

Loopless Generation of Multiset Permutations using a Constant Number of Variables by Prefix Shifts

Aaron Williams *

Abstract

This paper answers the following mathematical question: Can multiset permutations be ordered so that each permutation is a prefix shift of the previous permutation? Previously, the answer was known for the permutations of any set, and the permutations of any multiset whose corresponding set contains only two elements. This paper also answers the following algorithmic question: Can multiset permutations be generated by a loopless algorithm that uses sublinear additional storage? Previously, the best loopless algorithm used a linear amount of additional storage. The answers to these questions are both yes.

1 Introduction

The research conducted in this paper falls under the category of combinatorial generation. The area is so important to computer science that Knuth has dedicated over 400 pages to the subject in his upcoming volume of *The Art of Computer Programming* [27, 28]. The research area is applicable whenever it is necessary to efficiently consider every possible object of a particular type, such as binary strings of length n , permutations of $\{1, 2, \dots, n\}$, binary trees with n nodes, linear extensions of a partially-ordered set, spanning trees of a directed graph, or perfect elimination orders of a chordal graph.

The most useful results in combinatorial generation tend to have a mathematical aspect and an algorithmic aspect. For example, two of the most well-known results in combinatorial generation are the binary reflected Gray code [24] and the de Bruijn cycle [22]. Both results provide a clever order for the binary strings of length n . The binary reflected Gray code provides an order in which each successive string can be obtained from the previous by changing the value of a single bit, while de Bruijn cycles provide an order in which each successive string can be obtained from the previous by removing the rightmost bit and inserting a new leftmost bit. In general, the mathematical aspect of combinatorial generation involves the discovery of a

minimal-change order. A minimal-change order is an order in which each successive object can be obtained from the previous by making one small modification of a certain type. The existence or non-existence of minimal-change orders depend upon the type of object and the type of modification. New results in this area are often quite difficult to find, but the results that are found tend to be elegant and simple. The mathematical question answered in this paper is the following.

QUESTION 1. Can multiset permutations be ordered so that each permutation is a prefix shift of the previous permutation?

Given a string $\mathbf{s} = s_1 s_2 \dots s_n$, a *prefix shift of length k* is denoted by $\sigma_k(\mathbf{s})$ and is the result of moving s_k into the leftmost position. That is,

$$\sigma_k(\mathbf{s}) = s_k s_1 \dots s_{k-1} s_{k+1} \dots s_n.$$

For example, the order listed below affirmatively answers Question 1 for the multiset $\{1, 1, 2, 4\}$

$$421\underline{1}, 142\underline{1}, 412\underline{1}, 141\underline{2}, 114\underline{2}, 411\underline{2}, \\ 241\underline{1}, 124\underline{1}, 214\underline{1}, 121\underline{4}, 112\underline{4}, 211\underline{4}.$$

In particular, each successive permutation is obtained from the previous permutation by moving the underlined symbol into the leftmost position. (In this case a prefix shift also changes the last permutation into the first permutation, and so the listed order is considered a *circular* minimal-change order with respect to prefix shifts.)

The algorithmic aspect of combinatorial generation involves the creation of algorithms for generating all possible objects of a particular type. For the sake of efficiency, algorithms often use the idea of a *single shared object*. This means that the generating algorithm and the user of the generating algorithm share a section of memory that stores one object of the type being generated. The generating algorithm modifies this single shared object and informs the user when a new object is ready for use. This allows the user of the generating algorithm to *visit* each instance without an entire list being constructed. If the generating algorithm

*Department of Computer Science, University of Victoria, PO Box 3010 STN CSC, Victoria BC, V8W 3N4, Canada

can always perform its modifications in $O(1)$ -time then it is said to be a *loopless* algorithm. Loopless algorithms have been the subject of a great amount of study since being introduced by Ehrlich [23]. While loopless algorithms are best-possible in a theoretical sense, in practice it is also important that these algorithms are simple. In particular, a complex loopless algorithm will run no faster than a simple *constant amortized time*. An algorithm runs in constant amortized time if which the average amount of time taken to perform each modification is $O(1)$.

Although it is now commonplace for the consumed time of generating algorithms to be measured using the single shared object model, the consumed memory has been less frequently measured using this model. In particular, the memory used to store the single shared object should not be counted against the generating algorithm's consumed memory since it can be seen as an expense for the user of the generating algorithm, just like the contents of each `visit` call. One reason for this discrepancy is that efficient generating algorithms often use at least as much *additional memory* as is required to store the single shared object. For example, recursive algorithms typically use an amount of stack memory that is proportional to the amount of memory used to store the single shared object. On the other hand, an iterative algorithm using a constant number of *additional variables* may end up consuming less memory than is required to store the single shared object. If the shared single object is assumed to require linear storage, then these algorithms can be said to use *sublinear additional storage*, or simply *sublinear storage*. Loopless algorithms using sublinear storage have been shown to exist for some combinatorial objects, including balanced parentheses and binary trees [5], but not for multiset permutations.

QUESTION 2. Can the permutations of any multiset be generated by a loopless algorithm that uses sublinear storage?

The generating algorithm presented in this paper answers Question 2 in the affirmative. In particular, the algorithm uses only four pointers to repeatedly modify the shared singly-linked list containing $O(n)$ nodes, each of which contains a value in the multiset permutation and a next pointer. (In fact, the algorithm can be modified to use only two pointers, although the result is more complicated and less efficient.) Furthermore, the order in which the multiset permutations are generated also affirmatively answers Question 1.

Before proceeding with the rest of the paper, it is useful to add one more comment into the discussion of combinatorial generation. Users of combinatorial gener-

ation algorithms often compute an associated *value* for each object with the aim of solving a particular optimization problem. Depending on the type of minimal-change order being used by the generating algorithm, and the nature of the problem being solved by the user, these associated values may also appear in a minimal-change order. In situations like these, the user of the combinatorial generation algorithm should be informed of the modification that is made in creating each successive object, since that allows the user to avoid scanning the object to infer the modification. In particular, the user may be able to update the associated values in sub-linear time. The potential for simultaneous generating and evaluating efficiency is one of the primary motivations for having different minimal-change orders of the same objects.

1.1 Applications Efficient algorithms for generating multiset permutations have a number of applications. If the multiset is simply a set, then applications include communication in point-to-point multiprocessor networks [21]. If the multiset's corresponding set contains only two elements, then applications include cryptography (where orders have been implemented in hardware at NSA), genetic algorithms, software and hardware testing, statistical computation (e.g., for the bootstrap, and Diaconis and Holmes [18]).

Minimal-change orders also tend to have diverse applications. For example, the binary reflected Gray code was designed at Bell Labs for telephone systems, but has since found applications in information and communication technology, analog-to-digital conversion, error correction, and decreased power consumption in handheld devices. It has also been used in the CODACON spectrometer, and appears in research titles ranging from measurement and instrumentation [1] to quantum chemistry [20]. The minimal-change order discovered in this paper has potential applications in genetics since prefix shifts are akin to splicing segments of genetic material.

1.2 Previous Results The history of combinatorial generation is rich and fascinating. The reader is directed towards [28, 27, 29] and [31] for excellent treatments of the subject. In terms of minimal-change orders for multiset permutations, the most relevant previous results are found in [21, 9, 4] and [3] (with earlier version [2]). The first three results provide minimal-change orders for set permutations by prefix shifts, however, generalizing these orders to the permutations of multisets has so far proved impossible. For example, the order found in [4] uses only prefix shifts of length n and $n - 1$, and thereby creates the first explicit *shorthand universal cy-*

cle for set permutations, which is essentially a de Bruijn cycle for set permutations (c.f. [14, 25]). Unfortunately, the existence of shorthand universal cycles for multiset permutations is still open. On the other hand, the fourth result uses prefix shifts to generate the permutations of multisets whose corresponding set contains two elements. Such multisets are also referred to as *combinations*. The order presented in this paper generalizes the order found in [3], and in fact, the order for balanced parentheses found in [5] is a suborder of these orders. Collectively, these orders form a hierarchy of *cool-lex* orders (named for their similarity to *co-lexicographic* or *co-lex* order) and are the topic of this author’s upcoming thesis.

Besides prefix shifts, another well-studied modification is an adjacent transposition. Given a string $s = s_1s_2 \dots s_n$, an *adjacent transposition* results in a string of the form

$$s_1 \dots s_{i-1}s_{i+1}s_i s_{i+2} \dots s_n.$$

The beautiful (and oft-rediscovered) Steinhaus-Johnson-Trotter order [26] proves that a minimal-change order using adjacent transpositions exists for set permutations. On the other hand, the same is not true for multiset permutations, with [30] providing exact conditions for their existence. There are also minimal-change orders for multiset permutations using (non-adjacent) transpositions [7].

Many efficient algorithms for generating multiset permutations can be found in the literature [33, 35]. However, no previous algorithm is loopless while using sublinear storage. Loopless algorithms using a linear number of additional variables (with respect to the size of the multiset) do exist for implementations that store the current permutation in an array [16] (which answered a conjecture in [33]) or a linked list [19]. The most common approach in these papers is to combine the Steinhaus-Johnson-Trotter order for permutations with a minimal-change order for combinations. Unfortunately, the challenge of simultaneously creating both orders results in programs that are decidedly more complicated than the one presented in this paper. (In fact, Algorithm 1 is one of the simplest ever created for generating multiset permutations, and can be implemented without reading the remainder of this document.) Loopless algorithms also exist for generating linear extensions of partially ordered sets, which include multiset permutations as a special case [8, 17, 15].

1.3 Outline Section 2 introduces notation and defines a simple lexicographic order for the permutations of a multiset. This lexicographic order is modified in Section 2.3 to create the new minimal-change order that

answers Question 1. This new minimal-change order is then used in Section 3 to create the algorithm that answers Question 2. Besides answering the two main questions, Section 3.1 also provides an interesting analysis of the algorithm when applied to set permutations. Specifically, it is shown that the average length of the prefix shifts performed in the minimal-change order is less than 3. Section 4 concludes with open problems.

2 Multiset Permutations

2.1 Notation and Conventions There are two main ways to describe the elements of a multiset. Every element in the multiset can be stated, or every element in its corresponding set can be stated along with its frequency. For example, the multiset $\{1, 1, 2, 4, 4, 4\}$ can be described by its elements 1, 1, 2, 4, 4, 4 or by noting that it contains two copies of 1, one copy of 2, and three copies of 4. Every multiset in this paper is assumed to contain integer values, and so it is also possible to specify the cumulative frequencies of the elements less than or equal to a certain value. For example, $\{1, 1, 2, 4, 4, 4\}$ can also be specified by noting that it contains two elements ≤ 1 , three elements ≤ 2 , and six elements ≤ 4 .

Throughout this document, \mathbb{E} is used to represent a multiset and it will be assumed that \mathbb{E} contains n elements, and its corresponding set has m elements. Furthermore, e_i is used for the i th smallest element in the multiset (for $1 \leq i \leq n$), and d_i is used for the i th smallest element in its corresponding set (for $1 \leq i \leq m$) with f_i giving the frequency of d_i . For cumulative frequencies, \vec{f}_t represents the number of elements in \mathbb{E} that are less than or equal to d_t . Thus, if $\mathbb{E} = \{1, 1, 2, 4, 4, 4\}$ then $n = 6$, $m = 3$, and

$$\begin{aligned} \{e_1, e_2, e_3, e_4, e_5, e_6\} &= \{1, 1, 2, 4, 4, 4\} \\ \{d_1, d_2, d_3\} &= \{1, 2, 4\} \\ f_1, f_2, f_3 &= 2, 1, 3 \\ \vec{f}_1, \vec{f}_2, \vec{f}_3 &= 2, 3, 6 \end{aligned}$$

Given multiset \mathbb{E} , the set of permutations of \mathbb{E} is denoted by $M_{\mathbb{E}}$. Each permutation is treated as a string so that certain string-related concepts are natural. For example, the terms *prefix* and *suffix* are used along with the adjective *proper* to describe a prefix or suffix that is not equal to the entire string. *Concatenation* is used to add symbols to the end of strings, and is represented by adjacent symbols or by “.”. In particular, concatenation can be applied to the end of every string in a list. For example, given list $\mathcal{L} = 42, 24$, then

$$\mathcal{L} \cdot 11 = 42 \cdot 11, 24 \cdot 11 = 4211, 2411.$$

While the term concatenation and \cdot are reserved for making strings longer, the term *append* and “,” are

reserved for making lists longer. For example, if $\mathcal{L}_1 = 1222, 2122$ and $\mathcal{L}_2 = 2212$ then

$$\mathcal{L}_1, \mathcal{L}_2, 2221 = 1222, 2122, 2212, 2221.$$

The symbol \bigoplus is also used for automating the process of appending lists. If the initial value of the index variable is below the \bigoplus symbol, then the index variable counts upwards (as usual). If the initial value of the index variable is above the \bigoplus symbol, then index variable counts downwards. For example,

$$\bigoplus_{i=1}^3 \mathcal{L}_i = \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3 \quad \bigoplus_{i=3}^1 \mathcal{L}_i = \mathcal{L}_3, \mathcal{L}_2, \mathcal{L}_1.$$

Lists are used to in this paper for describing orders of $M_{\mathbb{E}}$. Thus, lists will be constructed to contain each string in $M_{\mathbb{E}}$ exactly once.

The symbol \setminus is used to represent set difference in terms of multisets. Furthermore, the symbol $-$ is used as a shorthand for removing all of the symbols of a string from a multiset. For example,

$$\{1, 2, 2, 3, 3, 3\} - 223 = \{1, 2, 2, 3, 3, 3\} \setminus \{2, 2, 3\} \\ = \{1, 3, 3\}.$$

The *tail* of a multiset is the string containing its elements in non-increasing order.

DEFINITION 2.1. (TAIL)

$$\text{tail}(\mathbb{E}) = e_n e_{n-1} \dots e_1$$

This moniker is used so that $\text{tail}(\mathbb{E})$ can be compared with other strings called *scuts*, which are defined in the next section.

2.2 Lexicographic Order *Co-lex order* for multiset permutations orders the strings of $M_{\mathbb{E}}$ in increasing lexicographic order when the strings are read from right-to-left. $\mathcal{L}_{\mathbb{E}}$ is used to denote co-lex order for $M_{\mathbb{E}}$. For example,

$$\mathcal{L}_{\{1,1,2,4\}} = 4211, 2411, 4121, 1421, 2141, 1241, \\ 4112, 1412, 1142, 2114, 1214, 1124.$$

Recursively, the most natural way to describe $\mathcal{L}_{\mathbb{E}}$ is by the value of the rightmost symbol which appear in bold above.

DEFINITION 2.2. (CO-LEX BY RIGHTMOST SYMBOL)

$$\mathcal{L}_{\mathbb{E}} = \bigoplus_{i=1}^m \mathcal{L}_{\mathbb{E}-d_i} \cdot d_i$$

where $\mathcal{L}_{\{a\}} = a$ for the single symbol a .

$\mathcal{L}_{\mathbb{E}}$ can also be recursively defined in a less natural, but ultimately more useful manner. In English, a *scut* is a short thick tail found on an animal such as a deer or rabbit. In the context of this paper, every multiset permutation that is not equal to $\text{tail}(\mathbb{E})$ has some shortest suffix that is not a suffix of $\text{tail}(\mathbb{E})$; this suffix is referred to as the *scut* of the permutation. For example, the scuts are bolded in the restatement of $\mathcal{L}_{\{1,1,2,4\}}$ below

$$\mathcal{L}_{\{1,1,2,4\}} = 4211, 241\mathbf{1}, 412\mathbf{1}, 142\mathbf{1}, 214\mathbf{1}, 124\mathbf{1}, \\ 411\mathbf{2}, 141\mathbf{2}, 114\mathbf{2}, 211\mathbf{4}, 121\mathbf{4}, 112\mathbf{4}.$$

Notice that the scuts appear in the following order: 411, 21, 41, 2, 4.

Every scut can be written as $d_j e_k e_{k-1} \dots e_1$ where $d_j > e_{k+1}$. This idea is formalized by the following definition.

DEFINITION 2.3. (SCUT) *If $d_j > e_{k+1}$ then*

$$\text{scut}(j, k) = d_j e_k e_{k-1} \dots e_1.$$

(Note: The condition $d_j > e_{k+1}$ is equivalent to $2 \leq j \leq m$ and $0 \leq k \leq f_{j-1} - 1$, and this alternate expression is more useful when providing Definition 2.4.)

$\mathcal{L}_{\mathbb{E}}$ can be viewed as the order that starts with $\text{tail}(\mathbb{E})$, and then orders the remaining strings by decreasing values of k followed by increasing values of j , with respect to $\text{scut}(j, k)$. (The base case occurs when $m = 1$ and $\mathcal{L}_{\mathbb{E}} = \text{tail}(\mathbb{E})$.) For example, recall that in the list $\mathcal{L}_{\{1,1,2,4\}}$ given above, the scuts appeared in the following order: 411, 21, 41, 2, 4. In other words, they are ordered by decreasing length and then by increasing leftmost symbol. The minimal-change order defined in the next section is a slight perturbation of this alternate view of co-lex order. In particular, $\text{tail}(\mathbb{E})$ is ordered last (instead of first), and then the remaining strings are ordered by increasing values of j followed by decreasing values of k (instead of vice-versa).

2.3 Cool-lex Order This section defines a new order of $M_{\mathbb{E}}$. The order is referred to as the *cool-lex order* for multiset permutations and is denoted by $\mathcal{C}_{\mathbb{E}}$. The term *cool-lex* is a modification of the term *co-lex* and is further justified at the end of this section.

DEFINITION 2.4. (COOL-LEX BY SCUT)

$$\mathcal{C}_{\mathbb{E}} = \bigoplus_{j=2}^m \bigoplus_{k=\overrightarrow{f_{j-1}-1}}^0 \mathcal{C}_{\mathbb{E}-\text{scut}(j,k)} \cdot \text{scut}(j, k), \text{tail}(\mathbb{E})$$

The base case occurs when $m = 1$ and $\mathcal{C}_{\mathbb{E}} = \text{tail}(\mathbb{E})$.

which maps any string \mathbf{s} into the string that follows \mathbf{s} within $\mathcal{C}_{\mathbb{E}}$. (The symbol \triangleleft was chosen because the operation uses prefix shifts to move a single symbol into the leftmost position.)

DEFINITION 2.7. (\triangleleft) Let $\mathbf{s} = s_1 \dots s_n$ and $i = \searrow(\mathbf{s})$. Then,

$$\triangleleft(\mathbf{s}) = \begin{cases} \sigma_{i+1}(\mathbf{s}) & \text{if } i \leq n-2 \text{ and } s_{i+2} > s_i \\ \sigma_{i+2}(\mathbf{s}) & \text{if } i \leq n-2 \text{ and } s_{i+2} \leq s_i \\ \sigma_n(\mathbf{s}) & \text{otherwise (if } i \geq n-1). \end{cases}$$

Notice that the definition is incredibly simple: To obtain the permutation that follows \mathbf{s} , either a prefix shift of length $i+1$ or $i+2$ or n will be performed, where $i = \searrow(\mathbf{s})$. Moreover, at most two comparisons are needed to determine which length of prefix shift to perform. To illustrate the definition,

$$\begin{aligned} \triangleleft(6442313134) &= \triangleleft(6442 \cdot 31 \cdot 3134) \\ &= \triangleleft(64423 \cdot 1 \cdot 3134) \\ &= 1644233134 \\ \triangleleft(6442343131) &= \triangleleft(6442 \cdot 34 \cdot 3131) \\ &= \triangleleft(6442 \cdot 3 \cdot 43131) \\ &= 3644243131 \end{aligned}$$

where the \cdot are used to visually separate the prefix of length $i = \searrow(\mathbf{s})$ and the symbols s_{i+1} and s_{i+2} . The reader can also verify that Definition 2.7 properly describes the string-by-string behavior found in Figure 2. One important aspect of \triangleleft is that it does not change certain suffixes. In particular, the following lemma shows the relationship between \triangleleft and the last string in $\mathcal{C}_{\mathbb{E}}$.

LEMMA 2.2. (INVARIANT FOR \triangleleft) Suppose $\mathbf{s} = \mathbf{p} \cdot \mathbf{z}$, $\mathbf{s} \in \mathcal{M}_{\mathbb{E}}$, $\mathbf{p} \neq \text{last}(\mathcal{C}_{\mathbb{E}-\mathbf{z}})$, and \mathbf{z} is the scut of \mathbf{s} . Then,

$$\triangleleft(\mathbf{s}) = \triangleleft(\mathbf{p}) \cdot \mathbf{z}.$$

To describe multiple applications of \triangleleft let $\triangleleft^0(\mathbf{s}) = \mathbf{s}$ and

$$\triangleleft^k(\mathbf{s}) = \triangleleft(\triangleleft^{k-1}(\mathbf{s}))$$

for all $k > 0$. For example, $\triangleleft^3(\mathbf{s}) = \triangleleft(\triangleleft(\triangleleft(\mathbf{s})))$, which would result in the third string following \mathbf{s} within $\mathcal{C}_{\mathbb{E}}$. The main result of this section is stated in Theorem 2.1.

THEOREM 2.1. (EQUIVALENCE OF DEFINITIONS)

$$\mathcal{C}_{\mathbb{E}} = \bigoplus_{k=0}^{|\mathcal{M}_{\mathbb{E}}|-1} \triangleleft^k(\text{tail}(\mathbb{E})).$$

In other words, Theorem 2.1 states that \triangleleft does in fact provide an iterative description of $\mathcal{C}_{\mathbb{E}}$. The result can be proven by induction on n using Lemmas 2.1 and 2.2. In particular, one must show that \triangleleft does not alter any given scut until the last string in the order with that scut is reached, and that one more application of \triangleleft produces the first string in the order with the next scut. Since \triangleleft always performs a single prefix shift, a simple corollary to Theorem 2.1 is that $\mathcal{C}_{\mathbb{E}}$ provides a constructive and affirmative answer to Question 1. Theorem 2.1 can also be strengthened slightly by the following lemma, which shows why \triangleleft circularly generates $\mathcal{C}_{\mathbb{E}}$. This slight extension is used in the statement of Theorem 3.1 found in Section 3.1.

LEMMA 2.3. (CIRCULAR GENERATION)

$$\triangleleft(\text{last}(\mathcal{C}_{\mathbb{E}})) = \text{first}(\mathcal{C}_{\mathbb{E}})$$

To conclude this section, it is noted that \triangleleft actually provides a subtle generalization of the iterative rule for generating multiset permutations with $m = 2$ originally described in [3, 2]. The order in that paper is described as the *cool-lex order for combinations*, and so $\mathcal{C}_{\mathbb{E}}$ is referred to as the *cool-lex order for multiset permutations*.

3 Algorithms

This section describes an algorithm that generates every multiset permutation for any specified multiset \mathbb{E} . The algorithm is loopless, iterative (i.e. not recursive), uses a constant number of additional variables, and generates the permutations in the order given by $\mathcal{C}_{\mathbb{E}}$. (To be more precise, the algorithm generates the permutations in the order given by $\mathcal{C}_{\mathbb{E}}$, with the one exception that $\text{tail}(\mathbb{E})$ is generated first instead of last.)

There is one hurdle in translating the iterative description of $\mathcal{C}_{\mathbb{E}}$ from Definition 2.7, into such an algorithm: For each successive permutation \mathbf{s} , the algorithm must determine the value of $i = \searrow(\mathbf{s})$ in constant time, and without the use of any complex data structures. Fortunately, the value of i for the current permutation is strongly related to the value of i for the previous permutation.

LEMMA 3.1. Let $\mathbf{s} = s_1 s_2 \dots s_n$ and $\triangleleft(\mathbf{s}) = \mathbf{s}' = s'_1 s'_2 \dots s'_n$. Then

$$\searrow(\triangleleft(\mathbf{s})) = \begin{cases} 1 & \text{if } s'_1 < s'_2 \\ \searrow(\mathbf{s}) + 1 & \text{otherwise (if } s'_1 \geq s'_2). \end{cases}$$

In other words, after each application of \triangleleft , the length of the longest non-increasing prefix is either increased by one or is reset to the value of 1, and the

correct value can be determined by a single comparison. The proof of this lemma follows relatively easily from Definition 2.7.

Algorithm 1 Visits the permutations of multiset \mathbb{E} . The permutations are stored in a singly-linked list pointed to by head pointer \mathbf{h} . Each node in the linked list has a value field v and a next field n . The `init(\mathbb{E})` call creates a singly-linked list storing the elements of \mathbb{E} in non-increasing order with \mathbf{h} , \mathbf{i} , and \mathbf{j} pointing to its first, second-last, and last nodes, respectively. The null pointer is given by ϕ . Note: If \mathbb{E} is empty, then `init(\mathbb{E})` should exit. Also, if \mathbb{E} contains only one element, then `init(\mathbb{E})` does not need to provide a value for \mathbf{i} .

```

[h, i, j] ← init( $\mathbb{E}$ )
visit(h)
while j.n ≠  $\phi$  or j.v < h.v do
  if j.n ≠  $\phi$  and i.v ≥ j.n.v then
    s ← j
  else
    s ← i
  end if
  t ← s.n
  s.n ← t.n
  t.n ← h
  if t.v < h.v then
    i ← t
  end if
  j ← i.n
  h ← t
  visit(h)
end while

```

Instead of visiting `tail(\mathbb{E})` last (as described in Definition 2.4) the algorithm visits `tail(\mathbb{E})` first. The algorithm then continues until `tail(\mathbb{E})` is encountered again. For this reason, the first iteration is somewhat of a special case with `init(\mathbb{E})` initializing \mathbf{i} to be “off-by-one”. After the first iteration, \mathbf{i} points to the last node in the multiset permutation’s non-increasing prefix and \mathbf{j} points to the next node. Variables \mathbf{s} and \mathbf{t} are also pointer variables and are used for performing each shift. The first three iterations of Algorithm 1 are illustrated in Figure 3. As mentioned in Section 1, it is often useful for generating algorithms to inform their users of the modifications being made during each iteration, and Algorithm 1 can be modified fairly easily to fit into this style. Clearly Algorithm 1 is loopless because its loop contains a constant number of elementary instructions including a single `visit` call. It also uses sublinear storage since it introduces only four additional pointer variables, namely \mathbf{i} , \mathbf{j} , \mathbf{s} , and \mathbf{t} . (The number of additional variables can be reduced from four to two at

the expense of simplicity and efficiency.) Therefore, the algorithm provides an affirmative answer to Question 2.

3.1 Analysis Let $\langle n \rangle$ represent the set $\{1, 2, \dots, n\}$. In this section the average length of the prefix shift performed within $\mathcal{C}_{\langle n \rangle}$ is analyzed. Remarkably, the value is found to be less than three. To formalize the result, let $\pi(\mathbf{s})$ be the length of the prefix shift performed in $\mathcal{C}_{\mathbb{E}}$ when making the transition from \mathbf{s} to $\triangleleft(\mathbf{s})$. (Notice that the statement of Theorem 3.1 includes the application of \triangleleft that maps the last permutation of $\mathcal{C}_{\langle n \rangle}$ back into the first permutation of $\mathcal{C}_{\langle n \rangle}$.)

THEOREM 3.1. (SHORT PREFIX SHIFTS)

$$\sum_{i=0}^{n!} i \cdot \pi(\triangleleft^i(\text{tail}(\langle n \rangle))) < 3 \cdot n!$$

The proof of this theorem requires two lemmas. The first lemma is an interesting combinatorial identity, and the second counts the number of permutations that require a prefix shift of length k while generating $\mathcal{C}_{\langle n \rangle}$. These lemmas are stated and proven, and then combined to prove the theorem.

LEMMA 3.2. (COMBINATORIAL IDENTITY)

$$2(n^2 - n + 1) + \sum_{k=2}^{n-1} k \cdot \frac{n! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} = 3 \cdot n!$$

when $n \geq 2$.

Proof. The proof is by induction on n . When $n = 2$,

$$2(n^2 - n + 1) + \sum_{k=2}^{n-1} k \cdot \frac{n! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} = 2(4 - 2 + 1) = 3 \cdot 2!$$

So assume the identity is true when $n = x$, and let us use that assumption to prove that it is true when $n = x + 1$. The important steps in the following derivation are separating the $k = x$ term within the sum, factoring $(x + 1)$ out of each term in the remaining sum, and adding and subtracting the value of $2(x^2 - x + 1)$ in

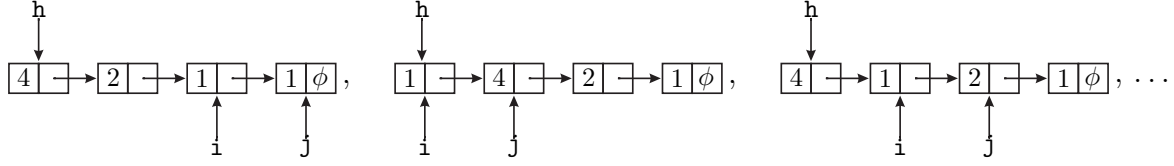


Figure 3: The first three $\text{visit}(\mathbb{E})$ calls in Algorithm 1 for $\mathbb{E} = \{1, 1, 2, 4\}$ will produce the configurations given above, where the left and right boxes of each node refer to its value v and next n fields, respectively.

order to apply the inductive hypothesis.

$$\begin{aligned}
& 2((x+1)^2 - (x+1) + 1) + \\
& \sum_{k=2}^x k \cdot \frac{(x+1)! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \\
& = 2(x^2 + x + 1) + x \cdot \frac{(x+1)! \cdot 2 \cdot (x^2 - x - 1)}{(x+1)!} + \\
& \sum_{k=2}^{x-1} k \cdot \frac{(x+1)! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \\
& = 2(x^3 + 1) + \sum_{k=2}^{x-1} k \cdot \frac{(x+1)! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \\
& = 2(x^3 + 1) + (x+1) \cdot \sum_{k=2}^{x-1} k \cdot \frac{x! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \\
& = 2(x^3 + 1) + (x+1) \cdot \left(2(x^2 - x + 1) - \right. \\
& \quad \left. 2(x^2 - x + 1) + \sum_{k=2}^{x-1} k \cdot \frac{x! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \right) \\
& = 2(x^3 + 1) + (x+1) \cdot \left(2(x^2 - x + 1) + \right. \\
& \quad \left. \sum_{k=2}^{x-1} k \cdot \frac{x! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} - 2(x^2 - x + 1) \right) \\
& = 2(x^3 + 1) + (x+1)(3 \cdot x! - 2(x^2 - x + 1)) \\
& = 3 \cdot (x+1)!
\end{aligned}$$

LEMMA 3.3. (STRINGS FOR EACH VALUE OF $\pi(\mathbf{s})$)

$$|\{\mathbf{s} : \pi(\mathbf{s}) = k\}| = \begin{cases} \frac{n! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} & \text{if } 2 \leq k < n \\ 2n - 2 & \text{otherwise (if } k = n). \end{cases}$$

Proof. When $\pi(\mathbf{s}) = n$ then there are three possibilities to consider. First, it might be that $\Delta(\mathbf{s}) = n$, and then $\pi(\mathbf{s}) = n$ is guaranteed. This can happen only if $\mathbf{s} = \text{tail}(\mathbb{E})$. Second, it might be that $\Delta(\mathbf{s}) = n - 1$, and then again $\pi(\mathbf{s}) = n$ is guaranteed. This can happen in $n - 1$ ways since there are $n - 1$ choices for the value of s_n (it cannot be that $s_n = d_1$) and then the rest of the string is determined by this choice. Third, it might be that $\Delta(\mathbf{s}) = n - 2$ and $\pi(\mathbf{s}) = n$. In order for this

to occur it must be that $s_n = d_1$. Then there are $n - 2$ choices for the value of s_{n-1} (it cannot be d_1 or d_2) and then the rest of the string is determined by this choice. In total there are $1 + (n - 1) + (n - 2) = 2n - 2$ possibilities for \mathbf{s} , thereby proving the $k = n$ case in the stated equality.

When $\pi(\mathbf{s}) = k$ and $2 \leq k < n$ then there are two possibilities. First, it might be that $\Delta(\mathbf{s}) = k - 1$ and $\pi(\mathbf{s}) = k$. In order for this to occur there are $\binom{n}{k+1}$ ways of choosing the first $k + 1$ symbols. Of these symbols, s_{k-1} must be the smallest, and then there are $k(k - 1)$ ways of choosing $s_k s_{k+1}$. Then $s_1 s_2 \dots s_{k-1}$ is uniquely determined since $\Delta(\mathbf{s}) = k - 1$. Finally, there are $n - k - 1$ symbols that are not within the first $k + 1$ symbols, and these can be placed in any order. Therefore, in total there are

$$\begin{aligned}
\binom{n}{k+1} k(k-1)(n-k-1)! &= \frac{n! k(k-1)(n-k-1)!}{(k+1)!(n-k-1)!} \\
&= \frac{n! k(k-1)}{(k+1)!}
\end{aligned}$$

choices for \mathbf{s} in the first possibility. Second, it might be that $\Delta(\mathbf{s}) = k - 2$ and $\pi(\mathbf{s}) = k$. (This possibility cannot occur when $k = 2$, however, the expression obtained below equals zero when $k = 2$.) In order for this to occur there are $\binom{n}{k}$ ways of choosing the first k symbols. Of these symbols, s_k must be the smallest, and then there are $(k - 2)$ ways of choosing s_{k-1} since s_{k-1} can be any of the first k symbols except for the smallest and second-smallest. Then $s_1 s_2 \dots s_{k-2}$ is uniquely determined since $\Delta(\mathbf{s}) = k - 2$. Finally, there are $n - k$ symbols that are not within the first k symbols, and these can be placed in any order. Therefore, in total there are

$$\begin{aligned}
\binom{n}{k} (k-2)(n-k)! &= \frac{n!(k-2)(n-k)!}{k!(n-k)!} \\
&= \frac{n!(k-2)}{k!} \\
&= \frac{n!(k-2)(k+1)}{(k+1)!}
\end{aligned}$$

choices for \mathbf{s} in the second possibility. Therefore, in

total there are

$$\frac{n!k(k-1) + n!(k-2)(k+1)}{(k+1)!} = \frac{n! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!}$$

choices for \mathbf{s} as claimed in the $2 \leq k < n$ case of the stated equality.

Now that Lemmas 3.2 and 3.3 have been proven, the following derivation proves Theorem 3.1

$$\begin{aligned} & \sum_{i=0}^{n!} \pi(\langle^i(\text{tail}(\mathbb{E}))\rangle) \\ &= \left(\sum_{k=2}^{n-1} k \cdot \frac{n! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \right) + n \cdot (2n - 2) \\ &= 2(n^2 - n) + \sum_{k=2}^{n-1} k \cdot \frac{n! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \\ &< 2(n^2 - n + 1) + \sum_{k=2}^{n-1} k \cdot \frac{n! \cdot 2 \cdot (k^2 - k - 1)}{(k+1)!} \\ &= 3 \cdot n!. \end{aligned}$$

4 Open Problems

Several interesting problems arise from the material in this paper.

QUESTION 3. Theorem 3.1 provides an extremely low value for the average length of prefix shift performed within $\mathcal{C}_{\langle n \rangle}$. Is it possible that $\mathcal{C}_{\langle n \rangle}$ provides the lowest possible average length of prefix shift when considering all possible orders of $M_{\langle n \rangle}$ using prefix shifts? Furthermore, can a similar result be proven when an arbitrary multiset \mathbb{E} replaces $\langle n \rangle$?

QUESTION 4. Experimentally, the average value of $\pi(\mathbf{s})$ for $M_{\mathbb{E}}$ depends only upon the multiset of frequencies of \mathbb{E} . For example, the average value of $\pi(\mathbf{s})$ is the same in $\mathcal{C}_{\{1,1,2,2,3,3,3,4\}}$ as it is in $\mathcal{C}_{\{1,1,1,2,3,3,4,4\}}$ since the multiset of frequencies is $\{1, 2, 2, 3\}$ in both cases. Can this result be proven?

QUESTION 5. An important part of combinatorial generation is *ranking*, which provides a mapping between each string and its position in the order. Due to the straight-forward nature of Definition 2.4 it seems plausible that the strings in $\mathcal{C}_{\mathbb{E}}$ could be ranked efficiently. How fast can they be ranked?

QUESTION 6. The iterative rule for combinations in cool-lex order was modified slightly to obtain a minimal-change order for balanced parentheses strings and binary trees [5]. Can the new generalized rule be modified

to obtain minimal-change orders for other interesting combinatorial objects such as fixed-content necklaces? Necklaces are equivalence classes of strings under rotation, and necklaces with fixed-content \mathbb{E} are a subset of $M_{\mathbb{E}}$. Currently no loopless algorithm is known for fixed-content necklaces, although an efficient CAT algorithm does exist [32]. Efficient algorithms for generating unlabeled necklaces [12] and fixed-density necklaces [10, 11] exist. In this last case, transposition Gray codes are also possible [6, 34]. Binary necklaces (with no density restriction) do not have a Gray code changing a single bit [6] but do have a Gray code changing at most two bits [36]. Gray codes changing at most three symbols also exist for unrestricted necklaces over arbitrary bases [13].

QUESTION 7. It has been observed that within $\mathcal{C}_{\mathbb{E}}$ the strings with prefix d_m appear in the same relative order as they do in $\mathcal{C}_{\mathbb{E}-d_m}$. What properties of this type can be proven?

5 Acknowledgements

The author would like to wholeheartedly thank Frank Ruskey who has been very influential to my research, and has been a coauthor on previous publications [2] [3] [4]. The author would also like Don Knuth for kindly making last-minute additions to *The Art of Computer Programming* to account for several recent results. Wendy Myrvold is also thanked for helpful comments on short notice, as is the Rollins family for providing housing during the writing of this paper. Finally, an anonymous referee is thanked for catching several important typographical errors.

References

- [1] G. Betta A. Pietrosanto A. Scaglione. A Gray-code-based fiber optic liquid level transducer. *IEEE Transactions on Instrumentation and Measurement*, 47(1):174–178, February 1998.
- [2] F. Ruskey A. Williams. Generating combinations by prefix shifts. In *COCOON '05: Computing and Combinatorics, 11th Annual International Conference*, volume 3595 of *Lecture Notes in Computer Science*, Kunming, China, 2005. Springer-Verlag.
- [3] F. Ruskey A. Williams. The coolest way to generate combinations. *Discrete Mathematics*, (in press), 2008.
- [4] F. Ruskey A. Williams. An explicit universal cycle for the $(n - 1)$ -permutations of an n -set. *ACM Transactions on Algorithms*, (submitted), 2008.
- [5] F. Ruskey A. Williams. Generating balanced parentheses and binary trees by prefix shifts. In *CATS '08: Fourteenth Computing: The Australasian Theory Symposium*, volume 77 of *CRPIT*, Wollongong, Australia, 2008. ACS.

- [6] T.M.Y. Wang C. Savage. A Gray code for necklaces of fixed density. *SIAM Journal on Discrete Mathematics*, 9(4):654–673, 1996.
- [7] C. W. Ko F. Ruskey. Generating permutations of a bag by interchanges. *Information Processing Letters*, 41:263–269, 1992.
- [8] G. Pruesse F. Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, April 1994.
- [9] M. Jiang F. Ruskey. Determining the Hamilton-connectedness of certain vertex-transitive graphs. *Discrete Mathematics*, 133:159–170, 1994.
- [10] F. Ruskey J. Sawada. An efficient algorithm for generating necklaces with fixed density. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on discrete algorithms*, pages 729–758, Baltimore, Maryland, United States, 1999. Society for Industrial and Applied Mathematics.
- [11] F. Ruskey J. Sawada. An efficient algorithm for generating necklaces with fixed density. *SIAM Journal of Computing*, 29(2):671–684, 1999.
- [12] F. Ruskey J. Sawada. A fast algorithm to generate unlabeled necklaces. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on discrete algorithms*, pages 256–262, San Francisco, California, United States, 2000. Society for Industrial and Applied Mathematics.
- [13] V. Vajnovszki M. Weston. Gray codes for necklaces and lyndon words of arbitrary base. *Pure Mathematics and Applications/Algebra and Theoretical Computer Science*, 17(1-2):175–182, 2006.
- [14] F. Chung P. Diaconis R. Graham. Universal cycles for combinatorial structures. *Discrete Mathematics*, 110, 1992.
- [15] J. F. Korsh P. S. LaFollette. Loopless generation of linear extensions of a poset. *Order*, 18(2):115–126, 2002.
- [16] J. F. Korsh P. S. LaFollette. Loopless array generation of multiset permutations. *The Computer Journal*, 47(5):612–621, 2004.
- [17] E. R. Canfield S. G. Williamson. A loop-free algorithm for generating the linear extensions of a poset. *Order*, 12(1):57–75, 1995.
- [18] P. Diaconis S. Holmes. Gray codes for randomization procedures. *Statistical Computing*, 4:207–302, 1994.
- [19] J. F. Korsh S. Lipschutz. Generating multiset permutations in constant time. *Journal of Algorithms*, 25:321–335, 1997.
- [20] R. Sawae T. Sakata M. Tei K. Takarabe Y. Manmoto. Gray code and the initialization problem of NMR quantum computers. *International Journal of Quantum Chemistry*, 95:558–560, 2003.
- [21] P. F. Corbett. Rotator graphs: An efficient topology for point-to-point multiprocessor networks. *IEEE Transactions on Parallel and Distributed Systems*, 3:622–626, 1992.
- [22] N.G. de Bruijn. A combinatorial problem. *Koninkl. Nederl. Acad. Wetensch. Proc. Ser. A*, 49:758–764, 1946.
- [23] G. Ehrlich. Loopless algorithms for generating permutations, combinations and other combinatorial configurations. *Journal of the ACM*, 20:500–513, 1973.
- [24] F. Gray. Pulse code communication. *U.S. Patent 2,632,058*, 1947.
- [25] J. R. Johnson. Universal cycles for permutations. *Discrete Mathematics*, (in press), 2008.
- [26] S. M. Johnson. Generation of permutations by adjacent transpositions. *Mathematics of Computation*, 17:282–285, 1963.
- [27] D. E. Knuth. *The Art of Computer Programming*, volume 4 fascicle 3 - Generating All Combinations and Partitions. Updated 10/02/2008.
- [28] D. E. Knuth. *The Art of Computer Programming*, volume 4 fascicle 2 - Generating All Tuples and Permutations. Updated 10/02/2008.
- [29] D. E. Knuth. *The Art of Computer Programming*, volume 4 fascicle 4 - Generating All Trees. Updated 10/02/2008.
- [30] F. Ruskey. Generating linear extensions of posets by transpositions. *Journal of Combinatorial Theory (B)*, 54:77–101, 1992.
- [31] C. Savage. A survey of combinatorial Gray codes. *SIAM Review*, 39(4):605–629, 1997.
- [32] J. Sawada. A fast algorithm to generate necklaces with fixed-content. *Theoretical Computer Science*, 1-3(301):477–489, 2003.
- [33] T. Takaoka. An $O(1)$ time algorithm for generating multiset permutations. In *ISAAC '99: Algorithms and Computation, 10th International Symposium*, volume 1741 of *Lecture Notes in Computer Science*, pages 237–246, Chennai, India, 1999. Springer.
- [34] T. Ueda. Gray codes for necklaces. *Discrete Mathematics*, 219(1-3):235–248, 2000.
- [35] V. Vajnovszki. A loopless algorithm for generating the permutations of a multiset. *Theoretical Computer Science*, 2(307):415–431, 2003.
- [36] V. Vajnovszki. More restrictive Gray codes for necklaces and Lyndon words. *Information Processing Letters*, 106(3):96–99, 2008.