

# Combinatorial Generation by Fusing Loopless Algorithms

Tadao Takaoka

Stephen Violich

Department of Computer Science and Software Engineering  
University of Canterbury  
Christchurch, New Zealand  
{tad, ssv10}@cosc.canterbury.ac.nz

## Abstract

Some combinatorial generation problems can be broken into subproblems for which loopless algorithms already exist. We discuss means by which loopless algorithms can be fused to produce a new loopless algorithm that solves the original problem. We demonstrate this method with two new loopless algorithms, MIXPAR and MULTPERM. MIXPAR generates well-formed parenthesis strings containing two different types of parentheses. MULTPERM generates multiset permutations in linear space using only arrays; it is simpler and more efficient than the recent algorithm of Korsh and LaFollette (2004).

## 1 Introduction

The generation of combinatorial objects, such as combinations, permutations and parenthesis strings, is a well studied area, covered by Nijenhuis and Wilf (1975), Reingold, Nievergelt and Deo (1977), Wilf (1989) and Savage (1997).

Loopless algorithms for combinatorial generation were introduced by Ehrlich (1973). These algorithms generate each combinatorial object from its predecessor using no more than a constant number of instructions, thus they are ‘loop-free’. It follows that it should be possible to combine loopless algorithms in such a way that the resulting algorithm still satisfies this property. If a combinatorial generation problem can be broken down into subproblems for which loopless algorithms already exist, then combining those algorithms might lead to a loopless algorithm for the original problem.

This idea is not new, for example Korsh and Lipschutz (1997) and Korsh and LaFollette (2004) give loopless algorithms for multiset permutations that combine existing loopless algorithms for element selection and combination movement. We believe, however, that combining loopless algorithms has not been discussed in general before. We refer to the combining of algorithms as *fusing* because this does not limit us to any particular structures or patterns.

We introduce general program structures for fused loopless algorithms and discuss implementation issues in Section 2. We then cover Williamson’s (1985) algorithm for variations in Gray code order in Section 3, as it is the basis for many of the subsequent algorithms we discuss. We use fusing to produce MIXPAR, an algorithm for generating mixed parenthesis strings, which comprise parentheses of different types,

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 51. Barry Jay and Joachim Gudmundsson, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

in Section 4. A second new algorithm, MULTPERM, is presented in Section 5, and experimentally evaluated against the algorithm recently published by Korsh and LaFollette. Finally, we draw some conclusions in Section 6.

## 2 Fusing Loopless Algorithms

A generalised a loopless algorithm is shown in Figure 1(a). Function *init* initialises the algorithm and generates the first object, *next* generates each successive object, while *last* returns whether this current object is the final one in the sequence. Functions *next* and *last* run in  $O(1)$  time, while *init* is allowed  $O(n)$  time. ‘Loopless’ may seem a misnomer, since a control loop is required, but it is the generation of *each* object that is loop-free.

Two loopless algorithms can be nested so that a complete cycle of the inner algorithm runs during each iteration of the outer algorithm, as shown in Figure 1(b). Functions *next\_1* and *isnext\_1* belong to the inner algorithm, while *next\_2* and *isnext\_2* belong to the outer. Because the initial and final states of a loopless algorithm differ, a new function, *reinit\_1*, is required to reinitialise the inner algorithm before it begins a new cycle. There are two ways an algorithm can be reinitialised: *refreshing* means to reset an al-

```
1. init
2. while not last do
3.     next
```

(a) Single loopless algorithm

```
1. init
2. while not last_2 do
3.     while not last_1 do
4.         next_1
5.         reinit_1
6.         next_2
```

(b) Two loopless algorithms, nested

```
1. init
2. while not last_2 do
3.     if not last_1 then
4.         next_1
5.     else
6.         reinit_1
7.         next_2
```

(c) Two loopless algorithms, un-nested

Figure 1: Program structures for loopless algorithms.

**Algorithm 1** Williamson’s (1985) loopless algorithm for variations in Gray code order.

```

/* Initialise */
1. procedure init_Wil
2.   read  $n$ 
3.   for  $i = 1$  to  $n$  do read  $r[i]$ 
4.   for  $i = 1$  to  $n$  do  $v[i] = 1$ 
5.   for  $i = 1$  to  $n$  do  $d[i] = 1$ 
6.   for  $i = 0$  to  $n$  do  $e[i] = i$ 
7.    $j = n$ ;

/* Generate */
8. procedure next_Wil
9.    $e[n] = n$ 
10.  add  $d[j]$  to  $v[j]$ 
11.  if  $v[j]$  is either 1 or  $r[j]$  then
12.     $e[j] = e[j - 1]$ 
13.     $e[j - 1] = j - 1$ 
14.     $d[j] = -d[j]$ 
15.     $j = e[n]$ 

/* Main */
16. init_Wil
17. print  $v$ 
18. while  $j$  is not 0 do
19.   next_Wil
20. print  $v$ 

```

gorithm to its initial state; *reversing* means to alter the algorithm so it will run from its final state back to its initial state over a cycle. Since reinitialisation occurs between objects, *reinit* is only allowed  $O(1)$  time. Although these nested loopless algorithms contain an extra while loop, successive objects are still generated in no more than a constant number of instructions.

For greater clarity, the nested structure can be modified into an *un-nested* structure by replacing the second while loop with an if-then-else statement, as shown in Figure 1(c). This un-nested configuration executes the functions in the same order as the nested configuration, but now a single loop-free algorithm that generates exactly one object per iteration can be isolated within the program. The new algorithms that we develop in Sections 4 and 5 adhere to this un-nested structure.

Although *reinit\_1* is limited to  $O(1)$  time, there are a couple of tricks for fitting  $O(n)$ -time reinitialisation into this framework. For example, the final state of an algorithm might include some array  $a_{1..n}$  that has  $O(n)$  points of difference from its initial state. Supposing the algorithm is irreversible, then it requires  $O(n)$  time to reinitialise. One option, available if the algorithm finishes with different  $a_i$  at different stages during its cycle, is to reinitialise each  $a_i$  as soon as it becomes obsolete, during iterations of *next\_1*. In this way,  $O(n)$  reinitialising steps can be executed in  $O(1)$  time per object, a technique we call *time-stealing*. In the best case, this algorithm would give cues as to exactly when each  $a_i$  becomes obsolete; in the worst, a for-loop would be simulated, using a counter variable and an arbitrary start cue. We use this time-stealing technique to iteratively re-initialise array  $s$  in algorithm MULTPERM in Section 5. A second option is less elegant and much less efficient, although it seems universally applicable: maintain two separate versions of the troublesome arrays or variables. Then, in any given cycle of the inner algorithm, one version can be used while the other is reinitialised as per time-stealing.

	$v_{1..3}$	$e_{0..3}$	$j$		$v_{1..3}$	$e_{0..3}$	$j$
1.	1 1 1	0 1 2 3	3	10.	<u>2</u> 3 3	<u>0</u> 0 2 3	3
2.	1 1 2	0 1 2 3	3	11.	2 3 2	0 0 2 3	3
3.	1 1 <u>3</u>	0 1 <u>2</u> 2	2	12.	2 3 <u>1</u>	0 0 <u>2</u> 2	2
4.	1 2 3	0 1 2 3	3	13.	2 2 1	0 0 2 3	3
5.	1 2 2	0 1 2 3	3	14.	2 2 2	0 0 2 3	3
6.	1 2 <u>1</u>	0 1 <u>2</u> 2	2	15.	2 2 <u>3</u>	0 0 <u>2</u> 2	2
7.	1 <u>3</u> 1	0 <u>1</u> 1 3	3	16.	2 <u>1</u> 3	0 <u>1</u> 0 3	3
8.	1 3 2	0 1 1 3	3	17.	2 1 2	0 1 0 3	3
9.	1 3 <u>2</u>	0 1 <u>2</u> 1	1	18.	2 1 <u>1</u>	0 1 <u>2</u> 0	0

Figure 2: Output for Williamson’s algorithm for inputs  $n = 3$ ,  $r = \{2, 3, 3\}$ . Each  $v[i]$  varies between 1 and  $r[i]$  inclusive. Underlines indicate when  $v[j]$  becomes extremal, and the corresponding conveying from  $e[j - 1]$  to  $e[j]$  and resetting of  $e[j - 1]$ . Note that  $j$  changes at the end of each iteration, so the value of  $j$  used to generate any  $v$  and  $e$  is on the preceding line.

### 3 Williamson’s Algorithm

We include a discussion of Williamson’s (1985, p.112) loopless algorithm for generating variations in Gray code order because its recursion-simulation technique is used by three out of the four subsequent algorithms in this paper. The algorithm generates elements of the product space  $S = S_1 \times S_2 \times \dots \times S_n$ , with  $S_i = 0, 1, \dots, r_i - 1$  for  $i = 1, 2, \dots, n$ . Williamson’s algorithm is shown in Algorithm 1.

The variables in Williamson’s algorithm are:  $v_{1..n}$ , the current variation;  $j$ , the current position in  $v$  to change;  $d_{1..n}$ , the current increment (1 or -1) for each position in  $v$ ; and  $e_{0..n}$ , which determines the order in which positions in  $v$  should be selected as values for  $j$ . Values for  $n$  and all  $r[i]$  are read from the user. The remaining variables are initialised as follows: all  $v_i$  are set to 0; all  $d_i$  are set to 1; all  $e_i$  are set to  $i$ ; and  $j$  is set to  $n$ . Array  $e$  is used to looplessly simulate a recursive tree traversal. Though this technique is well known and comprises only a few lines of code, it is nontrivial and rarely explained.

When  $e_i$  is set to  $i$ , we say that  $e_i$  is *reset*, since  $i$  was the initialised value of  $e_i$ . When  $v_j$  becomes extremal, the value at  $e_{j-1}$  is passed along one place to  $e_j$ , then  $e_{j-1}$  is reset. Referring to the coding tree in Figure 2, this can be seen when  $v = \{1, 3, 3\}$ , for example. Because  $v_3$  has become a last child,  $e_3$  inherits the value 1 from  $e_2$ , while  $e_2$  is reset to 2.

A similar pass-reset pattern occurs between  $e_n$  and variable  $j$ . At the end of every iteration of *next* the value at  $e_n$  is passed along to variable  $j$ ; at the start of the next iteration,  $e_n$  is reset. Referring again to Figure 2, the resetting of  $e_3$  is visible when  $v = \{1, 1, 3\}$ ,  $\{1, 2, 1\}$ , and so on. It *happens* on every line, of course, but can only be seen when  $e_3$  was not already 3 and was not subsequently changed.

In effect,  $e$  can be thought of as a conveyor belt that passes information along towards variable  $j$ . It is helpful to picture variable  $j$  as positioned immediately after  $e_n$ , since information flows along array  $e$  and into  $j$ . Whenever information is passed along, the source of that information is reset.

Any value  $i$  can only enter the array by resetting  $e_i$ . When  $e_i$  inherits a value from  $e_{i-1}$ , that value instead of  $i$  will be carried towards variable  $j$ . That means that  $v_i$  will be skipped over on the next occasion that would have otherwise been its turn to be changed.

When  $v_i$  is skipped, and one of its ancestors is changed,  $v_i$  becomes a first child, so it should not be skipped again. Thus, as soon the value of  $e_i$  is passed on,  $e_i$  is reset. This means the *subsequent* value to be passed from  $e_i$  will be  $i$  again, making  $v_i$  available

<i>par</i>	<i>mix</i>	<i>mixpar</i>
( ( ) ( ) )	⋮ ( ( ( ( ( ) ( ) ) ) ) ) ( ( [ ( ( ) [ ] ) ) ) ( [ [ ( [ ] [ ] ) ) ) ( [ ( [ ] ( ) ) ) [ [ ( [ ] ( ) ) ] [ [ [ [ ] [ ] ] ] [ ( [ [ ( ) [ ] ] ) ] [ ( ( [ [ ( ) ( ) ] ) ] ) ]	⋮ ( ( ( ) ( ) ) ) ⋮

(a) Par-outside-mix (par-mix)

<i>mix</i>	<i>par</i>	<i>mixpar</i>
( [ [ ( ( ) ( ) ) ] ] )	⋮ ( ( ) ( ) ( ) ) ( ( ) ( ) ( ) ) ( ( ) ( ) ( ) ) ( ( ( ) ) ) ) ( ( ) ) ( ) )	⋮ ( ) [ ] [ ] ( ) [ [ ] ] ( [ [ ] ] ) ( [ [ ] ] ) ( [ ] ) [ ]

(b) Mix-outside-par (mix-par)

Figure 3: Sample outputs for mixpar algorithms with opposite nesting configurations.

for change. Note that if the value of  $e_i$  was already  $i$  before it was passed along then resetting  $e_i$  has no effect.

Both of our new algorithms, MIXPAR and MULTPERM, in Sections 4 and 5 respectively, use the Williamson’s variables  $j$ ,  $d$  and  $e$  to select elements for change. MIXPAR uses a second set, labelled  $jj$ ,  $dd$  and  $ee$ , since both of its component algorithms follow the Williamson model.

#### 4 Mixed Parenthesis Strings

The first combinatorial generation problem we apply our fusing framework to is in the area of parenthesis strings. A well-formed parenthesis string, or *par* for short, can be derived from the grammar  $P \rightarrow \epsilon \mid (P) \mid PP$ . A par has  $n$  pairs, and so its size is  $2n$ .

We introduce a new combinatorial object: *mixed parenthesis strings*, or *mixpars* for short, which comprise parentheses of different types. In this paper we limit the number of types to two, but it is trivial to extend the ideas beyond binary. The grammar for a mixpar is a modification of that for a par, in this case  $M \rightarrow \epsilon \mid (M) \mid [M] \mid MM$ . Thus, a mixpar is well-formed if its parentheses are arranged as per an ordinary par, *and* if both parentheses in each pair share the same type. For example,  $( ) [ ]$  and  $( [ ] )$  are a valid mixpars, while  $( ] [ )$  and  $( [ ) ]$  are not.

A mixpar can be thought of as a par with a certain *mix* of types. For example, the mixpar  $( ) [ ]$  can be described as the par  $( ) ( )$  with the mix  $[$ . Note that with only two types, a mix corresponds to a binary string. It follows that generating all mixpars for some  $n$  is a matter of generating either all mixes for each par or all pars for each mix. Thus, an algorithm for generating mixpars nests algorithms for generating pars and mixes in some way. Because loopless algorithms for pars and binary strings exist, we hypothesized that a loopless algorithm for generating mixpars could be fused from these. This fusion is carried out within the framework discussed in Section 2.

<i>par</i>	<i>mix</i>	<i>mixpar</i>
( ( ) ( ) )	⋮ ( ( ( ( ( ) ( ) ) ) ) ) ⋮ [ ( ( [ ( ) ( ) ] ) ) ] ( ( ( ) ) ) ) ⋮ [ ( ( [ ( ) ( ) ] ) ) ] ( ( ) ) ( ) ) ( ( ( ( ( ) ( ) ) ) ) ) ⋮	⋮ ( ( ( ) ( ) ) ) ⋮

(a) Refreshing.

<i>par</i>	<i>mix</i>	<i>mixpar</i>
( ( ) ( ) )	⋮ ( ( ( ( ( ) ( ) ) ) ) ) ⋮ [ ( ( [ ( ) ( ) ] ) ) ] ( ( ( ) ) ) ) ⋮ [ ( ( [ ( ) ( ) ] ) ) ] ( ( ) ) ( ) ) ( ( ( ( ( ) ( ) ) ) ) ) ⋮	⋮ ( ) [ ] [ ] ( ) [ [ ] ] ( [ [ ] ] ) ( [ [ ] ] ) ( [ ] ) [ ]

(b) Reversing.

Figure 4: Sample outputs for mixpar algorithms refreshing and reversing the inner mix algorithm respectively.

The way in which the two algorithms are nested affects the modifications required to make each algorithm operate directly on mixpars. Figure 3 shows output for mixpar algorithms with the two possible nesting configurations, par-outside-mix and mix-outside-par. (The par algorithm used is that of Xiang and Ushijima (2001), which is the one we ultimately chose and will discuss later; the mix algorithm is simply a Gray code generator.) From Figure 3(a) it can be seen that each iteration of the mix algorithm must change the type of one pair in the mixpar. Figure 3(b) shows that each iteration of the par algorithm must change the places or types of two to four parentheses. The mix-par configuration seemed to require more difficult modification to its inner algorithm, so we opted for the par-mix arrangement.

The method of reinitialising the inner algorithm also has an impact on the difficulty of fusing the algorithms. Recalling Section 2, inner algorithms can be reinitialised by either refreshing or reversing. Figure 4 shows output for mixpar algorithms that refresh and reverse their inner algorithms respectively. From Figure 4(a) it can be seen that refreshing the mix algorithm means that all parentheses are round whenever it is the par algorithm’s turn to operate. (This takes advantage of the fact that the last object in a Gray code has only one point of difference to the first object.) Figure 4(b) shows that reversing the mix algorithm means the par algorithm will frequently have to cope with one pair of an alternate type. Again, we opted for the simpler option, that of refreshing rather than reversing the mix algorithm.

In order to change the types of pairs, the positions of the parentheses in each pair must be known. Let  $l_i$  be the position of the  $i$ th left parenthesis, and let  $r_i$  be the position of *the partner of the  $i$ th left parenthesis* (that is, *not* simply the  $i$ th right parenthesis as counted from the start). For example, for the mixpar  $( ( ( ) ) ) )$ ,  $l_2 = 2$  and  $r_2 = 5$ .

Although we do not know of a loopless par algorithm that correctly maintains all  $l_i$  and  $r_i$ , Xiang and Ushijima’s (2001) algorithm does correctly maintain all  $l_i$ . We now present a method for finding all  $r_i$  in

**Algorithm 2** Xiang and Ushijima’s (2001) loop-less algorithm for parenthesis strings.

```

/* Initialise */
1. procedure init_XU
2.   read n;
3.   for i = 1 to 2n by 2 do
4.     set par[i] to ‘(’, par[i + 1] to ‘)’
5.   for i = 1 to n do l[i] = 2i - 1
6.   for i = 1 to n do d[i] = 1
7.   for i = 0 to n do e[i] = i
8.   j = n

/* Generate */
9. procedure next_XU
10.  e[n] = n
11.  i = l[j]
12.  if d[j] is 1 then
13.    if l[j] is 2j - 1 then
14.      l[j] = l[j - 1] + 1
15.    else
16.      add 1 to l[j]
17.    else
18.      if l[j] is l[j - 1] + 1 then
19.        l[j] = 2j - 1
20.      else
21.        subtract 1 from l[j]
22.        swap par[i] and par[l[j]]
23.        if l[j] ≥ 2j - 2 then
24.          d[j] = -d[j]
25.          e[j] = e[j - 1]
26.          e[j - 1] = j - 1
27.          j = e[n]

/* Main */
28. init_XU
29. print par
30. while j is not 1 do
31.   next_XU
32.   print par

```

constant time per object. The entire mixpar cannot be scanned after every iteration of the par algorithm, as that would require  $O(n)$  time, so the solution is to use the time-stealing technique mentioned in Sect. 2, finding each  $r_i$  in  $O(1)$  time during iterations of the mix algorithm.

We say a parenthesis pair is *empty* if no pairs are nested inside it. Recalling the grammar for a par, the  $n$ th pair must be empty, since no subsequent pairs exist. Thus:

$$r_n = l_n + 1 \quad (1)$$

It follows that the  $(n - 1)$ th pair must be empty or nested around the  $n$ th pair. Our algorithm is based on the idea that, if we start from the  $n$ th pair and work backwards to the first, each pair must be either empty or nested around some substring comprising pairs we have already encountered. Thus, information about substrings must be stored. Let  $s_{l_i}$  be the position after the longest well-formed substring beginning at  $l_i$ . For example, for the mixpar  $(( ( ) ) )$ ,  $l_2 = 2$  and  $s_{l_2} = s_2 = 6$ . Because we cannot know all  $s_i$  immediately, our algorithm initialises array  $s_{1..2n}$  such that all  $s_i = i$ . Equation (1) is the base step of our induction. We now show how each successive  $s_{l_i}$  and  $r_i$  can be found in constant time by working backwards from  $i = n$ .

If there is no  $j$ th left parenthesis immediately after  $r_i$ , then the substring beginning at  $l_i$  ends at  $r_i$ , and  $s_{r_i+1}$  will not have changed since initialisation.

	<i>par</i>	<i>l</i>
1.	( ) ( ) ( ) ( )	1 3 5 7
2.	( ) ( ) ( ) ( )	1 3 5 6
3.	( ) ( ( ) ) ( )	1 3 4 6
4.	( ) ( ( ) ) ( )	1 3 4 5
5.	( ) ( ( ) ) ( )	1 3 4 7
6.	( ( ) ( ) ) ( )	1 2 4 7
7.	( ( ) ( ) ) ( )	1 2 4 5
8.	( ( ) ( ) ) ( )	1 2 4 6
9.	( ( ( ) ) ) ( )	1 2 3 6
10.	( ( ( ) ) ) ( )	1 2 3 5
11.	( ( ( ) ) ) ( )	1 2 3 4
12.	( ( ( ) ) ) ( )	1 2 3 7
13.	( ( ) ) ( ) ( )	1 2 5 7
14.	( ( ) ) ( ( ) )	1 2 5 6

Figure 5: Xiang and Ushijima’s algorithm output for  $n = 4$ .

On the other hand, if  $r_i$  is adjacent to some  $l_j$ , then the substrings beginning at  $l_i$  and  $l_j$  end in the same position, and because we are working backwards from the  $n$ th pair,  $s_{l_j}$  will already have been set correctly. Thus, we derive an unconditional equation for  $s_{l_i}$  independent of  $j$ :

$$s_{l_i} = \begin{cases} r_i + 1 & = s_{r_i+1} \text{ iff } r_i + 1 \neq l_j \\ s_{l_j} & = s_{r_i+1} \text{ iff } r_i + 1 = l_j \\ & = s_{r_i+1} \end{cases} \quad (2)$$

Similarly for  $r_i$ , if the  $(i + 1)$ th left parenthesis is not immediately after  $l_i$ , then  $r_i$  must be, and  $s_{l_{i+1}}$  will not have changed since initialisation. Conversely, if the  $i$ th and  $(i + 1)$ th left parentheses are adjacent, then  $r_i$  must be immediately after the substring starting at  $l_{i+1}$ . Because we are working backwards from the  $n$ th pair,  $s_{l_{i+1}}$  will already have been set correctly. Thus, we derive an unconditional equation for  $r_i$ :

$$r_i = \begin{cases} l_i + 1 & = s_{l_{i+1}} \text{ iff } l_i + 1 \neq l_{i+1} \\ s_{l_{i+1}} & = s_{l_{i+1}} \text{ iff } l_i + 1 = l_{i+1} \\ & = s_{l_{i+1}} \end{cases} \quad (3)$$

Thus, using (1), (2) and (3), right parentheses from  $n$ th to first can be found in  $O(1)$  time each, during iterations of the first half of the Gray cycle. As we finish with each  $r_i$  during the second half of the Gray cycle, we reset each  $s_{l_i}$ .

We now cover Xiang and Ushijima’s par algorithm. In addition to correctly maintaining all  $l_i$ , it a very efficient loopless par algorithm in terms of time and space. It is also very simple, which helped keep our final MIXPAR algorithm simple. Xiang and Ushijima’s algorithm is shown in Algorithm 2 (note that we have renamed their array for the positions of the left parentheses from  $p$  to  $l$  for consistency with our right-finding approach). Its element-selection mechanism is familiar from Williamson’s algorithm in Section 3, although its element-change code is a little more complex.

Xiang and Ushijima’s algorithm introduces several new variables. As mentioned, the number of parenthesis pairs is  $n$ , which is read from the user. The par is stored in  $par_{1..2n}$ , while the left parentheses positions are stored in  $l_{1..n}$ . These are initialised to  $( ) ( ) \dots ( )$  and  $1, 3, \dots, 2n - 1$  respectively. Finally,  $i$  and  $c$  are temporary variables used to facilitate an array swap, storing an integer and character respectively. Variables  $j$ ,  $d_{1..n}$  and  $e_{0..n}$  are inherited from Williamson’s algorithm, and relate to the left parentheses; initialisations remain the same.

It works in the same way as their combinations algorithm from the same paper; both are variations

**Algorithm 3** MIXPAR, a new, loopless algorithm for mixed parenthesis strings

```

/* Initialise */
1. procedure init_Mix
2.   init_XU /* From Alg. 2 */
3.   for  $i = 1$  to  $n$  do  $dd[i] = 1$ 
4.   for  $i = 0$  to  $n$  do  $ee[i] = i$ 
5.   for  $i = 1$  to  $2n$  do  $s[i] = i$ 
6.    $jj = n$ 
7.    $t = n$ 

/* Find right parenthesis */
8. procedure find
9.   if  $dd[1]$  is 1 then
10.     $r[jj] = s[l[jj] + 1]$ 
11.    if  $jj$  is not 1 then
12.      $s[l[jj]] = s[r[jj] + 1]$ 
13.     subtract 1 from  $t$ 
14.   else
15.     $s[l[jj]] = l[jj]$ 
16.    add 1 to  $t$ 

/* Generate by Gray */
17. procedure next_Gray
18.    $ee[n] = n$ 
19.   if  $jj$  is  $t$  then find
20.   change  $par[l[jj]]$  and  $par[r[jj]]$  from
   round to square or vice versa
21.    $ee[jj] = ee[jj - 1]$ 
22.    $ee[jj - 1] = jj - 1$ 
23.    $dd[jj] = -dd[jj]$ 
24.    $jj = ee[n]$ 

/* Re-initialise Gray */
25. procedure reinit_Gray
26.   change  $par[l[1]]$  and  $par[r[1]]$  to round
27.    $dd[1] = 1$ 
28.    $jj = n$ 
29.    $t = n$ 

/* Main */
30. init_Mix
31. print  $par$ 
32. while  $j$  is not 1 or  $jj$  is not 0 do
33.   if  $jj$  is not 0 then
34.    next_Gray
35.   else
36.    reinit_Gray
37.   next_XU /* From Alg. 2 */
38.   print  $par$ 

```

on Williamson's algorithm in which no two elements in the same object can have the same value. Xiang and Ushijima noted that parentheses maintain a relative order, that is  $l_1 < l_2 < \dots < l_n$ , and that well-formedness dictates how far to the right each left parenthesis can travel, that is  $l_i \leq 2i - 1$  for  $1 \leq i \leq n$ . At any time, these principles determine the upper and lower bounds for left parenthesis travel.

Xiang and Ushijima extended Williamson's algorithm to have four patterns of change:  $O^+$ ,  $O^{+'}$ ,  $O^-$  and  $O^{-'}$ . The regular positive direction,  $O^+$ , causes a parenthesis to move steadily right between its current bounds. The prime positive direction,  $O^{+'}$ , causes a parenthesis to jump from its lower bound to its upper bound, then move steadily left through all remaining values. The negative directions have the opposite effects. These jumps in the prime directions allow the algorithm to avoid clashes (different elements sharing

```

1. ( ) ( ) ( )      25. ( ( ( ) ) )
2. ( ) ( ) [ ]      26. ( ( [ ] ) )
3. ( ) [ ] [ ]      27. ( [ [ ] ] )
4. ( ) [ ] ( )      28. ( [ ( ) ] )
5. [ ] [ ] ( )      29. [ [ ( ) ] ]
6. [ ] [ ] [ ]      30. [ [ [ ] ] ]
7. [ ] ( ) [ ]      31. [ ( [ ] ) ]
8. [ ] ( ) ( )      32. [ ( ( ) ) ]

9. ( ) ( ( ) )      33. ( ( ) ) ( )
10. ( ) ( [ ] )     34. ( ( ) ) [ ]
11. ( ) [ [ ] ]     35. ( [ ] ) [ ]
12. ( ) [ ( ) ]     36. ( [ ] ) ( )
13. [ ] [ ( ) ]     37. [ [ ] ] ( )
14. [ ] [ [ ] ]     38. [ [ ] ] [ ]
15. [ ] ( [ ] )     39. [ ( ) ] [ ]
16. [ ] ( ( ) )     40. [ ( ) ] ( )

17. ( ( ) ( ) )
18. ( ( ) [ ] )
19. ( [ ] [ ] )
20. ( [ ] ( ) )
21. [ [ ] ( ) ]
22. [ [ ] [ ] ]
23. [ ( ) [ ] ]
24. [ ( ) ( ) ]

```

Figure 6: MIXPAR algorithm output for  $n = 3$ . Line-breaks have been inserted to highlight when the par is changed by the outer algorithm.

the same value) while generating all combinations of left parenthesis positions.

Output for Xiang and Ushijima's algorithm for  $n = 4$  is shown in Figure 5. All  $l_i$  begin maximally, and increment or decrement in a pattern similar, at first glance, to that of Williamson's algorithm. Closer examination of lines 2–5, however, reveals the effect of a prime direction jump. On line 2,  $l_4$  is minimal, so in Williamson's algorithm you would expect it to reverse direction next time it moved. But on line 3, the change to  $l_3$  means that  $l_4$  is no longer minimal. On line 4, a prime jump is employed so that  $l_4$  can take the newly available minimum value before ascending as per usual to the maximum on line 5.

Algorithm MIXPAR, our new mixed parenthesis strings algorithm, is given in Algorithm 3. A complete C++ program is given in Appendix A. Its main statements (lines 30–38) reveal that it fits exactly into the un-nested structure outlined in Section 2. The initialisation and next methods belonging to Xiang and Ushijima's algorithm are able to be incorporated verbatim.

Most of the variables in MIXPAR are inherited from its constituent algorithms. From Xiang and Ushijima's algorithm come the variables  $n$ ,  $par_{1\dots 2n}$ ,  $l_{1\dots n}$ ,  $j$ ,  $d_{1\dots n}$ ,  $e_{0\dots n}$ ,  $i$  and  $c$ . From Williamson's algorithm, to run our mix (Gray code) algorithm, come the variables  $jj$ ,  $dd_{1\dots n}$  and  $ee_{0\dots n}$ . All initialisations are as previously described.

Three new variables are introduced. Finding right parentheses requires arrays  $r_{1\dots n}$  and  $s_{1\dots 2n}$ , of which  $r$  is not initialised and the initialisation of  $s$  has already been covered. Finally, to keep track of which right parenthesis is due to be found during the first half of the Gray cycle, and which value of  $s$  is due to be refreshed during the second half, we use variable  $t$ ; initially  $t = n$ .

A sample output of MIXPAR for  $n = 3$  is shown in Figure 6. The output is displayed in columns separated by newlines, where each column begins with a par generated by Xiang and Ushijima's algorithm. The remaining lines in each column show complete Gray code cycles of mixes for that column's par.

**Algorithm 4** Chase’s (1989) loopless algorithm for combinations by  $O(1)$ -distance transpositions.

```

/* Initialise */
1. procedure init_Chase
2.   read  $n$  and  $r$ 
3.   for  $i = 1$  to  $n$  do  $comb[i] = i$ 
4.    $comb[i] = 2r - 1$ 
5.    $z = n + 1$ 
6.   Set  $b$  to 1 if  $r$  is even, else 2

/* Next */
7. procedure next_Chase
8.   if  $z$  is 1 then
9.     if  $inc(1)$  then
10.      if  $adj(1)$  then
11.        if  $inc(2)$  then  $move(1, 1, 2)$ 
12.        else  $move(2, -1, 2)$ 
13.        else  $move(1, 1, 1)$ 
14.        else  $move(1, -1, 1)$ 
15.      else
16.        if  $inc(z - 1)$  then
17.          if  $z > 2$  and  $inc(z - 2)$  then
18.             $move(z - 2, 1, 2)$ 
19.          else  $move(z - 1, 1, 1)$ 
20.        else
21.          if not  $adj(z)$  then
22.            if  $inc(z)$  then  $move(z, 1, 1)$ 
23.            else  $move(z, -1, 1)$ 
24.          else
25.            if  $inc(z + 1)$  then
26.               $move(z, 1, 2)$ 
27.            else  $move(z + 1, -1, 2)$ 

/* Move comb elements */
28. procedure move( $p, d, s$ )
29.    $x = comb[p]$ 
30.    $y = x + s \times d$ 
31.    $comb[p] = x + d$ 
32.    $comb[p + d(s - 1)] = y$ 
33.   if  $comb[z]$  is  $z$  then
34.     add  $s$  to  $z$ 
35.   if  $comb[z]$  is  $z$  then add  $s$  to  $z$ 
36.   else if  $comb[z - 1]$  is not  $z - 1$  then
37.     subtract  $s$  from  $z$ 

/* Returns comb[i] increasing? */
38. function inc( $i$ )
39.   return  $comb[i + 1]$  is odd

/* Returns comb[i] and [i+1] adjacent? */
40. function adj( $i$ )
41.   return  $comb[i] + 1$  is  $comb[i + 1]$ 

/* Main */
42. init_Chase
43. print  $comb$ 
44. while  $comb[n - b]$  is not minimal or
    $comb[n - b + 1]$  is not maximal do
45.   next_Chase
46.   print  $comb$ 

```

## 5 Multiset Permutations

The second combinatorial generation problem we apply our fusing framework to is that of multiset permutations. A multiset, or set with repetitions, has  $k$  distinct elements, which we assume without loss of generality to be the integers  $[1, k]$ . Each distinct el-

	<i>comb</i>	bit vector	$z$	case
1.	1 2 3 4	1 1 1 1 0 0	5	6
2.	1 2 3 5	1 1 1 0 1 0	4	5
3.	1 3 4 5	1 0 1 1 1 0	2	6
4.	2 3 4 5	0 1 1 1 1 0	1	2
5.	1 2 4 5	1 1 0 1 1 0	3	9
6.	1 2 5 6	1 1 0 0 1 1	3	6
7.	1 3 5 6	1 0 1 0 1 1	2	6
8.	2 3 5 6	0 1 1 0 1 1	1	1
9.	3 4 5 6	0 0 1 1 1 1	1	4
10.	2 4 5 6	0 1 0 1 1 1	1	4
11.	1 4 5 6	1 0 0 1 1 1	2	10
13.	2 3 4 6	0 1 1 1 0 1	1	2
14.	1 2 4 6	1 1 0 1 0 1	3	8
15.	1 2 3 6	1 1 1 0 0 1	4	

Figure 7: Chase’s algorithm output for  $n = 4, r = 6$ . The case column identifies which of the ten move calls is used to generate each object.

ement  $i$  has a multiplicity  $m_i$ , which is the number of times it appears in the multiset. The size  $n$  of the multiset is the sum of all multiplicities. For example, the multiset  $\{1, 1, 1, 2, 2, 3\}$  has  $k = 3, m = \{3, 2, 1\}$  and  $n = 6$ . Indistinguishable elements are called *similar*.

Our approach to generating multiset permutations is based on the Johnson (1963) and Trotter (1962) algorithms for set permutations, which work by iteratively moving single elements through subpermutations. We reasoned that a modified algorithm could iteratively move groups of similar elements through subpermutations, thereby generating multiset permutations, and that this grouped element movement could be achieved using a combinations algorithm. A similar approach was taken by Korsh and LaFollette (2004) to develop the first linear-space loopless multiset permutations algorithm using only arrays. We subsequently draw attention to several important design differences that led us to choose a more advantageous combinations algorithm than Korsh and LaFollette, and ultimately develop a simpler and more efficient algorithm. Other multiset permutations algorithms based on combining algorithms include Korsh and Lipschutz (1997) and Vajnowski (2003).

A recursive algorithm for multiset permutations is as follows. Let  $perm$  be a multiset permutation of  $n$  integers. Let  $subp_i$  be a subpermutation of  $perm$  comprising all elements *greater than*  $i$ . Initially  $perm$  is the lexicographically least permutation. If  $k = 1$  then  $perm$  is the only permutation. Otherwise, the 1s are placed among  $subp_1$  in all remaining distinct ways such that the relative order of elements of  $subp_1$  is maintained, and  $subp_1$  is contiguous in the final permutation. This generates all permutations containing  $subp_1$ . If there is another  $subp_1$  of  $perm$ , it is generated recursively, and the next  $perm$  becomes this next  $subp_1$  bounded by the 1s. The 1s are now placed among this next  $subp_1$  in all remaining distinct ways, subject to the same conditions as before. This generates all permutations containing this next  $subp_1$ . This process of moving 1s through  $subp_1$ s continues until they have appeared in all distinct ways in the last  $subp_1$ . When the  $k$  integers are distinct this algorithm mimics the Johnson-Trotter.

The recursive algorithm we describe is similar to that described by Korsh and LaFollette, with one important difference: when the similar elements finish moving through a subpermutation, Korsh and LaFollette require that they all be at one end (left or right) of the subpermutation; we require only that the subpermutation be contiguous, meaning the similar elements may finish distributed across both ends. This

**Algorithm 5** MULTPERM, a new multiset permutations algorithm.

```

/* Initialise */
1.  procedure init_Mul
2.    read k
3.    for i = 1 to k do read m[i]
4.    set n to the sum of all m
5.    for i = 1 to k do
6.      set o[i] to the sum of m[1] to [i - 1]
7.    for i = 1 to k do
8.      set r[i] to the sum of m[i] to [k]
9.    for i = 1 to k do
10.     for j = 1 to m[i] do
11.       perm[j + o[i]] = i
12.     for i = 1 to k do d[i] = 1
13.     for i = 1 to k do e[i] = i
14.     for i = 1 to k do
15.       for j = 1 to m[i] do comb[i][j] = j
16.       comb[i][m[i] + 1] = 2r[i] + 1
17.       z[i] = m[i] + 1
18.     for i = 1 to k - 1 do
19.       a[i] = i + 1
20.       set b[i] to 1 if m[i] is 1 or r[i] is even,
21.       else 2
22.     j = 1

/* Generate */
23. procedure next_Mul
24.   e[1] = 1
25.   determine x and y as per Chase, but
26.   using comb[j] and z[j]
27.   perm[x + o[j]] = perm[y + o[j]]
28.   perm[y + o[j]] = j
29.   if a[j] < k then
30.     o[a[j]] = o[a[j]] - b[j] × d[j]
31.     add 1 to a[j]
32.   if (comb[j][m[j] - b[j]] is minimal
33.   and comb[j][m[j] - b[j] + 1] is maximal)
34.   or comb[j][m[j] - b[j]] is minimal then
35.     e[j] = e[j + 1]
36.     e[j + 1] = j + 1
37.     d[j] = -d[j]
38.     a[j] = j + 1
39.     j = e[1]

/* Main */
40. init_Mul
41. print perm
42. while j is not k do
43.   next_Mul
44.   print perm

```

more relaxed requirement meant we had more combinations algorithms to choose from than Korsh and LaFollette. Besides requiring that the 0s (in terms of bit vector notation) finish as a contiguous substring, we also required, as per our recursive algorithm, that the relative order of 0s be maintained, and that the algorithm be reversible in  $O(1)$  time. We preferred that the algorithm's transpositions be limited to  $O(1)$  distance, as this would avoid significant extra book-keeping.

The combinations algorithm we chose was that of Chase (1989), shown in Algorithm 4. We regret that a full explanation of Chase's algorithm is outside the scope of this paper, but we hope that our overview will satisfy the reader's curiosity enough to accept Chase's algorithm as component for use in our MULTPERM algorithm.

1. 1 1 2 2 3	11. 2 3 2 1 1	21. 1 1 3 2 2
2. 1 2 1 2 3	12. 2 3 1 2 1	22. 1 3 1 2 2
3. 2 1 1 2 3	13. 2 1 3 2 1	23. 3 1 1 2 2
4. 2 2 1 1 3	14. 1 2 3 2 1	24. 3 2 1 1 2
5. 2 1 2 1 3	15. 1 2 3 1 2	25. 3 1 2 1 2
6. 1 2 2 1 3	16. 2 1 3 1 2	26. 1 3 2 1 2
7. 1 2 2 3 1	17. 2 3 1 1 2	27. 1 3 2 2 1
8. 2 1 2 3 1	18. 2 1 1 3 2	28. 3 1 2 2 1
9. 2 2 1 3 1	19. 1 2 1 3 2	29. 3 2 1 2 1
10. 2 2 3 1 1	20. 1 1 2 3 2	30. 3 2 2 1 1

Figure 8: MULTPERM output for  $k = 3$ ,  $m = 2, 2, 1$ .

We have altered the algorithm so that all decision making is clear (optimised shortcuts have been replaced with assumed original conditional statements) and so that the algorithm can run both forwards and backwards. Its 1- or 2-apart transpositions means the relative order 0s is easily maintained. It is easily reversible, requiring only the inversion of one boolean function. It starts with 1s *all-left* ( $1^n 0^k$ ) and finishes in one of two easily recognisable arrangements: *one-right* ( $1^{n-1} 0^{k-n} 1$ ) iff  $n = 1$  or  $k$  is even; or *two-right* ( $1^{n-2} 0^{k-n} 11$ ) iff  $n > 1$  and  $k$  is odd. Another benefit is that it uses very few variables.

The variables in Chase's algorithm are:  $comb_{1..n+1}$ , the current combination;  $z$ , the position in  $comb$  of the first non-minimal element, that is the lowest  $i$  such that  $comb_i > i$ ; and  $x$  and  $y$ , the values exiting and entering  $comb$  respectively. Values for  $n$  and  $r$  are read from the user. All  $comb_i$  are set to  $i$ , except  $comb_{n+1}$  which is initialised to  $2r + 1$ . Variable  $z$  is set to  $n + 1$ .

The functions in Chase's algorithm are:  $adj(i)$ , which returns whether  $comb_i$  and  $comb_{i+1}$  are adjacent, that is whether  $comb_i + 1 = comb_{i+1}$ ; and  $inc(i)$ , which returns whether  $comb_i$  is increasing or not, which is equivalent to  $comb_{i+1} \bmod 2$ . In Chase's algorithm, each position's direction is determined by the next position's parity; inverting function  $inc$  makes the algorithm run in reverse.

The many nested if-then-else statements evaluate directions and adjacencies of certain elements within one or two positions of  $comb_z$ , the first non-minimal element. These classify the current state of  $comb$  and  $z$  into one of ten cases, which determine which transposition to make. We have isolated this transposition in procedure *move*, whose parameters are the position, direction and span (distance) of the transposition. Output for Chase's algorithm is shown in Figure 7.

To fuse a loopless multiset permutations algorithm from Williamson's and Chase's algorithms required surprisingly few modifications. Each of the  $k$  groups of similar elements moves as a combination through its subpermutation, requiring its own Chase data. Thus, Chase's variable  $z$  and array  $comb_{1..n}$  were extended by one dimension each to  $z_{1..k}$  and  $comb_{1..k, 1..m_i}$  respectively. Each  $comb_i$  is of length  $m_i$ . An extra terminating condition was added, since Chase's algorithm would now be running backwards as well as forwards. Williamson's algorithm was altered to start with  $j = 1$  instead of  $n$ , and its second (incrementing/decrementing) step was replaced with the modified Chase's algorithm. Thus Williamson's algorithm selects the similar elements  $j$  to move, and Chase's algorithm moves them among  $subp_j$  in combination fashion, using  $comb_j$  and  $z_j$ .

Algorithm MULTPERM, our new multiset permutations algorithm, is given in Algorithm 5; a complete C++ program is given in Appendix B. The appendix was written to match the style of Korsh and LaFollette's algorithm, for a more accurate comparison.

	Uniform	
	KL04	MULTPERM
Permutations	168,168,000	168,168,000
Mean Time (s)	31.3	21.5
	Varied	
	KL04	MULTPERM
Permutations	75,675,600	75,675,600
Mean Time (s)	14.2	9.4

Table 1: Results from experimental evaluation showing that MULTPERM runs 31–34% faster than KL04. Evaluation was over two multisets with many million permutations; multiplicities were uniform  $\{3, 3, 3, 3, 3\}$  and varied  $\{2, 3, 5, 2, 3\}$  respectively. Both algorithms generated the expected numbers of permutations.

To translate the relative transpositions of elements in Chase combinations to absolute transpositions in the multiset permutation,  $perm_{1\dots n}$ , required several new variables:  $o_{1\dots k}$ , the absolute offsets for each combination;  $a_{1\dots k}$ , which keeps track of the offsets that have been updated for the current  $j$ 's Chase cycle; and  $b_{1\dots k}$ , the number (one or two) of elements that finish right for each combination. For any selected group of similar elements  $j$ , each complete Chase cycle displaces subsequent subpermutations by  $b_j$  (reverse cycle) or  $-b_j$  (forward cycle) positions. Thus all  $o_i$  for  $i > j$  must be updated during the Chase cycle for  $j$ . This is achieved using the time-stealing method mentioned in Section 2, in which what would be a for-loop is distributed over subsequent calls to function *next*. In this case, over several calls to *next*,  $a_j$  counts from  $j + 1$  to  $k - 1$ , and each  $o_{a_j}$  is incremented or decremented by  $b_j$ . To recognise when forward Chase cycles are complete, that is when combinations are one-right or two-right, array  $r_{1\dots k}$  stores the maximum value that may appear in each of the combinations.

Reversing Chase's algorithm requires no reinitialisation. We have tied function *inc* to Williamson's array  $d$ , so changing the sign of  $d_j$  inverts *inc*, reversing the algorithm.

MULTPERM runs in constant time per object and requires linear space. Referring to Algorithm 5, lines 24, 31–35, and 36 correspond to the first, third and fourth steps of Williamson's algorithm respectively. Line 25 is where Chase's algorithm is used, while lines 26–30 translate Chase's transpositions to the multiset permutation; these steps together correspond to the second step of Williamson's algorithm. A sample output of MULTPERM for  $k = 3$ ,  $m = \{3, 2, 1\}$  is shown in Figure 8.

We experimentally evaluated MULTPERM against Korsh and LaFollette's algorithm, which we label KL04. Both programs were implemented in C++, and the structure, procedure calls, and I/O were made as similar as possible; this is evident in Appendix B. Timing included the initialisation and memory-clearing procedures. By convention, output statements were replaced by statements incrementing a counter, whose final value was output to verify that the correct number of objects were generated.

We ran the experiment over two multisets, each with millions of distinct permutations, but with *uniform* and *varied* multiplicities respectively: both multisets had  $k = 5$  distinct integers, but the uniform had  $m = \{3, 3, 3, 3, 3\}$  and the varied had  $m = \{2, 3, 5, 2, 3\}$ . Our mean times and standard deviation were produced over 10 iterations.

As can be seen from Table 1, MULTPERM runs 31–34% faster than KL04 across both multisets.

MULTPERM generated the 168 million permutations of the uniform multiset in an average of 21.5s ( $\sigma = 0.11$ ) to KL04's 31.3s ( $\sigma = 0.11$ ), and the 75 million permutations of the varied multiset in 9.4s ( $\sigma = 0.05$ ) to KL04's 14.2s ( $\sigma = 0.05$ ). We attribute the extra speed and simplicity of MULTPERM over KL04 to the advantages of our component algorithm for combinations over that used by Korsh and LaFollette.

## 6 Conclusion

There is room for further investigation and improvement in both of the problems we applied our framework to. Algorithm MIXPAR could be modified to allow a variable number of parenthesis types, and nesting a second Gray coder could allow it to cycle through another property of parentheses, e.g. colour. Regarding MULTPERM, there may yet be more advantageous loopless combinations algorithms than Chase's.

More interesting would be investigating which other combinatorial generation problems can be solved looplessly by fusion. Both of the problems we addressed quite obviously comprise two combinatorial subproblems, and therefore were conducive to this approach. We wonder:

- Can fusion be used for more complicated combinatorial generation problems?
- Can fusion be used where the decomposition into subproblems is not so obvious?

## Acknowledgements

The authors would like to thank the referees for their constructive criticism.

## References

- Chase, P. J. (1989), 'Combination generation and graylex ordering.', *Congressus Numerantium* **69**, 215–242.
- Ehrlich, G. (1973), 'Loopless algorithms for generating permutations, combinations, and other combinatorial configurations.', *J. ACM* **20**(3), 500–513.
- Johnson, S. M. (1963), 'The generation of permutations by adjacent transpositions.', *Math. Comp.* **17**, 282–285.
- Korsh, J. F. & LaFollette, P. S. (2004), 'Loopless array generation of multiset permutations.', *The Computer Journal* **47**(5), 612–621.
- Korsh, J. & Lipschutz, S. (1997), 'Generating multiset permutations in constant time.', *J. Alg.* **25**(2), 321–335.
- Nijenhuis, A. & Wilf, H. S. (1975), *Combinatorial Algorithms*, Academic Press.
- Reingold, E. M., Nievergelt, J. & Deo, N. (1977), *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall.
- Savage, C. (1997), 'A survey of combinatorial gray codes', *SIAM Review* **39**(4), 605–629.
- Trotter, H. F. (1962), 'Perm (algorithm 115)', *Commun. ACM* **5**(8), 434–435.

Vajnovszki, V. (2003), 'A loopless algorithm for generating the permutations of a multiset.', *Theor. Comput. Sci.* **307**(2), 415–431.

Wilf, H. S. (1989), *Combinatorial Algorithms: An Update*, SIAM.

Williamson, S. G. (1985), *Combinatorics for Computer Science*, Computer Science Press.

Xiang, L. & Ushijima, K. (2001), 'On  $o(1)$  time algorithms for combinatorial generation.', *Comput. J.* **44**(4), 292–302.

## A mixpar.cpp

```
/* Same style as Appendix B for consistency. */
#include <iostream>

using namespace std;

int n, j, *d, *e, jj, *dd, *ee, *l, *r, *s, t, i, num;
char *par, c;

void init() {
    cin>>n;
    par = new char[2*n+1]; d = new int[n+1]; e = new int[n+2];
    dd = new int[n+1]; ee = new int[n+2]; l = new int[n+1];
    r = new int[n+1]; s = new int[2*n+1];
    for (i=1; i<=n; i++) { par[2*i-1] = '('; par[2*i] = ')'; }
    for (i=1; i<=n; i++) { d[i] = 1; dd[i] = 1; }
    for (i=1; i<=n+1; i++) { e[i] = i-1; ee[i] = i-1; }
    for (i=1; i<=n; i++) { l[i] = 2*i-1; }
    for (i=1; i<=2*n; i++) { s[i] = i; }
    j = n; jj = n; t = n; num = 1;
}

void output() {
    cout<<num<<" ";
    for (i=1; i<=2*n; i++) { cout<<par[i]<<" "; }
    cout<<endl;
}

void next() {
    if (jj > 0) {
        ee[n+1] = n;
        if (dd[1] > 0 && jj == t) {
            r[jj] = s[l[jj]+1];
            if (jj > 1) { s[l[jj]] = s[r[jj]+1]; t = t-1; }
        }
        if (par[l[jj]] == '(')
            { par[l[jj]] = '['; par[r[jj]] = ')'; }
        else { par[l[jj]] = '('; par[r[jj]] = ')'; }
        ee[jj+1] = ee[jj]; ee[jj] = jj-1; dd[jj] = -dd[jj];
        if (dd[1] < 0 && jj == t) { s[l[jj]] = l[jj]; t = t+1; }
        jj = ee[n+1];
    } else {
        par[l[1]] = '('; par[r[1]] = ')';
        jj = n; t = n;
        dd[1] = 1; ee[n] = n-1;
        e[n+1] = n; i = l[j];
        if (d[j] > 0) {
            if (l[j] == 2*j-1) { l[j] = l[j-1]+1; }
            else { l[j] = l[j]+1; }
        } else {
            if (l[j] == l[j-1]+1) { l[j] = 2*j-1; }
            else { l[j] = l[j]-1; }
        }
        c = par[i]; par[i] = par[l[j]]; par[l[j]] = c;
        if (l[j] > 2*j-3)
            { e[j+1] = e[j]; e[j] = j-1; d[j] = -d[j]; }
        j = e[n+1];
    }
    num++;
}

void clean() {
    delete[] par; delete[] d; delete[] dd; delete[] e;
    delete[] ee; delete[] l; delete[] r; delete[] s;
}

int main() {
    init();
    output();
    while (j != 1 || jj != 0) {
        next();
        output();
    }
    clean();
}
```

## B multperm.cpp

```
/* Same style as Korsh and LaFollette 2004 for comparison. */
#include <iostream>
using namespace std;

int k, n, j, x, y, i, u, v, w, num, *perm, **comb, *m, *d,
*e, *o, *r, *z, *a, *b;

void init() {
    cin>>k; n = 0; m = new int[k+1];
    for (i=1; i<=k; i++) { cin>>m[i]; n += m[i]; }
    perm = new int[n+1];
    comb = new int*[k+1];
    for (i=1; i<=k; i++) { comb[i] = new int[m[i]+2]; }
    d = new int[k+1]; e = new int[k+1]; o = new int[k+1];
    r = new int[k+1]; z = new int[k+1]; a = new int[k+1];
    b = new int[k+1];
    o[1] = 0; for (i=2; i<=k; i++) { o[i] = o[i-1]+m[i-1]; }
    r[k] = m[k]; for (i=k-1; i>=1; i--) { r[i] = r[i+1]+m[i]; }
    for (i=1; i<=k; i++)
        { for (j=1; j<=m[i]; j++) { perm[j+o[i]] = i; } }
    for (i=1; i<=k; i++) { d[i] = 1; }
    for (i=0; i<=k+1; i++) { e[i] = i; }
    for (i=1; i<=k; i++) {
        for (j=1; j<=m[i]; j++) { comb[i][j] = j; }
        comb[i][m[i]+1] = 2*r[i]+1;
        z[i] = m[i]+1;
    }
    for (i=1; i<=k-1; i++)
        { a[i] = i+1; b[i] = 1+(m[i]>1 && r[i]%2); }
    j = 1; num = 1;
}

int adj(int i) {
    return comb[j][i]+1 == comb[j][i+1];
}

int inc(int i) {
    return comb[j][i+1]%2 == d[j]>0;
}

void output() {
    cout<<num<<" ";
    for (i=1; i<=n; i++) { cout<<perm[i]<<" "; } cout<<endl;
}

void next() {
    e[1] = 1;
    if (z[j] == 1) {
        v = 1;
        if (inc(1)) {
            if (adj(1)) { u = 2; w = 2*inc(2)-1; }
            else { u = 1; w = 1; };
        } else { u = 1; w = -1; };
    } else {
        if (inc(z[j]-1))
            { u = (z[j]>2 && inc(z[j]-2))+1; v = z[j]-u; w = 1; }
        else { v = z[j]; u = 1+adj(v); w = 2*inc(v-1+u)-1; }
    }
    i = v+(w-1)*(u-1)/-2;
    x = comb[j][i]; y = x+u*w;
    comb[j][i] = x+w; comb[j][i+(u-1)*w] = y;
    z[j] = z[j]-((comb[j][v] == v)*u*w-(v<z[j])*u);
    perm[x+o[j]] = perm[y+o[j]]; perm[y+o[j]] = j;
    if (a[j]<k) { o[a[j]] = o[a[j]]-b[j]*d[j]; a[j] = a[j]+1; };
    if (comb[j][m[j]-b[j]+1] == r[j]-b[j]+1
        && comb[j][m[j]-b[j]] == m[j]-b[j] || comb[j][m[j]] == m[j])
        { d[j] = -d[j]; e[j] = e[j+1]; e[j+1] = j+1; a[j] = j+1; }
    j = e[1]; num++;
}

void clean() {
    for (i=1; i<=k; i++) { delete[] comb[i]; }
    delete[] perm; delete[] comb; delete[] a; delete[] b;
    delete[] d; delete[] e; delete[] m; delete[] o; delete[] r;
    delete[] z;
}

int main() {
    init();
    output();
    while (j != k) {
        next();
        output();
    }
    clean();
}
```