# THE COOLEST WAY TO GENERATE COMBINATIONS

FRANK RUSKEY AND AARON WILLIAMS

ABSTRACT. We present a practical and elegant method for generating all $(s,t)$-combinations (binary strings with $s$ zeros and $t$ ones): Identify the shortest prefix ending in 010 or 011 (or the entire string if no such prefix exists), and rotate it by one position to the right. This iterative rule gives an order to $(s,t)$-combinations that is circular and genlex. Moreover, the rotated portion of the string always contains at most four contiguous runs of zeros and ones, so every iteration can be achieved by transposing at most two pairs of bits. This leads to an efficient loopless and branchless implementation that consists only of two variables and six assignment statements. The order also has a number of striking similarities to colex order, especially its recursive definition and ranking algorithm. In light of these similarities we have named our order *cool-lex*!

## 1. Background and Motivation

An important class of computational tasks is the listing of fundamental combinatorial structures such as permutations, combinations, trees, and so on. Regarding combinations, Donald E. Knuth writes in his upcoming volume of *The Art of Computer Programming* [11] "Even the apparently lowly topic of combination generation turns out to be surprisingly rich, .... I strongly believe in building up a firm foundation, so I have discussed this topic much more thoroughly than I will be able to do with material that is newer or less basic."

The applications of combination generation are numerous and varied, and Gray codes for them are particularly valuable. We mention as application areas cryptography (where they have been implemented in hardware at NSA), genetic algorithms, software and hardware testing, statistical computation (e.g., for the bootstrap, Diaconis and Holmes [4]), and, of course, exhaustive combinatorial searches.

As is common, combinations are represented as binary strings, or bitstrings, of length $n = s+t$ containing $s$ zeros and $t$ ones. We denote this set as $\mathbf{B}(s,t) = \{b_1 b_2 \cdots b_n \mid \sum b_i = t\}$. Another way of representing combinations is as increasing sequences of the elements in the combination. Such representations are often referred to as position vectors, and we denote this set as $\mathbf{C}(s,t) = \{c_1 c_2 \cdots c_t \mid 1 \le c_1 < c_2 < \cdots < c_t \le s+t\}$.

Our initial motivation was to consider the problem of listing the elements of $\mathbf{B}(s,t)$ so that successive bitstrings differ by a prefix that is cyclically shifted by one position to the right. We refer to such shifts as *prefix shifts*, or *rotations*, and they may be represented by a cyclic permutation $\sigma_k = (1\ 2\ \cdots\ k)$ for some $2 \le k \le n$, where this permutation acts on the indices of a bitstring.

As far as we are aware, the only other class of strings that has a listing by prefix shifts are permutations, say of $\{1, 2, \ldots, n\}$. In Corbett [1] and Jiang and Ruskey [9] it is shown that all permutations may be listed circularly by prefix shifts. That is, the directed Cayley graph with generators $(1\ 2)$, $(1\ 2\ 3)$, $\ldots$, $(1\ 2\ \cdots\ n)$ is Hamiltonian. In our case we have the same set of generators acting on the indices of the bitstring, but the underlying graph is not vertex-transitive; in fact, it is not regular.

There are many algorithms for generating combinations. The one presented here has the following characteristics.

1. Successive combinations differ by a prefix shift. There is no other algorithm for generating combinations with this feature. In some applications combinations are represented in a single computer word; our algorithm is very fast in this scenario. It is also very suitable for hardware implementation.
2. Successive combinations differ by one or two transpositions of a 0 and a 1. There are other algorithms where successive combinations differ by a single transposition (Tang and Liu [18]). Furthermore, that transposition can be further restricted in various ways. For example, so that only zeros are between the

*Key words and phrases.* Gray code order, combinations, binary strings, colex, loopless algorithm, branchless algorithm, constant-extra-space, prefix rotation, prefix shift.

Research supported in part by an NSERC Discovery Grant.

Research supported in part by a NSERC PGS-D..

transposed bits (Eades and McKay [6]), or so that the transposed bits are adjacent or have only one bit between (Chase [3]). When $n$ is even and $k$ is odd it is possible to restrict the transposed bits to be adjacent (Eades, Hickey, and Read [5], and see Hough and Ruskey [8] for an efficient algorithm). Along with ours, these other variants are ably discussed in Knuth [11].
3. The list is circular; the first and last bitstrings differ by a prefix shift.
4. The algorithm can be implemented so that *in the worst case* only a small number of operations are done between successive combinations, *independent* of $s$ and $t$. Such algorithms are said to be *loopless*, an expression coined by Ehrlich [7]. In fact, the algorithm has a loopless implementation regardless of whether the combination is stored in an array, a computer word, or a linked list. In the first two cases the algorithm can also be implemented to be loopless and branchless (no *if*-statements). Existing loopless algorithms are discussed further in Section 6.3.
5. The list for $(s,t)$ begins with the list for $(s-1,t)$. Usually, this property is incompatible with Property 3, relative to the elementary operation used to transform one string to the next. For example, colex order has Property 4 but not Property 3. Colex is defined recursively so that every bitstring ending in 0 appears before every bitstring ending in 1

$$\mathbf{L}_{s,t} = \mathbf{L}_{s-1,t}0,\ \mathbf{L}_{s,t-1}1.$$

6. When the elements are expressed as $c_1c_2\cdots c_t \in \mathbf{C}(s,t)$, the list has the genlex property. A list of strings has the *genlex* property if the strings with any given suffix appear consecutively within the list. The term is due to Walsh [21]. We mention that the cool-lex algorithm cannot be implemented in loopless time when the combination is stored in this manner.
7. Unlike other Gray codes for combinations, this one has a simple ranking function whose running time is $O(n)$ arithmetic operations.
8. Unlike every other recursive Gray code definition for combinations, cool-lex has the remarkable property that it can be defined without using list reversals. Refer to [14] for examples of Gray codes that use list reversals.
9. The list is remarkably similar to the colex list for combinations.

The listing discussed here appears in Knuth's prefasicle [11]. The output of the algorithm is illustrated in Figure 26 on page 17. He refers to the listing as suffix-rotated (since he indexes the bitstrings $b_{n-1}\cdots b_1b_0$). See also Exercise 55 on page 30 and its solution on page 97.

To overview the remainder of the paper, Section 2 gives several definitions of cool-lex and proves that they are equivalent, Section 3 provides algorithms and implementations, Section 4 contains the ranking function for cool-lex, Section 5 discusses the genlex property, and Section 6 concludes with several open problems and an extension to permutations of a multi-set.

## 2. Cool-lex Definitions

In this section, we provide one iterative definition and two recursive definitions for cool-lex. Theorem 1 proves that all three definitions are equivalent, and gives several immediate consequences. We also provide an iterative and recursive definition for colex.

### 2.1. Preliminaries and Notation.

Before defining the cool-lex order, we introduce a number of secondary definitions. Let $\mathbf{S} = \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m$ be a sequence of strings, let $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ be individual strings, let $x$ be a symbol, let $k \geq 0$, and let $1 \leq i \leq m$. The string $\mathbf{bc}$ is obtained by appending $\mathbf{c}$ to the end of $\mathbf{b}$. If $\mathbf{d} = \mathbf{bc}$, then $\mathbf{b}$ is a *prefix* of $\mathbf{d}$, and $\mathbf{c}$ is a *suffix* of $\mathbf{d}$. The sequence of strings $\mathbf{s}_1\mathbf{b}, \mathbf{s}_2\mathbf{b}, \ldots, \mathbf{s}_m\mathbf{b}$ is represented by $\mathbf{Sb}$. Also, $x^k$ is the string with symbol $x$ repeated $k$ times. Let $\mathbf{S}[i] = \mathbf{s}_i$. We frequently access the first and last strings in a sequence, so if $\mathbf{S}$ is non-empty, then $first(\mathbf{S}) = \mathbf{s}_1$ and $last(\mathbf{S}) = \mathbf{s}_m$. If $\mathbf{S}$ contains at least two strings, then $second(\mathbf{S}) = \mathbf{s}_2$. Furthermore, if $\mathbf{S}$ contains at least two strings, then $\overrightarrow{\mathbf{S}}$ is the rotated sequence of strings $\mathbf{s}_2, \mathbf{s}_3, \ldots, \mathbf{s}_m, \mathbf{s}_1$; otherwise if $\mathbf{S}$ does not contain at least two strings, then $\overrightarrow{\mathbf{S}} = \mathbf{S}$. In this paper, every string will be binary, so that every symbol will be in $\{0,1\}$.

When $\mathbf{b}$ is a bitstring of length $n$, let $l(\mathbf{b})$ be the length of its shortest prefix ending in 010 or 011, or $n$ if no such prefix exists. Let $p(\mathbf{b})$ be the prefix of $\mathbf{b}$ that has length $l(\mathbf{b})$, and let $s(\mathbf{b})$ be the suffix such that $\mathbf{b} = p(\mathbf{b})s(\mathbf{b})$. Let $\sigma(\mathbf{b})$ be the result of rotating $p(\mathbf{b})$ by one position to the right, and appending $s(\mathbf{b})$. Recursively define $\sigma^i(\mathbf{b}) = \sigma(\sigma^{i-1}(\mathbf{b}))$, where $\sigma^0(\mathbf{b}) = \mathbf{b}$.

2.1.1. *Properties of* $\sigma$. The strings $1^t0^s$ and $1^{t-1}0^s1$ play special roles in cool-lex, because these are the only strings with no prefix ending in 010 or 011, and their importance and relationship are given by the following three remarks.

**Remark 1.** $\sigma(\mathbf{b})0 = \sigma(\mathbf{b}0)$ *if and only if* $\mathbf{b} \neq 1^{t-1}0^s1$.

**Remark 2.** $\sigma(\mathbf{b})1 = \sigma(\mathbf{b}1)$ *if and only if* $\mathbf{b} \neq 1^t0^s$ *with* $s \geq 1$.

**Remark 3.** $\sigma(1^{t-1}0^s1) = 1^t0^s$.

**Remark 4.** $\sigma(\mathbf{b}) = \sigma(p(\mathbf{b}))s(\mathbf{b})$.

Lemma 1 shows how transpositions can take the place of rotations.

**Lemma 1.** $\sigma(\mathbf{b})$ *can be obtained from* $\mathbf{b}$ *by transposing one or two pairs of bits.*

*Proof.* If $p(\mathbf{b})$ does not end in 010 or 011, then $\mathbf{b} = 1^t0^s$ and $\sigma(\mathbf{b}) = 01^t0^{s-1}$, or $\mathbf{b} = 1^{t-1}0^s1$ and $\sigma(\mathbf{b}) = 1^t0^s$. In both of these cases, $\sigma(\mathbf{b})$ can be obtained from $\mathbf{b}$ with one transposition. Otherwise, $p(\mathbf{b})$ does end in 010 or 011: so it must be of the form $00^i10$, $11^i00^j10$, $00^i11$, or $11^i00^j11$, where $i, j \geq 0$. For each case we verify the claim by illustrating the first positions to be transposed in $p(\mathbf{b})$ using underlines, and if necessary, the second positions to be transposed in $p(\mathbf{b})$ using overlines. Remark 4 justifies why the transpositions are contained within $p(\mathbf{b})$.

Case 1: $\sigma(00^i10) = 00^i\underline{10} = 00^i01$.

Case 2: $\sigma(11^i00^j10) = \underline{1}1^i\underline{0}0^j\overline{10} = 01^i10^j\overline{10} = 011^i00^j1$.

Case 3: $\sigma(00^i11) = \underline{0}0^i\underline{1}1 = 100^i1$.

Case 4: $\sigma(11^i00^j11) = 11^i\underline{0}0^j\underline{1}1 = 111^i00^j1$. □ □

2.2. **Iterative Definition.** Formally, the iterative definition of cool-lex with $s$ zeros and $t$ ones is

$$\mathbf{R}_{s,t} = \sigma^0(\mathbf{b}),\ \sigma^1(\mathbf{b}),\ \sigma^2(\mathbf{b}),\ \ldots,\ \sigma^z(\mathbf{b}), \tag{1}$$

where $\mathbf{b} = 1^t0^s$ and $z = \binom{s+t}{t} - 1$. When $s = 1$ or $t = 1$, the strings in $\mathbf{R}_{s,t}$ are given explicitly by the following two remarks. The center column of Figure 1 gives $\mathbf{R}_{3,3}$.

**Remark 5.** $\mathbf{R}_{1,t} = 1^t0,\ 01^t,\ 101^{t-1},\ 1^201^{t-2},\ \ldots,\ 1^{t-1}01.$

**Remark 6.** $\mathbf{R}_{s,1} = 10^s,\ 010^{s-1},\ 0^210^{s-2},\ \ldots,\ 0^s1.$

To complement the iterative definition of cool-lex, let us consider the well-known iterative definition of colex [11], the lexicographic order applied to the reversal of strings, which begins with $1^t0^s$ and ends with $0^s1^t$. Colex has many uses, for example in Frankl's now standard proof of the Kruskal-Katona Theorem [16]. Let $\mathbf{b}$ be a bitstring of length $n$. Given that $\mathbf{b} \neq 0^s1^t$, let $l'(\mathbf{b})$ be the length of the shortest prefix in $\mathbf{b}$ that ends in 10, let $p'(\mathbf{b})$ be the prefix of $\mathbf{b}$ that has length $l'(\mathbf{b})$, and let $s'(\mathbf{b})$ be the suffix of $\mathbf{b}$ such that $\mathbf{b} = p'(\mathbf{b})s'(\mathbf{b})$. Let $\varsigma(\mathbf{b})$ be the result of replacing $p'(\mathbf{b}) = 0^i1^j0$ by $1^{j-1}0^{i+1}1$ and appending $s'(\mathbf{b})$. Recursively define $\varsigma^i(\mathbf{b}) = \varsigma(\varsigma^{i-1}(\mathbf{b}))$, where $\varsigma^0(\mathbf{b}) = \mathbf{b}$. Notice that $\varsigma(\mathbf{b})$ is well-defined, except for $\mathbf{b} = 0^s1^t$, which is the last string in colex. The iterative definition of colex with $s$ zeros and $t$ ones is

$$\mathbf{I}_{s,t} = \varsigma^0(\mathbf{b}),\ \varsigma^1(\mathbf{b}),\ \varsigma^2(\mathbf{b}),\ \ldots,\ \varsigma^z(\mathbf{b}), \tag{2}$$

where $\mathbf{b} = 1^t0^s$ and $z = \binom{s+t}{t} - 1$. When $s = 1$ or $t = 1$, the strings in $\mathbf{I}_{s,t}$ are given explicitly by the following two remarks. The third column of Figure 2 gives $\mathbf{I}_{3,3}$.

**Remark 7.** $\mathbf{I}_{1,t} = 1^t0,\ 1^{t-1}01,\ 1^{t-2}01^2,\ 1^{t-3}01^3,\ \ldots,\ 01^t.$

**Remark 8.** $\mathbf{I}_{s,1} = 10^s,\ 010^{s-1},\ 0^210^{s-2},\ \ldots,\ 0^s1.$

| $\mathbf{M}_{3,2}$ | $\overrightarrow{\mathbf{M}_{3,2}}$ | $\mathbf{M}_{2,3}$ | $\mathbf{M}_{3,3} = \mathbf{R}_{3,3}$ | $l(\mathbf{b})$ | $p(\mathbf{b}) \cdot s(\mathbf{b})$ | $\sigma(\mathbf{b})$ |
|---|---|---|---|---|---|---|
| | | 11100 | 111000 | 6 | 111000 · | 011100 |
| | | 01110 | 011100 | 3 | 011 · 100 | 101100 |
| | | 10110 | 101100 | 4 | 1011 · 00 | 110100 |
| | | 11010 | 110100 | 5 | 11010 · 0 | 011010 |
| | | 01101 | 011010 | 3 | 011 · 010 | 101010 |
| | | 10101 | 101010 | 4 | 1010 · 10 | 010110 |
| | | 01011 | 010110 | 3 | 010 · 110 | 001110 |
| | | 00111 | 001110 | 4 | 0011 · 10 | 100110 |
| | | 10011 | 100110 | 5 | 10011 · 0 | 110010 |
| | | 11001 | 110010 | 6 | 110010 · | 011001 |
| 11000 | 01100 | | 011001 | 3 | 011 · 001 | 101001 |
| 01100 | 10100 | | 101001 | 4 | 1010 · 01 | 010101 |
| 10100 | 01010 | | 010101 | 3 | 010 · 101 | 001101 |
| 01010 | 00110 | | 001101 | 4 | 0011 · 01 | 100101 |
| 00110 | 10010 | | 100101 | 5 | 10010 · 1 | 010011 |
| 10010 | 01001 | | 010011 | 3 | 010 · 011 | 001011 |
| 01001 | 00101 | | 001011 | 4 | 0010 · 11 | 000111 |
| 00101 | 00011 | | 000111 | 5 | 00011 · 1 | 100011 |
| 00011 | 10001 | | 100011 | 6 | 100011 · | 110001 |
| 10001 | 11000 | | 110001 | 6 | 110001 · | 111000 |

FIGURE 1. Recursive and iterative structure of cool-lex with $\mathbf{M}_{3,3} = \mathbf{R}_{3,3}$ in the middle column. The leftmost three columns show its recursive structure since $\mathbf{M}_{3,3} = \mathbf{M}_{2,3}0, \overrightarrow{\mathbf{M}_{3,2}}1$. The rightmost three columns show its iterative structure since each string, $\mathbf{b}$ in $\mathbf{R}_{3,3}$, is broken into its prefix $p(\mathbf{b})$ of length $l(\mathbf{b})$, and its suffix $s(\mathbf{b})$. The prefix is rotated by one position to the right to obtain $\sigma(\mathbf{b})$, which is the next string in $\mathbf{R}_{3,3}$.

| $\mathbf{L}_{3,2}$ | $\mathbf{L}_{2,3}$ | $\mathbf{L}_{3,3} = \mathbf{I}_{3,3}$ | $l'(\mathbf{b})$ | $p'(\mathbf{b}) \cdot s'(\mathbf{b})$ | $\varsigma(\mathbf{b})$ |
|---|---|---|---|---|---|
| | 11100 | 111000 | 4 | 1110 · 00 | 110100 |
| | 11010 | 110100 | 3 | 110 · 100 | 101100 |
| | 10110 | 101100 | 2 | 10 · 1100 | 011100 |
| | 01110 | 011100 | 5 | 01110 · 0 | 110010 |
| | 11001 | 110010 | 3 | 110 · 010 | 101010 |
| | 10101 | 101010 | 2 | 10 · 1010 | 011010 |
| | 01101 | 011010 | 4 | 0110 · 10 | 100110 |
| | 10011 | 100110 | 2 | 10 · 0110 | 010110 |
| | 01011 | 010110 | 3 | 010 · 110 | 001110 |
| | 00111 | 001110 | 6 | 001110 · | 110001 |
| 11000 | | 110001 | 3 | 110 · 001 | 101001 |
| 10100 | | 101001 | 2 | 10 · 1001 | 011001 |
| 01100 | | 011001 | 4 | 0110 · 01 | 100101 |
| 10010 | | 100101 | 2 | 10 · 0101 | 010101 |
| 01010 | | 010101 | 3 | 010 · 101 | 001101 |
| 00110 | | 001101 | 5 | 00110 · 1 | 100011 |
| 10001 | | 100011 | 2 | 10 · 0011 | 010011 |
| 01001 | | 010011 | 3 | 010 · 011 | 001011 |
| 00101 | | 001011 | 4 | 0010 · 11 | 000111 |
| 00011 | | 000111 | - | - | - |

FIGURE 2. Recursive and iterative structure of colex with $\mathbf{L}_{3,3} = \mathbf{I}_{3,3}$ in the third column. The leftmost two columns show its recursive structure since $\mathbf{L}_{3,3} = \mathbf{L}_{2,3}0, \mathbf{L}_{3,2}1$. The rightmost three columns show its iterative structure since each string, $\mathbf{b}$ in $\mathbf{I}_{3,3}$, is broken into its prefix $p'(\mathbf{b})$ of length $l'(\mathbf{b})$, and its suffix $s'(\mathbf{b})$. The prefix is updated to obtain $\varsigma(\mathbf{b})$, which is the next string in $\mathbf{I}_{3,3}$.

| | | |
|---|---|---|
| 11000 | $1\boxed{1000}$ | 11000 |
| 10100 | $10\boxed{100}$ | 01100 |
| 01100 | $01\boxed{100}$ | 10100 |
| 10010 | $100\boxed{10}$ | 01010 |
| 01010 | $010\boxed{10}$ | 00110 |
| 00110 | $001\boxed{10}$ | 10010 |
| 10001 | $1000\boxed{1}$ | 01001 |
| 01001 | $0100\boxed{1}$ | 00101 |
| 00101 | $0010\boxed{1}$ | 00011 |
| 00011 | $0001\boxed{1}$ | 10001 |

FIGURE 3. On the left is colex $\mathbf{L}_{3,2}$ and on the right is cool-lex $\mathbf{M}_{3,2}$. The middle column contains $\mathbf{L}_{3,2}$ and its suffixes beginning with 1 (1000, 100, 10, and 1) are highlighted by rectangles. In order to transform colex into cool-lex each sublist associated with one of these suffixes is cyclically moved up one row.

2.3. **Recursive Definitions.** Although we presented an iterative definition of colex, it is perhaps more commonly expressed recursively. We use $\mathbf{L}_{s,t}$ in the recursive definition, and we note that $\mathbf{I}_{s,t} = \mathbf{L}_{s,t}$ [11]. The colex list $\mathbf{L}_{s,t}$ is given by the following:

$$\mathbf{L}_{s,t} = \mathbf{L}_{s-1,t}0,\ \mathbf{L}_{s,t-1}1, \tag{3}$$

where $\mathbf{L}_{0,t} = 1^t$ and $\mathbf{L}_{s,0} = 0^s$. Interestingly, cool-lex can be defined in a very similar manner. The cool-lex list $\mathbf{M}_{s,t}$ is given by the following:

$$\mathbf{M}_{s,t} = \mathbf{M}_{s-1,t}0,\ \overrightarrow{\mathbf{M}_{s,t-1}}1, \tag{4}$$

where $\mathbf{M}_{0,t} = 1^t$ and $\mathbf{M}_{s,0} = 0^s$. Equations (3) and (4) imply that colex can be transformed into cool-lex by a series of sublist manipulations. Figure 3 illustrates this transformation for $s = 3$ and $t = 2$, while transformations for larger values of $t$ work recursively. Remark 9 follows immediately from (4).

**Remark 9.** *Each bitstring with $s$ zeros and $t$ ones appears exactly once in $\mathbf{M}_{s,t}$.*

Although the representation of cool-lex in (4) has certain advantages, it can also be useful to have a recursive definition that does not reorder strings as in $\overrightarrow{\mathbf{S}}$. By using the same base cases, we can define cool-lex recursively by $\mathbf{W}'_{s,t} = 1^t0^s, \mathbf{W}_{s,t}$ where

$$\mathbf{W}_{s,t} = \mathbf{W}_{(s-1),t}0,\ \mathbf{W}_{s,(t-1)}1,\ 1^{t-1}0^s1. \tag{5}$$

In fact, this definition is used in [11] and in a conference paper containing preliminary results [15]. When $s = 1$ or $t = 1$, the strings in $\mathbf{M}_{s,t}$ and $\mathbf{W}'_{s,t}$ are given explicitly in the following two remarks.

**Remark 10.** $\mathbf{M}_{1,t} = \mathbf{W}'_{1,t} = 1^t0,\ 01^t,\ 101^{t-1},\ 1^201^{t-2},\ \ldots,\ 1^{t-1}01.$

**Remark 11.** $\mathbf{M}_{s,1} = \mathbf{W}'_{s,1} = 10^s,\ 010^{s-1},\ 0^210^{s-2},\ \ldots,\ 0^s1.$

Lemma 3 proves that $\mathbf{M}_{s,t} = \mathbf{W}'_{s,t}$. One advantage of $\mathbf{W}'_{s,t}$ is that it is easy to identify the first and last strings in the cool-lex. We will also find it useful to know the second string in cool-lex order, which we compute using $\mathbf{M}_{s,t}$.

**Lemma 2.** *The first, last, and second strings in cool-lex are as follows*

$$first(\mathbf{W}'_{s,t}) = 1^t0^s \tag{6}$$

$$last(\mathbf{W}'_{s,t}) = 1^{t-1}0^s1 \tag{7}$$

$$second(\mathbf{M}_{s,t}) = 01^t0^{s-1} \text{ for } s,t > 1. \tag{8}$$

*Proof.* Parts (a) and (b) follow immediately from the definitions. For part (c),

$$second(\mathbf{M}_{s,t}) = second(\mathbf{M}_{s-1,t})0 = \ldots = second(\mathbf{M}_{1,t})0^{s-1} = 01^t0^{s-1}$$

by Remark 10. □ □

2.4. **Equivalence of Definitions.** Now we are ready to state the main result of this section, Theorem 1.

**Theorem 1.** $\mathbf{R}_{s,t} = \mathbf{W'}_{s,t} = \mathbf{M}_{s,t}$. *Moreover,*

- *The lists are circular.*
- *The lists contains each bitstring with $s$ zeros and $t$ ones exactly once.*
- *Successive bitstrings differ by a prefix shift of one position to the right.*
- *Successive bitstrings differ by the transposition of one or two pairs of bits.*
- *The first bitstring is $1^t0^s$, and the last bitstring is $1^{t-1}0^s1$.*

The proof of Theorem 1 involves two lemmas. We first prove that the two recursive definitions of cool-lex, $\mathbf{M}_{s,t}$ and $\mathbf{W}'_{s,t}$ are equivalent, and then we prove that these definitions are equivalent to the iterative definition of cool-lex, $\mathbf{R}_{s,t}$.

**Lemma 3.** $\mathbf{M}_{s,t} = \mathbf{W}'_{s,t}$.

*Proof.* From Remarks 10 and 11, the result is true when $s = 1$ or $t = 1$. Otherwise, suppose that $s, t > 1$ and inductively assume that $\mathbf{M}_{i,j} = \mathbf{W}'_{i,j}$ whenever ($i < s$ and $j \le t$) or ($i \le s$ and $j < t$). Then we have the following:

$$\begin{aligned}
\mathbf{W}'_{s,t} &= 1^t0^{s-1}0,\ \mathbf{W}_{s,t} \\
&= 1^t0^{s-1}0,\ \mathbf{W}_{s-1,t}0,\ \mathbf{W}_{s,t-1}1,\ 1^{t-1}0^s1 \\
&= \mathbf{W}'_{s-1,t}0,\ \mathbf{W}_{s,t-1}1,\ 1^{t-1}0^s1 \\
&= \mathbf{M}_{s-1,t}0,\ \mathbf{W}_{s,t-1}1,\ 1^{t-1}0^s1 \\
&= \mathbf{M}_{s-1,t}0,\ \overrightarrow{(1^{t-1}0^s1,\ \mathbf{W}_{s,t-1}1)} \\
&= \mathbf{M}_{s-1,t}0,\ \overrightarrow{(\mathbf{W}'_{s,t-1}1)} \\
&= \mathbf{M}_{s-1,t}0,\ \overrightarrow{(\mathbf{M}_{s,t-1}1)} \\
&= \mathbf{M}_{s,t}.
\end{aligned}$$

□ □

**Lemma 4.** $\mathbf{M}_{s,t} = \mathbf{R}_{s,t}$.

*Proof.* Remarks 5, 6, 10, and 11 provide the result when $s = 1$ or $t = 1$. Otherwise, suppose that $s, t > 1$ and inductively assume that $\mathbf{M}_{i,j} = \mathbf{R}_{i,j}$ whenever ($i < s$ and $j \le t$) or ($i \le s$ and $j < t$). The following list gives an overview of $\mathbf{M}_{s,t} = \mathbf{M}_{s-1,t}0,\ \overrightarrow{\mathbf{M}_{s,t-1}1}$, with a horizontal line separating the two sublists. We wish to show that each successive string in $\mathbf{M}_{s,t}$ is the result of applying $\sigma$ to the previous string.

$$\begin{array}{cc}
11^{t-2}100^{s-2}0 & \text{(S1)} \\
011^{t-2}10^{s-2}0 & \\
\vdots & \\
11^{t-2}00^{s-2}10 & \text{(S2)} \\ \hline
011^{t-2}00^{s-2}1 & \text{(S3)} \\
\vdots & \\
1^{t-2}00^{s-2}011 & \text{(S4)} \\
11^{t-2}00^{s-2}01 & \text{(S5)}
\end{array}$$

The strings (S1)-(S5) are identified by Lemma 2 (equations (6) and (7)). The strings from (S1) to (S2) are the strings in $\mathbf{M}_{s-1,t}0$. From Remark 1 appending 0 does not affect the operation of $\sigma$, except for the string labeled (S2). Therefore, the fact that each successive string from (S1) to (S2) is obtained from applying $\sigma$ is a result of the inductive assumption that $\mathbf{M}_{s-1,t} = \mathbf{R}_{s-1,t}$. Next, note that applying $\sigma$ to the string (S2) results in the string (S3).

The strings from (S3) to (S5) are the strings in $\overrightarrow{\mathbf{M}_{s,t-1}1}$. From Remark 2 appending 1 does not affect the operation of $\sigma$, for every string from (S3) to (S4). Therefore, the fact that each successive string from (S3) to (S4) is obtained from applying $\sigma$ is a result of the inductive assumption that $\mathbf{M}_{s,t-1} = \mathbf{R}_{s,t-1}$. Finally, note that applying $\sigma$ to the string (S4) results in the string (S5). □ □

$$
\begin{array}{ccc}
\cline{2-2}
010^{s-2}0 & 011^{t-2}1 & 011^{t-2}10^{s-2}0 \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \underline{1}1^{t-2}\underline{0}0^{s-2}\overline{10} \\
\cline{3-3}
\vdots & \vdots & 01^{t-2}10^{s-2}01 \\
\vdots & \vdots & \vdots \\
0^{s-2}0\underline{10} & 1^{t-2}\underline{01}1 & 1^{t-2}\underline{0}00^{s-2}\underline{1}1 \\
\hline \cline{1-1}
00^{s-2}01 & 11^{t-2}01 & 1^{t-2}100^{s-2}01
\end{array}
$$

FIGURE 4. Illustrating the transpositions at the two interfaces in $\mathbf{W}_{s,t}$.

Now we prove Theorem 1.

*Proof.* The first point follows from Remark 3. The second point follows from Remark 9. The third point follows from the definitions of $\mathbf{R}_{s,t}$ and $\sigma$. The fourth point follows from Lemma 1. The last point follows from Lemma 2 (equations (6) and (7)). □ □

## 3. Algorithms and Implementation

In this section, we concentrate on efficient algorithms for generating cool-lex. In particular, we provide a recursive algorithm, a loopless iterative algorithm, and a loopless and branchless iterative algorithm, each of which is implemented in a procedural language. We also provide a loopless iterative algorithm that is implemented using linked lists instead of arrays, and a second loopless and branchless iterative algorithm that is implemented in machine language and is due to Knuth [11]. Every algorithm presented uses constant extra space; that is, besides storing the $(s, t)$-combination only $\mathbf{O}(1)$ space is required.

Within each algorithm we follow the convention that $\leftarrow$ represents assignments, and $=$ represents testing for equality. Also, every array has 1-*based-indexing*; that is, if $b$ is an array then $b[1]$ represents its first element.

### 3.1. Recursive Algorithm.

To generate cool-lex recursively we use the definitions of $\mathbf{W}'_{s,t}$ and $\mathbf{W}_{s,t}$ that we recall here:

$$
\begin{aligned}
\mathbf{W}'_{st} &= 1^t0^s,\ \mathbf{W}_{st} \\
\mathbf{W}_{st} &= \mathbf{W}_{(s-1)t}0,\ \mathbf{W}_{s(t-1)}1,\ 1^{t-1}0^s1
\end{aligned}
$$

Figure 4 shows the strings in $\mathbf{W}_{s,t}$, where the two long horizontal lines represent the transitions between $\mathbf{W}_{(s-1)t}0$, $\mathbf{W}_{s(t-1)}1$, and $1^{t-1}0^s1$. The left column shows the base case of $t = 1$, the middle column shows the base case of $s = 1$, and the right column shows the remaining case of $s, t > 1$. The short underlines, and overlines, represent which bits are transposed at each interface.

In the right column, the transposed bits at the first interface are at positions $(1, t)$ and $(n-1, n)$, and at the second interface are at positions $(t-1, n-1)$ (Lemma 1). We use the function call swap$(i, j)$ to swap the $i$th and $j$th bits in $b$. To generate all of the strings in $\mathbf{W}'_{s,t}$ we call **Recursive**$(s, t)$ to visit $1^t0^s$ and $01^t0^{s-1}$, and then calls **Recurse**$(s, t)$ to recursively visit the remaining strings in $\mathbf{W}_{s,t}$. During this process we assume that **Recurse**$(s, t)$ has access to $b$. In other words, $b$ is a global variable. Since every recursive call is followed by a visit, the algorithm runs in constant amortized time.

**Recursive**$(s,t)$

```
Require: s, t > 0
 1: b ← array(1^t 0^s)
 2: visit(b)
 3: swap(1, t+1)
 4: visit(b)
 5: Recurse(s, t)
```

**Recurse**$(s,t)$

```
 1: if s > 1 then
 2:    Recurse(s−1, t)
 3:    swap(1, t)
 4:    swap(s+t, s+t−1)
 5:    visit(b)
 6: end
 7: if t > 1 then
 8:    Recurse(s, t−1)
 9:    swap(t−1, s+t−1)
10:    visit(b)
11: end
```

### 3.2. Iterative Algorithms.

3.2.1. *Rotations and Linked lists.* The simplest iterative algorithms for cool-lex are those that closely follow its iterative definition: rotate the shortest prefix ending in 010 or 011, or the entire bitstring if no such prefix exists, by one position to the right. In **Rotate**$(s,t)$ we store the bitstring in an array (line 1) and we assume that rotate$(b,i)$ rotates the first $i$ bits of $b$ by one position to the right. In particular, we maintain a variable $x$ that is equal to the smallest index for which $b[x-1]=0$ and $b[x]=1$, and then we rotate the first $x+1$ bits of $b$ at each iteration (line 6). After a rotation the leftmost 01 is moved one position to the right, or a new leftmost 01 is created at the beginning of the bitstring, and so the value of $x$ is updated accordingly (lines 7 to 11). The algorithm ends when the last bitstring in cool-lex is reached, $1^{t-1}0^s1$, which is the unique bitstring where the value of $x$ is equal to $s+t$ (line 4). The algorithm begins with the first string in cool-lex, $1^t0^s$, and initializes $x$ to $t$ since rotate$(1^t0^s, t+1)$ produces the second string in cool-lex, $01^t0^{s-1}$. Before describing the next algorithm, we mention that updating the value of $x$ (lines 7 to 11) can be accomplished by a single operation (see line 10 in **Branchless**$(s,t)$).

The **LinkedList**$(s,t)$ algorithm is essentially the same as the **Rotate**$(s,t)$ algorithm, except that we store the bitstring in a singly-linked list, and we perform our rotations by using an auxiliary variable $y$ and four elementary pointer operations (lines 5 to 8). Since every operation in **LinkedList**$(s,t)$ is elementary, the algorithm is loopless. As far as the authors are aware, **LinkedList**$(s,t)$ is the first $(s,t)$-combination algorithm using linked lists with this property.

**Rotate**$(s,t)$

```
Require: t > 0
 1: b ← array(1^t 0^s)
 2: x ← t
 3: visit(b)
 4: while x < s+t do
 5:
 6:    rotate(b, x+1)
 7:    x ← x+1
 8:
 9:    if b[1] = 0 and b[2] = 1 then
10:       x ← 2
11:    end
12:    visit(b)
13: end
```

**LinkedList**$(s,t)$

```
Require: t > 0
 1: b ← linkedlist(1^t 0^s)
 2: x ← findnode(b, t)
 3: visit(b)
 4: while x.next ≠ NULL do
 5:    y ← x.next
 6:    x.next ← x.next.next
 7:    y.next ← b
 8:    b ← y
 9:    if b.val = 0 and b.next.val = 1 then
10:       x ← b.next
11:    end
12:    visit(b)
13: end
```

3.2.2. *Loopless Algorithm.* Although **Rotate**$(s,t)$ relied upon the function rotate$(b,i)$, we do not need to perform arbitrary rotations to generate cool-lex. In particular, each successive bitstring can be generated by one or two transpositions (see Lemma 1), or equivalently by two or four array assignments. In **Loopless**$(s,t)$ we find it useful to maintain another variable in addition to $x$. Let $y$ be the smallest index for which $b[y]=0$. Referring back to Figure 4 we observe that in every case $b[x]$ becomes 0 and $b[y]$ becomes 1 (lines 6 and 7). The test $b[x+1]=0$ determines whether we are at the first or the second interface (line 10, with respect to line 8). If we are at the first interface, then set $b[x+1]$ to 1 and $b[0]$ to 0 (lines 11 and 12, with respect to line 8). It now remains to update $x$ and $y$. At the second interface they are simply incremented (lines 8 and 9). At the first interface $y$ always becomes 1 (line 16); also, $x$ is incremented unless $y$ is equal to 1, in which case $x$ becomes two (line 16, with respect to line 9) (see Remark 11). The algorithm has the same ending condition as **Rotate**$(s,t)$ (line 5). The algorithm initializes $x$ and $y$ to $t$ (lines 2 and 3) and the reader can verify that the first iteration of the while loop correctly changes $b$ from the first string in cool-lex, $1^t0^s$, to the second string in cool-lex, $01^t0^{s-1}$, and $y$ is properly set to 1 and $x$ is properly set to two.

```
Loopless(s, t)
Require: t > 0
 1: b ← array(1^t 0^s)
 2: x ← t
 3: y ← t
 4: visit(b)
 5: while x < s + t do
 6:    b[x] ← 0
 7:    b[y] ← 1
 8:    x ← x + 1
 9:    y ← y + 1
10:    if b[x] = 0 then
11:       b[x] ← 1
12:       b[1] ← 0
13:       if y > 2 then
14:          x ← 2
15:       end
16:       y ← 1
17:    end
18:    visit(b)
19: end
```

```
Branchless(s, t)
Require: t > 0
 1: b ← array(1^t 0^s)
 2: x ← t
 3: y ← t
 4: visit(b)
 5: while x < s + t do
 6:    b[x] ← 0
 7:    b[y] ← 1
 8:    b[1] ← b[x + 1]
 9:    b[x + 1] ← 1
10:    x ← x + 1 − (x − 1) · b[2] · (1 − b[1])
11:    y ← b[1] · y + 1
12:    visit(b)
13: end
```

The structure of **Loopless**$(s,t)$ allows us to completely determine the number of times each statement is executed. Let $X(s,t)$, $Y(s,t)$, and $Z(s,t)$, represent the number of times lines 6, 11, and 14 are executed, respectively. Line 6 is executed for every $(s,t)$-combination except the last in cool-lex order, $1^{t-1}0^s1$. Line 11 is executed for every $(s,t)$-combination that contains a 010 before any 011, of which there are $\binom{s+t-1}{t} - 1$ possibilities, as well as the first bitstring in cool-lex order, $1^t0^s$. Line 14 is executed for every $(s,t)$-combination that starts with 1 and is also executed by line 11. Thus,

$$X(s,t) = \binom{s+t}{t} - 1, \quad Y(s,t) = \binom{s+t-1}{t}, \quad \text{and } Z(s,t) = \binom{s+t-2}{t-1}.$$

3.2.3. *Loopless and Branchless Algorithm.* **Loopless**$(s,t)$ generates the cool-lex ordering by transposing either one pair or two pairs of bits at each step. Interestingly, the cool-lex ordering can also be generated by **Branchless**$(s,t)$ that *always* swaps two pairs of bits. In particular, by maintaining the variables as before, each successive string can be obtained by swap$(x,y)$ and swap$(0,x+1)$. As before, the first string in cool-lex order, $1^t0^s$, is a special case, and the algorithm terminates once visiting the last string in cool-lex order, $1^{t-1}0^s1$. In all other cases, there is a shortest prefix ending in 010 or 011 (referred to by $p(\mathbf{b})$), which explains the hypothesis of the following lemma.

**Lemma 5.** *If $p(\mathbf{b})$ ends in 010 or 011, then $\sigma(\mathbf{b})$ can be obtained from $\mathbf{b}$ by transposing bits $(x,y)$, followed by transposing bits $(0,x+1)$.*

*Proof.* Since $p(\mathbf{b})$ ends in 010 or 011, then it must be of the form $00^i10$, $11^i00^j10$, $00^i11$, or $11^i00^j11$, where $i,j \geq 0$. By Remark 4 we need only transpose bits in $p(\mathbf{b})$, and for each case we verify the claim by illustrating the transposition to be made in positions $(x,y)$ using underlines, and the transposition to be made in positions $(0,x+1)$ using overlines.

Case 1: $\sigma(00^i10) = \underline{\overline{0}}0^i\underline{1}\overline{0} = \overline{1}0^i0\overline{0} = 00^i01$.

Case 2: $\sigma(11^i00^j10) = \overline{1}1^i\underline{0}0^j\underline{1}\overline{0} = \overline{1}1^i10^j0\overline{0} = 011^i00^j1$.

Case 3: $\sigma(00^i11) = \underline{\overline{0}}0^i\underline{1}\overline{1} = \overline{1}0^i0\overline{1} = 100^i1$.

Case 4: $\sigma(11^i00^j11) = \overline{1}1^i\underline{0}0^j\underline{1}\overline{1} = \overline{1}1^i10^j0\overline{1} = 111^i00^j1$. □ □

Given the correct values of $x$ and $y$, Lemma 5 allows us to generate the next string without branching (lines 6 to 9). Once the next string has been generated we can easily compute the correct values of $x$ and $y$. In particular, the value of $y$ is incremented by one, unless the first bit is set to 0, in which case $y$ is set to 1 (line 11). Likewise, the value of $x$ is incremented by one, unless the first two bits are set to 01, in which case $x$ is set to two (line 10).

3.3. **Implementation in Computer Words.** The final implementation we present is of a different nature from the previous three. In this case we assume that our $n$-bit binary string can fit in a single machine word, and we operate on this word using machine language. By using shifts, bitmasks, and arithmetic, there are a number of ways to accomplish this goal. The approach we follow here is due to Knuth [11], and it gives a loopless and branchless MMIX implementation. To understand the algorithm, we need the following two lemmas, which show how the operation of $\sigma$ can be simulated by using addition and subtraction on words. To allow addition and subtraction to achieve this goal we must reverse the order of the bits (and in [11] the cool-lex ordering is referred to as *suffix-rotated*). Again, we focus only on $p(\mathbf{b})$ thanks to Remark 4.

**Lemma 6.** *If* $p(\mathbf{b}) = 1^x00^y10$*, then* $\sigma(\mathbf{b}) = \mathbf{b} + \mathbf{c}$*, for* $\mathbf{c} = 1^x00^y10^{|s(\mathbf{b})|}$.

*Proof.* To verify that we can obtain $\sigma(1^x00^y10) = 01^x0^y01$ by adding $\mathbf{c}$, we write each string from right to left while omitting the unchanged bits from $s(\mathbf{b})$:

$$\begin{array}{rl} & 010^y01^x \\ + & 010^y01^x \\ \hline & 100^y1^x0 \end{array}$$

□ □

**Lemma 7.** *If* $p(\mathbf{b}) = 1^x00^y11$*, then* $\sigma(\mathbf{b}) = \mathbf{b} - \mathbf{c}$*, for* $\mathbf{c} = 0^x11^y000^{|s(\mathbf{b}|)}$.

*Proof.* To verify that we can obtain $\sigma(1^x00^y11) = 11^x0^y01$ by subtracting $\mathbf{c}$, we write each string from right to left while omitting the unchanged bits from $s(\mathbf{b})$:

$$\begin{array}{rl} & 110^y01^x \\ - & 001^y10^x \\ \hline & 100^y11^x \end{array}$$

□ □

For the implementation, we assume that every register has length $w$, and that $n < w$, where $n = s+t$. We will write the contents of each register as $R = r_1r_2 \ldots r_w$ where $r_1$ is the least significant bit. In other words, $R = 1000\ldots$ is equivalent to the integer value 1. The operator $\ll$ represents the shifting of bits towards greater significance, so $1 \ll k$ equals $0^k10^{w-k-1}$. The operator $\wedge$ represents bitwise-and. The operator $\oplus$ represents bitwise-xor. The operator $\ominus$ is a specialized form of subtraction, called *saturating subtraction*, where the result of $i \ominus j$ is $i - j$ if $i \geq j$, and is 0 if $i < j$. Although this operation is not available in all machine languages, it is available in MMIX, and can easily be simulated using other instructions.

Register $R_3$ is used to store the combination. Its value is initialized to $1^t0^{w-t}$ by line 2, and its last $w - n$ bits will have value 0 throughout the course of the algorithm. Registers $R_0$ and $R_1$ are used as temporary variables. Register $R_2$ is used as a mask for the $(n+1)$st bit (its value is $0^n10^{w-n-1}$ by line 1). The algorithm terminates its loop on line 3 when the $(n+1)$st bit of $R_3$ is set to 1.

**Word**$(s,t)$
**Require:** $t > 0$
1: $R_2 \leftarrow (1 \ll s+t)$
2: $R_3 \leftarrow (1 \ll t) - 1$
3: **while** $R_3 \wedge R_2 = 0$ **do**
4: visit$(R_3)$
5: $R_0 \leftarrow R_3 \wedge (R_3 + 1)$
6: $R_1 \leftarrow R_0 \oplus (R_0 - 1)$
7: $R_0 \leftarrow R_1 + 1$
8: $R_1 \leftarrow R_1 \wedge R_3$
9: $R_0 \leftarrow (R_0 \wedge R_3) \ominus 1$
10: $R_3 \leftarrow R_3 + R_1 - R_0$
11: **end**

To understand the implementation, suppose $R_3 = 1^x 0 0^y 1 d s(\mathbf{b})$ where $d$ is a single bit. Line 5 places $0^{x+y+1} 1 d s(\mathbf{b})$ into $R_0$ (this is the value of $R_3$ with leading 1s changed to 0s). Line 6 places $1^{x+y+2} 0^{w-x-y-2}$ into $R_1$ (this is a mask for the shortest prefix ending 01 in $R_3$). Line 7 places $0^{x+y+2} 1 0^{w-x-y-3}$ into $R_0$ (this will be used as a mask for the bit with value $d$ in the fifth statement). Line 8 puts the value of $1^x 0^y 010^{w-x-y-2}$ into $R_1$ (this is the shortest prefix in $R_3$ ending in 01, with the remaining bits set to 0). If $d = 0$, then line 9 puts the value of $0^w$ into $R_0$, and then line 10 puts the correct value into $R_3$ via Lemma 6. If $d = 1$, then line 9 puts the value of $1^{x+y+2} 0^{w-x-y-2}$ into $R_0$, and then line 10 puts the correct value of into $R_3$ via Lemma 7 since $R_0 - R_1 = 0^x 1 1^y 0^{w-x-y}$.

## 4. Ranking Algorithm

In this section we examine the ranking functions of colex and cool-lex, and this provides another interesting link between the two lists. Given a listing of combinatorial structures, the *rank* of a particular structure is the number of structures that precede it in the listing.

Given an $(s,t)$-combination represented as a bitstring $b_1 b_2 \cdots b_n$ the corresponding set elements can be listed as $c_1 < c_2 < \cdots < c_t$ where $c_i$ is the position of the $i$-th 1 in the bitstring. As is well-known ([11],[16]) in colex order the rank of $c_1 c_2 \cdots c_t$ is

$$\sum_{j=1}^{t} \binom{c_i - 1}{i}. \tag{9}$$

As we see in the statement of the theorem below, in cool-lex order there is a very similar rank function. Let $rank(c_1 c_2 \cdots c_t)$ denote the rank of $c_1 c_2 \cdots c_t \in \mathbf{C}(s,t)$ in our order.

**Theorem 2.** *Let $r$ be the smallest index such that $c_r > r$ (so that $c_{r-1} = r - 1$). Then*

$$rank(c_1 c_2 \cdots c_t) = \binom{c_r}{r} - 1 + \sum_{j=r+1}^{t} \left( \binom{c_j - 1}{j} - 1 \right). \tag{10}$$

*Proof.* Directly from the recursive construction (5) we have

$$rank(b_1 b_2 \cdots b_n) = \begin{cases} rank(b_1 b_2 \cdots b_{n-1}) & \text{if } b_n = 0, \\ \binom{n}{t} - 1 & \text{if } b_1 b_2 \cdots b_n = 1^{t-1} 0^s 1, \\ \binom{n-1}{t-1} - 1 + rank(b_1 b_2 \cdots b_{n-1}) & \text{otherwise.} \end{cases}$$

We now consider the rank in terms of the corresponding list of elements $1 \le c_1 < c_2 < \cdots < c_t$. If $b_1 b_2 \ldots b_n = 1^{t-1} 0^s 1$, then $c_t = n$ and $c_{t-1} = t - 1$, so that $rank(c_1 c_2 \ldots c_t) = \binom{c_t}{t} - 1$. Otherwise, suppose

| $\mathbf{L}_{2,3}$ | | $\mathbf{R}_{2,3}$ | |
|---|---|---|---|
| 11100 | 123 | 11100 | 123 |
| 11010 | 124 | 01110 | 234 |
| 10110 | 134 | 10110 | 134 |
| 01110 | 234 | 11010 | 124 |
| 11001 | 125 | 01101 | 235 |
| 10101 | 135 | 10101 | 135 |
| 01101 | 235 | 01011 | 245 |
| 10011 | 145 | 00111 | 345 |
| 01011 | 245 | 10011 | 145 |
| 00111 | 345 | 11001 | 125 |

FIGURE 5. For $s = 2$ and $t = 3$, from left to right, colex as bitstrings, colex as elements, cool-lex as bitstrings, and cool-lex as elements. Only the third column is not genlex.

that $c_t = n - k$ for some $k \geq 0$, so that $b_n = b_{n-1} = \ldots = b_{n-k+1} = 0$ but $b_{n-k} = 1$. Then

$$\begin{aligned} rank(b_1 b_2 \ldots b_n) &= rank(b_1 b_2 \ldots b_{n-1}) \\ &= \ldots \\ &= rank(b_1 b_2 \ldots b_{n-k}) \\ &= \begin{cases} \binom{n-k}{t} - 1 & \text{if } b_1 b_2 \cdots b_{n-k} = 1^{t-1}0^{s-k}1, \\ \binom{n-k-1}{t-1} - 1 + rank(b_1 b_2 \cdots b_{n-k-1}) & \text{otherwise.} \end{cases} \end{aligned}$$

Therefore, in either case,

$$rank(b_1 b_2 \cdots b_n) = \begin{cases} \binom{c_t}{t} - 1 & \text{if } c_{t-1} = t-1 \text{ (that is, } t = r) \\ \binom{c_{t-1}-1}{t-1} - 1 + rank(c_1 c_2 \cdots c_{t-1}) & \text{otherwise (that is, } t > r). \end{cases} \tag{11}$$

By the definition of $r$, (10) follows from the first line of (11) by successive applications of the second line of (11). Note that (11), like (9) and 10, depends only on $t$ and not on $s$. ☐

Using standard techniques, as explained for example in [11] the expression in (10) can be evaluated in $O(n)$ arithmetic operations.

## 5. Genlex

A list of strings, $\mathbf{S} = \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m$, is *genlex* [21] if every suffix appears within consecutive strings in $\mathbf{S}$ (that is, if for every suffix, all the strings with that suffix form an interval of consecutive strings in $S$). In other words, $\mathbf{S}$ is genlex unless there exists a string $\mathbf{x}$, and integers $i, j, k$ with $1 \leq i < j < k \leq m$, such that $\mathbf{x}$ is a suffix of $\mathbf{s}_i$ and $\mathbf{s}_k$, but is not a suffix of $\mathbf{s}_j$. This property is common to several other orderings for combinations [11], and depending on the setting may be defined for prefixes instead of suffixes.

For example, when $s = 2$ and $t = 3$, Figure 5 illustrates that colex is genlex when it is represented by binary strings. This implies that colex is also genlex when it is represented by the elements contained in each combination, which is how genlex is defined for combinations in [11]. To verify this fact, note that every one-element suffix that appears in the list (3, 4, and 5) does so in consecutive strings, as does every two-element suffix (23, 24, 25, 34, 35, and 45), and every three-element suffix appears exactly once since $t = 3$. Remark 12 states that colex actually has a stronger property than genlex. In particular, the strings with a particular suffix give a smaller colex list.

**Remark 12.** *If $\mathbf{x}$ is a binary string with $s'$ zeros and $t'$ ones, then the strings in $\mathbf{L}_{s-s',t-t'}\mathbf{x}$ appear in the same order, and are consecutive, within $\mathbf{L}_{s,t}$. Moreover, no other strings in $\mathbf{L}_{s,t}$ have $\mathbf{x}$ as a suffix.*

On the other hand, Figure 5 shows that cool-lex is not genlex when it is represented by binary strings since the suffix 01 appears in three non-consecutive strings. However, we will show that cool-lex is genlex when it is represented by the elements contained in each combination. In terms of binary strings, this is equivalent to showing that every suffix that begins with a 1 appears in consecutive strings in cool-lex. For example, consider the combination 11001001 and its position vector 1 2 5 8. The suffixes beginning in 1 are 1, 1001, 1001001, and 11001001, and these suffixes correspond to the position vector suffixes 8, 5 8, 2 5 8, and 1 2 5 8, respectively. In Theorem 3, we will prove a result that is stronger than genlex for suffixes that begin with 1, and we will prove a result that is weaker than genlex for suffixes that begin with 0. For intuition, the reader may wish to refer to Figure 3, which illustrates the suffix properties that are maintained when colex is transformed into cool-lex. Before stating the theorem, we mention the following remark that follows directly from Remark 9.

**Remark 13.** *If $\mathbf{x}$ is a binary string with $s'$ zeros and $t'$ ones, then the strings in $\mathbf{R}_{s,t}$ with suffix $\mathbf{x}$ are exactly the strings within $\mathbf{R}_{s-s',t-t'}\mathbf{x}$ (or equivalently, within $\overrightarrow{\mathbf{R}_{s-s',t-t'}\mathbf{x}}$).*

**Theorem 3.** *(1) If $\mathbf{x} = 1\mathbf{x}'$ is a binary string with $s' \geq 0$ zeros and $t' > 0$ ones, then the strings in $\overrightarrow{\mathbf{R}_{s-s',t-t'}}\mathbf{x}$ appear in the same order, and are consecutive, within $\mathbf{R}_{s,t}$.*

*(2) If $\mathbf{x} = 0\mathbf{x}'$ is a binary string with $s' > 0$ zeros and $t' \geq 0$ ones, then the strings in $\overrightarrow{\mathbf{R}_{s-s',t-t'}}\mathbf{x}$, except for $last(\overrightarrow{\mathbf{R}_{s-s',t-t'}})\mathbf{x}$, appear in the same order, and are consecutive, within $\mathbf{R}_{s,t}$.*

*Proof.* In this proof we will make implicit use of Remarks 2, 9, and 13, Lemma 2, Theorem 1, and the definition of $\overrightarrow{\mathbf{S}}$. For the first result, if $s' \geq s$ or $t' \geq t$, then the result is immediate since $\mathbf{R}_{s-s',t-t'}$ contains at most one string. Otherwise, the string

$$\begin{aligned} 01^{t-t'}0^{s-s'-1}\mathbf{x} &= second(\mathbf{R}_{s-s',t-t'})\mathbf{x} \\ &= first(\overrightarrow{\mathbf{R}_{s-s',t-t'}})\mathbf{x} \end{aligned}$$

must occur somewhere within $\mathbf{R}_{s,t}$. Since $\mathbf{x} = 1\mathbf{x}'$, by Remark 2,

$$\sigma(\overrightarrow{\mathbf{R}_{s-s',t-t'}}[i])\mathbf{x} = \sigma(\overrightarrow{\mathbf{R}_{s-s',t-t'}}[i]\mathbf{x})$$

unless

$$\begin{aligned} \overrightarrow{\mathbf{R}_{s-s',t-t'}}[i] &= 1^{t-t'}0^{s-s'} \\ &= first(\mathbf{R}_{s-s',t-t'}) \\ &= last(\overrightarrow{\mathbf{R}_{s-s',t-t'}}). \end{aligned}$$

Therefore, the strings in $\overrightarrow{\mathbf{R}_{s-s',t-t'}}\mathbf{x}$ appear in the same order, and are consecutive, within $\mathbf{R}_{s,t}$.

For the second result, if $s' \geq s$ or $t' \geq t$, then the result is immediate since $\mathbf{R}_{s-s',t-t'}$ contains at most one string. Otherwise, the string

$$\begin{aligned} 01^{t-t'}0^{s-s'-1}\mathbf{x} &= second(\mathbf{R}_{s-s',t-t'})\mathbf{x} \\ &= first(\overrightarrow{\mathbf{R}_{s-s',t-t'}})\mathbf{x} \end{aligned}$$

must occur somewhere within $\mathbf{R}_{s,t}$. Since $\mathbf{x} = 0\mathbf{x}'$, by Lemma 1,

$$\sigma(\overrightarrow{\mathbf{R}_{s-s',t-t'}}[i])\mathbf{x} = \sigma(\overrightarrow{\mathbf{R}_{s-s',t-t'}}[i]\mathbf{x})$$

unless

$$\begin{aligned} \overrightarrow{\mathbf{R}_{s-s',t-t'}}[i] &= 1^{t-t'-1}0^{s-s'}1 \\ &= last(\mathbf{R}_{s-s',t-t'}). \end{aligned}$$

Notice that $last(\mathbf{R}_{s-s',t-t'})$ is the penultimate string in $\overrightarrow{\mathbf{R}_{s-s',t-t'}}$. Therefore, the strings in $\overrightarrow{\mathbf{R}_{s-s',t-t'}}\mathbf{x}$, except for $last(\overrightarrow{\mathbf{R}_{s-s',t-t'}})\mathbf{x}$, appear in the same order, and are consecutive, within $\mathbf{R}_{s,t}$. □ □

Before concluding this section, we provide a remark that follows directly from the recursive definition of cool-lex (4), and gives a slight strengthening of Theorem 3. This result was also mentioned in the proof of Theorem 2.

**Remark 14.** *If $\mathbf{x} = 0^{s'}$ for $0 \leq s' \leq s$, then the strings in $\mathbf{R}_{s-s',t}\mathbf{x}$ appear in the same order, and are consecutive, within $\mathbf{R}_{s,t}$.*

## 6. Final Remarks

6.1. **Permutations of a Multi-Set.** A multi-set is a collection of integers, with repetition allowed. For example, $S = \{2, 1, 1, 0\}$ is the multi-set with one copy of 2, two copies of 1, and one copy of 0. A permutation of a multi-set, or a *multi-perm*, is any arrangement of these integers. Notice that $(s, t)$-combinations are simply permutations of the multi-set with $s$ copies of 0 and $t$ copies of 1. Several algorithms exist for generating multi-perms [2, 10, 17, 19].

It appears that the iterative definition of cool-lex can be generalized to generate multi-perms. Starting from any multi-perm, let $\mathbf{p}$ be its shortest prefix ending with $xy$ where $x < y$. If there is no such prefix, then let $\mathbf{p}$ be the entire multi-perm. If $\mathbf{p}$ is not the entire multi-perm, and if the next symbol after $y$ is $z$, where $z \le x$, then let $\mathbf{p}$ instead be the prefix ending in $xyz$. Rotate the symbols in $\mathbf{p}$ by one position to the right. For example, the permutations of the multi-set $\{2, 1, 1, 0\}$ are:

$$2110, 0211, 2011, 1201, 0121, 1021, 2101, 1210, 1120, 0112, 1012, 1102.$$

Experimental results have validated the effectiveness of this iterative rule in a number of cases, and we hope to prove its correctness in a follow-up paper. When applied to binary multi-sets, the rule reduces to rotating the shortest prefix ending in 010 or 011 (notice that rotating a prefix ending in 011 is equivalent to rotating the same prefix ending in 01).

6.2. **Artistic Representations.** The iterative cool-lex list $\mathbf{R}_{s,t}$ has been rendered musically by George Tzanetakis and is available for download as a .wav file on the page `http://www.cs.uvic.ca/~ruskey/Publications/Coollex/Coollex.html`. A visual comparison of colex and cool-lex is illustrated artistically in the *The Feast* (ISBN 0-978066-30-98) and is available on `http://www.pmntmrkr.com/`.

6.3. **Open Problems.**

- Is it possible to generate combinations if the allowed operations are further restricted? For example, all permutations can be generated by letting the permutations $(1\ 2)$ and $(1\ 2\ \cdots\ n)$ and their inverses act on the indices, although this is not possible for combinations (for example, try $s = t = 3$).
- What is the fastest combination generator when carefully implemented? It would be interesting to undertake a comparative evaluation in a controlled environment, say of carefully implemented MMIX or ANSI C programs. Testing should be done in the four cases depending on whether the combination is represented by a single computer word, a linked list, an element of $\mathbf{B}(s, t)$, or an element of $\mathbf{C}(s, t)$. In the first three cases cool-lex should perform very well, however it will not perform as well in the last case since successive position vectors can change in an arbitrary number of positions. Thus, the cool-lex order on $\mathbf{C}(s, t)$ is not a Gray code and cannot be implemented as a loopless algorithm. Such a comparative evaluation should include every loopless algorithm. The authors mention the following algorithms: [7], [13], [3] and a simplified version [20], loopless implementations of [18] appear in [12] and [23], and a loopless implementation of [6] appears in [22].
- Suppose we view long prefixes as requiring more work to rotate than short prefixes. With respect to any possible algorithm that generates $\mathbf{B}(s, t)$, does cool-lex require the least amount of work? This question is motivated by the observation that cool-lex rarely rotates long prefixes.
- What is the computational complexity of determining if an arbitrary subset of $(s, t)$-combinations can be generated by prefix shifts?

## References


[1] P. F. Corbett, *Rotator Graphs: An Efficient Topology for Point-to-Point Multiprocessor Networks*, IEEE Transactions on Parallel and Distributed Systems, 3 (1992) 622–626

[2] P. J. Chase, *Algorithm 383: permutations of a set with repetitions*, CACM, 13 (1970), 368–369.

[3] P. J. Chase, *Combination Generation and Graylex Ordering*, Congressus Numerantium, 69 (1989) 215–242.

[4] P. Diaconis and S. Holmes, *Gray codes for randomization procedures*, Statistical Computing, 4 (1994) 207–302.

[5] P. Eades, M. Hickey and R. Read, *Some Hamilton Paths and a Minimal Change Algorithm*, Journal of the ACM, 31 (1984) 19–29.

[6] P. Eades and B. McKay, *An Algorithm for Generating Subsets of Fixed Size with a Strong Minimal Change Property*, Information Processing Letters, 19 (1984) 131–133.

[7] G. Ehrlich, *Loopless Algorithms for Generating Permutations, Combinations and Other Combinatorial Configurations*, Journal of the ACM, 20 (1973) 500–513.

[8] T. Hough and F. Ruskey, *An Efficient Implementation of the Eades, Hickey, Read Adjacent Interchange Combination Generation Algorithm*, Journal of Combinatorial Mathematics and Combinatorial Computing, 4 (1988), 79–86.
[9] M. Jiang and F. Ruskey, *Determining the Hamilton-connectedness of certain vertex-transitive graphs*, Discrete Mathematics, 133 (1994) 159–170.
[10] J. Korsh and S. Lipschutz, *Generating multiset permutations in constant time*, Journal of Algorithms, 25 (1997), 321–335.
[11] Donald E. Knuth, *The Art of Computer Programming, Volume 4: Generating all Combinations and Partitions*, Fascicle 3, Addison-Wesley, July 2005, 150 pages.
[12] C. N. Liu and D. T. Tang, *Algorithm 452: Enumerating Combinations of $m$ out of $n$ Objects*, Comm. ACM 16, (1973), 485.
[13] J. R. Ritner, G. Ehlrich, and E. M. Reingold, *Efficient Generation of the Binary Reflected Gray Code and its Applications*, Comm. ACM 19, (1976), 517–521.
[14] F. Ruskey, *Simple combinatorial Gray codes constructed by reversing sublists*, 4th ISAAC (International Symposium on Algorithms and Computation), Lecture Notes in Computer Science, #762 (1993) 201–208.
[15] F. Ruskey and A. Williams, *Generating Combinations By Prefix Shifts*, Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings. Lecture Notes in Computer Science 3595 Springer (2005).
[16] D. Stanton and D. White, *Constructive Combinatorics*, Springer-Verlag, 1986.
[17] T. Takaoka, *An O(1) Time Algorithm for Generating Multiset Permutations*, Proceedings of the 10th International Symposium on Algorithms and Computation, 25 (1999), 237–246.
[18] D. T. Tang and C.N. Liu *Distance-2 Cycle Chaining of Constant Weight Codes*, IEEE Transactions, **C-22** (1973) 176–180.
[19] V. Vajnovszki, *A loopless algorithm for generating the permutations of a multiset*, Theoretical Computer Science, 307 (2003), 415–431.
[20] V. Vajnovszki and T. R. Walsh, *A loopless two-close Gray-code algorithm for listing k-ary Dyck Words*, Journal of Discrete Algorithms, Vol. 4, No. 4 (2006) 633–648.
[21] T. R. Walsh, *A Simple Sequencing And Ranking Method That Works On Almost All Gray Codes*, Research report 243, Department of Mathematics and Computer Science, Université du Québec à Montréal, April 1995 (53 pages).
[22] T. R. Walsh, *Gray codes for involutions*, The Journal of Combinatorial Mathematics and Combinatorial Computing 36, (2001), 95–118.
[23] T. R. Walsh, *Generating Gray codes in O(1) worst-case time per word*, Lecture Notes in Computer Science 2731, Proceedings of the 4h International Conference, Discrete Mathematics and Theoretical Computer Science 2003, Dijon, France, July 7-12, 2003, Springer-Verlag, New York, (2003), 73–88.

Dept. of Computer Science, University of Victoria, CANADA