BIT 20 (1980), 2–7

A NEW ALGORITHM FOR GENERATING DERANGEMENTS

SELIM G. AKL

Abstract.

A new algorithm for generating derangements based on a well known permutation generation method is presented and analysed. The algorithm is shown to be superior in storage and time requirements to the existing method.

Key words: analysis of algorithms, combinatorics, derangements, permutations, recursion.

1. Introduction.

Several algorithms have been designed recently to generate various combinatorial items like partitions, permutations, combinations, compositions, trees, Gray code, etc. [2, 3, 8]. In this paper we address the problem of listing derangements.

A derangement of the set of n integers $\{1,2,\ldots,n\}$ is defined as a permutation P of these integers which changes every element so that no integer appears in its natural position. Formally, if P is the ordered n-tuple $(p_1,p_2,\ldots,p_n)$, then P is a derangement of $\{1,2,\ldots,n\}$ provided that $p_i \neq i$ for $i=1,2,\ldots,n$. The number of derangements of n integers is given by [4]:

$$D(n) = n![1-(1!)^{-1}+(2!)^{-1}-\ldots+(-1)^n(n!)^{-1}]\,. \tag{1}$$

For example, there are nine derangements for $n=4$, namely

2143	3412	4321
2341	3142	4312
2413	3421	4123 .

If one is interested in generating the $D(n)$ derangements of the set $\{1,2,\ldots,n\}$, then the obvious method is to list all $n!$ permutations of this set rejecting the ones in which at least one element appears in its original position. However, a faster and more elegant technique would certainly be preferable. In the following we compare two algorithms — *DERANGE*1 and *DERANGE*2 — which possess these two desirable properties. Throughout, P will denote a one-dimensional array of size n storing a permutation of the set $\{1,2,\ldots,n\}$.

Received August 13, 1979. Revised January 3, 1980.

This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant NSERC-A3336.

2. The first algorithm.

It is known that the number of derangements can also be expressed as follows [4]:

$$D(n) = (n-1)[D(n-1)+D(n-2)] \quad \text{for } n \geqq 2 . \tag{2}$$

This recurrence relation is usually shown to be true by considering the derangements of the integers $\{1,2,\ldots,n\}$, in which $p_1 = m$ for $m \neq 1$. If, in turn, $p_m = 1$, then there are $D(n-2)$ ways to derange the remaining $n-2$ integers $\{2,3,\ldots,m-1,m+1,\ldots,n\}$. On the other hand, if we look at the mth position as being the (new) natural position for the integer 1, then it is required that $p_m \neq 1$, and thus there are $D(n-1)$ ways to derange the integers $\{1,2,\ldots,m-1, m+1,\ldots,n\}$. Noting that m can take the $n-1$ values $2,3,\ldots,n$, we conclude that relation (2) is true, in general, for $n \geqq 2$, with the boundary conditions $D(0)=1$ and $D(1)=0$.

A direct implementation of this idea led in [1] to the development of the recursive procedure *DERANGE* (m) which generates the $D(n)$ derangements of the n integers $\{1,2,\ldots,n\}$. The procedure together with its driver program are shown in Figure 1 where B is a one-dimensional array of size n such that upon entry to the procedure

$$b_i = \begin{cases} 1 & \text{if the element in } p_i \text{ must not be moved,} \\ 0 & \text{otherwise.} \end{cases}$$

```
for i = 1 to n do
    p_i ← i; b_i ← 0
DERANGE (n)

procedure DERANGE (m)
  if m > 1 and b_m = 0 then for i = m−1 to 1 do
    if b_i = 0 then
        p_i ↔ p_m; b_i ← 1; DERANGE (m−1)
        b_i ← 0; DERANGE (m−1); p_m ↔ p_i
  else if m > 1 and b_m = 1 then DERANGE (m−1)
      else if m = 1 and b_m = 1 then output (p_1, p_2, ..., p_n)
return
```

Fig. 1

Being a straightforward implementation of relation (2), *DERANGE* (m) is conceptually very simple. Its analysis, however, seems quite complicated. In contrast, another algorithm (also based on the principle of relation (2)) is presented in [5] which is slightly more difficult to derive but whose analysis is far less involved. Figure 2 gives this procedure *DERANGE*1 (m) together with its

driver program where X and Y are two one-dimensional arrays, both permutations of $\{1,2,\ldots,n\}$ such that

$$p_{x_i} = y_i \neq x_i \qquad m+1 \leqq i \leqq n$$

$$x_i = y_i \qquad 1 \leqq i \leqq m-2$$

and $$x_m \neq y_{m-1}, \qquad y_m \neq x_{m-1}$$

upon entry to the procedure.

```
for i = 1 to n do x_i ← y_i ← i
DERANGE1 (n)

procedure DERANGE1 (m)
  if m ≠ 0 then for k := 1 to m−1 do
    x_{m−1} ↔ x_k; x_{m−1} ↔ x_m; y_{m−1} ↔ y_k
    p_{x_m} ← y_m; DERANGE1 (m−1)
    p_{x_m} ← y_{m−1}; DERANGE1 (m−2)
    y_k ↔ y_{m−1}; x_m ↔ x_{m−1}; x_k ↔ x_{m−1}
  else output (p_1, p_2, ..., p_n)
return
```

Fig. 2

To analyse *DERANGE1*, let Q_m be the total number of procedure calls during the execution of *DERANGE1* (m).

Then, $$Q_0 = Q_1 = 0$$

$$Q_m = (m-1)(Q_{m-1}+Q_{m-2}+2) \qquad \text{for } m \geqq 2\,,$$

which has the solution

$$Q_m = m! \sum_{k=0}^{[m/2]} \frac{1}{(2k)!} - 1$$

$$\cong m!(\cosh 1) \cong 1.54308m!\,.$$

The driver program therefore makes $Q_n+1 \cong 1.54308n!$ calls to *DERANGE1*.

If we let A_n be the number of times each one of the assignment statements in the **then** clause is executed, then

$$A_n = \tfrac{1}{2}Q_n \cong 0.77154n!\,.$$

As there are sixteen assignments in the body of *DERANGE1*,

(i) four for each of "$x_{m-1} \leftrightarrow x_k$, $x_{m-1} \leftrightarrow x_m$" and "$x_m \leftrightarrow x_{m-1}$, $x_k \leftrightarrow x_{m-1}$";

(ii) three for each of "$y_{m-1} \leftrightarrow y_k$" and "$y_k \leftrightarrow y_{m-1}$";

and (iii) one for each of "$p_{x_m} \leftarrow y_m$" and "$p_{x_{m-1}} \leftarrow y_{m-1}$",

the total number of assignments is $16A_n \cong 12.34464n!$.

Finally, since $D(n) \sim e^{-1}n!$ [4], the number of times the **else** clause in *DERANGE*1 is executed is approximately equal to $0.36788n!$.

3. The second algorithm.

Consider the well known recursive permutation generation procedure *PERM* (m) shown in Figure 3 together with its driver program.

```
for i = 1 to n do p_i ← i
PERM (n)

procedure PERM (m)
  if m ≠ 1 then for j = m to 1 do
    p_m ↔ p_j; PERM (m−1); p_j ↔ p_m
  else output (p_1, p_2, . . . , p_n)
return
```

Fig. 3

Although — as pointed out in [7] — this algorithm is not a very efficient one, it becomes particularly attractive when only a subset of the permutations satisfying some criterion is required. The idea is to prevent whole sequences of permutations from being generated by requiring partial permutations to satisfy some derivative of the criterion. In our case, procedure *PERM* (m) can be modified only slightly to produce derangements. The resulting procedure *DERANGE*2 (m) together with its driver program are shown in Figure 4.

```
for i = 1 to n do p_i ← i
DERANGE2 (n)

procedure DERANGE2 (m)
  if m ≠ 1 then
    if p_m = m then l ← m−1 else l ← m
    for j = l to 1 do
      p_m ↔ p_j; DERANGE2 (m−1); p_j ↔ p_m
  else if p_1 ≠ 1 then output (p_1, p_2, . . . , p_n)
return
```

Fig. 4

The algorithm works as follows. When *DERANGE*2 (m) is called all the elements in positions p_{m+1} to p_n are already deranged. If $p_m \neq m$ then we proceed in two stages. In the first stage the elements in positions p_1 to p_{m-1} are deranged;

in the second stage the elements in p_1 to p_m are deranged by exchanging the element in p_m with that in p_j, $1 \leqq j \leqq m-1$, and then deranging the elements in p_1 to p_{m-1} as before. If, on the other hand, $p_m = m$, then only the second stage just described is applied. It is obvious that when $m=1$, the array is output as a derangement only if $p_1 \neq 1$.

To analyse *DERANGE2* let us define $d_{n,k}$ as the number of times procedure *DERANGE2* (k) is called from *DERANGE2* $(k+1)$ in the process of generating all derangements of $\{1,2,\ldots,n\}$. Clearly,

$$d_{n,k} = (n-1)d_{n-1,k} + (n-k-1)d_{n-2,k} \quad \text{for } 1 \leqq k \leqq n-1 . \tag{3}$$

As *DERANGE2* (n) is called precisely once from the driver program we let $d_{n,n}=1$. Also $d_{n,k}=0$ for $k>n$. It follows that the total number of calls to procedure *DERANGE2* can be written as

$$T_n = \sum_{k=1}^{n} d_{n,k} . \tag{4}$$

Solving the recurrence in (3) we get [6]

$$d_{n,k} = \frac{1}{k!} \sum_{r=0}^{n-k} (-1)^r \binom{n-k}{r} (n-r)! \quad \text{for } 1 \leqq k \leqq n . \tag{5}$$

Substitution of (5) in (4) yields

$$\begin{aligned} T_n &= \sum_{k=1}^{n} \sum_{r=0}^{n-k} \frac{1}{k!} (-1)^r \binom{n-k}{r} (n-r)! \\ &\sim (1-e^{-1})n! \cong 0.63212n! . \end{aligned}$$

As there are seven assignments in the body of the procedure

(i) one for assigning a value to l;
and (ii) three for each of "$p_m \leftrightarrow p_j$" and "$p_j \leftrightarrow p_m$",

Table 1.

n	Q_n	T_n
1	0	1
2	2	2
3	8	6
4	36	22
5	184	103
6	1110	588
7	7776	3970
8	62216	30908
9	559952	272349
10	5599530	2677750

the total number of assignments is $7T_n \cong 4.42484n!$. As before the output statement is executed approximately $0.36788n!$ times.

Table 1 shows the values of Q_n and T_n for $n \leqq 10$. It is clear that, even for small values of n, *DERANGE2* is superior to *DERANGE1*.

4. Conclusions.

To the best of this writer's knowledge only two publications have previously dealt with the problem of generating derangements. The two algorithms described therein, *DERANGE* (*n*) [1] and *DERANGE1* (*n*) [5], are based on a well known recurrence formula which computes the number of derangements of n objects. The new algorithm presented in this paper, *DERANGE2* (*n*), uses a totally different approach. A recursive permutation generation procedure is modified to produce derangements. Besides its novelty and simplicity, *DERANGE2* enjoys the important advantage of being faster than its predecessors. It is also interesting to point out in conclusion that, unlike the two procedures of [1] and [5], *DERANGE2* does not use any arrays in addition to the array *P*.

REFERENCES

1. S. G. Akl, *An Algorithm for generating derangements*, Technical Report No. 78-71, DOCIS, Queen's University, Kingston, Ontario, Canada.
2. J. R. Bitner, G. Ehrlich and E. M. Reingold, *Efficient generation of the binary reflected Gray code and its applications*, CACM, Vol. 19, No. 9 (1976), 517–521.
3. G. Ehrlich, *Loopless algorithms for generating permutations, combinations and other combinatorial configurations*, JACM, Vol. 20, No. 3 (1973), 500–513.
4. S. Even, *Algorithmic Combinatorics*, Macmillan, New York, 1973, pp. 55–56.
5. E. M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms, Solutions Manual*, Prentice–Hall, Englewood Cliffs, New Jersey, 1978, pp. 111–119.
6. J. Riordan, *An Introduction to Combinatorial Analysis*, John Wiley, New York, 1958, p. 188.
7. R. Sedgewick, *Permutation generation methods*, Computing Surveys, Vol. 9, No. 2 (1977), 137–164.
8. S. Zacks, and D. Richards, *Generating trees and other combinatorial objects lexicographically*, SIAM J. Comput., Vol. 8, No. 1 (1979), 73–81.

DEPARTMENT OF COMPUTING AND INFORMATION SCIENCE
QUEEN'S UNIVERSITY
KINGSTON, ONTARIO
CANADA