A SIMPLE SEQUENCING AND RANKING METHOD THAT WORKS ON ALMOST ALL GRAY CODES

Timothy R. Walsh, Department of Mathematics and Computer Science, UQAM
P.O. Box 8888, Station A, Montreal, Quebec, Canada H3C-3P8

Abstract: We present a method of deriving non-recursive sequencing and ranking algorithms for any list of words which obeys the condition that all the words in the list with a common prefix are consecutive in the list. We also generalize to these lists the Bitner-Ehrlich-Reingold method of designing loop-free generation algorithms. We first apply these methods to the recursively-described Ruskey-Proskurowski Gray code for balanced parenthesis systems. We thus find two iterative sequencing algorithms, of which one uses constant extra space and the other is loop-free, for generating balanced parenthesis systems. We also find algorithms which require constant extra space and a linear number of arithmetic operations for ranking and unranking balanced parenthesis systems, both in lexicographical order and in Gray code order. Next, we consider some simpler Gray codes: the classical Gray code for subsets, the Nijenhuis-Wilf Gray codes for combinations and permutations, the Knuth-Klingsberg Gray code for integer compositions and the Knuth-Kaye Gray code for set partitions (which cannot be made loop-free). We present a simple constant-extra-space constant-time implementation of the Nijenhuis-Wilf algorithm for generating the k-subsets of an n-set, and we modify it to obtain a simple constant-extra-space constant-time algorithm for generating the compositions of an integer. We also show how already-published iterative sequencing algorithms for all these Gray codes could have been more simply derived using our method, we design loop-free versions wherever it is possible, and we derive new ranking and unranking algorithms. Finally, we present an implementation of the Hinz algorithm for moving the rings of the Towers of Hanoi puzzle from one arbitrary legal position to another which uses constant extra space and constant time except for a linear number of operations scattered throughout the entire algorithm.

0. Introduction

While teaching a graduate course in combinatorial algorithms, in which [NW] and [Wi] are required textbooks, I discovered a systematic method of deriving iterative sequencing, ranking and unranking algorithms for recursively-described Gray codes which was easier to explain to the students than the various methods presented in these texts and in some of the articles they reference. I applied it to all the simpler Gray codes in these texts: the classical Gray code for subsets [Gr], the ones for combinations and permutations in [NW], and Knuth's recursively-described Gray codes for compositions of an integer and partitions of a set which were announced in [NW], and in each case it was an easy matter to come up with non-recursive sequencing algorithms which were close to those already in the literature (see [Kl] for integer combinations, [Ka] for set partitions, and [NW] for the others), to make loop-free versions of these algorithms (except for the set-partition Gray code which uses an array in which arbitrarily many elements can change from one partition to the next), to derive efficient ranking and unranking algorithms - and to explain the process of deriving these results to the class. Later I applied it to a more challenging Gray code for which only a recursive description was published - the Proskurowski-Ruskey Gray code for balanced parenthesis systems [PR] - and obtained similar results which I tested by computer.

In Section 1 of this paper we describe this method and show that it works on any list of words in which all the words with a common prefix are consecutive in the list. This condition generalizes the graylex order defined in [Ch] and used there to enable a computer to derive Gray codes; here we take the Gray code as a given. The Bitner-Ehrlich-Reingold method of deriving loop-free algorithms for many of the classical Gray codes [BER] was generalized in [Wm] and [JWW]; here we generalize it further to all the lists obeying this condition and show that it works under only slightly more restrictive conditions. In Section 2 we apply these methods to the recursively-described Proskurowski-Ruskey Gray code for balanced parenthesis systems [PR]. We thus obtain two non-recursive sequencing algorithms of which one uses constant extra space and the other is loop-free. We also derive algorithms which require constant extra space and a linear number of arithmetic operations for ranking and unranking balanced parenthesis systems, both in lexicographical order (improving the results in [Pa], [Pl], [RH] and [Z]) and in Gray code order. In Section 3 we apply this method to the other above-mentioned Gray codes and present constant-time, constant-extra-space generators of the k-subsets of an n-set and of the k-compositions of n which we believe to be simpler than any of their competitors. We also mention a number of other Gray codes in the literature to which the method could be applied. In Section 4 we apply it to the Towers of Hanoi puzzle and two of its generalizations, including Hinz' algorithm [Hi] for moving the rings from one arbitary legal position to another. We present a constant-extra-space implementation of Hinz' algorithm which is loop-free except for a linear number of operations scattered throughout the entire algorithm. In Section 5 we mention a Gray code to which the method cannot be applied and pose an open problem. The Appendix contains a listing of the computer programs for sequencing, ranking and unranking the balanced parenthesis systems in the Proskurowski-Ruskey Gray code.

1. The method and the lists on which it works

Let $L=(w_0,w_1,...,w_{\#(L)-1})$ be a list of distinct combinatorial objects; $\#(L)$ is the number of objects in L. The *sequencing* problem is: given an object w_r in the list, either determine that it is the last object $w_{\#(L)-1}$ or else find the next object w_{r+1}. The *ranking* problem is: given an object w_r in the list, determine its *rank* r - that is, the number of objects which precede it in the list. The *unranking* problem is: given a non-negative integer $r<\#(L)$, determine the object w_r.

If one has an algorithm which solves the sequencing problem for a list L, the entire list can be generated by successively applying the sequencing algorithm to w_0. Suppose a given sequencing algorithm maintains some auxiliary variables. The *resumption* problem is: given an object w in L, determine the values that all the auxiliary variables used by the algorithm will have just after w has been generated so that the generation of L can be resumed from that point. For example, suppose L to be the list of positions assumed by the rings of the Towers of Hanoi puzzle as the unique minimal-move solution is being executed. To make the next move of the standard iterative sequencing algorithm (see Figure 4.2) one has to examine the topmost ring on each peg to find the smallest and the second smallest one and to remember if the smallest one is now to be moved. This last bit of information is stored in an auxiliary variable. If the monk who has been executing this algorithm suddenly dies, his successor will have to determine from the current position of the rings whether to resume the solution by moving the smallest ring, and this information can be determined in O(1) time [Wa1].

Suppose now that the objects in L are length-n *words*: strings $g_1g_2...g_n$ of letters from a finite alphabet A (we can remove the restriction that all the words be of the same length by padding the shorter ones on the right with blanks).

A linear order imposed on A (with the blank, if it is used, preceeding the first letter of A) induces the following criterion for L to be in lexicographical order: all the words with a common prefix $g_1g_2...g_{i-1}$ are consecutive in L, and the values assumed by the next letter g_i increase monotonely as we traverse the interval of words with that prefix. The distinct values assumed by g_i are not necessarily consecutive in A: if L is the list of permutations of $\{1,2,...,n\}$ in lexicographical order, then for any prefix $g_1g_2...g_{i-1}$, g_i traverses the set $\{1,2,...,n\}-\{g_1,g_2,...,g_{i-1}\}$ sorted in increasing order (see the last column of Figure 3.2). In this example the set of distinct values assumed by g_i is determined by the prefix $g_1g_2...g_{i-1}$, and this is true for any lexicographical order. The concept of lexicographical order was generalized to *graylex order* in [Ch]: all the words with a common prefix $g_1g_2...g_{i-1}$ are consecutive in L, the values assumed by the next letter g_i either increase monotonely or decrease monotonely as we traverse the interval of words with that prefix, and the prefix determines both the set of distinct values assumed by g_i and the *direction of motion of g_i* - that is, whether g_i increases or decreases (see the first column of Figure 3.2 and Figure 3.5).

We generalize further by dropping all restrictions on the values assumed by g_i, so that A does not even have to be linearly ordered. We say that L is in *generalized lexicographical order*, abbreviated *genlex order*, if all the words in L with a common prefix are consecutive in L. This is sufficient to ensure that for any prefix $g_1g_2...g_{i-1}$ the sequence of distinct values assumed by g_i as we traverse the interval of words with that prefix is determined by the prefix; so we can treat that sequence as a function $s(g_1g_2...g_{i-1})$ of $g_1g_2...g_{i-1}$. The list

11,13,12,14,21,22,23,24,31,33,32,34,41,42,43,44

is in genlex order but not in graylex order; $s(g_1g_2...g_{i-1})=(1,2,3,4)$ if $g_1+g_2+...+g_{i-1}$ is even and (1,3,2,4) otherwise.

For any list L in genlex order there is a conceptually simple sequencing algorithm. Given any prefix $g_1g_2...g_{i-1}$, $i<n$, we define a_i to be the first value in $s(g_1g_2...g_{i-1})$, z_i to be the last value in $s(g_1g_2...g_{i-1})$, and $h(g_i)$ to be the successor of g_i in $s(g_1g_2...g_{i-1})$ if $g_i \neq z_i$ (bearing in mind that a_i, z_i, and $h(g_i)$ all depend upon $g_1g_2...g_{i-1}$). Then $z_1z_2...z_n$ is the last object in L and, given any other object $g_1g_2...g_{i-1}g_iz_{i+1}...z_n$, where $g_i \neq z_i$, its successor in L is $g_1g_2...g_{i-1}h(g_i)a_{i+1}...a_n$. The *generic sequencing algorithm*, then, consists of scanning a word $g_1g_2...g_n$ from right to left until we either determine that it is $z_1z_2...z_n$ or else find the rightmost $g_i \neq z_i$ in which case we then replace $g_iz_{i+1}...z_n$ by $h(g_i)a_{i+1}...a_n$. For this algorithm to be practical, of course, the sequence $s(g_1g_2...g_{i-1})$ must be an easily computable function of $g_1g_2...g_{i-1}$. The design of a simple sequencing algorithm for any list in genlex order is thus reduced to finding a simple rule, if one exists, for calculating $s(g_1g_2...g_{i-1})$ as a function of $g_1g_2...g_{i-1}$.

A *Gray code* is an infinite sequence of lists L(0),L(1),L(2),... such that L(n) consists of length-n words and each word (except the last one) in each list can be changed to its successor by changing a number of letters which is bounded by a constant independent of n. In this case one can hope to find a constant-time algorithm for replacing $g_iz_{i+1}...z_n$ by $h(g_i)a_{i+1}...a_n$. The scanning of the current word from right to left to find the rightmost $g_i \neq z_i$ can be avoided by maintaining an auxiliary array $e_0e_1e_2...e_n$ which keeps track of the maximal subwords $g_{j+1}...g_k$ which are equal to $z_{j+1}...z_k$ (we call such a maximal subword a *z-subword*). This trick was first used by Bitner, Ehrlich and Reingold to design loop-free algorithms for generating subsets, combinations, permutations, integer compositions and set-partitions ([Eh1],[Eh2],[BER]) and later generalized in [JWW] to any genlex order in which s(the successor of $g_1g_2...g_{i-1}$) is always the reverse of $s(g_1g_2...g_{i-1})$ and is never of length 1. In Figure 1.1 below we generalize it to any genlex order in which $a_i \neq z_i$ for any prefix $g_1g_2...g_{i-1}$.

```
i:=$e_n$;
IF i=0 THEN (* $g_1g_2...g_n = z_1z_2...z_n$ *)
  Done := TRUE (* Done is a BOOLEAN variable which says to quit generating *)
ELSE (* i is the index of the rightmost $g_i \neq z_i$ *)
  Replace $g_i z_{i+1}...z_n$ by $h(g_i) a_{i+1}...a_n$;
  Update any other auxiliary variables as needed;
  $e_n$:=n;
  IF $g_i$=$z_i$ THEN
    $e_i$:=$e_{i-1}$;
    $e_{i-1}$:=i-1
  END IF
END IF.
```

Figure 1.1: Generic loop-free sequencing algorithm.
(Initially - that is, for the first word - g_j=a_j and e_j=j for all j and Done=FALSE)

It is clear that if each of the statements of this algorithm can be executed in constant time, then so can the entire algorithm. We prove its correctness by solving the resumption problem for the array $e_0e_1e_2...e_n$.

Theorem. Suppose that L is any list of length-n words in genlex order such that $a_i \neq z_i$ for any prefix $g_1g_2...g_{i-1}$. Then the algorithm of Figure 1.1 preserves the property that e_0=0 and for every k>0, e_k=j if $g_{j+1}...g_k$ is a z-subword and e_k=k otherwise.

Proof. It is easy to verify that the conclusion is true initially (this follows from the initial values and the fact that $a_i \neq z_i$ for each i so that there are no z-subwords), and we suppose it to be true at the beginning of a given execution of the algorithm. In particular, setting k=n we find that if $e_n \neq 0$ then it is the index of the rightmost $g_i \neq z_i$ and if e_n=0 then $g_1g_2...g_n = z_1z_2...z_n$, so that in either case the sequencing is done correctly. Assume the former case to be true. If i=n the assignment e_n:=n changes nothing. If i<n then replacing $g_i z_{i+1}...z_n$ by $h(g_i) a_{i+1}...a_n$ destroys the rightmost z-subword $z_{i+1}...z_n$ (since $a_k \neq z_k$ for $i+1 \leq k \leq n$) so that the assignment e_n:=n makes the conclusion true for all k>i (for i<k<n, e_i was equal to i because g_i was not the rightmost member of the z-subword and now it is not part of any z-subword). If $g_i \neq z_i$ none of the other z-subwords are changed, nor is any other e_k, so that the conclusion is still true for all $k \leq i$. Suppose now that g_i=z_i. If i>1 and g_{i-1}=z_{i-1} then we have extended the z-subword $z_{j+1}...z_{i-1}$ to $z_{j+1}...z_i$, where j=e_{i-1}, so that g_{i-1} is no longer its rightmost member but g_i is, and the two assignments 'e_i:=e_{i-1}; e_{i-1}:=i-1' make the conclusion true for k=i-1 and k=i. If i=1 or $g_{i-1} \neq z_{i-1}$ then we have created a new z-subword z_i, so that e_{i-1} was already i-1 (even if i=1: e_0 is always 0) and this value is correctly assigned to e_i, and again the conclusion is true for k=i-1 and k=i. No e_k gets changed for any k<i-1, nor does any other z-subword, so that the conclusion is true for all k at the end of that execution of the algorithm, QED.

If we have a way of calculating $\#(g_1g_2...g_{i-1})$, the number of elements of L with prefix $g_1g_2...g_{i-1}$, we have a conceptually simple ranking algorithm, which is an adaptation to genlex order of the algorithm in [Le] which ranks words in lexicographical order. It is easy to verify the following three formulae:

$$\text{rank}(g_1g_2...g_{i-1}g_ia_{i+1}...a_n) = \text{rank}(g_1g_2...g_{i-1}a_ia_{i+1}...a_n)+\Sigma_{f_i \text{ precedes } g_i}\#(g_1g_2...g_{i-1}f_i) \quad (1.1)$$

$$\text{rank}(g_1g_2...g_{i-1}g_iz_{i+1}...z_n) = \text{rank}(g_1g_2...g_{i-1}z_iz_{i+1}...z_n)-\Sigma_{h_i \text{ follows } g_i}\#(g_1g_2...g_{i-1}h_i) \quad (1.2)$$

$$\text{rank}(g_1g_2...g_{i-1}z_i...z_n) = \text{rank}(g_1g_2...g_{i-1}a_i...a_n)+\#(g_1g_2...g_{i-1}) -1. \quad (1.3)$$

To compute $\text{rank}(g_1g_2...g_n)$ we could move forward through L and assign to a variable r the values $\text{rank}(a_1a_2...a_n)=0$, $\text{rank}(g_1a_2...a_n)$, ... , $\text{rank}(g_1g_2...g_n)$ obtained by successive substitution into (1.1). If the sum in (1.2) is easier to evaluate, we could move backward through L and assign to r the values $\text{rank}(z_1z_2...z_n)=\#(L)-1$, $\text{rank}(g_1z_2...z_n)$, ... , $\text{rank}(g_1g_2...g_n)$ obtained by successive substitution into (1.2). One of these two alternatives is usually used to compute ranks in lexicographical order. However, for graylex or genlex order it may be convenient to evaluate the sum in (1.2) sometimes but not all the time, and before switching from one to the other we need to substitute into (1.3): we add $\#(g_1g_2...g_{i-1})-1$ to r if we are switching from (1.1) to (1.2) and subtract the same quantity from r if we are switching from (1.2) to (1.1). We substitute into (1.1) by adding the sum to r or into (1.2) by subtracting the sum from r. The generic ranking algorithm, then, is to initialize r to 0 and assume we will first substitute into (1.1), and then for i=1,2,...,n we do the actual substitutions, after which r will be $\text{rank}(g_1g_2...g_n)$.

To find the word $g_1g_2...g_n$ of rank r we maintain a variable q which is always assigned the value $r-\text{rank}(g_1g_2...g_{i-1}a_ia_{i+1}...a_n)$ if we are going to substitute into (1.1) and $\text{rank}(g_1g_2...g_{i-1}z_iz_{i+1}...z_n)-r$ if we are going to substitute into (1.2). Before switching from one to the other we substitute into (1.3) by replacing q by $\#(g_1g_2...g_{i-1})-1-q$. We choose the g_i which maximizes the sum on the right side of (1.1) or (1.2) subject to the constraint that it not exceed q, and we do the substitution by subtracting this sum from q. The generic unranking algorithm is to initialize q to r and assume we will first substitute into (1.1), and then for i=1,2,...,n we do the actual substitutions, after which q will be 0 and $g_1g_2...g_n$ will be the word of rank r in L.

2. The Proskurowski-Ruskey Gray code for balanced parenthesis systems

A *balanced parenthesis system* of length 2n is a bitstring of n zeros and n ones such that no prefix contains more zeros than ones. In [PR] a recursive description is given of a Gray code in which each system of length 2n is changed to its successor by transposing a single pair of bits. T(n,k) is defined as the list of systems of length 2n with prefix 1^k0. For bitstrings x and $y=1^k0x$, the operations *flip* and *insert* are defined as $\text{flip}(y)=1^{k-1}01x$ and $\text{insert}(y)=1^{k+1}00x$. Two operations on lists are defined: AoB means A followed by B and A^R means A reversed. Then T(n,k) is defined by the following recursion:

$$T(n,k) = \begin{cases} \text{flip}(T(n,2)) & \text{if } k = 1 \\ \text{flip}(T^R(n,k+1)) \text{ o } \text{insert}(T(n-1,k-1)) & \text{if } 1 < k < n \\ 1^n0^n & \text{if } k = n. \end{cases} \tag{2.1}$$

The first object in T(n,k) is defined as

$$\text{first}(T(n,k)) = \begin{cases} 101100(10)^{n-3} & \text{if } k = 1 \text{ and } n \geq 3 \\ 1^k010^k(10)^{n-k-1} & \text{if } 1 < k < n \text{ or } (k = 1 \text{ and } n = 2) \\ 1^n0^n & \text{if } k = n \end{cases} \tag{2.2}$$

(we added the conditions 'and n≥3' and 'or (k=1 and n=2)' to make the definition in [PR] work for the trivial case when k=1 and n=2).

Two Gray codes for the set of all systems of length 2n are given: T(n+1,1) with the prefix 10 removed, and $T(n,n)oT(n,n-1)o...oT(n,2)oT^R(n,1)$. A recursive algorithm is given which generates T(n,k) in constant average time. The authors challenge the reader to modify their algorithm so that it generates all the systems of length 2n in constant worst-case time and we answer their challenge here. Balanced parenthesis systems is one of many representations of binary trees, and for one of these there is already a loop-free generation algorithm [vB]; however, we believe that we have the only existing loop-free algorithm for generating balanced parenthesis systems.

First we generate T(n,k) from (2.1) for small values of n (see Figure 2.1).

T(1,1)
10

T(2,2)	T(2,1)
1100	1010

T(3,3)	T(3,2)	T(3,1)
111000	110100	101100
	110010	101010

T(4,4)	T(4,3)	T(4,2)	T(4,1)
11110000	11101000	11010010	10110010
	11100100	11010100	10110100
	11100010	11011000	10111000
		11001100	10101100
		11001010	10101010

T(5,5)	T(5,4)	T(5,3)	T(5,2)	T(5,1)
1111100000	1111010000	1110100010	1101001010	1011001010
	1111001000	1110100100	1101001100	1011001100
	1111000100	1110101000	1101011000	1011011000
	1111000010	1110110000	1101010100	1011010100
		1110010010	1101010010	1011010010
		1110010100	1101110000	1011110000
		1110011000	1101101000	1011101000
		1110001100	1101100100	1011100100
		1110001010	1101100010	1011100010
			1100110010	1010110010
			1100110100	1010110100
			1100111000	1010111000
			1100101100	1010101100
			1100101010	1010101010

Figure 2.1: T(n,k), the list of balanced parenthesis systems of length 2n with prefix 1^k0

We observe the following pattern for T(n,k). Given a bitstring in T(n,k), call 1_i the ith 1 from the left of the bitstring. By definition, $1_1,1_2,...,1_k$ are fixed; we call the other 1s free. In every interval of T(n,k) in which 1_{k+1}, 1_{k+2}, ... ,1_{i-1} stay in one place, 1_i moves between its leftmost position adjacent to 1_{i-1} (except that 1_{k+1} starts with one zero between it and 1_k) and its rightmost position at index 2i-1 in the bitstring, moving right if and only if the number of free 1s to its left which are not in their rightmost positions is even. This can easily be proved from the recursive definition of T(n,k): the crucial argument in the induction step is that if k>2 the flip operator creates a free 1 which is not in its rightmost position and is to the left of all the other free 1s, and this is precisely when the reversal operator is applied. Let g_i be the index of 1_i in the bitstring, and let L be the list of words

$g_{k+1}g_{k+2}...g_n$ corresponding to T(n,k). Then L is in genlex order - in fact, graylex order - with s()=(k+2,k+3,...,2k+1) and

$$s(g_{k+1}g_{k+2}\cdots g_{i-1}) = \begin{cases} (g_{i-1}+1,\ldots,2i-1) \text{ if } g_j = 2j-1 \text{ for an even number of } j,\ k+1 \le j \le i-1, \\ (2i-1,\ldots,g_{i-1}+1) \text{ if } g_j = 2j-1 \text{ for an odd number of } j,\ k+1 \le j \le i-1. \end{cases}$$

Using these observations we can specialize the generic sequencing algorithm to T(n,k) without explicitly storing the array $g_{k+1}g_{k+2}...g_n$. We maintain one bit of auxiliary information: the BOOLEAN variable LastRight, which is true if there are an even number of free 1s which are not in their rightmost positions (initially LastRight is false if n>2 and k<n, since $1_{\max(3,k+1)}$ is the only free 1 which is not in its rightmost position). We initialize another BOOLEAN variable Right, which is true if there are an even number of free 1s to the left of the current 1 which are not in their rightmost positions, to LastRight, and then we scan the current bitstring from right to left, skipping over the zeros and keeping track of the index i of 1_i and its position j in the bitstring. Whenever we come across a 1_i such that j<2i-1, we negate Right. If we come to a 1_i, i>k, such that j<2i-1 and Right is true, or else a 1_i which has a 0 on its left and Right is false, then 1_i moves (we give the details in the next two paragraphs); if we get to 1_k first then the current bitstring is the last one in T(n,k) and we set a termination flag to true.

Suppose that we come to a 1_i, i>k, such that j<2i-1 and Right is true, so that we must move 1_i one position to the right. If i=n then that is all we do, except that if it is now in its rightmost position we negate LastRight. The case when i<n, divided into two subcases depending upon whether moving 1_i to its right takes it to its rightmost position, is illustrated at the top of Figure 2.2: the arrows over the 1s indicate their direction of motion. The directions of motion are calculated from the graylex order and the 'before' and 'after' positions of the 1s to the right of 1_i from the terminal and initial positions, respectively, for each 1 given its direction of motion.

```
           .,          . .   .                  .,     . .  .
BEFORE  1i1 00...0001010...10        BEFORE  1i1 0001010...10

             .       , , ,   ,                 ..   , ,   ,
AFTER       1i00...0101010...10      AFTER       1i1001010...10  (1i at rightmost)


             ,       . . .   .                  ,,    . .  .
BEFORE      1i00...0101010...10      BEFORE      1i1001010...10  (1i at rightmost)

          , .        , ,   ,                 , .    , ,   ,
AFTER     1i1 00...0001010...10      AFTER   1i1 0001010...10
```

Figure 2.2. The suffix beginning with 1_i before and after 1_i moves.

Now we suppose that we come to a 1_i, i>k, which has a 0 on its left and Right is false, so that we must move 1_i one position to the left. If i=n then that is all we do, except that if it was in its rightmost position we negate LastRight. The case when i<n, divided into two subcases depending upon whether 1_i was originally in its rightmost position, is illustrated at the bottom of Figure 2.2.

Whenever i<n the number of free 1s not in their rightmost positions changes by 1 and so we negate LastRight.

If we are generating all the balanced parenthesis systems by generating T(n+1,1) without the prefix 10, the algorithm remains unchanged: 1_2 which is free but at its rightmost position in T(n+1,1) is simply renamed 1_1 which is now fixed. A few minor changes are made if we are generating $T(n,n)oT(n,n-1)o...oT(n,2)oT^R(n,1)$. When we get to i_k we are at the last bitstring in T(n,k), and we want to change it to the first bitstring in T(n,k-1) (or $T^R(n,1)$ if k=2). So, instead of setting the termination flag to true, we move 1_k, and only 1_k, to the right. This creates a new free 1 not at its rightmost position (or changes all the directions of motion in passing from T(n,2) to $T^R(n,1)$); so we negate LastRight. Since we are now generating T(n,k-1) or its reversal we decrease k by 1. When the current bitstring is the last one in $T^R(n,1)$, this algorithm would instruct us to move 1_2 to its left; so as a special fix we skip over 1_2. Finally, we set the termination flag to true when we get to 1_1.

To make the sequencing algorithm run in constant time we specialize the generic algorithm of Figure 1.1 to the genlex-order list L of words $g_1g_2...g_n$, where g_i is the index of 1_i, the ith 1, in the bitstring. Now we have to store $g_1g_2...g_n$ explicitly as an auxiliary array, as well as the array $e_1e_2...e_n$ of Figure 1.1 (we don't need e_0 because 1_1 never moves) and an array $d_1d_2...d_n$, where d_i=1 if 1_i is moving right (g_i is increasing) and 0 otherwise. The array $e_1e_2...e_n$ is updated as in Figure 1.1; we know that $g_i=z_i$ if it is equal to either 2i-1 or $g_{i-1}+1$, since g_i has just changed and cannot change to a_i. The updating of the bitstring is done as in the previous version (see Figure 2.2), and the corresponding change made to the position vector $g_1g_2...g_n$ is easy to deduce. If 1_i moves, g_i either increases or decreases by 1, and if k<i<n then g_{i+1} also changes: if g_i decreases by 1 then g_{i+1} assumes the old value of g_i, and otherwise g_{i+1} becomes 2i if g_i=2i-1 and 2i+1 otherwise. The new value of d_{i+1} should be easy to deduce from the arrows in Figure 2.2: it is 1 unless g_{i+1} becomes 2i+1. From these arrows it would appear that $d_{i+2},d_{i+3},...,d_n$ should all change from 1 to 0, so that the algorithm wouldn't run in constant time. But we don't need to know d_j until we have to change g_j, and since all these changes to d_j take place when g_j=2j-1 we simply set d_j to 0 whenever we have to change g_j which is equal to 2j-1; the only side-effect of this fix

is that 1_2 would get moved left after generating T(n,1), and we fix that up by modifying the termination condition (we set the flag to true if $e_n \le \max(2,k)$). The only modifications needed to generate T(n,n)oT(n,n-1)o...oT(n,2)oT^R(n,1) is that instead of stopping when e_n=k we move 1_k (and only 1_k, as in the previous version) and decrease k by 1 and we stop when e_n=1: the passage from T(n,2) to T^R(n,1) falls out automatically.

To express the ranking and unranking problem in terms of formulae (1.1)-(1.3) we refer to the position vector $g_1g_2...g_n$ even though we don't have to store it explicitly. The quantity $\#(g_1g_2...g_i)$ is the number of balanced parenthesis systems of length 2n with a prefix containing i ones and j=g_i symbols altogether, so that it has j-i zeros. The rest of the bitstring has n-i ones and n-j+i zeros, and the number of such suffixes [Fe, p70] is

$$\binom{2n-j}{n+i-j} - \binom{2n-j}{n+i-j+1}. \qquad (2.3)$$

To substitute into (1.1) or (1.2) we have to sum these binomial coefficients over j, and since the sign in front of j is negative it is more convenient to do so over large j than small j; so we will use (1.1) when j=g_i is decreasing and (1.2) when it is increasing. Now the direction in which g_i changes stays constant as long as g_i=2i-1, and in this case the sum in (1.1) (when $g_i=a_i$) and in (1.2) (when $g_i=z_i$) is 0. The only time we have to actually substitute into (1.1) or (1.2) is when g_i<2i-1, and then we substitute into (1.3) as well (with i replaced by i+1) since g_{i+1} is going to change in the opposite direction. Combining these two substitutions, we are going to add or subtract the sum of (2.3) over j from its current value to 2i-1, which is

$$\binom{2n-j+1}{n+i-j} - \binom{2n-j+1}{n+i-j+1}. \qquad (2.4)$$

Simplifying (2.4) and subtracting 1 (as in (1.3)), we obtain the algorithm of Figure 2.3 for ranking the balanced parenthesis system $p_1p_2...p_{2n}$ in T(n,k).

```
Rank:= $\binom{2n-k-1}{n-1}\frac{k}{n} - 1$;
Dir:=TRUE; (* the current 1 is moving right *)
j:=k+2;
FOR i:=k+1 TO n DO
  WHILE $p_j$=0 DO j:=j+1 END; (* find $1_i$ *)
  IF j<2i-1 THEN
    Term:= $\binom{2n-j+1}{n+i-j}\left\{\frac{2i-j}{n-j+i+1}\right\} - 1$;
    IF Dir THEN Rank:=Rank-Term ELSE Rank:=Rank+Term END IF;
    Dir:=NOT Dir;
  END IF;
  j:=j+1;
END FOR.
```

Figure 2.3. Ranking the balanced parenthesis system $p_1p_2...p_{2n}$ in T(n,k).

To rank $p_1p_2...p_{2n}$ in T(n,n)oT(n,n-1)o...oT(n,2)oT^R(n,1), we must first find which T(n,k) it is in. We search for the first 0, and the number of 1s we skip over is k. If k>1 we initialize Rank to #(T(n,n)oT(n,n-1)o...oT(n,k)) - 1, which is

$$\binom{2n-k}{n}\left\{\frac{k+1}{n+1}\right\} - 1$$

and proceed from the second line of the algorithm. If k=1, we set it to 2, add 1 to the above initial value of Rank, set Dir to FALSE and proceed from the third line of the algorithm. To rank $p_1p_2...p_{2n}$ in T(n+1,1) with the prefix 10 removed, we set n to n+1 and k to 1 and then start the algorithm from the first line except that in the WHILE loop we test p_{j-2} instead of p_j.

Of course, if we are actually programming this algorithm we neither calculate the binomial coefficients individually (which wastes time) nor precompute them and store them in a table (which wastes space). We calculate the one necessary to initialize Rank (in Figure 2.3) - call it f - and then we calculate the others from their predecessors: before increasing j in the WHILE loop we multiply f by n+i-j and divide it by 2n-j+1 and before increasing j at the end of the FOR loop we multiply f by n-i+1 and divide it by 2n-j+1 (since i is going to increase too). This makes the whole ranking algorithm run in O(n) arithmetic operations and O(1) extra space.

The same optimization can be applied to the unranking algorithm of Figure 2.4.

```
FOR i:=1 TO k DO $p_j$:=1 END FOR;
$p_{k+1}$:=0;
Rank:= $\binom{2n-k-1}{n-1}\frac{k}{n}-1-\text{Rank}$;
j:=k+2;
FOR i:=k+1 TO n DO
  WHILE $\binom{2n-(j+1)+1}{n+i-(j+1)}\left\{\frac{2i-(j+1)}{n-(j+1)+i+1}\right\} > \text{Rank}$ DO
    $p_j$:=0;
    j:=j+1;
  END WHILE;
  $p_j$:=1;
  IF j<2i-1 THEN
    Rank:= $\binom{2n-j+1}{n+i-j}\left\{\frac{2i-j}{n-j+i+1}\right\}-1-\text{Rank}$;
  END IF;
  j:=j+1;
END FOR;
FOR i:=j TO 2n DO $p_i$:=0 END FOR.
```

Figure 2.4: Constructing the parenthesis system $p_1p_2...p_{2n}$ of a given Rank in T(n,k).

If we are working with T(n,n)oT(n,n-1)o...oT(n,2)oT^R(n,1) we set k to the smallest integer such that Rank is not less than

$$\binom{2n-k-1}{n}\left\{\frac{k+2}{n+1}\right\}$$

and subtract this value from Rank, and then we begin the algorithm from the first line except that we skip the third line if k=1. If we are working with T(n+1,1) with the prefix 10 removed, we set k=1 and n=n+1 and begin the algorithm from the third line except that we subtract 2 from every subscript of p.

We have programmed optimized versions of the ranking and unranking algorithms and both sequencing algorithms (the loop-free one and the one which uses no auxiliary arrays) in Modula-2 and tested them by generating T(n,k), T(n,n)oT(n,n-1)o...oT(n,2)oT^R(n,1) and T(n+1,1) with the prefix 10 removed, ranking each parenthesis system generated and constructing the parenthesis system of that rank for comparison with the original. Listings appear in the appendix.

The above optimization trick is well known but not well published: for example, it is not included in the lexicographical-order ranking of balanced parenthesis systems in [Pa], [PL], [RH] or [Z]; so we include it below.

From [PL] (or by substituting (2.4) into (1.1)) we have

$$\mathrm{Rank}(g_1 g_2 \ldots g_n) = \sum_{i=2}^{n} \binom{2n-g_i}{n+i-g_i-1} \left\{ \frac{2i-g_i-1}{n+i-g_i} \right\}. \tag{2.5}$$

Applying the same optimization trick we have the ranking and unranking algorithms of Figure 2.5 (which can be further optimized to avoid recomputing products and quotients).

```
Rank:=0;  f:=1;  j:=2n-1;
FOR i:=n DOWNTO 2 DO
  WHILE  $p_j$=0 DO DO
    f:=f*(2n-j+1)/(n+i-j);
    j:=j-1
  END WHILE;
  Rank:=Rank+f*(2i-j-1)/(n+i-j);
  f:=f*(2n-j+1)/(n-i+2);
  j:=j-1;
END FOR.
```

```
|f:=Binomial_Coeff(2n-2,n-1);
|$p_1$:=1;  j:=2;
|FOR i:=2 TO n DO
|  WHILE f*(2i-j-1)/(n+i-j)>Rank DO
|    $p_j$:=0;  j:=j+1;
|    f:=f*(n+i-j)/(2n-j+1);
|   END WHILE;
|  Rank:=Rank-f*(2i-j-1)/(n+i-1);
|  $p_j$:=1; j:=j+1;
|  f:=f*(n-i+1)/(2n-j+1)
|END FOR;
|FOR i:=j TO 2n DO $p_i$:=0 END FOR.
```

Figure 2.5.
Ranking and unranking the balanced parenthesis system $p_1 p_2 \ldots p_{2n}$ in lexicographical order.

3. Other Gray codes

a) The classical binary reflected Gray code

In this Gray code [Gr], L(n) consists of all the 2^n strings $g_1 g_2 \ldots g_n$ of n bits and each one differs from its successor in a single bit. It was shown in [Ch] that $s(g_1 g_2 \ldots g_{i-1})=(0,1)$ if $g_1 g_2 \ldots g_{i-1}$ contains an even number of 1s and (1,0) otherwise (see Figure 3.1). For the sake of completeness we give a sketch of a proof which is easy to explain to students. The recursive description of this Gray code is given by

$$L(0) \text{ is the empty word and } L(n)='0'L(n-1)\text{o}'1'L^R(n-1) \text{ for } n>0. \tag{3.1}$$

The inductive step follows from the fact that the reversal operator is applied when the first symbol changes from 0 to 1, reversing the parity of the number of 1s in any non-null prefix.

```
i  g1 g2 g3 g4      e0 e1 e2 e3 e4  (b0)b1 b2 b3 b4

   0  0  0  0       0  1  2  3  4   0  0  0  0  0
4  0  0  0  1       0  1  2  3  3   0  0  0  0  1
3  0  0  1  1       0  1  2  2  4   0  0  0  1  0
4  0  0  1  0       0  1  2  3  2   0  0  0  1  1
2  0  1  1  0       0  1  1  3  4   0  0  1  0  0
4  0  1  1  1       0  1  1  3  3   0  0  1  0  1
3  0  1  0  1       0  1  2  1  4   0  0  1  1  0
4  0  1  0  0       0  1  2  3  1   0  0  1  1  1
1  1  1  0  0       0  0  2  3  4   0  1  0  0  0
4  1  1  0  1       0  0  2  3  3   0  1  0  0  1
3  1  1  1  1       0  0  2  2  4   0  1  0  1  0
4  1  1  1  0       0  0  2  3  2   0  1  0  1  1
2  1  0  1  0       0  1  0  3  4   0  1  1  0  0
4  1  0  1  1       0  1  0  3  3   0  1  1  0  1
3  1  0  0  1       0  1  2  0  4   0  1  1  1  0
4  1  0  0  0       0  1  2  3  0   0  1  1  1  1
0  Done=TRUE
```

Figure 3.1

The classical Gray code for bitstrings: the Gray code word $g_1g_2g_3g_4$, the index i such that g_i gets changed, the binary ranking function $b_1b_2b_3b_4$ with sentinel $b_0=0$, and the auxiliary array $e_0e_1e_2e_3e_4$ used in Ehrlich's loop-free algorithm.

We demonstrate the pedagogical value of our method by showing how easily the classical sequencing algorithm [NW, p16] and ranking algorithm [Wi] can be derived from the graylex order of L(n). It follows from this order that $g_i=z_i$ if and only if $g_1g_2...g_i$ contains an odd number of 1s. If $g_1g_2...g_n$ has an even number of 1s, $g_n\neq z_n$; so by the generic sequencing algorithm we change g_n. If $g_1g_2...g_n$ has an odd number of 1s, then the rightmost $g_i\neq z_i$ comes immediately to the left of the rightmost 1 (unless the string is $10...0=z_1z_2...z_n$) and we change g_i which converts the suffix 10...0 from $z_{i+1}...z_n$ to $a_{i+1}...a_n$ as required. This algorithm becomes more efficient if we maintain an auxiliary BOOLEAN variable which is true if $g_1g_2...g_n$ has an even number of 1s. The loop-free version in [BER] and [RND, p179] is a special case of Figure 1.1, where the test 'if $g_i=z_i$' never has to be made because once g_i is changed it must be z_i. To rank a bitstring we observe that $\#(g_1g_2...g_i)=2^{n-i}$; so that by (1.1) 2^{n-i} will get added to the rank for each i such that $g_1g_2...g_i$ has an odd number of 1s (so that $g_i=z_i$ and the sum in (1.1) is $\#(g_1g_2...g_{i-1}a_i)=2^{n-i}$). This means that if $b_1b_2...b_n$ is the binary expansion of the rank of $g_1g_2...g_n$, then $b_i=g_1+g_2+...+g_i$ mod 2, which gives a ranking algorithm which requires O(n) arithmetic operations. Solving for g_i we obtain $g_i=b_i+b_{i-1}$ mod 2, where b_0 is taken to be 0, which gives an unranking algorithm which requires O(n) arithmetic operations. Note that $g_i=z_i$ if and only if $b_i=1$, and that (see Figure 3.1) $e_i=j$ if $b_{j+1}...b_i$ is a maximal string of 1s, and otherwise $e_i=i$.

b) The Nijenhuis-Wilf Gray code for permutations.

Given a permutation $p_1p_2...p_n$ of {1,2,...,n}, let $g_1g_2...g_{n-1}$ be its inversion vector: g_i is the number of elements of $p_{i+1}...p_n$ which are smaller than p_i. If we impose upon the list of inversion vectors the order given in [Wm, p112], where $s(g_1g_2...g_{i-1})=(0,1,...,n-i)$ if $g_1+g_2+...+g_{i-1}$ is even and (n-i,...,1,0) otherwise, then the permutations are ordered (apart from left-right reversal) as in the Gray code of [NW, p58], where each permutation differs from its successor by a single transposition but not always of adjacent elements (see Figure 3.2, where the direction of motion of g_i is stored in its sign).

p_1	p_2	p_3	p_4	g_1	g_2	g_3	e_0	e_1	e_2	e_3	j	b_1	b_2	b_3	lex order
1	2	3	4	0	0	0	0	1	2	3	4	0	0	0	1 2 3 4
1	2	4	3	0	0	-1	0	1	2	2	4	0	0	1	1 2 4 3
1	3	4	2	0	1	-1	0	1	2	3	4	0	1	0	1 3 2 4
1	3	2	4	0	1	0	0	1	2	2	4	0	1	1	1 3 4 2
1	4	2	3	0	-2	0	0	1	1	3	4	0	2	0	1 4 2 3
1	4	3	2	0	-2	-1	0	1	2	1	4	0	2	1	1 4 3 2
2	4	3	1	1	-2	-1	0	1	2	3	4	1	0	0	2 1 3 4
2	4	1	3	1	-2	0	0	1	2	2	4	1	0	1	2 1 4 3
2	3	1	4	1	-1	0	0	1	2	3	4	1	1	0	2 3 1 4
2	3	4	1	1	-1	-1	0	1	2	2	4	1	1	1	2 3 4 1
2	1	4	3	1	0	-1	0	1	1	3	4	1	2	0	2 4 1 3
2	1	3	4	1	0	0	0	1	2	1	3	1	2	1	2 4 3 1
3	1	2	4	2	0	0	0	1	2	3	4	2	0	0	3 1 2 4
3	1	4	2	2	0	-1	0	1	2	2	4	2	0	1	3 1 4 2

```
 3  2  4  1    2  1 -1    0  1  2  3   4   2  1  0   3  2  1
4
 3  2  1  4    2  1  0    0  1  2  2   4   2  1  1   3  2  4
1
 3  4  1  2    2 -2  0    0  1  1  3   4   2  2  0   3  4  1
2
 3  4  2  1    2 -2 -1    0  1  2  1   2   2  2  1   3  4  2
1
 4  3  2  1   -3 -2 -1    0  0  2  3   4   3  0  0   4  1  2
3
 4  3  1  2   -3 -2  0    0  0  2  2   4   3  0  1   4  1  3
2
 4  2  1  3   -3 -1  0    0  0  2  3   4   3  1  0   4  2  1
3
 4  2  3  1   -3 -1 -1    0  0  2  2   4   3  1  1   4  2  3
1
 4  1  3  2   -3  0 -1    0  1  0  3   4   3  2  0   4  3  1
2
 4  1  2  3   -3  0  0    0  1  2  0       3  2  1   4  3  2
1
```

Figure 3.2

The permutations $p_1p_2p_3p_4$ generated in the Nijenhuis-Wilf Gray code order. Abs(g_i) is the number of elements p_j such that $p_j<p_i$ but $j>i$; the sign of g_i is the direction in which abs(g_i) is moving. The array $e_0e_1e_2e_3$ is used to find the value of $i=e_3$ such that p_i and p_j, $j>i$, get swapped. The rank of the permutation is $3!b_1+2!b_2+1!b_3$, and the permutation whose inversion vector is $b_1b_2b_3$ is given to its right.

```
IF EvenPerm THEN              (* g[1]+...+g[n-1] is even; so g[n-1] can change *)
  Swap(p[n-1],p[n]); EvenPerm:=FALSE; RETURN
ELSE
  i:=n-1; Rise:=FALSE;                          (* Rise means that g[i] is rising.
*)
  Max:=p[n]; Min:=p[n];                          (* Max/Min of a[i+1]...a[n] *)
  LOOP
    IF p[i]<Min THEN                             (* g[i]=0. Rise stays fixed. *)
      Min:=p[i];
      IF Rise THEN                               (* g[i] should and can rise. *)
        EXIT
      ELSE                                       (* g[i] should fall but can't. *)
        i:=i-1; IF i=0 THEN Done:=TRUE; RETURN END IF
      END IF
    ELSE IF p[i]>Max THEN                                      (* g[i]=n-i. *)
      Max:=p[i];
      IF n-i is odd then Rise:=NOT(Rise) END IF;
      IF NOT(Rise) THEN                          (* g[i] should and can fall. *)
        EXIT
      ELSE                                       (* g[i] should rise but can't. *)
        i:=i-1; IF i=0 THEN Done:=TRUE; RETURN END IF
      END IF
    END IF;                    (* g[i] can rise or fall, but we must update Rise
*)
    FOR k FROM i+1 TO n DO
      IF p[k]<p[i] THEN Rise:=NOT(Rise) END IF
    END FOR;
    EXIT;          (* so we only execute the FOR loop once within the other loop *)
  END LOOP;                    (* Now we know i and Rise, and must calculate j. *)
  IF Rise THEN                                   (* We search min p[j] >p[i]. *)
    Min:=n+1;
    FOR k FROM i+1 TO n DO
      IF (p[k]>p[i]) AND (p[k]<Min) THEN j:=k; Min:=p[k] END
IF
    END FOR
  ELSE                                           (* We search max p[j]<p[i]. *)
    Max:=0;
    FOR k FROM i+1 TO n DO
      IF (p[k]<p[i]) AND (p[k]>Max) THEN j:=k; Max:=p[k] END
IF
    END FOR
  END IF;
  Swap(p[i],p[j]); EvenPerm:=TRUE; RETURN
END IF.
```

Figure 3.3

A linear-time constant-extra-space algorithm for finding the next permutation in the Nijenhuis-Wilf Gray code (initially p[i]=i for all i, EvenPerm is TRUE and Done is FALSE).

The sequencing algorithm given in [NW, p58] consists essentially of calculating the inversion vector from the permutation directly from the definition from right to left until one finds the rightmost $g_i \neq z_i$ (which takes quadratic time in the worst case), then finding the p_j which is the nearest integer to p_i, in the direction in which g_i is moving, among $p_{i+1}...p_n$, and swapping p_i with p_j. The algorithm can be made to run in linear time if we store the inversion vector as an auxiliary array, but we can find the rightmost $g_i \neq z_i$ in linear time without an auxiliary array (see Figure 3.3) by observing that if $p_j < p_i$ for some $j > i$ then $g_i > 0$ and can fall, and if $p_j < p_i$ for some $j > i$ then $g_i < n-i$ and can rise.

We can also use the loop-free algorithm of [Wm, p112] for generating the Cartesian product of integer intervals (a generalization of the Bitner-Ehrlich-Reingold method and a special case of Figure 1.1) to design a loop-free algorithm for generating permutations in this order. We store the inversion vector and the direction vector (or we store the direction of motion of g_i in its sign - see Figure 3.2) as well as the array $e_0e_1...e_{n-1}$. We update $e_0e_1...e_{n-1}$ as in Figure 1.1 (with n replaced by n-1); we know that $g_i = z_i$ if it is equal to 0 or n-i. If we are storing the direction of motion as a sign, then updating g_i consists of adding 1 and then changing its sign if it is equal to n-i. To update the permutation itself we must swap p_i with p_j for the appropriate j, and it remains to find this value of j in constant time.

Suppose g_i has to rise, so that $g_1+...+g_{i-1}$ is even. Then we want to find j such that p_j is the smallest of the integers $p_{i+1},...,p_n$ which are larger than p_i.

By the definition of inversion vector, g_i of the integers $p_{i+1},...,p_n$ are smaller than p_i, so that the desired value of j will be known once we have sorted $p_{i+1}...p_n$. Since g_i is the rightmost integer which is not at its last value, the suffix $g_{i+1}...g_n$ is at its last value and so, therefore, is the suffix $p_{i+1}...p_n$. If g_{i+1} is supposed to fall, $g_{i+1}...g_n$ is 0...0, so that the integers $p_{i+1},...,p_n$ are in increasing order. It remains to sort the suffix $p_{i+1}...p_n$ in the case when g_{i+1} is supposed to rise.

To this end we first find the last string $g_1...g_{n-1}$ and the corresponding permutation $p_1...p_n$. Since g_1 rises from 0 to n-1, it is n-1. If n is even, n-1 is odd; so g_2 falls and is 0. Then g_1+g_2 is also odd, so that g_3 is also zero, and so on. Thus the last inversion vector is n-1 0...0 and the last permutation is n1234..n-1. If n is odd, n-1 is even; so g_2 rises and is n-2. Now g_1+g_2 is odd; so g_3 falls and is 0, and all subsequent g_i are 0. Thus the last inversion vector is n-1 n-2 0...0 and the last permutation is n n-1 123...n-2. It follows that when g_{i+1} is supposed to rise, $p_{i+1}...p_n$ is sorted with the largest integer first, then the second largest if n-i is odd, and then the rest in increasing order.

We recall that we have supposed that $g_1+...+g_{i-1}$ is even. If g_i is odd, g_{i+1} is supposed to fall, and so the integers $p_{i+1}...p_n$ are all in increasing order, and g_i of them are smaller than p_i; so the next one, in position $j=i+g_i+1$, is the smallest one which is larger than p_i.

Suppose that g_i is even, so that g_{i+1} is supposed to rise. There are 2 cases to consider, depending on whether n-i is odd or even.

If n-i is even, $p_{i+2}<...<p_n<p_{i+1}$. Now $p_i<p_n<p_{i+1}$: if p_i were greater than p_{i+1}, g_i would be n-i and would be unable to rise, and if p_i were between p_n and p_{i+1}, g_i would be n-i-1, which is odd. This means that the g_i integers among $p_{i+1},...,p_n$ which are smaller than p_i all follow p_{i+1}, and the smallest one which is larger than p_i is in position $j=i+g_i+2$.

If n-i is odd, $p_{i+3}<...<p_n<p_{i+2}<p_{i+1}$. If $i+g_i=n-1$, only p_{i+1} is greater than p_i, so that $j=i+1$. Now $i+g_i$ cannot be n-2 because n-i is odd and g_i is even; so if $i+g_i<n-1$, $p_i<p_n$. Then all the g_i integers among $p_{i+1},...,p_n$, all follow both p_{i+1} and p_{i+2}, so that the smallest one which is larger that p_i is in position $j=i+g_i+3$.

The loopless algorithm of Figure 3.4 finds j so that (p_i,p_j) should be swapped.

Recall that we have supposed that g_i is to rise. If g_i is about to fall, then to pass from the next string to the current one, g_i rises. So to compute j, we use the same algorithm except that instead of the old value of g_i we use the new one, which is g_i-1.

```
IF g[i] is odd THEN
  j:=i+g[i]+1
ELSE IF n-i is even THEN
  j:=i+g[i]+2
ELSE
  IF i+g[i]=n-1 THEN j:=i+1 ELSE j:=i+g[i]+3 END IF
END IF.
```

Figure 3.4
A loopless algorithm which finds j so that (p_i,p_j) should be swapped if g_i is to rise; otherwise g_i is decreased by 1 before the algorithm is executed.

If we are coding the direction of motion of g_i in its sign, we replace g[i] by abs(g[i]) throughout.

We note that there already exist loop-free versions ([Eh1],[Ds]) of the Trotter-Johnson Gray code for permutations ([Tr],[Jo]), in which each permutation differs from its successor by a transposition of adjacent elements, and the graylex analysis of that Gray code has already been done [Ch]. The reason for including this loop-free permutation generator is to show that our method is general enough to work on one's favourite order.

To rank and unrank permutations according to this Gray code, we observe (see figure 3.2) that the inversion vector $g_1g_2...g_{n-1}$ is related to $b_1b_2...b_{n-1}$, the inversion vector of the same rank generated in lexicographical order, by the following well-known formula, which can easily be proved by the same line of argument used to rank the classical Gray code.

$$g_i = \begin{cases} b_i \text{ if } g_1+...+g_{i-1} \text{ is even ,} \\ n-i-b_i \qquad \text{otherwise .} \end{cases} \tag{3.2}$$

One can pass between these two inversion vectors in linear time using (3.2) and between $b_1b_2...b_{n-1}$ and its rank $(n-1)!b_1+(n-2)!b_2+...+1!b_{n-1}$ in linear time; so that the complexity of ranking and unranking permutations in this Gray code order, as in lexicographical order, is dominated by the complexity of passing between a permutation and its inversion vector. There are O(n log n) algorithms in [Kn, p. 578-579 (answers to exercises 5 and 6, p. 19)], and I have been told that asymptotically faster ones exist, but I have never been able to find them.

c) The Nijenhuis-Wilf (Tang-Liu) Gray code for combinations

–	+	–	+	
g_1	g_2	g_3	g_4	i
1	2	3	4	4
1	2	4	5	2
2	3	4	5	1
1	3	4	5	3
1	2	3	5	4
1	2	5	6	2
2	3	5	6	1
1	3	5	6	2
3	4	5	6	1
2	4	5	6	1
1	4	5	6	3
1	2	4	6	2
2	3	4	6	1
1	3	4	6	3
1	2	3	6	5

Figure 3.5
The 4-combinations $g_1g_2g_3g_4$ of {1,2,3,4,5,6}. The signs indicate the direction of motion of the integers and i is the index of the leftmost integer $g_i \neq z_i$.

A k-combination, or k-subset, of {1,2,...,n} is coded by listing its members in increasing order. The Gray code given in [NW, p28] lists all these combinations for fixed n and k in such an order that each combination differs from its predecessor by the inclusion of one element and the exclusion of one other element. Apart from left-right reversal, the lists are in graylex order [Ch]: for each suffix $g_{i+1}...g_n$, $s(g_{i+1}...g_n)$ runs through consecutive integers from a minimum of i to a maximum which is g_{i+1}-1 if i<k and n if i=k, rising if k-i is even and falling otherwise (see Figure 3.5).

For the sake of completeness we present a sketch of a proof of the Graylex order from the recursive definition of this Gray code [NW] given in formula (3.3)

$$G(n,k) = G(n-1,k)oG^R(n-1,k-1)'n' \text{ for } n>0 \text{ and } k>0, \tag{3.3}$$

anchored by setting G(n,k) to the empty list if n=0 and k>0 or n>0 and k=0 and G(0,0) to the singleton consisting of the empty word. The inductive step follows from the fact that after the concatenation operator both the direction of motion of each g_i changes and the parity of k-i changes, g_{k-1} has n-1=g_k-1 as its maximum value, and g_k increases from n-1 to n.

The generic sequencing algorithm leads to the iterative algorithm given in [NW, p. 32], and the derivation of this algorithm from the graylex order is considerably simpler than the derivation directly from the recursive definition using the revolving door method [NW, p. 29]. The algorithm itself takes linear worst-case time to pass from one combination to the next because the search for the leftmost $g_i \neq z_i$ always begins at g_1. But it can be made to run in constant worst-case time without using an auxiliary array, because the index i of the leftmost $g_i \neq z_i$ varies by at most 2 from one combination to the next, as the following theorem will show.

Theorem. Given a k-combination $g_1g_2...g_k$, let i be the index of the leftmost g_i which is not at its final value z_i. Then for the next combination the corresponding value of i will lie between i-2 and i+1 if k-i is even or between i-1 and i+2 if k-i is odd.

Proof. We first prove the upper bound. We assume that i<k-1; otherwise the result is trivial. If k-i is even, g_i increases; so g_{i+1} must decrease. If g_{i+1} were at its final value of i+1, g_i would be bounded by i, its minimum value, and would have no room to increase, but since g_i is not at its final value, neither is g_{i+1}. But g_{i+1} does not change in passing to the next combination; so it can still decrease, and the new value of i is bounded above by i+1. The same argument shows that if k-i is odd the new value of i is bounded above by i+2,

since now g_{i+2} decreases and could not be at its final value of i+2 without bounding g_i by its minimum value of i.

We now prove the lower bound, assuming that i>2. If k-i is odd, g_i decreases; so g_{i-1} was supposed to increase and is instead set to its first value of i-1 for the next combination. This means that g_j=j for all j<i-1, and since g_{i-2} is supposed to decrease, these integers are all at their final values, so that the next value of i is bounded below by i-1. If k-i is even, g_i increases; so g_{i-1} was supposed to decrease but must have been at its final value of i-1, so that again g_j=j for all j<i-1. In passing to the new combination, g_{i-1} is raised to its first value of g_i-1, so that g_{i-2}, which is supposed to increase, is not necessarily at its final value. However, all the integers to its left are at their final values; so the next value of i is bounded below by i-2. This completes the proof.

We use this theorem to derive from the graylex order an algorithm which generates the combinations in constant worst-case time with no auxiliary array. Aside from the combination itself, there are only 4 variables: i (the index of the current integer), m (the maximum value of g_i), Rise (a BOOLEAN variable which is true if k-i is even so that g_i should be increasing), and Done (which is true if we have reached the last combination). A pseudo-code for an algorithm which updates all the variables in constant time is given in Figure 3.6.

```
LOOP                                          (* iterated AT MOST FOUR TIMES *)
  IF Rise THEN                                    (* g[i] should increase *)
    IF i=k THEN m:=n ELSE m:=g[i+1]-1 END IF;
    IF g[i]<m THEN                                      (* g[i] can increase
*)
      g[i]:=g[i]+1;
      IF i>1 THEN
        g[i-1]:=g[i]-1;                                 (* its first value *)
        IF i=2 THEN i:=1; Rise:=FALSE ELSE i:=i-2 END IF;
      END IF;
      RETURN;
    END IF                   (* otherwise g[i] cannot increase so we increase i
*)
  ELSE                        (* Rise is FALSE and g[i] should decrease *)
    IF i>k THEN Done:=TRUE; RETURN END IF;
    IF g[i]>i THEN                                  (* g[i] can decrease *)
      g[i]:=g[i]-1;
      IF i>1 THEN
        g[i-1]:=i-1;                                    (* its first value *)
        i:=i-1; Rise:=TRUE
      END IF;
      RETURN;
    END IF                 (* otherwise g[i] cannot decrease so we increase i *)
  END IF;
  i:=i+1; Rise:=NOT(Rise)
END LOOP.
```

Figure 3.6

An algorithm for generating the next k-combination of {1,2,...,n} in constant worst-case time and constant extra space. For the first combination, $g_j=j$ for each j from 1 to k, i=k (since only g_k can change), Rise is TRUE and Done is FALSE.

The essential difference between this algorithm and the one in [NW, p. 32] is that in the latter, i is always initialized to 1 and Rise to TRUE if n is odd and FALSE otherwise, whereas in this one, i is updated to its lower bound according to the above theorem and Rise is adjusted accordingly. The comments explain how the algorithm follows from the graylex order.

We note that there are other loop-free combination generators. The one in [BER] and [RND, p186] uses an auxiliary array and generates the Liu-Tang Gray code [LT] which is the same as the one in [NW] except for left-right and first-last reversal and the representation of combinations by bitstrings. The one in [Eh1] uses an auxiliary array, and the one in [Ch] does not; they both create the graylex order on the fly, and it is a challenging exercise to determine just what this order turns out to be (for [Eh1], try generating the position-vectors of the zeros in the bitstring). We present the above algorithm to show that our method finds efficient algorithms without sacrificing aesthetically pleasing orders.

To find Rank($g_1g_2...g_k$) we use (1.1)-(1.3), modified by replacing n with k and applying left-right reversal (since the graylex order is expressed in terms of suffixes rather than prefixes). For i=k down to 1 we substitute into (1.1) and then (1.3) when k-i is even and into (1.2) and then (1.3) when k-i is odd. The sum in (1.1) or (1.2) combines with the appropriate term in (1.3) (with i replaced by i+1) to give the number of combinations of g_i objects taken i at a time; so we have

$$\text{Rank}(g_1g_2...g_k) = \left(\sum_{i=1}^{k}(-1)^{k-i}\binom{g_i}{i}\right) - k \bmod 2. \qquad (3.4)$$

To substitute efficiently into (3.4), we note that its binomial coefficient takes i multiplications and divisions to compute individually and g_i-g_{i-1} of them to compute from its predecessor, and we choose the shortest path for each i, so that the total number of multiplications and divisions is bounded by k(k-1)/2 (if we never use the previous value) and also by $g_k \le n$ (if we always do). For unranking, for i from k down to 1 we let g_i be the smallest integer x such that b=Binomial_Coeff(x,i)-1≥Rank and replace Rank by b-Rank; by computing each binomial coefficient from its predecessor we can do this in O(n) arithmetic operations.

d) The Knuth-Klingsberg Gray code for integer compositions and a simpler one

A k-composition of n is a string of k non-negative integers whose sum is n. In a personal communication to H. Wilf [Wi], D. Knuth presented a Gray code for these compositions such that each composition differs from its predecessor in that one integer is increased by 1 and one integer is decreased by 1. The recursive description given by Knuth for his Gray code is

$$G(n,k+1) = G(n,k)'0'\mathrm{o}G^{R}(n-1,k)'1'\mathrm{o}G(n-2,k)'2'\mathrm{o}G^{R}(n-3,k)'3'\mathrm{o}\ldots\mathrm{o}G^{R?}(0,k)'n' \quad (3.5)$$

where $G^{R?}$ means G if n is even and G^{R} if n is odd. This recursion is anchored by G(n,1) = 'n'. An iterative version is contained in [Kl].

This Gray code is in the following graylex order apart from left-right reversal (see figure 3.7 for the case when n=5 and k=4). For each integer g_i, let $S_i=g_{i+1}+...+g_k$. Then $g_1=n-S_1$, and for each index i>1, the extreme values of g_i are 0 and $n-S_i$, and g_i rises if S_i is even and falls if S_i is odd.

g_1	g_2	g_3	g_4
5	0	0	0
4	1	0	0
3	2	0	0
2	3	0	0
1	4	0	0
0	5	0	0
0	4	1	0
1	3	1	0
2	2	1	0
3	1	1	0
4	0	1	0
3	0	2	0
2	1	2	0
1	2	2	0
0	3	2	0
0	2	3	0
1	1	3	0
2	0	3	0
1	0	4	0
0	1	4	0
0	0	5	0

g_1	g_2	g_3	g_4
0	0	4	1
1	0	3	1
0	1	3	1
0	2	2	1
1	1	2	1
2	0	2	1
3	0	1	1
2	1	1	1
1	2	1	1
0	3	1	1
0	4	0	1
1	3	0	1
2	2	0	1
3	1	0	1
4	0	0	1

g_1	g_2	g_3	g_4
3	0	0	2
2	1	0	2
1	2	0	2
0	3	0	2
0	2	1	2
1	1	1	2
2	0	1	2
1	0	2	2
0	1	2	2
0	0	3	2
0	0	2	3
1	0	1	3
0	1	1	3
0	2	0	3
1	1	0	3
2	0	0	3
1	0	0	4
0	1	0	4
0	0	1	4
0	0	0	5

Figure 3.7
The 4-compositions $g_1g_2g_3g_4$ of 5 in Gray code order.

The inductive step necessary to prove this proposition from (3.5) by induction on k is the fact g_{k+1} always rises, and for each i from 2 to k, the parity of S_i is the same for G(n,k)'0' as for G(n,k) and is reversed, together with the direction of motion, across each o, while n-S_i is the same for G(n,k)'0' as for G(n,k) and remains constant across each o (o is the concatenation symbol in formula 3.5).

From the graylex order we can see that the first composition must be n00...00, since S_i is always 0 and each part except the first one is increasing and is at its first value. The generic sequencing algorithm, specialized to this graylex order, leads to the non-recursive sequencing algorithm shown in Figure 3.8.

```
S:=n-g[1]; i:=2;
LOOP
  S:=S-g[i];                                    (* S is now g[i+1]+...+g[k]
*)
  IF S is even THEN                             (* g[i] should increase
*)
    IF g[i]<n-S THEN                   (* g[i] can increase and will shortly
*)
      IF g[i] is even THEN             (* S[i-1] is even; so g[i-1]=n-S[i-1]
*)
        g[i-1]:=g[i-1]-1         (* its new first value since S[i-1] will be odd
*)
      ELSE               (* S[i-1] is odd; so g[i-1]=0 and so is each g[j] for 2≤j≤i-1
*)
        g[1]:=g[1]-1                    (* g[j] is at its first value for 2≤j≤i-1
*)
      END IF;
      g[i]:=g[i]+1; RETURN
    END IF         (* otherwise we cannot increase g[i] and will instead increase i
*)
  ELSE                                   (* S is odd so g[i] should decrease *)
    IF g[i]>0 THEN                      (* g[i] can decrease and will shortly
*)
      IF g[i] is odd THEN              (* S[i-1] is even; so g[i-1]=n-S[i-1] *)
        g[i-1]:=g[i-1]+1         (* its new first value since S[i-1] will be odd
*)
      ELSE               (* S[i-1] is odd; so g[i-1]=0 and so is each g[j] for 2≤j≤i-1
*)
```

```
        g[1]:=g[1]+1                    (* g[j] is at its first value for 2≤j≤i-1
*)
      END IF;
      g[i]:=g[i]-1; RETURN
    END IF          (* otherwise we cannot decrease g[i] and will instead increase i
*)
  END IF;
  i:=i+1;
  IF i>k THEN                           (* each g[j] is at its final value *)
    Done:=TRUE
  END IF
END LOOP.
```

Figure 3.8

Finding the next k-composition of n in the Knuth-Klingsberg Gray code
(the first one is n0...0).

```
IF g[k]=n THEN Done:=TRUE; RETURN END IF;
IF g[1]>0 THEN
  S:=n-g[1]-g[2];
  IF S is even THEN                                   (* g[2] should and can increase *)
    g[2]:=g[2]+1; g[1]:=g[1]-1;
    IF g[2]=1 THEN t:=t+1; p[t]:=2 END IF                        (* push 2 onto stack *)
  ELSE
    i:=p[t]; S:=n-g[1]-g[i];                         (* g[i] is second positive element *)
    IF S is even THEN                              (* g[i] increases and is odd and j>2*)
      g[i]:=g[i]+1; g[1]:=g[1]-1
    ELSE                          (* g[i] decreases and either g[i] is even or else j=2 *)
      g[i]:=g[i]-1; g[1]:=g[1]+1;
      IF (j=2) AND (g[j]=0) THEN t:=t-1 END IF                    (* pop 2 from stack *)
    END IF
  END IF
ELSE                                                                        (* g[1]=0 *)
  i:=p[t]; S:=n-g[i];
  IF S is odd THEN                                    (* g[i] should and can decrease *)
    IF g[i] is odd THEN g[j-1]:=g[j-1]+1 ELSE g[1]:=g[1]+1 END IF;
    g[i]:=g[i]-1;
    IF g[i]=0 THEN                                               (* pop i from stack *)
      IF i>2 THEN                                          (* and push i-1 onto stack *)
        p[t]:=i-1
      ELSE                                                  (* just pop i from stack *)
        t:=t-1
      END IF
    ELSE IF i>2 THEN                                      (* just push i-1 onto stack *)
      t:=t+1; p[t]:=i-1
    END IF
  ELSE                       (* g[i] should increase but can't.  We must handle g[i+1] *)
    i:=i+1; S:=S-g[i];
    IF S is odd THEN                                                     (* so is g[i] *)
      g[i]:=g[i]-1; g[i-1]:=g[i-1]+1;
      IF g[i]=0 THEN                 (* j ust pop i from stack since g[i-1] was already >0 *)
        t:=t-1; p[t]:=i-1;                             (* actually, from underneath i-1 *)
      END IF
    ELSE                                                     (* S is even and so is g[i] *)
      g[i]:=g[i]+1; g[i-1]:=g[i-1]-1;
      IF g[i]=1 THEN                                              (* push i onto stack *)
        IF g[i-1]=0 THEN                                       (* and pop i-1 from stack *)
          p[t]:=i
        ELSE                                          (* just insert i beneath i-1 on stack *)
          p[t]:=i; t:=t+1; p[t]:=i-1
        END IF
      ELSE IF g[i-1]=0 THEN                                   (* just pop i-1 from stack *)
        t:=t-1
      END IF
    END IF
  END IF
END IF;
RETURN.
```

Figure 3.9

A loop-free version of the Klingsberg algorithm for finding the next k-composition $g_1g_2...g_k$ of n using a stack $p_1p_2...p_t$ (initially the composition is n0...0, the stack is empty, and the top-of-stack index t=0).

The algorithm in [Kl] is essentially the one in Figure 3.8 with the following refinement. If g_i increases or decreases at the expense of g_{i-1} or g_1, all the integers $g_2,...,g_{i-2}$ are 0; so if $g_1=0$, then either g_{i-1} or g_i is the first positive element, and otherwise it is the second one. The algorithm in [Kl] stores the index of the first positive element, which is easy to update; if this is 1, the second one is sought by a linear search, and this is the only loop in the algorithm. Unfortunately, the index of the second positive element cannot be updated unless we know the indices of all the positive elements. To make this algorithm loop-free, then, we store the indices of all the positive elements except for g_1 on a stack, implemented with the array $p_1p_2...p_t$ which is sorted with the smallest index on top, and to update this stack we never have to touch more than the top two indices (see Figure 3.9). This seems like the only way to make this algorithm loop-free: the generic loop-free sequencing algorithm of Figure 1.1 is not guaranteed to work in this case because if $g_{i+1}+...+g_n=n$ then $a_i=z_i=0$. We note that a loop-free composition-generating algorithm already exists [Eh2], and a much simpler one which takes constant extra space can be constructed by applying the natural bijection between compositions and combinations [NW, p47] to the loop-free combination generator of Figure 3.6 (see Figure 3.12 below); we present this one to demonstrate the generality of our method.

To find Rank($g_1...g_k$), for i from k down to 2 we substitute into (1.1) if g_i is decreasing - that is, if $S_i=g_{i+1}+...+g_k$ is odd - and into (1.2) otherwise, and we then substitute into (1.3) if we are going to switch - that is, if g_i is odd. Now $\#(x,g_{i+1}...g_k)$ is the number of compositions of $n-S_i-x$ into i-1 parts, which is the number of combinations of $n-S_i-x+i-2$ objects taken i-2 at a time, and the sum in (1.1) or (1.2) has x running from its minimum value of g_{i+1} to its maximum value of $n-S_i$; so we have

$$\text{Rank}(g_1g_2\ldots g_k) = \binom{n+k-1}{k-1} - (n-g_1+1) \bmod 2 - \sum_{i=2}^{k} (-1)^{S_i} \begin{cases} \binom{n-S_{i-1}+i-1}{i-1}, & g_i \text{ odd} \\ \binom{n-S_{i-1}+i-2}{i-1}, & g_i \text{ even} \end{cases}, \quad (3.6)$$

where $S_i=g_{i+1}+...+g_k$.

The corresponding unranking algorithm is given in Figure 3.10. Both ranking and unranking can be done using O(n) arithmetic operations by computing each binomial coefficient from its predecessor.

```
t:= $\binom{n+k-1}{k-1}$ -r; s:= n;
                    (* t is greater by 1 than the q at the end of section 1 *)
FOR i FROM k DOWNTO 3 DO
  g[i]:= the largest integer such that $\binom{s-g[i]+i-1}{i-1} \geq t$;
  IF g[i] is odd then
    t:= $\binom{s-g[i]+i-1}{i-1} - t + 1$
  ELSE
    t:= t- $\binom{s-g[i]+i-2}{i-1}$
  END IF;
  s:=s-g[i];
END FOR;
g[1]:=t-1; g[2]:=s-t+1.
```

Figure 3.10
Finding the k-composition of n of rank r in the Knuth-Klingsberg Gray code.

But the natural bijection between the composition $g_1...g_k$ of n and the combination $c_1...c_{k-1}$ of n+k-1 given by $g_i=c_i-c_{i-1}-1$ (with c_0 taken to be 0) can be used to convert the Nijenhuis-Wilf-Liu-Tang Gray code for combinations into a Gray code for compositions. Here, $s(g_{i+1}...g_n)=(0,1,...,n-S_i)$ if n-k is odd and $(n-S_i,...,1,0)$ otherwise (see Figure 3.11, where the 5-compositions of 2 are listed beside the 4-combinations of 6 given in Figure 3.5).

combination	composition
1 2 3 4	0 0 0 0 2
1 2 4 5	0 0 1 0 1
2 3 4 5	1 0 0 0 1
1 3 4 5	0 1 0 0 1
1 2 3 5	0 0 0 1 1
1 2 5 6	0 0 2 0 0
2 3 5 6	1 0 1 0 0
1 3 5 6	0 1 1 0 0
3 4 5 6	2 0 0 0 0
2 4 5 6	1 1 0 0 0
1 4 5 6	0 2 0 0 0
1 2 5 6	0 0 2 0 0
1 2 4 6	0 0 1 1 0
2 3 4 6	1 0 0 1 0
1 3 4 6	0 1 0 1 0
1 2 3 6	0 0 0 2 0

Figure 3.11
Compositions corresponding to the combinations listed according to the Tang-Liu Gray code

Applying this bijection to the algorithm of Figure 3.6 we obtain the constant-time, constant-extra-space algorithm of Figure 3.12 for finding the next composition.

```
IF k=1 THEN Done:=TRUE; RETURN END;
LOOP                                        (* iterated AT MOST FOUR TIMES *)
  IF Rise THEN                                            (* g[i] should rise *)
    IF i>k THEN Done:=TRUE; RETURN END;
    IF g[i]<m THEN            (* g[i] can rise.   m=n-(g[i+1]+...+g[n])  *)
      g[i]:=g[i]+1;
      IF i>2 THEN
        m:=m-g[i];     i:=i-1;     g[i]:=m;     g[i-1]:=0;     Rise:=FALSE;
      ELSE
        g[1]:=g[1]-1;
      END IF i;
      RETURN;
    END IF g[i];                      (* else g[i] cannot rise so we increase i *)
  ELSE                                (* Rise is FALSE and g[i] should fall. *)
    IF g[i]>0 THEN                                          (* g[i] can fall *)
      g[i]:=g[i]-1;
      IF i>2 THEN
        m:=m-g[i]; i:=i-1; g[i]:=0; g[i-1]:=m;
        IF i=2 THEN Rise:=TRUE ELSE i:=i-1 END;
      ELSE
        g[1]:=g[1]+1;
      END IF i;
      RETURN;
    END IF g[i]                       (* else g[i] cannot fall so we increase i *)
  END IF Rise;
  i:=i+1; Rise:=NOT Rise; IF i≤k THEN m:=m+g[i] END;
END LOOP.
```

Figure 3.12
Constant-time constant-extra-space algorithm for generating the next k-composition of n (initially the composition is 0...0n, m=n, i=k, Rise=FALSE and Done=FALSE)

e) The Knuth-Kaye Gray code for set partitions

The classes of a partition of {1,2,...,n} are assumed to be indistiguishable; so they are ordered by their smallest member: (15)(24)(3), for instance. A partition is coded by the word $c_1c_2...c_n$, where c_i is the index of the class to which the number i belongs, so that (15)(24)(3) is coded by the word 12321.

A Gray code for set partitions, in which each partition differs from its predecessor in that one object moves from one class into another (possibly empty) class, was found by D. Knuth and communicated to H. Wilf, who described it in [Wi]. The description is recursive: given a Gray code for the partitions of {1,2,..,n-1}, n is inserted into each such partition in all possible ways, moving either from the first class to its own class after the last one or back the other way with the direction of motion changing for each partition of {1,2,..,n-1} (see Figure 3.13).

The graylex order that this Gray code imposes on the list of words $c_1c_2...c_n$ is as follows. For each prefix $c_1,...,c_{i-1}$, let m(i)=1+max($c_1,...,c_{i-1}$). Then s($c_1,...,c_{i-1}$) = (1,2,...,m(i)) if the rank of $c_1c_2...c_{i-1}$ among the partitions of {1,2,...,i-1} is even and (m(i),...,2,1) otherwise (see Figure 3.4: c_i begins by rising and changes direction of motion every time it hits its final value z_i).

PARTITION	c_1	c_2	c_3	c_4	d_1	d_2	d_3	d_4
(1234)	1	1	1	1	F	F	F	F
(123)(4)	1	1	1	2	F	F	F	F
(12)(3)(4)	1	1	2	3	F	F	F	T
(12)(34)	1	1	2	2	F	F	F	T
(124)(3)	1	1	2	1	F	F	F	T
(14)(2)(3)	1	2	3	1	F	F	T	F
(1)(24)(3)	1	2	3	2	F	F	T	F
(1)(2)(34)	1	2	3	3	F	F	T	F
(1)(2)(3)(4)	1	2	3	4	F	F	T	F
(1)(23)(4)	1	2	2	3	F	F	T	T
(1)(234)	1	2	2	2	F	F	T	T
(14)(23)	1	2	2	1	F	F	T	T
(134)(2)	1	2	1	1	F	F	T	F
(13)(24)	1	2	1	2	F	F	T	F
(13)(2)(4)	1	2	1	3	F	F	T	F

Figure 3.13
The Knuth-Kaye Gray code for set partitions of {1,2,3,4}; c_i is the class to which the object i belongs, and d_i is TRUE if c_i is descending.

The generic sequencing algorithm, specialized to this Gray code, leads to the algorithm in Figure 3.14. The maximum values for the c_i are stored in an auxilliary array: m_i is the value of $\max(c_1,...,c_{i-1})$. Since the parity of $c_1+...+c_{i-1}$ may or may not change as we pass from one partition of {1,2,...,i-1} to the next, there is no obvious way of calculating the direction in which c_i must move; so these directions are stored in an array of Boolean variables: d_i is TRUE if and only if c_i is descending.

```
i:=n;
LOOP
  IF d[i] THEN                                     (* c[i] should decrease *)
    IF c[i]=1 THEN                                 (* c[i] cannot decrease *)
      d[i]:=FALSE; i:=i-1
    ELSE                                           (* c[i] can increase *)
      c[i]:=c[i]-1; EXIT
    END IF
  ELSE                                             (* c[i] should increase *)
    IF c[i]>m[i] THEN                              (* c[i] cannot increase *)
      d[i]:=TRUE; i:=i-1;
      IF i=1 THEN Done:=TRUE; RETURN; END IF
    ELSE                                           (* c[i] can increase *)
      c[i]:=c[i]+1; EXIT
    END IF
  END IF
END LOOP;                 (* now we have to update the whole suffix c[i+1]...c[n] *)
WHILE i<n DO
  i:=i+1;
  m[i]:=max(c[i-1],m[i-1]);
  IF d[i] THEN c[i]:=m[i]+1; END IF;               (* its first value *)
END WHILE;
RETURN.
```

Figure 3.14
Finding the next set partition of {1,2,..,n} in the Knuth-Kaye Gray code; c_i is the class to which the object i belongs, m_i+1 is the largest value that c_i may attain, and d_i is TRUE if c_i is descending. Initially, c_i=1, m_i=1 and d_i=FALSE for each i from 1 to n (except that m_1=0), and Done is FALSE.

This is essentially the algorithm in [Ka], except that in [Ka] the maximum value of c_i is not stored explicitly. Instead, the value s[k], the smallest element in the class k, is stored; if s[c_i]=i, then c_i is at its maximum.

To solve the resumption problem for this algorithm we need to be able to calculate $m_1m_2...m_n$ and $d_1d_2...d_n$ from $c_1c_2...c_n$. The former can be done in linear time, but the latter is much more difficult: we have only the formula

$$d_i = \text{rank}(c_1c_2...c_{i-1}) \text{ is odd.} \tag{3.7}$$

This means that the resumption problem is dependent upon the ranking problem, but the ranking problem is also dependent upon the resumption problem, since we need to know the directions in which the integers are moving to be able to calculate the rank of a word. We solve both problems simultaneously below.

A formula for the lex-order rank of $c_1c_2...c_n$ is given in [Wm, p. 99]. A table of $t_{m,s}$, the number of ways to put s objects into classes if m classes are already occupied, is precomputed for $0 \le s \le n-2$ and $1 \le m \le n-s-1$ using the formula

$$t_{m,s} = \begin{cases} 1 & \text{if } s = 0 \\ mt_{m,s-1} + t_{m+1,s-1} & \text{otherwise.} \end{cases} \tag{3.8}$$

The part of the table necessary for n=7 is given in Figure 3.15.

s \ m=	1	2	3	4	5	6
0	1	1	1	1	1	1
1	2	3	4	5	6	
2	5	10	17	26		
3	15	37	77			
4	52	151				
5	203					

Figure 3.15
Table of $t_{m,s}$ needed to rank set partitions.

From (1.1),

$$\text{rank}(c_1c_2...c_ia_{i+1}...a_n)=\text{rank}(c_1c_2...c_{i-1}a_i...a_n)+(c_i-1)t_{m(i),n-i}, \tag{3.9}$$

so that

$$\text{lex-rank}(c_1c_2...c_n) = \Sigma_{i=2,...,n}\, (c_i-1)t_{m(i),n-i}\,. \tag{3.10}$$

The Gray code rank is calculated similarly, using (1.1) whether c_i is increasing or decreasing. If c_i is increasing, (3.9) still holds. Now suppose that c_i is decreasing. If $c_i>m(i)$, then $c_i=a_i$; so the sum in (1.1) is 0. Otherwise, it is equal to $\#(c_1,c_2,...,c_{i-1},a_i)$, which is $t_{m(i)+1,n-i}$, plus the rest of the sum, which is $(m(i)-c_i)t_{m(i),n-i}$. This observation, together with the observation that c_i decreases if and only if $\text{rank}(c_1c_2...c_{i-1})$ is odd, yields the formula

$$\text{rank}(c_1c_2...c_n) = \sum_{i=2}^{n} \begin{cases} (c_i - 1)t_{m(i),n-i} & \text{if rank}(c_1c_2...c_{i-1}) \text{ is even} \\ t_{m(i)+1,n-i} + (m(i) - c_i)t_{m(i),n-i} & \text{otherwise (or 0 if } c_i > m(i)), \end{cases} \quad (3.11)$$

where $m(i) = \max(c_1,c_2,...,c_{i-1})$.

The term $t_{m(i)+1,n-i}$ in (3.11) makes it necessary to compute the table of $t_{m,s}$ for $0\le s\le n-2$ and $1\le m\le n-s$.

We know that $\text{rank}(c_1)=0$, and once we have computed $\text{rank}(c_1c_2...c_{i-1})$ we can compute $\text{rank}(c_1c_2...c_i)$ from (3.11), which yields an algorithm, quadratic in the number of arithmetic operations, for ranking a word in graylex order.

```
FOR i FROM 1 TO n DO d[i]:=FALSE END FOR;
r:=0; m:=1;
FOR i FROM 2 TO n DO
  m:=max(m,c[i-1]);                    (* m = m(i) = max(c[1],..,c[i-1]). *)
  IF NOT d[i] THEN                     (* rank(c[1]c[2]...c[i-1]) is even.
*)
    IF c[i] is even THEN (* change d[j] for each j>i such that t[m(i),j-i] is
odd*)
      k:=1 - (m mod 2);                (* when j=i+1, t[m(i),j-i] ≡ m(i)+1 (mod 2)
*)
      FOR j FROM i+1 TO n DO
        IF k=0 THEN                    (* t[m(i),j-i] is even.
*)
          k:=2                         (* k follows j-i mod 3. *)
        ELSE
          d[j]:=NOT d[j];              (* now t(m(i),j-i) is odd. *)
          k:=k-1   (* when j=i+2, t[m(i),j]≡m(i) (mod 2); the next time it's odd. *)
        END IF
      END FOR
    END IF;
    r:=r+(c[i]-1)*t[m,n-i]
  ELSE                                 (* rank(c[1]c[2]...c[i-1]) is odd.
*)
    IF c[i]≤m  THEN
```

```
      IF m-c[i] is odd THEN   (* change d[j] if t[m(i)+1,j-i]+t[m(i),j-i] is odd. *)
        k:=2                  (* t[m(i)+1,j-i]+t[m(i),j-i] will be odd until j=i+3. *)
      ELSE                    (* change d[j] if t[m(i)+1,j-i] is odd. *)
        k:= m mod 2           (* when j=i+1, t[m(i)+1,j-i]≡m(i) (mod 2). *)
      END IF;
      FOR j FROM i+1 TO n DO
        IF k=0 THEN k:=2 ELSE d[j]:=NOT d[j]; k:=k-1 END IF;
      END IF;
      r:=r+t[m+1,n-i]+(m-c[i])*t[m,n-i]
    END IF
  END IF
END FOR.
```

Figure 3.16
Algorithm to find the rank r of the set-partition $c_1c_2...c_n$ in the Knuth-Kaye Gray code and the Boolean array $d_1d_2...d_n$, where d_i is TRUE if c_i is descending.

But since we only need to know the parity of the ranks of all the prefixes, we can do all but the last substitution into (3.11) modulo 2. From (3.8) it is easy to prove by induction on s that $t_{m,s}$ is odd if s mod 3 = 0, has the opposite parity as m if s mod 3 = 1, and has the same parity as m if s mod 3 = 2. To avoid storing or recomputing m(i), we use all the available information from each prefix before moving on to the next one. The algorithm is shown in Figure 3.16.

This algorithm also requires a quadratic number of operations, but the only operations which are iterated that often are on j, k and Boolean variables. There are only O(n) operations on large integers; so the algorithm actually runs in quadratic time. If all we want is $d_1d_2...d_n$, we can omit all reference to r; so we don't need to precompute the table.

This algorithm can be modified to find the word $c_1c_2...c_n$ of rank r and the corresponding $d_1d_2...d_n$. Remove all statements that contain r. After the second line (m:=1;) insert

```
c[1]:=1; s:=r;
```

After the fifth line (IF NOT d[i] THEN) insert

```
    c[i]:=1 + (s div t[m,n-i]); s:=s-(c[i]-1)*t[m,n-i];
```

Replace

```
     IF c[i]≤m
```

by

```
IF s<t[m+1,n-i] THEN
  c[i]:=m+1
ELSE
  s:=s-t[m+1,n-i];
  c[i]:=m - (s div t[m,n-i]);
  s:=s-(m-c[i])*t[m,n-i];
```

These modifications do not change the time-complexity.

Using these algorithms we find that the last word starting with 11 is 1121341561781.. (coding the partition (1247..)(3)(5)(6)(8)(9)..), its direction vector is FFFTFFTFFTFFT.., and its successor is 1231451671891.. (coding the partition (147..)(2)(3)(5)(6)(8)(9)..), so that 2/3 of its elements change. This means that the words do not obey any minimal-change property, so that there is no way to make the next-string algorithm loop-free with this order of generating partitions. Of course, there already is a loop-free set-partition generating algorithm ([Eh1], [Ev, p68]). The Gray code used there differs from this one in that $s(c_1c_2...c_{i-1})$ is (2,3,...,m(i),1) instead of (1,2,3,...,m(i)) and (1,m(i),...,3,2) instead of (m(i),...,3,2,1). This is genlex but it is not graylex unless the positive integers are reordered so that 1 is greater than all the others. All the Gray codes discussed in sections 3 and 4 of [JWW] are in a similar genlex order: 0 acts like the greatest integer, and the direction of motion changes with each new prefix. We invite the reader to apply our method to the set-partition Gray-code in [Eh1] to solve the ranking, unranking and resumption problems.

4. The Towers of Hanoi

In the by-now-infamous Towers of Hanoi puzzle (see [Hi] for a history of this problem), n rings of different sizes, originally stacked on a source peg in order of size with the smallest ring on top, have to be stacked on a destination peg, using a third peg as a spare, without moving more than one ring at a time or putting any ring on top of a smaller ring. The classical recursive algorithm for solving this problem is shown in Figure 4.1 and the classical iterative one (which was proved correct in [BL]) is shown in Figure 4.2.

```
IF n>1 THEN stack the n-1 smallest rings on the spare peg END IF;
Move the largest ring from the source peg to the destination peg;
IF n>1 THEN stack the n-1 smallest rings on the destination peg END IF.
```

Figure 4.1

The recursive algorithm for stacking n rings on the destination peg if they are initially stacked on the source peg.

```
Cyclically order the pegs so that destination follows source iff n is odd;
LOOP
  Move the smallest ring according to this cyclic order;
  IF all the rings are on a single peg THEN EXIT END IF;
  Move the second-smallest topmost ring but not on top of the smallest ring
END LOOP.
```

Figure 4.2
The classical iterative algorithm for stacking n rings on the destination peg if they are initially stacked on the source peg.

When we execute the iterative algorithm, the only variable information we need in order to decide what ring to move - aside from the topmost ring on each peg, which is part of the data structure being manipulated - is whether it is time to move the smallest ring. As was shown in [Wa1], this information can be found in constant time by examining the three topmost rings or empty pegs: if the rings are labelled 1,2,...,n in increasing order of size, the source peg labelled n+1, the spare peg n+2 and the destination peg n+3, then exactly one of the three topmost rings or empty pegs will have an even label, and this is where ring 1 should be moved, so that if this move is compatible with the cyclic order, it is time to move ring 1. Other results in [Wa1], which were described in [Hi] as open problems whose solutions could be derived from results in [Hi], are a criterion for the current position of the rings to be part of a minimal solution (every ring is on a ring or empty peg of opposite parity), and a linear-time ranking and unranking procedure (if $b_{n+1}b_n...b_2b_1$ is the rank in binary, then for each i from n down to 1, $b_i=b_{i+1}$ if and only if ring i is on ring or empty peg i+1).

If we store the positions of the rings in an array Peg, where Peg[i] is the peg on which ring i is sitting, it takes O(n) time in the worst case to find the second-smallest topmost ring - the computer searches the rings in increasing order of size until it finds one which is not on the same peg as its predecessor. When we execute the algorithm with real rings and pegs we compare the sizes of the three topmost rings, and it is both more natural and more efficient to incorporate this approach into a computer program. The rings on each peg form a sorted stack, and the allowable operation is popping a ring off the top of one stack and pushing it onto the top of another one (keeping the stacks sorted). This operation can be done in constant time using linked lists; so the algorithm can be made loop-free by storing the positions of the rings by a simulated linked list [Er2]: a size-3 array Top, where Top[i] is the topmost ring on peg i, and a size-n array Beneath, where Beneath[i] is the ring beneath ring i, with n+1 or some other sentinel to simulate the nil pointer. Since the array Peg is rendered redundant, this implementation uses constant extra space.

A recursive algorithm for stacking the rings onto a given destination peg from any legal position was found independently by Er [Er1] and Scarioni and Speranza [SS], and is shown in its essentials in Figure 4.3.

```
IF the largest ring is on the destination peg THEN
   IF n>1 THEN stack the n-1 smallest rings on the destination peg END IF
ELSE
   The 'other' peg := neither the destination peg nor the peg containing ring n;
   IF n>1 THEN stack the n-1 smallest rings on the 'other' peg END IF;
   Move the largest ring onto the destination peg;
   IF n>1 THEN stack the n-1 smallest rings on the destination peg END IF
END IF.
```

Figure 4.3
The recursive algorithm for stacking n rings on the destination peg from any legal position.

Iterative versions appear in [Hi], [Er2], and [Wa2]. All of these are equivalent to the recursive algorithm, and a proof that all these algorithms are indeed minimal appears in [Hi]. We compare them now for the minimal extra space needed for a loop-free implementation.

These algorithms all define a target peg for each ring according to the algorithm shown in Figure 4.4.

```
Target[n]:=destination peg;
FOR i FROM n-1 BY -1 TO 1 DO
  IF Peg[i+1]=Target[i+1] THEN
    Target[i]:=Target[i+1]
  ELSE
    Target[i]:=the peg other than Peg[i+1] or Target[i+1]
  END IF
END FOR.
```

Figure 4.4
Algorithm for finding the target pegs for each ring.

The iterative algorithm in [Hi] for stacking the rings on the destination peg is essentially the one shown in Figure 4.5.

```
Find Target[i] for each i;              (* using the algorithm of Figure 4.4
*)
FOR i:=1 TO n DO
  IF Peg[i]≠Target[i] THEN
    Peg[i]:=Target[i];
    IF i>1 THEN                 (* the smaller rings are all stacked on Target[i-1]
*)
      Stack all the smaller rings on Peg[i]   (* algorithm of Fig. 4.2
*)
    END IF;
  END IF
END FOR.
```

Figure 4.5
Er-Hinz iterative algorithm for stacking n rings on the destination peg from any legal position.

The iterative algorithm in [Er2] is essentially the same, except that an explicit loop-free version is given, using the arrays Top and Beneath as well as Peg and Target. The array Target is constantly updated to keep ring 1 moving in the correct cyclic direction. In any loop-free implementation of the algorithm in [Hi] we would need all these arrays, although it would not be necessary to update Target: it suffices to update the cyclic direction for ring 1 for each new value of i.

The algorithm in [Wa2], which does not store the array Target, is given in Figure 4.6.

```
Find Target[1]; (* Fig. 4.4 with all indices of the array Target changed to 1 *)
IF ring 1 is not on Target[1] THEN move it there END IF;
WHILE at most one peg is empty DO
  Move second-smallest topmost ring i but not onto ring 1;
  IF i is even THEN
    Move ring 1 onto ring i
  ELSE
    Move ring 1 but not onto ring i
  END IF
END WHILE.
```

Figure 4.6
Walsh algorithm for stacking n rings on the destination peg from any legal position.

This algorithm is proved correct in [Wa2] by showing, using induction on n, that the recursive algorithm makes the same sequence of moves. Here we derive our algorithm from the recursive one using genlex order. From the recursive algorithm of Figure 4.3 we see that if ring n is not on its target peg - the destination peg - it will be moved there before it is moved anywhere else. The same is true of any other ring i - this is arranged by the recursive call for stacking i rings - and no larger ring will be moved in the meantime. It follows that for any position of the larger rings, ring i assumes either one position, Target[i], or two, the latter of which is Target[i]. This defines a genlex order on the positions of the rings, and the generic sequencing algorithm says to find the smallest ring i which is not on Target[i], if there is one, and move it there. We can calculate Target[1] without storing the array Target, and if ring 1 is not on Target[1] it must be moved there. To find the smallest ring i which is not on Target[i], we observe from Figure 4.4 that for any $j>1$, Peg[j], Target[j] and Target[j-1] are either all the same or all different. It can be shown by induction on j that if rings 1,2,...,i-1 are all on the same peg they are all on their target pegs. If all n rings are on the same peg, then this peg is Target[n]=destination, and the algorithm must terminate. Otherwise, for the second-smallest topmost ring i, Peg[i], Target[i] and Target[i-1]=Peg[i-1]=Peg[1] are all different, and ring i must be moved onto Target[i], which is the peg containing neither ring 1 nor ring i. Now Target[i-1] becomes Target[i], the new value of Peg[i]. But now Target[i-1], Peg[i-1] and the old value of Peg[i] are all different, so that Target[i-2] becomes the old value of Peg[i]. Continuing in this way we find that Target[i-3] becomes the new value of Peg[i], Target[i-4] becomes the old value of Peg[i] and so on, so that Target[1] becomes the new value of Peg[i] if i is even and the old value of Peg[i] if i is odd, and in either case it is different from Peg[1]; so that ring 1 must be moved onto ring i if i is even and onto the peg containing neither ring i nor ring 1 if i is odd. Thus the generic sequencing algorithm leads directly to the algorithm in Figure 4.6.

We define the binary number $b_n \ldots b_2 b_1$ where $b_i=0$ if and only if Peg[i]=Target[i] and prove that each move reduces this number by 1. In the above discussion we showed that at the beginning of the loop rings 1,2,...,i-1 are on their target pegs but not ring i, so that $b_i=1$ but $b_{i-1}=\ldots=b_1=0$. The move of ring i puts it on its target peg but changes the target pegs of all the rings 1,2,...,i-1, so that now $b_i=0$ but $b_{i-1}=\ldots=b_1=1$, which reduces the binary number by 1. Then ring 1 is moved to its target peg, changing b_1 from 1 to 0, again reducing this number by 1. When the binary number drops to 0, all the rings are on a single peg - the destination peg - and the algorithm terminates, so that the initial value of this binary number is the number of moves required to complete the algorithm (this fact was proved in both [Wa2] and [Hi]).

We note that this algorithm provides an alternative to Figure 4.2 for solving the original Towers of Hanoi problem. In this special case, the first two lines can be replaced with 'IF n is odd THEN move ring 1 onto the destination peg ELSE move ring 1 onto the spare peg END IF'.

Any loop-free implementation of this algorithm requires the arrays Top and Beneath to make it loop-free and Peg to find Target[1] by scanning the rings from largest to smallest as done in Figure 4.4, since the rings cannot be scanned from largest to smallest using the arrays Top and Beneath. They can, however, be scanned from smallest to largest using these two arrays and a size-3 array Current. If we assume that Target[1]=Peg[1], we can then use that fact that Target[i], Peg[i] and Target[i-1] are either all the same or all different to find what the value of Target[n] would be under that assumption - that is, on what peg all the rings will end up being stacked if we don't move ring 1 first. If this turns out to be the destination peg, ring 1 is on Target[1] and we don't have to move it. If not, we order the pegs cyclically so that the destination peg follows the value that Target[n] would have if Target[1] were Peg[1]. If we change Target[n] to destination, Target[n-1] moves in the opposite cyclic order, Target[n-2] in the original cyclic order, and so on, so that to get the true value of Target[1] we have to move it from the peg containing ring 1 in the above-mentioned cyclic order if n is odd and in the opposite cyclic order if n is even, and this is the cyclic order in which we have to move ring 1. After that we follow Figure 4.6, finding the second-smallest topmost ring by sorting the size-3 array Top and moving the rings by popping and pushing. Each move is made in constant time, so that the algorithm is loop-free, and since only one size-n array is used - Beneath - the algorithm uses O(1) extra space.

A more general problem is solved in [Hi]: moving the rings from one legal position to another. Two algorithms are considered. Let ring i be the largest ring that is not on its destination peg Dest[i]. Ring i is moved onto Dest[i] in either one move or two. Before moving ring i, all the smaller rings must be stacked on the appropriate 'other' peg. Once ring i is on Dest[i], the sequence of moves which would take the smaller rings from their destination pegs and stack them on the peg on which they are currently stacked is reversed. One of these two algorithms is minimal, and since one can compute the number of moves that each of them will take from the formula in [Hi] and [Wa2], one chooses the minimal algorithm and then executes it.

The smaller rings can be stacked the first time using Figure 4.6 and the second time, if necessary, using Figure 4.2. The hard part is the reversal of the sequence of moves which stacks the rings on a given peg from any legal position. But to take rings from a single peg and put them onto designated pegs it is not necessary to even consider the reverse problem, let alone store the entire sequence of moves necessary to solve the reverse problem and then reverse the sequence. The algorithm in Figure 4.7 solves this problem directly and does not require the foresight of calculating temporary target pegs. It is clearly the unique minimal-move algorithm in which the largest ring which is not on its destination peg moves only once, and we have it on the authority of [Hi] that it is therefore the minimal algorithm which does the job.

```
FOR i FROM n BY -1 TO 1 DO
  IF ring i is not on peg Dest[i] THEN
    Stack all the smaller rings                (* which are now on ring i
*)
    on the 'other' peg besides Dest[i];                (* Figure 4.2
*)
    Move ring i to Dest[i]
  END IF
END FOR.
```

Figure 4.7
An algorithm for taking n rings which are stacked on a source peg
and putting each ring i on a specified peg Dest[i].

To solve the general problem we need to know how many moves the algorithms of Figures 4.6 and 4.7 will take. An algorithm for finding in advance the moves that Figure 4.6 will take while calculating Target[1] appears in [Wa2]; we repeat it in Figure 4.8.

```
Target[1]:=destination; m:=0;
FOR i FROM n BY -1 TO 2 DO
  IF Peg[i]≠Target[1] THEN
    Target[1]:=the peg other than Target[1] or Peg[i]
    m:=m+1;
  END IF;
  m:=m*2;
END FOR;
IF Peg[1]≠Target[1] THEN m:=m+1 END IF.
```

Figure 4.8
An algorithm for finding m, the number of moves needed to execute Figure 4.6, while calculating Target[1]

A similar algorithm (see Figure 4.9) computes the number of moves necessary to execute Figure 4.7.

```
StackPeg:=source; m:=0;       (* StackPeg is the destination for a stack of rings
*)
FOR i FROM n BY -1 TO 1 DO
  m:=m*2;
  IF StackPeg≠Dest[i] THEN
    StackPeg:=the peg other than StackPeg or Dest[i];
    m:=m+1
  END IF
END FOR.
```

Figure 4.9
An algorithm for finding m, the number of moves needed to execute Figure 4.7

Thus the entire algorithm for moving the rings from one legal position to another can be implemented using the array Dest (which, being part of the problem, is not considered extra space) and whatever representation we are using for the position of the rings. If we are prepared to search for the second-smallest topmost ring we can represent the rings by the array Peg; otherwise we need the arrays Top and Beneath. The algorithm of Figure 4.7 will still not be totally loop-free, because the condition that ring i is not on peg Dest[i] may be false for several consecutive rings. However, the violation of the loop-free condition is only linear in total time, since that condition is tested only once for each ring and can be done in constant time even if we only use Top and Beneath, since the ring to be tested is on the same peg as ring 1. If we use only Top and Beneath we must modify the algorithm of Figure 4.8 so that it computes Target[1] by scanning the rings from smallest to largest and evaluates m starting from the least significant bit of its binary expansion. With these modifications, the entire algorithm is almost loop-free and uses constant extra space.

From the algorithm of Figure 4.7 it is easy to derive the genlex order in which the rings move. For each position of the larger rings, ring i follows a sequence of one peg ($a_i=z_i$) or a sequence of two pegs (a_i,z_i). If i=n or all the larger rings are on their destination pegs, then z_i=Dest[i]; otherwise z_i is computed from Peg[i+1] and z_{i+1} the way Target[i] is computed in Figure 4.4: if Peg[i+1]=z_{i+1} then $z_i=z_{i+1}$ and otherwise z_i= the peg other than Peg[i+1] and z_{i+1}. Similarly, a_n is the source peg, and for each i<n, if Peg[i+1]=a_{i+1} then $a_i=a_{i+1}$ and otherwise a_i= the peg other than Peg[i+1] and a_{i+1}. The genlex order followed by the rings moving from a given legal position (Peg[i]=Source[i]) to a single destination peg can be derived from the above by reversing the roles of a_i and z_i and of Source and Destination. For the original Towers of Hanoi problem, each ring follows a two-peg sequence: a_n=source, z_n=destination, and for i<n, if Peg[i+1]=z_{i+1} then $z_i=z_{i+1}$ and a_i= the peg other than a_{i+1} and z_{i+1}, and otherwise $a_i=a_{i+1}$ and z_i is the 'other' peg. For the more general problem of moving the rings from one legal position to another, the largest ring not originally on its destination peg follows either a two-peg or three-peg sequence depending on which of the two algorithms presented in [Hi] turns out to be minimal, and the other rings follow one of the three above genlex orders depending upon which part of the algorithm is being executed; the details are left to the reader. We note that this genlex order is not graylex unless we artificially impose a linear order on the pegs instead of the more natural cyclic order.

5. Conclusion

The method presented here can be used to sequence and rank any list of words in genlex order, and this includes almost all the classical combinatorial Gray codes. There are some exceptions: the list of integer partitions of 10 shown in [Wi, p12], computed using the Gray code published in [Sa], will convince the reader that this Gray code is not in genlex order, and so it is unlikely that an iterative description can be found by using any method less general (and less complicated) than the revolving door method illustrated in [NW, p29]. This leads to the following open problem. The 'method' presented in this article begins by generating the list L_n for small values of n from the recursive description, observing if it is in genlex order and, if so, discovering a rule for finding $s(g_1g_2...g_{i-1})$ as a function of $g_1g_2...g_{i-1}$, and then proving these observations by induction using the recursive description. Does there exist an algorithm which accepts as input the recursive description of a Gray code and decides whether or not it is genlex and, if so, outputs the simplest possible rule for finding $s(g_1g_2...g_{i-1})$ as a function of $g_1g_2...g_{i-1}$?

REFERENCES

[BER]: J.R. Bitner, G. Ehrlich, E.M. Reingold, Efficient Generation of the Binary Reflected Gray Code and its Applications, Comm. ACM 19 (1976), p. 517-521.

[BL]: P. Buneman and L. Levy, The towers of Hanoi problem, Information Processing Letters 10 (1980), p. 243-244.

[Ch]: P.J. Chase, Combination generation and graylex ordering, Proceedings of the 18th Manitoba Conference on Numerical Mathematics and Computing, Winnipeg, 1988, Congressus Numerantium 69 (1989), p. 215-242.

[Ds]: N. Dershowitz, A simplified loop-free algorithm for generating permutations, BIT 15 (1975), 158-164.

[Eh1]: G. Ehrlich, Loopless algorithms for generating permutations, combinations, and other combinatorial configurations, J. ACM 20 (1973), p. 500-513.

[Eh2]: G. Ehrlich, Algorithm 466, Four Combinatorial Algorithms [G6], Comm. ACM 16 (1973), p. 690-691.

[Er1]: M.C. Er, The generalized towers of Hanoi problem, J. Inform. Optim. Sci. 5 (1984), p. 89-94.

[Er2]: M.C. Er, An iterative solution to the generalized towers of Hanoi problem, BIT 23 (1983), 295-302.

[Ev]: S. Even, Algorithmic combinatorics, MacMillan, N.Y. 1973.

[Fe]: W. Feller, An introduction to probability theory and its applications, Vol. 1, Second edition, Wiley and Sons, N.Y., 1957.

[Gr]: F. Gray, Pulse code communication, U.S. Patent 2 632 058, March 17, 1953.

[Hi]: A.M. Hinz, The tower of Hanoi, L'Enseignement Mathématique 35 (1989), p. 289-321.

[Jo]: S.M. Johnson, Generation of permutations by adjacent transpositions, Mathematics of Computation 17 (1963), p. 282-285.

[JWW]: J.T. Joichi, D.E. White and S.G. Williamson, Combinatorial Gray Codes, SIAM J. Computing 9 (1980), p. 130-141.

[Ka]: R. Kaye, A Gray code for set partitions, Information Processing Letters 5 (1976), p. 171-173.

[Kl]: P. Klingsberg, A Gray code for compositions, Journal of Algorithms 3 (1982), p. 41-44.

[Kn]: D.E. Knuth, The art of computer programming, Vol. 3 (sorting and searching), Addison-Wesley, Reading, Mass., 1973.

[Le]: D.H. Lehmer, The machine tools of combinatorics, in Applied Combinatorial Mathematics, E.F. Beckenbach (ed.), Wiley and Sons, N.Y., 1964, p. 5-31.

[LT]: C.N. Liu and D.T. Tang, Algorithm 452, Enumerating M out of N objects, Comm. ACM 16 (1973), p. 485.

[NW]: A. Nijenhuis and H.S. Wilf, Combinatorial algorithms for computers and calculators, second edition, Academic Press, N.Y., 1978.

[Pa]: J. Pallo, Enumerating, ranking and unranking binary trees, Computer Journal 29 (1986), 171-175.

[PL]: A. Proskurowski and E. Laiman, Fast enumeration, ranking and unranking of binary trees, Proceedings of the 13th Southeast conference on Computing, Graph Theory and Combinatorics, Congressus Numerantium 35 (1982), 401-413.

[PR]: A. Proskurowski and F. Ruskey, Generating binary trees by transpositions, Proceedings of the first SWAT conference, Stockholm, 1988, Lecture Notes in Computer Science 318, Springer-Verglag, N.Y., 1988, p. 199-207.

[RH]: F. Ruskey and T.C. Hu, Generating binary trees lexicographically, SIAM J. Computing 6 (1977), p. 745-758.

[RND]: E.M. Reingold, J. Nievergelt and N. Deo, Combinatorial algorithms, theory and practice, Prentice-Hall, New Jersey, 1977.

[Sa]: C. Savage, Gray code sequences of partitions, Journal of Algorithms 10 (1989), 577-595.

[SS]: F. Scarioni and M.G. Speranza, A probabilistic analysis of an error-correcting algorithm for the towers of Hanoi puzzle, Information Processing Letters 18 (1984), p. 99-103.

[Tr]: H.F. Trotter, Algorithm 115: Perm, Communications of the ACM 5 (1962), 434-435.

[vB]: D.R. van Baronaigien, A loopless algorithm for generating binary tree sequences, Information Processing Letters 39 (1991), p. 189-194.

[Wa1]: T.R. Walsh, The towers of Hanoi revisited: moving the rings by counting the moves, Information Processing Letters 15 (1982), 64-67.

[Wa2]: T.R. Walsh, A case for iteration, Proceedings of the 14th Southeastern Conference on Computing, Graph Theory and Combinatorics, 1983, Congressus Numerantium 40 (1983), p. 38-43.

[Wi]: H. S. Wilf, Combinatorial algorithms: an update, SIAM, Philadelphia, 1989.

[Wm]: S.G. Williamson, Combinatorics for computer science, Computer Science Press, Rockville, 1985.

[Z]: S. Zaks, Lexicographic generation of ordered trees, Theoretical Computer Science 10 (1980), p. 63-82.

APPENDIX

Listings of the computer programs for sequencing, ranking and unranking the Proskurowski-Ruskey Gray code for balanced parenthesis systems

Two modules were written: RankTree, which contains the constant-extra-space sequencing algorithm and the ranking and unranking algorithms, and EhrlichTree, which contains the loop-free sequencing algorithm. This page contains the declarations of both modules, the next five pages contain the rest of RankTree, and the last three pages contain the rest of EhrlichTree. The bitstring is stored in the variable A. The variables n and k are the parameters in T(n,k): the bitstring is of length 2n and has a prefix 1^k0. The variable i is a loop index for printing the bitstring, r is for its rank, and B is for constructing the bitstring of that rank for comparison with A. Done is true if the current bitstring is the last one in the list. LastRight is true if the total number of free 1s not in their rightmost positions is even. OK is true if r is less than the total number of bitstrings. Pos is the position vector of the 1s in the bitstring, Dir[i]=1 if Pos[i] is increasing and 0 otherwise, and Ehr is the Bitner-Ehrlich-Reingold array. Finally, List is Fixed if we are generating T(n,k) for fixed k, Variable if we are generating T(n,n)oT(n,n-1)o...oT(n,2)oT^R(n,1) and Special if we are generating T(n+1,1) with the prefix 10 removed.

```
MODULE RankTree; (* Constant extra-space generation, ranking
and unranking of balanced parenthesis systems *)
FROM InOut IMPORT WriteString, WriteLn, ReadCard, WriteCard;
TYPE Dyck=ARRAY[1..80] OF CARDINAL;
     ListType=(Fixed, Variable, Special);
VAR i,n,k,r: CARDINAL;
    Done, LastRight, OK: BOOLEAN;
    A,B: Dyck;
    List: ListType;
```

```
MODULE EhrlichTree; (* Loop-free generation of balanced
parenthesis systems *)
FROM InOut IMPORT WriteString, WriteLn, ReadCard, WriteCard;
TYPE Dyck=ARRAY[1..80] OF CARDINAL;
     ListType=(Fixed, Variable, Special);
VAR i,n,k: CARDINAL;
    Done: BOOLEAN;
    A, Pos, Dir, Ehr: Dyck;
    List: ListType;
```

```
PROCEDURE First(n:CARDINAL; VAR k:CARDINAL; VAR A:Dyck;
                VAR Done, LastRight: BOOLEAN; List:
ListType);
(* FIRST PARENTHESIS SYSTEM.  NO AUXILIARY ARRAYS NEEDED *)
VAR i: CARDINAL;
BEGIN
  Done:=FALSE;
  IF List=Variable THEN
    k:=n; LastRight:=TRUE
  ELSIF List=Special THEN
    k:=1; LastRight:=n<=1
  ELSE
    LastRight:=(k=n) OR (n<=2)
  END; (* IF List *)
  FOR i:=1 TO k DO
    A[i]:=1;
  END; (* FOR *)
  IF k=n THEN
    FOR i:=n+1 TO 2*n DO
      A[i]:=0;
    END; (* FOR *)
  ELSIF (k>1) OR ((n=2) AND (List=Fixed)) THEN
    A[k+1]:=0; A[k+2]:=1;
    FOR i:=k+3 TO 2*k+2 DO
      A[i]:=0
    END; (* FOR *)
    FOR i:=k+2 TO n DO
      A[2*i-1]:=1; A[2*i]:=0;
    END; (* FOR *)
  ELSE (* k=1 *)
    IF List=Special THEN
      IF n=1 THEN
        A[2]:=0;
      ELSE
        A[2]:=1; A[3]:=0; A[4]:=0;
        FOR i:=3 TO n DO
          A[2*i-1]:=1; A[2*i]:=0;
        END; (* FOR *)
      END; (* IF n *)
    ELSE
      A[2]:=0;
      IF n=2 THEN
        A[3]:=1; A[4]:=0;
      ELSIF n>2 THEN
        A[3]:=1; A[4]:=1; A[5]:=0; A[6]:=0;
        FOR i:=4 TO n DO
          A[2*i-1]:=1; A[2*i]:=0;
        END; (* FOR *)
      END (* IF n *)
    END; (* IF List *)
  END; (* IF k *)
END First;
```

```
PROCEDURE Next(n: CARDINAL; VAR k: CARDINAL; VAR A: Dyck;
               VAR Done, LastRight: BOOLEAN; List: ListType);
(* NEXT  PARENTHESIS SYSTEM.  NO AUXILIARY ARRAYS NEEDED *)
VAR i,j: CARDINAL;
    Right: BOOLEAN;
BEGIN
  i:=2*n-1;                          (* position of symbol in Dyck-word *)
  j:=n;                                        (* index among 1s only
*)
  Right:=LastRight;     (* true if the number of free 1s to the left of A[j] is even *)
  LOOP
    IF ((List=Fixed) AND (j<=k)) OR (j<=1) THEN
      Done:=TRUE; EXIT;
    END; (* IF *)
    WHILE A[i]=0 DO i:=i-1; END; (* WHILE *)
    IF (i<2*j-1) AND ((List<>Variable)OR(i>k)) THEN
      Right:=NOT Right;
    END; (* IF *)
    IF Right AND (i<2*j-1) THEN                      (* jth 1 moves right
*)
      A[i]:=0;
      IF j=k THEN                            (* List=Variable and k drops
*)
        A[i+1]:=1; LastRight:=NOT LastRight; k:=k-1;
      ELSIF j=n THEN
        A[i+1]:=1;
        IF i+1=2*j-1 THEN LastRight:=NOT LastRight; END;
      ELSE
        LastRight:=NOT LastRight;
        IF i+1=2*j-1 THEN A[i+2]:=1 ELSE A[2*j+1]:=1 END;
      END; (* IF j *)
      EXIT;
    ELSIF (NOT Right) AND (A[i-1]=0)
    AND((List<>Variable)OR(k>1)OR(j>2)) THEN       (* jth 1 moves left *)
      A[i-1]:=1;
      IF j=n THEN
        A[i]:=0;
        IF i=2*j-1 THEN  LastRight:=NOT LastRight; END;
      ELSE
        LastRight:=NOT LastRight;
        IF i=2*j-1 THEN
          A[i+1]:=0
        ELSE
          A[2*j+1]:=0;
        END; (* IF i *)
      END; (* IF j *)
      EXIT;
    ELSE                                              (* jth 1 can't move
*)
```

```
      i:=i-1; j:=j-1;
    END; (* IF Right *)
  END; (* LOOP *)
END Next;
```

```
PROCEDURE Rank(n,k:CARDINAL; A:Dyck; List:ListType):
CARDINAL;
VAR i,j,f,t,r,d:CARDINAL;
    Dir: BOOLEAN;
BEGIN
  IF List=Variable THEN
    k:=1;
    WHILE A[k+1]=1 DO k:=k+1 END;
    IF k=1 THEN k:=2 END;
  ELSIF List=Special THEN
    k:=1; n:=n+1;
  END; (* IF List *)
  f:=1;                              (* computing first binomial coefficient
*)
  FOR i:=1 TO n-k DO
    f:=f*(2*n-k-i) DIV i;
  END; (* FOR *)
  IF List=Variable THEN   (* computing rank of last bitstring for this value of k
*)
    r:=f*(2*n-k)*(k+1) DIV (n*(n+1));
    IF A[2]=1 THEN  (* k was always greater than 1 - if k was 1 we changed it to 2
*)
      Dir:=TRUE;
      r:=r-1;
    ELSE
      Dir:=FALSE;
    END; (* IF A[2] *)
  ELSE
    r:=(f*k DIV n) - 1;
    Dir:=TRUE;
  END; (* IF *)
  j:=k+2;
  IF List=Special THEN  (* subtract 2 from all indices of A to remove prefix 10 *)
    d:=2
  ELSE
    d:=0;
  END; (* IF List *)
  FOR i:=k+1 TO n DO
    WHILE A[j-d]=0 DO
      f:=f*(n+i-j) DIV (2*n-j+1);          (* updating binomial coefficient
*)
      j:=j+1;
    END; (* WHILE *)
    IF j<2*i-1 THEN
      t:=(f*(2*i-j) DIV (n-j+i+1)) - 1;
      IF Dir THEN r:=r-t ELSE r:=r+t END;          (* updating rank
*)
      Dir:=NOT Dir;
    END; (* IF j *)
    f:=f*(n-i+1) DIV (2*n-j+1);
                              (* updating binomial coefficient when both i and j change *)
    j:=j+1;
  END; (* FOR *)
  RETURN(r);
END Rank;
```

```
PROCEDURE Unrank(n,k,r:CARDINAL; VAR A:Dyck;
                VAR OK: BOOLEAN; List:ListType);
VAR i,j,f,nf,d: CARDINAL;
BEGIN
  IF List=Variable THEN
    k:=n; f:=0; nf:=1;
    WHILE r>=nf*(k+1) DIV (n+1) DO
      IF k=1 THEN OK:=FALSE; RETURN; END;
      k:=k-1; f:=nf;
      nf:=f*(2*n-k) DIV (n-k);
    END; (* WHILE *)
    r:=r-f*(k+2) DIV (n+1);
  END; (* IF *)
  IF List=Special THEN
    d:=2; k:=1; n:=n+1;
  ELSE
    d:=0;
    FOR i:=1 TO k DO
      A[i]:=1;
    END; (* FOR *)
    A[k+1]:=0;
  END; (* IF *)
  f:=1;
  FOR i:=1 TO n-k DO
    f:=f*(2*n-k-i) DIV i;
  END; (* FOR *)
  nf:=f*k DIV n;
```
(*nf means new value of f; when nf meets the stopping condition we use f*)

```
  OK:=r<nf;
  IF NOT OK THEN RETURN END;
  IF (List<>Variable) OR (k>1) THEN
    r:=(f*k DIV n) - 1 - r;
  END; (* IF *)
  j:=k+2;
  FOR i:=k+1 TO n DO
    LOOP
      nf:=f*(n+i-j) DIV (2*n-j+1);
      IF nf*(2*i-j-1) DIV (n+i-j) <= r THEN EXIT END;
      A[j-d]:=0; j:=j+1; f:=nf;
    END; (* LOOP *)
    A[j-d]:=1;
    IF j<2*i-1 THEN
      r:=(f*(2*i-j) DIV (n-j+i+1)) - 1 - r;
    END; (* IF *)
    f:=f*(n-i+1) DIV (2*n-j+1);
    j:=j+1;
  END; (* FOR *)
  FOR i:=j TO 2*n DO
    A[i-d]:=0;
  END; (* FOR *)
END Unrank;
```

```
BEGIN (* RankTree *)
  WriteString('Generating Dyck-Words by Transpositions.');
WriteLn;
  LOOP
    WriteLn;
    WriteString('Enter number of pairs, 0 to quit: ');
    ReadCard(n);
    IF n=0 THEN
      EXIT
    END; (* IF n *)
    WriteString('Enter length of prefix of 1s, 0 for variable:
');
    ReadCard(k);
    IF (k>n) THEN
      WriteString('The maximum is '); WriteCard(n,1);
WriteLn;
    ELSE
      IF k>0 THEN
        List:=Fixed;
      ELSE (* k=0 *)
        WriteString('Enter 1 to do n=n+1 and k=1, 0 otherwise:
');
        ReadCard(k);
        IF k>0 THEN List:=Special ELSE List:=Variable END;
      END; (* outer IF k *)
      First(n,k,A,Done,LastRight,List);
      WriteLn;
      WHILE NOT Done DO
        FOR i:=1 TO 2*n DO
          WriteCard(A[i],1);
        END; (* FOR *)
        r:=Rank(n,k,A,List);
        WriteCard(r,10);
        Unrank(n,k,r,B,OK,List);
        IF OK THEN
          WriteString("    ");
          FOR i:=1 TO 2*n DO
            WriteCard(B[i],1);
          END; (* FOR *)
        ELSE
          WriteString("Rank too big");
        END; (* IF *)
        WriteLn;
        Next(n,k,A,Done,LastRight,List);
      END; (* WHILE *)
      Unrank(n,k,r+1,B,OK,List);
      IF OK THEN
        WriteString("Out of bounds test NOT
working");WriteLn;
      ELSE
```

```
        WriteString("Out of bounds test working");WriteLn;
      END; (* IF *)
    END; (* IF k *)
  END; (* LOOP *)
END RankTree.
```

```
PROCEDURE First(n:CARDINAL; VAR k:CARDINAL;VAR Done: BOOLEAN;
                VAR A,Pos,Dir,Ehr:Dyck;  List: ListType);
                               (* for loop-free algorithm
*)
VAR i: CARDINAL;
BEGIN
  Done:=FALSE;
  IF List=Variable THEN
    k:=n;
  ELSIF List=Special THEN
    k:=1;
  END; (* IF List *)
  FOR i:=1 TO k DO
    A[i]:=1; Pos[i]:=i;
  END; (* FOR *)
  IF k=n THEN
    FOR i:=n+1 TO 2*n DO
      A[i]:=0;
    END; (* FOR *)
  ELSIF (k>1) OR ((n=2) AND (List=Fixed)) THEN
    A[k+1]:=0; A[k+2]:=1; Pos[k+1]:=k+2;
    FOR i:=k+3 TO 2*k+2 DO A[i]:=0 END; (* FOR *)
    FOR i:=k+2 TO n DO
      A[2*i-1]:=1; A[2*i]:=0; Pos[i]:=2*i-1;
    END; (* FOR *)
  ELSE (* k=1 *)
    IF List=Special THEN
      IF n=1 THEN
        A[2]:=0;
      ELSE
        A[2]:=1; A[3]:=0; A[4]:=0; Pos[2]:=2;
        FOR i:=3 TO n DO
          A[2*i-1]:=1; A[2*i]:=0; Pos[i]:=2*i-1;
        END; (* FOR *)
      END; (* IF n *)
    ELSE
      A[2]:=0;
      IF n=2 THEN
        A[3]:=1; A[4]:=0; Pos[2]:=3;
      ELSIF n>2 THEN
        A[3]:=1; A[4]:=1; A[5]:=0; A[6]:=0; Pos[2]:=3;
Pos[3]:=4;
        FOR i:=4 TO n DO
          A[2*i-1]:=1; A[2*i]:=0; Pos[i]:=2*i-1;
        END; (* FOR *)
      END (* IF n *)
    END; (* IF List *)
  END; (* IF k *)
  FOR i:=1 TO n DO
    IF Pos[i]=2*i-1 THEN Dir[i]:=0 ELSE Dir[i]:=1 END;
    Ehr[i]:=i;
  END; (* FOR *)
END First;
```

```
PROCEDURE Next(n: CARDINAL; VAR k: CARDINAL;VAR Done: 
BOOLEAN;
                VAR A,Pos,Dir,Ehr:Dyck;  List: ListType);
(* LOOP-FREE SEQUENCING ALGORITHM *)
VAR i,j: CARDINAL;
BEGIN
  j:=Ehr[n];                                    (* the jth 1 is to be moved *)
  IF ((List=Fixed) AND ((j<=k) OR (j<=2))) OR (j<=1) THEN
    Done:=TRUE; RETURN;
  END; (* IF *)
  Ehr[n]:=n;
  i:=Pos[j];                                    (* the position of the jth 1 *)
  IF i=2*j-1 THEN                               (* it must move left *)
    Dir[j]:=0;
  END; (* IF *)
  IF Dir[j]=1 THEN                              (* jth 1 moves right *)
    A[i]:=0; Pos[j]:=i+1;
    IF j=k THEN                                 (* List=Variable and k drops
*)
      A[i+1]:=1;
      k:=k-1;
    ELSIF j=n THEN                              (* only one 1 moves *)
      A[i+1]:=1;
    ELSE                  (* the jth 1 displaces the j+1st 1 which moves too. *)
      IF i+1=2*j-1 THEN                         (* It moves one space to the right *)
        A[i+2]:=1; Pos[j+1]:=i+2; Dir[j+1]:=1;
      ELSE                                      (* It moves as far right as possible. *)
        A[2*j+1]:=1; Pos[j+1]:=2*j+1; Dir[j+1]:=0;
      END; (* IF i+1 *)
    END; (* IF j *)
  ELSE                                          (* jth 1 moves left *)
    A[i-1]:=1; Pos[j]:=i-1;
    IF j=n THEN                                 (* only one 1 moves *)
      A[i]:=0;
    ELSE                                        (* the j+1st 1 displaces the jth 1 *)
      Pos[j+1]:=i; Dir[j+1]:=1;
      IF i=2*j-1 THEN                           (* It used to be adjacent to the jth 1 *)
        A[i+1]:=0
      ELSE                                      (* It used to be as far right as possible *)
        A[2*j+1]:=0;
      END; (* IF i *)
    END; (* IF j *)
  END; (* IF Dir[j] *)
  IF (Pos[j]=2*j-1) OR (Pos[j]=Pos[j-1]+1) THEN
                                                (* the jth 1 can't move further*)
    Ehr[j]:=Ehr[j-1];
    Ehr[j-1]:=j-1;
  END; (* IF *)
END Next;
```

```
BEGIN (* EhrlichTree *)
  WriteString('Generating Dyck-Words by Transpositions.');
  WriteLn;
  LOOP
    WriteLn;
    WriteString('Enter number of pairs, 0 to quit: ');
    ReadCard(n);
    IF n=0 THEN
      EXIT
    END; (* IF n *)
    WriteString('Enter length of prefix of 1s, 0 for variable:
');
    ReadCard(k);
    IF (k>n) THEN
      WriteString('The maximum is '); WriteCard(n,1); WriteLn;
    ELSE
      IF k>0 THEN
        List:=Fixed;
      ELSE (* k=0 *)
        WriteString('Enter 1 to do n=n+1 and k=1, 0 otherwise:
');
        ReadCard(k);
        IF k>0 THEN
          List:=Special;
        ELSE
          List:=Variable;
        END; (* inner IF k *)
      END; (* outer IF k *)
      First(n,k,Done,A,Pos,Dir,Ehr,List);
      WriteLn;
      WHILE NOT Done DO
        FOR i:=1 TO 2*n DO
          WriteCard(A[i],1);
        END; (* FOR *)
        WriteString("    ");
        FOR i:=1 TO n DO
          WriteCard(Pos[i],1);
        END; (* FOR *)
        WriteString("    ");
        FOR i:=1 TO n DO
          WriteCard(Dir[i],1);
        END; (* FOR *)
        WriteString("    ");
        FOR i:=1 TO n DO
          WriteCard(Ehr[i],1);
        END; (* FOR *)
        WriteLn;
        Next(n,k,Done,A,Pos,Dir,Ehr,List);
      END; (* WHILE *)
    END; (* IF k *)
  END; (* LOOP *)
END EhrlichTree.
```