Chapter Thirteen
RANDOM COMBINATORIAL OBJECTS

1. GENERAL PRINCIPLES.

1.1. Introduction.

Some applications demand that random combinatorial objects be generated: by definition, a combinatorial object is an object that can be put into one-to-one correspondence with a finite set of integers. The main difference with discrete random variate generation is that the one-to-one mapping is usually complicated, so that it may not be very efficient to generate a random integer and then determine the object by using the one-to-one mapping. Another characteristic is the size of the problem: typically, the number of different objects is phenomenally large. A final distinguishing feature is that most users are interested in the uniform distribution over the set of objects.

In this chapter, we will discuss general strategies for generating random combinatorial objects, with the understanding that only uniform distributions are considered. Then, in different subsections, particular combinatorial objects are studied. These include random graphs, random free trees, random binary trees, random search trees, random partitions, random subsets and random permutations. This is a representative sample of the simplest and most frequently used combinatorial objects. It is hoped that for more complicated objects, the readers will be able to extrapolate from our examples. A good reference text is Nijenhuis and Wilf(1978).

1.2. The decoding method.

Since we want to generate only one of a finite number of objects, it is possible to find a function f such that for every pair of objects (ξ,ς) in the collection of objects Ξ, we have

$$f(\xi)\neq f(\varsigma)\in\{1, \ldots, n\},$$

where n is an integer, which is usually equal to $|\Xi|$, the number of elements in Ξ. Such a function will be called a coding function. By $f^{-1}(i)$, we define the object ξ in Ξ for which $f(\xi)=i$ (if this object exists). When $|\Xi|=n$, the following decoding algorithm is valid.

The decoding method

[NOTE: f is a coding function.]
Generate a uniform random integer $X\in\{1, \ldots, n\}$.
RETURN $f^{-1}(X)$

The expected time taken by this algorithm is the average time needed for decoding f:

$$\frac{1}{n}\sum_{i=1}^{n}\text{TIME}(f^{-1}(i)).$$

The advantage of the method is that only one uniform random variate is needed per random combinatorial object. The decoding method is optimal from a storage point of view, since each combinatorial object corresponds uniquely to an integer in $1, \ldots, n$. Thus, about $\log_2 n$ bits are needed to store each combinatorial object, and this cannot be improved upon. Thus, the coding functions can be used to store data in compact form. The disadvantages usually outweigh the advantages:

1. Except in the simplest cases, $|\Xi|$ is too large to be practical. For example, if this method is to be used to generate a random permutation of $1, \ldots, 40$, we have $|\Xi|=40!$, so that multiple precision arithmetic is necessary. Recall that $12!<2^{35}<13!$.
2. The expected time taken by the decoding algorithm is often unacceptable. Note that the time taken by the uniform random variate generator is negligible compared to the time needed for decoding.
3. The method can only be used when for the given value of n, we are able to count the number of objects. This is not always the case. However, if we use rejection (see below), the counting problem can be avoided.

Example 1.1. Random permutations.

Assume that $\Xi = \{$ all permutations of $1, \ldots, n$ $\}$. There are a number of possible coding functions. For example, we could use the factorial representation of Lehmer (1964), where a permutation $\sigma_1, \ldots, \sigma_n$ is uniquely described by a sequence of $n-1$ integers $a_1, \ldots, a_{n-1}$ (where $0 \leq a_i \leq n-i$) according to the following rule: start with $1, \ldots, n$. Let σ_1 be the a_1+1-st integer from this list, and delete this number. Let σ_2 be the a_2+1-st number of the remaining numbers, and so forth. Then, define

$$f(a_1, \ldots, a_{n-1}) = a_1(n-1)!+a_2(n-2)!+\cdots+a_{n-1}1!+1$$

It is easy to see that f is a proper coding function giving all values between 1 and $n!$. Just observe that

$$n! = (n-1)!n = (n-1)!(n-1)+(n-1)!$$
$$= (n-1)!(n-1)+(n-2)!(n-2)+\cdots+1!1+1 .$$

The algorithm consists of generating a random integer X between 1 and $n!$, determining $a_1, \ldots, a_{n-1}$ from X, and determining the random permutation $\sigma_1, \ldots, \sigma_n$ from the a_i sequence. First, the a_i's are obtained by repeated divisions by $(n-1)!, (n-2)!$, etcetera. The σ_i's can be obtained by an exchange algorithm. Formally, we have:

Random permutation generator

Generate a random integer X uniformly distributed on $\{1, \ldots, n!\}$. $X \leftarrow X-1$.
FOR $i := 1$ TO $n-1$ DO
$(a_i, X) \leftarrow \left(\left\lfloor \frac{X}{(n-i)!} \right\rfloor, X \bmod (n-i)! \right)$ (This determines all the a_i's.)
Set $\sigma_1, \ldots, \sigma_n \leftarrow 1, \ldots, n$.
FOR $i := 1$ TO $n-1$ DO
Exchange (swap) σ_{a_i+1} and σ_{n-i+1}
RETURN $\sigma_1, \ldots, \sigma_n$.

In the exchange step of the algorithm, we exchange a randomly picked element with the last element in every iteration. The time taken by the algorithm is $O(n)$. ■

Sometimes simple coding functions can be found with the property that $n > |\Xi|$, that is, some of the integers in $1, \ldots, n$ do not correspond to any combinatorial object in Ξ. When n is not much larger than $|\Xi|$, this is not a big problem, because we can apply the rejection principle:

Decoding with rejection

REPEAT
  Generate a random integer X with a uniform distribution on $\{1, \ldots, n\}$.
  Accept $\leftarrow [f(\xi)=X$ for some $\xi \in \Xi]$
UNTIL Accept
RETURN $f^{-1}(X)$

Just how one determines quickly whether $f(\xi)=X$ for some $\xi \in \Xi$ depends upon the circumstances. Usually, because of the size of $|\Xi|$, it is impossible or not practical to store a vector of flags, flagging the bad values of X. If $|\Xi|$ is moderately small, then one could consider doing this in a preprocessing step. Most of the time, it is necessary to start decoding X, until in the process of decoding one discovers that there is no combinatorial object for the given value of X. In any case, the expected number of iterations is $\frac{n}{|\Xi|}$. What we have bought here is (i) simplicity (the decoding function can be simpler if we allow gaps in our enumeration) and (ii) convenience (it is not necessary to count $|\Xi|$; in fact, this value need not be known at all !).

1.3. Generation based upon recurrences.

Most combinatorial objects can be counted indirectly via recurrence relations. Direct counting, as in the case of random permutations, addresses itself to the decoding method. Counting via recurrences can be used to obtain alternative generators. The idea has been around for some time. It was first developed thoroughly by Wilf (1977) (see also Nijenhuis and Wilf (1978)).

We need to have two things:

1. A formula for the number of combinatorial objects with a certain parameter (or parameters) k in terms of the number of combinatorial objects with smaller parameter(s). This will be called our recurrence relation.
2. A good understanding of the recurrence relation, so that the relation itself can be linked in a constructive way to a combinatorial object.

For example, consider Ξ_n, the collection of permutations of $1, \ldots, n$. We have

$$|\Xi_n| = n\,|\Xi_{n-1}|\,.$$

The meaning of this relation is clear: we can obtain $\xi \in \Xi_n$ by considering all permutations Ξ_{n-1}, padding them with the single element n (in the last position), and then swapping the n-th element with one of the n elements. The swapping

operation gives us the factor n in the recurrence relation. We will rewrite the recurrence relation as follows:

$$\Xi_n(1,2,\ldots,n) = \bigcup_{i=1}^{n} \Xi_{n-1}(1,2,\ldots,i-1,i+1,\ldots,n).i$$

where $\Xi_{n-1}(1,2,\ldots,i-1,i+1,\ldots,n)$ is the collection of all permutations of the given $n-1$ elements, and . is the concatenation operator. To generate a random element from Ξ_n, it suffices to choose a random term in the union (with probability proportional to the cardinality of the chosen term), and to construct the part of the combinatorial object that corresponds to this choice. In the case of the random permutations, each of the n terms in the union shown in the recurrence relation has equal cardinality, and should thus be chosen with equal probability. But choosing the i-th term corresponds to putting the i-th element of the n-vector at the end of the permutation, and generating a random permutation for the $n-1$ remaining elements. This leads quite naturally to the swapping method for random permutations:

The swapping method for random permutations

Set $\sigma_1, \ldots, \sigma_n \leftarrow 1, \ldots, n$.
FOR $i := n$ DOWNTO 2 DO
    Generate X uniformly in $1, \ldots, i$.
    Swap σ_X and σ_i.
RETURN $\sigma_1, \ldots, \sigma_n$.

There are obviously more complicated situations: see for example the subsections on random partitions and random binary trees in the corresponding subsections. For now, we will merely apply the technique to the generation of random subsets of size k out of n elements, and see that it reduces to the sequential method in random sampling.

There are

$$\left[\begin{matrix} n \\ k \end{matrix}\right] = \left[\begin{matrix} n-1 \\ k \end{matrix}\right] + \left[\begin{matrix} n-1 \\ k-1 \end{matrix}\right]$$

sets of size $k \geq 1$ consisting of different integers picked from $\{1, \ldots, n\}$, where $n \geq k$. Clearly, as boundary conditions, we have

$$\left[\begin{matrix} n \\ n \end{matrix}\right] = 1 ; \left[\begin{matrix} n \\ 1 \end{matrix}\right] = 1 .$$

The recurrence can be interpreted as follows: k integers can be drawn from $2, \ldots, n$ (thus, ignoring 1), or by choosing 1 and choosing a random subset of size $k-1$ from $2, \ldots, n$ (thus, including 1). The probability of inclusion of 1 is

therefore

$$\frac{\binom{n-1}{k-1}}{\binom{n}{k}} = \frac{k}{n}.$$

This leads directly to the following algorithm:

Random subset of size k from 1,...,n

$S \leftarrow \emptyset$ (set to be returned is empty)
FOR $i := 1$ TO n DO
    Generate a uniform [0,1] random variate U.
    IF $U \leq \frac{k}{n-i+1}$ THEN $S \leftarrow S \cup \{i\}$; $k \leftarrow k-1$
RETURN S

We can also look at the method of recurrences as some sort of composition method. Typically, Ξ_n is split into a number of subsets of objects, each having a special property. Let us write

$$\Xi_n = \bigcup_{i=1}^{k} \Xi_n(i)$$

where the sets $\Xi_n(i)$ are non-overlapping. If an integer i is picked with probability

$$\frac{|\Xi_n(i)|}{|\Xi_n|} \quad (1 \leq i \leq k),$$

and if we generate a uniformly distributed object in $\Xi_n(i)$, then the random object is uniformly distributed over Ξ_n. Of course, we are allowed to apply the same decomposition principle to the individual subsets in turn. The subsets have generally speaking some property which allows us to construct part of the solution, as was illustrated with random permutations and random subsets.

2. RANDOM PERMUTATIONS.

2.1. Simple generators.

The decoding method of section XIII.1.2 requires only one uniform random variate per random permutation of $1, \ldots, n$. It was suggested in a number of papers (see e.g. Robinson (1967), Jansson (1966), de Balbine (1967), and the survey paper by Plackett (1968)). Given an arbitrary array of length n, and one uniformly distributed random integer on $1, \ldots, n!$, the decoding method constructs in time $O(n)$ one random permutation of $1, \ldots, n$. The algorithm of section XIII.1.2 is a two-pass algorithm. Robson (1969) has pointed out that there is a simple one-pass algorithm based upon decoding:

Robson's decoding algorithm

[NOTE: This algorithm assumes that some permutation $\sigma_1, \ldots, \sigma_n$ of $1, \ldots, n$ is given. Usually, this permutation is a previously generated random permutation.]

Generate a random integer X uniformly on $1, \ldots, n!$.

FOR $i := n$ DOWNTO 2 DO

$$(X, Z) \leftarrow \left(\left\lfloor \frac{X}{i} \right\rfloor, X \bmod i + 1\right)$$

Swap σ_i and σ_Z

RETURN $\sigma_1, \ldots, \sigma_n$

Despite the obvious improvement over the algorithm of section XIII.1.2, the decoding method remains of limited value because $n!$ increases too quickly with n.

The exchange method of section XIII.1.3 on the other hand does not have this drawback. It is usually attributed to Moses and Oakford (1963) and to Durstenfeld (1964). The method requires $n-1$ independent uniform random variates per random permutation, but it is extremely simple in conception, requiring only one pass and no multiplications, divisions or truncations.

2.2. Random binary search trees.

Random permutations are useful in a number of applications. As we have pointed out earlier, the swapping method can be stopped after a given number of iterations to yield a method for generating a random subset of $1, \ldots, n$ of size $k < n$. This was dealt with in chapter XII on random sampling. Another application deals with the generation of a random binary search tree.

A random binary search tree with n nodes is defined as a binary search tree constructed from a random permutation, where each permutation is equally likely. It is easy to see that different permutations can yield a tree of the same shape, so all trees are not equally likely (but the permutations are !). It is clear that if we proceed by inserting the elements of a random permutation in turn, starting from an empty tree, then the expected time of the algorithm can be measured by

$$\sum_{i=1}^{n} E(D_i)$$

where D_i is the depth (path length from root to node) of the i-th node when inserted into a binary search tree of size $i-1$ (the depth of the root is 0). The following result is well-known, but is included here because of its short unorthodox proof, based upon the theory of records (see Glick (1978) for a recent survey):

Lemma 2.1. In a random binary search tree,

$$E(D_n) \leq 2(\log(n)+1) .$$

In fact $E(D_n) \sim 2\log(n)$. Based upon Lemma 2.1, it is not difficult to see that the expected time for the generator is $O(n\log(n))$. Since $E(D_n) \sim 2\log(n)$, the expected time is also $\Omega(n\log(n))$.

Proof of Lemma 2.1.

D_n is equal to the number of left turns plus the number of right turns on the path from the root to the node corresponding to the n-th element. By symmetry, $E(D_n)$ is twice the expected number of right turns. These right turns can conveniently be counted as follows. Consider the random permutation of $1, \ldots, n$, and extract the subsequence of all elements smaller than the last element. In this subsequence (of length at most $n-1$), flag the records, i.e. the largest values seen thus far. Note that the first element always represents a record. The second element is a record with probability one half, and the i-th element is a record with probability $1/i$. Each record corresponds to a right turn and vice versa. This can be seen by noting that elements following a record which are not records themselves are in a left subtree of a node on the path to the record, whereas the n-th original element is in the right subtree. Thus, these elements cannot have any influence on the level of the n-th element. The subsequence has length between 0 and $n-1$, and to bound the expected number of records from above, it suffices to consider subsequences of length equal to $n-1$. Therefore, the expected depth of the last node is not more than

$$2\sum_{i=1}^{n-1} \frac{1}{i} \leq 2\left(1+\int_1^n \frac{1}{x}\, dx\right) = 2(1+\log(n)) . \blacksquare$$

But just as with the problem of the generation of an ordered random sample, there is an important short-cut, which allows us to generate the random binary search tree in linear expected time. The important fact here is that if the root is fixed (say, its integer value is i), then the left subtree has cardinality $i-1$, and the right subtree has cardinality $n-i$. Furthermore, the value of the root itself is uniformly distributed on $1, \ldots, n$. These properties allow us to use recursion in the generation of the random binary search tree. Since there are n nodes, we need no more than n uniform random variates, so that the total expected time is $O(n)$. A rough outline follows:

Linear expected time algorithm for generating a random binary search tree with n nodes

[NOTE: The binary search tree consists of cells, having a data field "Data", and two pointer fields, "Left" and "Right". The algorithm needs a stack S for temporary storage.]

MAKENULL (S) (stack S is initially empty).

Grab an unused cell pointed to by pointer p.

PUSH $[p, 1, n]$ onto S.

WHILE NOT EMPTY (S) DO

    POP S, yielding the triple $[p, l, r]$.

    Generate a random integer X uniformly distributed on $l, \ldots, r$.

    $p\uparrow$.Data$\leftarrow X$, $p\uparrow$.Left$\leftarrow$NIL, $p\uparrow$.Right$\leftarrow$NIL

    IF $X < r$ THEN

        Grab an unused cell pointed to by q^*.

        $p\uparrow$.Right$\leftarrow q^*$ (make link with right subtree)

        PUSH $[q^*, X+1, r]$ onto stack S (remember for later)

    IF $X > l$ THEN

        Grab an unused cell pointed to by q.

        $p\uparrow$.Left$\leftarrow q$ (make link with left subtree)

        PUSH $[q, l, X-1]$ onto stack S (remember for later)

2.3. Exercises.

1. Consider the following putative swapping method for generating a random permutation:

Start with an arbitrary permutation $\sigma_1, \ldots, \sigma_n$ of $1, \ldots, n$.
FOR i:=1 TO n DO
    Generate a random integer X on $1, \ldots, n$ (note that the range does not depend upon i).
    Swap σ_i and σ_X
RETURN $\sigma_1, \ldots, \sigma_n$

Show that this algorithm does not yield a valid random permutation (all permutations are not equally likely). Hint: there is a three line combinatorial proof(de Balbine, 1967).

2. The distribution of the height H_n of a random binary search tree is very complicated. To simulate H_n, we can always generate a random binary search tree and find H_n. This can be done in expected time $O(n)$ as we have seen. Find an algorithm for the generation of H_n in sublinear expected time. The closer to constant expected time, the better.
3. Show why Robson's decoding algorithm is valid.
4. Show that for a random binary search tree, $E(D_n) \sim 2 \log(n)$ by employing the analogy with records explained in the proof of Lemma 2.1.
5. Give a linear expected time algorithm for constructing a random trie with n elements. Recall that a trie is a binary tree in which left edges correspond to zeroes and right edges correspond to ones. The n elements can be considered as n independent infinite sequences of zeroes and ones, where all zeroes and ones are obtained by perfect coin tosses. This yields an infinite tree in which there are precisely n paths, one for each element. The trie defined by these elements is obtained by truncating all these paths to the point that any further truncation would lead to two identical paths. Thus, all internal nodes which are fathers of leaves have two children.
6. **Random heap.** Give a linear expected time algorithm for generating a random heap with elements $1, \ldots, n$ so that each heap is equally likely. Hint: associate with integer i the i-th order statistic of a uniform sample of size n, and argue in terms of order statistics.

3. RANDOM BINARY TREES.

3.1. Representations of binary trees.

A binary tree consists of a root, or a root and a left and/or a right subtree, and each of the subtrees in turn is a binary tree. Two binary trees are similar if they have the same shape. They are equivalent if they are similar, and if the corresponding nodes contain the same information. The distinction between similarity and equivalence is thus based upon the absence or presence of labels for the nodes. If there are n nodes, then every permutation of the labels of the nodes yields another labeled binary tree, and all such trees are similar.

A random binary tree with n nodes is a random unlabeled binary tree which is uniformly distributed over all nonsimilar binary trees with n nodes. The uniform distribution on the n nodes causes some problems, as we can see from the following simple example: there are 5 different binary trees with 3 nodes. Yet, if we generate such a tree either by generating a random permutation of 1,2,3 and constructing a binary search tree from this permutation, or by growing the tree via uniform replacements of NIL pointers by new nodes, then the resulting trees are not equally likely. For example, the complete binary tree with 3 nodes has probability $\frac{1}{3}$ in both schemes, instead of $\frac{1}{5}$ as is required. The uniformity condition will roughly speaking stretch the binary trees out, make them appear more unbalanced, because less likely shapes (under standard models) become equally likely.

In this section, we look at some handy representations of binary trees which can be useful further on.

Theorem 3.1.

Let $p_1, p_2, \ldots, p_{2n}$ be a balanced sequence of parentheses, i.e. each p_i belongs to $\{(,)\}$, for every partial sequence $p_1, p_2, \ldots, p_i$, the number of opening parentheses is at least equal to the number of closing parentheses, and in the entire sequence, there are an equal number of opening and closing parentheses.

Then there exists a one-to-one correspondence between all such balanced sequences of $2n$ parentheses and all binary trees with n nodes.

Proof of Theorem 3.1.

We will prove this constructively. Consider an inorder traversal of a binary tree, i.e. a traversal whereby each node is visited after its left subtree has been visited, but before its right subtree is visited. In the traversal, a stack S is used. Initially the root is pushed onto the stack. Then, a move to the left down the tree corresponds to another push. If there is no left subtree, we pop the stack and go the the right subtree if there is one (this requires yet another push). If there is no right subtree either, then we pop again, and so forth until we try to pop an empty stack. The algorithm is as follows:

Inorder stack traversal of a binary tree

[NOTE: The binary tree consists of n cells, each having a left and a right pointer field. S is a stack, and $p_1, \ldots, p_{2n}$ is the sequence of pushes (opening parentheses) and pops (closing parentheses) to be returned.]

$p \leftarrow$ root of tree (p is a pointer)
$i \leftarrow 2$ (i is a counter)
MAKENULL (S)
PUSH p onto S; $p_1 \leftarrow$(
REPEAT
    IF $p\uparrow$.Left$\neq$NIL
        THEN PUSH $p\uparrow$.Left onto S; $p \leftarrow p\uparrow$.Left; $p_i \leftarrow$(; $i \leftarrow i+1$
        ELSE
            REPEAT
                POP S, yielding p; $p_i \leftarrow$); $i \leftarrow i+1$
            UNTIL $i > 2n$ OR $p\uparrow$.Right$\neq$NIL
            IF $i \leq 2n$
                THEN PUSH $p\uparrow$.Right onto S; $p \leftarrow p\uparrow$.Right; $p_i \leftarrow$(; $i \leftarrow i+1$
UNTIL $i > 2n$
RETURN $p_1, \ldots, p_{2n}$

Different sequences of pushes and pops correspond to different binary trees. Also, every partial sequence of pushes and pops is such that the number of pushes is at least equal to the number of pops. Upon exit from the algorithm, both numbers are of course equal. Thus, if a push is identified with an opening parenthesis, and a pop with a closing parenthesis, then the equivalence claimed in the theorem is obvious. ■

For example, the sequence $()()()() \cdots ()$ corresponds to a binary tree in which all nodes have only right subtrees. And the sequence $(((((\cdots)))))$ corresponds to a binary tree in which all nodes have only left subtrees. The representation of a binary tree in terms of a balanced sequence of parentheses comes in very handy. There are other representations that can be derived from Theorem 3.1.

Theorem 3.2.

There is a one-to-one correspondence between a balanced sequence of $2n$ parentheses and a random walk of length $2n$ which starts at the origin and returns to the origin without ever crossing the zero axis.

Proof of Theorem 3.2.

Let every opening parenthesis correspond to a step of size "+1" in the random walk, and let every closing parenthesis correspond to a step of size "-1" in the random walk. Obviously, such a random walk returns to the origin if the string of parentheses is balanced. Also, it does not take any negative values. ■

Theorem 3.2 can be used to obtain a short proof for counting the number of different (i.e., nonsimilar) binary trees with n nodes.

Theorem 3.3.

There are

$$\frac{1}{n+1}\binom{2n}{n}$$

different binary trees with n nodes.

Proof of Theorem 3.3.

The proof uses the celebrated mirror principle (Feller, 1965). Consider a random walk starting at $(2k,0)$ ($2k \geq 0$ is the initial value; 0 is the initial time): in one time unit, the value of the random walk either increases by 1 or decreases by 1. The number of paths ending up at $(0,2n)$ which take at least one negative value is equal to the number of unrestricted paths from $(2k,0)$ to $(-2,2n)$. This can most easily be seen by the following argument: there is a one-to-one correspondence between the given restricted and unrestricted paths. Note that each restricted path must take the value -1 at some point in time. Let t be the first time that this happens. From the restricted path to $(0,2n)$, construct an unrestricted path to $(-2,2n)$ as follows: keep the initial segment up to time t, and flip the tail segment between time t and time $2n$ around, so that the path ends up at $(-2,2n)$. Each different restricted path yields a different unrestricted path. Vice versa, since the unrestricted paths must all cross the horizontal line at -1, time t is well defined, and each unrestricted path corresponds to a restricted path.

The number of paths from $(2k,0)$ to $(0,2n)$ which do not cross the zero axis equals the total number of unrestricted paths minus the number of paths that do

cross the zero axis, i.e.

$$\binom{2n}{k+n} - \binom{2n}{k+n+1},$$

which is easily seen by using a small argument involving numbers of possible subsets. In particular, if we set $k=0$, we see that the total number of binary trees (or the total number of nonnegative paths from (0,0) to $(0,2n)$) is

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

The number of binary trees with n nodes grows very quickly with n (see table below).

n	Number of binary trees with n nodes
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	3430

One can show (see exercises) that this number $\sim 4^n/(\sqrt{\pi}n^{3/2})$. Because of this, the decoding method seems once again impractical except perhaps for n smaller than 15, because of the wordsize of the integers involved in the computations.

3.2. Generation by rejection.

Random binary trees or random strings of balanced parentheses can be generated by the rejection method. This could be done for example by generating a random permutation of n opening parentheses and n closing parentheses, and accepting only if the resulting string satisfies the property that all partial substrings have at least as many opening parentheses as closing parentheses. There are

$$\binom{2n}{n}$$

initial strings, all equally likely. By Theorem 3.3, the probability of acceptance of a string is thus $\frac{1}{n+1}$. Furthermore, to decide whether a string has the said

property takes expected time proportional to n. Thus, the expected time taken by the algorithm varies as n^2. For this reason, the rejection method is not recommended.

3.3. Generation by sequential sampling.

It is possible to generate a random binary tree with n nodes in time $O(n)$ by first generating a random string of balanced parentheses of length $2n$ in time $O(n)$ and then reconstructing the tree by mimicking the inorder traversal given in the proof of Theorem 3.1. The string can be generated in one pass, from left to right, similar to the sequential sampling method for generating a random subset. It is perhaps best to consider the analogy with random walks once again. We start at (0,0), and have to end up at $(0,2n)$. At each point, say (k,t), we decide to generate a (with probability equal to the ratio of the number of nonnegative paths from $(k+1,t+1)$ to $(0,2n)$ to the number of nonnegative paths from (k,t) to $(0,2n)$. We generate a) otherwise. It is clear that this method uses a recurrence relation for binary trees, but the explanation given here in terms of random walks is perhaps more insightful. The number of nonnegative paths from (k,t) to $(0,2n)$ is (see the proof of Theorem 3.3):

$$\binom{2n-t}{\frac{k+2n-t}{2}} - \binom{2n-t}{\frac{k+2+2n-t}{2}} = \binom{2n-t}{\frac{k+2n-t}{2}} \frac{2k+2}{2n-t+k+2}.$$

The probability of a (at (k,t) is thus

$$\frac{\binom{2n-t-1}{\frac{k+2n-t}{2}} \frac{2k+4}{2n-t+k+2}}{\binom{2n-t}{\frac{k+2n-t}{2}} \frac{2k+2}{2n-t+k+2}} = \frac{k+2}{k+1} \frac{2n-t-k}{2(2n-t)}.$$

The resulting algorithm for generating a random string of balanced parentheses is due to Arnold and Sleep (1980):

Sequential method for generating a random string of balanced parentheses

[NOTE: The string generated by us is returned in $p_1,\ldots,p_{2n}$.]
$X \leftarrow 0$ (X holds the current "value" of the corresponding random walk.)
FOR $t := 0$ TO $2n-1$ DO
    Generate a uniform [0,1] random variate U.
    IF $U \leq \frac{X+2}{X+1}\frac{2n-t-X}{2(2n-t)}$
        THEN $X \leftarrow X+1$, $p_{t+1} \leftarrow ($
        ELSE $X \leftarrow X-1$, $p_{t+1} \leftarrow)$
RETURN $p_1, \ldots, p_{2n}$

It is relatively straightforward to check that the random walk cannot take negative values because when $X=0$, the probability of generating (in the algorithm is 1. It is also not possible to overshoot the origin at time $2n$ because whenever $X=2n-t$, the probability that a (is generated is 0.

The reconstruction in linear time of a binary tree from a string of balanced parentheses is left as an exercise to the reader. Basically, one should mimic the algorithm of Theorem 3.1 where such a string is constructed given a binary tree.

3.4. The decoding method.

There are a number of sophisticated coding functions for binary trees, which can be decoded in linear time, but all of them require extra storage space for auxiliary constants. See e.g. Knott (1977), Ruskey (1978), Ruskey and Hu (1977) and Trojanowski (1978). See also Tinhofer and Schreck (1984).

3.5. Exercises.

1. Show that the number of binary trees with n nodes $\sim \frac{4^n}{\sqrt{\pi} n^{\frac{3}{2}}}$.

2. Consider an arbitrary (unrestricted) random walk from (0,0) to (0,$2n$) (this can be generated by generating a random permutation of n 1's and n -1's). Define another random walk by taking the absolute value of the unrestricted random walk. This random walk does not take negative values, and corresponds therefore to a string of balanced parentheses of length $2n$. Show that the random strings obtained in this manner are not uniformly

distributed.

3. Give a linear time algorithm for reconstructing a binary tree from a string of balanced parentheses of length $2n$ using the correspondence established in Theorem 3.1.

4. **Random rooted trees.** A rooted tree with n vertices consists of a root and an ordered collection of nonempty rooted subtrees when $n>1$. When $n=1$, it consists of just a root. The vertices are unlabeled. Thus, there are 5 different rooted trees when $n=4$. There are a number of representations of rooted trees, such as:

   A. A vector of degrees: write down for each node the number of children (nonempty subtrees) when the tree is traversed in preorder or level order.

   B. A vector of levels: traverse the tree in preorder or postorder and write down the level number of each node when it is visited.

   We can call these vectors of length n codewords. There are other more storage-efficient codewords: find a codeword of length $2n$ consisting of bits only, which uniquely represents a rooted tree. Show that all codewords for representing rooted trees or binary trees must take at least $(2+o(1))n$ bits of storage. Generating a codeword is equivalent to generating a rooted tree. Pick any codeword you like, and give a linear time algorithm for generating a valid random codeword such that all codewords are equally likely to be generated. Hint: notice the connection between rooted trees and binary trees.

5. Let us grow a tree by replacing on a sequential basis all NIL pointers by new nodes, where the choice of a NIL pointer is uniform over the set of such pointers (see section 3.1). Note that there are $n+1$ NIL pointers if the tree has n nodes. Let us generate another tree by generating a random permutation and constructing a binary search tree. Are the two trees similar in distribution, i.e. is it true that for each shape of a tree with n nodes, and for all n, the probability of a tree with that shape is the same under both schemes ? Prove or disprove.

6. Find a coding function for binary trees which can be decoded in time $O(n)$.

4. RANDOM PARTITIONS.

4.1. Recurrences and codewords.

Many problems can be related to the generation of random partitions of $\{1, \ldots, n\}$ into k nonempty subsets. We know that there are $\left\{ {n \atop k} \right\}$ such partitions, where $\{.\}$ denotes the Stirling number of the second kind. Rather than give a formula for the Stirling numbers in terms of a series, we will employ the

recursive definition:

$$\left\{ {n \atop k} \right\} = k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\} \quad (0<k<n)\,,$$

$$\left\{ {n \atop 1} \right\} = 1\,; \left\{ {n \atop n} \right\} = 1\,.$$

Using this, we can form a table of Stirling numbers, just as we can form a table (Pascal's triangle) from the well-known recursion for binomial numbers. We have:

$n=$	1	2	3	4	5	6
$k=$						
1	1	1	1	1	1	1
2		1	3	7	15	31
3			1	6	25	90
4				1	10	65
5					1	15
6						1

The recursion has a physical meaning: we can form a partition into k nonempty subsets by considering a partition of $\{1, \ldots, n-1\}$ and adding one number, n. That number n can be considered as a new singleton set in the partition (this explains the contribution

$$\left\{ {n-1 \atop k-1} \right\}$$

in the recursion). It can also be added to one of the sets in the partition of $\{1, \ldots, n-1\}$. In this case, we can add it to one of the k sets in the latter partition. To have a unique way of addressing these sets, we order the sets according to the value of their smallest elements, and label the sets $1,2,3, \ldots, k$. The addition of n to set i implies that we must include

$$\left\{ {n-1 \atop k} \right\}$$

in the recursion.

Before we proceed with the generation of a random partition based upon this recursion, it is perhaps useful to describe one kind of codeword for random partitions. Consider the case $n=5$ and $k=3$. Then, the partition (1,2,5),(3),(4) can be represented by the n-tuple 11231 where each integer in the n-tuple represents the set to which each element belongs. By convention, the sets are ordered according to the values of their smallest elements. So it is easy to see that different codewords yield different partitions, and vice versa, that all n-tuples of integers from $\{1, \ldots, k\}$ (such that each integer is used at least once) having this ordering property correspond to some partition into k nonempty subsets. Thus, generating random codewords or random partitions is equivalent. Also, one can be constructed from the other in time $O(n)$.

4.2. Generation of random partitions.

The generator described below produces a random codeword, uniformly distributed over the collection of all possible codewords. It is based upon the recursion explained above. To add n to a partition of $\{1, \ldots, n-1\}$, we should define a singleton set $\{n\}$ (in which case it must have set number k) with probability

$$\frac{\left\{ {n-1 \atop k-1} \right\}}{\left\{ {n \atop k} \right\}}$$

and add it to a randomly picked set from among $1, \ldots, k$ with probability

$$\frac{\left\{ {n-1 \atop k} \right\}}{\left\{ {n \atop k} \right\}}$$

each. Obviously, we have to generate the random codeword backwards.

Random partition generator based upon recurrence relation for Stirling numbers

[NOTE: n and k are given and will be destroyed.]
REPEAT
  Generate a uniform [0,1] random variate U.
  IF $U \le \dfrac{\left\{ {n-1 \atop k-1} \right\}}{\left\{ {n \atop k} \right\}}$
    THEN $X_n \leftarrow k$, $k \leftarrow k-1$
    ELSE Generate X_n uniformly on $1, \ldots, k$
  $n \leftarrow n-1$
UNTIL $n = 0$
RETURN the codeword $X_1, X_2, \ldots, X_n$

If the Stirling numbers can be computed in time $O(1)$ (for example, if they are stored in a two-dimensional table), then the algorithm takes time $O(n)$ per codeword. The storage requirements are proportional to nk. The preprocessing time needed to set up the table of size n by k is also proportional to nk if we use the fundamental recursion.

We conclude this section by noting that the algorithm given above is a slightly modified version of an algorithm given in Wilf (1977) and Nijenhuis and Wilf (1978).

4.3. Exercises.

1. Define a coding function for random partitions, and find an $O(n)$ decoding algorithm.
2. **Random partitions of integers.** Let $p(n,k)$ be the number of partitions of an integer n such that the largest part is k. The following recurrence holds:

$$p(n,k) = p(n-1,k-1)+p(n-k,k) .$$

The first term on the right-hand side represents those partitions of n whose largest part is k and whose second largest part is less than k (because such partitions can be obtained from one of $n-1$ whose largest part is $k-1$ by adding 1 to the largest part). The partitions of n whose largest two parts are both k come from partitions of $n-k$ of largest part k by replicating the largest part. Arguing as in Wilf (1977), a partition is a series of decisions "add 1 to the largest part" or "adjoin another copy of the largest part".

A. Give an algorithm for the generation of such a random partition (all partitions should be equally likely of course), based upon the given recurrence relation.

B. Find a coding function for these partitions. Hint: base your function on the parts of the partition given in descending order.

C. How would you generate an unrestricted partition of n ? Here, unrestricted means that no bound is given for the largest part in the partition.

D. Find a recurrence relation similar to the one given above for the number of partitions of n with parts less than or equal to k.

E. For the combinatorial objects of part D, find a coding function and a decoding algorithm for generating a random object. See also McKay (1965).

5. RANDOM FREE TREES.

5.1. Prufer's construction.

A free tree is a connected graph with no cycles. If there are n nodes, then there are $n-1$ edges. The distinction between labeled and unlabeled free trees is important. Note however that unlike other trees, free trees do not have a given root. All nodes are treated equally. We will however keep using the term leaf for nodes with degree one.

The generation of a random free tree can be based upon the following theorem:

Theorem 5.1.

Cayley's theorem. There are exactly n^{n-2} labeled free trees with n nodes.

Prufer's construction. There exists a one-to-one correspondence between all $(n-2)$-tuples ("codewords") of integers $a_1, \ldots, a_{n-2}$, each taking values in $\{1, \ldots, n\}$, and all labeled free trees with n nodes. The relationship is given in the proof below.

Proof of Theorem 5.1.

Cayley's theorem follows from Prufer's construction. Let the nodes of the labeled free tree have labels $1, \ldots, n$. From a labeled free tree a codeword can be constructed as follows. Let a_1 be the label of the neighbor of the leaf with the smallest label. Delete the corresponding edge. Since one of the endpoints of the edge is a leaf, removal of the edge will leave us with a labeled free tree of size $n-1$. Repeat this process until $n-2$ components of the codeword have been calculated. At the end, we have a labeled free tree with just 2 nodes, which can be discarded. For example, for the labeled free tree with 6 nodes and edges (1,2), (2,3), (4,3), (5,3), (6,3), the codeword (2,3,3,3) is obtained.

Conversely, from each codeword, we can construct a free tree having the property that if we use the construction given above, the initial codeword is obtained again. This is all that is needed to establish the one-to-one correspondence. For the construction of the tree from a given codeword, we begin with three lists:

A. The codeword: $a_1, \ldots, a_{n-2}$.

B. A list of n flags: $f_1, \ldots, f_n$, where $f_i = 1$ indicates that node i is available. Initially, all flags are 1. Flag i is set to 0 only when i is a leaf, and the edge connected to i is suddenly removed from the tree.

C. A list of n flags indicating whether a node is a leaf or not: $l_1, \ldots, l_n$. $l_i = 1$ indicates that node i is a leaf. Note that this list is redundant, since a node is a leaf if and only if its label can be found in the codeword. The initialization of this list of flags is simple.

The construction proceeds by first recreating the $n-2$ edges that correspond to the $n-2$ components of the codeword. This is done simply as follows: choose node a_1 (this is not a leaf, since it is in the codeword), and choose the smallest leaf v (flag $l_v = 1$ and availability flag $f_v = 1$). Return the edge (a_1, v), and set the flag of v to 0, which effectively eliminates v. If a_1 cannot be found in the remainder of the codeword, then a_1 becomes a leaf in the new free tree, and the flag l_{a_1} must be set to 1. This process can be repeated until $a_1, \ldots, a_{n-2}$ is exhausted. The last $(n-1\text{-st})$ edge at the end is simply found by taking the only two nodes whose availability flags are still 1. This concludes the construction. It is easy to verify that if the tree is used to construct a codeword, the initial codeword is obtained. ■

The degree of a node is one plus the number of occurrences of the node in the codeword, at least if codewords are translated into free trees via Prufer's construction. To generate a random labeled free tree with n nodes (such that all such trees are equally likely), one can proceed as follows:

Random labeled free tree generator

FOR $i := 1$ TO $n-2$ DO
    Generate a_i uniformly on $\{1, \ldots, n\}$.
Translate the codeword into a labeled free tree via Prufer's construction.

A careless translation of the codeword could be inefficient. For example, the verification of whether an internal node becomes a leaf during construction, when done by traversing the leftover part of the codeword, yields an $\Omega(n^2)$ contribution to the total time. Using linear search to find the smallest available leaf would give a contribution of $\Omega(n^2)$ to the total time. Even if a heap were used for this, we would still be facing a contribution of $\Omega(n \log(n))$ to the total time. In the next section, a linear time translation algorithm due to Klingsberg (1977) is presented.

5.2. Klingsberg's algorithm.

The purpose of this section is to explain Klingsberg's $O(n)$ algorithm for translating a codeword $a_1, \ldots, a_{n-2}$ into a labeled free tree. His solution requires one additional array $T[1], \ldots, T[n]$, which is used to return the edges and to keep information about the availability flags and about the leaf flags (see proof of Theorem 1). The edges returned are

$$(1, T[1]), (2, T[2]), \ldots, (n-1, T[n-1]) .$$

The other uses of this array are:

A. $T[i]$=available_not_leaf means that node i is still available and is not a leaf. The constant is set to -1 in Klingsberg's work.

B. $T[i]$=available_leaf means that node i is an available leaf. The constant is set to 0 in Klingsberg's work.

C. $T[i]=j>0$ indicates that node i is no longer available, and in fact that (i, j) is an edge of the labeled free tree.

In the example of codeword (2,3,3,3) given in section 5.1, the array T would initially be set to (available_leaf , available_not_leaf , available_not_leaf , available_leaf , available_leaf , available_leaf) since only nodes 2 and 3 are internal nodes.

To speed up the determination of when an internal node becomes a leaf, we merely flag the last occurrence of every node in the codeword. This can conveniently be done by changing the signs of these entries. In our example, the codeword would initially be replaced by (–2,3,3,–3).

Finally, to find the smallest available leaf quickly, we note that in the construction, these leaf labels increase except when a new leaf is added, and its label is smaller than the current smallest leaf label. This can be managed with the aid of two moving pointers: there is a masterpointer which moves up monotonically from 1 to n; in addition, there is a temporary pointer, which usually moves with the masterpointer, except in the situation described above, when it is temporarily set to a value smaller than that of the masterpointer. The temporary pointer always points at the smallest available leaf. It is this ingenious device which permitted Klingsberg to obtain an $O(n)$ algorithm. We can now summarize his algorithm.

Klingsberg's algorithm for constructing a labeled free tree from a codeword

```
[PREPARATION.]
FOR i:=1 TO n DO T[i]← available_leaf
FOR i:=n-2 DOWNTO 1 DO
    IF T[a_i]=available_leaf THEN
        T[a_i]=available_not_leaf; a_i ←-a_i
Master ←1
a_{n-1}←n (for convenience in defining last edge)
Master ←min(j:T[j]= available_leaf )
Temp ← Master
[TRANSLATION.]
FOR i:=1 TO n-1 DO
    Select ← | a_i |      (select internal node)
    T[Temp]← Select  (return edge)
    IF i<n-1 THEN
        IF a_i >0
            THEN
                Master ←min(j:T[j]= available_leaf )
                Temp ← Master
            ELSE
                T[Select]← available_leaf
                IF Select ≤ Master THEN Temp ← Select     (temporary step
                up)
RETURN (1,T[1]), . . . , (n-1,T[n-1])
```

The linearity of the algorithm follows from the fact that the masterpointer can only increase, and that all the operations in every iteration that do not involve the masterpointer are constant time operations.

5.3. Free trees with a given number of leaves.

Assume next that we wish to generate a labeled free tree with n nodes and l leaves where $2 \leq l \leq n-1$. For the solution of this problem, we recall Prufer's codeword. The codeword contains the labels of all internal nodes. Thus, it is necessary to generate only codewords in which precisely $n-l$ labels are present. The actual labels can be put in by selecting $n-l$ labels from n labels by one of the random sampling algorithms. Thus, we have:

Generator of a labeled free tree with n nodes and l leaves

Generate a random subset of $n-l$ labels from $1, \ldots, n$.

Perform a random permutation on these labels (this may not be necessary, depending upon the random subset algorithm.)

Generate a random partition of $n-2$ elements into $n-l$ non-empty subsets, and assign the first label to the first subset, etcetera. This yields a codeword of length $n-2$ with precisely $n-l$ different labels.

Translate the codeword into a labeled free tree (preferably using Klingsberg's algorithm).

In this algorithm, we need algorithms for random subsets, random partitions and random permutations. It goes without saying that some of these algorithms can be combined. Another by-product of the decomposition of the problem into manageable sub-problems is that it is easy to count the number of combinatorial objects. We obtain, in this example:

$$\left\{ n \atop l \right\} (n-l)! \left\{ n-2 \atop n-l \right\} = \frac{n!}{l!} \left\{ n-2 \atop n-l \right\} .$$

5.4. Exercises.

1. Let $d_1, \ldots, d_n$ be the degrees of the nodes $1, \ldots, n$ in a free tree. (Note that the sum of the degrees is $2n-2$.) How would you generate such a free tree ? Hint: generate a random Prufer codeword with the correct number of occurrences of all labels. The answer is extremely simple. Derive also a simple formula for the number of such labeled free trees.
2. Give an algorithm for computing the Prufer codeword for a labeled free tree with n nodes in time $O(n)$.
3. Prove that the number of free trees that can be built with n labeled edges (but unlabeled nodes) is $(n+1)^{n-2}$. Hint: count the number of free trees with n labeled nodes and $n-1$ labeled edges first.
4. Give an $O(n)$ algorithm for the generation of a random free tree with n labeled edges and $n+1$ unlabeled nodes. Hint: try to use Klingsberg's algorithm by reducing the problem to one of generating a labeled free tree.
5. **Random unlabeled free trees with n vertices.** Find the connection between unlabeled free trees with n vertices and rooted trees with n vertices. Exploit the connection to generate random unlabeled free trees such that all trees are equally likely (Wilf, 1981).

6. RANDOM GRAPHS.

6.1. Random graphs with simple properties.

Graphs are the most general combinatorial objects dealt with in this chapter. They have applications in nearly all fields of science and engineering. It is quite impossible to give a thorough overview of the different subclasses of graphs, and how objects in these subclasses can be generated uniformly and at random. Instead, we will just give a superficial treatment, and refer the reader to general principles or specific articles in the literature whenever necessary.

We will use the notation n for the number of nodes in a graph, and e for the number of edges in a graph. A random graph with a certain property P is such that all graphs with this property are equally likely to occur. Perhaps the simplest property is the property: "Graph G has n nodes". We know that there are

$$2^{\binom{n}{2}}$$

objects with this property. This can easily be seen by considering that each of the $\binom{n}{2}$ possible edges can either be present or absent. Thus, we should include each edge in a random graph with this property with probability 1/2.

The number of edges chosen is binomially distributed with parameters n and 1/2. It is often necessary to generate sparser graphs, where roughly speaking e is $O(n)$ (or at least not $\Omega(n^2)$). This can be done in two ways. If we do not require a specific number of edges, then the simplest solution is to select all edges independently and with probability p. Note that the expected number of edges is $p\binom{n}{2}$. This is most easily implemented, especially for small p, by using the fact that the waiting time between two selected edges is geometrically distributed with parameter p, where by "waiting time" we mean the number of edges we must manipulate before we see another selected edge. This requires a linear ordering on the edges, which can be done by the coding function given below.

If the property is "Graph G has n nodes and e edges", then we should first select a random subset of e edges for the set of $\binom{n}{2}$ possible edges. This property is simple to deal with. The only slight problem is that of establishing a simple coding function for the edges, which is easy to decode. This is needed since we have to access the endpoints of the edges some of the time (e.g., when returning edges), and the coded edges most of the time (e.g., when random sampling

based upon hashing). One possibility is shown below:

Node u	Node v	Coded version of edge (u,v)
1	2	1
1	3	2
...	...	...
...	...	...
1	n	$n-1$
2	3	$(n-1)+1$
...	...	...
...	...	...
2	n	$(n-1)+(n-2)$
...	...	...
...	...	...
$n-1$	n	$(n-1)+(n-2)+\cdots+2+1$

The coding function for this scheme is

$$f(u,v) = (u-1)n - \frac{u(u-1)}{2} + (v-u).$$

Interestingly, this function can be decoded in time $O(1)$ (see exercise 6.1). Whether random sampling should be done on coded integers with decoding only at the very end, or on sets of edges (u,v) without any decoding, depends upon the sampling scheme. In classical sampling schemes for example, it is necessary to verify whether a certain edge has already been selected. The verification can be based upon a vector of flags (which can be done here by using a lower triangular n by n matrix of flags). When a heap or a tree structure is used, there is no need ever for coding. When hashing is used, coding seems appropriate. In sequential sampling, no coding is needed, as long as we can easily implement the function NEXT(u,v) (IF $v=n$ THEN NEXT$(u,v)\leftarrow(u+1,u+2)$ ELSE NEXT$(u,v)\leftarrow(u,v+1)$). However, if sequential sampling is accelerated by taking giant steps, then coding the edges seems the wise thing to do.

6.2. Connected graphs.

Most random graphs that people want to generate should be of the connected type. From the work of Erdos and Renyi (1959, 1960), we know that if e is much larger than $\frac{1}{2}n\log(n)$ (or if p is much larger than $\frac{\log(n)}{n}$), then the probability that a random graph with e (or binomial (n,p)) edges is connected tends to 1 as $n\rightarrow\infty$. In those situations, it is clear that we could use the rejection algorithm:

Rejection method for generating a connected random graph with n nodes and e edges

REPEAT
    Generate a random graph G with e edges and n nodes.
UNTIL G is connected
RETURN G

To verify that a graph is connected is a standard operation: if we use depth first search, this can be done in time $O(\max(n,e))$ (Aho, Hopcroft and Ullman, 1983). Thus, the expected time needed by the algorithm is $O(\max(n,e))$ when

$$\liminf_{n\to\infty} \frac{e}{n\log(n)} > \frac{1}{2}.$$

In fact, since in those cases the probability of acceptance tends to 1 as $n\to\infty$, the expected time taken by the algorithm is $(1+o(1))$ times the expected time needed to check for connectedness and to generate a random graph with e edges. Unfortunately, the condition given above is asymptotic, and it is difficult to verify whether for given values of e and n, we have a good rejection constant. Also, there is a gap for precisely the most interesting sorts of graphs, the very sparse graphs when e is of the order of n. This can be done via a general graph generation technique of Tinhofer's (1978,1980), which is explained in the next section. In it, we recognize ingredients of Wilf's recurrence based method.

6.3. Tinhofer's graph generators.

In two publications, Tinhofer (1978,1980) has proposed useful random graph generators, with applications to connected graphs (with or without a specific number of edges), digraphs, bichromatic graphs, and acyclic connected graphs. His algorithms require in all cases that we can count certain subclasses of graphs, and they run fastest if tables of these counts can be set up beforehand. We will merely give the general outline, and refer to Tinhofer's work for the details.

Let us represent a graph by a sequence of adjacency lists, with the property that each edge should appear in only one adjacency list. The adjacency list for node i will be denoted by A_i. Thus, the graph is completely determined by the sequence

$$A_1 A_2 \cdots A_n .$$

We will generate the adjacency lists in some (usually random) order, $A_{v_1}, A_{v_2}, \ldots,$ where $v_1, v_2, \ldots, v_n$ is a permutation of $1, \ldots, n$. To avoid

the duplication of nodes, we require that all nodes in adjacency list A_{v_j} fall outside $\bigcup_{i=1}^{j} \{v_i\}$. The following sets of nodes will be needed:

1. The set U_j of all nodes in $\bigcup_{i=1}^{j} A_{v_i}$ with label not in $v_1, \ldots, v_j$. This set contains all neighbors of the first j nodes outside $v_1, \ldots, v_j$.
2. The set V_j which consists of all nodes with label outside $v_1, \ldots, v_j$ that are not in U_j.
3. The special sets $U_0=\{1\}$, $V_0=\{2,3, \ldots, n\}$.

When the adjacency lists are being generated, it is also necessary to do some counting: define the quantity N_j as the total number of graphs with the desired property, having fixed adjacency lists $A_{v_1}, \ldots, A_{v_j}$. Sometimes we will write $N_j(A_{v_1}, \ldots, A_{v_j})$ to make the dependence explicit. Given $A_{v_1}, \ldots, A_{v_{j-1}}$, we should of course generate A_{v_j} according to the following distribution:

$$P(A_{v_j}=A) = \frac{N_j(A_{v_1}, \ldots, A_{v_{j-1}}, A)}{N_{j-1}(A_{v_1}, \ldots, A_{v_{j-1}})}.$$

It is easy to see that this is indeed a probability vector in A. We are now ready to give Tinhofer's general algorithm.

Tinhofer's random graph generator

$U_0 \leftarrow \{1\}$; $V_0 \leftarrow \{2, \ldots, n\}$
FOR $j:=1$ TO n DO
  IF EMPTY (U_{j-1})
    THEN $v_j \leftarrow \min(i : i \in V_{j-1})$
    ELSE $v_j \leftarrow \min(i : i \in U_{j-1})$
  Generate a random subset A_{v_j} on $U_{j-1} \cup V_{j-1} - \{v_j\}$ according to the probability distribution given above.
  $U_j \leftarrow U_{j-1} \cup A_j - \{v_j\}$
  $V_j \leftarrow V_{j-1} - A_j - \{v_j\}$
RETURN $A_{v_1}, A_{v_2}, \ldots, A_{v_{n-1}}$

The major problem in this algorithm is to compute (on-line) the probability distribution for A_{v_j}. In many examples, the probabilities depend only upon the cardinalities of U_{j-1} and V_{j-1} and possibly some other sets, and not upon the actual structure of these sets. This is the case for the class of all connected graphs with n nodes, or all connected graphs with n nodes and e edges (see Tinhofer, 1980). Nevertheless, we still have to count, and run into numerical problems when n or e are large.

6.4. Bipartite graphs.

A **bipartite graph** is a graph in which we can color all vertices with two colors (baby pink and mustard yellow) such that no two vertices with the same color are adjacent. There exists a useful connection with matrices which makes bipartite graphs a manageable class of graphs. If there are b baby vertices and m mustard vertices, then a bipartite graph is completely defined by a $b \times m$ incidence matrix of 0's and 1's. At this point we may recall the algorithms of section XI.6.3 for generating a random $R \times C$ table with given row and column totals. This leads directly to a rejection algorithm for generating a random bipartite graph with given degrees for all vertices:

Bipartite graph generator

[NOTE: This algorithm returns a $b \times m$ incidence matrix defining a random bipartite graph with b baby vertices and m mustard vertices. The row totals are r_i, $1 \le i \le b$, and the column totals are c_j, $1 \le j \le m$.]

REPEAT

Generate a random $R \times C$ matrix of dimension $b \times m$ with the given row and column totals.

UNTIL all elements in the matrix are 0 or 1

RETURN the matrix

The reduction to a random $R \times C$ matrix was suggested by Wormald (1984). By Wald's equation, we know that the expected time taken by the algorithm is equal to the product of the expected time needed to generate one random $R \times C$ matrix and the expected number of iterations. For example, if we use the ball-in-urn method of section XI.6.3, then a random $R \times C$ matrix can be obtained in time proportional to e, the total number of edges (which is also equal to $\sum r_i$ and to $\sum c_j$). The analysis of the expected number of iterations is also due to Wormald (1984):

Theorem 6.1.

Assume that all r_i's and c_j's are at most equal to k. The expected number of iterations in the rejection algorithm is

$$(1+o(1))\exp\left(\frac{2}{e^2}\sum_{i=1}^{b}\binom{r_i}{2}\sum_{j=1}^{m}\binom{c_j}{2}\right),$$

where e is the total number of edges, and $o(1)$ denotes a function tending to 0 as $e \to \infty$ which depends only upon k and not on the r_i's and b_j's.

As a corollary of this Theorem, we see that the expected number of iterations is uniformly bounded over all bipartite graphs whose degrees are uniformly bounded by some number k.

Bipartite graphs play a crucial role in graph theory partly because of the following connection. Consider a $b \times m$ incidence matrix for a bipartite graph in which all baby vertices have degree 2, i.e. all r_i's are equal to 2. This defines a graph on m nodes in the following manner: each pair of edges connected to a baby vertex defines an edge in the graph on m nodes. Thus, the new graph has b edges. We can now generate a random graph with given collection of degrees as follows:

Random graph generator

[NOTE: This algorithm returns an array of b edges defined on a graph with vertices $\{1, \ldots, m\}$. The degree sequence is $c_1, \ldots, c_m$.]

REPEAT

Generate a random $b \times m$ bipartite graph with degrees all equal to two for the baby vertices ($r_i = 2$), and degrees equal to $c_1, \ldots, c_m$ for the mustard vertices.

UNTIL no two baby vertices share the same two neighbors

RETURN $(k_1, l_1), \ldots, (k_m, l_m)$ where k_i and l_i are the columns of the two 1's found in the i-th row of the incidence matrix of the bipartite graph.

Again we use the rejection principle, in the hope that for many graphs the rejection constant is not unreasonable. Note that we need to check that there are no duplicate edges in the graph. This is done by checking that no two rows in the bipartite graph's incidence matrix are identical. It can be verified that the procedure takes expected time $O(b+m)$ where b is the number of edges in the graph, provided that all degrees of the vertices in the graph are bounded by a constant k (Wormald, 1984). In particular, the method seems to be ideally suited for generating random **r-regular graphs**, i.e. graphs in which all degrees are equal to r. It can be shown that the expected number of $R \times C$ matrices needed before halting is roughly speaking $e^{(r^2-1)/4}$. This increases rapidly with r. Wormald also gives a particular algorithm for generating 3-regular, or cubic, graphs.

6.5. Exercises.

1. Find a simple $O(1)$ decoding rule for the coding function for edges in a graph given in the text.
2. Prove Theorem 6.1.
3. Prove that if random graphs with b edges and m vertices are generated by Wormald's method, then, provided that all degrees are bounded by k, the expected time is $O(b+m)$. Give the details of all the data structures involved in the solution.
4. **Event simulators.** We are given n events with the following dependence structure. Each individual event has probability p of occurring, and each pair of events has probability q of occurring. All triples carry probability zero. Determine the allowable values for p, q. Also indicate how you would handle one simulation. Note that in one simulation, we have to report all the indices of events that are supposed to occur. Your procedure should have constant expected time.
5. **Random strings in a context-free language.** Let S be the set of all strings of length n generated by a given context-free grammar. Assume that the grammar is unambiguous. Using at most $O(n^{r+1})$ space where r is the number of nonterminals in the grammar, and using any amount of preprocessing time, find a method for generating a uniformly distributed random string of length n in S in linear expected time. See also Hickey and Cohen (1983).