Generating Balanced Parentheses and Binary Trees by Prefix Shifts

XXXXX

XXXXX
XXXXX
XXXXX
XXXXX

Abstract

We show that the set $\mathbf{B}_n$ of balanced parentheses strings with n left and n right parentheses can be generated by prefix shifts. If $b_1, b_2, \ldots, b_{2n}$ is a member of $\mathbf{B}_n$, then the k-th prefix shift is the string $b_1, b_k, b_2, \ldots, b_{k-1}, b_{k+1}, \ldots, b_{2n}$. Prefix shift algorithms are also known for combinations, and permutations of a multiset; the combination algorithm appears in fascicles of Knuth vol 4. We show that the algorithm is closely related to the combination algorithm, and like it, has a loopless implementation, and a ranking algorithm that uses $O(n)$ arithmetic operations. Additionally, the algorithm can be directly translated to generate all binary trees by a loopless implementation that makes a constant number of pointer changes for each successively generated tree.

Keywords: Gray codes, Catalan numbers, balanced parentheses, binary trees, combinatorial generation, loopfree algorithm.

1 Introduction

Balanced parentheses strings are one of the most important of the many discrete structures that are counted by the Catalan numbers, $C_n = \binom{2n}{n}/(n+1)$. The Catalan numbers and the objects counted by them are extensively discussed in Stanley (1999). The online supplement lists 149 distinct discrete structures counted by the Catalan numbers (Stanley (2007)).

Binary trees and ordered trees are also counted by the Catalan numbers; these tree structures are of paramount importance to computer scientists. These is a large number of papers dealing with the fundamental problem of exhaustively listing and ranking binary trees. In this paper we develop an algorithm that has a number of attractive and unique features as compared with existing algorithms.

Let $\mathbf{B}_{t,s}$ be the set of all bitstrings containing t 1s and s 0s and satisfying the constraint that the number of 1s in any prefix is at least as large as the number of 0s. For example, $\mathbf{B}_{3,2} = \{11100, 11010, 11001, 10110, 10101\}$. In particular, $\mathbf{B}_{t,s}$ is empty if $t < s$. Furthermore, if $t = s$ then $\mathbf{B}_{t,s}$ can be thought of as the set of all balanced parentheses strings by mapping 1 to a left parenthesis and 0 to a right parenthesis. In this case, we sometimes drop the s from the notation; $\mathbf{B}_n = \mathbf{B}_{n,n}$.

If $b_1, b_2, \ldots, b_{2n}$ is a member of $\mathbf{B}_{t,s}$, then the k-th *prefix shift* is the string $b_1, b_k, b_2, \ldots, b_{k-1}, b_{k+1}, \ldots, b_{t+s}$. Note that the first bit, b_1 is not part of this definition; this is natural since b_1 is always 1. Furthermore, it is impossible to generate $\mathbf{B}_{t,s}$ as if b_1 is included in the shifts (e.g., 1^t0^s is the only valid shift of both $1^{t-1}0^s1$ and $1^{t-1}0^{s-1}10$). In order to entice the reader into reading further, below we show the simple iterative rule, whose successive application will generate $\mathbf{B}_{t,s}$ using prefix shifts.

Iterative successor rule: Locate the leftmost 01 and suppose that its 1 is in position k. If the $(k+1)$-st prefix shift is valid (a member of $\mathbf{B}_{t,s}$), then it is the successor; if it is not valid then the k-th prefix shift is the successor.

The only string without a 01 is 1^t0^s, which is the final string. The initial string is $101^{t-1}0^{s-1}$. Applying the rule to $\mathbf{B}_{3,2}$ gives the sequence 10110, 11010, 10101, 11001, 11100.

This is the first paper that considers whether balanced parentheses can be generated by prefix shifts. It is known that $\mathbf{B}_{t,s}$ can be generated by transposing a pair of bits (Ruskey & Proskurowski (1990)), a pair of bits with only 0s in between (Bultena & Ruskey (1998)), or by transposing one or two pairs of adjacent bits (Vajnovszki & Walsh (2006)). In general it is impossible to generate $\mathbf{B}_{t,s}$ by transposing only one pair of adjacent bits (Ruskey & Proskurowski (1990)). Our algorithm will be shown to generate $\mathbf{B}_{t,s}$ by transposing one or two pairs of bits, but those bits are not adjacent in general.

An algorithm for generating combinatorial objects is said to be *loopless* if only a constant amount of computation is used in transforming the current structure into its successor. Loopless algorithms are known for various classes of discrete structures that are counted by the Catalan numbers. See, for example, the papers Roelants (1991), Korsh, LaFolette, & Lipschutz (2003), Matos, Pinho, Silveira-Neto & Vajnovszki (1998), Vajnovszki & Walsh (2006) and Takaoka & Violich (2006).

There is a paper that shows that binary trees in their conventional representation of a node with two pointers can efficiently be generated by only making a constant number of pointer changes between successive trees (Lucas, Roelants, & Ruskey (1993)). This algorithm can be implemented looplessly and is presented in Knuth (2006). The current paper gives the basis for another such algorithm.

The approach taken in this paper was initiated in the papers of Ruskey & Williams (2005, 2008) for generating combinations that are represented by bitstrings in the usual way. There the bitstrings are also generated by prefix shifts. It is remarkable how many of the results of those papers have close analogues with the results of the current paper. The ordering of combinations in (Ruskey & Williams 2005, 2008) was called cool-lex order because of its close connection with the well-known colex order of combinations. In a similar spirit, we have dubbed our order "CoolCat" order because of its close connections with cool-lex order and with the Catalan numbers.

Relative to a list of objects, the *rank* of a particular object is the position that it occupies in the list, counting from zero.

To summarize, our method has the following properties:

1. Each successive string differs from its predecessor by the rotation of a prefix of the string. Furthermore, the list of strings is circular in the sense that the first and last also differ by a prefix rotation.

2. Each successive string differs from its predecessor by the interchange of one or two pairs of bits.

3. It has a simple recursive description. This description does not involve the reversal of sublist, as is usually the case for Gray codes. The underlying graph is a *directed* graph; that is, if $\mathbf{b}_1$ differs from $\mathbf{b}_2$ by a prefix rotation, then in general it is *not* the case that $\mathbf{b}_2$ differs from $\mathbf{b}_1$ by a prefix rotation.

4. It has a remarkably simple iterative successor rule. This rule was stated above.

5. The iterative successor rule can be implemented as a loopless algorithm. Also, the successor rule can be translated to a loopless algorithm for generating binary trees. No previous listing of balanced parentheses strings is simultaneously a Gray code for the strings *and* for the corresponding binary trees.

6. It has a ranking algorithm that uses $O(n)$ arithmetic operations. No previous Gray code for balanced parentheses strings has this property.

2 Generating Binary Trees

To give the reader a flavor of how useful the iterative successor rule is, in this section we translate the rule so that it applies to binary trees, as traditionally implemented on a computer. The result is a loopless algorithm that makes at most 14 pointer updates between successive trees. An implementation of this algorithm is available from the authors.

The standard bijection between $\mathbf{B}_{n,n}$ and extended binary trees with n internal nodes is to associate each internal node with a 1 and each leaf with a 0 and then do a preorder traversal of the tree, ignoring the final leaf. If z is a node in a binary tree, then we use $l(z)$ and $r(z)$ to denote the pointers to the left and right children of z. Unfortunately, we also need to maintain the parent of each internal node; this is denoted $p(z)$.

To update the tree we maintain three pointers: x, the first node that is not on the leftmost path of internal nodes; y, the parent of x; and R, the root of the tree. The assignments below represent parallel executions, so that, for example, $[a, b] \leftarrow [b, a]$ swaps the two values a and b. The algorithm terminates when x becomes nil.

According to the iterative successor rule there are three cases to consider: (a) the string is of the form $1^p0^q11\alpha$, (b) the string is of the form $1^p0^q10\alpha$, with $p > q$, and (c) the string is of the form 1^p0^p1. Below we show the updates that are necessary in each of

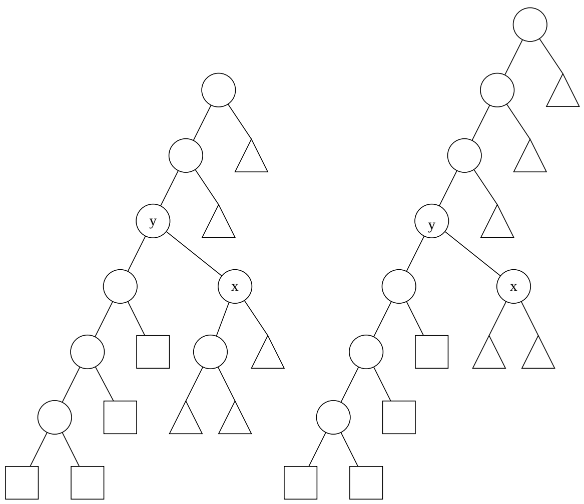

Figure 1: The trees corresponding to 111111000011... → 111111100001....

the three cases. Important note: The updates to the parent field are not shown explicitly below, but every time that an update is done to $r(.)$ or $l(.)$, then an update must be done to $p(.)$. E.g., if the update is $r(v) \leftarrow w$, then it should be followed with **if** $w \neq$ nil **then** $p(w) \leftarrow v$.

Case (a): The new string is $1^{p+1}0^q1\alpha$. This case occurs when $l(x) \neq$ nil. The corresponding update to the binary tree is

$$[r(y), r(x), l(x), l(y)] \leftarrow [r(x), l(x), l(y), x]$$

$$[y, x] \leftarrow [x, r(y)]$$

Case (b): The new string is $101^p0^q1\alpha$. This case occurs when $l(x) =$ nil and $R \neq y$. The corresponding update to the binary tree is

$$[l(p(y)), r(p(y)), l(x), r(x), l(y), r(y)] \leftarrow$$
$$[l(y), x, r(x), r(p(y)), \text{nil}, R]$$

$$[R, x] \leftarrow [y, r(y)]$$

Case (c): the new string is $1^{p+1}0^q\alpha$. This case occurs when $l(x) =$ nil and $R = y$. The corresponding update to the binary tree is

$$[l(x), r(y)] \leftarrow [y, l(x)];\ [R, y, x] \leftarrow [x, x, r(y)]$$

After this update the algorithm terminates if $x =$ nil.

These three cases are illustrated in Figures 1, 2, and 3. Circles are used for internal nodes, squares are used for leaves, and the triangles represent subtrees whose structure is not specified (but whose preorder order must be preserved).

3 Recursive Structure

In this section we examine the recursive structure of the CoolCat ordering on balanced parenthesis. In particular, we provide two recursive formulae and prove that they produce lists that are identical those produced by the iterative rule. A corollary to this result is that the iterative rule generates every string in $\mathbf{B}_{t,s}$. For comparison purposes we also provide the recursive structure for co-lexicographic, or colex ordering. We begin this section by giving a formal definition of the iterative rule.

The CoolCat iterative rule maps a binary string $\mathbf{b} \in \mathbf{B}_{t,s}$ to another binary string $\sigma(\mathbf{b}) \in \mathbf{B}_{t,s}$. When $\mathbf{b}$ does not contain any 010 or 011 as a substring then

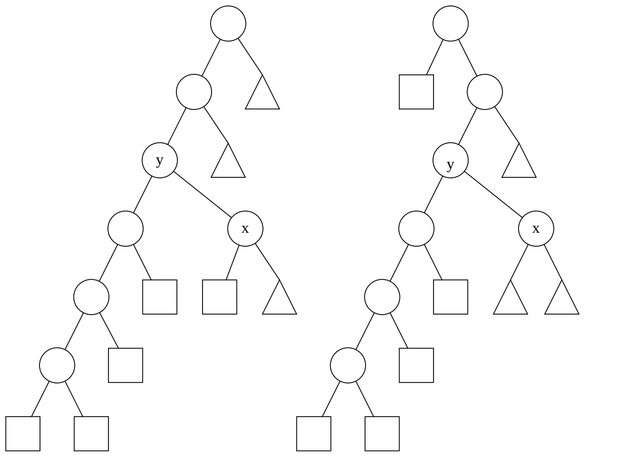

Figure 2: The trees corresponding to 111111000010... → 101111100001....

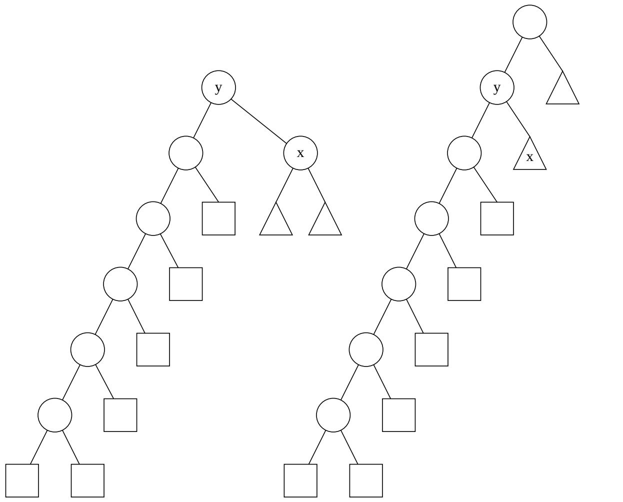

Figure 3: The trees corresponding to 11111000001... → 11111100000....

it is easiest to define $\sigma(\mathbf{b})$ using the following two special cases, which simply move the rightmost symbol into the second leftmost position.

$$\sigma(\mathbf{b}) = \begin{cases} 101^i0^j & \text{if } \mathbf{b} = 11^i0^j0 \quad \text{(1a)} \\ 111^i0^j0 & \text{if } \mathbf{b} = 11^i0^j01 \quad \text{(1b)} \end{cases}$$

Otherwise, we can assume that $\mathbf{b} = 11^i00^j1z\mathbf{b'}$ for some symbol z and some (possibly empty) string $\mathbf{b'}$.

$$\sigma(\mathbf{b}) = \begin{cases} 111^i00^jz\mathbf{b'} & \text{if } i = j \quad \text{(2a)} \\ 1z1^i00^j1\mathbf{b'} & \text{if } i > j \quad \text{(2b)} \end{cases}$$

We inductively let $\sigma^0(\mathbf{b}) = \mathbf{b}$ and $\sigma^k(\mathbf{b}) = \sigma(\sigma^{k-1}(\mathbf{b}))$ for $k > 0$, so that we can define an iterative list $\mathbf{R}_{t,s}$ that uses σ.

$$\mathbf{R}_{t,s} = \mathbf{b},\ \sigma(\mathbf{b}),\ \sigma^2(\mathbf{b}),\ \ldots, \sigma^{k-1}(\mathbf{b}) \quad (3)$$

where $\mathbf{b} = 1^t0^s$ and $k = |\mathbf{B}_{t,s}|$. We'll also find it useful to start the iterative process at the successor of $\mathbf{b}$, and in fact our first recursive structure will equal this secondary listing. Instead of starting the iterative process at the successor of $\mathbf{b}$, this secondary listing can also be seen as the result of applying σ to each string in $\mathbf{R}_{t,s}$.

$$\begin{aligned} \mathbf{S}_{t,s} &= \sigma(\mathbf{b}),\ \sigma^2(\mathbf{b}),\ \ldots,\ \sigma^k(\mathbf{b}) && (4) \\ &= \sigma(\mathbf{R}_{t,s}) && (5) \end{aligned}$$

To better illustrate our first recursive formula, let us begin by examining the recursive structure of the colex list $\mathbf{L}_{4,4}$ and then comparing it to the CoolCat list $\mathbf{S}_{4,4}$. The term colex refers to the fact that the strings in $\mathbf{B}_{t,s}$ are in increasing lexicographic order when each string is read from right to left. The colex list $\mathbf{L}_{4,4}$ can be built recursively from the smaller lists $\mathbf{L}_{3,i}$ for $0 \leq i \leq 3$. Each of these lists appears as a column within Figure 5. Notice that in each column the suffixes beginning with 1 are underlined, and all of the strings with a given underlined suffix appear consecutively. In the case of $\mathbf{L}_{4,4}$ (where $t = s$) the suffixes beginning with 1 are 10000, 1000, 100, and 10. Notice that there is no suffix 1 since there is no string in $\mathbf{B}_{4,4}$ with that suffix. However, the suffix 1 does appear in $\mathbf{L}_{3,2}$ (where $t > s$) since there is a string with that suffix in $\mathbf{B}_{3,2}$. Finally, in each case the suffixes are ordered by decreasing number of zeros. In general each of these observations holds true, and it leads to the following recursive formula for $\mathbf{L}_{t,s}$

$$= \begin{cases} \mathbf{L}_{t-1,0}10^s, \mathbf{L}_{t-1,1}10^{s-1}, \ldots, \mathbf{L}_{t-1,s-1}10 & \text{if } t = s \\ \mathbf{L}_{t-1,0}10^s, \mathbf{L}_{t-1,1}10^{s-1}, \ldots, \mathbf{L}_{t-1,s}1 & \text{if } t > s. \end{cases}$$

To compact expressions of this kind we introduce $\prod$ to combine short lists of strings into larger lists, and we restate the recursive formula for $\mathbf{L}_{t,s}$ as follows

$$\mathbf{L}_{t,s} = \begin{cases} \displaystyle\prod_{i=0}^{s-1} \mathbf{L}_{t-1,i}10^{s-i} & \text{if } t = s \quad \text{(6a)} \\ \displaystyle\prod_{i=0}^{s} \mathbf{L}_{t-1,i}10^{s-i} & \text{if } t > s. \quad \text{(6b)} \end{cases}$$

Now we turn our attention to the recursive structure of $\mathbf{W}_{4,4}$ that is illustrated in Figure 4. As in colex the suffixes beginning with 1 are underlined and the strings with a given underlined suffix appear consecutively within each list. However, in this case the suffixes beginning with 1 are ordered by decreasing

$\mathbf{L}_{3,0}$	$\mathbf{L}_{3,1}$	$\mathbf{L}_{3,2}$	$\mathbf{L}_{3,3}$	$\mathbf{L}_{4,4}$
$11\underline{1}$	$11\underline{10}$	$11\underline{100}$	$11\underline{1000}$	$111\underline{10000}$
	$110\underline{1}$	$110\underline{10}$	$110\underline{100}$	$1110\underline{1000}$
	$101\underline{1}$	$101\underline{10}$	$101\underline{100}$	$1101\underline{1000}$
		$1100\underline{1}$	$1100\underline{10}$	$1011\underline{1000}$
		$1010\underline{1}$	$1010\underline{10}$	$11100\underline{100}$
				$11010\underline{100}$
				$10110\underline{100}$
				$11001\underline{100}$
				$10101\underline{100}$
				$111000\underline{10}$
				$110100\underline{10}$
				$101100\underline{10}$
				$110010\underline{10}$
				$101010\underline{10}$

Figure 4: The recursive structure of colex.

$\mathbf{W}_{3,0}$	$\mathbf{W}_{3,1}$	$\mathbf{W}_{3,2}$	$\mathbf{W}_{3,3}$	$\mathbf{W}_{4,4}$
$11\underline{1}$	$101\underline{1}$	$101\underline{10}$	$101\underline{100}$	$1011\underline{1000}$
	$110\underline{1}$	$110\underline{10}$	$110\underline{100}$	$1101\underline{1000}$
	$11\underline{10}$	$1010\underline{1}$	$1010\underline{10}$	$1110\underline{1000}$
		$1100\underline{1}$	$1100\underline{10}$	$10110\underline{100}$
		$11\underline{100}$	$11\underline{1000}$	$11010\underline{100}$
				$10101\underline{100}$
				$11001\underline{100}$
				$11100\underline{100}$
				$101100\underline{10}$
				$110100\underline{10}$
				$101010\underline{10}$
				$110010\underline{10}$
				$111000\underline{10}$
				$111\underline{10000}$

Figure 5: The first recursive structure of CoolCat.

number of zeros, *except* for the suffix 10^s that appears last instead of first. Of course, there is only a single string in $\mathbf{B}_{t,s}$ that has the suffix 10^s, namely 1^t0^s. Amazingly, the alternate placement of this single string fully captures the difference between the recursive structure of CoolCat and colex. We define the list $\mathbf{W}_{t,s}$ as follows, and we prove that it is equal to $\mathbf{S}_{t,s}$ in Theorem 1

$$\mathbf{W}_{t,s} = \begin{cases} \prod_{i=1}^{s-1} \mathbf{W}_{t-1,i}10^{s-i},\ 1^t0^s & \text{if } t = s \quad (7a) \\ \prod_{i=1}^{s} \mathbf{W}_{t-1,i}10^{s-i},\ 1^t0^s & \text{if } t > s. \quad (7b) \end{cases}$$

Since the recursive structure of $\mathbf{W}_{t,s}$ is a reordering of the strings in $\mathbf{L}_{t,s}$ we have the following remark.

Remark 1. *$\mathbf{W}_{t,s}$ contains each string in $\mathbf{B}_{t,s}$ exactly once.*

An important step towards proving Theorem 1 is the following lemma, that explicitly identifies the first and last strings that appear in $\mathbf{W}_{t,s}$ when $s > 0$.

Lemma 1. *For $s > 0$*

$$first(\mathbf{W}_{t,s}) = 101^{t-1}0^{s-1} \quad (8)$$
$$last(\mathbf{W}_{t,s}) = 1^t0^s. \quad (9)$$

Proof. The value of $last(W_{t,s})$ follows immediately from (7). To determine the value of $first(\mathbf{W}_{t,s})$ we have the following

$$\begin{aligned} first(\mathbf{W}_{t,s}) &= first(\mathbf{W}_{t-1,1})10^{s-1} \\ &= first(\mathbf{W}_{t-2,1})110^{s-1} \\ &= first(\mathbf{W}_{t-3,1})1110^{s-1} \\ &= \ldots \\ &= first(\mathbf{W}_{1,1})1^{t-1}0^{s-1} \\ &= 101^{t-1}0^{s-1}. \end{aligned}$$

□

Now we are in a position to prove the main result of this section.

Theorem 1. $\mathbf{S}_{t,s} = \mathbf{W}_{t,s}$.

Proof. To prove the result we need to show that within $\mathbf{W}_{t,s}$ the first string in each sublist is obtained by applying σ to the last string of the previous sublist. The sublists in $\mathbf{W}_{t,s}$ are slightly different depending on whether $t = s$ (7a) or $t > s$ (7b), so we proceed in two cases. First we prove the result when $t > s$. For the last transition we have

$$\begin{aligned} \sigma(last(\mathbf{W}_{t-1,s}1)) &= \sigma(1^{t-1}0^s1) \\ &= 1^t0^s \end{aligned}$$

which follows from Lemma 1 and the definition of σ (1b). For the remaining transitions we have, for $1 \le i \le s-1$,

$$\begin{aligned} \sigma(last(\mathbf{W}_{t-1,s-i}10^i)) &= \sigma(1^{t-1}0^{s-i}10^i) \\ &= 101^{t-2}0^{s-i}10^{i-1} \\ &= first(\mathbf{W}_{t-1,s-i+1}10^{i-1} \end{aligned}$$

which follows from Lemma 1 and the definition of σ (2b). In particular, (2b) applies here since $t > s$ and $i \ge 1$ imply that $t-1 > s-i$.

Next we prove the result when $t = s$. For the last transition we have

$$\begin{aligned} \sigma(last(\mathbf{W}_{t-1,s-1}10)) &= \sigma(1^{t-1}0^{s-1}10) \\ &= 1^t0^s \end{aligned}$$

which follows from Lemma 1 and the definition of σ (2a). In particular, (2a) applies here since $t = s$. For the remaining cases we have, for $1 \le i \le s-2$,

$$\begin{aligned} \sigma(last(\mathbf{W}_{t-1,s-i}10^i)) &= \sigma(1^{t-1}0^{s-i}10^i) \\ &= 101^{t-2}0^{s-i}10^{i-1} \\ &= first(\mathbf{W}_{t-1,s-i+1}10^{i-1} \end{aligned}$$

which follows from Lemma 1 and the definition of σ (2b). In particular, (2b) applies here since $t = s$ and $i \ge 2$ imply that $t-1 > s-i$. □

Theorem 1 allows us to show that the iterative definition of CoolCat produces lists that are *circular*. That is, in both $\mathbf{R}_{t,s}$ and $\mathbf{S}_{t,s}$, the first string can be obtained by applying σ to the last string. More generally we have the following corollary.

Corollary 1. *For any $\mathbf{b} \in \mathbf{B}_{t,s}$ and $k = |\mathbf{B}_{t,s}|$*

$$\sigma^k(\mathbf{b}) = \mathbf{b}.$$

Proof. We can prove this result by showing that the list $\mathbf{S}_{t,s}$ is circular. This proves the statement of the corollary and also proves that $\mathbf{R}_{t,s}$ is circular by (4) and (3). We accomplish our goal through the following chain of equalities that reference Theorem 1, Lemma 1, and (1a)

$$\begin{aligned}\sigma(last(\mathbf{S}_{t,s})) &= \sigma(last(\mathbf{W}_{t,s})) \\ &= \sigma(1^t0^s) \\ &= 101^{t-1}0^{s-1} \\ &= first(\mathbf{W}_{t,s}) \\ &= first(\mathbf{S}_{t,s}).\end{aligned}$$

□

Theorem 1 also allows us to prove that the iterative definition of CoolCat generates every string in $\mathbf{B}_{t,s}$.

Corollary 2. $\mathbf{R}_{t,s}$ *and* $\mathbf{S}_{t,s}$ *contain each string in* $\mathbf{B}_{t,s}$ *exactly once.*

Proof. The result for $\mathbf{S}_{t,s}$ follows from Remark 1 and Theorem 1. The result for $\mathbf{R}_{t,s}$ follows from the fact that

$$\sigma^k(1^t0^s) = 1^t0^s$$

for $k = |\mathbf{B}_{t,s}|$ by Corollary 1, and thus $\mathbf{S}_{t,s}$ is a reordering of $\mathbf{S}_{t,s}$ by (3) and (4). □

Although the recursive definition of $\mathbf{W}_{t,s}$ has its benefits, sometimes it is more convenient to work with a recursive definition that contains fewer terms. For example, in Section 5 we rank the order of the strings within CoolCat utilizing the following definition

$$\mathbf{K}_{t,s} = \begin{cases} \mathbf{K}_{t,s-1}0 & \text{if } t = s \\ \mathbf{K}_{t-1,s}1,\ 1^{t-1}01 & \text{if } s = 1 \\ \mathbf{K}_{t,s-1}0,\ \mathbf{K}_{t-1,s}1,\ 1^{t-1}0^s1 & \text{if } 1 < s < t. \end{cases} \tag{10}$$

In Theorem 2 we prove that $\mathbf{K}_{t,s}$ is identical to $\mathbf{W}_{t,s}$ except that it is missing the string 1^t0^s. The proof is involved, so we provide an illustration for each of the three cases of (10) in Figure 6. In each column the overlined and underlined strings denote whether the number of zeros or ones are being recursively decreased, respectively. Strings without an overline or underline are of the form $1^{t-1}0^s1$ are not involved in the next lower level of recursion, while the strings below the horizontal line are of the form 1^t0^s and represent the unique string that is in $\mathbf{W}_{t,s}$ but is not in $\mathbf{K}_{t,s}$. For the sake of saving space we only produce the columns with a smaller number of zeros, until the number of zeros equals one.

$\mathbf{K}_{3,1}$	$\mathbf{K}_{4,1}$	$\mathbf{K}_{4,2}$	$\mathbf{K}_{4,3}$	$\mathbf{K}_{4,4}$
$101\underline{1}$	$1011\underline{1}$	$10111\overline{0}$	$101110\overline{0}$	$1011100\overline{0}$
1101	$1101\underline{1}$	$11011\overline{0}$	$110110\overline{0}$	$1101100\overline{0}$
	11101	$11101\overline{0}$	$111010\overline{0}$	$1110100\overline{0}$
		$10110\underline{1}$	$101101\overline{0}$	$1011010\overline{0}$
		$11010\underline{1}$	$110101\overline{0}$	$1101010\overline{0}$
		$10101\underline{1}$	$101011\overline{0}$	$1010110\overline{0}$
		$11001\underline{1}$	$110011\overline{0}$	$1100110\overline{0}$
		111001	$111001\overline{0}$	$1110010\overline{0}$
			$101100\underline{1}$	$1011001\overline{0}$
			$110100\underline{1}$	$1101001\overline{0}$
			$101010\underline{1}$	$1010101\overline{0}$
			$110010\underline{1}$	$1100101\overline{0}$
			1110001	$1110001\overline{0}$
1110	11110	111100	1111000	11110000

Figure 6: The second recursive structure for CoolCat.

Theorem 2. $\mathbf{W}_{t,s} = \mathbf{K}_{t,s},\ 1^t0^s$.

Proof. We prove the result by a double induction. The first induction will be on the number of zeros, and the second induction will be on the number of ones. For the base case of the first induction we have $s = 1$ and it is easy to verify that

$$\begin{aligned}\mathbf{W}_{t,1} &= \prod_{i=1}^{t-1} 1^i01^{t-i} \\ &= \prod_{i=1}^{t-2} 1^i01^{t-i},\ 1^t0 \\ &= \mathbf{K}_{t,1},\ 1^t0.\end{aligned}$$

Now suppose that $s = k > 1$ and that the theorem holds for all $s < k$. At this point we start our second induction. For the base case of the second induction we have $t = k$. In other words the number of ones is equal to the number of zeros, which is the minimum possible number of ones. We have the following expression for $\mathbf{W}_{k,k}$

$$\begin{aligned}&= \prod_{i=1}^{k-1} \mathbf{W}_{k-1,i}10^{k-i},\ 1^k0^k \\ &= \prod_{i=1}^{k-1} \mathbf{W}_{k-1,i}10^{k-1-i}0,\ 1^k0^{k-1}0 \\ &= \left(\prod_{i=1}^{k-1} \mathbf{W}_{k-1,i}10^{k-1-i},\ 1^k0^{k-1}\right)0 \\ &= (\mathbf{W}_{k,k-1})0 \\ &= (\mathbf{K}_{k,k-1}, 1^k0^{k-1})0 \\ &= \mathbf{K}_{k,k-1}0,\ 1^k0^k \\ &= \mathbf{K}_{k,k},\ 1^k0^k.\end{aligned}$$

Now to continue with the second induction we suppose that $t = k + j$, for some $j > 0$, and that the theorem holds for all $t < k + j$. In other words, we are supposing that there are j more ones than zeros, and that the theorem holds when the there are fewer than j additional ones. Then we have the following expression for $\mathbf{W}_{k+j,k}$

$$\begin{aligned}&= \prod_{i=1}^{k} \mathbf{W}_{k+j-1,i}10^{k-i},\ 1^{k+j}0^k \\ &= \prod_{i=1}^{k-1} \mathbf{W}_{k+j-1,i}10^{k-i},\ \mathbf{W}_{k+j-1,k}1,\ 1^{k+j}0^k \\ &= \left(\prod_{i=1}^{k-1} \mathbf{W}_{k+j-1,i}10^{k-1-i}\right)0,\ \mathbf{W}_{k+j-1,k}1,\ 1^{k+j}0^k.\end{aligned}$$

The bracketed product has fewer than k zeros and equals $\mathbf{W}_{k+j,k-1}$ except that it is missing $1^{k+j}0^{k-1}$ as its last string. Therefore, by the first induction we continue as follows

$$= \mathbf{K}_{k+j,k-1}0,\ \mathbf{W}_{k+j-1,k}1,\ 1^{k+j}0^k.$$

The second term has fewer than $k+j$ ones. Therefore, by the second induction we continue as follows

$$\begin{aligned}&= \mathbf{K}_{k+j,k-1}0,\ (\mathbf{K}_{k+j-1,k},\ 1^{k+j-2}0^k)1,\ 1^{k+j}0^k \\ &= \mathbf{K}_{k+j,k-1}0,\ \mathbf{K}_{k+j-1,k}1,\ 1^{k+j-2}0^k1,\ 1^{k+j}0^k \\ &= \mathbf{K}_{k+j,k},\ 1^{k+j}0^k.\end{aligned}$$

This completes the inductive case of the second induction, and so the theorem is true for $s = k$ and all $t \geq k$. This completes the inductive case of the first induction, and so the theorem is true for all $s \geq 1$. □

Before closing this section we explicitly state the first and last strings of $\mathbf{R}_{t,s}$ since it will be useful in the next section.

Lemma 2. *For $s > 0$*

$$\begin{aligned} first(\mathbf{R}_{t,s}) &= 1^t 0^s \\ last(\mathbf{R}_{t,s}) &= \begin{cases} 1^{t-1}0^s10 & \text{if } t = s \\ 1^{t-1}0^{s-1}1 & \text{if } t > s. \end{cases} \end{aligned}$$

4 Algorithm

In this section we present an algorithm to generate $\mathbf{R}_{t,s}$. That is, we present an algorithm that iteratively visits each successive string in the CoolCat ordering starting with 1^t0^s. The algorithm is remarkably efficient in terms of time and storage. In particular it is loopless in the sense that each successive string is generated in $\mathbf{O}(1)$ time, and it is constant extra-space in the sense that it uses $\mathbf{O}(1)$ storage when excluding the array b that holds the binary string. The array b uses 1-based indexing, so $b[1]$ is the first value in the array.

CoolCat(t, s)
Require: $t \geq s > 0$

```
1: n ← t + s
2: b ← array(1^t 0^s)
3: x ← t
4: y ← t
5: visit(b)
6: while x < n − (t = s) do
7:    b[x] ← 0
8:    b[y] ← 1
9:    x ← x + 1
10:   y ← y + 1
11:   if b[x] = 0
12:      if x = 2y − 2
13:         x ← x + 1
14:      else
15:         b[x] ← 1
16:         b[2] ← 0
17:         if y > 3
18:            x ← 3
19:         end
20:         y ← 2
21:      end
22:   end
23:   visit(b)
24: end
```

Besides b, the variables in the program are x and y, and their purpose will be explained after Lemma 4; n can be viewed as the constant $s + t$ (see line 1). We track the values of the three variables from one visit call to the next visit call by letting $b_1, b_2, \ldots$ represent the values taken by variable b at each subsequent visit, and we use the same convention for x and y. For example, b_1 will be the first and only value of b visited at line 5, while b_2 will be the first value of b visited at line 23. For convenience we also let $\mathbf{V}_{t,s} = b_1, b_2, \ldots, b_k$ where b_k is the last value of b that is visited before the program terminates. Ultimately we will show that the program does in fact terminate, and that $\mathbf{V}_{t,s} = \mathbf{R}_{t,s}$ (Theorem 3). We refer to the current values of b, x, and y as the current *configuration.* From lines 2-4 we see that $b_1 = 1^t0^s$, $x_1 = t$ and $y_1 = t$, so the initial configuration before entering the while loop is

$$b = 1^t0^s \quad y = t \quad x = t.$$

By Lemma 2, $first(\mathbf{R}_{t,s}) = 1^t0^s$ so b is initialized to the correct value. The program terminates once $x = n - (t = s)$ (line 6), where $(t = s)$ equals one if $t = s$, and zero otherwise. In other words, if $t = s$ then **CoolCat** terminates once $x = n - 1$, and otherwise it terminates once $x = n$. Recall that this condition echoes the two cases of (7). Finally, we point out **CoolCat**'s explicit requirement that $t \geq s > 0$. The next two lemmas will address the first two iterations of the algorithm.

Lemma 3. $\mathbf{V}_{t,s} = \mathbf{R}_{t,s}$ *when $t \leq 2$.*

Proof. It is easy to verify that $\mathbf{V}_{1,1} = 10$, $\mathbf{V}_{2,1} = 110, 101$, and $\mathbf{V}_{2,2} = 1100, 1010$. In the first case the program does not enter the while loop and in the last two cases the program terminates after the while loop's first iteration. □

Lemma 4. *If $t > 2$ then $b_2 = \sigma(b_1)$, $x_2 = 3$, and $y_2 = 2$.*

Proof. When $t > 2$ the program enters the while loop and after lines 7-10 we have the following configuration

$$b = 1^{t+1}0^{s-1} \quad y = t + 1 \quad x = t + 1.$$

Since $b[x] = b[t + 1] = 0$ the program enters the if statement on line 11. Since $t > 2$ it does not enter the if statement on line 12 and so lines 15 and 16 are executed to give the following configuration

$$b = 101^{t-1}0^{s-1} \quad y = t + 1 \quad x = t + 1.$$

Now since $y > 3$ the program enters the if statement on line 17. After line 18 and line 20 we have the following configuration

$$b = 101^{t-1}0^{s-1} \quad y = 2 \quad x = 3.$$

Since the next line to execute is a visit statement we have $b_2 = 101^{t-1}0^{s-1}$. Therefore, we have proven the result since $b_1 = 1^t0^s$ and $\sigma(1^t0^s) = 101^{t-1}0^{s-1}$ by (1a). □

At this point we are ready to explain the values of x and y. As long as $t > 2$ we have the following configuration for b_2, x_2, and y_2

$$b = 101^{t-1}0^{s-1} \quad y = 2 \quad x = 3.$$

We use x and y as indices into b, where y is the smallest index with $b[y] = 0$, and x is the smallest index with $b[x] = 1$ and $x > y$. In other words, y gives the location of the leftmost 0, and x gives the location of the leftmost 1 that appears after a 0. For example, when $t = 4$ and $s = 4$ then b_7, x_7, and y_7 give the following configuration

$$b = 11001100 \quad y = 3 \quad x = 5.$$

When x_i and y_i satisfy these conditions for b_i then we will say that x_i and y_i are *correct.* Since $b_1 = 1^t0^s$ is the only member of $\mathbf{B}_{t,s}$ without a 01 substring, there are correct values of x_i and y_i for every b_i except b_1. (The values of x_1 and y_1 were chosen to allow $b_2 = \sigma(b_1)$.) The next lemma explains how the algorithm terminates (the values for $last(\mathbf{R}_{t,s})$ are recalled from Lemma 2).

Lemma 5. *If $t > 1$, every $b_i \in \mathbf{B}_{t,s}$, and x_i is correct then*

$$last(\mathbf{V}_{t,s}) = last(\mathbf{R}_{t,s}) = \begin{cases} 1^{t-1}0^s10 & \text{if } t = s \\ 1^{t-1}0^{s-1}1 & \text{if } t > s \end{cases}$$

Proof. When $t = s$, the condition on the while loop is $x < n-1$. If $b_k = 1^{t-1}0^s10$ and x_k is updated correctly then $x_k = n-1$, so once b_k is visited the program will terminate. Furthermore, by (6a) we have that $x_i < n-1$ for all $i \neq k$ since by the assumption all $b_i \in \mathbf{B}_{t,s}$.

When $t > s$, the condition on the while loop is $x < n$. If $b_k = 1^{t-1}0^s1$ and x_k is updated correctly then $x_k = n$, so once b_k is visited the program will terminate. Furthermore, by (6b) we have that $x_i < n$ for all $i \neq k$ since by the assumption all $b_i \in \mathbf{B}_{t,s}$. ◻

Now that the extreme cases of **CoolCat** have been accounted for, we can focus on the general behavior of the algorithm. In particular, 1^t0^s and $1^{t-1}0^s1$ have been dealt with in Lemma 4 and Lemma 5 respectively, so we need only consider the behavior of the algorithm on binary strings that contain a leftmost 01 and at least one additional symbol following it. In other words, we assume that $b = 11^p00^q1z\ldots$ where $z \in \{0,1\}$. From Section 3 we recall our iterative definition for $\sigma(\mathbf{b})$

$$= \begin{cases} 111^p00^qz\ldots & \text{if } p = q \text{ and } z = 0 \quad (11a)\\ 1z1^p00^q1\ldots & \text{if } p > q \text{ or } z = 1. \quad (11b)\end{cases}$$

Notice that when $z = 1$ then the left side of (11a) and (11b) are identical. Therefore, we can interchange their roles when the condition of $z = 1$ is satisfied. Thus, the conditions in (11a) and (11b) can be equivalently stated as $p = q$ and $p > q$, respectively. In fact, the conditions were originally stated this way in (2a) and (2b); we make the change here since it optimizes the logic of the resulting program. Another way of stating the equivalence is that if $b = 11^p00^p11\ldots$ then it does not matter if we move the $(2p+3)$rd symbol or the $(2p+4)$th symbol since both are equal to 1. We now are able to complete this section with three lemmas. The first lemma corresponds to (11a), while the next two correspond to (11b).

Lemma 6. *Suppose $p = q$ and $z = 0$, so that $b_i = 11^p00^p10\ldots$. If x_i and y_i are correct, then $b_{i+1} = \sigma(b_i)$ and x_{i+1} and y_{i+1} are correct.*

Proof. From the statement of the lemma, we can assume that the current configuration appears below and the program just satisfied the condition of the while loop

$$b = 11^p00^p10\ldots \quad y = p+2 \quad x = 2p+3.$$

After executing lines 7-10 the current configuration becomes

$$b = 11^p10^p00\ldots \quad y = p+3 \quad x = 2p+4.$$

Since $b[x] = 0$ the program enters the if statement on line 11. Since $x = 2y-2$ the program enters the if statement on line 12. After executing line 13 the current configuration becomes

$$b = 11^p10^p00\ldots \quad y = p+3 \quad x = 2p+5.$$

At this point the program makes the next visit in line 23, so b_{i+1}, x_{i+1}, and y_{i+1} are equal to their respective values above. From (11a), $\sigma(b_i) = b_{i+1}$. Furthermore, the value of y_{i+1} is correct. However, can we be certain that the value of x_{i+1} is correct? Notice that the explicitly displayed portion of b in the above configuration contains an equal number of 1s and 0s. Hence, the next symbol must be 1, and so the value of x_{i+1} is also correct. ◻

Lemma 7. *Suppose $z = 1$, so that $b_i = 11^p00^q11\ldots$. If x_i and y_i are correct, then $b_{i+1} = \sigma(b_i)$ and x_{i+1} and y_{i+1} are correct.*

Proof. From the statement of the lemma, we can assume that the current configuration appears below and the program just satisfied the condition of the while loop

$$b = 11^p00^q11\ldots \quad y = p+2 \quad x = p+q+3.$$

After executing lines 7-10 the current configuration becomes

$$b = 11^p10^q01\ldots \quad y = p+3 \quad x = p+q+4.$$

Since $b[x] = 1$ the program does not enter the if statement on line 11 and so b_{i+1}, x_{i+1}, and y_{i+1} are equal to their respective values above. From (11b), $\sigma(b_i) = b_{i+1}$. Furthermore, the values of y_{i+1} and x_{i+1} are correct. ◻

Lemma 8. *Suppose $p > q$ and $z = 0$, so that $b_i = 11^p00^q10\ldots$ with $p > q \geq 0$. If x_i and y_i are correct, then $b_{i+1} = \sigma(b_i)$ and x_{i+1} and y_{i+1} are correct.*

Proof. From the statement of the lemma, we can assume that the current configuration appears below and the program just satisfied the condition of the while loop

$$b = 11^p00^q10\ldots \quad y = p+2 \quad x = p+q+3.$$

After executing lines 7-10 the current configuration becomes

$$\begin{aligned} b &= 11^p10^q00\ldots \quad y = p+3 \quad x = p+q+4\\ &= 111^p0^q00\ldots\end{aligned}$$

Since $b[x] = 0$ the program enters the if statement on line 11. Since $x = 2y-2$ would imply that $p+q+4 = 2p+4$ and $p = q$, then the if statement on line 12 is not entered. After executing lines 15 and 16 the configuration becomes

$$b = 101^p0^q01\ldots \quad y = p+3 \quad x = p+q+4.$$

The program enters the if statement on line 17 if and only if $p = 0$. However, $p > q \geq 0$ and so the program does not enter, and after executing line 20 the configuration becomes

$$b = 101^p0^q01\ldots \quad y = 2 \quad x = p+q+4.$$

At this point the program makes the next visit in line 23, so b_{i+1}, x_{i+1}, and y_{i+1} are equal to their respective values above. From (11b), $\sigma(b_i) = b_{i+1}$. Furthermore, the value of y_{i+1} is correct. Finally, the value of x_{i+1} is also correct since $p > 0$. ◻

The result of Lemmas 3-8 is that $\mathbf{CoolCat}(t,s)$ correctly visits and updates $first(\mathbf{R}_{t,s})$, and then correctly visits and updates every other string in $\mathbf{R}_{t,s}$ up to and including $last(\mathbf{R}_{t,s})$ after which it terminates. Therefore, we have the following theorem.

Theorem 3. $\mathbf{V}_{t,s} = \mathbf{R}_{t,s}$ *for all* $t \geq s > 0$.

5 Ranking

In this section we develop a ranking algorithm that uses $O(n)$ arithmetic operations. We will need to know the number of elements in $\mathbf{K}_{t,s}$, which we denote by $K_{t,s} = |\mathbf{B}_{t,s}| - 1$. Table 1 shows $K_{t,s}$ for $0 \leq s \leq t \leq 8$.

	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	2						
3	1	3	5	5					
4	1	4	9	14	14				
5	1	5	14	28	42	42			
6	1	6	20	48	90	132	132		
7	1	7	27	75	165	297	429	429	
8	1	8	35	110	275	572	1001	1430	1430

Table 1: The Catalan triangle. The row t, column s entry is $K_{t,s} = \frac{t-s+1}{t+1}\binom{t+s}{t}$.

Theorem 4. *For all* $0 \le s \le t$,

$$K_{t,s} + 1 = \frac{t-s+1}{t+1}\binom{t+s}{t} = \binom{t+s}{t} - \binom{t+s}{t+1}.$$

Proof. These are well-known properties of the "Catalan triangle" (Knuth (2006), Stanley (1999)). ☐

Let $\mathbf{b} = b_0 b_2 \cdots b_{t+s-1} \in \mathbf{B}_{t,s}$. We use $\rho(\mathbf{b})$ to denote the rank of $\mathbf{b}$ in the list $\mathbf{K}_{t,s}$. Here is a recursive description of the ranking process; it follows directly from (10). Let $\mathbf{b}' = b_0 b_2 \cdots b_{t+s-2}$.

$$\rho(\mathbf{b}) = \begin{cases} \rho(\mathbf{b}') & \text{if } b_{t+s-1} = 0 \\ K_{t,s} - 1 & \text{if } \mathbf{b} = 1^{t-1}0^s 1 \\ K_{t-1,s} + \rho(\mathbf{b}') & \text{otherwise.} \end{cases} \quad (12)$$

For example,

$$\begin{aligned} \rho(1010101) &= K_{4,2} + \rho(101010) \\ &= 8 + \rho(10101) \\ &= 8 + K_{3,1} + \rho(1010) \\ &= 8 + 2 + \rho(101) \\ &= 10 + K_{2,1} - 1 \\ &= 10 \end{aligned}$$

Note that (12) ignores trailing 0s; the rank therefore depends only on the positions of the 1s. If $c_1, c_2, \ldots, c_t$ are the positions occupied by the 1s and q is the minimum value for which $c_q > q$, then (12) can be iterated to obtain

$$\rho(c_1 c_2 \ldots c_t) = K_{q,c_q-q} - 1 + \sum_{j=q+1}^{t} K_{j,c_j-j-1}. \quad (13)$$

We now show that there is a nice way to view the ranking process as a walk on a certain integer lattice. Refer to Figure 7. The walk starts at the upper left; each 1 is a vertical step down and each 0 is a horizontal step to the right. The vertical edges are labeled, where the t-th row of vertical edges (counting from 1) gets labeled as follows from left-to-right: (no label), $K_{t,0}, K_{t,1}, \ldots, K_{t,t-1}$. The label furthest to the right in each row is not on an edge. Figure 7 illustrates the path for the bitstring 11100110101100. The square marks the endpoint of the part of the path that ends at the leftmost 01; i.e, the string 111001 in the example bitstring. The rank of the bitstring is obtained by summing the edge labels on the path after the square, adding the edge label on the edge to the right of the one that precedes the square (the circled label in the figure), and then subtracting 1. Thus $\rho(11100110101100) = 4 + 19 + 74 + 109 + 8 - 1 = 213$.

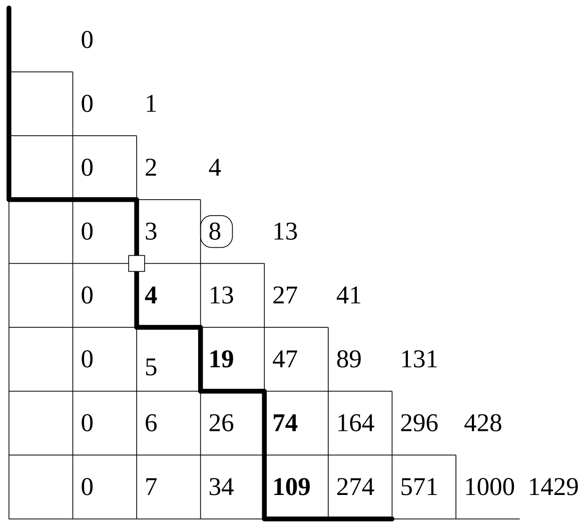

Figure 7: Ranking 11100110101100.

To unrank we reverse the process. We use $\rho^{-1}_{t,s}(m)$ to denote the string $\mathbf{b} \in \mathbf{B}_{t,s}$ whose rank in $\mathbf{K}_{t,s}$ is m. Suppose, for example, that we want the rank 212 bitstring with $t = 8$ and $s = 6$; i.e., $\rho^{-1}_{(8,6)}(212)$. We start where the example path ends. We move to the left so long as the edge labels exceed the remaining rank, then move up and repeat. Arriving at the old square, we are at an impasse; the remaining rank is 7, so we have yet to encounter the square. So we so up and the rank becomes 4, which is what remains if we make the current location (one move above the old square) the new square. Thus $\rho^{-1}_{(8,6)}(212) = 11001110101100$. We leave it to the reader to turn this description into an algorithm.

What is the running time of the ranking algorithm? Let $n = t+s$. Note that (12) and (13) involve $O(n)$ additions and other operations. We can avoid computing the entire table by only computing the values needed along the path. First compute $K_{t,s}$, which takes $O(n)$ arithmetic operations. Then make use of the following relations which can be checked using Theorem 4:

$$1 + K_{t-1,s} = \frac{(t+1)(t-s)}{(t-s+1)(t+s)}(1 + K_{t,s}) \text{ and}$$

$$1 + K_{t,s-1} = \frac{s(t-s+2)}{(t-s+1)(t+s)}(1 + K_{t,s}).$$

Of course, if many ranking/unranking operations are being performed then it will be better to pre-compute the $K_{t,s}$ table.

6 Final Remarks

For future research, it would be interesting to determine whether the results of this paper can be extended to the natural 0/1 representation of k-ary trees, or to ordered trees with prescribed degree sequence (Zaks & Richards (1979)).

References

B. Bultena & F. Ruskey (1998), *An Eades-McKay Algorithm for Well-Formed Parentheses Strings*, Information Processing Letters, 68, pp. 255–259.

Donald E. Knuth (2005), *The Art of Computer Programming, Volume 4: Generating all Combinations* *and Partitions*, Fascicle 3, Addison-Wesley, 150 pages.

Donald E. Knuth (2005), *The Art of Computer Programming, Volume 4: Generating all Trees; History of Combinationatorial Generation*, Fascicle 4, Addison-Wesley, 120 pages.

J. Korsh, P. LaFolette, & S. Lipschutz (2003), *Loopless Algorithms and Schröder Trees*, International Journal of Computer Mathematics, 80, pp. 709–725.

J. Lucas, D. Roelants, and F. Ruskey (1993), *On Rotations and the Generation of Binary Trees*, Journal of Algorithms, 15, pp. 343–366.

D. Roelants (1991), *A Loopless Algorithm for Generating Binary Tree Sequences*, Information Processing Letters, 39, pp. 184–194.

A.d. Matos, F.A.A. Pinho, A. Silveira-Neto & V. Vajnovszki (1998), *On the Loopless Generation of Binary Tree Sequences*, Information Processing Letters, 68, pp. 113–117.

S. Zaks & D. Richards (1979), *Generating Trees and Other Combinatorial Objects Lexicographically*, SIAM J. Computing, 8, pp. 73–81.

F. Ruskey (1979), *Simple combinatorial Gray codes constructed by reversing sublists*, 4th ISAAC (International Symposium on Algorithms and Computation), Lecture Notes in Computer Science, #762, pp. 201–208.

F. Ruskey and A. Proskurowski (1990), *Generating Binary Trees by Transpositions*, Journal of Algorithms, 11, pp. 68–84.

F. Ruskey & A. Williams (2005), *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-Verlag.

F. Ruskey and A. Williams (2008), *The Coolest way to Generate Combinations*, Discrete Mathematics, to appear, 2008.

R.P. Stanley (1999)*Enumerative Combinatorics, vol. 2*, Cambridge University Press, New York/Cambridge, 1999, xii + 581 pages.

R.P. Stanley (2007), *Catalan Addendum*, version of 20 June 2007; 61 pages, `http://www-math.mit.edu/~rstan/ec/`.

T. Takaoka (1999), $O(1)$ *Time Algorithms for Combinatorial Generation by Tree Traversal*, The Computer Journal, vol. 42, no. 5, pp. 400–408.

T. Takaoka & S. Violich (2006), *Combinatorial Generation by Fusing Loopless Algorithms*, In Proc. Twelfth Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. CRPIT, 51. Gudmundsson, J. and Jay, B., Eds., ACS. 69–77.

V. Vajnovszki & T. Walsh (2006), *A loopless two-close Gray-code algorithm for listing k-ary Dyck Words*, Journal of Discrete Algorithms, Vol. 4, No. 4, pp. 633–648.

R. Walsh, *A Simple Sequencing And Ranking Method That Works On Almost All Gray Codes*, Unpublished Research Report, Department of Mathematics and Computer Science, UQAM P.O. Box 8888, Station A, Montreal, Quebec, Canada H3C 3P8, 68 pages.

T. R. Walsh (2003), *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.

V.Vajnovszki & T. Walsh (2006), *A loop-free two-close Gray-code algorithm for listing k-ary Dyck words*, J. Discrete Algorithms 4(4), pp. 633–648.