Discrete Applied Mathematics 217 (2017) 722–728

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Note

Linear-time generation of uniform random derangements encoded in cycle notation

Kenji Mikawa [a,*], Ken Tanaka [b]

[a] Center for Academic Information Service, Niigata University, 8050 Ikarashi 2-no-cho, Nishi-ku, Niigata 950-2181, Japan

[b] Faculty of Science, Kanagawa University, 2946 Tsuchiya, Hiratsuka, Kanagawa 259-1293, Japan

ARTICLE INFO

Article history:
Received 24 September 2014
Received in revised form 27 May 2015
Accepted 1 October 2016
Available online 22 October 2016

Keywords:
Combinatorial problem
Algorithm
Random generation
Linear-time
Derangement

ABSTRACT

We present a linear-time algorithm for generating random derangements. Several algorithms for generating random derangements have recently been published. They are enhancements of the well-known Fisher–Yates shuffle for random permutations, and use the rejection method. The algorithms sequentially generate random permutations, and therefore pick only derangements by omitting the other permutations. A probabilistic analysis has shown that these algorithms could be run in an amortized linear-time. Our algorithm is the first to achieve an exact linear-time generation of random derangements. We use a computational model such that arithmetic operations and random access for $O(\log n!) = O(n \log n)$ bits can be executed in constant time.

1. Introduction

A derangement over n integers $[n] = \{1, 2, \ldots, n\}$ is defined as a permutation $\delta = \delta(1)\delta(2)\ldots\delta(n)$ of the integers, without fixed points, i.e., $\delta(i) \neq i$ for all $i \in [n]$. The subfactorial $!n$ is the number of possible derangements. It is given by the recurrence relation

$$!n = (n-1)\,(!(n-1)+!(n-2))\,, \tag{1}$$

with the initial conditions $!0 = 1$ and $!1 = 0$. We can derive a recurrence relation from Eq. (1), that is,

$$!n = n\times!(n-1) + (-1)^n. \tag{2}$$

The subfactorial also satisfies $!n = \lfloor (n! + 1)/e \rfloor$, where e is Napier's constant.

Durstenfeld proposed an excellent implementation of the Fisher–Yates shuffle, which is one of the best-known algorithms for generating random permutations [1]. Whereas a naive implementation of the Fisher–Yates shuffle takes $O(n^2)$ time to generate one random permutation, Durstenfeld's implementation realizes $O(n)$ time to generate one random permutation. The stringent condition that derangements have no fixed points complicates linear-time algorithms for generating random derangements, in contrast with linear-time algorithms for generating random permutations. Existing algorithms have enhanced the Fisher–Yates shuffle to suit this problem, using the rejection method to sequentially generate random permutations, thus picking up only derangements by omitting other permutations [3,2]. The probability that a random permutation is a derangement is expected to be close to $1/e$. In fact, a probabilistic analysis has shown that both the proposed algorithms [3,2] should run in an amortized linear-time.

* Corresponding author.
E-mail addresses: mikawa@cais.niigata-u.ac.jp (K. Mikawa), ktanaka@info.kanagawa-u.ac.jp (K. Tanaka).

http://dx.doi.org/10.1016/j.dam.2016.10.001
0166-218X/

In our recent report [4], we proposed a lexicographic ranking and unranking of derangements encoded in cycle notation. The unranking algorithm takes $O(n \log n)$ time to generate one derangement for a given random integer. The time complexity is estimated on the traditional computational model, such that arithmetic operations and random access with $O(\log x)$ bits for input size x can be done in $O(1)$ time. Our algorithm requires arithmetic on large integers to calculate a probability of belonging to two classes of derangements. The large integers in the algorithm are at most $!n = \lfloor (n! + 1)/e \rfloor$. We use a computational model such that arithmetic operations and random access with $O(\log !n) = O(n \log n)$ bits can be done in $O(1)$ time. Hence, we achieve a linear-time generation of uniform random derangements under this model.

2. Uniformly random derangement generation

Cycle notation is an intuitive way to describe the order of a permutation that gives a mapping from $[n]$ to $[n]$ as a list of disjoint cycles. Stanley introduced the *standard representation* for this notation, and described some of its properties in [5]. In our proposed algorithm, we deal with permutations which are written in standard representation with *right-to-left* minima. For example, given the permutation $\pi = 5743126$, the standard representation of π is $\sigma = 5176243$. For derangements, we observe the property that there are no two consecutive right-to-left minima and the first value cannot be 1. The linear-time complexity of converting between a permutation and its standard representation is also well known.

We explain the Fisher–Yates shuffle introduced by Durstenfeld in [1]. The Fisher–Yates shuffle for $\pi = \pi(1)\pi(2)\ldots\pi(n)$ on $[n]$ is shown below. Initially, $\pi = 12\ldots n$. The function rand(i) outputs a uniform random integer between 1 and i.

for $i := 1$ **downto** $n - 1$ **do begin**
  $j :=$ rand$(n - i + 1) + i - 1$;
  swap$(\pi(i), \pi(j))$;
end;

Once $\pi(i)$ is fixed, the algorithm will never alter the prefix $\pi(1)\ldots\pi(i)$ in subsequent steps.

Our approach is to specialize the Fisher–Yates shuffle as applied to random derangements encoded in cycle notation. A candidate element for the position $\sigma(i)$ exists in the remaining elements $\{\sigma(i), \ldots, \sigma(n)\}$, but the candidates do not have equal probabilities $1/(n - i + 1)$. When a cycle in the prefix $\sigma(1)\ldots\sigma(i-1)$ closes at $\sigma(i)$ (which implies that $i > 1$ and no cycles close at $\sigma(i-1)$), the minimum element must be uniquely selected from $\{\sigma(i), \ldots, \sigma(n)\}$. Otherwise, when a cycle does not close at $\sigma(i)$, an element can be selected from the $n - i$ possible elements in $\{\sigma(i), \ldots, \sigma(n)\}$, except for the minimum. The $n-i$ possible elements have an equal probability of $1/(n-i)$. Thus, the probability of selecting the minimum element for the position $\sigma(i)$ is given by

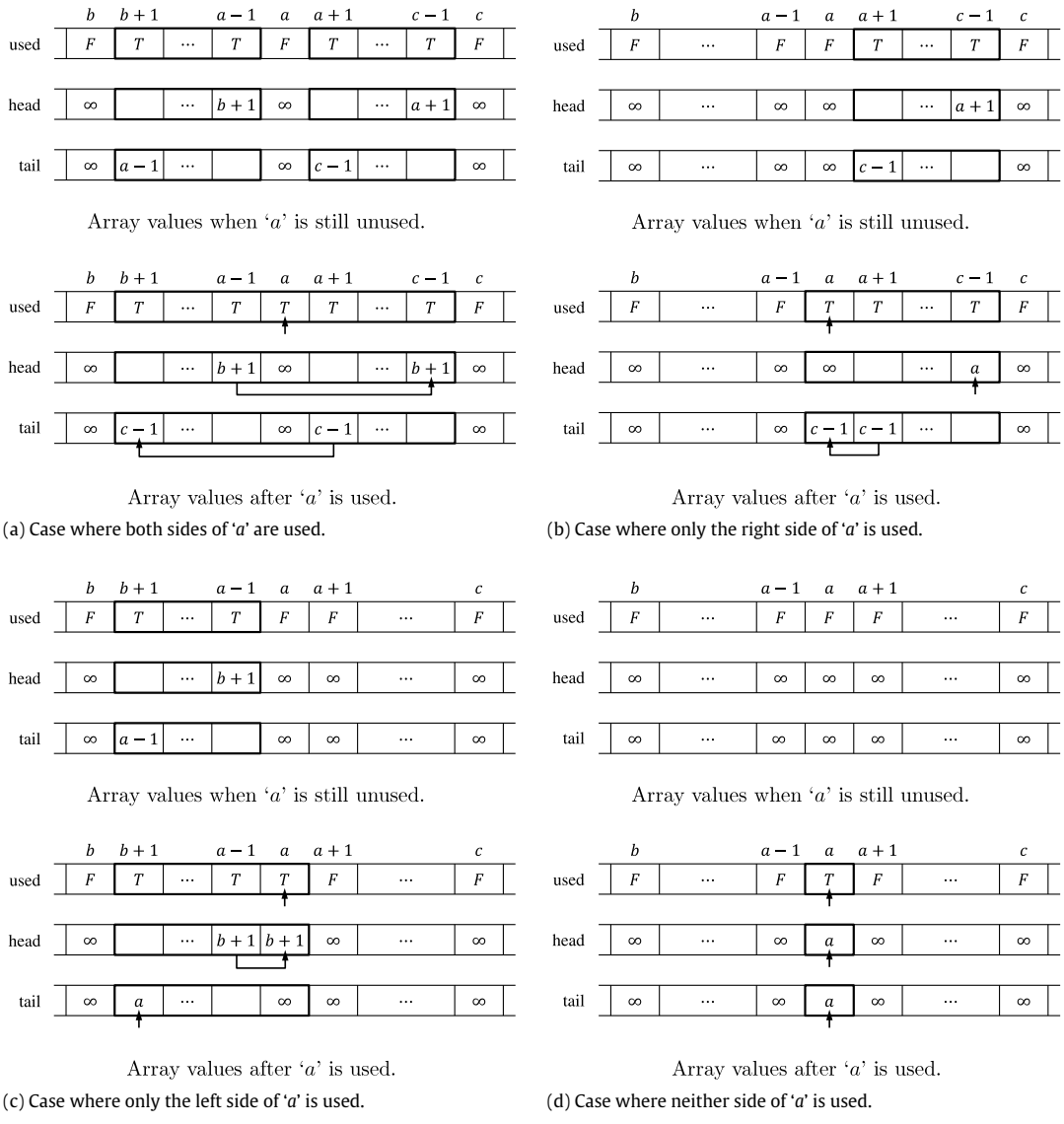
$$P(A_i) = \frac{!(n-i)}{!(n-i)+!(n-i+1)}, \tag{3}$$

under the conditions that a cycle does not close at $\sigma(i-1)$, where $i > 1$. The outline of our algorithm is shown below. We use a Boolean flag, *closed*, to determine if a cycle closes at the previous step. The function $\sigma^{-1}(x)$ returns the position of a given element x, and the function $\min(X)$ returns the minimum element in a given set X. Initially, $\sigma = 12\ldots n$.

```
A 1: closed := true;
A 2: for i := 1 to n − 1 do begin
A 3:   if (i > 1) and (closed = false) and
           rand(!(n − i)+!(n − i + 1)) ≤ !(n − i) then begin
       // some cycle closes at σ(i).
A 4:   j := σ^{-1}(min({σ(i), σ(i + 1), ..., σ(n)}));
A 5:   swap(σ(i), σ(j));
A 6:   closed := true;
A 7:   end else begin
       // otherwise, a cycle does not close at σ(i).
A 8:   j := rand(n − i) + i − 1;
A 9:   if σ(j) = min({σ(i), σ(i + 1), ..., σ(n)})
           then swap(σ(i), σ(n)) else swap(σ(i), σ(j));
A10:   closed := false;
A11:   end;
A12: end;
```

The algorithm contains a tricky swap process in lines A8 and A9. This process corresponds to selecting a non-minimum element among the $n - i$ possible elements in $\{\sigma(i), \ldots, \sigma(n)\}$. It is equivalent to selecting an element from the set $\{\sigma(i), \ldots, \sigma(n)\} \setminus \{\min(\{\sigma(i), \ldots, \sigma(n)\})\}$. To achieve a linear-time generation, the time taken to calculate $\min(X)$ and the probability of selecting the minimum element for the position $\sigma(i)$ must be constant for each loop.

Our strategy to find the minimum element of a given set is based on a linked list using three arrays: used[i], which stores a Boolean value indicating whether an element $i \in [n]$ is used or not, head[i], which is a pointer to the head of successive used



(a) Case where both sides of ‘a’ are used.
(b) Case where only the right side of ‘a’ is used.
(c) Case where only the left side of ‘a’ is used.
(d) Case where neither side of ‘a’ is used.

Fig. 1. Data structure.

elements containing i, and tail[i], which is a pointer to the tail of successive used elements containing i. They are initialized to used[i] = **false**, head[i] = ∞, and tail[i] = ∞, for all i. Note that it is important that both head[i] and tail[i] are almost incomplete. This means that head[i] only works if i is located at the tail of successive used elements. In other words, it does not matter if head[i] is correct at any other positions of i in the successive used elements. Similarly, tail[i] only works if i is located at the head of successive used elements.

We now explain how to update the array values when an element $a \in [n]$ is determined for the position $\sigma(i)$. There are four cases to consider with respect to the position of a, as shown in Fig. 1. The first case is when both adjacent sides of a have been used, as shown in Fig. 1(a). In this case, both the adjacent runs of a are concatenated into a single run. That is,

$$\begin{aligned} \text{used}[a] &\leftarrow \textbf{true}, \\ \text{head}[\,\text{tail}[a+1]\,] &\leftarrow \text{head}[a-1], \\ \text{tail}[\,\text{head}[a-1]\,] &\leftarrow \text{tail}[a+1]. \end{aligned}$$

In the example in Fig. 1(a), we find that the pointers head[$c-1$] and tail[$b+1$] at both ends of the single run were correctly updated to proper elements. The second case is when only the right adjacent side of a is used, as shown in Fig. 1(b). In this case, the a is added to its right adjacent run. That is,

$$\begin{aligned} \text{used}[a] &\leftarrow \textbf{true}, \\ \text{head}[\,\text{tail}[a+1]\,] &\leftarrow a, \\ \text{tail}[a] &\leftarrow \text{tail}[a+1]. \end{aligned}$$

The third case is the opposite of the second case, when only the left adjacent side of a is used, as shown in Fig. 1(c). In this case, the element a is added to its left adjacent run. That is,

$$\begin{aligned} \text{used}[a] &\leftarrow \textbf{true}, \\ \text{head}[a] &\leftarrow \text{head}[a-1], \\ \text{tail}[\,\text{head}[a-1]\,] &\leftarrow a. \end{aligned}$$

The final case is when both adjacent sides of a are unused, as shown in Fig. 1(d). In this case, only the used element a exists around itself, and both pointers must be updated to a. That is,

$$\begin{aligned} \text{used}[a] &\leftarrow \textbf{true}, \\ \text{head}[a] &\leftarrow a, \\ \text{tail}[a] &\leftarrow a. \end{aligned}$$

Lemma 1. *Let $U \subseteq [n]$ be the set of used elements, and the arrays* used[i], head[i], *and* tail[i] *be maintained for all $i \in U$. Then,*

$$\min([n] \setminus U) = \begin{cases} 1 & \text{if used}[1] = \textbf{false}, \\ \text{tail}[1] + 1 & \text{if used}[1] = \textbf{true}. \end{cases}$$

Proof. From the definition, if the value of used[1] is **false**, the element 1 is still unused (i.e., $1 \in [n] \setminus U$). Thus, $\min([n] \setminus U) = 1$. If the element 1 has already been used, the element pointed by tail[1] is the rightmost element of the continuous succession of used elements that begins with the first element 1. Thus, the element pointed by tail[1] + 1 is the leftmost unused element. □

The update procedure for used[i], head[i], and tail[i] should run after some element for the position $\sigma(i)$ is determined. Hence, the procedure is embedded just after lines A5 and A9. The procedure is given by

```
B 1: procedure update(var a: integer);
B 2: begin
B 3:   used[a] := true;
B 4:   if used[a − 1] = true and used[a + 1] = true then begin
B 5:     head[tail[a + 1]] := head[a − 1];
B 6:     tail[head[a − 1]] := tail[a + 1];
B 7:   end if used[i − 1] = false and used[a + 1] = true then begin
B 8:     head[tail[a + 1]] = a;
B 9:     tail[a] = tail[a + 1];
B10:   end if used[i − 1] = true and used[a + 1] = false then begin
B11:     head[a] = head[a − 1];
B12:     tail[head[a − 1]] = a;
B13:   end if used[i − 1] = false and used[a + 1] = false then begin
B14:     head[a] = a;
B15:     tail[a] = a;
B16: end;
```

Since the procedure above is called once a loop by our algorithm, the computational complexity of the proposed algorithm to generate one random derangement remains still a linear-time.

As mentioned above, the proposed algorithm uses at most two subfactorials for a loop, since the probability of selecting the minimum element for each position of a derangement is given by Eq. (3). Using Eq. (2), the value of $!(n-i)$ can be calculated using a few constant arithmetic operations from the previous subfactorial:

$$!(n-i) = \frac{!(n-i+1) - (-1)^{n-i+1}}{n-i+1}.$$

The value of $!(n-i+1)$ can be derived from $!(n-i+2)$, which was calculated in the previous loop. Thus, when the value of $!n$ has been stored into somewhere on working memory before the beginning of the proposed algorithm, the subfactorial required on each loop can be obtained sequentially from the subfactorial on the previous loop.

3. Theoretical evaluation of uniformity

In this section, we show that the proposed algorithm achieves a uniform random generation of derangements over $[n]$. That is, the occurrence probability of generated derangements is equal to $1/!n$. Let $P(B_i)$ be the probability of selecting a particular element among all possible values for position $\sigma(i)$. The first element $\sigma(1)$ is always selected from $[n] \setminus \{1\}$. Thus, $P(B_1) = 1/(n-1)$. The elements $\sigma(n-1)$ and $\sigma(n)$ are uniquely selected from the largest and smallest of the last two elements, respectively. Thus, $P(B_{n-1}) = P(B_n) = 1$. For $1 < i < n-1$, $P(B_i)$ depends on a combination of $\sigma(i)$ and $\sigma(i-1)$. Since any two minima are not adjacent in σ, there are three cases to consider: $\sigma(i)$ is a minimum, $\sigma(i-1)$ is a minimum, and neither $\sigma(i)$ nor $\sigma(i-1)$ is a minimum. In the first case, when $\sigma(i)$ is the minimum of $\{\sigma(i), \ldots, \sigma(n)\}$, $P(B_i)$ is given

by Eq. (3). In the second case, when $\sigma(i-1)$ is the minimum of $\{\sigma(i-1), \sigma(i), \ldots, \sigma(n)\}$, we have $P(B_i) = 1/(n-i)$, since $\sigma(i)$ must be selected from the $n-i$ possible elements in $\{\sigma(i), \ldots, \sigma(n)\}$, except for the minimum. The last case is similar to the second case, but $\sigma(i)$ just happens to be a non-minimum element in spite of the possibility of being the minimum. The probability of not selecting the minimum for the position $\sigma(i)$ is $1 - P(A_i)$ and the probability of selecting a non-minimum element from the $n-i$ possible elements is $1/(n-i)$. Thus, the compound probability is estimated as $P(B_i) = (1 - P(A_i)) \times (1/(n-i))$. To summarize the discussion above, for $n \geq 2$, we have

$$P(B_i) = \begin{cases} \dfrac{1}{n-i} & \text{if } i = 1 \text{ or } \sigma(i-1) = \min(\{\sigma(i-1), \sigma(i), \ldots, \sigma(n)\}), \\ \dfrac{!(n-i+1)}{!(n-i)+!(n-i+1)} \times \dfrac{1}{n-i} & \text{if } \sigma(i) \neq \min(\{\sigma(i), \ldots, \sigma(n)\}) \text{ and} \\ & \sigma(i-1) \neq \min(\{\sigma(i-1), \sigma(i), \ldots, \sigma(n)\}), \\ \dfrac{!(n-i)}{!(n-i)+!(n-i+1)} & \text{if } \sigma(i) = \min(\{\sigma(i), \ldots, \sigma(n)\}). \end{cases} \tag{4}$$

Eq. (4) includes boundary conditions $P(B_1) = 1/(n-1), P(B_{n-1}) = 1$ because $\sigma(n-1) \neq \min(\{\sigma(n-1), \sigma(n)\})$, and $P(B_n) = 1$ because $\sigma(n) = \min(\{\sigma(n)\})$.

Let $P(B)$ denote the occurrence probability for derangements generated by the proposed algorithm. It can be calculated using

$$P(B) = P(B_1) \times P(B_2) \times \cdots P(B_n).$$

For example, the occurrence probabilities of $\sigma = 5176243$ and $\sigma = 2345671$ are

$$\frac{1}{6} \times \left(\frac{!5}{!5+!6}\right) \times \frac{1}{4} \times \left(\frac{!4}{!3+!4} \times \frac{1}{3}\right) \times \left(\frac{!2}{!2+!3}\right) \times 1 \times 1 = \frac{1}{!7}$$

and

$$\frac{1}{6} \times \left(\frac{!6}{!5+!6} \times \frac{1}{5}\right) \times \left(\frac{!5}{!4+!5} \times \frac{1}{4}\right) \times \left(\frac{!4}{!3+!4} \times \frac{1}{3}\right) \times \left(\frac{!3}{!2+!3} \times \frac{1}{2}\right) \times 1 \times 1 = \frac{1}{!7},$$

respectively. Now, we consider a k-cycles derangement $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$, where σ_i consists of m_i elements. The compound of the probabilities of selecting elements in σ_i, denoted by $P(C_i)$, is

$$P(C_i) = P(B_{t_i+1}) \times P(B_{t_i+2}) \times \cdots \times P(B_{t_i+m_i}), \tag{5}$$

where $t_i = \sum_{l=1}^{i-1} m_l$ for $i > 1$ and $t_1 = 0$. We then have the following lemma.

Lemma 2. *Let $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$ be a derangement with k cycles, and let σ_i contain m_i elements. The compound probability, $P(C_i)$ of selecting elements in σ_i is reduced to*

$$P(C_i) = \frac{!\left(n - \sum_{l=1}^{i} m_l\right)}{!\left(n - \sum_{l=1}^{i-1} m_l\right)}$$

where $\sum_{l=1}^{0} m_l = 0$.

Proof. From Eqs. (4) and (5), $P(C_i)$ can be reduced to

$$\begin{aligned} P(C_i) = {} & \frac{1}{n-1-t_i} \times \frac{!(n-1-t_i)}{!(n-2-t_i)+!(n-1-t_i)} \times \frac{1}{n-2-t_i} \times \frac{!(n-2-t_i)}{!(n-3-t_i)+!(n-2-t_i)} \times \frac{1}{n-3-t_i} \\ & \vdots \\ & \times \frac{!(n-m_i+2-t_i)}{!(n-m_i+1-t_i)+!(n-m_i+2-t_i)} \times \frac{1}{n-m_i+1-t_i} \times \frac{!(n-m_1-t_i)}{!(n-m_i-t_i)+!(n-m_i+1-t_i)} \\ = {} & \frac{!(n-m_i-t_i)}{!(n-t_i)} \\ = {} & \frac{!(n-t_{i+1})}{!(n-t_i)} \end{aligned}$$

where $t_i = \sum_{l=1}^{i-1} m_l$ for $i > 1$ and $t_1 = 0$. □

Thus, we obtain the following theorem on the uniformity of the distribution of random derangements generated by the proposed algorithm.

Theorem 3. *The occurrence probabilities of derangements over* $[n]$ *generated by the proposed algorithm are equal to* $1/!n$.

Proof. Let $\sigma = \sigma_1\sigma_2 \dots \sigma_k$ be a k-cycle derangement generated by the proposed algorithm, and let the ith cycle contain m_i elements. $P(B)$ is calculated by $P(B) = P(C_1) \times P(C_2) \times \cdots \times P(C_k)$. Then,

$$\begin{aligned} P(B) &= \frac{!(n-t_2)}{!n} \times \frac{!(n-t_3)}{!(n-t_2)} \times \cdots \times \frac{!(n-t_k)}{!(n-t_{k-1})} \times \frac{!(n-t_{k+1})}{!(n-t_k)} \\ &= \frac{1}{!n}, \end{aligned}$$

where $t_i = \sum_{l=1}^{i-1} m_l$ for $i > 1$, and $t_1 = 0$. □

4. Conclusion

In this paper, we proposed an algorithm for generating random derangements based on the Fisher–Yates shuffle. Our method is the first to achieve an exact linear-time generation. Whereas the Fisher–Yates shuffle never requires arithmetic on large integers, our algorithm must calculate the value of $!n$ before running the main loop. Future work should attempt to remove the necessity of pre-computing the value of $!n$. A complete implementation of our algorithm, written in C, is shown in Appendix.

Acknowledgments

We would like to thank the reviewers for their detailed comments and helpful suggestions. This work was supported in part by a Grant-in-Aid for Scientific Research (C) 15K00182 from the Japan Society for the Promotion of Science.

Appendix. Complete implementation of the proposed algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int subfactorial(int);
void swap(int, int);
int getmin(int);
void update(int);
int p[100], q[100], d[100], used[100], head[100], tail[100];
int main(int argc, char *argv[]) {
  int n, min, d1, d2, i, j, closed = 1;
  if (argc != 2) exit(0);
  n = atoi(argv[1]); srand(time(NULL));
  // initialized
  d1 = subfactorial(n);
  if (n%2 != 0) d2 = (d1 + 1) / n; else d2 = (d1 - 1) / n;
  for (j = 1; j <= n; j++) {
    p[j] = j; q[j] = j; d[j] = 0;
    used[j] = 0; head[j] = 0; tail[j] = 0;
  }
  // generating random derangement in cycle notation
  if (n > 1) {
    for (i = 1; i < n; i++) {
      if ((i > 1)&&(closed == 0)&&((rand()%(d2 + d1)) < d2)) {
        j = q[getmin(i)];
        swap(i, j); update(p[i]); closed = 1;
      }
      else {
        j = rand()%(n - i);
        if (p[i + j] == getmin(j)) swap(i, n); else swap(i, i + j);
        update(p[i]); closed = 0;
      }
      if (i != n - 1) {
        d1 = d2;
```

```
        if ((n - i)%2 != 0) d2 = (d1 + 1)/(n - i);
        else d2 = (d1 - 1)/(n - i);
      }
    }
  }
  // translation of cycle notation to derangement and output
  p[0] = 0; min = p[n];
  for (j = n; j >= 1; j--)
    if (p[j - 1] < min) {
      d[min] = p[j]; min = p[j - 1];
    }
    else {
      d[p[j - 1]] = p[j];
    }
  for (j = 1; j <= n; j++) fprintf(stdout, "%d ", p[j]);
  fprintf(stdout, "\n");
  exit(1);
}
int subfactorial(int n) {
  int i, s = 1;
  if (n == 1) return(0); if (n == 2) return(1);
  for (i = 3; i <= n; i++) {
    if (i % 2 != 0) s = s * i - 1; else s = s * i + 1;
  }
  return(s);
}
void swap(int i, int j) {
  int k;
  q[p[i]] = j; q[p[j]] = i; k = p[i]; p[i] = p[j]; p[j] = k;
}
int getmin(int n) {
  if (used[1] == 1) return(tail[1] + 1); else return(1);
}
void update(int a) {
  used[a] = 1;
  if ((used[a - 1] == 1)&&(used[a + 1] == 1)) {
    head[tail[a + 1]] = head[a - 1];
    tail[head[a - 1]] = tail[a + 1];
  }
  if ((used[a - 1] == 0)&&(used[a + 1] == 1)) {
    head[tail[a + 1]] = a;
    tail[a] = tail[a + 1];
  }
  if ((used[a - 1] == 1)&&(used[a + 1] == 0)) {
    head[a] = head[a - 1];
    tail[head[a - 1]] = a;
  }
  if ((used[a - 1] == 0)&&(used[a + 1] == 0)) {
    head[a] = a;
    tail[a] = a;
  }
}
```

References

[1] R. Durstenfeld, Algorithm 235: Random permutation, Commun. ACM 7 (7) (1964) 420.
[2] C. Martínez, A. Panholzer, H. Prodinger, Generating random derangements, in: Proc. ACM-SIAM Workshop on Analytic Algorithms and Combinatorics, 2008, pp. 234–240.
[3] D. Merlini, R. Sprugnoli, M.C. Verri, An analysis of a simple algorithm for random derangements, in: Proc. 10th Italian Conf. Theoretical Computer Science, 2007, pp. 139–150.
[4] K. Mikawa, K. Tanaka, Lexicographic ranking and unranking of derangements in cycle notation, Discrete Appl. Math. 166 (2014) 164–169.
[5] R.P. Stanley, Enumerative Combinatorics, Vol. 1, second ed., Cambridge University Press, Cambridge, 2012.