O(1)-Time Unsorting by Prefix-Reversals in a Boustrophedon Linked List

Aaron Williams

Dept. of Computer Science, University of Victoria, Canada.

Abstract. Conventional wisdom suggests that $O(k)$-time is required to reverse a substring of length k. To reduce this time complexity, a simple and unorthodox data structure is introduced. A *boustrophedon linked list* is a doubly-linked list, except that each node does not differentiate between its backward and forward pointers. This lack of information allows substrings of any length to be reversed in O(1)-time. This advantage is used to efficiently unsort permutations using prefix-reversals. More specifically, this paper presents two algorithms that visit each successive permutations of $\langle n \rangle = \{1, 2, \ldots, n\}$ in worst-case O(1)-time (i.e. loopless). The first visits the permutations using a prefix-reversal Gray code due to Zaks [22], while the second visits the permutations in colexicographic order. As an added challenge, both algorithms are non-probing since they rearrange the data structure without querying its values. To accomplish this feat, the algorithms are based on two integer sequences: A055881 in the OEIS [17] and an unnamed sequence.

Key words: unsorting, permutations, lexicographic order, prefix-reversal, Gray code, loopless algorithm, boustrophedon linked list, BLL, integer sequences

1 Introduction

Suppose $\mathbf{p} = p_1 \ldots p_n$ is a string containing n symbols. The following operations replace its substring $p_i p_{i+1} \cdots p_{j-1} p_j$ where $1 \leq i < j \leq n$. A *transposition* of the ith and jth symbols replaces the substring by $p_j p_{i+1} \ldots p_{j-1} p_i$. A *shift* of the ith symbol into the jth position replaces the substring by $p_{i+1} \ldots p_{j-1} p_j p_i$. A *reversal* between the ith and jth symbols replaces the substring by $p_j p_{j-1} \ldots p_{i+1} p_i$.

Reversals are the most powerful of these operations. This is because any transposition or shift can be accomplished by at most two reversals. Similarly, reversals are also the most expensive of these operations. More precisely, $O(k)$-time is required to reverse a substring of length k in an array or linked list, whereas O(1)-time is sufficient for array transpositions and linked list shifts.

Reversals are a bottleneck when *unsorting permutations* in *lexicographic order*. Unsorting begins with the string 1 2 ... n and concludes when this string has been rearranged in all $n! - 1$ ways. (This is dual to sorting, which starts with an arbitrary string over $\langle n \rangle$ and rearranges it into 1 2 ... n.) In lexicographic order, worst-case $O(n)$-time is required to create successive permutations of $\langle n \rangle$

when the string is stored in an array or linked list. To illustrate why this is true, consider the permutations of $\langle 8 \rangle$ given below in lexicographic order

$$1\,2\,3\,4\,5\,6\,7\,8, \ldots, 3\,8\,7\,6\,5\,4\,2\,1, 4\,1\,2\,3\,5\,6\,7\,8, \ldots, 8\,7\,6\,5\,4\,3\,2\,1.$$

Let $\mathbf{p} = 3\ 8\ 7\ 6\ 5\ 4\ 2\ 1$ and $\mathbf{q} = 4\ 1\ 2\ 3\ 5\ 6\ 7\ 8$. Notice that no positions match in $\mathbf{p}$ and $\mathbf{q}$ ($p_i \neq q_i$ for all $1 \leq i \leq n$). Therefore, if the string is stored in an array then every entry needs to be changed when rearranging $\mathbf{p}$ into $\mathbf{q}$. Similarly, no substrings of length two match in $\mathbf{p}$ and $\mathbf{q}$ (i.e., $\{p_1p_2, p_2p_3, \cdots, p_{n-1}p_n\} \cap \{q_1q_2, q_2q_3, \cdots, q_{n-1}q_n\} = \emptyset$). Therefore, if the string is stored in a linked list then every forward pointer needs to be changed when rearranging $\mathbf{p}$ into $\mathbf{q}$.

The object of this paper is to reduce the worst-case from $\mathrm{O}(n)$-time to $\mathrm{O}(1)$-time. In other words, the object is to create a *loopless* algorithm for unsorting permutations in lexicographic order. This goal is achieved by using two integer sequences from Section 2, and a data structure from Section 4. The same goal is achieved for a second order of permutations discussed in Section 3.

This section concludes with additional context on (un)sorting, permutations, algorithms, and reversals. Reversing a prefix of a string is known as a *prefix-reversal*. *Pancake sorting* refers to sorting algorithms that use prefix-reversals, with focus on upper-bounds on the number of reversals used in sorting an arbitrary permutation [7, 2]. The minimum number of reversals needed to sort a particular permutation is known as *sorting by reversals*. This problem arises in biology when determining hereditary distance [1, 6]. In *burnt pancake sorting* [4] and *signed sorting by reversals* [9], each pancake or element has two distinct sides; the boustrophedon linked list is an analogous data structure (except individual nodes are unaware of their orientation). Algorithms for unsorting permutations were surveyed as early as the 1960s [14]. The term *loopless* was coined by Ehrlich [5], and loopless algorithms for unsorting permutations (using non-lexicographic orders) exist using transpositions in arrays [18, 8], and shifts in linked lists [13, 20]. However, lexicographic order has distinct advantages including linear-time ranking [15]. The subject of *combinatorial generation* is devoted to efficiently unsorting various combinatorial objects. Section 7.2.1 of Knuth's *The Art of Computer Programming* [10–12] is an exceptional reference.

2 Integer Sequences

This section defines two integer sequences, explains their connection to staircase strings, and discusses their efficient generation. The first sequence is denoted $\mathcal{R}$ and is known as OEIS A055881 [17], while the second sequence is denoted $\mathcal{T}$ and is otherwise unnamed. The first $4! = 24$ terms of each sequence appear below

$$\mathcal{R} = 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 4, \ldots \quad \text{(OEIS A055881)}$$
$$\mathcal{T} = \ 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 3, 1, 1, 1, 2, 1, 1, \ldots \quad \text{(unnamed)}.$$

The ith values in each sequence are respectively denoted r_i and t_i. The first $n! - 1$ values in each sequence are respectively denoted by $\mathcal{R}_n$ and $\mathcal{T}_n$. That is,

$\mathcal{R}_n = r_1, r_2, \ldots, r_{n!-1}$ and $\mathcal{T}_n = t_1, t_2, \ldots, t_{n!-1}$. The sequences can be defined recursively as follows: Let $\mathcal{R}_2 = r_1 = 1$ and $\mathcal{T}_2 = t_1 = 1$, and then for $n > 2$,

$$\mathcal{R}_n = \overbrace{\mathcal{R}_{n-1},\ n,\ \mathcal{R}_{n-1},\ n,\ \ldots,\ \mathcal{R}_{n-1},\ n,\ \mathcal{R}_{n-1},\ n,\ \mathcal{R}_{n-1}}^{n \text{ total copies of } \mathcal{R}_{n-1}} \tag{1}$$

$$\mathcal{T}_n = \mathcal{T}_{n-1},\ 1,\ \mathcal{T}_{n-1},\ 2,\ \ldots,\ \mathcal{T}_{n-1},\ n{-}2,\ \mathcal{T}_{n-1},\ n{-}1,\ \mathcal{T}_{n-1}. \tag{2}$$

To explain the origins of these sequences, let us consider the staircase strings of length $n-1$. A *staircase string* of length $n-1$ is a string $\mathbf{a} = a_1\, a_2 \cdots a_{n-1}$ with the property that $0 \le a_i \le i$ for each $1 \le i \le n-1$. In other words, the ith symbol can take on the $i+1$ values in $\{0, 1, \ldots, i\}$, and so staircase strings are simply the *mixed-radix strings* with bases $2, 3, \ldots, n$. Notice that there are $n!$ staircase strings of length $n-1$.

Sequences $\mathcal{R}$ and $\mathcal{T}$ can also be defined by their connection to staircase strings in co-lex order. (Strings are ordered from right-to-left in *co-lexicographic* (*co-lex*) order, and this order is often more convenient than lexicographic order.) When the ith staircase string in co-lex order is followed by the $(i+1)$st staircase string in co-lex order, then r_i is the position of the rightmost symbol that changes value, and t_i is the new value. For example, 1 2 3 4 1 0 0 $\cdots$ is the 240th staircase string in co-lex order and 0 0 0 0 2 0 0 $\cdots$ is the 241st staircase string in co-lex order. Therefore, $r_{240} = 5$ and $t_{240} = 2$. For a more complete example, the staircase strings of length 3 appear in column (a) of Table 1 in co-lex order, while columns (b) and (c) contain $\mathcal{R}_4$ and $\mathcal{T}_4$. (Table 1 appears on page 4 and columns (b) and (c) have been offset by half a row to emphasize the transition between successive staircase strings.) To generate $\mathcal{R}$ and $\mathcal{T}$, one can simply augment Algorithm M (Mixed-radix generation) in 7.2.1.1 of [10] while using $m_i = i$ for the bases, for all $1 \le i \le n-1$.

To generate $\mathcal{R}$ and $\mathcal{T}$ by a loopless algorithm, we can again follow [10]. Algorithm H in [10] generates multi-radix strings in *reflected Gray code* order. In this order, successive strings differ in ± 1 at a single position, and all "roll-overs" are suppressed. Furthermore, the position whose value changes is simply the rightmost position whose value changes in co-lex order. Within Algorithm H, each position is assigned a *direction* from $\{+1.-1\}$, and *focus pointers* determine the positions whose value will change. TR(n) gives pseudocode for Algorithm H to the right of Table 1 in a **Fun** function. Lines 10–15 output successive values of $\mathcal{R}$ and $\mathcal{T}$ into arrays r and t, which are indexed by $i = 1, 2, \ldots, n! - 1$. The staircase strings are stored in array a, the directions are stored in array d, and the focus pointers are stored in array f. Columns (d),(e), and (f) in Table 1 respectively provide the values stored in these three arrays when TR(n) is run with $n = 4$. (Within our programs all array indexing is 1-based, and if a is an array then $a[i]$ denotes its ith entry.) Using TR(n), successive entries of $\mathcal{R}$ and $\mathcal{T}$ can be computed in worst-case O(1)-time. In Section 5, we will modify TR(n) so that its ith iteration provides the r_ith and t_ith pointers into a (boustrophedon) linked list. The remaining columns in Table 1 are explained in Section 3.

staircase in co-lex			staircase in Gray code			perms in Zaks'		perms in co-lex			
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)
000			000	+++	1234	4321		4321			
100	1	1	100	−++	2234	3421	2	3421	1	2	1
010	2	1	110	−++	1234	2431	3	4231	1	3	2
110	1	1	010	+++	2234	4231	2	2431	1	2	1
020	2	2	020	+−+	1334	3241	3	3241	2	3	2
120	1	1	120	−−+	3234	2341	2	2341	1	2	1
001	3	1	121	−−+	1234	1432	4	4312	1	4	3
101	1	1	021	+−+	2234	4132	2	3412	1	2	1
011	2	1	011	+−+	1234	3142	3	4132	1	3	2
111	1	1	111	−−+	2234	1342	2	1432	1	2	1
021	2	2	101	−++	1334	4312	3	3142	2	3	2
121	1	1	001	+++	3234	3412	2	1342	1	2	1
002	3	2	002	+++	1234	2143	4	4213	2	4	3
102	1	1	102	−++	2234	1243	2	2413	1	2	1
012	2	1	112	−++	1234	4213	3	4123	1	3	2
112	1	1	012	+++	2234	2413	2	1423	1	2	1
022	2	2	022	+−+	1334	1423	3	2143	2	3	2
122	1	1	122	−−+	3234	4123	2	1243	1	2	1
003	3	3	123	−−−	1244	3214	4	3214	3	4	3
103	1	1	023	+−−	2244	2314	2	2314	1	2	1
013	2	1	013	+−−	1244	1324	3	3124	1	3	2
113	1	1	113	−−−	2244	3124	2	1324	1	2	1
023	2	2	103	−+−	1434	2134	3	2134	2	3	2
123	1	1	003	++−	4234	1234	2	1234	1	2	1

```
% Loopless R_n and T_n
Fun TR(n)
 1: r := [−1,...,−1] (size n!−1)
 2: t := [−1,...,−1] (size n!−1)
 3: f := [1,2,...,n] (size n)
 4: d := [1,1,...,1] (size n − 1)
 5: a := [0,0,...,0] (size n − 1)
 6: i := 1
 7: while f[1] < n
 8:     j := f[1]
 9:     f[1] := 1
10:     r[i] := j
11:     if d[j] = 1
12:         t[i] := a[j] + 1
13:     else
14:         t[i] := j − a[j] + 1
15:     end if
16:     a[j] := a[j] + d[j]
17:     if a[j] = 0 OR a[j] = j
18:         d[j] := −d[j]
19:         f[j] := f[j + 1]
20:         f[j + 1] := j + 1
21:     end if
22:     i := i + 1
23: end while
```

Table 1. Relationships between staircase strings, permutation orders, and sequences. Columns (b), (c), (h), (j), (k), (m) respectively contain $\mathcal{R}_4$,$\mathcal{T}_4$,$\mathcal{R}_4+1$,$\mathcal{T}_4$,$\mathcal{R}_4+1$,$\mathcal{R}_4$. Sequences $\mathcal{R}_n$ and $\mathcal{T}_n$ are generated by loopless algorithm TR(n) in arrays r and t.

3 Permutations of Permutations

This section discusses two orders (or permutations) of the permutations of $\langle n \rangle$. The first is Zaks' prefix-reversal Gray code [22], and the second is co-lex order. These two orders can be created from the sequences found in Section 2. In particular, $\mathcal{R}$ and $\mathcal{T}$ have been named after the Reversal and Transposition operations that produce the two orders.

Zaks' original paper describes a poor waiter who must flip n pancakes of different sizes into all $n!$ possible stacks. In order to do this, Zaks proposes an order where "in 1/2 of these steps he will reverse the top 2 pancakes, in 1/3 of them the top 3, and, in general, in (k-1)/k! of them he will reverse the top k pancakes". As Zaks points out, the waiter can achieve such an ordering directly from sequence $\mathcal{R}$. In particular, his $(i+1)$st stack is obtained from his ith stack by flipping the top r_i+1 pancakes. In the parlance of permutations, Zaks proves that a prefix-reversal Gray code for the permutations of $\langle n \rangle$ is obtained by successively reversing the prefix of length r_i+1 in the ith permutation. This is illustrated for

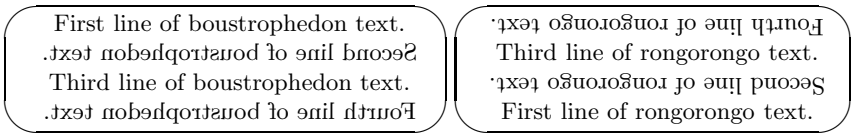
$n = 4$ in Table 1, with columns (g) and (h) providing the permutations of $\langle 4 \rangle$ in Zaks' order and $\mathcal{R}_4 + 1$. (For convenience, $\mathcal{R}_n + 1$ denotes the sequence obtained by adding 1 to each entry in sequence $\mathcal{R}_n$.) Zaks' paper shows how to generate $\mathcal{R}_n + 1$ using a loopless algorithm, and also proves that the average value of $\mathcal{R}_n + 1$ is less than $e \cong 2.8$. For this reason, it is possible to generate Zaks' prefix-reversal Gray code with a *constant amortized-time algorithm (CAT)*, meaning that each successive permutation is created in amortized O(1)-time. Such an algorithm can be constructed from TR(n) by initializing an array $p = [1, 2, \ldots, n]$, and then by replacing lines 10–15 with a loop that reverses its first $j + 1$ entries.

As Knuth points out at the beginning of Section 7.2.1.2 in *The Art of Computer Programming* [10], an iterative algorithm for creating the permutations of $\langle n \rangle$ in co-lex order dates back to the 14th-century. This historical algorithm is based on scanning the current permutation. Less obviously, co-lex order can be created using $\mathcal{R}$ and $\mathcal{T}$. In particular, the $(i + 1)$st permutation can be obtained from the ith permutation in two steps. In the first step, the symbols in positions t_i and $r_i + 1$ are transposed. In the second step, the prefix of length r_i is reversed. This is illustrated for $n = 4$ in Table 1. Column (i), (j), (k), and (l) provide the permutations of $\langle 4 \rangle$ in co-lex order, $\mathcal{T}_4$, $\mathcal{R}_4 + 1$, and $\mathcal{R}_4$. As in the case of Zaks' algorithm, TR(n) can be modified to provide an array-based CAT algorithm for creating permutations in co-lex order.

By using sequences $\mathcal{R}$ and $\mathcal{T}$, the difference between successive permutations in Zaks' order and co-lex order can be described in worst-case O(1)-time. Section 4 will allow these differences to be performed in worst-case O(1)-time.

4 Boustrophedon Linked Lists

In Greek, *βουστροφηδόν* means "as the ox turns while plowing". In English, *boustrophedon* is commonly used to describe ancient texts that are reflected horizontally on every second line [19]. A variation from Rapa Nui is *rongorongo* that rotates every second line 180°.



The term boustrophedon also describes back-and-forth motion without reflections or rotations. For example, dot-matrix printers are more efficient when printing *boustrophedonically*. Similarly, Major League Eating world-recorder holder Jason "Crazy Legs" Conti often eats sweet corn in a *boustrophedonic* pattern [21]. Scientifically, the *boustrophedon transform* and *boustrophedon cellular decomposition* appear in mathematics [16] and computer science [3], respectively.

As in a standard doubly-linked list (DLL), the first node of a *boustrophedon linked list* (BLL) is the *head* and the last node is the *tail*. Similarly, each node in a BLL has a *value* and is connected to its two *adjacent* nodes by a *forward* pointer (.fwd) and a *backward* pointer (.bwd). However, in a BLL the forward pointers do

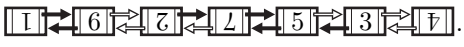
not necessarily point towards the tail. In other words, the backward and forward pointers of any node in a BLL can be independently interchanged. This includes the head (resp. tail), where one pointer must be NULL and the other is directed to the second (resp. second-last) node. Nodes are *normal* or *reversed* if their forward pointer is directed towards the tail or head, respectively.

Despite the uncertainty at each node, a BLL containing n nodes can be traversed in $O(n)$-time. Furthermore, this $O(n)$-time traversal could transform the BLL into a DLL by "flipping" each reversed node. As in a DLL, nodes in a BLL can be inserted, removed, transposed, and shifted in worst-case $O(1)$-time. However, a BLL has a distinct advantage over a DLL: any sublist can be reversed in worst-case $O(1)$-time. This section describes these results, and builds a small "boustrophedon toolkit" that includes the concepts of twin-pointers and balancing. Again it is stressed that no additional directional information is stored within each node of the BLL nor within an auxiliary data structure. (One can consider BLL alternatives that add an extra bit of information to each node of a DLL. This bit could not denote whether a node is normal or reversed without precluding $O(1)$-time reversals. However, this worst-case time could be obtained by having the first i bits determine whether the ith node is normal or reversed.)

Before discussing these results, let us have fun considering pictographic representations for a BLL. In each proposal the nodes appear left-to-right from head-to-tail, with backward pointers in black (⟶), forward pointers in white (⇒), and NULL pointers omitted at the head and tail. Thus, normal and reversed internal nodes appear respectively as ⟵□⇒ and ⇐□⟶. A BLL storing the permutation $\mathbf{p} = 1\ 6\ 2\ 7\ 5\ 3\ 4$ is shown below in its *standard representation*

From the head 1 to the tail 4, successive nodes are reached by traveling backwards, forwards, backwards, backwards, forwards, and forwards (as seen by the pointers on the top row). To draw attention to the reversed nodes, we can also represent this BLL using its *boustrophedonic representation* (where the values of reversed nodes are reflected horizontally)

or its *rongorongo representation* (where reversed nodes values are rotated 180°)



Rongorongo representation is nice since reversals are visualized as 180° rotations.

In a BLL it is often helpful to use two pointers when one would suffice in a DLL. This *twin-pointer* approach is demonstrated for traversing the nodes in a BLL from head to tail. Say that pointers x and y are *adjacent* if they point to two nodes that are adjacent in the BLL. When x and y are adjacent pointers, then x's *other* node is its adjacent node that is not y. (If x is the head or tail of the BLL, then either x's other node or y will be NULL.) The simple logic needed to determine other(x, y) appears below, (All variables in this section are pointers to

nodes in a BLL.) Using this routine we can easily traverse a BLL in traverse(head), where pointer x traverses the BLL while pointer y follows one node behind, and t is a temporary variable.

```
% Traverse BLL
Fun traverse(head)
  y := NULL
  x := head
  while x ≠ NULL
    t := x
    x := other(x, y)
    y := t
  end while
```

```
% Other neighbor of x
Fun other(x, y)
  if x.fwd = y
    return x.bwd
  else if x.bwd = y
    return x.fwd
  end if
```

```
% Flip reversed ↔ normal
Fun flip(x)
  t := x.fwd
  x.fwd := x.bwd
  x.bwd := t
```

Clearly, traverse(head) takes $O(n)$-time when the BLL contains n nodes. The above code can be expanded to convert the BLL into a DLL in $O(n)$-time by remembering the orientation of successive nodes starting from head and "flipping" those nodes that are reversed. Pseudocode for flip(x) appears above and also changes normal nodes into reversed nodes.

Another way to overcome the lack of information in a BLL is to make a portion of the BLL behave like a standard DLL. In a DLL, x.fwd $= y$ implies y.bwd $= x$, so long as $y \neq$ NULL. In other words, proceeding forwards and then backwards will return you to the initial node, so long as the intermediate node exists. On the other hand, the same statement is not true in a BLL unless x and y both point to normal nodes, or reversed nodes. We say that an adjacent pair of nodes $\{x, y\}$ is *balanced* or *imbalanced* depending on isBalanced(x, y).

```
% Is adjacent pair {x, y} balanced?
Fun isBalanced(x, y)
  return (x = NULL) OR (y = NULL) OR
  (x.fwd = y AND y.bwd = x) OR
  (x.bwd = y AND y.fwd = x)
```

```
% Balance pair {x, y}
Fun balance(x, y)
  if NOT isBalanced(x, y)
    flip(x)
  end if
```

Pictorially, adjacent nodes are balanced if and only if there is one white and one black pointer between them. For example, the previously-drawn BLL for $\mathbf{p} = 1\ 6\ 2\ 7\ 5\ 3\ 4$ appears below (in its boustrophedonic representation)

Notice that the only balanced pairs are the nodes containing $\{2, 7\}$ and $\{5, 3\}$. Any pair of adjacent nodes can be assured to be balanced by calling balance(x, y). For example, flipping the node containing 2 causes the nodes containing $\{6, 2\}$ to become balanced at the expense of imbalancing the nodes containing $\{2, 7\}$

When x is not in an imbalanced pair, then x is a *balanced node*. Otherwise, if x is in an imbalanced pair, then x is an *imbalanced node*. To change a node from imbalanced to balanced, either the node itself or one of its neighbors must be flipped. Any node can be assured to be balanced by calling balance(x).

```
% Balance x
Fun balance(x)
  if NOT isBalanced(x, x.fwd) AND NOT isBalanced(x, x.bwd)
    flip(x)
  else if NOT isBalanced(x, x.fwd)
    flip(x.fwd)
  else if NOT isBalanced(x, x.bwd)
    flip(x.bwd)
  end if
```

By balancing nodes and pairs of nodes, BLL deletions and insertions proceed as in a DLL. In turn, these routines allow simple worst-case O(1)-time transpositions and shifts. Balancing also allows simple worst-case O(1)-time reversals.

```
% Remove node x
Fun delete(x)
  balance(x)
  x.bwd.fwd := x.fwd
  x.fwd.bwd := x.bwd

% Insert x between u v
Fun insert(x, u, v)
  x.bwd := v
  x.fwd := u
  balance(u, v)
  if u.fwd = v
    u.fwd := x
    v.bwd := x
  else
    u.bwd := x
    v.fwd := x
  end if

% Shift x between u v
Fun shift(x, u, v)
  if x = u OR x = v
    return
  end if
  delete(x)
  insert(x, u, v)
```

```
% Transpose x and y
Fun transpose(x, y)
  if x = y
    return
  end if
  if x = NULL
    return
  else if y = NULL
    return
  end if
  balance(x)
  f_x := x.fwd
  b_x := x.bwd
  delete(x)
  if f_x = y
    insert(x, y, y.fwd)
  else if b_x = y
    insert(x, y, y.bwd)
  else
    f_y := y.fwd
    b_y := y.bwd
    delete(y)
    insert(x, f_y, b_y)
    insert(y, f_x, b_x)
  end if
```

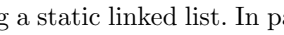
```
% Reverse between u x
% through between y v
Fun reverse(u, x, y, v)
  if x = y
    return
  end if
  balance(u, x)
  balance(y, v)
  f_x := x.fwd
  f_y := y.fwd
  if f_x = u
    x.fwd = v
    u.bwd = y
  else
    x.bwd = v
    u.fwd = y
  end if
  if f_y = v
    y.fwd = u
    v.bwd = x
  else
    y.bwd = u
    v.fwd = x
  end if
```

In shift(x, u, v), u and v are adjacent. In reverse(u, x, y, v), u and x are adjacent, as are y and v; nodes between x and y are reversed while u and v are stationary.

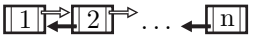
5 Algorithms

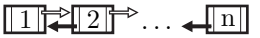
At this point we have seen how Zaks' order and lexicographic order can be created using transpositions and reversals (Section 3), we have shown that the indices

r_i and t_i that describe these transpositions and reversals can be computed in worst-case O(1)-time (Section 2), and we have presented a data structure that allows for worst-case O(1)-time transpositions and reversals (Section 4). This section ties these results together by providing worst-case O(1)-time algorithms for creating permutations in Zaks' order and in lexicographic order.

The first hurdle is that TR(n) creates integers r_i and t_i, while pointers to the r_ith and t_ith nodes are required for BLL transpositions and reversals. As an intermediate step, algorithms R(n) and T(n) create the sequences $\mathcal{R}_n$ and $\mathcal{T}_n$ using a static linked list. In particular, head is initialized to 

which can be viewed as a standard DLL or a BLL with no reversed nodes. At each iteration, a pointer points to its r_ith node (on line 17 in R(n)) and the t_ith node (on line 17 in T(n)) while the linked list is never changed.

```
 % Loopless generation of R_n in a BLL
Fun R(n)
 1: head := 
 2: R := [NULL, NULL, ..., NULL] (size n)
 3: r := [-1, -1, ..., -1] (size n!-1)
 4: f := [1, 2, ..., n] (size n)
 5: d := [1, 1, ..., 1] (size n - 1)
 6: a := [0, 0, ..., 0] (size n - 1)
 7: i := 1
 8: while f[1] < n
 9:    j := f[1]
10:    f[1] := 1
11:    a[j] := a[j] + d[j]
12:    if R[1] = NULL
13:       R[1] := head
14:    end if
15:    Rnode := R[1]
16:    R[1] := NULL
17:    r[i] := Rnode.value
18:    if a[j] = 0 OR a[j] = j
19:       d[j] := -d[j]
20:       f[j] := f[j + 1]
21:       f[j + 1] := j + 1
22:       if R[j + 1] = NULL
23:          R[j] := Rnode.fwd
24:       else
25:          R[j] := R[j + 1]
26:       end if
27:       R[j + 1] := NULL
28:    end if
29:    i := i + 1
30: end while
```

```
 % Loopless generation of T_n in a BLL
Fun T(n)
 1: head := 
 2: T := [NULL, NULL, ..., NULL] (size n)
 3: t := [-1, -1, ..., -1] (size n!-1)
 4: f := [1, 2, ..., n] (size n)
 5: d := [1, 1, ..., 1] (size n - 1)
 6: a := [0, 0, ..., 0] (size n - 1)
 7: i := 1
 8: while f[1] < n
 9:    j := f[1]
10:    f[1] := 1
11:    a[j] := a[j] + d[j]
12:    if T[j] = NULL
13:       T[j] := head
14:    end if
15:    Tnode := T[j]
16:    T[j] := Tnode.fwd
17:    t[i] := Tnode.value
18:    if a[j] = 0 OR a[j] = j
19:       d[j] := -d[j]
20:       f[j] := f[j + 1]
21:       f[j + 1] := j + 1
22:       T[j] := NULL
23:    end if
24:    i := i + 1
25: end while
```

```
% Loopless permutations in Zaks' order
Fun Zaks(n)
 1: head := [n] ⇨ ... ⇦ [2] ⇨ [1]
 2: R := [NULL, NULL, ..., NULL] (size n)
 3: S := [NULL, NULL, ..., NULL] (size n)
 4: f := [1, 2, ..., n] (size n)
 5: d := [1, 1, ..., 1] (size n − 1)
 6: a := [0, 0, ..., 0] (size n − 1)
 7: while f[1] < n
 8:    j := f[1]
 9:    f[1] := 1
10:    a[j] := a[j] + d[j]
11:    if R[1] = NULL
12:       R[1] := head
13:       S[1] := other(head, NULL)
14:    end if
15:    Rnode := R[1]
16:    Snode := S[1]
17:    R[1] := NULL
18:    S[1] := NULL
19:    if a[j] = 0 OR a[j] = j
20:       d[j] := −d[j]
21:       f[j] := f[j + 1]
22:       f[j + 1] := j + 1
23:       if R[j + 1] = NULL
24:          temp := R[j]
25:          R[j] := S[j]
26:          S[j] := other(S[j], temp)
27:       else
28:          R[j] := R[j + 1]
29:          S[j] := S[j + 1]
30:       end if
31:       R[j + 1] := NULL
32:       S[j + 1] := NULL
33:    end if
34:    temp := other(Snode, Rnode)
35:    reverse(temp, Snode, head, NULL)
36:    if f[j] = j + 1
37:       R[j] := head
38:    end if
39:    head := Snode
40:    visit(head)
41: end while
```

```
% Loopless permutations in co-lex order
Fun Lex(n)
 1: head := [n] ⇨ ... ⇦ [2] ⇨ [1]
 2: R := [NULL, NULL, ..., NULL] (size n)
 3: S := [NULL, NULL, ..., NULL] (size n)
 4: T := [NULL, NULL, ..., NULL] (size n)
 5: U := [NULL, NULL, ..., NULL] (size n)
 6: f := [1, 2, ..., n] (size n)
 7: d := [1, 1, ..., 1] (size n − 1)
 8: a := [0, 0, ..., 0] (size n − 1)
 9: while f[1] < n
10:    j := f[1]
11:    f[1] := 1
12:    a[j] := a[j] + d[j]
13:    if R[1] = NULL
14:       R[1] := head
15:       S[1] := other(head, NULL)
16:    end if
17:    Rnode := R[1]
18:    Snode := S[1]
19:    R[1] := NULL
20:    S[1] := NULL
21:    if T[j] = NULL
22:       T[j] := head
23:       U[j] := other(head, NULL)
24:    end if
25:    T[j] := Unode
26:    U[j] := other(Unode, Tnode)
27:    if a[j] = 0 OR a[j] = j
28:       d[j] := −d[j]
29:       f[j] := f[j + 1]
30:       f[j + 1] := j + 1
31:       if R[j + 1] = NULL
32:          temp := R[j]
33:          R[j] := S[j]
34:          S[j] := other(S[j], temp)
35:       else
36:          R[j] := R[j + 1]
37:          S[j] := S[j + 1]
38:       end if
39:       R[j + 1] := NULL
40:       S[j + 1] := NULL
41:       T[j] := NULL
42:       U[j] := NULL
43:    end if
44:    transpose(Snode, Tnode)
45:    if Tnode = head
46:       head := Snode
47:    end if
48:    if f[j] = j + 1
49:       R[j] := Tnode
50:    end if
51:    if Rnode ≠ Tnode
52:       reverse(Tnode, Rnode, head, NULL)
53:       head := Rnode
54:    else if Snode ≠ head
55:       reverse(Tnode, Snode, head, NULL)
56:       head := Snode
57:    end if
58:    visit(head)
59: end while
```

To understand R(n), observe that r_i equals the first focus pointer in TR(n) (see lines 8-10). Therefore, we mimic the focus pointers in f using real pointers

in a new array R. More precisely, if $f[i] = j$ then $R[i]$ will point to the jth node in the linked list. One caveat is that NULL is used when a focus pointer points to itself. That is, $R[i] =$ NULL when $f[i] = i$. This convention limits the changes made to R when reversing prefixes in the final algorithms. Given this description, we can equate instructions involving f with instructions involving R in R(n). In particular, line 9 matches with lines 12-14, line 10 matches with line 16, line 20 matches with lines 22-26, and line 21 matches with line 27. To track the $(r_i + 1)$st node instead of the r_ith node, line 13 can simply be changed to $R[1] :=$ other(head, NULL). In Lex(n) and Zaks(n), the r_ith node is tracked in array R, the $(r_i + 1)$st node is tracked in array S, and the resulting twin-pointers give a BLL-friendly alternative to line 23.

To understand T(n) consider its recursive formula originally found in (2)

$$\mathcal{T}_n = \mathcal{T}_{n-1},\ 1,\ \mathcal{T}_{n-1},\ 2,\ \ldots,\ \mathcal{T}_{n-1},\ n{-}2,\ \mathcal{T}_{n-1},\ n{-}1,\ \mathcal{T}_{n-1}.$$

The values in $1, 2, \ldots, n-1, 1$ are obtained by "marching" the pointers forward in line 16, before "retreating" in line 22. To track the $(t_i + 1)$st node instead of the t_ith node, line 13 can simply be changed to $T[j] :=$ other(head, NULL). In Lex(n), the t_ith node is tracked in array T, the $(t_i + 1)$st node is tracked in array U, and the resulting twin-pointers give a BLL-friendly alternative to line 16.

The second hurdle is to update these pointers during each transposition and reversal. The updates are simplified by the fact that $f[i] = i$ for $1 \le i < r[i]$ during the ith iteration of TR(n). Therefore, the corresponding NULL values in R, S, T, and U do not require updating. The additional code in the final algorithms account for the remaining updates to R, S, T, U, and head. (A tail pointer can also be easily updated.)

When investigating the final algorithms on page 9, notice the absence of ".value". We call the algorithms *non-probing* since they never query values stored in any node. One application is that the algorithms will function regardless of the initial values pointed to by head. In particular, co-lex and reverse co-lex are created by initializing head :=  and head :=  respectively. Furthermore, lexicographic and reverse lexicographic are created by maintaining tail and replacing visit(head) with visit(tail).

Both algorithms are implemented in C, and are available by request.

6 Open Problems

This paper uses a BLL to unsort permutations in worst-case O(1)-time in lexicographic order. Can this result be extended to multiset permutations? Where else does a BLL provide advantages over a DLL? Do non-probing algorithms have applications to privacy or security? What other combinatorial objects can be generated without probing? TR(n) gives a common framework for understanding lexicographic order and Zaks' order. Which other orders of permutations can be explained by the specialization of Algorithm H [10] to staircase strings or *downward staircase strings* (using bases $n, n-1, \ldots, 2$)?

References

1. V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM J. Comput.*, 25(2):272–289, 1996.
2. B. Chitturi, W. Fahle, Z. Meng, L. Morales, C.O. Shields, I.H. Sudborough, and W. Voit. An $(18/11)n$ upper bound for sorting by prefix reversals. *Theoretical Computer Science*, 410:3372–3390, 2009.
3. H. Choset. Coverage of known spaces: The boustrophedon cellular decomposition. *Journal Autonomous Robots*, 9(3):247–253, 2000.
4. D.S. Cohen and M. Blum. On the problem of sorting burnt pancakes. *Discrete Applied Mathematics*, 61:105–120, 1995.
5. G. Ehrlich. Loopless algorithms for generating permutations, combinations and other combinatorial configurations. *Journal of the ACM*, 20(3):500–513, 1973.
6. G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of Genome Rearrangements*. MIT Press, 2009.
7. W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27:47–57, 1979.
8. A.J. Goldstein and R.L. Graham. Sequential generation by transposition of all the arrangements of n symbols. *Bell Telephone Laboratories (Internal Memorandum), Murray Hill, NJ*, 1964.
9. H. Kaplan, R. Shamir, and R.E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms*, pages 178–187, New Orleans, Louisiana, United States, 1997.
10. D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 2: Generating All Tuples and Permutations. Addison-Wesley, 2005.
11. D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 3: Generating All Combinations and Partitions. Addison-Wesley, 2005.
12. D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 4: Generating All Trees, History of Combinatorial Generation. Addison-Wesley, 2006.
13. J. F. Korsh and S. Lipschutz. Generating multiset permutations in constant time. *Journal of Algorithms*, 25:321–335, 1997.
14. D.H. Lehmer. Teaching combinatorial tricks to a computer. In *Combinatorial Analysis*, volume 10 of *Proc. of Symposlum Appl. Math*, pages 179–193, Providence, R.I,, United States, 1960. American Mathematical Society.
15. M. Mares and M. Straka. Linear-time ranking of permutations. In *Algorithms - ESA 2007*, volume 4698 of *Lecture Notes in Computer Science*, pages 187–193, Providence, R.I,, United States, 2007. Springer Berlin / Heidelberg.
16. J. Millar, N.J.A. Sloane, and N.E. Young. A new operation on sequences: the boustrouphedon transform. *Journal of Comb. Theory, Series A*, 76(1):44–54, 1996.
17. N. Sloane. `http://www.research.att.com/~njas/sequences/A055881`, 2010.
18. H. Steinhaus. *One Hundred Problems in Elementary Mathematics*. Pergamon Press, 1963. Reprinted by Dover Publications in 1979.
19. Wikipedia. `http://en.wikipedia.org/wiki/Boustrophedon`, 2009.
20. A. Williams. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *SODA '09: The Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, New York, New York, USA, 2009.
21. YouTube. 2009 National Sweet Corn Eating Championship. `http://www.youtube.com/watch?v=EUy56pBA-HY`, 2009.
22. S. Zaks. A new algorithm for generation of permutations. *BIT Numerical Mathematics*, 24(2):196–204, 1984.