Generating Gray Codes in $O(1)$ Worst-Case Time per Word

Timothy Walsh

Department of Computer Science, UQAM
Box 8888, Station A, Montreal, Quebec, Canada H3C-3P8
walsh.timothy@uqam.ca

Abstract. We give a definition of Gray code that, unlike the standard "minimal change" definition, is satisfied by the word-lists in the literature called "Gray codes" and we give several examples to illustrate the various concepts of minimality. We show that a non-recursive generation algorithm can be obtained for a word-list such that all the words with the same prefix (or, equivalently, suffix) are consecutive and that the Bitner-Ehrlich-Reingold method of generating each word in a time bounded by a constant works under the additional condition that in the interval of words with the same prefix or suffix the next letter assumes at least two values. Finally we generalize this method so that it works under a weaker condition satisfied by almost all the Gray codes in the literature: if the next letter assumes only one value, then the interval contains only one word.

1 Definition of a Gray Code

Gray codes have nothing to do with anything so sinister as the color gray, despite the fact that the first article on the subject [1] was published by the military. They were named after the author of that article, Frank Gray, who ordered the set of length-n *binary strings* (words on a two-letter alphabet, usually {0,1}, which represent subsets of $\{1, 2, ..., n\}$) into a list in which two consecutive strings differ in only one position (see Table 1 below), and probably also after the title: Pulse code communication. Subsequently, other sets of words that represent various combinatorial objects such as combinations or k-subsets (see Table 2 below), permutations (see Table 3 below), integer compositions, set partitions, integer partitions and binary trees (represented by Dyck words - see Table 4 below) were ordered into lists in which two consecutive words differ in only a small number of positions, and these lists were called Gray codes because they share the same "minimal change" property as Gray's pulse code.

A Gray code is sometimes defined as a word-list in which the *Hamming distance* between any two adjacent words in the list (the number of positions in which these two words differ) is minimal. Most of the word-lists in the literature that are called Gray codes do not satisfy that definition or are published without a minimality proof. For example, if a k-subset of $\{1, 2, ..., n\}$ is represented by its members in increasing order, then the Hamming distance between adjacent

C.S. Calude et al. (Eds.): DMTCS 2003, LNCS 2731, pp. 73–88, 2003.

words in the Liu-Tang Gray code [2] is either 1 or 2, which is not minimal because it is only 1 in the Eades-McKay Gray code [3] (see Table 2 below).

In addition, if a new Gray code has the same maximal Hamming distance between any two adjacent words as an already-published one, the authors of the new one can invent new criteria to claim that their Gray code is "more minimal" than the old one. If two letters change (the minimal change for a permutation), then the change can be called more minimal if these letters are closer together - for example, those two letters are always adjacent in the Johnson-Trotter Gray code [4],[5] but not in the Nijenhuis-Wilf one [6] (see Table 3 below).

Table 1. Binary strings in lexicographical (*left*) and Gray code (*right*) order. The letters that change from from one word to the next are underlined

0 0 $\underline{0}$	0 0 $\underline{0}$
0 $\underline{0}$ $\underline{1}$	0 $\underline{0}$ 1
0 1 $\underline{0}$	0 1 $\underline{1}$
$\underline{0}$ $\underline{1}$ $\underline{1}$	$\underline{0}$ 1 0
1 0 $\underline{0}$	1 1 $\underline{0}$
1 $\underline{0}$ $\underline{1}$	1 $\underline{1}$ 1
1 1 $\underline{0}$	1 0 $\underline{1}$
1 1 1	1 0 0

Table 2. The Liu-Tang (*left*), Eades-McKay (*second and third columns*) and Chase-Ruskey (*right*) Gray codes for combinations. The one or two letters that change are underlined

1 2 $\underline{3}$ $\underline{4}$	1 2 3 $\underline{4}$	1 $\underline{2}$	$\underline{2}$ 3
$\underline{1}$ $\underline{2}$ 4 5	1 2 $\underline{3}$ 5	$\underline{1}$ 3	1 $\underline{3}$
$\underline{2}$ 3 4 5	1 $\underline{2}$ 4 5	2 $\underline{3}$	1 $\underline{2}$
1 $\underline{3}$ $\underline{4}$ 5	$\underline{1}$ 3 4 5	$\underline{2}$ 4	$\underline{1}$ 4
1 2 $\underline{3}$ $\underline{5}$	2 3 4 $\underline{5}$	$\underline{1}$ 4	$\underline{2}$ 4
$\underline{1}$ $\underline{2}$ 5 6	$\underline{2}$ 3 4 6	3 $\underline{4}$	3 $\underline{4}$
$\underline{2}$ 3 5 6	1 $\underline{3}$ 4 6	$\underline{3}$ 5	$\underline{3}$ 5
$\underline{1}$ $\underline{3}$ 5 6	1 2 $\underline{4}$ 6	$\underline{2}$ 5	$\underline{1}$ 5
$\underline{3}$ 4 5 6	1 2 $\underline{3}$ 6	$\underline{1}$ 5	$\underline{2}$ 5
$\underline{2}$ 4 5 6	1 $\underline{2}$ 5 6	4 5	4 5
1 $\underline{4}$ $\underline{5}$ 6	$\underline{1}$ 3 5 6		
$\underline{1}$ $\underline{2}$ 4 6	2 $\underline{3}$ 5 6		
$\underline{2}$ 3 4 6	$\underline{2}$ 4 5 6		
1 $\underline{3}$ $\underline{4}$ 6	$\underline{1}$ 4 5 6		
1 2 3 6	3 4 5 6		

If the letters are numbers, then the change can be called more minimal if they change by a smaller amount - for example, the one letter that changes in

the Chase-Ruskey Gray code for combinations [7],[8] does so by either 1 or 2 but can change by an arbitrarily large amount in the Eades-McKay Gray code (see Table 2 above). We note at this point that P. Chase and F. Ruskey discovered almost identical Gray codes independently; in [9] we deciphered these two Gray codes and showed how similar they actually are.

Table 3. The Nijenhuis-Wilf (*left*) and Johnson-Trotter (*right*) Gray codes for permutations. The two letters that swap positions are underlined

1 <u>2</u> <u>3</u>	1 <u>2</u> <u>3</u>
<u>1</u> 3 <u>2</u>	<u>1</u> <u>3</u> 2
2 <u>3</u> <u>1</u>	3 <u>1</u> <u>2</u>
<u>2</u> 1 <u>3</u>	<u>3</u> <u>2</u> 1
3 <u>1</u> <u>2</u>	2 <u>3</u> <u>1</u>
3 2 1	2 1 3

A *Dyck word* is a binary string with the same number of 1s and 0s any prefix of which has at least as many 1s as 0s (Dyck words code binary trees). In the Ruskey-Proskurowski Gray code [10], the Bultena-Ruskey Gray code [11] and the Vajnovszki-Walsh Gray code [9] (see columns 1, 2 and 3, respectively, of Table 4 below) two successive words always differ by the transposition of a single 1 with a single 0, which is the smallest number of letters in which two Dyck words can differ. If, in addition, the letters between the 1 and the 0 that are transposed in passing from one word to the next are separated only by 0s, then the Gray code is called *homogeneous*, and if, in addition, at most one letter separates them, then the Gray code is called *two-close*. The Ruskey-Proskurowski Gray code is not homogeneous. The Bultena-Ruskey Gray code is homogeneous, and so it can be considered more minimal than the Ruskey-Proskurowski one, but it is not two-close, although a larger value of n than the one shown in Table 4 would be necessary to demonstrate this fact. In both cases the 1 and the 0 can be separated by an arbitrarily large number of letters for large n. The Vajnovszki-Walsh Gray code is two-close; so it can be considered even more minimal than the Bultena-Ruskey one.

To protect Gray codes from losing their status as such when a "more minimal" one is discovered, we propose a definition of Gray code that is generous enough to be satisfied by all the word-lists that are normally called Gray codes. Roughly speaking, the change doesn't have to be minimal, only "small" - that is, bounded independently of the word-length. But for this definition to make sense, the word-length itself must be unbounded, which is not the case for a single word-list. We would therefore suggest that a suitable definition of a Gray code is an infinite set of word-lists with unbounded word-length such that the Hamming distance between any two adjacent words in any list is bounded independently of the word-length. Since this journal is Lecture Notes in Computer Science instead of Lecture Notes in Mathematics, the reader may abbreviate

"bounded independently of the word-length" to $O(1)$ and then pose the question: "If the number of changes made in transforming one word to the next on the list is $O(1)$, can the transformation be made in $O(1)$ time even in the worst case?"

Table 4. The Ruskey-Proskurowski (*left*), Bultena-Ruskey (*centre*) and Vajnovszki-Walsh (*right*) Gray codes for Dyck words. The letters that swap places are underlined and the letter that separates them, if there is one, is italicized

1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	1 1 1 1 *0* 0 0 0
1 1 1 0 1 0 0 0	1 1 1 0 1 0 0 0	1 1 1 0 0 1 0 0
1 1 1 0 0 1 0 0	1 1 1 0 0 1 0 0	1 1 1 0 0 *0* 1 0
1 1 1 0 0 0 1 0	1 1 1 *0* 0 0 1 0	1 1 1 0 1 0 0 0
1 1 0 1 0 0 1 0	1 1 0 0 1 0 1 0	1 1 0 1 1 0 0 0
1 1 0 1 0 1 0 0	1 1 0 0 1 1 0 0	1 1 0 1 0 1 0 0
1 1 0 1 *1* 0 0 0	1 1 0 1 0 1 0 0	1 1 0 1 0 0 1 0
1 1 0 0 1 1 0 0	1 1 0 1 0 *0* 1 0	1 1 0 0 1 0 1 0
1 1 0 0 1 0 1 0	1 1 0 1 1 0 0 0	1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0	1 0 1 1 1 *0* 0 0	1 0 1 0 1 1 0 0
1 0 1 0 *1* 1 0 0	1 0 1 1 0 0 1 0	1 0 1 0 1 0 1 0
1 0 1 1 1 0 0 0	1 0 1 1 0 1 0 0	1 0 1 1 0 0 1 0
1 0 1 1 0 1 0 0	1 0 1 0 1 1 0 0	1 0 1 1 0 1 0 0
1 0 1 1 0 0 1 0	1 0 1 0 1 0 1 0	1 0 1 1 1 0 0 0

2 Non-recursive Description of a Word-List in Generalized Lexicographical Order

An algorithm for generating a word-list must be non-recursive if it to have any hope of generating each word in $O(1)$ worst-case time, so that a non-recursive description of the word-list must first be found. Aside from any considerations of computational complexity, there is something intrinsically satisfying about knowing a rule for *sequencing* an object in a list - that is, determining whether it is the last one and, if not, transforming it into the next one - and certainly knowing such a rule makes the list easier to generate by hand.

Such a rule can always be derived if the list satisfies a condition that generalizes lexicographical order. We say that a word-list is *prefix-partitioned* if all the words with the same prefix form an interval of consecutive words in the list. The term *suffix-partitioned* is defined analogously, and a list that is either prefix- or suffix-partitioned is called *genlex*. If a list of length-n words is prefix-partitioned, then for any positive integer $i < n$ the interval of words with the same prefix of length $i - 1$ is partitioned into sub-intervals with the same prefix of length i, so that in this interval the ith letter follows a sequence of values, depending on the prefix, such that all the copies of the same value are consecutive in the

sequence. The subsequence of distinct values assumed by the ith letter is called the *defining sequence* of the prefix because, as we show below, once we know the defining sequence as a function of the prefix we have a sequencing algorithm for the list. Turning to Table 1, the reader can easily see that for binary strings generated in lexicographical order the defining sequence for every prefix is (0,1) and for the original Gray code it is (0,1) for every prefix with an even number of 1s and (1,0) for every prefix with an odd number of 1s. This last observation was published by Chase [7], who also coined the term *graylex order*, which describes a genlex word-list on a totally ordered alphabet in which the defining sequence is always monotone; this concept, in turn, generalizes lexicographical order, which describes a prefix-partitioned list on a totally ordered alphabet in which the defining sequence is always increasing.

In the list of length-7 "words" on the English alphabet, the first word is *aaaaaaa*, the last word is *zzzzzzz* and the successor of *wordzzz* is *woreaaa*. The algorithm for sequencing a word in a prefix-partitioned list generalizes this observation. The first word $(g[1], ..., g[n])$ in the list is found by setting, for i running from 1 to n, each $g[i]$ to the first value in the defining sequence of the prefix $(g[1], ..., g[i-1])$. We call the *pivot* of the word $(g[1], ..., g[n])$ the largest integer i such that $g[i]$ is not equal to the last value in the defining sequence of the prefix $(g[1], ..., g[i-1])$ if there is such an i, and otherwise the pivot is defined to be 0. If the pivot is 0, then the current word is the last word in the list; if the pivot is some positive integer i, then the successor to the current word is found by replacing $g[i]$ by the next value in this defining sequence and then setting each $g[j]$ to the right of $g[i]$ to the first value in the defining sequence of $(g[1], ..., g[j-1])$ if it is not already at this value (a bounded number of letters actually change value in the case of a Gray code). For example, for the binary strings in lexicographical order, the successor of 0$\underline{1}$*0*$\underline{111}$ is 0$\underline{11}$00*0* (here the letters that are at their last value in the defining sequence of their prefix are underlined and the pivotal letter is italicized). In Gray code order, the successor of $\underline{10}$*1*$\underline{10}$ is $\underline{100}$1*0*.

The first step, then, in designing a non-recursive sequencing algorithm for a set of genlex word-lists is to generate the smallest lists in the set and examine them to determine the defining sequences as a function of the prefix or suffix. The reader is invited to examine the first column of Table 2 and determine the defining sequences for the Liu-Tang Gray code; the answer will be given near the beginning of Section 4. In the Johnson-Trotter Gray code (column 2 of Table 3) the pattern followed by the letters is more easily described as a change of position rather than value: the largest number moves alternately to the left and right by swapping places with its immediate neighbor, and while it is temporarily stationary the next largest number moves, and so on. Some transformations are necessary to make this Gray code genlex; they are illustrated in Table 6 near the end of Section 3. The other Gray codes illustrated above are more complicated; we discovered the defining sequences for them and will refer the reader to the appropriate sources later on. Most of the Gray codes in the literature are either genlex or else can be made genlex by a transformation (some transformations

will be discussed in the next section), a notable exception being C. Savage's Gray code for integer partitions [12].

If the defining sequence of a prefix is an easily calculable function of the prefix, which is not the case for an English dictionary but is for most of the Gray codes in the literature, then the above-mentioned generic sequencing algorithm is practical and can usually be optimized by deriving from it a specialized sequencing rule. For the length-n binary strings in lexicographical order, the rule would be the one followed by a binary odometer: "Starting from the right, change all the 1s into 0s and the rightmost 0 to a 1 if there is one; otherwise the current string was the last one." In the original Gray code, the corresponding rule is: "If there is an even number of 1s, change the last letter; otherwise, change the letter immediately to the left of the rightmost 1 unless the word is 10...0, which is the last word in the list." The reader is invited to derive this sequencing rule from the defining sequence; such a derivation (which is probably folkloric) appears in the research report [13] which is available (by e-mail as a Microsoft Word file) on request (this research report also contains a discussion of designing efficient ranking and unranking algorithms and other results that will be indicated later). The sequencing rule, together with the observation that the parity of the number of 1s in a binary string always changes in passing from one word to the next, yields a very efficient sequencing algorithm for the original Gray code, but because it has to search the word from right to left to find the rightmost 1, it is does not run in $O(1)$ worst-case time.

3 The Bitner-Ehrlich-Reingold Method of Finding the Successor of a Word in $O(1)$ Worst-Case Time

In 1973 G. Ehrlich published an article [14] in which he called a sequencing algorithm *loopless* if it runs in $O(1)$ worst-case time and presented loopless sequencing algorithms for a number of word-lists, of necessity Gray codes, representing subsets, combinations, permutations and several other combinatorial objects, but no general method was proposed in that article. Three years later, he collaborated with J.R. Bitner and E.M. Reingold [15] to publish a loopless algorithm for generating the original Gray code, and their method, which finds the pivot in $O(1)$ worst-case time, has been applied in slightly modified form by other researchers, notably J.T. Joichi, D.E. White and S.G. Williamson [16], to other Gray codes. In this section we describe their method, which we call BER after its authors, and we apply the concept of defining sequences to show that it works on any genlex word-list that satisfies the added condition that every defining sequence has at least two values.

To sequence the word $(g[1], ..., g[n])$ in a prefix-partitioned list the BER method uses an auxiliary array $(e[0], e[1], ..., e[n])$ which is defined in such a way that $e[n]$ will always contain the pivot. To abbreviate the following discussion we use the following notation: for any defining sequence we call the first letter a and the last letter z, and the next letter after g we call $s(g)$. A *z-subword* $(g[i+1], ..., g[j])$ of $(g[1], ..., g[n])$ is a maximal subword of adjacent letters all of

which are z. If $g[j]$ is the rightmost letter in a z-subword $(g[i+1], ..., g[j])$, then $e[j] = i$; otherwise $e[j] = j$. Since the pivot is the largest integer i such that $g[i]$ is not a z (or 0 if all the letters are z), this definition guarantees that $e[n]$ contains the pivot: if $g[n] \neq z$, then the pivot is n and $e[n] = n$, and if $g[n] = z$, then $g[n]$ is the rightmost letter in some z-subword $(g[i+1], ..., g[n])$, the pivot is i and $e[n] = i$.

We now assume that any defining sequence has at least two values, so that $z \neq a$ for any defining sequence. Initially, for all i, $g[i] = a$ so that $g[i] \neq z$ and is not, therefore, the rightmost letter of a z-subword. Accordingly, $e[j]$ is initialized to j for all j, satisfying the definition. The updating algorithm is given in Fig. 1. Since it contains no loops, it finds the pivot and updates the array $(e[0], e[1], ..., e[n])$ in $O(1)$ worst-case time. In a Gray code the number of letters to the right of $g[i]$ that have to be changed is also $O(1)$; if these changes can be made in $O(1)$ worst-case time as well (by determining the successor to each word by hand and incorporating this result into the program), then the sequencing algorithm is loopless.

```
i:=e[n];                                { i is the pivot }
if i>0 then
  g[i]:=s(g[i]);
  make each letter to the right of g[i] an a;
                        {O(1) changes for a Gray code }
  e[n]:=n;
  if g[i] is now z then
    e[i]:=e[i-1];
    e[i-1]:=i-1
  end if
else
  quit generating
end if.
```

Fig. 1. The Bitner-Ehrlich-Reingold method for loopless Gray code sequencing

As an example, suppose we are generating the length-5 binary strings in Gray code order. As usual, letters in the binary string that are z are underlined and the pivotal letter is italicized. Initially the binary string is (0,0,0,0,*0*) and the auxiliary array is (0,1,2,3,4,5). At some intermediate time the binary string is (1,0,*1*,1,0) and the auxiliary array is (0,1,0,3,4,3). After one more execution of the updating algorithm, the binary string is (1,0,0,1,*0*) and the auxiliary array is (0,1,2,0,4,5). When the binary string has reached the last word (1,0,0,0,0), the auxiliary array will be (0,1,2,3,4,0), and the next execution of the updating algorithm sets i to 0, thus instructing the main program to quit generating. Note that the definition of the content of the auxiliary array is satisfied at each step.

A formal proof that the definition of the auxiliary array is preserved by the updating algorithm, and is therefore always satisfied, appears in [17] and

will therefore not be included here. Instead we show the contents of the word $(g[1], ..., g[n])$ and the auxiliary array $(e[0], e[1], ..., e[n])$ before and after the execution of the updating algorithm (see Fig. 2 below), enabling the reader to construct the proof. We note that the fact that the updating algorithm preserves the definition of the auxiliary array depends on the fact that when a z becomes an a, it stops being a z, which is a consequence of the assumption that every defining sequence has at least two values.

```
If s(g[i]) is not z:
        the g-word                                the auxiliary array
index:.............. i  i+1 ...  n    index: ...... i i+1 ... n-1 n
                                      before
.................. g[i]  z  ...  z              ... i i+1 ... n-1 i
                                      after
................. s(g[i]) a  ...  a              ... i i+1 ... n-1 n

If s(g[i]) is z,
it is the rightmost letter in the z-subword beginning with g[k+1]:
        the g-word                                  the auxiliary array
index:..k k+1...i-1  i  i+1 ...  n    index:... i-1 i i+1 ... n-1 n
                                      before
......g[k] z ... z g[i]  z  ...  z          ...  k  i i+1 ... n-1 i
                                      after
......g[k] z ... z   z   a  ...  a          ... i-1 k i+1 ... n-1 n
```

Fig. 2. The updating algorithm of Fig. 1 preserves the definition of the auxiliary array

In [15] there is no condition for updating $e[i]$ and $e[i-1]$; no condition is necessary because once $g[i]$ changes it must become a z since each defining sequence has exactly two values. In the other works that use the BER method, a special condition is given that is logically equivalent to "if $g[i]$ is now z" for the particular Gray code being sequenced. The general treatment given here and in [17] shows how to apply the method to any prefix-partitioned word-list such that every defining sequence has at least two values (we call such a word-list *strictly* prefix-partitioned because each interval gets divided into at least two subintervals). If the word-list is strictly suffix-partitioned, the method can be modified by a simple left-right reflection. The auxiliary array is now $(e[1], ..., e[n], e[n+1])$. The pivot i is set to $e[1]$ and the condition for not quitting is $i \leq n$ instead of $i > 0$. The letters to the left of $g[i]$ are set to a. At the end, $e[1]$ is set to 1, and if $g[i]$ is now z, then $e[i]$ is set to $e[i+1]$ and $e[i+1]$ is set to $i+1$.

When the letters change position rather than value in a regular manner, an auxiliary position vector is used. We modify the first and third columns of Table 4 by giving the position vectors of the 1s in the Dyck words (see Table

5 below). Both of the resulting lists are strictly prefix-partitioned if we ignore the first letter; so the algorithm of Fig. 1 can be used. This was done for the Ruskey-Proskurowski Gray code in [18] and for the Vajnovszki-Walsh Gray code generalized to the suffixes of k-ary Dyck words in [9]. A loopless algorithm for the k-ary generalization of the Bultena-Ruskey Gray code was obtained by D. Roelants van Baronaigien [19]. Note that in the left column, two numbers can change from word to word, whereas in the right column, only one number changes and it does so by either 1 or 2.

Table 5. The position vectors of the 1s in the Ruskey-Proskurowski (*left*) and Vajnovszki-Walsh Gray code (*right*) for Dyck words. The pivot is italicized and any other letter that changes is underlined

1 2 3 *4*	1 2 3 *4*
1 2 3 *5*	1 2 3 *6*
1 2 3 *6*	1 2 3 *7*
1 2 *3* 7	1 2 *3* 5
1 2 4 *7*	1 2 4 *5*
1 2 4 *6*	1 2 4 *6*
1 2 *4* 5	1 2 *4* 7
1 2 5 *6*	1 2 5 *7*
1 *2* 5 7	1 *2* 5 6
1 3 5 *7*	1 3 5 *6*
1 3 *5* 6	1 3 *5* 7
1 3 4 *5*	1 3 4 *7*
1 3 4 *6*	1 3 4 *6*
1 3 4 7	1 3 4 5

For permutations, an inversion vector is used as an auxiliary array because it fills in the holes made by the elements of the prefix in its defining sequence. For the permutation $(p[1], ..., p[n])$ the *right-inversion vector* $(g[1], ..., g[n-1])$ is defined by the rule that $g[i]$ is the number of $p[j]$ to the right of but smaller than $p[i]$; in the *left-inversion vector*, $g[i]$ is the number of $p[j]$ to the left of but larger than $p[i]$. In the Johnson-Trotter Gray code, the letters change position rather than value; so the position vector - the inverse permutation - is also used. In the version of the Trotter-Johnson Gray code shown in Table 2, the list of left-inversion vectors is strictly prefix-partitioned (see Table 6 below); so it can be sequenced by the BER method [20]. Then the changes made in both the original and the inverse permutation can be effected in $O(1)$ worst-case time. This is almost the method used in [14]. We also used BER to design a loopless sequencing algorithm for the Nijenhuis-Wilf Gray codes for permutations [13] but, since it is much more complicated than the one for the Johnson-Trotter Gray code, we do not include it here.

Other transformations that have been used are Roelants van Baronaigien's I-code for involutions [21] and Vajnovszki's shuffles [22]. In [17] we used the I-code

and the BER method to design a loopless sequencing algorithm for fixed-point-free involutions, found a non-recursive description and a loopless sequencing algorithm for the Eades-McKay Gray code for combinations and then combined these two results to find a loopless sequencing algorithm for involutions with a given number of fixed points. We also found a Gray code for all involutions but while we were able to find a loopless sequencing algorithm for a necessary auxiliary array we were unable transform the change made in the auxiliary array into the corresponding change in the involution in $O(1)$ worst-case time. And we had to use an ad hoc method to design a loopless sequencing algorithm for the Eades-McKay Gray code because (see Table 2) neither this Gray code nor any other one for combinations is strictly genlex.

Table 6. The Johnson-Trotter Gray code for permutations (*left*), the inverse permutations (*centre*) and the left-inversion vector of the inverses (*right*)

1 2 3	1 2 3	0 0
1 3 2	1 3 2	0 1
3 1 2	2 3 1	0 2
3 2 1	3 2 1	1 2
2 3 1	3 1 2	1 1
2 1 3	2 1 3	1 0

4 A Generalized Version of the Bitner-Ehrlich-Reingold Method that Works on almost All Gray Codes

There are several loopless sequencing algorithms in the literature for Gray codes that are not strictly genlex. These include Ehrlich's algorithm for combinations [14], Chase's algorithm for his own Gray code for combinations [7], and our own algorithms: in [17] for the Eades-McKay Gray code for combinations and in [13] for the Knuth-Klingsberg Gray code for integer compositions [23] and for the Liu-Tang Gray code for combinations. Since this last algorithm is very efficient and has not been published, we include it here.

Chase noticed [7] that the Liu-Tang Gray code for n-combinations of m is suffix-partitioned, with each $g[i]$ moving in steps of 1 between its minimum of i and its maximum of $g[i+1]-1$ (or m if $i=n$), increasing if $n-i$ is even and decreasing otherwise. Thus the defining sequence of the empty suffix is $(n, n+1, ..., m)$ and the defining sequence of the positive-length suffix $(g[i+1], ..., g[n])$ is $(i, i+1, ..., g[i+1]-1)$ if $n-i$ is even and the reverse of this sequence otherwise. The loopless algorithm is based on the observation (see Table 2) that the pivot does not change very much from one word to the next. We make this observation more precise in the following theorem which we formally state and prove to show the mathematically inclined reader that we are actually capable of doing so.

Theorem 1. *Given an n-combination* $(g[1], g[2], ..., g[n])$*, let i be the pivot. Then for the next combination the pivot will lie between* $i-2$ *and* $i+1$ *if* $n-i$ *is even or between* $i-1$ *and* $i+2$ *if* $n-i$ *is odd.*

Proof. We first prove the upper bound. We assume that $i < n-1$; otherwise the result is trivial. If $n-i$ 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 above 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 pivot is bounded above by $i+1$. The same argument shows that if $n-i$ is odd the new pivot 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 $g[i]$ above by its minimum value of i.

We now prove the lower bound, assuming that $i > 2$. If $n-i$ 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 new pivot is bounded below by $i-1$. If $n-i$ 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 new pivot is bounded below by $i-2$. This completes the proof.

We use this theorem to derive from the defining sequences an algorithm which generates the combinations in $O(1)$ worst-case time with no auxiliary array. Aside from the combination itself, there are only 4 variables: i (the index of the current integer), x (the maximum value of $g[i]$), *Rise* (a Boolean variable which is true if $n-i$ 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 that updates all the variables in constant time is given in Fig. 3 below.

All of the above-mentioned loopless sequencing algorithms including the one in Fig. 3 (which satisfies Ehrlich's definition of loopless even though its pseudo-code contains a loop because the loop is guaranteed to be executed at most four times) are ad hoc because no systematic method was known for designing loopless sequencing algorithms for Gray codes that are not strictly genlex. But in [24] we published a generalization of the BER method that works for Gray codes that are *almost strictly genlex*: if a defining sequence of a prefix or suffix has only one value, then only one word in the list contains this prefix or suffix. This condition is satisfied by almost all the Gray codes in the literature, including Liu-Tang, Eades-McKay, Chase, Knuth-Klingsberg and the modification of the Knuth-Klingsberg Gray code we made in [24] for integer compositions whose parts are independently bounded above.

```
loop                        { iterated at most four times }
  if Rise then                      { g[i] should increase }
    if i=n then x:=m else x:=g[i+1]-1 end if;
    if g[i]<x 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 { else g[i] cannot increase so we increase i }
  else         { Rise is false and g[i] should decrease }
    if i>n 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 { else g[i] cannot decrease so we increase i }
  end if;
  i:=i+1; Rise:=not(Rise)
end loop.
```

Fig. 3. An algorithm for generating the next n-combination of {1,2,...,m} in $O(1)$ worst-case time and $O(1)$ extra space. For the first combination, $g[j] = j$ for each j from 1 to n, $i = n$ (since only $g[n]$ can change), *Rise* is true and *Done* is false

Clearly, a word-list of length-n words is almost strictly prefix-partitioned if and only if for each word $(g[1], ..., g[n])$ there is a positive integer $q \leq n$ such that the defining sequence of the prefix $(g[1], ..., g[j-1])$ has more than one value for each $j \leq q$ but only one value for each $j > q$. If $q = n$ for each word, then the list is strictly prefix-partitioned. The BER array $(e[0], ..., e[n])$ is defined in the same way whether the list is strictly prefix-partitioned or almost strictly: $e[j] = j$ unless $g[j]$ is the rightmost letter of a z-subword $(g[i+1], ..., g[j])$, in which case $g[j] = i$. In the first word all the letters are a, but if $q < n$ then all the letters to the right of $g[q]$ are also z; so $e[n]$ is initialized to the initial value of q and $e[j]$ is initialized to j for all $j < n$. The loopless sequencing algorithm for almost strictly partitioned Gray codes is shown in Fig. 4 below.

We illustrate this algorithm on a left-right-reversed version of the Liu-Tang Gray code (see Table 7 below).

Since the proof that the sequencing algorithm preserves the definition of the array $(e[0], ..., e[n])$ appears in [24] we instead give the contents of the word and

```
i:=e[n];                                  { i is the pivot }
if i>0 then
  g[i]:=s(g[i]);
  make each letter to the right of g[i] an a
                          {O(1) changes for a Gray code }
  update q;
  e[n]:=q;
  if g[i] is now z then
    if i=q then e[n]:=e[i-1] else e[i]:=e[i-1] end if;
    e[i-1]:=i-1
  end if
else
  quit generating
end if.
```

Fig. 4. The generalized Bitner-Ehrlich-Reingold method for loopless Gray code sequencing

Table 7. The generalized BER method applied to the left-right-reversed Liu-Tang Gray code. Letters that are z are preceded by a minus sign and the pivot is given in the last column

g[1]	g[2]	g[3]	g[4]	q	e[0]	e[1]	e[2]	e[3]	e[4]
4	-3	-2	-1	1	0	1	2	3	1
5	4	2	-1	3	0	1	2	3	3
5	4	-3	2	4	0	1	2	2	4
5	4	-3	-1	4	0	1	2	3	2
5	-3	-2	-1	2	0	1	2	3	1
-6	5	2	-1	3	0	0	2	3	3
-6	5	3	2	4	0	0	2	3	4
-6	5	3	-1	4	0	0	2	3	3
-6	5	-4	3	4	0	0	2	2	4
-6	5	-4	2	4	0	0	2	2	4
-6	5	-4	-1	4	0	0	2	3	2
-6	4	2	-1	3	0	0	2	3	3
-6	4	-3	2	4	0	0	2	2	4
-6	4	-3	-1	4	0	0	2	3	2
-6	-3	-2	-1	2	0	1	2	3	0

the BER array before and after the execution of the sequencing algorithm (see Fig. 5 below).

In [24] we modified the Knuth-Klingsberg Gray code for integer compositions to derive a new Gray code for compositions of an integer r whose parts are bounded between 0 and a positive integer which can be different for each part. If the bounds are $n-1$, $n-2$, ... , 2, 1, then these are the right-inversion vectors of the permutations of $1, 2, ..., n$ with r inversions. In this way we constructed a

```
If s(g[i]) is not z:
index:           ..................... i  i+1 ... q-1 q q+1 ... n-1 n
         the g-word
before           ...................g[i]  z  ...  z  z  z  ...  z  z

after            .................s(g[i]) a  ...  a  a  z  ...  z  z
         the auxiliary array
before                         ........ i  i+1 ... q-1 q q+1 ... n-1 i
after                          ........ i  i+1 ... q-1 q q+1 ... n-1 q

If s(g[i]) is z:
it is the rightmost letter in the z-subword beginning with g[k+1].

If q=i:
index:           ..... k k+1 ... i-1  i  i+1 .................. n-1 n
         the g-word
before           ....g[k] z  .... z g[i]  z  ..................  z  z

after            ....g[k] z  .... z   z   z  ..................  z  z
         the auxiliary array
before                         .... k   i  i+1 .................. n-1 i
after                          ... i-1  i  i+1 .................. n-1 k

If q>i (and q cannot be less than i):
index:           ..... k k+1 ... i-1  i  i+1 ... q-1 q q+1 ... n-1 n
         the g-word
before           ....g[k] z  .... z g[i]  z  ...  z  z  z  ...  z  z

after            ....g[k] z  .... z   z   a  ...  a  a  z  ...  z  z
         the auxiliary array
before                         .... k   i  i+1 ... q-1 q q+1 ... n-1 i
after                          ... i-1  k  i+1 ... q-1 q q+1 ... n-1 q
```

Fig. 5. The algorithm of Fig. 4 preserves the definition of the auxiliary array

Gray code for the permutations of $\{1, 2, ..., n\}$ with r inversions such that each permutation except the last one is transformed into its successor by either swapping two disjoint pairs of letters or rotating three letters so that the Hamming distance between two adjacent permutations is either 3 or 4 (see Table 8 below). The list of bounded compositions is almost strictly suffix-partitioned; so we used the above method to design a loopless sequencing algorithm for both bounded integer compositions and permutations with a fixed number of inversions. Usually loopless sequencing algorithms take a little longer to generate the whole word-list than it takes to generate the same set in lexicographical order because the latter runs in $O(1)$ average-case time and the former has to update auxiliary arrays, but the time trials we conducted showed that our loopless sequencing

algorithm for permutations with a fixed number of inversions runs about 20% faster than lexicographical-order generation.

Table 8. The 4-compositions of 7 with parts bounded by 4, 3, 2, 1 (*left*) and the permutations of {1,2,3,4} with 7 inversions (*right*) whose right-inversion vectors are the compositions to their left. The numbers that change are underlined

4 3 0 0	5 4 1 2 3
3 3 1 0	4 5 2 1 3
4 2 1 0	5 3 2 1 4
4 1 2 0	5 2 4 1 3
3 2 2 0	4 3 5 1 2
2 3 2 0	3 5 4 1 2
1 3 2 1	2 5 4 3 1
2 2 2 1	3 4 5 2 1
3 1 2 1	4 2 5 3 1
4 0 2 1	5 1 4 3 2
4 1 1 1	5 2 3 4 1
3 2 1 1	4 3 2 5 1
2 3 1 1	3 5 2 4 1
3 3 0 1	4 5 1 3 2
4 2 0 1	5 3 1 4 2

The reader will find in [24] a half-page-long formula for the number of permutations of $\{1, 2, ..., n\}$ with r inversions. We discovered this formula as an M. Sc. student and tried to publish it, but failed because it was slower to substitute into the formula than to use the recurrence relation satisfied by these numbers even to evaluate a single number. Including an algorithm that was faster than the recurrence was greeted with a grudging suggestion to revise and resubmit, which we declined. Finally, by including the Gray code and the loopless sequencing algorithm for these permutations we finally managed to claim priority for the formula some thirty-five years after discovering it, showing that Gray codes do indeed have practical applications.

Acknowledgment

I wish to thank Prof. Vincent Vajnovszki for having invited me to present these results at DMTCS 2003 and for suggesting several improvements to this article.

References

1. Gray, F.: Pulse Code Communication. U.S. Patent 2 632 058 (March 17, 1953)
2. Liu, C.N., Tang, D.T.: Algorithm 452, Enumerating M out of N objects. Comm. ACM 16 (1973) 485

3. Eades, P., McKay, B.: An Algorithm for Generating Subsets to Fixed Size with a Strong Minimal Interchange Property. Information Processing Letters 19 (1984) 131-133
4. Johnson, S.M.: Generation of Permutations by Adjacent Transpositions. Mathematics of Computation 17 (1963) 282-285
5. Trotter, H.F.: Algorithm 115: Perm. Comm. ACM 5 (1962) 434-435
6. Nijenhuis, A., Wilf, H.S.: Combinatorial Algorithms for Computers and Calculators, second edition. Academic Press, N.Y. (1978)
7. Chase, P.J.: Combination Generation and Graylex Ordering. Proceedings of the 18th Manitoba Conference on Numerical Mathematics and Computing, Winnipeg, 1988. Congressus Numerantium 69 (1989) 215-242
8. Ruskey, F.: Simple Combinatorial Gray Codes Constructed by Reversing Sublists. L.N.C.S. 762 (1993) 201-208
9. Vajnovszki, V., Walsh, T.R.: A loopless two-close Gray code algorithm for listing k-ary Dyck words. Submitted for publication
10. Ruskey, F., Proskurowski, A.: Generating Binary Trees by Transpositions. J. Algorithms 11 (1990) 68-84
11. Bultena, B., Ruskey, F.: An Eades-McKay Algorithm for Well-Formed Parentheses Strings. Inform. Process. Lett. 68 (1998), no. 5, 255-259
12. Savage, C.: Gray Code Sequences of Partitions. Journal of Algorithms 10 (1989) 577-595
13. Walsh, T.R.: A Simple Sequencing and Ranking Method that Works on Almost All Gray Codes. Research Report No. 243, Department of Mathematics and Computer Science, Université du Québec à Montréal (April 1995)
14. Ehrlich, G.: Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations: J. ACM 20 (1973) 500-513
15. Bitner, J.R., Ehrlich, G., Reingold, E.M.: Efficient Generation of the Binary Reflected Gray Code and its Applications. Comm. ACM 19 (1976) 517-521
16. Joichi, J.T., White, D.E., Williamson, S.G.: Combinatorial Gray Codes. SIAM J. Computing 9 (1980) 130-141
17. Walsh, T.R.: Gray Codes for Involutions. JCMCC 36 (2001) 95-118
18. Walsh, T.R.: Generation of Well-Formed Parenthesis Strings in Constant Worst-Case Time. J. Algorithms 29 (1998) 165-173
19. Roelants van Baronaigien, D.: A Loopless Gray-Code Algorithm for Listing k-ary Trees. J. Algorithms 35 (2000), no. 1, 100-107
20. Williamson, S.G.: Combinatorics for Computer Science. Computer Science Press, Rockville (1985)
21. Roelants van Baronaigien, D.: Constant Time Generation of Involutions. Congressus Numerantium 90 (1992) 87-96
22. Vajnovszki, V.: Generating Multiset Permutations. Accepted for publication in Theoretical Computer Science
23. Klingsberg, P.: A Gray Code for Compositions. Journal of Algorithms 3 (1982) 41-44
24. Walsh, T.R.: Loop-free sequencing of bounded integer compositions, JCMCC 33 (2000) 323-345