Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Processing Letters 90 (2004) 181–186

Information Processing Letters

www.elsevier.com/locate/ipl

Constant time generation of derangements

James F. Korsh [*], Paul S. LaFollette

Department of CIS, Temple University, Philadelphia, PA 19122, USA

Received 10 December 2002; received in revised form 13 January 2004

Communicated by L.A. Hemaspaandra

Abstract

This paper presents the first constant time generation algorithm for derangements—permutations with no fixed points. Each derangement is obtained from its predecessor by making either one transposition or one rotation of three elements. It also generalizes this to permutations with a bounded number of fixed points.

Keywords: Algorithms; Combinatorial problems; Loopless; Derangements; Gray code; Fixed points

1. Introduction

A derangement $d = d_1, d_2, \ldots, d_n$ of length n is a permutation of the integers $[n] = \{1, 2, \ldots, n\}$ for which $d_i \neq i$ for $i \in [n]$. The number of such derangements

$$N(n) = (n-1)\big(N(n-1) + N(n-2)\big)$$

with $N(1) = 0$ and $N(2) = 1$. Recently, Baril and Vajnovszki [1] have given a recursive constant average time algorithm for the generation of all length n derangements. That is, the time required to generate all $N(n)$ derangements is $O(N(n))$.

Erhlich [3] introduced the concept of generating or listing combinatorial objects using at most constant worst case time between them. A constant time, or loopless, generation algorithm must execute no more than a fixed number of constant time instructions to produce the next object from its predecessor. Such an algorithm cannot contain loops dependent upon the size of an object. We will present the first constant time algorithm to generate all length n derangements. This algorithm is based on the approach of Korsh and LaFollette [4], which was motivated by that of Takaoka [7].

2. The recursive algorithm

Suppose we start with an initial derangement d of length n. All remaining derangements of length n can be generated by the recursive algorithm (shown in Algorithm 2.1).

The initial d and how the next first integer is selected determine the actual order in which the derangements are generated. Making that integer first will require either one transposition or one rotation of three elements.

[*] Corresponding author.
E-mail addresses: korsh@temple.edu (J.F. Korsh), lafollet@joda.cis.temple.edu (P.S. LaFollette).

0020-0190/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2004.02.006

```
*To generate the remaining derangements*:
If $n$ is 2 then the only derangement is $d$
Else
  Generate the remaining derangements having the same
  first integer as $d$
  For every other integer that can be the first
    Make that integer the first and generate the
    remaining derangements in which it is the first
    integer.
```

Algorithm 2.1.

For example, for $n = 5$, Algorithm 2.1 could generate all 44 derangements in the order:

1.	2 3 4 5 1	23.	5 3 4 2 1
2.	2 3 1 5 4	24.	5 3 4 1 2
3.	2 3 5 1 4	25.	5 3 2 1 4
4.	2 5 1 3 4	26.	5 3 1 2 4
5.	2 5 4 3 1	27.	5 1 2 3 4
6.	2 5 4 1 3	28.	5 1 4 3 2
7.	2 1 4 5 3	29.	5 1 4 2 3
8.	2 1 5 3 4	30.	5 4 1 2 3
9.	2 4 5 3 1	31.	5 4 1 3 2
10.	2 4 5 1 3	32.	5 4 2 3 1
11.	2 4 1 5 3	33.	5 4 2 1 3
12.	4 1 2 5 3	34.	3 4 2 5 1
13.	4 1 5 2 3	35.	3 4 1 5 2
14.	4 1 5 3 2	36.	3 4 5 1 2
15.	4 5 1 3 2	37.	3 4 5 2 1
16.	4 5 1 2 3	38.	3 5 4 2 1
17.	4 5 2 1 3	39.	3 5 4 1 2
18.	4 5 2 3 1	40.	3 5 2 1 4
19.	4 3 2 5 1	41.	3 5 1 2 4
20.	4 3 1 5 2	42.	3 1 5 2 4
21.	4 3 5 1 2	43.	3 1 4 5 2
22.	4 3 5 2 1	44.	3 1 2 5 4

Generating 4, 8, 12, 27, 34 and 43 from their predecessors involved a rotation, and the rest, a transposition.

Algorithm 2.1 actually works as follows. It starts with d and, except for bookkeeping, executes either a transposition or a rotation to produce the next derangement. If m is the leftmost position involved in the transposition or rotation, all subderangements involving positions $m + 1$ to n and their integers have been generated. For example, in 18, when 3 is transposed with 5 to produce 19, $m = 2$, and all subderangements involving positions 3 to 5 and integers 1, 2 and 3 have been generated and, in 11, when 2, 4 and 1 rotate to produce 12, $m = 1$, and all subderangements involving positions 2 to 5 and integers 1, 3, 4 and 5 have been generated. The integer transposed or rotated into position m is actually the next integer that can be first in its subderangement and it replaces the current one. This next integer can be selected arbitrarily from among those eligible, i.e., those integers (except m) not yet selected, that are in positions $m + 1$ to n.

We want to create a non-recursive loopless version of Algorithm 2.1. To do so, it is necessary to be able to determine m, the next integer to be made the current one, and which transposition or rotation is required and to do it—all in at most a constant time. In the next section we will do this.

3. The constant time algorithm

Let the *current position m* be the rightmost position of a derangement for which all subderangements involving positions $m + 1$ to n and their integers have been generated. The integer in the current position is the *current integer in m*. The *next current integer* is the next integer that can become the current integer in m. Let $\mathbf{d} = i_1, \ldots, i_{m-1}, i_m, i_{m+1}, \ldots, i_n$ be the first derangement generated by Algorithm 2.1 with prefix $i_1, \ldots, i_{m-1}$. Algorithm 2.1 then generates all derangements with this prefix consecutively. It does this by first generating all subderangements involving positions m to n and their integers with the current integer in position m. Then, while there is a next current integer, it is made the current one and all subderangements involving positions m to n and their integers are generated.

We associate with m a list of those integers that can be the next current one. Lemma 3.1 and its proof provide the basis for determining these lists and for obtaining the successor of a derangement.

Lemma 3.1. *Let the current position be m and let* $\mathbf{d} = i_1, \ldots, i_{m-1}, i_m, i_{m+1}, \ldots, i_n$ *be the first derangement generated by Algorithm* 2.1 *with prefix* $i_1, \ldots, i_{m-1}$. *Every one of the integers in positions m to n that can be in position* $m - 1$, *other than m, will appear in position m as all subderangements involving positions m to n and their integers are generated by Algorithm* 2.1 *for* $1 < m < n$.

Proof. Integer m cannot be in position m of **d** or the derangement condition fails. Let $m < k \leqslant n$. If $i_k \neq m$ and $i_m \neq k$ then transposing i_k and i_m produces a subderangement with i_k at m. If $i_k \neq m$ and $i_m = k$ then the rotation $(k, k+1, m)$ produces a subderangement with i_k in position m, i_m in position $k+1$ and i_{k+1} in position k, as long as $k < n$. If $k = n$, the rotation $(m, n, n-1)$ produces a subderangement with i_k at m, i_{k-1} at k and i_m at $k-1$. □

In our example, in derangement 1, $m = 3$, the prefix is $i_1, i_2 = 2, 3$ and, as all subderangements involving 4, 5 and 1 are generated (1 through 3), since each can be in position m, they all appear in position m. Also, in derangement 3, $m = 2$, the prefix is $i_1 = 2$ and, as all subderangements involving 5, 1, 3 and 4 are generated (1 through 11), each of these, except $m = 2$, appears in position m. Note that *not* all integers appearing in m *can* be in $m - 1$.

Also, notice that, when m is $n - 2$, it is not the case that all integers that can appear in m will have appeared in $m + 1$.

The import of Lemma 3.1 is that we can monitor the integers that appear in position m to determine those that can be in $m - 1$. Moreover, the successor of d can be obtained, as indicated in the proof, by one transposition or a rotation of three elements.

We use an array L whose $(m-1)$th row will contain the integers that can be in $m - 1$ and an index, $t[m-1]$, into the row, to keep track of the next current integer. When all subderangements involving positions m to n have been generated, $t[m]$ will be 0 and all next current integers for m have been used. When this occurs, we say m has *finished*. The next current position must then be found.

In order to find the next current position in constant time, we use an array e, as in [8,9]. It contains information about the *ready* positions, those positions k whose subderangements involving $k+1$ to n, have *not* all been generated. When m has finished, we must stop adding integers to $(m-1)$'s list if it is not empty. If it is empty, we can resume adding integers to its list. The use of finished and ready is not unlike that of finished and mobile in Korsh and Lipschutz [6]. We use an array F in which $F[m-1]$ is CLOSED when integers must no longer be added to $m - 1$'s list and is not CLOSED otherwise. The lists for $m = n - 2$ or $n - 1$ are updated as special cases.

When the next current integer, say y, is to be used, it is necessary to know where it is in d. This is accomplished in constant time by keeping array P, d's inverse, in which $P[y]$ contains the index of y in d. Fig. 1 contains the C++ implementation of our loopless algorithm for $n > 1$.

It is important to note that the algorithm of Fig. 1 uses a specific initial d and corresponding P. However, given any derangement d and corresponding P, the algorithm actually generates all remaining derangements. This will be needed in Section 4.

4. Permutations with a bounded number of fixed points

Let $c(k) = c_1, c_2, \ldots, c_k$ be a k-combination of n sequence as in [2,9], where $1 \leqslant c_i < c_{i+1} \leqslant n$ for $1 \leqslant i < k$. Let $d(c(k))$ be a length n permutation of the integers in $[n]$ whose d_j corresponding to $j \in \{c_1, c_2, \ldots, c_k\}$ are a subderangement of integers in $\{c_1, c_2, \ldots, c_k\}$ and in which $d_j = j$ for $j \in [n] - \{c_1, c_2, \ldots, c_k\}$. Clearly $d(c(k))$ is a permutation with $n - k$ fixed points. For example, if $k = 5$, $n = 8$ and $c = 13\,468$ then $d(c(k))$ might be $d_1 = 3$, $d_2 = 2$, $d_3 = 4$, $d_4 = 6$, $d_5 = 5$, $d_6 = 8$, $d_7 = 7$, $d_8 = 1$.

This connection between $c(k)$ and its permutation $d(c(k))$ with $n - k$ fixed points provides the basis, Algorithm 4.1, for generating all such permutations in [1] and also for us.

In [1], `gen_up` started with a specific initial derangement, and generated all remaining derangements, while `gen_down` started with the last derangement from `gen_up`, and generated the remaining derangements in reverse order from `gen_up`. These were constant average time algorithms. They can be easily modified to obtain constant average time generation algorithms to generate all remaining $d(c(k))$'s as required by steps 2 and 5 of Algorithm 4.1. These modified versions, along with a "strong" Gray code

```
To generate all d with $n - k$ fixed points:
1. Create first $c(k)$ and its first $d(c(k))$.
2. Generate the remaining $d(c(k))$'s.
3. While (there is another $c(k)$)
        4. Generate the next $c(k)$ and its $d(c(k))$.
        5. Generate the remaining $d(c(k))$'s.
```

Algorithm 4.1.

```
#include <iostream.h>
const int CLOSED=1; int n, m, M, x, y, count, d[20], P[20], t[20],
L[20][20], F[20], e[20]; void initialize(), nextderang(), print();
int main()
{
   cout<<’’Enter n>1 ’’<<endl; cin>>n; initialize(); print();
   while (m!=0) {nextderang(); print();}
   cout<<’’The number of derangements is ’’<<count<<endl;
}
void initialize()
{
   for (int i=1; i<n; i++)
   {d[i]=i+1; t[i]=0; P[i+1]=i; F[i]=!CLOSED; e[i]=i-1;}
   d[n]=1; P[1]=n; count=0; m=n-2; t[m]=1; L[m][1]=d[n-1];
   if (n!=3) {t[m]++; L[m][t[m]]=d[n];} e[n]=m;
}
void nextderang()
{
   e[n]=n-1; x=d[m];
   //update the list of m’s left neighbor w.r.t. x and m
   if ((m!=1)&&(F[m-1]!=CLOSED))
{
      if (t[m-1]==0)
      {
         if (x!=m-1) {t[m-1]=1; L[m-1][1]=x;}
         if (P[m]>m) {t[m-1]++; L[m-1][t[m-1]]=m;}
      }
   }
   y=L[m][t[m]]; M=P[y]; t[m]--;
   //update the list of m’s left neighbor w.r.t. y
   if ((m!=1)&&(F[m-1]!=CLOSED))
   {if (y!=m-1) {t[m-1]++; L[m-1][t[m-1]]=y;}}
   d[m]=y; P[y]=m;
   if (m==n-1) {d[n]=x; P[x]=n;} else if (M!=x) {d[M]=x; P[x]=M;}
   else if (M!=n) {d[M]=d[M+1]; P[d[M]]=M; d[M+1]=x; P[x]=M+1;}
   else {d[M]=d[M-1]; P[d[M]]=M; d[M-1]=x; P[x]=M-1;}
   // if m is finished, remove it from the ready list and update F[m-1]
   if ((t[m]==0))
   {
      e[m+1]=e[m]; e[m]=m-1;
      if (m!=1) if (t[m-1]==0) F[m-1]=!CLOSED; else F[m-1]=CLOSED;
   }
   m=e[n];
   if (m==n-1)
   {
      if ((d[n-1]==n)||(d[n]==n-1)) {e[m+1]=e[m]; e[m]=m-1; m=e[n];}
      else {t[m]=1; L[m][1]=d[n]; F[m-1]=CLOSED;}
   }
   if (t[m]==0) // m must be n-2
   {
      if (d[n-1]!=n-2) {t[m]=1; L[m][1]=d[n-1];}
      if (d[n]!=n-2) {t[m]++; L[m][t[m]]=d[n];}
   }
}
void print()
{count++; cout<<endl; for (int i=1; i<=n; i++) cout<<d[i]<<’’ ’’;}
```

Fig. 1.

loopless algorithm for generating all k-combinations of n, were used to implement Algorithm 4.1 to obtain a constant average time generation algorithm for all permutations with $n-k$ fixed points. The initial $d(c(k))$ of step 1 was specified, `gen_up` was used for step 2 and in step 5, if `gen_up` was last used then `gen_down` was used next and vice-versa. The “strong” Gray code allows the $d(c(k))$ of step 4 to be obtained in constant time from the last $d(c(k))$ generated by step 2 or step 5, and differed in exactly three places from it.

It is straightforward to modify `nextderang` of Fig. 1 so that it will looplessly produce the next $d(c(k))$ with one transposition or a rotation of three elements. The modification, `nextfixed`, can then be used in

```
while (m!=0) {nextfixed(); print();}
```

to implement steps 2 and 5. Our loopless version of Algorithm 4.1 uses this, along with the loopless algorithm in [9] for the generation of the Eades and McKay [2] k-combination of n algorithm. This works here because `nextfixed` can start with the any initial derangement and generate the rest as noted in Section 3. A C++ implementation of our loopless algorithm is available at http://www.cis.temple.edu/~korsh/fixed.html.

A further generalization was described in [1] of a constant average time generation algorithm for all permutations with a bounded number of fixed points. That is, the generation of all permutations with $n-k$ fixed points for $k1 \leqslant k \leqslant k2$. Algorithm 4.2 is a version of that algorithm.

In [1] this was implemented in constant average time so, in step 4a, generating $c(k)$ from $c(k-1)$ and its $d(c(k))$ from the last $d(c(k-1))$ from step 3b or 5b *required more than* constant time. However, our loopless version does. This is accomplished by incorporating another loopless k-combination of n generation algorithm similar to the one used in our version of Algorithm 4.1, except it can generate the combinations in Eades and McKay order or in reverse order. It must be able to re-initialize itself in constant time. It is described in [5]. For the Eades and McKay order, the first k-combination of n is $c(k) = 12\ldots k$ and the last is

$$c(k) = (n-k+1)(n-k+2)\ldots n$$

```
To generate all d with n − k fixed points $k1 \leqslant k \leqslant k2$:
1. Create first $c(k1)$ and its $d(c(k1))$.
2. Generate the remaining $d(c(k1))$'s.
3. While (there is another $c(k1)$)
      a. Generate the next $c(k1)$ and its $d(c(k1))$.
      b. Generate the remaining $d(c(k1))$'s.
4. For $k$ from $k1+1$ to $k2$
      a. Create the first $c(k)$ from $c(k-1)$
         and its $d(c(k))$.
      b. Generate the remaining $d(c(k))$'s.
      5. While (there is another $c(k)$)
            a. Generate the next $c(k)$ and its $d(c(k))$.
            b. Generate the remaining $d(c(k))$'s.
```

Algorithm 4.2.

while the reverse starts with

$$c(k) = (n-k+1)(n-k+2)\ldots n$$

and ends with $c(k) = 12\ldots k$. Consequently, if we just ended with

$$c(k-1) = (n-k+2)(n-k+3)\ldots n$$

then

$$c(k) = (n-k+1)(n-k+2)(n-k+3)\ldots n$$

while if we just ended with $c(k-1) = 12\ldots(k-1)$ then $c(k) = 12\ldots(k-1)k$. This transformation can be done in constant time. Also, obtaining $d(c(k))$ from $c(k)$ and the last $d(c(k-1))$ can be done in constant time. It differs from the last $d(c(k-1))$ in exactly two places and requires one transposition.

As a result, we obtain a loopless generation algorithm for permutations with a bounded number of fixed points. A C++ implementation is available at http://www.cis.temple.edu/~korsh/boundedfixed.html.

5. Discussion

The constant average time derangement generation algorithm `gen_up(gen_down)` of [1] required one or two transpositions or a rotation of three elements. Our loopless version requires only one transposition or a rotation of three elements. Our loopless generation algorithms for permutations with $n-k$ fixed points and for bounded permutations similarly require one transposition or a rotation of three elements.

Finally, since our algorithm for loopless generation of derangements requires $O(n^2)$ storage, it remains an open problem to use $\mathrm{O}(n)$ storage. It should be observed from function `initialize()` in Fig. 1 that, although in fact L requires $\mathrm{O}(n^2)$ storage, the initialization needed for L takes only constant time and the rest of the initialization, only $\mathrm{O}(n)$ time.

Acknowledgement

The authors would like to thank the referees for their comments and suggestions which led to an improved presentation of the paper.

References

[1] J. Baril, V. Vajnovszki, Gray code for derangements, Discrete Appl. Math., in press; doi: 10.1016/j.dam.2003.06.002.

[2] P. Eades, B. McKay, An algorithm for generating subsets of fixed size with a strong minimal change property, Inform. Process. Lett. 19 (1984) 131–133.

[3] G. Ehrlich, Loopless algorithms for generating permutations, combinations, and other combinatorial objects, J. ACM 20 (1973) 500–513.

[4] J.F. Korsh, P.S. LaFollette, Loopless generation of linear extensions of a poset, Order 19 (2002) 115–126.

[5] J.F. Korsh, P.S. LaFollette, Loopless array generation of multiset permutations, CIS Department Technical Report, Temple University, Philadelphia, PA, October 2002.

[6] J.F. Korsh, S. Lipschutz, Generating multiset permutations in constant time, J. Algorithms 25 (1997) 321–335.

[7] T. Takaoka, An O(1) time algorithm for generating multiset permutations, in: ISAAC 99, in: Lecture Notes in Comput. Sci., vol. 1741, Springer, Berlin, pp. 237–246.

[8] V. Vajnovszki, On the loopless generation of binary tree sequences, Inform. Process. Lett. 68 (1998) 113–117.

[9] L. Xiang, K. Ushijima, On O(1) time algorithms for combinatorial generation, Comp. J. 44 (2001) 292–302.