O(1) Time Algorithms for Combinatorial Generation by Tree Traversal

Tadao Takaoka

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
E-mail: tad@cosc.canterbury.ac.nz

There are algorithms for generating combinatorial objects such as combinations, permutations and well-formed parenthesis strings in O(1) time per object in the worst case. Those known algorithms are designed based on the intrinsic nature of each problem, causing difficulty in applying a method in one area to the other. On the other hand, there are many results on combinatorial generation with minimal change order, in which just a few changes, one or two, are allowed from object to object. These results are classified in a general framework of combinatorial Gray code, many of which are based on a recursive algorithm, causing O(n) time from object to object. To derive O(1) time algorithms for combinatorial generation systematically, we formalize the idea of combinatorial Gray code by a twisted lexico tree, which is obtained from the lexicographic tree for the given set of combinatorial objects by twisting branches depending on the parity of the nodes. An iterative algorithm which traverses this tree will generate the given set of combinatorial objects in O(1) time as well as with a fixed number of changes from the present combinatorial object to the next. Although the idea of twisted lexico tree is not new, the mechanism of tree traversal and computation of changing places are new in this paper. As examples of this approach, we present new algorithms for generating well-formed parenthesis strings and combinations in O(1) time per object. The generation of combinations is done "in-place", that is, taking O(n) space to generate combinations of n elements out of r elements. Previous algorithms take O(r) space to represent a combination by a binary vector of size r.

1. Introduction

Generation of combinatorial objects such as combinations, permutations, and well-formed parenthesis strings, or parenthesis strings for simplicity, is a well studied area, documented in Reingold, Nievergelt and Deo [1], Nijenhuis and Wilf [2], Wilf [3], and Savage[4]. We consider the generation of combinations and parenthesis strings in this paper. Since there are at least an exponential number of those kinds of combinatorial objects, it is not hard to see that the lexicographic order generation of those objects based on a recursive algorithm takes O(f(n)) time, where there are f(n) objects of length n. That is, it is easy to generate those objects in O(1) time per object on average.

A less trivial problem is whether we can generate those objects in O(1) time per object in the worst case. There are such algorithms with O(1) worst case time. To name just a few, Bitner, Ehrlich and Reingold [5], Ehrlich [6], and Lehmer [7] for combinations, Johnson [8] and Heap [9] for permutations, Korsh and Lipschutz [10] for multiset permutations, and Mikawa and Takaoka [11] for parenthesis strings. Johnson's and Heap's algorithms for permutations take O(n) time from object to object, but it is straightforward to convert to ones with O(1) time as in [6]. If we relax the requirement from O(1) time to a fixed number of changes from object to object, which we refer to as O(1) changes, there are numerous results: Nijenhuis and Wilf [2] and Eades and McKay [12] for combinations, Proskurowski and Ruskey [13] and Ruskey and Proskurowski [14] for parenthesis strings, etc. Ruskey and Proskurowski generate parenthesis strings by adjacent transpositions only when n is even where the length of a string is 2n. Recently Walsh [15] successfully converted their recursive algorithm into an iterative one. Vjanowski [16] has yet another O(1) time algorithm based on a different approach.

In this paper, we propose a unified approach to the generation of combinatorial objects in O(1) time per object in the worst case, using the concept of twisted lexico tree, a computation method for changing places, and a tree traversal technique. Roughly speaking, the twisted lexico tree for a set of combinatorial objects can be obtained from the lexicographic tree for the set

by twisting branches depending on the parity of each node. We give an even parity to the root. As we traverse the children of a node, we give an even and odd parity alternatingly to the child nodes. If a node gets even, the branches from the node remain intact. If it gets odd, the branches from the node are arranged in the reverse order. This concept of parity is applied to the entire tree globally, so that the constant change property can be easily shown for each set of combinatorial objects. The global parity is more transparent than the local parity in [11], which is defined locally in each subtree and useful only for parenthesis strings. The lexico tree and the concept of global parity were introduced in Zerling [17] and Lucas [18] respectively. The computation of changing places is done by identifying a changing place at any level of the tree, called the difference point, and the corresponding changing position down the tree, called the solution point. The concept of this paper can be viewed as a refinement of combinatorial Gray code introduced in Joichi, White and Williamson [19], Wilf [3] and Savage [4], which is in turn a generalization of the generation of binary reflected Gray code [20].

Based on this approach, we derive a new O(1) time algorithm for generating parenthesis strings in an order different from that in [11] and an in-place algorithm for generating combinations of n elements out of r elements in O(1) time per combination. Here "in-place" means it requires only O(n) space. Only O(1) average time is known for the in-place generation of combinations in Nijenhuis and Wilf [11] with the revolving door function which requires O(n) time in the worst case. Eades and McKay also considered an in-place algorithm, which generate combinations with O(1) changes but requires O(n) time. O(1) worst case time is known for combination generation in [5], [6], [7], and so forth, at the cost of O(r) space of a binary vector of size r to represent a set of n elements out of r elements.

2. Constant Change Generation

Let $\Sigma = \{\sigma_0, \dots, \sigma_{r-1}\}$ be an alphabet for combinatorial objects. A combinatorial object is a string $a_1 \dots a_n$ of length n such that each a_i is taken from Σ and satisfies some property. A total order is defined on Σ with $\sigma_i < \sigma_{i+1}$. Let Σ^n be the set of all possible strings on Σ of length n. The lexicographic order $<$ on Σ^n is defined for $\mathbf{a} = a_1 \dots a_n$ and $\mathbf{b} = b_1 \dots b_n$ by

$$\mathbf{a} < \mathbf{b} \Leftrightarrow \exists j\ (1 \le j \le n)\ a_1 = b_1, \dots, a_{j-1} = b_{j-1}, a_j < b_j.$$

Let $S \subseteq \Sigma^n$ be a set of combinatorial objects. The order < on S is defined by projecting the lexicographic order on Σ^n onto S. The lexicographic tree, or lexico tree for short, of S is defined in the following way. Each $\mathbf{a} \in S$ corresponds to a path from the root to a leaf. The root is at level 0. If $\mathbf{a} = a_1 \dots a_n$, a_i corresponds to a node at level i. We refer to a_i as label for the node. We sometimes do not distinguish between node and label. If **a** and **b** share the same prefix of length k, they share the path of length k in the tree. The children of each node are ordered according to the labels of the children. A path from the root to a leaf corresponds to a leaf itself, so **a** corresponds to a leaf. The combinatorial objects at the leaves are thus ordered in lexicographic order on S.

The twisted lexico tree of a set S of combinatorial objects is defined as follows together with the parity function. We proceed to twist a given lexico tree from the root to leaves. Let the parity of the root be even. Suppose we processed up to the i-th level. If the parity of a node v at level i is even, we do not twist the branches from v to its children. If the parity of v is odd, we arrange the children of v in reverse order. If we process all nodes at level i, we give parity to the nodes at level i+1 from first to last alternatingly starting from even. We denote the parity of node v by parity(v). When we process nodes at level i in the following algorithms, which are children of a node v such that parity(v)=p, we say the current parity of level i is p. Note that (labels of) nodes at level i are in increasing order if the parity of the parent is even, or equivalently if the current parity of level i is even. If the parity is odd, they are in decreasing order. We draw trees lying horizontally for notational convenience. We refer to the top child of a node as the first child and the bottom as the last child.

If the labels on the paths from the root to two adjacent leaves in the twisted lexico tree for S are different at a fixed number of nodes, we can generate S with the fixed number of changes from object to object. We call the maximum number of changes the degree of S, denoted by deg(S). The above statement is restated as follows; if deg(S) is bounded by a

constant, we can generate S from object to object with the constant changes. In this case, we say S satisfies the constant change property (CCP). A general framework of a recursive algorithm for generating S with constant changes is given in the following. Since most variables take integer values, we omit variable declarations. The current parity at level i is given by p[i], and the procedure "output" is only to see the result; for O(1) time this will be eliminated. We allow O(n) time for initialization. The parity 0 is for even and 1 for odd. Let Σ = {0, ... ,r-1}.

```
1.  procedure generate(i);
2.  var k;
3.  begin
4.    if i ≤ n then begin
5.      if p[i] = 0 then
6.        for k:=0 to r-1 do
7.          if k is valid then begin
8.            perform changes on a[i] and related positions;
9.            generate(i+1)
10.         end
11.     else {p[i] = 1}
12.       for k:=r-1 downto 0 do begin
13.         if k is valid then begin
14            perform changes on a[i] and related positions;
15.           generate(i+1)
16.         end;
17.       p[i]:=1-p[i]
18.     end
19.   else { i = n+1} output(a)
20. end;
21. begin {main program}
22.   read(n);
23.   initialize array a;
24.   generate(1)
25. end.
```

Algorithm 1. Recursive framework for generating combinatorial objects

Note that this algorithm may generate S with constant changes, but it takes O(n) time from object to object in the worst case caused by the overhead time of recursive calls. In many applications, we perform changes on a[i] and a[j] for some j at lines 8 and 14. Then the objects, that is, array "a", before and after the changes are different at i and j and the rests are the same. In this context, we say that the difference at i is solved at j. We refer to i and j as the difference point and the solution point. Generally the computation of the solution point is difficult and will depend on each application.

3. Recursion Removal

We devise in this section a general framework of an iterative algorithm which avoids the O(n) overhead time by recursive calls. Although this method is well known, we include this algorithm for other applications in this paper and show that the concept of parity can be maintained in the iterative algorithm. The array "up" is to keep track of positions of ancestors to which the algorithm comes back from leaves. Note that this algorithm takes O(1) time in the worst case if S satisfies the CCP and the changing mechanism is properly given. The variable "v_i" is for the next node to be processed at level i, which is maintained by some data structure in each application. A generic algorithm and illustration follow.

```
 1. initialize array a to be the first object in S;
 2. initialize v_1, ... , v_n to nodes on the path to the first object (top path);
 3. for i:=0 to n do up[i]:=i;
 4. for i:=0 to n do p[i]:=0;
 5. repeat
 6.   output(a);
 7.   i:=up[n]; up[n]:=n;
 8.   perform changes on a at v_i and related positions;
 9.   let v_i go to the next node at level i based on the current parity;
10.   if v_i is the last child of its parent then begin
11.     let v_i further go to the next node at level i;
12.     up[i]:=up[i-1]; up[i-1]:=i-1;
13.     p[i]:=1-p[i]
14.   end
15. until I=0.
```

Algorithm 2. Iterative tree traversal

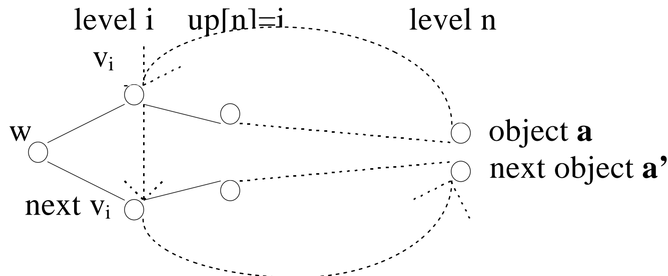

Figure 1. Going up and down

When we come to the last child of a parent (w in the above figure), we have to update up[i] to up[i-1] so that when we visit the last leaf of the subtree rooted at w, we can come back directly from the leaf to w or its ancestor if w itself is a last child. We refer to the paths from v_i to **a** and from next v_i to **a'** as the current path and the opposite path. A current path and opposite path consist of last children and first children respectively except for the left ends. If next v_i in the above figure is a last child, we further set v_i to the next node of next v_i, say u, so we can avoid O(n) time to set up the environment for such u's later. This is illustrated in the above figure by the path from "next v_i" to "next object **a'** ". That is, when we traverse down the current path, we prepare for the opposite path so that we can jump over the opposite path from level i to the leaf.

5. Parenthesis Strings

In this section, we consider the problem of generating all well-formed parenthesis strings of length 2n. We focus directly on the iterative algorithms based on Algorithm 2. We first define the characteristic sequence of a parenthesis string **a** = $a_1 ... a_n$ such that a_i is the number of right parentheses between the i-th and (i+1)th left parentheses for $i<n$ and a_n is the number of right parentheses after the n-th left parenthesis. This characerization is due to [9] and [19].

Example 1. For ( ) ( ) (( )), its characteristic sequence is given by (1, 1, 0, 2).

Let the sum s_i be defined by $s_i = a_1 + ... + a_i$ for i=0, ..., n, where $s_0 = 0$. Then $s_n = n$ and a_i can take the values 0, 1, ..., $i-s_{i-1}$, which are called

valid values. Note that $a_n = n - s_{n-1}$. Based on the order of a_i, we can define the lexico and twisted lexico trees for the set S of characteristic sequences.

Example 2. The twisted lexico tree for n=4 with parity function is given below. Even and odd parities are depicted by white and black circles.

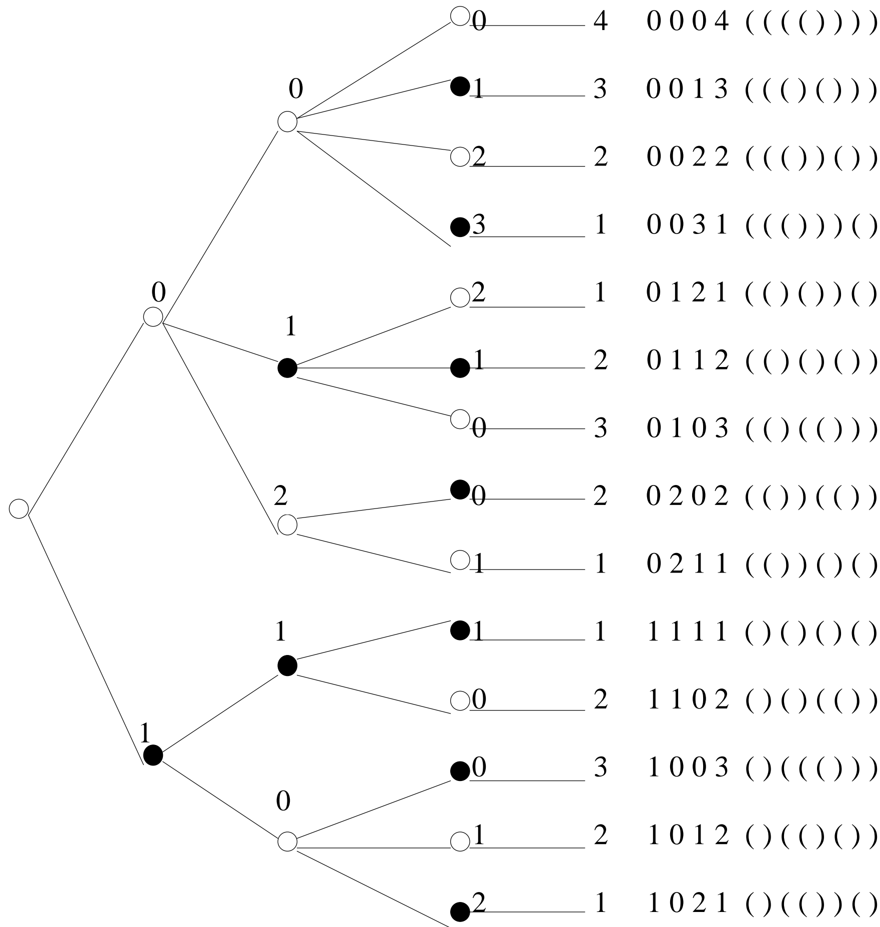

Figure 2. Twisted lexico tree for parenthesis strings

Theorem 1. The set S of characteristic sequences satisfies the constant change property.

Proof. Let **a** = $a_1 \ldots a_{i-1} a_i \ldots a_n$ and **a’** = $a_1 \ldots a_{i-1} a'_i \ldots a'_n$ (i<n) be two adjacent characteristic sequences. Then $a'_i = a_i + 1$ or $a'_i = a_i - 1$. Let parity(a_i) be even and hence parity(a'_i) be odd. Then a_{i+1} and a'_{i+1} are largest valid values at level i+1, which are $i+1-s_i$ and $i+1-s'_i$, where $s'_i = a_1 + \ldots + a_{i-1} + a'_i$. Note that

$$i + 1 - s'_i = i + 1 - s_i - 1, \quad \text{if } a'_i = a_i + 1,$$

and

$$i + 1 - s'_i = i + 1 - s_i + 1, \quad \text{if } a'_i = a_i - 1.$$

After this point, $a_j = a'_j$ for $i+1 < j \leq n$.

Now let parity(a_i) be odd and parity(a'_i) be even. Then $a_{i+1} = a'_{i+1} = 0$. Along the paths towards **a** and **a’**, we eventually hit k such that parity(a_{k-1}) is even and parity(a'_{k-1}) is odd, or k=n, that is, the leaves. Then similarly to the first case, we have

$$a'_k = a_k - 1, \text{ if } a'_i = a_i + 1, \text{ and}$$

$$a'_k = a_k + 1, \text{ if } a'_i = a_i - 1.$$

From this point on, we have $a_j = a'_j$ for $k < j \leq n$. Here we see that k is the solution point for the difference caused at a_i and a'_i.

In this proof we see that the distance c_i from the ancestors with difference to the solution point is given by k-i. Then the parenthesis string for **a**' can be obtained from that for **a** by swapping the two elements of the array containing parentheses at

$i + s_i + 1$ and $i + s_i + 1 + c_i$, if $a'_i = a_i + 1$

$i + s_i$ and $i + s_i + c_i$, if $a'_i = a_i - 1$.

Example 3. Let **a** = (0, 1, 0, 3) and **a**' = (0, 2, 0, 2). The corresponding parenthesis strings are (( ) (( ))) and (( )) (( )). Here we have i=2, $s_i = 1$ and $c_i = 2$. Note that $a'_i = a_i + 1$.

As in other applications, a_i increases if the current parity at level i is even, and decreases otherwise. Based on this theorem, we have the following iterative algorithm for generating parenthesis strings in O(1) time per string, in which we maintain only array "s" for the sum of array elements of array "a"; we do not use array "a" and identify each node by the value of s[i] at level i. Also we do not explicitly use the parity function; instead we use array "d" which determines the direction of s[i], increasing or decreasing. If d[i] = 1, it equivalently shows the parity is even at level i, and odd if d[i]= -1. We maintain the parenthesis strings in array "q". The Boolean array "solved" is used to indicate that the difference is solved at level i if solved[i]=true, and not solved otherwise. Note that if solved[i]=true, solved[j] will be true for j>i. The variable "b" is used to show the boundary value of s[i] for the last child, when parity is odd at level i. We go down the tree only to level n-1 since changes can be made at positions of level up to n-1. More explanation follows the algorithm.

```
 1. read(n);
 2. for i:=1 to n do begin q[i]:='('; q[i+n]:=')' end;
 3. for i:=1 to n do s[i]:=0;
 4. for i:=0 to n do d[i]:=1;
 5. for i:=0 to n do solved[i]:=false;
 6. for i:=0 to n do up[i]:=i;
 7. c[n-1]:=1;
 8. d[0]:=0;
 9. repeat
10.   output(q);
11.   i:=up[n-1];
12.   up[n-1]:=n-1;
13.   if d[i]>0 then swap(i+s[i]+1, i+s[i]+1+c[i])
14.             else swap(i+s[i], i+s[i]+c[i]);
15.   s[i]:=s[i]+d[i];
16.   if  up[i-1]=i-1 or solved[i-1] then b:=s[i-1] else b:=s[i-1]-d[up[i-1]];
17.   if (d[i]>0 and s[i]=i) or (d[i]<0 and s[i]=b) then begin
18.     if d[i]<0 and not solved[i-1] then begin
19.       solved[i]:=false;
20.       s[i]:=s[i]+d[up[i-1]];
21.     end else solved[i]:=true;
22.     up[i]:=up[i-1];
23.     up[i-1]:=i-1;
24.     if not solved[i-1] and solved[i] then c[up[i]]:=i-up[i];
25.     if not solved[i] and i=n-1 then c[up[i]]:=n-up[i];
26.     solved[i-1]:=false;
27.     d[i]:=-d[i]
28.   end
29. until i=0.
```

Algorithm 3. Iterative algorithm for parenthesis strings

We maintain array “c” to measure the distance from the difference point to the solution point. That is, “c” has the same meaning as “c_i“ in the proof of Theorem 1. The values of c are taken care of at lines 24 and 25. The boundary value for s[i] for the case of odd parity at level i is maintained in the variable “b”. If the difference is solved at level i-1, the value of b is equal to s[i-1] because at the parent the value of s[i-1] is the same for the next node at level i-1. If the difference is not solved at level i, the value of s[i-1] has been changed to s[i-1]+d[up[i-1]], so we have to subtract d[up[i-1]] to have a correct value for the boundary. When we hit a last child at line 17, we prepare an environment for the next node at the same level i. Those maintenance works are done at lines 18-21. The bookkeeping for the iterative framework is done at lines 22, 23 and 27. Line 26 is to avoid O(n) time to update array “solved” to false along the opposite path; each element of “solved” on the opposite path is updated by its child.

6. Combinations

We describe an O(1) algorithm for generating the set S of combinations of n elements out of the set {1, ..., r}. The CCP is slightly relaxed in the next theorem.

Theorem 2. The set S satisfies the constant change property.

Proof. Let $\mathbf{a} = a_1...a_{i-1}a_i...a_n$ and $\mathbf{a}' = a_1...a_{i-1}a'_i...a'_n$ be two adjacent objects in the twisted lexico tree. If i=n, the CCP is clear. Let i<n. We consider two cases.

Case 1. Parity(a_i) is even. Then $a_{i+1} = a'_{i+1} = r-n+i+1$ which is the maximum possible value at level i+1. After this point on towards leaves, all values of **a** and **a**’ are equal.

Case 2. Parity(a_i) is odd. When we traverse on the path to the leaf for **a**, we have differences with corresponding elements of **a**’ until we hit an even parity or a leaf and we can solve the differences. Those differences along the paths from the difference point to the solution point form a shift by one place left (a new element at the right end) or one place right (a new element at the left end), meaning that if we maintain another array for containing combinations, we can go from one combination to the next with constant changes.

Example 4. The twisted lexico tree for combinations of 4 elements out of 6 elements is given with the contents of array “a” and “q” at the leaves.. See Figure 3.

The implementation is similar to that for parenthesis strings. Based on the structure of the twisted lexico tree of the set S of combinations, however, we need some more techniques to traverse the tree without causing O(n) time. There are many nodes which have only one child down to the leaf, that is , causing straight lines. Traversal of these lines downwards will cause O(n) time. Having one child is caused by a node with the maximum possible label at the level. To control the position to which we come down, we keep an array “down”. In the above example, we use “up” to go from A to B, and use “down” to go from C to D. A snapshot of the movement is like (..., F,A,B,C,D,E, ...). As we saw in the proof of the previous theorem, we can not generate combinations in labels attached to the nodes of the twisted lexico tree with constant changes. Let array **a** contain those labels. Then we have a situation with $\mathbf{a} = (a_1, ..., a_{i-1}, a_i, ..., a_j, ..., a_n)$ and $\mathbf{a}' = (a_1, ..., a_{i-1}, a_i+1, ..., a_j+1 ..., a_n)$, where $a_{k+1}=a_k+1$ for $i \le k < j$, or a symmetric case where “+1” in the above is replaced by “-1”. Note that “i” is the difference point and “j” is the solution point. In the first case, “a_i” goes out of the combination and “a_j+1” comes into the combination. We keep the combinations in array “q” and use array “a” for book-keeping as we did for parenthesis strings. As we cannot change j-i+1 places in “a”, we just change a_i and a_j, and change the contents of q correspondingly. Thus we do not implement line 11 of Algorithm 2 to prepare for the opposite path to the next object. To keep track of the positions of these elements in q, we use array ”pos”. Solution points are maintained in array “solve”. Since some parts of array “a” are not maintained to the corresponding labels in the tree, we use Boolean array “mark” to show no maintenance. When mark[i]=true, the proper value of a[i-1] is given by its child a[i]. The main part of the algorithm follows in which up[i], down[i], solve[i], and pos[i] are initialized to i for all i. The values of d[i] are initialized to 1 and those of “mark” to false. Line-by-line explanation of Algorithm 4 follows.

Line 4: Update the parent when maintenance is not done.

Lines 5-12: Update the contents of q.

Lines 5-7: Note that d[i+1] was altered in a previous step. Thus we regard parity(a_i) as even

despite d[i+1]<0. Move from B to C with i=1 for example in the figure.
Lines 8-10: Similarly move from F to A with i=2 for example.
Lines 10-12: Similarly move from D to E with i=2 for example.
Line 13: Take the next child.
Line 14: Change the value of a at the solution point.
Line 15: Originate the destination for descendants to come up to. This value will possibly propagate down in line 17.
Line 16: Check if the node is a last child.
Line 17: Compute the correct value of “up”.
Line 18: Let the ancestor know where to come down next.
Line 19: If parity is odd, update the solution point for ancestor by i. This may further be updated by larger i. Also signal that the parent a[i-1] is not updated for the next object. If up[i]=i-1, a[i-1] has been given a proper value of a[i]-1, but signalling “mark[i]=true” will not cause any harm.
Line 20: Change the parity at i.
Line 21: Go up if you hit a last child with even parity or at a leaf. Otherwise go down.
Line 22: Go down if you do not hit a last child.
Line 23: Go out of iteration if you come to the root.

Note. If n=r, we have only one combination and the tree of a straight line. All nodes are last children. As a change is made on “a” in the first half of the repeat loop, the “last child” condition at line 16 does not work, causing an infinite loop. To avoid this we can amend the algorithm so that lines 3-14 are executed only when a[i] is not a last child. Alternatively we can detect the condition r=n at the beginning and finish the algorithm after outputting the one combination. We do not incorporate this note in the algorithm to make it as simple as possible.

```
 1. d[n+1]:= -1; up[0]:=0; i:=n;
 2. repeat
 3.   output(q);
 4.   if mark[i]=true then begin a[i-1]:=a[i]-1; mark[i]:=false end;
 5.   if d[i+1]<0 then begin
 6.     q[pos[a[i]]]:=a[i]+d[i]; pos[a[i]+d[i]]:=pos[a[i]]
 7.   end else
 8.   if d[i]>0 then begin
 9.     q[pos[a[i]]]:=a[solve[i]]+d[i]; pos[a[solve[i]]+d[i]]:=pos[a[i]]
10.   end else begin
11.     q[pos[a[solve[i]]]]:=a[i]+d[i]; pos[a[i]+d[i]]:=pos[a[solve[i]]]
12.   end;
13.   a[i]:=a[i]+d[i];
14.   if d[i+1]>0 then a[solve[i]]:=a[solve[i]]+d[i];
15.   up[i]:=i;
16.   if (d[i]>0) and (a[i]=r-n+i) or (d[i]<0) and (a[i]=a[i-1]+1) then begin
17.     up[i]:=up[i-1]; up[i-1]:=i-1;
18.     down[up[i]]:=i;
19.     if d[i]<0 then begin solve[[up[i]]:=i; mark[i]:=true end;
20.     d[i]:= -d[i];
21.     if (d[i]<0) or (i=n) then i:=up[i] else i:=down[i]
22.   end else i:=down[i]
23. until i=0;
24. output(q).
```

Algorithm 4. In-place algorithm for combinations

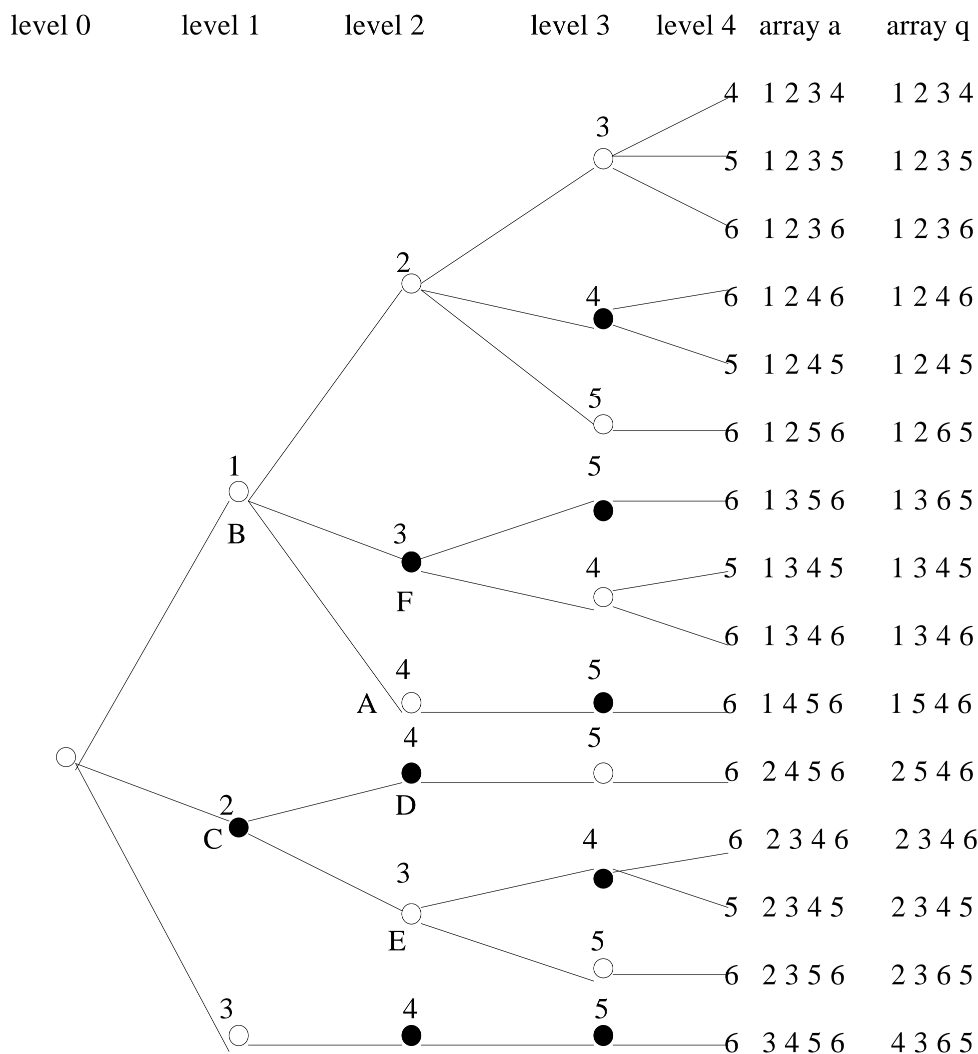

Figure 3. Twisted lexico tree for in-place combinations

7. Concluding Remarks

We developed several O(1) algorithms for generating combinatorial objects. Algorithms 3 and 4 are general enough to further adapt to other combinatorial objects. Recursive algorithms based on Algorithm 1, which were omitted in this paper, were first developed, and then converted to iterative ones based on Algorithm 2. Recursive algorithms are easier to develop since we can control the path to the leaf at the cost of O(n) time. The most difficult part is how to avoid this O(n) time by the aid of additional data structures.

It is straightforward to see that permutations and multiset permutations satisfy the constant change property, and thus recursive algorithms for generating them with constant changes are easy to implement. An O(1) time algorithm for generating multi-set permutations in-place has been successfully developed using the technique in this paper in Takaoka [20]. The previous algorithm by [10] is not an in-place version.

The set of multiset combinations is another challenging area. If we express a multiset combination by a vector whose i-th element holds the multiplicity of the i-th element, we can go from one combination to the next by increasing and decreasing multiplicities by one at two places. An O(1) time algorithm for generating multi-set combinations in this manner is designed and reported in [20]. Ruskey

and Savage [21] designed an algorithm for this problem with O(1) changes and left open the problem of even generating the set in O(1) average time.

After we successfully developed several O(1) time algorithms for combinatorial generation, it is interesting to see whether there is a well known set S of combinatorial objects with deg(S) > 2. The set of permutations with condition of $a_i \neq i$ for all i, that is, the set of derangements, does not satisfy the CCP, that is, deg(S) = O(n). It is not known whether there is an O(1) algorithm for this set.

Full Pascal programs for Algorithms 3 and 4 are attached as appendices.

Acknowledgement

The author is very grateful to anonymous referees whose careful review improved the quality of the paper to a great extent.

References

[1] Reingold, E.M., J. Nievergelt and N. Deo, Combinatorial Algorithms, Prentice-Hall, 1977.

[2] Nijenhuis, A and H.S. Wilf, Combinatorial Mathematics, Chapter 3, Next k-Subset of an n-Set (1975) pp. 21-34.

[3] Wilf, H.S., Combinatorial Algorithms: An Update, SIAM, Philadelphia, 1989.

[4] Savage, C., A survey of combinatorial Gray codes, SIAM Review, 39 (1997) 605- 629.

[5] Bitner, J.R., G. Ehrlich and E.M. Reingold, "Efficient Generation of Binary Reflected Gray Code and its Applications," CACM 19 (1976) pp. 517-521.

[6] Ehrlich, G., Loopless algorithms for generating permutations, combinations, and other combinatorial configurations, JACM, 20 (1973) 500-513.

[7] Lehmer, D.H., "The Machine Tools of Combinatorics," in Applied Combinatorial Mathematics (E.F. Beckenbach Ed.), Wiley, New York (1964) pp. 5-31.

[8] Johnson, S.M., "Generation of Permutations by Transpositions," Math. Comp. 15 (1963) pp. 282-285.

[9] Heap, B.R., "Permutations by Interchanges," Computer Journal 6 (1963) pp. 293-294

[10] Korsh, J and S. Lipschutz, Generating Multiset Permutations in Constant Time," Jour. Algorithms, 25 (1997) pp. 321-335.

[11] Mikawa, K and Takaoka, T, "Generation of Parenthesis Strings by transpositions," Proc. The Computing: The Australasian Theory Symposium (CATS '97) (1997) 51-58.

[12] Eades, P and McKay, B, Al algorithm for generating subsets of fixed size with a strong minimal change property, Info. Proc. Lett., 19 (1984) 131-133.

[13] Proskurowski, A. and F. Ruskey, Binary tree Gray codes, Jour. Algorithms, 6, (1985) 225-238.

[14] Ruskey, F. and A. Proskurowski, Generating binary trees by transpositions, Jour. Algorithms, 11, (1990) 68-84.

[15] Walsh, T.R., Generation of well-formed parenthesis strings in constant worst-case time, Jour. Algorithms, 29, (1998) 165-173

[16] Vjanowski, On the loopless generation of binary tree sequences, Info. Proc. Lett., 68 (1998) 113-117.

[17] Zerling, D, Generating binary trees by rotations, JACM, 32, (1985) 694-701.

[18] Lucas, J., The rotation graph of binary trees is Hamiltonian, Jour. Algorithms, 8, (1987) 503-535.

[19] Joichi, J.T., D.E. White, and S.G. Williamson, Combinatorial Gray codes, SIAM Jour. Comput., 9 (1980) 130-141.

[20] Takaoka, T., Generation of multi-set permutations and multi-set combinations in O(1) time, Technical Report, Univ. of Canterbury (1998).

[21] Ruskey, F. and Savage, C., A Gray code for combinations of a multi-set, European Jour. Combinatorics, 68 (1996) 1-8.

Appendix A. Pascal program for generating parenthesis strings

```
program ex(input,output);
var b,i,n:integer;
    c,d,s,up:array[-100..100] of integer;
    solved:array[0..100] of boolean;
    q:array[0..100] of char;
procedure out;
var i:integer;
begin
  for i:=1 to 2*n do write(q[i]:2); writeln
end;
procedure swap(i,j:integer);
var w :char;
begin
  w:=q[i]; q[i]:=q[j]; q[j]:=w
end;
begin {main}
  readln(n);
  for i:=1 to n do begin q[i]:=' (' ; q[i+n]:=' )' end;
  for i:=0 to n do s[i]:=0;
  for i:=1 to n do d[i]:=1;
  for i:=0 to n do solved[i]:=false;
  for i:=0 to n do up[i]:=i;
  c[n-1]:=1; d[0]:=0;
  repeat
    out;
    i:=up[n-1]; up[n-1]:=n-1;
    if d[i]>0 then swap(i+s[i]+1,i+s[i]+1+c[i])
              else swap(i+s[i],i+s[i]+c[i]);
    s[i]:=s[i]+d[i];
    if (up[i-1]=i-1) or solved[i-1] then b:=s[i-1] else b:=s[i-1]-d[up[i-1]];
    if (d[i]>0) and (s[i]=i) or (d[i]<0) and (s[i]=b) then begin
      if (d[i]<0) and (solved[i-1]=false) then begin
        solved[i]:=false;
        s[i]:=s[i]+d[up[i-1]];
      end
      else solved[i]:=true;
      up[i]:=up[i-1]; up[i-1]:=i-1;
      if (solved[i-1]=false) and (solved[i]=true) then c[up[i]]:=i-up[i];
      if (solved[i]=false) and (i=n-1) then c[up[i]]:=n-up[i];
      solved[i-1]:=false;
      d[i]:=-d[i]
    end;
  until i=0;
end.
```

Appendix B. Pascal program for generating combinations in-place

```
program ex(input,output);
var b,i,n,r:integer;
    c,d,pos,up,down,solve:array[-100..100] of integer;
    a,q:array[0..100] of integer;
procedure out;
var i:integer;
begin
  for i:=1 to n do write(q[i]:2);
  writeln
end;
begin {main}
  write('  input n, r '  ); readln(n,r);
  for i:=1 to n do begin a[i]:=i; q[i]:=i; up[i]:=i; down[i]:=i end;
  for i:=1 to n do begin solve[i]:=i; pos[i]:=i end;
  for i:=1 to n do d[i]:=1;
  d[n+1]:=-1; i:=n;
  repeat
    out;
    if d[i+1]<0 then begin
      q[pos[a[i]]]:=a[i]+d[i];
      pos[a[i]+d[i]]:=pos[a[i]];
    end else
    if d[i]>0 then begin
      q[pos[a[i]]]:=a[solve[i]]+d[i];
      pos[a[solve[i]]+d[i]]:=pos[a[i]];
    end else begin
      q[pos[a[solve[i]]]]:=a[i]+d[i];
      pos[a[i]+d[i]]:=pos[a[solve[i]]]
    end;
    a[i]:=a[i]+d[i];
    if d[i+1]>0 then a[solve[i]]:=a[solve[i]]+d[i];
    up[i]:=i;
    if (d[i]>0) and (a[i]=r-n+i) or
       (d[i]<0) and (a[i]=a[i-1]+1) then begin
      up[i]:=up[i-1]; up[i-1]:=i-1;
      down[up[i]]:=i;
      if d[i]<0 then solve[up[i]]:=i;
      d[i]:=-d[i];
      if (d[i]<0) or (i=n) then i:=up[i] else i:=down[i];
    end
    else i:=down[i];
  until i=0;
  out
end.
```