Abstract: We modify the Knuth-Klingsberg Gray code for unrestricted integer compositions to obtain a Gray code for integer compositions each of whose parts is bounded between zero and some positive integer. We also generalize Ehrlich's method for loop-free sequencing to implement this Gray code in $O(1)$ worst-case time per composition. The $(n-1)$-part compositions of $r$ whose $i$th part is bounded by $n-i$ are the inversion vectors of the permutations of $\{1,\ldots,n\}$ with $r$ inversions; we thus obtain a Gray code and a loop-free sequencing algorithm for this set of permutations.

0. Introduction

An $n$-part composition of a non-negative integer $r$ is an ordered $n$-tuple $(g_1, \ldots, g_n)$ of non-negative integers whose sum is $r$. Such a composition is said to be $(m_1, \ldots, m_n)$-bounded for an $n$-tuple $(m_1, \ldots, m_n)$ of positive integers if $0 \leq g_i \leq m_i$ for all $i$. Bousquet [Bous] used bounded integer compositions to count planar maps with two faces as a function of the degrees of the vertices. Here we apply them to permutations with a fixed number of inversions. The inversion vector of a permutation $(p_1, \ldots, p_n)$ is the ordered $(n-1)$-tuple $(g_1, \ldots, g_{n-1})$, where $g_i$ is the number of elements $p_j$ such that $j > i$ but $p_j < p_i$. Evidently $0 \leq g_i \leq n-i$ for all $i$, so that the inversion tables for all the permutations of $\{1, \ldots, n\}$ with $r$ inversions are the $(n-1, \ldots, 2, 1)$-bounded compositions of $r$. In Section 1 we discuss counting, ranking and lexicographical-order generation of bounded integer compositions. In particular, we show that if the $(n-1, \ldots, 2, 1)$-bounded compositions of $r$ are generated in inverse lexicographical order, then the corresponding permutations can be sequenced - transformed into their successors - in $O(n)$ worst-case time apiece. We also present a non-recursive formula for the number $M(n,r)$ of permutations of $\{1, \ldots, n\}$ with $r$ inversions and a practical algorithm for computing a single value of $M(n,r)$.

A Gray code is a family of lists of words such that in each list the number of letters by which each word differs from its successor is bounded independently of the length of the word. A recursive description of a Gray code for unbounded integer compositions was suggested by Knuth [Wi] and a non-recursive description was given by Klingsberg [KL]. Each composition of $r$ is obtained from its predecessor by raising one number by 1 and lowering another number by 1. When only the compositions which lie within a given $n$-tuple of bounds are selected from the list of unbounded compositions of $r$, each bounded composition is still obtained from its predecessor by raising one number by 1 and lowering another number by 1. This Gray code, like Klingsberg's, has the property that all the words in a list with a common suffix form an interval of consecutive words (we call such a list suffix-partitioned); so we can use Chase's method [Ch] to obtain a non-recursive description of a Gray code for bounded integer compositions: for each
suffix \((g_{i+1}, \ldots, g_n)\) we give the sequence of distinct values assumed by \(g_i\) in the interval in which that suffix is constant. If the \((n-1, \ldots, 2, 1)\)-bounded compositions of \(r\) are put into this order, then the corresponding permutations can be updated in \(O(1)\) worst-case time apiece and each permutation is obtained from its predecessor by either two independent two-element transpositions or a three-element rotation. These results are presented in Section 2.

A loop-free algorithm is one that runs in \(O(1)\) worst-case time. The obstacle to loop-free sequencing of a word in a Gray code is not changing the letters but finding the pivot - the index of the rightmost letter that must change to sequence the word. Ehrlich [Eh] solved this problem for several classical Gray codes. In [Wa1] and [Wa2] we showed that his method works for any suffix-partitioned word list with the additional property that for each suffix \((g_{i+1}, \ldots, g_n)\), \(g_i\) assumes at least two distinct values in the interval in which the suffix is constant. Many of the Gray codes in the literature, such as the Liu-Tang Gray code for combinations [LT] and the Knuth-Klingsberg Gray code are suffix-partitioned but do not have this additional property. Ad hoc methods can often be used to find the pivot in \(O(1)\) time - for example, in [Wa1] we showed that the Liu-Tang Gray code can be sequenced in \(O(1)\) time and \(O(1)\) extra space because the pivot differs by at most 2 from one combination to the next and that the Knuth-Klingsberg Gray code can be sequenced in \(O(1)\) time by storing the positions of the non-zero elements in a stack. This can also be done with our Gray code but it would require five auxiliary arrays. In Section 3 we generalize Ehrlich’s method so that it works even if there are suffixes \((g_{i+1}, \ldots, g_n)\) for which \(g_i\) assumes a single value in the interval in which the suffix is constant provided that this interval contains only a single word. The Liu-Tang Gray code, the Knuth-Klingsberg Gray code and ours all have this property; so our generalization of Ehrlich’s method works for them all, and in the case of our Gray code, only two auxiliary arrays are needed. In this way we can sequence \(n\)-permutations with \(r\) inversions in \(O(1)\) worst-case time apiece. Computer time-trials indicate that loop-free sequencing actually does save some CPU time.

1. Enumeration, ranking and lexicographical order sequencing

The rank of an item in a list is the number of items that precede it in the list. For a prefix-partitioned word list, the rank of the item \((g_1, \ldots, g_n)\) is

\[
\sum_{i=1}^{n} \sum_{j=0}^{g_j-1} \#(g_1, \ldots, g_{i-1}, j),
\]

where \(\#(g_1, \ldots, g_i, j)\) is the number of items in the list with prefix \((g_1, \ldots, g_i, j)\). In the case of \((m_1, \ldots, m_n)\)-bounded compositions of \(r\), \(\#(g_1, \ldots, g_i, j)\) is just the number of \((m_{i+1}, \ldots, m_n)\)-bounded compositions of \(r-(g_1+\ldots+g_{i-1}+j)\), so that the ranking problem is reduced to enumeration.
The enumerating generating function for \((m_1,\ldots,m_n)\)-bounded compositions is

\[
\prod_{i=1}^{n} \sum_{j=0}^{m_i-1} x^j = \left( \prod_{i=1}^{n} (1 - x^{m_i}) \right) (1 - x)^{-n},
\]

and the number \(\#()\) of \((m_1,\ldots,m_n)\)-bounded compositions of \(r\) is just the coefficient of \(x^r\) in (2), which can be computed from the right side of (2) in \(O(nr)\) time storing \(r\) large integers. The corresponding generating function for the number \(M(n,r)\) of \(n\)-permutations with \(r\) inversions was found by MacMahon [Mac], after whom those numbers were named mahonians:

\[
\prod_{i=1}^{n} \sum_{j=0}^{i-1} x^j = \left( \prod_{i=1}^{n} (1 - x^i) \right) (1 - x)^{-n}.
\]

Kendall [Ken] evaluated tables of values of \(M(n,r)\) by substituting into the recurrence

\[
M(n,r) = \begin{cases} 
1 & \text{if } r = 0, \\
M(n,r-1) + M(n-1,r) & \text{if } 1 \leq r \leq \min(n-1,n(n-1)/4), \\
M(n,r-1) + M(n-1,r) - M(n-1,r-n) & \text{if } n \leq r \leq n(n-1)/4, \\
M(n,n(n-1)/2 - r) & \text{if } n(n-1)/4 < r \leq n(n-1)/2.
\end{cases}
\]

If one wants to compute a single value of \(M(n,r)\) rather than a whole table, then using the right side of (3) has the advantage that the numbers one has to store and compute with before multiplying by \((1-x)^{-n}\) are bounded by \(2^n\) instead of \(n!\).

Further economies can be made by using two formulae proved by Euler [Eul] (a combinatorial proof of the first identity was given by Franklin [Fra]):

\[
\left( \prod_{i=1}^{\infty} (1 - x^i) \right)^k = 1 + \sum_{k=1}^{\infty} (-1)^k \left( x^{k(3k-1)/2} + x^{k(3k+1)/2} \right)
\]

\[
\prod_{i=1}^{n} (1 - x^i) = \left( \prod_{i=1}^{\infty} (1 - x^i) \right) \left( 1 + \frac{x^{n+1}}{1-x} + \frac{x^{2(n+1)}}{(1-x)(1-x^2)} \right).
\]
MahonQuick:=proc(n,r)
local a,b,i,k,Odd,nc2,i1,i2;
a:=table();
b:=table();
nc2:=n*(n-1)/2;
s:=min(r,nc2-r);
if s<0 then RETURN(0) fi;

# expansion of \((1-x)(1-x^2)(1-x^3)\) to \(\infty\) from the Euler-Franklin formula (5)

a[0]:=1;
for i from 1 to r do a[i]:=0 od;
k:=1; Odd:=true; i1:=1; i2:=2;
while i2<=r do
  if Odd then
    a[i1]:=-1; a[i2]:=-1;
  else
    a[i1]:=1; a[i2]:=1;
  fi;
  Odd:=not(Odd);
k:=k+1; i1:=i2+2*k-1; i2:=i1+k;
  od;
if i1<=r then
  if Odd then a[i1]:=-1 else a[i1]:=1 fi;
fi;

# expansion of \((1-x)(1-x^2)(1-x^3)\) \text{...to} \(\infty\) from Euler's formula (6)

for i from 0 to r do b[i]:=a[i] od;
i:=1; k:=n+1;
while k<=r do
  for i1 from i to r-k do a[i1]:=a[i1]+a[i1-i] od;
  for i1 from k to r do b[i1]:=b[i1]+a[i1-k] od;
i:=i+1; k:=k+n+1;
  od;

# m(n,r)=coefficient of x^r in \((1-x)(1-x^2)(1-x^3)\) \text{...to} \(\infty\) \(\text{from Euler's formula (6)}\)

il:=1; i2:=b[r];
for i from 1 to r do
  il:=il*(n+i-1)/i; i2:=i2+il*b[r-i];
od;
RETURN(i2);
end:

Algorithm 1
A Maple procedure for computing the number \(M(n,r)\) of permutations of \(\{1,...,n\}\) with \(r\) inversions using Euler's formula for \((1-x)(1-x^2)(1-x^3)\) \text{...to} \(\infty\) in terms of \((1-x)(1-x^2)(1-x^3)\) \text{...to} \(\infty\) and the Euler-Franklin expansion of \((1-x)(1-x^2)(1-x^3)\) \text{...to} \(\infty\).

Working up to the coefficient of \(x^r\), first substitute into (5), then into (6), and finally multiply by \((1-x)^n\). This takes \(O(r+r^2/n)\) operations, which is asymptotically faster than (4) if \(r = \)
\( o(n^2) \) and asymptotically equivalent to (4) in the worst case, which occurs when \( r = \text{floor}(n(n-1)/4) \) because for fixed \( n \), \( M(n,r) \) is a symmetric unimodal function of \( r \) which reaches its maximum at that value. We programmed this method in Maple (see the source code in Algorithm 1) and compared it to (4) on a machine which allows 1 megabyte of computation space. Using the worst value of \( r \) for each \( n \), (4) could calculate \( M(n,r) \) for \( n \) only as large as 80 without exceeding the memory limits (and took almost three times as long as (5) and (6)), whereas \( (5) \) and \( (6) \) did it for \( n \) as large as 160.

Using (3) and (5), Knuth (see [Kn], page 16) found a non-recursive formula for \( M(n,r) \) which is valid for \( r \leq n \). Using (6) as well, we can generalize this formula so that it works for any \( r \). The coefficient of \( x^i \) in the term \( 1/(1-x)(1-x^2)...(1-x^m) \) in the sum in the right side of (6) is the number \( D(i;1,2,...,m) \) of partitions of \( i \) whose parts do not exceed \( m \), and Colman [Col] showed how an explicit formula for these numbers can be found for any \( i \) and \( m \). We transfer the factor \((1-x)^{-1}\) from each term except 1 in the sum in (6) to the factor \((1-x)^n\) of (3); now the coefficient of this term is \( D(i;2,...,m) = D(i;1,...,m) - D(i-1;1,...,m) \), the number of partitions of \( i \) into parts ranging from 2 to \( m \), whence the following non-recursive formula for \( M(n,r) \):

\[
M(n,r) = \binom{n+r-1}{n-1} + \sum_{k=1}^{f(r)} (-1)^k \left( \binom{n+r-1-k(3k-1)/2}{n-1} + \binom{n+r-1-k(3k+1)/2}{n-1} \right) +
\]

\[
\binom{r-1}{n} + \sum_{k=1}^{f(r-n-1)} (-1)^k \left( \binom{r-1-k(3k-1)/2}{n} + \binom{r-1-k(3k+1)/2}{n} \right) +
\]

\[
\left\lfloor \frac{r}{n+1} \right\rfloor \sum_{i=0}^{r-m(n+1)} \binom{n+r-m(n+1)-i}{n} D(i;2,...,m) +
\]

\[
\sum_{m=2}^{r-m(n+1)} (-1)^k \sum_{k=1}^{f(r-m(n+1))} \binom{r-m(n+1)-k(3k-1)/2}{n} \times \left( \binom{n+r-m(n+1)-i-k(3k-1)/2}{n} + \binom{n+r-m(n+1)-i-k(3k+1)/2}{n} \right)
\]

\[
\text{where } f(r) = \left\lfloor \frac{1+\sqrt{1+24r}}{6} \right\rfloor.
\]

Finding a non-recursive formula from which a single value of \( M(n,r) \) can be found in \( O(nr) \) operations is still, as far as we know, an open problem.

For each prefix \((g_1,...,g_{i-1})\) of an \((m_1,...,m_n)\)-bounded composition \((g_1,...,g_n)\) of \( r, g_n \) must be equal to \( r-(g_1+...+g_{i-1}) \), and for all \( i < n \), the maximum value that \( g_i \) can take is \( \min(m_i,r-(g_1+...+g_{i-1})) \) and the minimum is \( \max(0,r-(g_1+...+g_{i-1})-(m_{i+1}+...+m_n)) \). We call 0 the fixed minimum, \( m_i \) the fixed maximum, \( r-(g_1+...+g_{i-1})-(m_{i+1}+...+m_n) \) the mobile minimum and
The mobile maximum. If \( g_i \) attains its mobile maximum, then \( g_j = 0 \) for all \( j > i \), and if \( g_i \) attains its mobile minimum, then \( g_j = m_j \) for all \( j > i \). In lexicographic order each \( g_i \) rises in steps of 1 from its minimum to its maximum, and in inverse lexicographic order it falls from its maximum to its minimum, in each interval of words in which the prefix \((g_1,...,g_{i-1})\) is constant. Sequencing any word list, including an \((m_1,...,m_n)\)-bounded composition \((g_1,...,g_n)\) of \( r \), in lexicographic or inverse lexicographic order is done the way an odometer flips to the next kilometer: scan the word from right to left to find the pivot - the rightmost letter which is not at its last value (if there is none, then the word is the last one in the list), change \( g_i \) to its next value, and then scan from left to right replacing each \( g_j, j > i \), by its first value. During each of those scans, the bounds can be updated in \( O(1) \) time for each value of the index \((i \text{ or } j)\), so that sequencing is done in \( O(n) \) worst-case time.

We choose inverse lexicographical order for the inversion vectors of permutations of \( \{1,...,n\} \) with \( r \) inversions; the corresponding permutations will also be in inverse lexicographical order. After the pivot \( g_i \) has fallen by 1, the suffix \((g_{i+1},...,g_n)\) will be at its lexicographically largest value: for some \( k \) such that \( i+1 \leq k \leq n \), \( g_j = m_j \) for all \( j, i < j < k \), and \( g_j = 0 \) for all \( k > j \). In the corresponding permutation the following changes occur: \( p_i \) changes to the largest number smaller than \( p_i \) in \( S(i) = \{1,...,n\} - \{p_1,...,p_{i-1}\} \); for all \( j \) such that \( i < j < k \), \( p_j \) is the largest number in \( S(j) \); \( p_k \) is the \( g_{k+1} \)st smallest number in \( S(k) \); and for all \( j > k \), \( p_j \) is the smallest number in \( S(j) \).

We store an auxiliary bit vector to represent the set \( S(j) \), which is initialized at \((0,...,0)\) and is updated in \( O(1) \) time for each value of the index during the two scans. The next value of \( p_i \), the set of values of \( p_j \) such that \( i < j < k \), the value of \( p_k \) and the set of values of \( p_j \) such that \( k < j \leq n \) can each be computed in \( O(n) \) worst-case time; so the same time bound holds for updating the entire permutation.

2. The Gray Code

Klingsberg's non-recursive description of Knuth's Gray code for unrestricted length-\( n \) compositions of \( r \) is essentially the following. The first composition is the lexicographically largest one \((r,0,...,0)\). For each interval of words in which the suffix \((g_{i+1},...,g_n)\) is fixed, \( g_i \) is bounded between 0 and \( r-(g_{i+1}+...+g_n) \), except that \( g_1 \) must be \( r-(g_2+...+g_n) \), and it rises in steps of 1 from its minimum to its maximum if \((g_{i+1}+...+g_n)\) is even and falls in steps of 1 from its maximum to its minimum otherwise. Changing the prefix \((g_1,...,g_{i-1})\) from its old last value to its new first value entails changing either \( g_1 \) or \( g_{i-1} \) by 1. Klingsberg showed that one could determine which of these two numbers to change by scanning from left to right to find the second leftmost non-zero element; this loop can be avoided by storing the positions of all the non-zero elements on a stack [Wa1].
We modify that Gray code so that it generates the \((m_1,...,m_n)\) bounded compositions \((g_1,...,g_n)\) of \(r\). The first composition is the lexicographically largest one. The \textit{sequencing rule} is the same as Klingsberg’s except for the bounds: the maximum value for \(g_i\) is now \(
abla \min(m_j,r-(g_{i+1}+...+g_n))\) and the minimum value is now \(
abla \max(0,r-(g_{i+1}+...+g_n)-(m_1+...+m_{i-1}))\). This assures that the list of words generated will be the sublist of the Knuth-Klingsberg Gray code consisting of those compositions which lie within the bounds.

The sequencing algorithm follows the same pattern as for lexicographical order. We scan from left to right to find the pivot (the smallest \(i\) such that \(g_i\) is not at its last value according to the sequencing rule). If no such \(i\) exists, then the current word is the last one. Otherwise, we change \(g_i\) to its next value (raising or lowering it by 1 according to the parity of \(g_{i+1}+...+g_n\)), and then scan from right to left beginning with \(i-1\) to find the largest \(j < i\) such that \(g_j\) is not at its first value and change \(g_j\) to its first value, which turns out, as we shall see below, to either lower or raise it by 1. As in the case of lexicographical order, this process takes \(O(n)\) worst-case time.

To show that this sequencing process generates the next word according to the sequencing rule, it suffices to show that for all \(k < j\), \(g_k\) is at its first value according to the sequencing rule. We recall that for \(i\) to be the pivot, \(g_k\) must be at its last value for all \(k < i\). When \(g_i\) rises or falls by 1, the parity of \((g_i+...+g_n)\) changes, so that for all \(k < i\), the numbers \(g_k\), which had risen to their maximum value must now fall and numbers which had fallen to their minimum value must now rise. The maximum value of \(g_k\) is either fixed at \(m_k\) or it moves with \(r-(g_{k+1}+...+g_n)\), falling by 1 if \(g_i\) rises by 1 and vice versa. Similarly, the minimum value of \(g_k\) is either fixed at 0 or it moves with \(r-(g_{k+1}+...+g_n)-(m_1+...+m_{k-1})\), falling by 1 if \(g_i\) rises by 1 and vice versa. If \(g_k\) is at a fixed maximum or minimum, then it is now at its first value. But this cannot be the case for all \(k < i\): if it is true for all \(k, 2 \leq k < i\), then \(g_{1}\), which was equal to the old value of \(r-(g_2+...+g_n)\), must fall by 1 if \(g_i\) rises by 1 and vice versa. Let \(j\) be the largest number less than \(i\) such that \(g_j\) is at a mobile maximum or minimum (we call \(j\) the secondary pivot). Then to attain its first value it must move with the maximum or minimum, rising by 1 if \(g_i\) falls by 1 and vice versa. If \(g_j\) is at a mobile maximum, then \(g_j = 0\) for all \(k < j\), and if \(g_j\) is at a mobile minimum, then \(g_k = m_k\) for all \(k < j\). But then there is only one possible value for each \(g_k\), \(k < j\): if \(g_k = 0\), then it cannot fall, and it cannot rise without pushing some number to its left below 0, and if \(g_k = m_k\), then it cannot rise, and it cannot fall without pulling some number to its left above its bound. Each \(g_k, k < j\), is at therefore its first value, so that the algorithm described in the previous paragraph does in fact generate the next word according to the sequencing rule. And since the sequencing rule makes each \(g_i\) vary between the same limits as with colexicographic order (lexicographic order with left-right reversal), every \((m_1,...,m_n)\)-bounded composition \((g_1,...,g_n)\) of \(r\) will be generated exactly once.
Table 1
Gray code for (4,3,2,1)-bounded compositions of 5, the corresponding permutations of \{1,...,5\} with 5 inversions and their inverses. The pivotal elements are in italics.

<table>
<thead>
<tr>
<th>composition</th>
<th>permutation</th>
<th>inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 1 0 0</td>
<td>5 2 1 3 4</td>
<td>3 2 4 5 1</td>
</tr>
<tr>
<td>3 2 0 0</td>
<td>4 3 1 2 5</td>
<td>3 4 2 1 5</td>
</tr>
<tr>
<td>2 3 0 0</td>
<td>3 5 1 2 4</td>
<td>3 4 1 5 2</td>
</tr>
<tr>
<td>1 3 1 0</td>
<td>2 5 3 1 4</td>
<td>4 1 3 5 2</td>
</tr>
<tr>
<td>2 2 1 0</td>
<td>3 4 2 1 5</td>
<td>4 3 1 2 5</td>
</tr>
<tr>
<td>3 1 1 0</td>
<td>4 2 3 1 5</td>
<td>4 2 3 1 5</td>
</tr>
<tr>
<td>4 0 1 0</td>
<td>5 1 3 2 4</td>
<td>2 4 3 5 1</td>
</tr>
<tr>
<td>3 0 2 0</td>
<td>4 1 5 2 3</td>
<td>2 4 5 1 3</td>
</tr>
<tr>
<td>2 1 2 0</td>
<td>3 2 5 1 4</td>
<td>4 2 1 5 3</td>
</tr>
<tr>
<td>1 2 2 0</td>
<td>2 4 5 1 3</td>
<td>4 1 5 2 3</td>
</tr>
<tr>
<td>0 3 2 0</td>
<td>1 5 4 2 3</td>
<td>1 4 5 3 2</td>
</tr>
<tr>
<td>0 2 2 1</td>
<td>1 4 5 3 2</td>
<td>1 5 4 2 3</td>
</tr>
<tr>
<td>1 1 2 1</td>
<td>2 3 5 4 1</td>
<td>5 1 2 4 3</td>
</tr>
<tr>
<td>2 0 2 1</td>
<td>3 1 5 4 2</td>
<td>2 5 1 4 3</td>
</tr>
<tr>
<td>3 0 1 1</td>
<td>4 1 3 5 2</td>
<td>2 5 3 1 4</td>
</tr>
<tr>
<td>2 1 1 1</td>
<td>3 2 4 5 1</td>
<td>5 2 1 3 4</td>
</tr>
<tr>
<td>1 2 1 1</td>
<td>2 4 3 5 1</td>
<td>5 1 3 2 4</td>
</tr>
<tr>
<td>0 3 1 1</td>
<td>1 5 3 4 2</td>
<td>1 5 3 4 2</td>
</tr>
<tr>
<td>1 3 0 1</td>
<td>2 5 1 4 3</td>
<td>3 1 5 4 2</td>
</tr>
<tr>
<td>2 2 0 1</td>
<td>3 4 1 5 2</td>
<td>3 5 1 2 4</td>
</tr>
<tr>
<td>3 1 0 1</td>
<td>4 2 1 5 3</td>
<td>3 2 5 1 4</td>
</tr>
<tr>
<td>4 0 0 1</td>
<td>5 1 2 4 3</td>
<td>2 3 5 4 1</td>
</tr>
</tbody>
</table>

We choose to stick with Klingsberg's suffix-partitioned list rather than apply left-right reversal to make the list prefix-partitioned because when the bounded compositions of \( r \) are the inversion vectors of permutations with \( r \) inversions the permutations are easier to update. When one element \( g_k \) of an inversion vector rises or falls by 1, a transposition of two elements is induced in the corresponding permutation. One of these two elements is \( p_i \) and the other one is the element closest in value to \( p_i \), larger if \( g_i \) rises and smaller if it falls, in the set \( S(i) \). By using a suffix-partitioned list, we ensure that for the secondary pivot \( j \), for all \( k < j \) either \( g_k = 0 \) or else \( g_k = m_k \), so that \( S(j) \) is an interval and the element with which \( p_i \) swaps is either \( p_j + 1 \) or \( p_j - 1 \). For the pivot \( i \), \( g_j \) will not necessarily be either 0 or \( m_i \), in which case the set of values with which \( p_i \) is allowed to swap will fail to be an interval because of a single missing number: \( p_j \). If \( |p_i - p_j| \leq 2 \) and the smaller of these two values is supposed to rise, then performing the transpositions in the wrong order would force \( p_i \) to swap with \( p_j \), which is not among the values with which \( p_i \) is allowed to swap, so that a 3-element rotation must be performed; otherwise, the two transpositions are independent. Table 1 illustrates the various cases that can arise: the first transformation consists of two independent transpositions, the second one is a 3-element rotation with \( |p_i - p_j| = 2 \) and the third is a 3-element rotation with \( |p_i - p_j| = 1 \). Once the two pivots are known, the permutation can be updated in \( O(1) \) time provided that the inverse permutation is also
stored so that the positions of the elements $p_i \pm 1$ and $p_j \pm 1$ can be determined quickly. A pseudo-code for the updating algorithm is given as Algorithm 2. The first permutation can be computed from the first composition in $O(n)$ worst-case time: the first composition is of the form $(n-1,n-2,...,n-i+1,g_i > 0,0,...,0)$ and the corresponding permutation is $(n,n-1,...,n-i+2,g_i+1,1,2,...,g_i,g_i+2,...,n-i+1)$.

**Algorithm 2**

Finding the next permutation in Gray code order and its inverse in $O(1)$ time given the two pivots and the direction in which the first one changes.
3. Loop-free implementation

To find the pivot $i$ the secondary pivot $j$ and the direction in which $p_i$ changes, a case-by-case analysis is necessary. In what follows, $d = p_{i+1} + ... + p_n$ mod 2, so that $d = 1$ means that $p_i$ falls by 1 and $p_j$ rises by 1, and $d = 0$ means vice versa. The individual cases are illustrated in Table 2. We recall that $j$ is the largest index $< i$ such that after $p_i$ changes to its next value, $p_j$ is not at its first value, and that in scanning left from $p_i$ "first value" alternates between "maximum" and "minimum" every time an odd number is passed.

(1): $0 < p_1 < m_1$. The maximum value for every other number is its fixed bound and the minimum value is 0, and every number between $p_1$ and $p_i$ is either at its fixed maximum or minimum, both before and after $p_i$ changes, and therefore at its first value, which implies that $j = 1$. There are two subcases (1.1) and (1.2) in which $i > 2$.

(1.1): $p_2 = 0$ and $p_3 + ... + p_n$ is odd. This subcase is further divided into two subsubcases.

(1.1.1): If $p_i$ and its suffix have opposite parity, then every number encountered while scanning left from $p_i$ is at its minimum value until an odd number is passed or until the index drops to 1, which implies that $p_{i-1} = ... = p_2 = 0$. See Case 1 ($d = 1$) and Case 2 ($d = 0$) in Table 2.

(1.1.2): If $p_i$ and its suffix have the same parity, then every number met while scanning left from $p_i$ is at its maximum value until an odd number is passed, and every number to its left is at its minimum value. This implies that there is some element $p_k$ between $p_2$ and $p_i$ which is at an odd bound, every number between $p_k$ and $p_i$ is at an even bound and every number between $p_1$ and $p_k$ is 0. See Case 3 ($d = 1$) and Case 4 ($d = 0$).

(1.2): $p_2 = m_2$ and $p_3 + ... + p_n$ is even. Then $p_i$ and its suffix must have the same parity, because otherwise every number to its left would have to be at its minimum value, which is 0, including $p_2$, which is $m_2 > 0$. Instead, every number between $p_2$ and $p_i$ is at its maximum value, which is an even bound because if an odd bound were reached, every number to its left would be 0 including $p_2$. See Case 5 ($d = 1$) and Case 6 ($d = 0$).

(1.3): In all other cases, $p_2$ is not at its last value; so $i = 2$. See case 7 ($d = 1$) and case 8 ($d = 0$).

(2): $p_1 = ... = p_{f-1} = 0$ and $p_f > 0$ for some $f > 1$. The minimum value for every other number is 0, and the maximum value of every other number except possibly $p_f$ is its fixed bound; so by the same argument used for (1) $j \leq f$. 
(2.1): Suppose that every number to the left of \( p_i \) is 0 \((f = i)\). Then no number to the left of \( p_i \) can fall; so \( p_i \) cannot rise, and so \( d \) must be 1. If \( p_i \) is odd, then when it falls the first (maximum) value of \( p_{i-1} \) rises from 0 to 1; so \( j = i-1 \) (see Case 9). If \( p_i \) is even, then when its falls the first (minimum) value of every number from \( p_{i-1} \) to \( p_2 \) stays at 0; so \( j = 1 \) (see Case 10).

(2.2): Suppose that not every number to the left of \( p_i \) is 0 \((f < i)\). Then \( p_i \) and its suffix must have the same parity: otherwise every number to its left would be at its minimum value, which is 0, including \( p_f \). Instead, every number between \( p_f \) and \( p_i \) is at its maximum value - an even bound.

(2.2.1): If \( p_i \) is odd it must fall because its suffix is odd too; so \( p_j \) must rise. If \( p_f < m_f \), then it can rise, and so \( j = f \) (see Case 11). Suppose that \( p_f = m_f \). If \( p_i \) is odd, then after \( p_i \) falls, the suffix of \( p_{f-1} \) will be even; so its first value is its minimum 0, and the same will be true for all the numbers between \( p_i \) and \( p_1 \), and so \( j = 1 \) (Case 12). If \( p_i \) is even, then after \( p_i \) falls, the suffix of \( p_{f-1} \) will be odd; so its first value is its maximum, which rises to 1, and so \( j = f-1 \) (Case 13).

(2.2.2): If \( p_i \) is even it must rise because its suffix is even too; so \( p_j \) must fall. Since \( p_f > 0 \), it can fall, and so \( j = f \) (Case 14).

(3): \( p_1 = m_1, \ldots, p_{f-1} = m_{f-1} \) and \( p_f < m_f \) for some \( f > 1 \). The maximum value for every other number is its fixed bound, and the minimum value of every other number except possibly \( p_f \) is 0. Again, \( j \leq f \).

(3.1): Suppose that every number to the left of \( p_i \) is at its bound \((f = i)\). Then no number to the left of \( p_i \) can rise; so \( p_i \) cannot fall, and so \( d \) must be 0. If \( p_i \) is odd, then when it rises the first (minimum) value of \( p_{i-1} \) falls, and \( p_{i-1} \) can fall; so \( j = i-1 \) (Case 15). If \( p_i \) is even, then when it rises every number met while scanning left from \( p_i \) must be set to its maximum value until an odd number is passed or the index drops to 1. If one of the numbers between \( p_1 \) and \( p_i \) is odd, let \( k \) be the index of the rightmost such number; then \( j = k-1 \) (Case 16). Otherwise \( j = 1 \) (Case 17).

(3.2): Suppose that not every number to the left of \( p_i \) is at its bound \((f < i)\).

(3.2.1): Suppose that \( p_i \) and its suffix have opposite parity. Then every number between \( p_f \) and \( p_i \) must be at its minimum value, which is 0. If \( p_i \) is odd (so that it rises and \( p_j \) falls) and \( p_f = 0 \) so that it can't fall, then \( j = f-1 \), because \( p_{f-1} \) can fall (Case 18). Otherwise \( j = f \) whether \( p_i \) is odd and \( p_f > 0 \) (Case 19) or \( p_i \) is even (Case 20).

(3.2.2): Suppose that \( p_i \) and its suffix have the same parity. There must be some odd number between \( p_f \) and \( p_i \); otherwise every number to the left of \( p_i \) would be at its bound. Let \( p_k \) be the rightmost such number. Then every number between \( p_k \) and \( p_i \) is at its maximum, an even
bound, $p_k$ is at its maximum too, an odd bound, and every number between $p_f$ and $p_k$ is at its minimum, which is 0. If $p_i$ is odd it must fall, and since $p_f < m_j$ it can rise; so $j = f$ (Case 21). If $p_i$ is odd it must rise. If $p_f > 0$ it can fall; so $j = f$ (Case 22); otherwise $j = f-1$, because $p_{f-1}$ can fall (Case 23).

This exhausts all the cases.

Case 1: \[ 0 < g < m \uparrow, 0, 0*, \text{even} \downarrow > 0 \downarrow, \text{odd suffix} \]
Case 2: \[ 0 < g < m \downarrow, 0, 0*, \text{odd} \uparrow < m \uparrow, \text{even suffix} \]
Case 3: \[ 0 < g < m \uparrow, 0, 0*, \text{odd} m, (\text{even} m)*, \text{odd} \downarrow \downarrow, \text{odd suffix} \]
Case 4: \[ 0 < g < m \downarrow, 0, 0*, \text{odd} m, (\text{even} m)*, \text{even} \downarrow < m \uparrow, \text{even suffix} \]
Case 5: \[ 0 < g < m \uparrow, m, (\text{even} m)*, \text{odd} \downarrow \downarrow, \text{odd suffix} \]
Case 6: \[ 0 < g < m \downarrow, m, (\text{even} m)*, \text{even} \downarrow < m \uparrow, \text{even suffix} \]
Case 7: \[ 0 < g < m \uparrow, > 0 \downarrow, \text{odd suffix} \]
Case 8: \[ 0 < g < m \downarrow, < m \uparrow, \text{odd suffix} \]
Case 9: \[ 0*, 0\uparrow, \text{odd} \downarrow \downarrow, \text{odd suffix} \]
Case 10: \[ 0 \uparrow, 0*, \text{even} \downarrow > 0 \downarrow, \text{odd suffix} \]
Case 11: \[ 0*, 0 < g < m \uparrow, (\text{even} m)*, \text{odd} \downarrow \downarrow, \text{odd suffix} \]
Case 12: \[ 0 \uparrow, 0*, \text{odd} m, (\text{even} m)*, \text{odd} \downarrow \downarrow, \text{odd suffix} \]
Case 13: \[ 0*, 0 \uparrow, \text{even} m, (\text{even} m)*, \text{odd} \downarrow \downarrow, \text{odd suffix} \]
Case 14: \[ 0, 0*, \text{odd} m, (\text{even} m)*, \text{even} \downarrow < m \uparrow, \text{even suffix} \]
Case 15: \[ m*, m \downarrow, \text{odd} < m \uparrow, \text{even suffix} \]
Case 16: \[ m*, m \downarrow, \text{odd} m, (\text{even} m)*, \text{even} \downarrow < m \uparrow, \text{even suffix} \]
Case 17: \[ m \downarrow, (\text{even} m)*, \text{even} \downarrow < m \uparrow, \text{even suffix} \]
Case 18: \[ m*, m \downarrow, 0*, \text{odd} < m \uparrow, \text{even suffix} \]
Case 19: \[ m, m*, 0 < g < m \downarrow, \text{odd} \downarrow < m \uparrow, \text{even suffix} \]
Case 20: \[ m, m*, g < m \uparrow, 0*, \text{even} \downarrow > 0 \downarrow, \text{odd suffix} \]
Case 21: \[ m, m*, g < m \uparrow, 0*, \text{odd} m, (\text{even} m)*, \text{odd} \downarrow \downarrow, \text{odd suffix} \]
Case 22: \[ m, m*, 0 < g < m \downarrow, 0*, \text{odd} m, (\text{even} m)*, \text{even} \downarrow < m \uparrow, \text{even suffix} \]
Case 23: \[ m*, m \downarrow, 0*, \text{odd} m, (\text{even} m)*, \text{even} \downarrow < m \uparrow, \text{even suffix} \]

**Table 2**

The possible transformations of a bounded integer composition.
A number which is equal to its bound is represented by $m$.
An asterisk represents 0 or more repetitions of the same type of number.
The two pivotal elements have arrows to their right indicating the direction of the change.

It would be possible to implement the sequencing algorithm so that it runs in \(O(1)\) worst-case time by storing the maximal intervals of zeros or numbers at their bounds on a stack. This would require, for each such interval, its left and right indices and an integer to indicate whether it consisted of zeros or, if not, had an even sum, for a total of three size-$n$ variable arrays. In addition we would need two size-$n$ fixed arrays: for each index $i$ we would need to know the index of the nearest odd bound to its left and to its right. At most the top three items on the stack would have to be accessed to find $i$, $j$ and $d$ (the bound of three is attained in Cases 21-23 of
Table 2). But we did not do so because a more space-efficient implementation can be found by generalizing Ehrlich's method for loop-free pivot-finding.

Ehrlich's method cannot be used as it stands, because it works only if for each suffix \((g_{i+1},\ldots,g_n)\), \(g_i\) assumes at least two distinct values in the interval in which the suffix is constant, whereas if \(g_{i+1}\) is at its mobile maximum or minimum, then \(g_i\) can assume only one value (0 or \(m_i\)) and in fact that interval consists of a single word. We generalize Ehrlich's method so that it works for this case too, provided that one can update in \(O(1)\) worst-case time the smallest index \(q\) such that \(g_q\) can assume at least two distinct values. The pseudo-code is listed as Algorithm 3, and a correctness proof is included in the comments.

Once we know the pivot \(i\), we can determine which of the 23 cases applies to the current bounded composition, find \(j\) and \(d\) and update \(g_i, g_j\) and \(f\) (defined as 1 + the length of a prefix of zeros or of numbers equal to their bounds, or 1 if neither such prefix exists) in \(O(1)\) worst-case time. The variable \(q\) of Algorithm 3 is equal to \(f\) if \(f > 1\); otherwise \(q = 2\). This follows from the observation that every number in a prefix of zeros or of numbers equal to their bounds can neither rise nor fall unless some number to its right moves, whereas the number following a maximal prefix of zeros can fall and the number following a maximal prefix of numbers equal to their bounds can rise, and if \(0 < g_1 < m_1\) then \(g_1\) is determined by its suffix but \(g_2\) is not. Algorithm 4 shows the entire updating procedure. It requires only two auxiliary arrays: the variable array \(e\) from Algorithm 3 and a fixed array \(L\), where \(L[i]\) is the largest index \(j < i\) such that \(m[j]\) is odd, or 0 if no such index exists. We recall that the initial composition is the lexicographically largest one; the initial value of the array \(e\) (in terms of \(q\) defined above) and of the Boolean variable Done are given in Algorithm 3. All the arrays including \(L\) can be initialized in \(O(n)\) time.

Algorithms 2 and 4, as well as inverse lexicographical order generation and Gray code generation without loop-free sequencing, were programmed in C and tested (source codes are available from the author on request). When the 25598185 permutations of \(\{1,\ldots,12\}\) with 33 inversions were generated, loop-free sequencing ran 27% faster than Gray code generation without loop-free sequencing and 19% faster than inverse lexicographical order generation. It would appear that loop-free sequencing affords some speed-up if it is sufficiently optimised.

We note that Ruskey [Ru] proved the existence of a different Gray code for bounded integer compositions. The problem of implementing loop-free sequencing of this Gray code is still open.

\(\{q\) is the smallest index such that \(g[q]\) assumes at least two values, determined by the sequencing rule, in the interval of words in which the current suffix of \(g[q]\) is constant\}
**initial value:** Done is **false**, $e[1]$ = the initial value of $q$ and $e[i]=i$ for $i$ from 2 to $n+1$;

{**invariant:** If $(g[i],...,g[j-1])$ is a $z$-word - that is, a maximal subword of letters, each at its last value - for some $j>i$, then $e[i]=j$; otherwise $e[i]=i$.}

{The invariant holds initially because each $g[i]$ is at its first value, which is equal to its last value for all $i\leq q$ but not for $q$. Assume it holds before the updating algorithm is executed.}
Procedure Update(n,q:integer; g:array[1..n]; e:array[1..n+1]; Done:boolean)
  i:=e[1];
  if i>n then {Each g[j] is at its last value; so the current word is the last one.}
    Done:=true; return
  end if;
  {Otherwise i is the pivot - the smallest index such that g[i] is not at its last value.}
  g[i]:=its next value;
  for j:=i-1 downto 1 do g[i]:=its first value end for; {O(1) changes for a Gray code}
  update q;
  e[1]:=q; {Since g[i] changed, q≤i. For each j<q, g[j] is at its first value = its last value.}
  if g[i] is at its last value then
    {If e[i+1]=i+1, then g[i+1] is not at at last value, because g[i] wasn't at its last value until now.
      Otherwise, i+1 and e[i+1]-1 are the smallest and largest indices of a z-word.}
    if i=q then
      {If e[i+1]=i+1, the z-word beginning at index 1 which used to end at index i-1 (or not exist, if i=1) now ends at index i; otherwise it merges with the z-word beginning at index i+1 and ending at index e[i+1]-1, and in either case it ends at index e[i+1]-1.}
      e[1]:=e[i+1]
    else
      {g[i] now begins a z-word. If e[i+1]=i+1, that z-word ends at index i. Otherwise, it prolongs to the left the subword beginning at index i+1 and ending at index e[i+1]-1. In either case, it ends at index e[i+1]-1.}
      e[i]:=e[i+1]
    end if;
    {The invariant holds for every index from 1 to i}
    e[i+1]:=i+1;
    {Since g[i] is at its last value, g[i+1] cannot begin a z-word; so the invariant now holds for i+1 too, and since no change is made beyond i+1, it holds everywhere.}
  end if
  {If g[i] is not at its last value, then either q=1 and g[1] does not begin a subword, or else q>1 and the subword beginning at index 1 ends at index q-1. In either case, the invariant holds for every index up to q, and since no change is made beyond q, it holds everywhere.}
end Update. {In all cases, if the invariant holds before execution, then it holds afterwards.}

Algorithm 3

Generalization of Ehrlich's $O(1)$-time pivot-finding algorithm to wordlists in which all the words with a common suffix form an interval of consecutive words in which the letter preceeding the suffix may take a single value if the interval consists of a single word.
Procedure NextComp(n,r,i,j,d; integer; g,m:array[1..n]; e:array[1..n+1]; Done:boolean)
    {The array g is the length-n composition of r bounded by the array m,
    i is the pivot, j is the secondary pivot, Done means g is the last composition,
    d=1 means g[i] goes down and g[j] goes up and d=0 means vice versa,
    f is 1 + the length of a prefix of zeros or elements of g at their bounds.}
    i:=e[1];
    if i>n then Done:=true; return; end if;
    if f=1 then \{0<g[1]<m[1]} \{end of all the cases; now e gets updated\}
        Cases_1-8(n,r,f,i,j,d,g,m)
    else if g[1]=0 then
        Cases_9-14(n,r,f,i,j,d,g,m)
    else \{g[1]=m[1]} \{end if; \{This is the variable q of Algorithm 3\}
        Cases_15-23(n,r,f,i,j,d,g,m)
    end if;
    if f=1 then e[1]:=2 else e[1]:=f end if;
    if g[i]=m[i] or g[i]=0 or f=i then \{g[i] is at its last value, a mobile limit if f=i\}
        if i=e[1] then e[1]:=e[i+1] else e[i]:=e[i+1] end if;
        e[i+1]:=i+1
    end if;
end NextComp.

Procedure Cases_1-8(n,r,f,i,j,d; integer; g,m:array[1..n])
    \{ f=1 \}
    j:=1;
    if i>2 then \{ either g[2]=0 or g[2]=m[2] \}
        Cases_1-6(n,r,f,i,j,d,g,m)
    else \{ 0<g[2]<m[2] \}
        Cases_7-8(n,r,f,i,j,d,g,m)
    end if;
end Cases_1-8.

Procedure Cases_1-6(n,r,f,i,j,d; integer; g,m:array[1..n])
    \{ f=1 and i>2 \}
    if g[2]=0 then
        Cases_1-4(n,r,f,i,j,d,g,m)
    else \{g[2]=m[2]\}
        Cases_5-6(n,r,f,i,j,d,g,m)
    end if;
end Cases_1-6.

Procedure Cases_1-4(n,r,f,i,j,d; integer; g,m:array[1..n])
    \{ f=1 and i>2 and g[2]=0 \}
    if g[i-1]=0 then
        Cases_1-2(n,r,f,i,j,d,g,m)
    else \{g[i-1]=m[i-1]\}
        Cases_3-4(n,r,f,i,j,d,g,m)
    end if;
end Cases_1-4.
Procedure Cases_1-2(n,r,f,i,j,d,: integer; g,m:array[1..n])
{ f=1 and i>2 and g[2]=0 and g[i-1]=0 }
    if g[i] is odd then {Case 2}
        d:=0; inc(g[i]); dec(g[1]);
        if g[1]=0 then f:=i; end if
    else {Case 1}
        d:=1; dec(g[i]); inc(g[1]);
        if g[1]=m[1] then f:=2 end if
    end if
end Cases_1-2.

Procedure Cases_3-4(n,r,f,i,j,d,: integer; g,m:array[1..n])
{ f=1 and i>2 and g[2]=0 and g[i-1]=m[i-1] }
    if g[i] is odd then {Case 3}
        d:=1; dec(g[i]); inc(g[1]);
        if g[1]=m[1] then f:=2 end if
    else {Case 4}
        d:=0; inc(g[i]); dec(g[1]);
        if g[1]=0 then f:=L[i] end if
    end if
end Cases_3-4.

Procedure Cases_5-6(n,r,f,i,j,d,: integer; g,m:array[1..n])
{ f=1 and i>2 and g[2]=m[2] }
    if g[i] is odd then {Case 5}
        d:=1; dec(g[i]); inc(g[1]);
        if g[1]=m[1] then f:=i; end if
    else {Case 6}
        d:=0; inc(g[i]); dec(g[1]);
        if g[1]=0 then f:=2; end if
    end if
end Cases_5-6.

Procedure Cases_7-8(n,r,f,i,j,d,: integer; g,m:array[1..n])
{ f=1 and i=2 }
    if r-g[1]-g[2] is odd then {Case 7}
        d:=1; dec(g[2]); inc(g[1]);
        if g[1]=m[1] then f:=2; end if
    else {Case 8}
        d:=0; inc(g[2]); dec(g[1]);
        if g[1]=0 then f:=2; end if
    end if
end Cases_7-8.
Procedure Cases_9-14(n,r,f,i,j,d; integer; g,m:array[1..n])
{ g[1]=0 }
  if i=f then
    Cases_9-10(n,r,f,i,j,d,g,m)
  else
    Cases_11-14(n,r,f,i,j,d,g,m)
  end if
end Cases_9-14.

Procedure Cases_9-10(n,r,f,i,j,d; integer; g,m:array[1..n])
{ g[1]=0 and i=f }
  d:=1;
  if g[i] is odd then {Case 9}
    dec(g[i]); j:=i-1; g[j]:=1;
    if g[1]<m[1] then dec(f) end if
  else {Case 10}
    dec(g[i]); j:=1; g[1]:=1;
    if m[1]=1 then f:=2 else f:=1 end if
  end if
end Cases_9-10.

Procedure Cases_11-14(n,r,f,i,j,d; integer; g,m:array[1..n])
{ g[1]=0 and i>f }
  if g[i] is odd then
    Cases_11-13(n,r,f,i,j,d,g,m)
  else {Case 14}
    d:=0; inc(g[i]); j:=f; dec(g[f]);
    if g[f]=0 then inc(f) end if
  end if
end Cases_11-14.
Procedure Cases_11-13(n,r,f,i,j,d,: integer; g,m:array[1..n])
{ g[1]=0 and i>f and g[i] is even }
⑨:=1; dec(g[i]);
if g[f]<m[f] then {Case 11}
  j:=f; inc(g[f])
else {Case 12 or 13}
  if g[f] is odd then {Case 12}
    j:=1; g[1]:=1;
    if m[1]>1 then
      f:=1
  else
    if f>2 or g[j]<m[j] then f:=2 else f:=i end if
  end if {end of Case 12}
else {Case 13}
  j:=f-1; g[j]:=1;
  if j>1 or m[1]>1 then
    dec(f)
  else
    if g[f]=m[f] then f:=i end if
  end if {end of Case 13}
end if {end of Cases 12 and 13}
end if
end Cases_11-13.

Procedure Cases_15-23(n,r,f,i,j,d,: integer; g,m:array[1..n])
{ g[1]=m[1] }
if i=f then
  Cases_15-17(n,r,f,i,j,d,g,m)
else if g[i-1]<m[i-1] then
  Cases_18-20(n,r,f,i,j,d,g,m)
else
  Cases_21-23(n,r,f,i,j,d,g,m)
end if
end Cases_15-23.

Procedure Cases_15-17(n,r,f,i,j,d,: integer; g,m:array[1..n])
{ g[1]=m[1] and i=f }
⑩:=0;
if g[i] is odd then {Case 15}
  inc(g[i]); j:=i-1; dec(g[j]);
  if g[1]>0 then dec(f) end if
else {Case 16 or 17}
  inc(g[i]);
  j:=L[i]-1;
  if j<1 {Case 17 else Case 16} then j:=1 end if;
  dec(g[j]);
  if g[1]=0 then f:=2 else f:=j end if
end if
end Cases_15-17.
**Procedure Cases_18-20** (n, r, f, i, j, d; integer; g, m: array[1..n])

{ g[1]=m[1] and i>f and g[i-1]<m[i-1] }

if g[i] is odd then {Case 18 or 19}
  d:=0; inc(g[i]);
  if g[f]>0 then {Case 19}
    j:=f; dec(g[j])
  else {Case 18}
    f:=f-1; j:=f; dec(g[j]);
    if g[1]=0 then f:=i end if
  end if
end if
end Cases_18-20.

**Procedure Cases_21-23** (n, r, f, i, j, d; integer; g, m: array[1..n])

{ g[1]=m[1] and i>f and g[i-1]=m[i-1] }

if g[i] is odd then {Case 21}
  d:=1; dec(g[i]); j:=f; inc(g[j]);
  if g[j]=m[j] then
    if g[j+1]>0 then f:=i else inc(f) end if
  end if {end of Case 21}
else {Case 22 or 23}
  d:=0; inc(g[i]);
  if g[f]>0 then {Case 22}
    j:=f; dec(g[j])
  else {Case 23}
    j:=f-1; dec(g[j]);
    if g[1]=0 then f:=L[i] else dec(f) end if
  end if {end of Cases 22 and 23}
end if
end Cases_21-23.

**Algorithm 4**

Loop-free updating of the \((m_1,...,m_n)\)-bounded composition \((g_1,...,g_n)\) of \(r\)
using Ehrlich’s variable auxiliary array \(e\) and the static auxiliary array \(L\).
The two pivots \(i\) and \(j\) and the direction variable \(d\) are returned for use in Algorithm 2.

**Acknowledgment:** I wish to thank Greg Egan for pointing out two errors in the pseudocode of Algorithm 4. I have fixed them in this version.
REFERENCES


[Col]: W.J.A. Colman, *A general method for determining a closed formula for the number of partitions of the integer n into m positive integers for small values of m*, Fibonacci Quarterly 21 (1983), 272-284.


[Ken]: M.G. Kendall, *A New Measure of Rank Correlation*, Biometrika 30 (1938), pp. 81-93.

[Mac]: P.A. MacMahon, *Two applications of general theorems in combinatorial analysis: (1) to the theory of inversions of permutations; (2) to the ascertainment of the numbers of terms in the development of a determinant which has amongst its elements an arbitrary number of zeros*, Proc. London Math. Soc. (2) 15 (1916), pp. 314-321.


[Wa1]: T.R. Walsh, *A simple sequencing and ranking method that works on almost all Gray codes*, Research Report No. 243, Department of Mathematics and Computer Science, University of Quebec in Montreal, April 1995.
