Available online at www.sciencedirect.com
ScienceDirect
Journal of Discrete Algorithms 4 (2006) 633–648
JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

A loop-free two-close Gray-code algorithm for listing k-ary Dyck words

Vincent Vajnovszki [a], Timothy Walsh [b,*]

[a] *LE21 FRE-CNRS 2309, Université de Bourgogne, B.P. 47 870, 21078 Dijon-Cedex, France*
[b] *Department of Computer Science, University of Quebec At Montreal, P.O. 8888, Station A, Montreal, Quebec, Canada, H3C 3P8*

Available online 30 August 2005

Abstract

P. Chase and F. Ruskey each published a Gray code for length n binary strings with m occurrences of 1, coding m-combinations of n objects, which is two-close—that is, in passing from one binary string to its successor a single 1 exchanges positions with a 0 which is either adjacent to the 1 or separated from it by a single 0. If we impose the restriction that any suffix of a string contains at least $k-1$ times as many 0's as 1's, we obtain k-suffixes: suffixes of k-ary Dyck words. Combinations are retrieved as special case by setting $k=1$ and k-ary Dyck words are retrieved as a special case by imposing the additional condition that the entire string has exactly $k-1$ times as many 0's as 1's. We generalize Ruskey's Gray code to the first two-close Gray code for k-suffixes and we provide a loop-free implementation for $k \geqslant 2$. For $k=1$ we use a simplified version of Chase's loop-free algorithm for generating his two-close Gray code for combinations. These results are optimal in the sense that there does not always exist a Gray code, either for combinations or Dyck words, in which the 1 and the 0 that exchange positions are adjacent.

Keywords: Gray code; k-ary Dyck words; Two-close; Loop-free algorithm

* Corresponding author. Tel.: +1 (514) 987-3000, extension 6139; fax: +1 (514) 987-8477.
E-mail addresses: vvajnov@u-bourgogne.fr (V. Vajnovszki), walsh.timothy@uqam.ca (T. Walsh).

1570-8667/$ – see front matter
doi:10.1016/j.jda.2005.07.003

1. Introduction

Combinatorial objects such as subsets of the n-set $\{1, 2, \ldots, n\}$, m-combinations (subsets of cardinality m) of the n-set, permutations of the n-set, binary trees and the more general k-ary trees (rooted trees of which every node has either k children or none) can be coded by words on a finite alphabet. To study various properties of these combinatorial objects one can generate a list of code-words of a fixed length, representing the objects of a fixed size, and test them all for these properties. The smaller the difference between consecutive code-words in the list, the less time it takes to generate each new word and to update the properties being studied in passing from one object to the next. It is therefore useful to put the set of code-words of fixed length into an order that minimizes the greatest difference between two consecutive code-words.

We call a family of word lists in which all the words in each list are of the same length a *Gray code* if the family contains arbitrarily long words but the number of positions in which two consecutive words in any list differ is bounded independently of the word length. An algorithm for generating a Gray code is called *loop-free* [5] if the number of operations necessary to transform each word into its successor in its list, or to determine that the current word is the last one in its list, is bounded independently of the word length.

A *binary string* is a word on the alphabet $\{0, 1\}$. The binary reflected Gray code, published by F. Gray [6], is a family of lists, one for each n, of length n binary strings in which consecutive strings in any list differ by only one letter. J.R. Bitner, G. Ehrlich and E.M. Reingold [1] used an auxiliary array to obtain a loop-free algorithm for implementing this Gray code.

An m-combination of the n-set can be coded by a binary string w with m 1's and $n - m$ 0's. Alternatively, we can use the length m array whose ith component is the position in w of the ith occurrence of 1 in w. We call this representation the 1-*vector* of w and we define the 0-*vector* analogously; we extend these definitions to lists of binary strings. A Gray code for combinations is called *minimal* if each binary string can be transformed into its successor by exchanging a single 1 with a 0. There are several minimal Gray codes in the literature for combinations. The simplest of these Gray codes [10] is called the Liu–Tang Gray code after its authors C.N. Liu and D.T. Tang; in the 1-vector (list) of this Gray code, at most two letters change from one 1-vector to the next. A loop-free implementation of this Gray code was obtained by T.R. Walsh [22] and possibly other researchers as well. Another Gray code for combinations was discovered and given a loop-free implementation by Ehrlich [5].

A minimal Gray code for combinations is called *homogeneous* if the 1 and the 0 that exchange positions are separated only by 0's, implying that in the 1-vector only a single letter changes value from one 1-vector to the next. Such a Gray code was discovered by P. Eades and B. McKay [4]; a non-recursive description and a loop-free implementation of this Gray code appear in [21].

A homogeneous Gray code for combinations is called *two-close* if the 1 and the 0 that exchange positions are either adjacent or separated by a single 0, implying that in the 1-vector the letter that changes value does so by at most 2. Such a Gray code is optimal in the sense that for some values of m and n there does not exist a Gray code in which the 1 and 0 that exchange positions are always adjacent [14]. A two-close Gray code for com-

Liu–Tang		Eades–McKay		Chase			Ruskey	
111000	123	111000	123	000111	456	123	001110	345
101100	134	110100	124	010011	256	134	100110	145
011100	234	101100	134	100011	156	234	010110	245
110100	124	011100	234	001011	356	124	011010	235
100110	145	011010	235	001101	346	125	101010	135
010110	245	101010	135	010101	246	135	110010	125
001110	345	110010	125	100101	146	235	111000	123
101010	135	100110	145	110001	126	345	110100	124
011010	235	010110	245	101001	136	245	101100	134
110010	125	001110	345	011001	236	145	011100	234
100011	156	001101	346	011100	234	156	011001	236
010011	256	100101	146	101100	134	256	101001	136
001011	356	010101	246	110100	124	356	110001	126
000111	456	011001	236	111000	123	456	100101	146
100101	146	101001	136	110010	125	346	010101	246
010101	246	110001	126	101010	135	246	001101	346
001101	346	100011	156	011010	235	146	001011	356
101001	136	010011	256	010110	245	136	100011	156
011001	236	001011	356	100110	145	236	010011	256
110001	126	000111	456	001110	345	126	000111	456

Fig. 1. The Liu–Tang, Eades–McKay, Chase and Ruskey Gray codes for 3-combinations of $\{1, 2, 3, 4, 5, 6\}$ in binary string and 1-vector form (and 0-vector form for Chase).

binations was discovered by P. Chase [3], who gave an obscure non-recursive description of its 1-vector and a FORTRAN program of a loop-free implementation of its 0-vector. F. Ruskey [15] then published a recursive description of another two-close Gray code for combinations, which is one of a family of Gray codes later published by T.A. Jenkyns and D. McCarthy [7].

In Fig. 1 we show the 3-combinations of $\{1, 2, 3, 4, 5, 6\}$ in binary string form and in 1-vector form as ordered by the Liu–Tang, Eades–McKay, Chase and Ruskey Gray codes; for the Chase Gray code the 0-vector is also shown.

A *Dyck word* is a binary string with the same number of 1's and 0's where any suffix has at least as many 0's as 1's. Dyck words code a wide variety of combinatorial objects including binary trees [24]. The terms minimal, homogeneous and 2-close will mean the same thing for Dyck words and their generalizations as they do for combinations. A minimal Gray code for Dyck words was published by Ruskey and A. Proskurowski [16]; a non-recursive description and a loop-free implementation of this Gray code appears in [20]. A homogeneous Gray code for Dyck words was discovered by B. Bultena and Ruskey [2].

A binary string with exactly $k-1$ times as many 0's as 1's, where any suffix has at least $k-1$ times as many 0's as 1's, is a generalization of a Dyck word because when $k=2$ we retrieve the definition of a Dyck word. An example of a Dyck word with $k=2$ is 101100; an example of a generalized Dyck word with $k=3$ is 100100. A bijection between these generalized Dyck words and k-ary trees was presented by J. Zaks [24] and used by D. Roelants van Baronaigien [12], who called such a binary string a bit sequence representation for a k-ary tree; we call it a *k-ary Dyck word*. Ruskey generated k-ary trees

lexicographically [13]; Roelants van Baronaigien [12] obtained a loop-free implementation of a homogeneous Gray code for k-ary Dyck words.

If the condition that a binary string has exactly $k-1$ times as many 0's as 1's is dropped from the definition of a k-ary Dyck word, then the string is a suffix of a k-ary Dyck word; we call such a string a *k-suffix*. When $k=1$, a k-suffix is an unrestricted binary string. Thus, k-suffixes with m 1's and $n \geqslant (k-1)m$ 0's generalize both m-combinations of the $(m+n)$-set and k-ary Dyck words with m 1's and $n=(k-1)m$ 0's: the combinations are retrieved by setting $k=1$ and the k-ary Dyck words by setting $n=(k-1)m$.

Until further notice, n will be the total number of 1's and 0's when we are referring to combinations and the number of 0's alone when we are referring to k-ary Dyck words for $k \geqslant 2$.

In this article we generalize Ruskey's two-close Gray code for combinations [15] (slightly modified to reverse all the strings) to obtain the first two-close Gray code for k-suffixes with m 1's and $n \geqslant (k-1)m$ 0's and, as a special case, k-ary Dyck words with m 1's and $n=(k-1)m$ 0's. This Gray code too is optimal in the sense that for some values of m and n there does not exist a Gray code for Dyck words in which the 1 and 0 that exchange positions are always adjacent [16]. In Section 2 we give a recursive description and then a non-recursive description of our Gray code, and in Section 3 we give a loop-free implementation of our Gray code when $k \geqslant 2$. In Section 4 we briefly discuss how we handled the case when $k=1$. The rest of this section contains an enumeration formula for k-suffixes with m 1's and $n \geqslant m(k-1)$ 0's.

In [13] there is a formula, a proof of which appears in [8] and other places, for the number of k-ary trees coded by k-ary Dyck words with m 1's and $(k-1)m$ 0's:

$$\binom{km}{m-1}\Big/ m \tag{1}$$

We provide an alternate proof of (1) in a form that enables us to deduce that the number of k-suffixes with m 1's and $n \geqslant (k-1)m$ 0's is

$$\binom{n+m}{m} - (k-1)\binom{n+m}{m-1}. \tag{2}$$

The approach we use is a standard one in combinatorial enumeration: we uniquely decompose an object of the type we want to enumerate into smaller objects of the same type or a type we have already enumerated, use the decomposition to find an equation satisfied by the generating function that counts the objects we want to enumerate, and then use Lagrange inversion to extract the coefficients of the generating function.

The general formula for Lagrange inversion is as follows. Let $y(x)$ be a power series in x that satisfies the following equation (in which we abbreviate $y(x)$ to y):

$$y = a + xg(y), \tag{3}$$

where $g(y)$ is a power series in y. Then for any other power series $f(y)$,

$$f(y) = f(a) + \sum_{n=1}^{\infty} \frac{x^n}{n!} * \frac{\mathrm{d}^{n-1}}{\mathrm{d}y^{n-1}} \left(\frac{\mathrm{d}(f(y))}{\mathrm{d}y} \big(g(y)\big)^n \right) \Bigg|_{y=a}. \tag{4}$$

An analytical proof of this formula is given in [23]; a combinatorial proof appears in [9]. Rather than differentiating $n-1$ times, we could instead make a change of variables, setting $y = z + a$ so that the derivatives in (4) are evaluated at $z = 0$. The $(n-1)$st derivative of a power series in z evaluated at $z = 0$ is just $(n-1)!$ times the coefficient of z^{n-1} in that power series. Thus (3) becomes

$$z = xg(z+a) \tag{5}$$

and (4) becomes

$$f(z+a) = f(a) + \sum_{n=1}^{\infty} \frac{x^n}{n} * \text{the coefficient of } z^{n-1} \text{ in } \left(\frac{\mathrm{d}}{\mathrm{d}z}\big(f(z+a)\big)\right)\big(g(z+a)\big)^n. \tag{6}$$

A non-empty k-ary Dyck word can be uniquely decomposed as $1S_1 0S_2 0S_3 0 \ldots 0S_k$, where each $S_i, i = k, k-1, \ldots, 2$, is the longest suffix (possibly empty) of $1S_1 0S_2 0S_3 0 \ldots 0S_i$ which is itself a k-ary Dyck word (S_1 is therefore also a k-ary Dyck word). It follows that if $y(x)$ is the generating function where the coefficient of x^m is the number of k-ary Dyck words with m 1's, then

$$y = 1 + xy^k. \tag{7}$$

Setting $y = z + 1$ in (7), we obtain

$$z = x(z+1)^k. \tag{8}$$

Eq. (8) is of the form of (5) with $a = 1$ and $g(z+1) = (z+1)^k$; also, we set $f(z+a) = y(x) = z + 1$. Substituting these values into (6), we see that the coefficient of x^m in the generating function $y(x) = z + 1$ is equal to the coefficient of z^{m-1} in $(z+1)^{km}/m$, which is given by (1).

A k-suffix with m 1's and $(k-1)m + j$ 0's can be uniquely decomposed as $S_1 0S_2 0S_3 0 \ldots 0S_{j+1}$, where each S_i is the longest suffix (possibly empty) of $S_1 0S_2 0S_3 0 \ldots 0S_i$ which is a k-ary Dyck word. It follows that the generating function in which the coefficient of x^m is the number of k-suffixes with m 1's and $(k-1)m + j$ 0's is $(y(x))^{j+1}$, where y is defined by (7). Again, we set $y = z + 1$ and obtain (8), which is of the form of (5) with $a = 1$ and $g(z+1) = (z+1)^k$, but this time we set $f(z+a) = (y(x))^{j+1} = (z+1)^{j+1}$. Substituting these values into (6), we see that the coefficient of x^m in the generating function $(y(x))^{j+1} = (z+1)^{j+1}$ is equal to the coefficient of z^{m-1} in $(j+1)(z+1)^{j+kn}/m$, which is equal to

$$\frac{j+1}{1}\binom{j+km}{m-1}.$$

Setting $j = n - (k-1)m$ and simplifying, we obtain (2).

2. A two-close Gray code for k-suffixes

In this section we present an alternative recursive description for Ruskey's two-close Gray code [15] for combinations represented by strings of m 1's and $n-m$ 0's. Then we modify it by reversing left to right all the binary strings and we generalize it from m-combinations of the $(n+m)$-set (represented by unrestricted binary strings, or 1-suffixes, with m 1's and n 0's) to k-suffixes with m 1's and $n \geqslant (k-1)m$ 0's. We give a recursive description of our Gray code and show that it is two-close, and then we give a non-recursive description.

We use the following notation. For each integer $m \geqslant 0$, the expression 0^m means m consecutive 0's with an analogous meaning for 1^m. The list obtained by appending the letter 1 at the end of each binary string in the list L is denoted by $L1$ with an analogous notation for 0 and for either letter preceding each binary string in the list L. The list L reversed (read from the last binary string to the first one) is denoted by L^R. The list L followed by the list M is denoted by L, M.

Ruskey's recursive description of his Gray code uses two lists, $L(n,m,1)$ and $L(n,m,0)$:

$$L(n,m,p) = \begin{cases} 0^n & \text{if } m = 0, \\ 1^{n-1}0, L(n-1,m-1,1)1 & \text{if } p = 1 \text{ and } m = n-1, \\ L^R(n-1,m,1)0, L(n-1,m-1,0)1 & \text{if } p = 1 \text{ and } \\ & 0 < m < n-1, \\ L(n-1,m,1)0, L(n-1,m-1,1)1 & \text{if } p = 0 \text{ and } \\ & 0 < m < n-1. \end{cases} \tag{9}$$

We modify it so that only one list is needed. Let $L(n,m) = L(n,m,1)$ if $m < n$ and $L(n,n) = 1^n$, so that $L(n,m)$ contains all the binary strings with m 1's and $n-m$ 0's. Eliminating $L(n,m,0)$ from (9) we obtain the following recursive description for $L(n,m)$:

$$\begin{aligned} &L(n,n) = 1^n, \\ &L(n,0) = 0^n, \\ &L(n,n-1) = 1^{n-1}0, L(n-1,n-2)1 \ (= 1^{n-1}0, 1^{n-2}01, \ldots, 01^{n-1}), \\ &L(n,1) = L^R(n-1,1)0, 0^{n-1}1, \\ &L(n,m) = L^R(n-1,m)0, L(n-2,m-1)01, L(n-2,m-2)11 \\ &\quad \text{if } 1 < m < n-1. \end{aligned} \tag{10}$$

The first binary string in $L(n,m)$ is $0^{n-m-1}1^m0$ when $n > m$ and the last one is always $0^{n-m}1^m$.

We modify Ruskey's Gray code as defined by (10) by reversing left-to-right all the binary strings and letting n be the number of 0's and then we generalize it from combinations (1-suffixes) to k-suffixes. Let $L_k(n,m)$ be the list defined by formula (11):

$$\begin{aligned} &L_k(n,0) = 0^n, \\ &L_k(n,m>0) = 1{L_k}^R(n,m-1) \quad \text{if } n = m(k-1), \end{aligned}$$

$$L_k(n,1) = \begin{cases} 0L_k(n-1,1), 10^n & \text{if } n = k, \\ 0{L_k}^R(n-1,1)10^n & \text{if } n > k, \end{cases}$$

$$L_k(n,m>1) = \begin{cases} 0L_k(n-1,m), 10L_k(n-1,m-1), 11L_k(n,m-2) & \text{if } n = m(k-1)+1, \\ 0{L_k}^R(n-1,m), 10L_k(n-1,m-1), 11L_k(n,m-2) & \text{if } n > m(k-1)+1. \end{cases} \tag{11}$$

It can easily be shown by induction on $n+m$ that each binary string in $L_k(n,m)$ is a k-suffix with m 1's and $n \geqslant m(k-1)$ 0's. The observation that follows formula (13) implies that $L_k(n,m)$ exhaustively lists those suffixes. When $k=1$, by reversing left-to-right all the binary strings in (11) one obtains a version of (10) that defines m-combinations of the $(n+m)$-set.

A Java applet generating the list $L_k(n,m)$ is available on the web site of the first author [18].

In what follows we will be referring to the *lines* of (11) according to the positions of the right-hand sides of (11). For example, line 4 of (11) is the equation "$L_k(n,1) = 0{L_k}^R(n-1,1), 10^n$ if $n > k$".

Theorem 1. *Eqs.* (11) *define* $L_k(n,m)$ *uniquely* (*that is, unambiguously*).

Proof. We observe that (11) is a well-defined recursive definition. If $m=0$, we apply line 1, the set of base cases. Suppose that $m>0$. The case where $n=m(k-1)$ is covered by line 2. The case where $n=m(k-1)+1$ is covered by line 3 (where $m=1$) and line 5 (where $m>1$). The case where $n>m(k-1)+1$ is covered by line 4 (where $m=1$) and line 6 (where $m>1$). Since $n \geqslant m(k-1)$, all the cases are covered, each one by a single line. Moreover, in each of the cases where $m>0$, at least one of the parameters (m or n) is reduced by at least 1 and the other one is not increased, so that eventually either m drops to 0 (a base case) or n drops to $m(k-1)$. In the latter case, m is reduced (line 2), so that the recursion eventually stops at some base case. □

Theorem 2. *If* $n=m(k-1)$, *then the first binary string in* $L_k(n,m)$ *is* 1^m0^n *and the last one is* $101^{m-1}0^{n-1}$ (*unless* $n=0$ *so that* $k=1$ *or* $m=0$, *in which case the list consists of the single binary string* 1^m). *If* $n>m(k-1)$, *then the first binary string is* 01^m0^{n-1} *and the last one is* 1^m0^n.

Proof. We proceed by induction on the total string length $n+m$. If $n+m=0$, then by line 1 of (11) the list consists of the empty binary string, in agreement with the theorem.

Now we suppose that $n+m>0$ and that the theorem holds for all lists of binary strings of length less than $n+m$ so that, in particular, it holds for all the sublists in (11) (minus the prefixes).

If $m=0$, then $n>m(k-1)$, so that the first binary string and the last one should both be 0^n, in agreement with first line of (11). Suppose that $m>0$ and $n=m(k-1)$. If $k=1$, then $n=0$. The second line of (11) implies that $L_1(0,m)=1^m$, in agreement with the theorem. If $k>1$, then $n>(m-1)(k-1)$; so by the induction hypothesis the first binary string of $L_k(n,m-1)$ is $01^{m-1}0^{n-1}$ and the last one is $1^{m-1}0^n$. By the second line of (11),

the first binary string of $L_k(n,m)$ is 1^m0^n and the last one is $101^{m-1}0^{n-1}$, in agreement with the theorem.

Now suppose that $n > m(k-1)$. If $n = m(k-1)+1$, then $n-1 = m(k-1)$, so that the first binary string in $L_k(n-1,m)$ is 1^m0^{n-1}, and if $n > m(k-1)+1$, then $n-1 > m(k-1)$, so that the first binary string in ${L_k}^R(n-1,m)$ is also 1^m0^{n-1}, and in either case, by lines 4 and 6 of (11) the first binary string of $L_k(n,m)$ is 01^m0^{n-1}, in agreement with the theorem. If $m = 1$, then by lines 3 and 4 of (11) the last binary string is 10^n, in agreement with the theorem. If $m > 1$, then $n > (m-2)(k-1)$, so that the last binary string of $L_k(n,m-2)$ is $1^{m-2}0^n$; thus by lines 5 and 6 of (11) the last binary string of $L_k(n,m)$ is 1^m0^n, in agreement with the theorem. This completes the proof. □

Theorem 3. *The list defined by (11) is a two-close Gray code.*

Proof. Again we use induction on the string length, anchored by the trivial case where the string length is 0. By the induction hypothesis, all the sublists in the right sides of (11) are two-close Gray codes; so it suffices to show that this condition is satisfied by the transition from one binary string to the next across the commas. Lines 1 and 2 have no transitions and the transition of lines 3 and 4 are special cases of the first transition of lines 5 and 6, respectively, when $m = 1$. If $k > 1$, then by Theorem 2 the first transition of line 5 is $0101^{m-1}0^{n-2}$, $1001^{m-1}0^{n-2}$, the first transition of line 6 is 001^m0^{n-2}, $1001^{m-1}0^{n-2}$ and the second transition of both lines 5 and 6 is $101^{m-1}0^{n-1}$, $1101^{m-2}0^{n-1}$. If $k = 1$, then by Theorem 2 the first transition of line 5 is 01^m, 101^{m-1}, the second transition of line 5 is 101^{m-1}, 1101^{m-2}, and the transitions of line 6 are the same as if $k > 1$. In all these transitions a single 1 exchanges positions with a 0 which is either adjacent to the 1 or separated from it by a single 0. This completes the proof. □

In the rest of this section we give a detailed non-recursive description of the Gray code defined by (11), using a general method from [21] and [22] that we summarize in the next three paragraphs.

A list of words is called *prefix-partitioned* if all the words in the list with the same prefix form an interval of contiguous words in the list; *suffix-partitioned* word lists are defined analogously. In each interval of words $(c[1],\ldots,c[m])$ in which the prefix $(c[1],\ldots,c[i-1])$ or the suffix $(c[i+1],\ldots,c[m])$ is constant, the letter $c[i]$ runs through a sequence of distinct values defined by the prefix or suffix. This property generalizes graylex order as defined in [3], in which the letters are numbers and the sequence of values assumed by each number is required to be monotone, and graylex order in turn generalizes lexicographical order, in which the sequence is required to be increasing. Most of the Gray codes in the literature are either prefix- or suffix-partitioned or else can be transformed into such lists; position vectors, 0- and 1-vectors, P-suites [11] and shuffles [17] are among the transformations that can be used.

All the Gray codes in Fig. 1 are suffix-partitioned, but of these only the Liu–Tang Gray code is graylex, and we use that one to illustrate how the sequence of distinct values assumed by each letter constitutes a non-recursive description of a prefix- or suffix-partitioned list. The following description of that Gray code in 1-vector form appears in [3]: the last letter $c[m]$ of the word $(c[1],\ldots,c[m])$ runs through the sequence

of distinct values $m, m+1, \ldots, n$, and in each interval of words in which the suffix $(c[i+1], \ldots, c[m])$ is constant, the ith letter $c[i]$ runs through the sequence of distinct values $i, i+1, \ldots, c[i-1]-1$ if $m-i$ is even or $c[i+1]-1, \ldots, i+1, i$ if $m-i$ is odd.

To get the first word in the list, we set $c[m]$ to its first value in the sequence for $c[m]$, which is m; then, for each i from $m-1$ down to 1 we set $c[i]$ to the only value it can have, which is i. To get the last word in the list, we set $c[m]$ to its last value, which is n, and (if $m \geqslant 2$) we set $c[m-1]$ to its last value, which is $m-1$; then, for each i from $m-2$ down to 1 we set $c[i]$ to the only value it can have, which is 1. To get the successor to the word $(c[1], \ldots, c[m])$, we find the smallest i such that $c[i]$ is not at its last value in the sequence defined by the suffix $(c[i+1], \ldots, c[m])$, change it to its next value in that sequence; then, for each j from $i-1$ down to 1, we set $c[j]$ to the first value in the sequence determined by the suffix $(c[j+1], \ldots, c[m])$. For example, let $n=6, m=3, c[1]=1, c[2]=2$ and $c[3]=4$. The sequence for $c[1]$ is (1); so $c[1]$ is at its last value. The sequence for $c[2]$ is $(3, 2)$; so $c[2]$ is also at its last value. The sequence for $c[3]$ is $(4, 5, 6)$; so $c[3]$ is not at its last value and we set $c[3]$ to its next value after 4, which is 5. The sequence for $c[2]$ is now $(4, 3, 2)$; so we set $c[2]$ to its first value, which is 4. The sequence for $c[1]$ is now $(1, 2, 3)$ and $c[1]$ is already at its first value of 1; so we do not change it.

Our generalization of Ruskey's Gray code, its 1-vector and its 0-vector are all prefix-partitioned: this assertion follows from the observation that only prefixes are appended to the lists on the right side of (11) and that in each line the appended prefixes are all distinct and none of them is a prefix of another. We opt for the 1-vector for reasons that are explained in [19].

Let $(c[1], \ldots, c[m])$ be the 1-vector of a binary string with m 1's and $n \geqslant m(k-1)$ 0's. In what follows, *the prefix of* $c[i]$ will mean the prefix $(c[1], \ldots, c[i-1])$. A necessary condition for (11) to list k-suffixes is that for each i, $1 \leqslant i \leqslant m$, the farthest right the ith last 1 can move is to the kith last position in the string, so that the maximum value that $c[i]$ can attain in an interval of 1-vectors in which its prefix is fixed is given by

$$max(i) = n + m + 1 - k(m - i + 1). \tag{12}$$

Since the Dyck word condition on suffixes permits 1's to go arbitrarily far to the left, the minimum value of $c[i]$ in the same interval of 1-vectors is (trivially):

$$min(i) = 1 \quad \text{if } i = 1 \quad \text{and} \quad c[i-1]+1 \quad \text{otherwise.} \tag{13}$$

The non-recursive description given by Theorem 4 includes the assertion that, in an interval of 1-vectors with the prefix of $c[i]$ fixed, $c[i]$ attains every value from $min(i)$ to $max(i)$, so that (11) does in fact generate all the k-suffixes with m 1's and $n \geqslant m(k-1)$ 0's.

We call a sequence of numbers that consists of all the integers from $min(i)$ to $max(i)$ *even-rising* if, aside from $min(i)$, which is either its first or its last number, it rises through consecutive even numbers and then falls through consecutive odd numbers, with $max(i)$ being part of the rising sequence if it is even or the falling sequence if it is odd. An analogous definition is given for an *odd-rising* sequence. An even- or odd-rising sequence is completely determined by $min(i)$, $max(i)$ and its first and last numbers. If the first number is $min(i)$, then the last number is either $min(i)+1$ or $min(i)+2$, and we say that the sequence is *of form* 1, 2 or 1, 3, respectively. Similarly, if the last number is $min(i)$, then

Table 1
The possible forms of even-rising and odd-rising sequences

Form	Even-rising	Odd-rising
1, 2	2, 4, 6, 7, 5, 3 and 2, 4, 6, 5, 3	1, 3, 5, 6, 4, 2 and 1, 3, 5, 7, 6, 4, 2
1, 3	1, 2, 4, 6, 7, 5, 3 and 1, 2, 4, 6, 5, 3	2, 3, 5, 6, 4 and 2, 3, 5, 7, 6, 4
2, 1	2, 4, 6, 7, 5, 3, 1 and 2, 4, 6, 5, 3, 1	3, 5, 6, 4, 2 and 3, 5, 7, 6, 4, 2
3, 1	4, 6, 7, 5, 3, 2 and 4, 6, 5, 3, 2	3, 5, 6, 4, 2, 1 and 3, 5, 7, 6, 4, 2, 1

the first number is either $min(i) + 1$ or $min(i) + 2$, and we say that the form is 2, 1 or 3, 1, respectively. These forms are illustrated in Table 1.

In [20] a 1 in a Dyck word is called a *liberal* if it is not in its rightmost position. Here we call a 1 a *tory* if it is in its rightmost position and is not the first symbol of the binary string, and we give the same name to a $c[i]$ in the 1-vector which is equal to $max(i)$ and is greater than 1.

Theorem 4. *Let* $(c[1], \ldots, c[m])$ *be the* 1*-vector of a word in the list* $L_k(n, m)$ *defined by* (11).

Assertion 1. *The sequence of distinct values assumed by* $c[i]$ *in the interval of* 1*-vectors in which its prefix is fixed attains all the values from* $min(i)$ *to* $max(i)$ *and only those values.*

Assertion 2. *It is even-rising if the prefix of* $c[i]$ *contains an even number of tories and odd-rising otherwise.*

Assertion 3. *If* $n = m(k-1)$*, then* $c[1]$ *is fixed at* 1; *otherwise* $c[1]$ *is of form* 2, 1.

Assertion 4. *For each* $i > 1$ *the form of* $c[i]$ *depends on* $c[i-1]$ *as follows.*

Assertion 4.1. *If* $c[i-1] = max(i-1)$ *and* $k = 1$*, then* $c[i]$ *stays at* $c[i-1] + 1 = max(i)$.

Assertion 4.2. *Suppose that* $c[i-1] = max(i-1)$ *and* $k > 1$. *If* $c[i-1]$ *is going to fall by* 1*, then the form is* 1, 2; *otherwise* (*i.e., if it has just risen by* 1) *the form is* 2, 1.

Assertion 4.3. *Suppose that* $c[i-1]$ *is rising and is not* $max(i-1)$. *If* $c[i-1]$ *is going to rise by* 1 *and* (*either* $k = 1$ *or* $c[i]$ *will not rise to* $max(i-1)$)*, then the form of* $c[i]$ *is* 1, 2; *otherwise the form is* 1, 3.

Assertion 4.4. *Suppose that* $c[i-1]$ *is falling and is not* $max(i-1)$. *If* $c[i-1]$ *has just fallen by* 1 *and* (*either* $k = 1$ *or* $c[i]$ *did not fall from* $max(i-1)$)*, then the form of* $c[i]$ *is* 2, 1; *otherwise the form is* 3, 1.

Proof. We first show that $min(i)$ and $max(i)$ are the actual minimum and maximum value that $c[i]$ can attain. No $c[i]$ can go lower than $min(i)$ because $1 \leqslant c[1] < c[2] < \cdots$; the assertion that this minimum can be attained can be proved by induction on the string length using the observation that each line of (11) contains either a prefix or a single binary string beginning with 1. The assertion that $max(i)$ is the actual maximum value of $c[i]$ can also be proved by induction. Putting prefixes in front all the binary strings in a list does not affect how far right a 1 can go; so the inductive hypothesis implies that all the 1's in the sublists of (11) that are not part of the prefixes go as far right as the theorem says they should and no farther. Lines 3 through 6 of (11) each have a prefix 0 in the first sublist, allowing even the first and second 1 to reach their rightmost positions as part of shorter binary strings. And in line 2, where the first 1 is constrained to be in position 1, $max(1) = 1$.

Assertion 2 of the theorem—that $c[i]$ is even-rising if the prefix contains an even number of tories and odd-rising otherwise—also follows by induction. It is trivial for $L_k(0,0)$; the inductive hypothesis implies that it is true for all the sublists minus their prefixes. Putting an odd-length prefix in front of all the binary strings in a list changes the parity of all the numbers in the 1-vector; this cancels the list-reversal everywhere except in the first sublists of lines 3 and 5. But in that case (and line 3 is a special case of line 5 with $m = 1$), $n - 1 = m(k-1)$, so that by substituting from line 2 we find that there is both a prefix 01 and a list-reversal. When $n = m(k-1)+1$, then $max(1) = 2$, so that the list-reversal cancels the effect of putting a $c[1] = max(1)$ to the left of all the other $c[i]$. This shows that all the $c[i]$ except possibly those that are part of the prefixes appended to the lists of binary strings in (11) behave properly. Furthermore, $c[1]$ and $c[2]$ also behave properly until they become part of the prefixes in (11), at which point $c[1]$ drops to 1 and $c[2]$ drops to 2, and these final values, being their respective minima, affect only the form. This completes the proof of assertion 2. Even- and odd-rising sequences both attain all the values from their minimum to their maximum; so assertion 1 holds as well.

Finally we prove the assertions about the form, beginning with assertion 3 about the form of $c[1]$.

If $n = m(k-1)$, then from line 2 we know that $c[1]$ stays at 1. Otherwise, since it is even-rising and ends at its minimum value 1, it must be of the form 2, 1 as asserted in the theorem.

We prove assertion 4 by examining all the cases covered by the sub-assertions 4.1 through 4.4.

Suppose that $c[i-1] = max(i-1)$.

If $k = 1$, then $max(i) = min(i)$; so $c[i]$ stays at that value, in accord with assertion 4.1.

We now prove assertion 4.2, which deals with the case where $k > 1$.

Suppose that $c[i-1]$ is going to fall by 1. Then either it just rose by 2 or else can assume only two values: $max(i-1)$ followed by $max(i-1)-1$.

Suppose it just rose by 2. Then $c[i]$ must start at $c[i-1]+1$. If it ended at $c[i-1]+3$, then it would have to hit $c[i-1]+2$ on the rise. But $c[i]$ is even-rising if and only if $c[i-1]$ is odd-rising; so $c[i-1]+2$ is of the wrong parity to be part of the rising sequence of $c[i]$. Thus, $c[i]$ must end at $c[i-1]+2$, and the form is 1, 2 as asserted.

Suppose that $c[i-1]$ can assume only the sequence of values $max(i-1), max(i-1)-1$. Then k must be 2, so that $c[i]$ can also assume only the two values $c[i-1]+1$ and $c[i-1]+2$. Consider the sublist S of binary strings consisting of just those for which $c[i-1]$ of the corresponding 1-vector follows the sequence of values $max(i-1), max(i-1)-1$, and then consider the list T of suffixes of S beginning one letter to the left of the $i-1$st 1. By successively applying (11) we eventually find either T or T^R. Since $c[i] = max(i)$, all the suffixes in T have exactly one more 0 than 1 ($m-i+3$ 0's and $m-i+2$ 1's); so T or T^R is generated by line 5 of (11):

$$
\begin{aligned}
&L_2(m-i+3, m-i+2)\\
&\quad = 0L_2(m-i+2, m-i+2), 10L_2(m-i+2, m-i+1),\\
&\qquad 11L_2(m-i+3, m-i).
\end{aligned}
$$

In this list, the first 1 moves left and the corresponding number $c[i-1]$ in the 1-vector falls; so $L_2(m-i+3, m-i+2) = T$ and not T^R. The sublist of T in which $c[i-1] = max(i)$ is $0L_2(m-i+2, m-i+2)$. From Theorem 2 the first binary string in this list is $01^{m-i+2}0^{m-i+2}$ and the last one is $0101^{m-i+1}0^{m-i+1}$. The second 1 starts adjacent to the first 1 and ends up separated from it by a single 0; so $c[i]$ follows a sequence of form $1, 2$ as asserted.

If instead $c[i-1]$ just rose by 1, then by the same argument applied to the reversed sequence we can show that $c[i]$ follows a sequence of form $2, 1$, proving assertion 4.2.

We now prove assertion 4.3. Suppose that $c[i-1]$ is rising. Then $c[i]$ cannot end at $c[i-1]+1$, or even at $c[i-1]+2$ if $c[i-1]$ is going to rise by 2, or else, when $c[i-1]$ does rise, it will bump into or pass $c[i]$, forcing $c[i]$ to rise at the same time, contradicting homogeneity; so $c[i]$ must start at $c[i-1]+1$ and end at $c[i-1]+3$ if $c[i-1]$ is going to rise by 2. The form of the sequence is thus $1, 3$ as asserted. If $c[i]$ is going to rise by 1 to $max(i)$ and $k > 1$, then since the form is going to be $2, 1$, $c[i]$ must now rise to $c[i-1]+3$ and the form is $1, 3$; if $k = 1$ then $max(i) = c[i-1]+2$, so the form is $1, 2$. Suppose that $c[i-1]$ is going to rise by 1 but not to $max(i-1)$. It will rise to $c[i-1]+1$ and then rise again; so $c[i]$ must end at $c[i-1]+2$ so that it will start there when $c[i-1]$ is at $c[i-1]+1$, allowing $c[i-1]$ to rise again. The form is thus $1, 2$ as asserted.

If $c[i-1]$ is falling, the arguments used in the case when it was rising can be applied to the reversed sequence to prove assertion 4.4.

All the cases having been exhausted, the proof is complete. □

3. A loop-free implementation when $k \geqslant 2$

In this section we obtain a loop-free implementation of the Gray code defined by formula (11) using a general method from [21] and [22] that we summarize below.

In a Gray code the number of letters that have to be changed in transforming a word into its successor is bounded above by a constant, but before these letters can be changed they must first be located. In a prefix-partitioned list, the leftmost letter that has to be changed is the rightmost letter $c[p]$ that is not at its last value in the sequence determined by its prefix $(c[1], \ldots, c[p-1])$; we call its position p the *pivot*. In [1] an auxiliary array was used to locate the pivot in the binary reflected Gray code [6] in constant time. In [21] and [22] we gave a general description of this array and showed that it finds the pivot in constant time for any *strictly* prefix- or suffix-partitioned list—that is, one where each sequence of distinct values assumed by a letter has at least two values, so that the first value is never also the last one—and we summarize these results below.

For a strictly prefix-partitioned list of length m words, the auxiliary array, which we call the e-array after its inventor Ehrlich, is of the form $(e[0], e[1], \ldots, e[m])$. This array serves to keep track of the left and right endpoints of the maximal (by inclusion) subwords $(c[j+1], \ldots, c[i])$ each of whose letters is in its last position; we call such a subword a *z-subword*. For each i, $e[i] = i$ unless $c[i]$ is the rightmost letter of a z-subword $(c[j+1], \ldots, c[i])$, in which case $e[i] = j$. For the first word, none of the letters is at its last value because the list is strictly partitioned; so $e[i]$ is initialized to i for each i. For any word, we first set p to $e[m]$. If $p = 0$, then all the letters are at their last value; so the

current word is the last one. If $p > 0$, then $g[p]$ is not at its last value but all the letters to its right are at their last values; so p is the pivot. We first update the word and then update the e-array as follows. Since $c[m]$ will be set to its first value unless m was the pivot, we set $e[m]$ to m. Then, if after the update $g[p]$ is now at its last value as determined by the prefix $(c[1], \ldots, c[p-1])$, it is now the rightmost letter of a z-subword which extends a z-subword that used to end in $c[p-1]$ if there was one; so we set $e[p]$ to $e[p-1]$ and then set $e[p-1]$ to $p-1$. A case-by-case proof that this update preserves the description of the e-array appears in [21].

When $k \geqslant 2$ the list of k-suffix 1-vectors is strictly prefix-partitioned (except that if $n = m(k-1)$, then $c[1]$ is always 1 and can be ignored); so the e-array of [1] enables the pivot to be found in constant time. If i is the pivot, we need to know whether $c[i]$ is rising or falling and whether the value to which we change it is its last value; a second array s does these duties and can also be updated in constant time. We also need a third auxiliary array p which determines if $c[i]$ has an even or odd number of tories to its left. When $c[i]$ reaches or leaves its maximum value, all the elements of p to the right of $p[i]$ would have to be changed, which could not be done in constant time if we insisted that all the $p[i]$ be correct even when they are not being used. Instead, we evaluate each $p[i]$ when i is the pivot and $c[i]$ is at its first value. We keep a variable *Odd* which is 1 if the whole 1-vector c contains an odd number of tories and 0 otherwise. Then $p[i]$ is set to *Odd* if and only if there are an even number of tories in the rest of c, and the following theorem shows that only one letter to the right of $c[i]$ has to be checked.

Theorem 5. *Let i be the pivot. If $k > 2$, then there are no tories to the right of $c[i]$. If $k = 2$, then the only letter to the right of $c[i]$ that can be a tory is $c[i+1]$.*

Proof. Since i is the pivot, all the letters to the right of $c[i]$ are at their last values; so we consider the conditions under which the last value in a sequence, which we call $last(j)$, can also be the maximum value $max(j)$.

For each $j > 1$, the number of values that $c[j]$ can assume is equal to $max(j-1) - c[j-1] + k$. If $k \geqslant 3$, then the only way for $last(j)$ to be equal to $max(j)$ is if $c[j-1] = max(j-1)$ and $c[j]$'s sequence is of the form 1, 3. But from Theorem 4 we see that the form is 1, 3 only when $c[j-1]$ is rising and not when it is equal to $max(j-1)$. Therefore, if $k \geqslant 3$, then there are no tories to the right of $c[i]$.

Now suppose that $k = 2$. Now there are two possibilities for $c[j]$ to be a tory: $c[j-1] = max(j) - 1$ and $c[j]$'s sequence is of form 1, 3 or $c[j-1] = max(j-1)$ and $c[j]$'s sequence is of form 1, 2.

Suppose that $c[i] = max(i)$. By Theorem 4, $c[i+1]$'s sequence is of form 1, 2 if and only if $c[i]$ is going to fall by 1. In this case, $c[i+1]$ is now a tory. But since it can only assume two values, it has just risen by 1; so, by Theorem 4, $c[i+2]$'s sequence is of form 2, 1 and $c[i+2]$ is not a tory. Whenever any $c[j-1]$ is falling, $c[j]$'s sequence is either of form 2, 1 or 3, 1, so that if $c[j]$ is at its last value, then it is falling too. It follows that none of the letters to the right of $c[i+1]$ can be tories.

Suppose that $c[i] = max(i) - 1$ and $c[i+1]$'s sequence is of form 1, 3. Then $c[i+1]$ is a tory which has just risen by 1; so by the same argument as above, none of the letters to

its right can be tories. It follows that $c[i+1]$ is the only letter to the right of $c[i]$ that can be a tory. □

Aside from the variables i (the pivot) and Odd, we keep three other variables for the sake of efficiency: $MN = min(i)$, $MX = max(i)$ and $M0 = n + m - k(m+1)$ (since $MX = M0 + ki$). The array element $s[i]$ is 0 if $c[i]$ is at its first or last value, and otherwise $s[i]$ is positive if $c[i]$ is rising and negative if it is falling and its absolute value is equal to $last(i) - min(i) + 1$. Initially $e[j] = j$ for $0 \leqslant j \leqslant m$ and $s[j] = p[j] = 0$ for $1 \leqslant j \leqslant m$ because only $c[1]$ can start by being a tory. If $n = m(k-1)$, then $c[j] = j$ for $1 \leqslant j \leqslant m$; otherwise $c[j] = j + 1$ for $1 \leqslant j \leqslant m$. Finally, $Odd = 0$ unless $n = m(k-1) + 1$ because

```
Procedure Next
  i:=e[m];
  if i=1 then MN:=1 else MN=c[i-1]+1 end if; {MN is the minimum value of c[i]}
  MX:=M0+k*i; { MX is the maximum value of c[i]}
  if s[i]=0 then { c[i] is at its first value }
    p[i]:=Odd; { parity of total number of tories }
    s[i]:=1; {c[i] starts rising unless it starts at max(i)}
    if c[i]=MX then p[i]=1-p[i]; s[i]:=-s[i] end if; {one of these tories is not to c[i]'s left}
    if (k=2) and (i<m) and (c[i+1]=MX+2) then p[i]=1-p[i] end if {see above comment}
  end if;
  if s[i]>0 then { c[i] is rising }
    if c[i]=MN then {MN is taken and c[i] can't end there}
      s[i]:=2
    else
      if (c[i]=MN+1) and (s[i]=2) then {MN+1 is also taken}
        s[i]:=3
      end if
    end if;
    if (c[i] mod 2 = p[i]) and (c[i]<MX-1) then
      c[i]:=c[i]+2
    else
      c[i]:=c[i]+1
    end if;
    if c[i]=MX then Odd:=1-Odd; s[i]:=-s[i] end if {one more tory}
  else { c[i] is falling }
    if c[i]=MX then Odd:=1-Odd end if; {one fewer tory}
    if (c[i] mod 2 ≠ p[i]) and (c[i]>MN+1) then
      c[i]:=c[i]-2
    else
      c[i]:=c[i]-1
    end if
  end if;
  e[m]:=m; {beginning to update Ehrlich array}
  if c[i]+s[i]=MN-1 then {c[i] is at its last value}
    s[i]:=0; {c[i] will be at its first value the next time i is the pivot}
    e[i]:=e[i-1]; e[i-1]:=i-1
  end if
end Next.
```

Fig. 2. A loop-free implementation of a two-close Gray code for k-suffixes ($k \geqslant 2$).

in this case $c[1]$ starts at $max(1) = 2$. The main program processes the current array c and calls the procedure *Next*, shown in Fig. 2, as long as the pivot $e[m]$ does not drop below the smallest index of an array element that can change value. If $n > m(k-1)$, then the execution terminates when $e[m]$ drops to 0. If $n = m(k-1)$, then $c[1]$ is always 1; so the execution terminates when $e[m]$ drops to 1.

This algorithm has been implemented in C and tested. For $k = 2, n = m = 5$, it gave the following list of 1-vectors of length 10 binary Dyck words: 12345, 12346, 12348, 12349, 12347, 12367, 12368, 12369, 12379, 12378, 12358, 12359, 12357, 12356, 12456, 12458, 12459, 12457, 12467, 12468, 12469, 12479, 12478, 12578, 12579, 12569, 12568, 12567, 13567, 13568, 13569, 13579, 13578, 13478, 13479, 13469, 13468, 13467, 13457, 13459, 13458, 13456.

4. The case when $k = 1$

As announced in the introduction, we have found the first two-close Gray code for k-suffixes, a generalization of k-ary Dyck words, and given it a loop-free implementation for every $k \geqslant 2$. For $k = 1$ the list of suffixes is not strictly partitioned; but there already exist two two-close Gray codes for 1-suffixes (combinations), Ruskey's [15] and Chase's [3], and Chase provided a loop-free implementation of his Gray code in 0-vector form as a FORTRAN program that requires no auxiliary array. The reader is invited to examine Fig. 1, compare Chase's Gray code with Ruskey's (the similarity is not quite as obvious when n is odd) and try to obtain a non-recursive description of Chase's Gray code in 0-vector form as a suffix-partitioned list. In [19] we present a non-recursive description of both Chase's and Ruskey's Gray codes in 0-vector form as suffix-partitioned lists, provide a pseudocode for our implementation of Chase's Gray code and prove that our non-recursive description of Ruskey's Gray code is equivalent to his recursive one. To prove that our description of Chase's Gray code is equivalent to his FORTRAN program we had to trace every path in his program. Such a proof is too tedious to be included even in a technical report; however a rough outline is available from the second author on request.

Note added in proof

A complete proof (in French), obtained by the second author's M.Sc. student Mohamed Abdo after the acceptance of this article, is also available.

References

[1] J.R. Bitner, G. Ehrlich, E.M. Reingold, Efficient generation of the binary reflected Gray code and its applications, Comm. ACM 19 (1976) 517–521.

[2] B. Bultena, F. Ruskey, An Eades–McKay algorithm for well-formed parentheses strings, Inform. Process. Lett. 68 (1998) 255–259.

[3] P. Chase, Combination generation and graylex ordering, in: Proceedings of the 18th Manitoba Conference on Numerical Methods and Computing, Winnipeg, 1988, Congressus Numerantium 19 (1989) 215–242.

[4] P. Eades, B. McKay, An algorithm for generating subsets to fixed size with a strong minimal interchange property, Inform. Process. Lett. 19 (1984) 131–133.
[5] G. Ehrlich, Loopless algorithms for generating permutations, combinations, and other combinatorial configurations, J. ACM 20 (1973) 500–513.
[6] F. Gray, Pulse code communication, U.S. Patent 2 632 058, March 17, 1953.
[7] T.A. Jenkyns, D. McCarthy, Generating all k-subsets of $\{1, \ldots, n\}$ with minimal changes, Ars Combinatoria 40 (1995) 153–159.
[8] D.A. Klarner, Correspondences between plane trees and binary sequences, J. Combin. Theory 9 (1970) 401–411.
[9] G. Labelle, Une nouvelle démonstration combinatoire des formules d'inversion de Lagrange, Adv. Math. 42 (1981) 217–247.
[10] C.N. Liu, D.T. Tang, Algorithm 452, Enumerating M out of N objects, Comm. ACM 16 (1973) 485.
[11] J.M. Pallo, R. Racca, A note on generating binary trees in A-order and B-order, Internat. J. Comput. Math. 18 (1) (1985) 27–39.
[12] D. Roelants van Baronaigien, A loopless Gray-code algorithm for listing k-ary trees, J. Algorithms 35 (2000) 100–107.
[13] F. Ruskey, Generating t-ary trees lexicographically, SIAM J. Comput. 7 (1978) 424–439.
[14] F. Ruskey, Adjacent interchange generation of combinations, J. Algorithms 9 (1988) 162–180.
[15] F. Ruskey, Simple combinatorial Gray codes constructed by reversing sublists, in: Lecture Notes in Comput. Sci., vol. 762, Springer, Berlin, 1993, pp. 201–208.
[16] F. Ruskey, A. Proskurowski, Generating binary trees by transpositions, J. Algorithms 11 (1990) 68–84.
[17] V. Vajnovszki, A loopless algorithm for generating the permutations of a multiset, Theoret. Comput. Sci. 307 (2003) 415–431.
[18] V. Vajnovszki, http://www.u-bourgogne.fr/v.vincent.
[19] V. Vajnovszki, T.R. Walsh, A loopless two-close Gray-code algorithm for listing k-ary Dyck words, Rapport de recherche No. 03-01, Département d'Informatique, Université du Québec à Montréal, March 2003.
[20] T.R. Walsh, Generation of well-formed parenthesis strings in constant worst-case time, J. Algorithms 29 (1998) 165–173.
[21] T.R. Walsh, Gray codes for involutions, JCMCC 36 (2001) 95–118.
[22] T.R. Walsh, Generating Gray codes in $O(1)$ worst-case time per word, in: DMTCS 2003, in: Lecture Notes in Comput. Sci., vol. 2731, Springer, Berlin, 2003, pp. 73–88, Invited paper.
[23] E.T. Whittaker. G.N. Watson, A Course of Modern Analysis, Cambridge, 1940.
[24] S. Zaks, Generation and ranking of k-ary trees, Inform. Process. Lett. 14 (1982) 44–48.