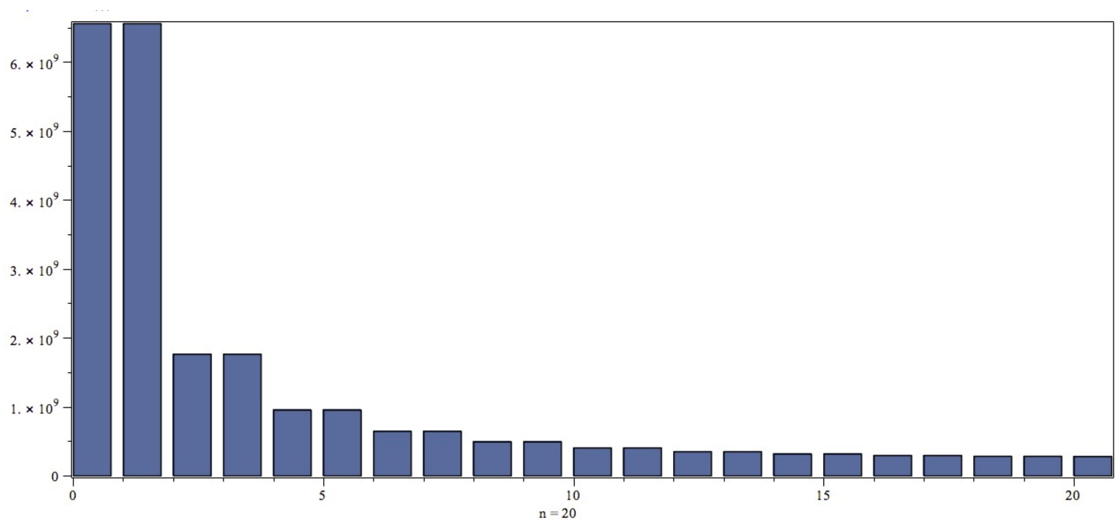
$$C_0 C_{m-1} = C_{m-1} C_0 > \frac{C_m}{4} = \frac{B_{2m+1}}{4},$$

so more than half of the binary trees with $2m+1$ nodes have just one node in either its left subtree or its right subtree! In over 25% of the calls to GENB$(2m+1)$, the `while s < x` loop will be executed m times (for $k = 1, 3, 5, \ldots, 2m+1$).

A better alternative is to use the "zig-zag" sequence to update s within in the `while s < x` loop.

$$C_0 C_{m-1},\ C_{m-1} C_0,\ C_1 C_{m-2},\ C_{m-2} C_1,\ \ldots,\ C_{\lfloor \frac{m-1}{2} \rfloor} C_{\lceil \frac{m-1}{2} \rceil}.$$

This sequence is plotted below with $m = 21$. With this ordering, more than half of the calls to GENB$(2m+1)$ will result in the `while s < x` loop being executed just once or twice!



The "zig-zag" permutation $[0 \quad (m-1) \quad 1 \quad (m-2) \quad 2 \quad (m-3) \quad \ldots]$ is called the *Boustrophedonic* ordering by Flajolet (1994) and other researchers, because it is reminiscent of *boustrophedon* writing, such as in some Ancient Greek manuscripts, where successive lines of text are read from left-to-right, and then from right-to-left, like a zig-zag.

In fact, the simple trick of using boustrophedonic order reduces the average number of iterations of the inner loop of GENB(n) from about $n/4$ to about $\sqrt{n/\pi}$ (Maple can verify this). Flajolet et al (1994) show that the net effect of using the boustrophedon order in the recursive procedure is an improvement in average time from $O(n^{3/2})$ to $O(n \log n)$ to generate a random binary tree on n nodes.