**Thai J**ournal of **Math**ematics
Volume 21 Number 4 (2023)
Pages 899–915

http://thaijmath.in.cmu.ac.th

ISSN 1686-0209

Discrete and Computational Geometry, Graphs, and Games

# Uniformly Generating Derangements with Fixed Number of Cycles in Polynomial Time

**Nattawut Phetmak and Jittat Fakcharoenphol***

*Department of Computer Engineering, Kasetsart University, 50 Ngamwongwan Rd., Bangkok*
*e-mail : nattawut.p@ku.th (N. Phetmak); jittat@gmail.com (J. Fakcharoenphol)*

**Abstract** We study the uniform sampling of permutations without fixed points, i.e., derangements, that can be decomposed into $m$ disjoint cycles. Since the number of cycles in a random derangement tends towards the standard distribution, rejection sampling may take exponential time when $m$ largely deviates from the mean of $\Theta(\log n)$. We propose an algorithm for generating a uniformly random derangement of $n$ items with $m$ cycles in $O(n^{2.5} \log n)$ time complexity using dynamic programming with an assumption that all arithmetic operations can be done in time $O(1)$. Taking into account the arithmetic operations on large integers, the running time becomes $O(n^{3.5} \log^3 n)$. Our algorithm uses permutation types to structure our uniform generation of derangements.

**MSC:** 05A05; 68R05; 90C39; 11B73

**Keywords:** derangement; random generation; dynamic programming; polynomial time algorithm

Submission date: 01.02.2022 / Acceptance date: 14.05.2023

## 1. Introduction

A *fixed point* in a permutation is an element that maps to itself. A *derangement* is a permutation without fixed points. The study of derangements can be traced back to 1708 when Pierre Rémond de Montmort proposed a problem for counting distinct derangements given $n$ unique items. Together with Nicolaus I Bernoulli, they solved the problem half a decade after [1]. Using the inclusion-exclusion principle, they showed that the number of derangements of $n$ items, in its modern form, is $d(n) = n! \sum_{i=0}^{n} \frac{(-1)^i}{i!}$.

Naturally, the problem of generating derangements follows. Since the probability that a random permutation is a derangement is $d(n)/n! \approx 1/e$, a simple rejection sampling technique with a standard algorithm for generating random permutations works with an expectation of $O(1)$ trials until a derangement is generated. Recently, there have been efforts to efficiently generate a random derangement with smaller numbers of trials [2–4].

A permutation can be decomposed into disjoint cycles by tracing the result of applying the permutation repeatedly on each element. A cycle of length $k$ is referred to as a $k$-cycle.

*Corresponding author.

Published by The Mathematical Association of Thailand.

From this definition, a derangement is a permutation with no 1-cycles. A permutation with only one cycle is called a *cyclic permutation*. When the number of elements is larger than one, a cyclic permutation is also a derangement.

In this paper, we are interested in generating derangements with exactly $m$ cycles. When $m = 1$, i.e., the goal is to generate a cyclic permutation; in this case, Sattolo [5] gave an algorithm that runs in $O(n)$ time. For other values of $m$, since the number of cycles obeys Gaussian distributions [6], rejection sampling also works when $m$ is close to the expectation, which is $\Theta(\log n)$. However, when $m$ deviates badly from the expectation, the number of feasible derangements can be tiny, and it may take exponentially many trials to get the desired number of cycles. For example, when $n$ is even and $m = n/2$, there are only

$$\frac{1}{(n/2)!}\binom{n}{2}\binom{n-2}{2}\cdots\binom{2}{2} = \frac{1}{(n/2)!}\binom{n}{2,2,\cdots,2} = \frac{n!}{(n/2)!\cdot 2^{(n/2)}}$$

derangements. The probability of successfully obtaining one from a random derangement is only $O(1/\sqrt{2^n})$.

In this paper, we propose algorithms for uniformly generating a derangement of $n$ items with $m$ cycles in time $O(n^{2.5}\log n)$, assuming that all arithmetic operations can be done in $O(1)$ time. When accounting for arithmetic operations of large integers, the algorithm runs in time $O(n^{3.5}\log^3 n)$.

For a set of possible permutations $\mathcal{P}$, a *ranking function* for $\mathcal{P}$ is a bijection from $\mathcal{P}$ to $\{0, 1, 2, \ldots, |\mathcal{P}|-1\}$, and an *unranking function* is its inverse. Notable algorithms for ranking and unranking functions are Myrvold–Ruskey on permutations [7] and Mikawa–Tanaka on derangements [8]. This paper focuses only on devising an unranking function and uses it as a core routine to our algorithms.

While the Stirling number of the first kind [9, 10] gives the number of permutations with $m$ cycles, it does not give enough structure for reconstructing the $i$-th derangement with $m$ cycles. Our key ingredient is a permutation type $\mathbf{t}$ (see Section 6.2 in [9]) that lists the number of cycles $\mathbf{t}(k)$ for every cycle length $k$ in a permutation. Our algorithm breaks down the random index $i$ of the derangements into separate index $\mathbf{i}(k)$ for each cycle length $k$. With these details information on $\mathbf{i}$ and $\mathbf{t}$, we can reconstruct the required random derangement with $m$ cycles.

While we work only on uniform generation, permutations, as well as derangements, can be weighted. One notable example of a weighting scheme is Ewens's Sampling Formula [11] in population genetics (see, e.g., the work by da Silva, Jamshidpey, and Tavaré [12]). Our approach can be adapted to deal with weighted derangements if one can sample permutation types correctly under the weighting scheme. Since our algorithm constructs and maintains partial types iteratively, if the weight function can be decomposed nicely with partially known types, we believe the approach here may be useful in that setting.

Section 2 gives definitions and related algorithms. Two extensions of the Stirling number of the first kind that we need are also defined in this section. We present a simpler $O(n^3)$ algorithm in Section 3 based on dynamic programming. Using binary search, we obtain an improved algorithm with an $O(n^{2.5}\log n)$ running time described in Section 4. We discuss the issues with algorithmic implementation with large integers in Section 5, resulting in an overhead of $O(n\log^2 n)$ time, primarily for computing factorials. Finally, we present empirical data on the uniformity and running time of our algorithm in Section 6.

*Preliminary version.* A preliminary version of this paper [13], appeared in TJCDCGGG 2020+1, only had N. Phetmak as a single author and presented an $O(n^3)$-time algorithm. J. Fakcharoenphol, the second author added in this version, contributed to a result in Section 4.

## 2. Preliminaries

In this section, we review definitions and results that we use. We also define two useful extensions on the Stirling numbers of the first kind.

### 2.1. Counting Permutations

A *permutation* is an arrangement of items. It is a bijection of a set onto itself. Let $\pi$ denotes a permutation of set $X$, then $\pi(x)$ denotes an arrangement of an item $x$ for $x \in X$. For convenience, when $x \notin X$, we let $\pi(x) = x$.

To write a permutation, we adopt Cauchy's two-line notation. For example if $\sigma(1) = 1$, $\sigma(2) = 3$, and $\sigma(3) = 2$, we write $\sigma = \left(\begin{smallmatrix}1&2&3\\1&3&2\end{smallmatrix}\right)$.

It is natural to work with a sequence of permutation applications, e.g., applying a permutation $\sigma$ *after* applied a permutation $\pi$. A *product* of a pair of such permutations, denotes as $\sigma{\cdot}\pi$, is a composition of them, i.e., $(\sigma{\cdot}\pi)(x) = \sigma(\pi(x))$. For example, $\left(\begin{smallmatrix}2&4\\4&2\end{smallmatrix}\right)\cdot\left(\begin{smallmatrix}1&2&3\\1&3&2\end{smallmatrix}\right) = \left(\begin{smallmatrix}1&2&3&4\\1&3&4&2\end{smallmatrix}\right)$. Observe that for arbitrary permutations, the product does not commute. However, when both permutations are disjoint, i.e., they move different sets of items, the commutative property holds. We also write a product in a juxtaposition form $\sigma\pi$. The notation $\pi^k$ denotes $\pi$ composed onto itself for $k$ times.

When we apply the same permutation repeatedly, the arrangement will eventually return to the initial arrangement. A *cycle* of a permutation $\pi$ that contains $x$ is a cyclic list $(x, \pi(x), \pi^2(x), \cdots, \pi^{k-1}(x))$, where $k$ is the smallest integer greater than zero such that $x = \pi^k(x)$. We say that such a cycle has length $k$, or it is a *$k$-cycle*. A cycle of length one is called a *fixed point*.

A permutation may consist of more than one cycle. A permutation of $n$ items with only one $n$-cycle is called a *cyclic permutation*, while a permutation that avoids creating any fixed points is called a *derangement*.

Given $n = kv$ distinct items, suppose that we would like to generate a permutation with $v$ cycles each of length $k$. We may start by generating a partition of items into $v$ disjoint sets, each of size $k$. We let $B(n, k)$, be the number of partitions; thus,

$$B(n,k) = \frac{1}{(n/k)!}\binom{n}{k, k, \cdots, k}. \tag{2.1}$$

To see this, note that $\binom{n}{k,k,\cdots,k}$ counts the number of ways one can choose a sequence of $v = n/k$ subsets of size $k$, so we divide it by $(n/k)!$ to obtain $B(n, k)$. We also note a recurrence relation $B(n,k) = \binom{n-1}{k-1}B(n-k,k)$. This sequence is useful in other contexts, and it appears as sequence A060540 in [14].

Our algorithm is structured around permutation types [9]. We say that a permutation is of *type* $\mathbf{t} = [\mathbf{t}(1), \mathbf{t}(2), \ldots, \mathbf{t}(n)]$, iff for $1 \le k \le n$, the permutation contains $\mathbf{t}(k)$ cycles of length $k$. Hence, the number of cycles $m$ in a permutation is $\sum_k \mathbf{t}(k) = m$. It is known (see, e.g.,[9]) that the number of distinct permutations for $n$ items of type $\mathbf{t}$ is

$$\frac{n!}{\prod_k k^{\mathbf{t}(k)}\mathbf{t}(k)!} = \frac{n!}{1^{\mathbf{t}(1)}2^{\mathbf{t}(2)}\cdots n^{\mathbf{t}(n)}\mathbf{t}(1)!\mathbf{t}(2)!\cdots\mathbf{t}(n)!}. \tag{2.2}$$

While permutation types give us insight on how cycles are formed, enumerating all types takes exponential time. The *Stirling number of the first kind* [10], denoted by $s(n,m)$, is the number of distinct permutation of $n$ items with $m$ cycles, which can be defined recursively as

$$s(n,m) = \begin{cases} 1 & \text{if } n=0 \text{ and } m=0, \\ 0 & \text{if } n \le 0 \text{ or } m \le 0, \\ (n-1)s(n-1,m) + s(n-1,m-1) & \text{otherwise.} \end{cases} \tag{2.3}$$

We shall define two related extensions of the Stirling number of the first kind. The first additionally takes the lower bound for the cycle length.

**Definition 2.1.** The *$r$-associated Stirling number of the first kind* [15, Chapter 12], denoted by $s_r(n,m)$, is the number of permutations of $n$ items with $m$ cycles, where each cycle has length at least $r$.

The following lemma states its recurrence.

**Lemma 2.2.**

$$s_r(n,m) = \begin{cases} 1 & \textit{if } n=0 \textit{ and } m=0, \\ 0 & \textit{if } n \le 0 \textit{ or } m \le 0, \\ (n-1)s_r(n-1,m) + \frac{(n-1)!}{(n-r)!}s_r(n-r,m-1) & \textit{otherwise.} \end{cases} \tag{2.4}$$

*Proof.* Similar to (2.3), this lemma can be proved by considering the last case. The first term is when we add a new item to a permutation of size $n-1$ with $m$ cycles each of length at least $r$, resulting in a permutation where item $n$ belongs to a cycle of length greater than $r$. The second term chooses a cycle of length exactly $r$ that contains $n$. ∎

By definition, $s_1(n,m)$ is the original Stirling number of the first kind, while $s_2(n,m)$ only counts derangements with $m$ cycles.

We can compute $s_r(n,m)$ using dynamic programming for a fixed $r$. Assuming that all factorials have been precomputed, since a single entry for $s_r(n,m)$ only looks up $O(1)$ other entries with smaller indices, the running time is $O(mn) \le O(n^2)$

Another extension considers the number of occurrences of the shortest cycles. Note that this *partial-type Stirling number of the first kind* basically gives recurrence for computing the number of derangements based on permutation types.

**Definition 2.3.** The *partial-type Stirling number of the first kind*, denoted by $s_k^v(n,m)$, is the number of permutations of $n$ items with $m$ cycles, where the smallest cycles have length exactly $k$ and appear exactly $v$ times.

The following lemma states the formula for $s_k^v(n,m)$, which is essentially an unpack of the analysis of (2.2).

**Lemma 2.4.**

$$s_k^v(n,m) = \frac{n!}{k^v v!(n-kv)!} s_{k+1}(n-kv, m-v). \tag{2.5}$$

*Proof.* The coefficient $\frac{n!}{k^v v!(n-kv)!}$ counts the number of ways one can choose $v$ cycles of length exactly $k$, and $s_{k+1}(n-kv, m-v)$ counts the number of ways for choosing the rest of the permutation. ∎

Note that when one have already computed $s_{k+1}(n,m)$ and all the necessary factorials, it only takes $O(1)$ to find $s_k^v(n,m)$. With all intermediate values of $s_{k+1}(\cdot,\cdot)$ from the computation of $s_{k+1}(n,m)$, one can also compute a list of $s_k^v(n,m)$ for all $0 \le v \le \lfloor \frac{n}{k} \rfloor$ under the same time limit.

### 2.2. Permutation Algorithms

We introduce 3 subroutines for permutation and partition generation. The first two functions UnrankChoose and UnrankPeriodicChoose are related to partition generation based on $B(n,k)$ shown in (2.1). The third function is fundamentally the cyclic permutation algorithm of Sattolo [5]. All three subroutines are essentially unranking functions that map integers to their corresponding permutations. We remark that these functions are discussed in [16, 17]. Throughout the paper, we use zero-based numbering when referring to the $i$-th item in some ordering and also when referring to items in lists or arrays.

We refer to a subset of size $k$ as a $k$-subset. The first function chooses the $i$-th $k$-subset from a list of $n$ items. Algorithm 1 describes function UnrankChoose$(X,k,i)$ that takes array $X$ with $n$ unique items, the number $k$ of items to choose, and index $i$, and returns the $i$-th $k$-subset $Y$ of $X$ as a list of items. If array $X$ is sorted, subset $Y$ is the $i$-th lexicographically smallest subset. For example, let $X$ be an array $[A,B,C,D,E]$. Choosing $k=3$ items at index $i=0$ yields $[A,B,C]$, choosing at index $i=1$ yields $[A,B,D]$, and choosing at the last index $i=\binom{5}{3}-1$ yields $[C,D,E]$.

**Algorithm 1** The $i$-th $k$-subset from $X$ with $n$ items

```
function UnrankChoose(X, k, i)
    if k = 0 then
        return [ ]
    end if
    x ← X[0]                                   ▷ Let x be the first item
    X' ← X[1, ...]                             ▷ Let X' be the rest of the array
    j ← i − C(n−1, k−1)
    if j ≥ 0 then
        return UnrankChoose(X', k, j)          ▷ Skip x
    end if
    return [ x ] + UnrankChoose(X', k−1, i)    ▷ Choose x
end function
```

Function UnrankChoose$(X,k,i)$ works in $O(n)$ time, since it recurses for at most $O(n)$ times and each call runs in $O(1)$ time. The term $\binom{n-1}{k-1}$ can be computed in $O(1)$ time, provided that all factorials have been precomputed. We remark that this function can be improved to run in time $O(k \log n)$ using binary search.

The next function UnrankPeriodicChoose$(X,k,i)$, shown as algorithm 2, essentially generates the $i$-th partition of $X$ into $v$ subsets of size $k$ using function UnrankChoose. It takes array $X$ with $n$ unique items, the size of subsets $k$, where $n=kv$, and an index $i$ such that $0 \le i < B(n,k)$. It returns the $i$-th partition of $X$ into an unordered list $v$ subsets of size $k$. Again if array $X$ is sorted, the partition returned is the $i$-th lexicographically smallest partition. The function works by repeatedly calling UnrankChoose with appropriate indices.

**Algorithm 2** $i$-th partition of $X$ into $n/k$ sets, each with $k$ items

**function** UnrankPeriodicChoose($X, k, i$)
  **if** $X = \emptyset$ **then**
    **return** [ ]
  **end if**
  $(j, i') \leftarrow \text{DivMod}(i, \binom{n-1}{k-1})$ ▷ $(q, r) = \text{DivMod}(a, b) \iff a = qb + r$
  $Y \leftarrow \text{UnrankChoose}(X, k, i')$
  **return** [ $Y$ ] + UnrankPeriodicChoose($X \backslash Y, k, j$)
**end function**

Function UnrankPeriodicChoose runs in $O(n^2/k)$ time, because there are $O(n/k)$ recursive steps and each step runs in time $O(n)$. However, if we use the $O(k \log n)$ implementation of UnrankChoose, this function may takes $O(n \log n)$ running time instead. Note that since items in $X$ and $Y$ are stored in ascending order, finding the array $X \setminus Y$ takes $O(n)$ time.

In each recursive call to UnrankPeriodicChoose, it chooses one $k$-subset from $X$. Recalled the relation for $B(n, k)$ from (2.1), there are $\binom{n-1}{k-1}$ ways to partition $k$-subset in this level, leaving $n-k$ items with $B(n-k, k)$ ways to generate the rest of the partition. Therefore to generate the $i$-th partition, we write $i$ as $\binom{n-1}{k-1} \cdot j + i'$ and find the $i'$-th subset of size $k$ at this level and recursively find the $j$-th partition of the rest of the array. This is essentially the implementation of the recursive form of $B(n, k)$ in (2.1). Note that index $i'$ ensures that UnrankChoose would choose the first item of $X$; thus, the subsets in the partition generated is unordered as claimed.

The last subroutine generates the $i$-th cyclic permutation based on Sattolo's random cyclic permutation algorithm [5]. Function UnrankCyclicPermute($X, i$) in algorithm 3, takes array $X$ of $n$ unique items and an index $i$. It permutes the given items into the $i$-th cyclic permutation. Unlike previous subroutines, this function does not preserve lexicographical order property. This is a trade-off for ease of implementation, which result in a simple algorithm with $O(n)$ time complexity.

**Algorithm 3** $i$-th cyclic permutation of $X$ with $n$ items

**function** UnrankCyclicPermute($X, i$)
  $X' \leftarrow$ a copy of array $X$
  **for** $k \in [n-1, n-2, \cdots, 1]$ **do**
    $(i, j) \leftarrow \text{DivMod}(i, k)$
    swap $X'[j]$ with $X'[k]$
  **end for**
  **return** $X'$
**end function**

## 3. The $O(n^3)$ Algorithm

In this section, we describe our $O(n^3)$-time algorithm to find the $i$-th derangement with $m$ cycles. To find the required random derangement with this algorithm, one first compute $s_2(n, m)$ in time $O(n^2)$, then choose a random index $0 \le i < s_2(n, m)$ as an input to the unranking function.

We briefly describe our approach based on dynamic programming. Typically, if one could count the number of objects with some property, one could follow the step backward to regenerate the $i$-th object. Our algorithm follows the same principle. While the 2-associated Stirling number of the first kind $s_2(n,m)$ counts the number of derangements with $m$ cycles, it does not give us enough structure for reconstruction. Our key ingredient is a permutation type $\mathbf{t}$ (see Section 6.2 in [9]). Instead of using recurrence for $s_2(n,m)$ to count derangements, we use recurrences for permutation types, more specifically the partial-type $s_k^v(\cdot,\cdot)$, to do so, and by reconstructing backward from the smallest cycles, one can reconstruct the complete $i$-th derangement. To see this, consider the topmost level of the recursion where we compute $s_2(n,m)$ as

$$s_2(n,m) = s_2^0(n,m) + s_2^1(n,m) + s_2^2(n,m) + \cdots + s_2^m(n,m). \tag{3.1}$$

Suppose that we want to find the $i$-th derangement. Since it is one of the $s_2(n,m)$ derangements, it is counted in one of the term $s_2^v(n,m)$ in the summation above. If we find $v$ such that

$$\sum_{j=0}^{v-1} s_2^j(n,m) \le i < \sum_{j=0}^{v} s_2^j(n,m), \tag{3.2}$$

we know partially that the type $\mathbf{t}$ of the $i$-th derangements is $\mathbf{t}(2) = v$. We also know that the $i$-th derangement is the $i'$-th derangement with $\mathbf{t}(2) = v$, where

$$i' = i - \left( \sum_{j=0}^{v-1} s_2^j(n,m) \right). \tag{3.3}$$

We define $\mathbf{i}(2)$ of this derangement to be $i'$. The formal definition of $\mathbf{i}$ will be given in the following subsection.

Using the same idea, recursively, we can find $\mathbf{t}(3)$, $\mathbf{t}(4)$, and so on, together with corresponding indices $\mathbf{i}(3)$, $\mathbf{i}(4)$. This is the first phase, the decomposition phase, of our algorithm described in Subsection 3.1.

The second phase, the reconstruction phase, described in 3.2 actually reconstructs the $i$-th derangement from the type $\mathbf{t}$ and index $\mathbf{i}$.

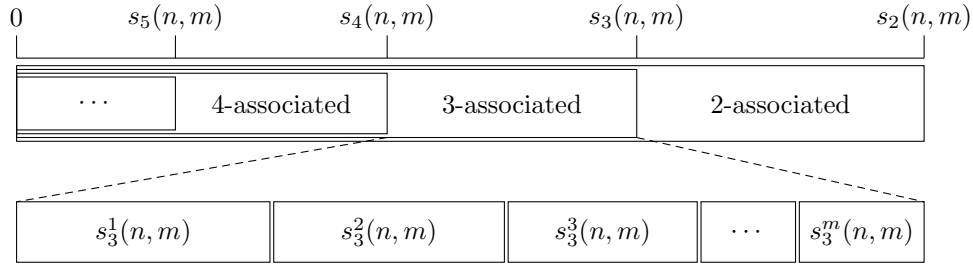


FIGURE 1. Type structure of the derangement

## 3.1. The Decomposition Phase

In this phase, we use $i$ to find a type signature $\mathbf{t}$. We also decompose the given index $i$ into a subindex $\mathbf{i}(k)$ for the corresponding $\mathbf{t}(k)$. To define $\mathbf{i}$ of a derangement formally, we

first have to define how one would order derangements. The derangements are ordered, first, by the lexicographical ordering of their types. For the derangements of the same types, they are ordered lexicographically by the cycles of each length, with priority given to shorter cycles.

Intuitively, the index $\mathbf{i}(k)$ of a derangement $\pi$, is the index for choosing and arranging cycles of length $k$. Let $X'$ be the set of items in $\pi$ such that its cycles are of length at least $k$. If we construct a derangement by choosing cycles from the shorter ones, we may choose items from $X'$ to form $\mathbf{t}(k)$ cycles of length $k$. Thus, $\mathbf{i}(k)$ is the index for the chosen cycles, each of length $k$, for $\pi$ among all possible $\binom{|X'|}{k\mathbf{t}(k)}B(k\mathbf{t}(k),k)$ choices.

For the given $n$, $m$, and $i$, we would like to recursively find $\mathbf{t}$ for the corresponding $i$-th derangement. The recursive problem also include parameter $r$, the lower bound of the length of the smallest cycle; at the top-most call $r=2$. More specifically, a function ReconstructType takes $n$,$m$,$r$, and $i$, where $i$ is the index of a derangement on $n$ items with $m$ cycles, with the restriction that the minimum length of each cycle is at least $r$. It returns the type $\mathbf{t}$ with subindices $\mathbf{i}$ of the corresponding derangement.

To do so, we first find (1) the minimum $k \geq r$ such that $\mathbf{t}(k) > 0$ and also (2) the value of $\mathbf{t}(k)$. To find $k$, we write $s_r(n,m)$ as

$$\begin{aligned} s_r(n,m) =& \big(s_r(n,m) - s_{r+1}(n,m)\big) \;+\; \big(s_{r+1}(n,m) - s_{r+2}(n,m)\big) \;+\; \cdots \;+ \\ & \big(s_{n-1}(n,m) - s_n(n,m)\big). \end{aligned} \tag{3.4}$$

Each term $s_k(n,m) - s_{k+1}(n,m)$ counts the number of derangements whose minimum cycle length is exactly $k$. One can thus find $k$ such that

$$s_r(n,m) - s_k(n,m) \leq i < s_r(n,m) - s_{k+1}(n,m). \tag{3.5}$$

Let $i' = i - (s_r(n,m) - s_k(n,m))$ be the ordering of the $i$-th derangement among the derangements whose type $\mathbf{t}'$ is such that $\mathbf{t}'(j) = 0$ for $j < k$ and $\mathbf{t}'(k) > 0$, i.e., the required $i$-th derangement is the $i'$-th derangement in these $s_k(n,m) - s_{k+1}(n,m)$ derangements. To find the number of cycles of length $k$ in the $i$-th derangement, we turn to partial-type Stirling number of the first kind and note that

$$s_k(n,m) - s_{k+1}(n,m) = s_k^1(n,m) + s_k^2(n,m) + \cdots + s_k^m(n,m). \tag{3.6}$$

Therefor, the number of cycles of length $k$ is the index $\ell$ such that

$$\sum_{j=1}^{\ell-1} s_k^j(n,m) \leq i' < \sum_{j=1}^{\ell} s_k^j(n,m). \tag{3.7}$$

We let $i'' = i' - \sum_{j=1}^{\ell-1} s_k^j(n,m)$.

From the above discussion, we see that to find the minimum cycle length $k$ and the number of cycles of that particular length $\mathbf{t}(k)$ can be done simply by searching the values of $s_k$ and $s_k^\ell$. Algorithm 4 describes the process, where function FindSmallestCycleLength searches for the index $k$ and the offset $i'$ and function FindNumSmallestCycles finds the value for $\mathbf{t}(k)$. Given $k$ and $\mathbf{t}(k)$, during the reconstruction phase we would choose $k\mathbf{t}(k)$ items from the item set and create $\mathbf{t}(k)$ random cycles from them, and the rest of the items should belong to longer cycles. Note that for each choice for choosing these $\mathbf{t}(k)$ cycles, there are

$$s_{k+1}(n - k\mathbf{t}(k), m - \mathbf{t}(k)) \tag{3.8}$$

choices for the rest of the permutation. Therefore, to continue the process for longer cycles, we can write the index $i''$ as

$$i'' = \mathbf{i}(k) \cdot s_{k+1}(n - k\mathbf{t}(k), m - \mathbf{t}(k)) + j, \tag{3.9}$$

where $j < s_{k+1}(n-k\mathbf{t}(k), m-\mathbf{t}(k))$. The index $\mathbf{i}(k)$ is used to construct $\mathbf{t}(k)$ cycles and $j$ is the ordering for the rest of the construction where we build a permutation containing longer cycles by recursively invoking ReconstructType$(n-k\mathbf{t}(k), m-\mathbf{t}(k), k+1, j)$.

**Algorithm 4** Reconstruction of type and subindices given an index of a permutation

```
function FindSmallestCycleLength(n, m, r, i)
    if i < s_r(n, m) then
        return FindSmallestCycleLength(n, m, r+1, i)
    end if
    return (r−1, i−s_r(n, m))
end function
function FindNumSmallestCycles(n, m, k, i)
    for v ∈ [1, 2, ··· , ⌊n/k⌋] do
        if i < s_k^v(n, m) then
            return (v, i)
        end if
        i ← i − s_k^v(n, m)
    end for
end function
function ReconstructType(n, m, r, i)
    if m = 0 then
        return ∅
    end if
    (k, i') ← FindSmallestCycleLength(n, m, r, i)
    (t(k), i'') ← FindNumSmallestCycles(n, m, k, i')
    (i(k), j) ← DivMod(i'', s_{k+1}(n−kt(k), m−t(k)))
    return [ (k, t(k), i(k)) ] + ReconstructType(n−kt(k), m−t(k), k+1, j)
end function
```

The next lemma considers the running time of ReconstructType.

**Lemma 3.1.** ReconstructType *in algorithm 4 runs in* $O(n^3)$ *time.*

*Proof.* We first deal with the running time for computing all Stirling numbers and their extensions needed. First of all we can compute $s_r(\cdot,\cdot)$ for every $r$ in time $O(n^3)$ since computing $s_r(n,m)$ for a particular $r$ takes $O(n^2)$ as discussed in Section 2. Therefore we assume that we have all values of the $r$-associated Stirling number of the first kinds available when running the recursive procedure. From this, we have that functions FindSmallestCycleLength runs in time $O(n)$.

After knowing $k$, Function FindNumSmallestCycles only needs $s_k^v(n,m)$ for every $v$, which can be computed in time $O(1)$ based on available $s_{k+1}(\cdot,\cdot)$. Thus FindNumSmallestCycles also works in $O(n)$ time.

Since we spend $O(n)$ time for each recursion level and there can be at most $n$ levels of the recursion, we have that the running time for ReconstructType is $O(n^2)$, without

the preprocessing for $s_k(\cdot,\cdot)$. Combining with the preprocessing time, we have that the whole algorithm runs in $O(n^3)$ time. ∎

It is useful to have a sharper bound on the number of levels of recursion and consider the running time of ReconstructType without the preprocessing needed for the Stirling numbers. The following Lemma will be very useful when deriving a faster algorithm.

**Lemma 3.2.** *Suppose that we can access to all $s_r(\cdot,\cdot)$ for every $r$ in time $O(1)$.* ReconstructType *runs in time $O(n^{1.5})$. More specifically,* ReconstructType *only makes at most $O(\sqrt{n})$ recursive calls.*

*Proof.* For the running time, we only need to bound the number of levels of recursion. Note that each recursive call involves a different value of cycle lowerbound $r$. Since the sum of cycle lengths in a permutation is at most $n$ and they are all different, there can be no more than $1 + \lceil 2\sqrt{n} \rceil$ cycle lengths since

$$\sum_{i=1}^{1+\lceil 2\sqrt{n} \rceil} i > n. \tag{3.10}$$

We conclude that the algorithm never makes more than $1 + \lceil 2\sqrt{n} \rceil = O(\sqrt{n})$ recursive calls. ∎

## 3.2. The Reconstruction Phase

In this phase, we take the output type signature $\mathbf{t}$ and subindices $\mathbf{i}$ from the previous phase to build up the final result derangement of desired number of cycles.

Based on the analysis of (2.2), each of $\mathbf{i}(k)$ corresponding to choosing $k\mathbf{t}(k)$ items from $n_k$ items, the number of remaining items after done arranging all of cycles of length less than $k$. Then partition selected items and make many cycles of length $k$. Thus, we must break down $\mathbf{i}(k)$ furthermore, which are $\mathbf{i}'(k)$ index for choosing items; $\mathbf{i}''(k)$ index for partitioning into base cycles; and $\mathbf{i}'''_\ell(k)$ index for cyclic permutation on each cycle of $0 \le \ell < \mathbf{t}(k)$. Function UnrankDerangementCycles in algorithm 5 describes the details of this assembling process. Also refer to Figure 2 for an example.

We have the following lemma.

**Theorem 3.3.** UnrankDerangementCycles *in algorithm 5 runs in time $O(n^3)$. Therefore, there exists an algorithm for uniformly generating a random derangement of $n$ items with $m$ cycles in $O(n^3)$ time.*

*Proof.* First consider UnrankDerangementCycles. The function ReconstructType runs in time $O(n^3)$ and returns a type signature $\mathbf{t}$ with at most $O(\sqrt{n})$ different cycle lengths. Each loop of the main function is dominated by the call to UnrankPeriodicChoose which runs in time $O(n^2/k) = O(n^2)$; thus given the type information, it takes $O(n^2\sqrt{n}) = O(n^{2.5})$ to reconstruct the derangement.

To randomly generate a derangement, consider function RandomDerangementCycles in algorithm 5 that first chooses a random index $i$ and then calls UnrankDerangementCycles to find the $i$-th derangement. ∎

**Algorithm 5** $i$-th derangement of $n$ items with $m$ cycles, and its random counterpart

**function** UnrankDerangementCycles($X, m, i$)
  $\pi \leftarrow \emptyset$
  $T \leftarrow$ ReconstructType($|X|, m, 1, i$)
  **for** $(k, \mathbf{t}(k), \mathbf{i}(k)) \in T$ **do**
    $(\mathbf{i}(k), \mathbf{i}'(k)) \leftarrow$ DivMod($\mathbf{i}(k), \binom{|X|}{k\mathbf{t}(k)}$)
    $X^* \leftarrow$ UnrankChoose($X, k\mathbf{t}(k), \mathbf{i}'(k)$)
    $(\mathbf{i}(k), \mathbf{i}''(k)) \leftarrow$ DivMod($\mathbf{i}(k), B(k\mathbf{t}(k), k)$)
    **for** $X^\dagger \in$ UnrankPeriodicChoose($X^*, k, \mathbf{i}''(k)$) **do**
      $(\mathbf{i}(k), \mathbf{i}'''(k)) \leftarrow$ DivMod($\mathbf{i}(k), (k-1)!$)
      $\pi \leftarrow \pi \cdot$ UnrankCyclicPermute($X^\dagger, \mathbf{i}'''(k)$)
    **end for**
    $X \leftarrow X \setminus X^*$
  **end for**
  **return** $\pi$
**end function**
**function** RandomDerangementCycles($n, m$)
  $X \leftarrow [0, 1, 2, \cdots, n-1]$
  $i \leftarrow$ uniform random an integer such that $0 \le i < s_2(n, m)$.
  **return** UnrankDerangementCycles($X, m, i$)
**end function**

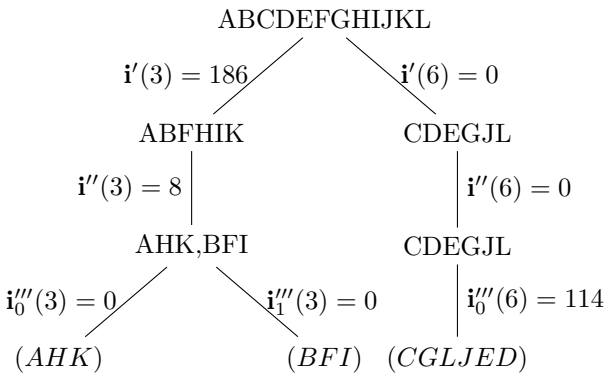


FIGURE 2. Example reconstruction given type signature and subindices

## 4. The $O(n^{2.5} \log n)$ Algorithm

In this section, we make an improvement on the algorithm shown in Section 3 to obtain an $O(n^{2.5} \log n)$-time algorithm.

Consider the search loop in FindSmallestCycleLength. Since there are at most $n$ possible values for $k$, we can use binary search to speed up the search to only $O(\log n)$ iterations. Not only that this implies a faster running time for this specific function, it implies that we only look at $O(\log n)$ values of $k$ for $s_k(n, m)$. (Figure 3 illustrates the idea.) Function FindSmallestCycleLength in algorithm 6 gives the implementation.

The same idea can be applied to function FindNumSmallestCycles to reduce the running time down to $O(\log n)$, provided that values of $s_k^v(\cdot,\cdot)$'s are available. This implies the total running time for a binary-search ReconstructType of $O(\sqrt{n}\log n)$, without preprocessing time.

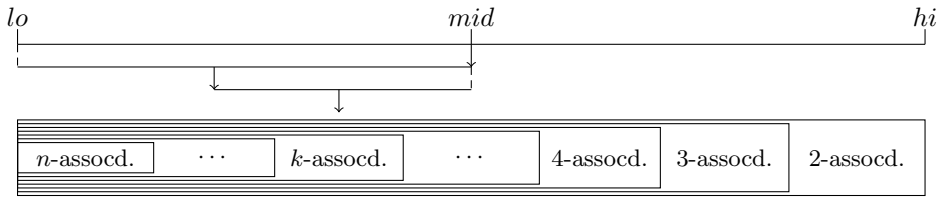


FIGURE 3. Binary search for $k$ over the type structure

**Algorithm 6** Improvement by binary search

```
function FindSmallestCycleLength(n, m, r, i)
    (lo, hi) ← (r, ⌊n/m⌋ + 1)
    while lo < hi do
        mid ← ⌊(lo+hi)/2⌋
        if i < s_mid(n, m) then
            lo ← mid + 1
        else
            hi ← mid
        end if
    end while
    k ← lo − 1
    return (k, i − s_{k+1}(n, m))
end function
```

We account for the preprocessing time by noting that when considering the $r$-associated Stirling numbers of the first kind, we only look at $O(\log n)$ values of $k$'s for each level, with the total of $O(\sqrt{n}\log n)$ values across all recursion levels. Computing those entries, only when needed, takes $O(n^2\sqrt{n}\log n)$ time as each $s_k(n,m)$ requires $O(n^2)$ time.

Since the function for the second phase only runs in time $O(n^{2.5})$, the total running time for UnrankDerangementCycles is $O(n^{2.5}\log n)$ as stated below.

**Theorem 4.1.** *Given an index $i$ of the derangement, there is an algorithm to find the $i$-th derangement with $m$ cycles in time $O(n^{2.5}\log n)$. This algorithm can be used to uniformly generate a random derangement of $n$ items with $m$ cycles in the same running time.*

We can a have a simple extension where we want to find a derangement with $n$ items with $m$ cycles whose cycle lengths are lower bounded by $r$.

**Corollary 4.2.** *There exists an algorithm for generating a uniform permutation of $n$ items with $m$ cycles, where each cycle have length at least $r$, in $O(n^{2.5}\log n)$ time.*

## 5. Dealing with Large Numbers

Previously we work under the RAM model of computation where we can perform simple arithmetic operations in constant time. All running time analysis in previous sections make this assumption when dealing with large integers, especially with factorials.

This is unrealistic as the values of factorials can be very large. For a given $n$, we note that $n! \leq n^n$; therefore, it requires an $O(n \log n)$-bit integer. To take this into account, we let $M(N)$ be the running time for an algorithm for multiplying $N$-bit integers; thus, our algorithm runs in time $O(n^{2.5} \log n \cdot M(n \log n))$ if all factorials are available. Computing all factorials takes only $O(n \cdot M(n \log n))$, which is dominated by the running time of the algorithm.

If one employs a straightforward multiplication algorithm, where $M(N) = N^2$, one would get an algorithm with running time $O(n^{4.5} \log^3 n)$, factor of $O(n^2 \log^2 n)$ slowdown in the running time. However, with a more efficient multiplication algorithm such as the one by Harvey and van der Hoven [18] where $M(N) = O(N \log N)$, one would obtain an $O(n^{3.5} \log^3 n)$-time algorithm, as claimed.

## 6. Experiments

In this section, we present empirical data for the uniformity and the claimed running time of the proposed algorithm in Section 4. The laptop used for this task runs on Intel Core i5-10210U with 16GB RAM. The operating system is Ubuntu 22.04 LTS on Windows 10 WSL2. We implemented the algorithm in Python 3.10 since it uses big-integer as a default, playing a vital role for precise factorial computations. Python also comes with the standard library `random` for generating pseudo-random number, implementing MT19937 [19] as a randomizer. The actual python implementation of the algorithm can be found at https://github.com/neizod/derangement.

### 6.1. Uniform Generation

We first demonstrate that the generated derangements are uniformly distributed. Since the number of distinct permutations is very large (i.e., $n!$), we restrict the values of $n$ and $m$ to small numbers. In each case, we run the algorithm for $10^7$ times to obtain the data.

For the first case, we deal with derangements with one cycle. We set $n = 8$ and $m = 1$, yielding $s_2(8, 1) = 5040$ distinct derangements. See the histogram in Figure 4. We observe that the standard deviation of the number of occurrences is 44.752 which is reasonably close to the expected 44.539, calculated with an assumption that each derangement is generated independently.

The second case considers derangements with two cycles. We let $n = 7$ and $m = 2$. There are $s_2(7, 2) = 924$ distinct derangements in this case. See histogram in Figure 5. We observe that the standard deviation of the number of occurrences is 103.182 (as compared to the expected 103.975, again calculated with an independence assumption).

For larger values of $n$, we investigate the distribution of $\pi(0)$ instead of the entire space of derangements. If the derangement algorithm works correctly, then $Pr[\pi(0) = u] = 1/(n-1)$ for every $u \neq 0$. We choose $n = 100$ and $m = 10$, then experiment $10^6$ times and observe the distribution of $\pi(0)$. The histogram is shown in Figure 6. We remark that we should see no generated derangements where $\pi(0) = 0$. The sampled standard deviation is 102.41, which is close to the expected 99.99 calculated under an independence assumption.

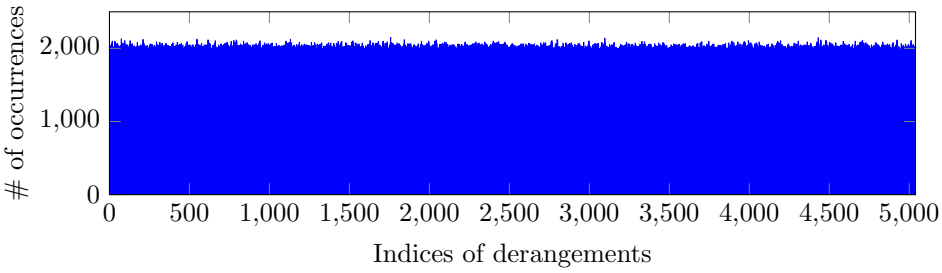


FIGURE 4. The distribution of generated derangements when $n = 8$ and $m = 1$.

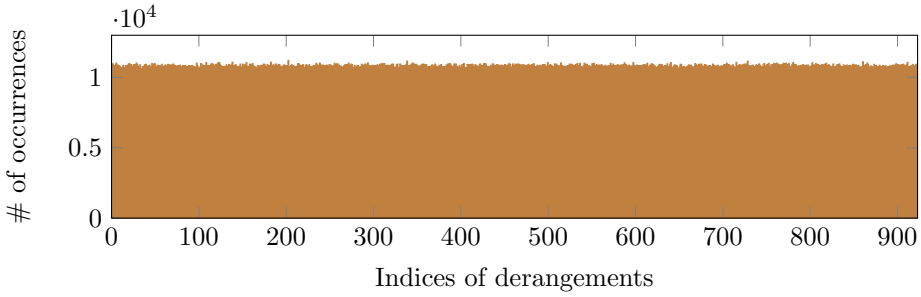


FIGURE 5. The distribution of generated derangements when $n = 7$ and $m = 2$.

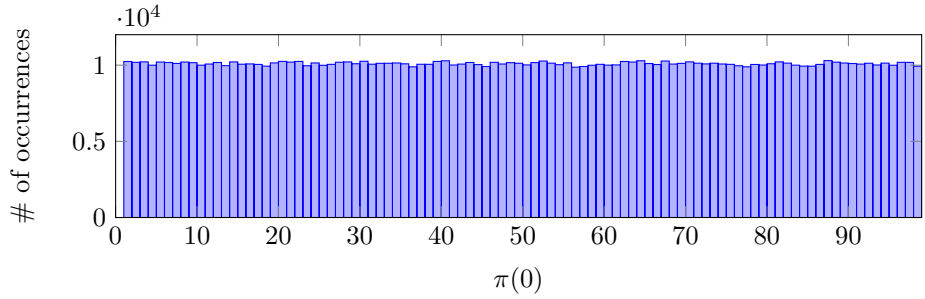


FIGURE 6. The distribution of $\pi(0)$ when $n = 100$ and $m = 10$.

## 6.2. RUNNING TIME

The worst-case running time bound of $O(n^{3.5} \log^3 n)$ makes no assumption on $m$, the number of cycles. To see the effects of $m$ to the running time, we perform an experiment where $n = 1000$ with many values of $m$. For each data point, we perform 10 runs. The result in Figure 7 demonstrates that $m$ is an important factor in the running time.

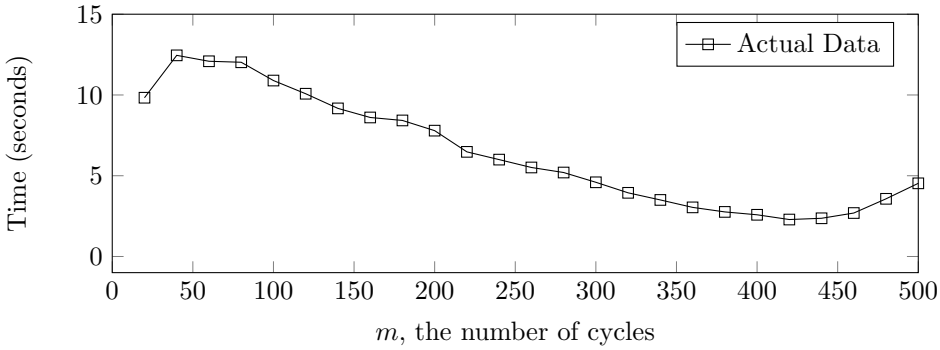


FIGURE 7. Effects of $m$ to the actual running times.

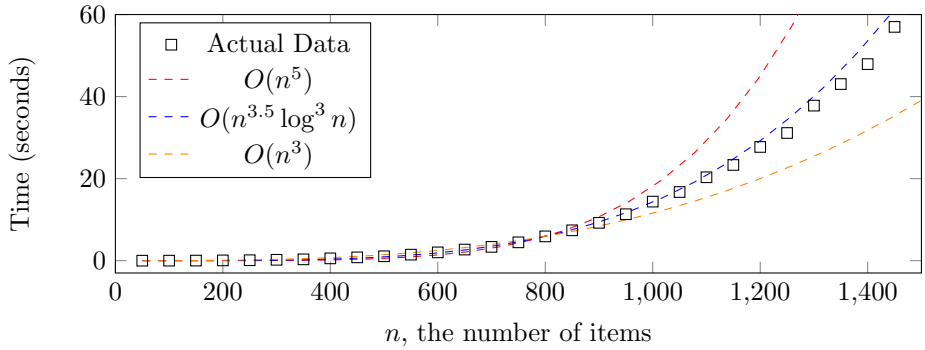


FIGURE 8. Running times for $n$ from 50 to $1,450$ with $m = 0.05n$.

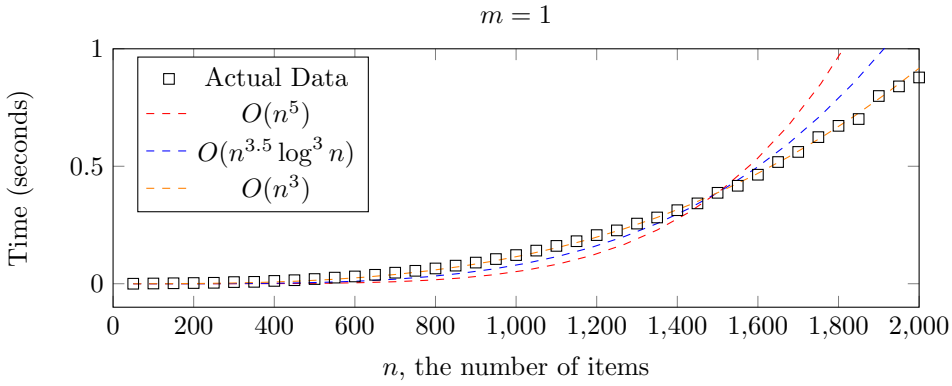


FIGURE 9. Experiment of running time, case 3

To demonstrate the running time for various values of $n$ from 50 to $1,450$, we let $m$ be 5% of $n$. For each configuration, we perform 10 experiments and show the average running

times in Figure 8. For comparison, we draw curves of different asymptotic complexities (normalized to fix small actual values) as well.

We consider the case where $m = 1$ shown in Figure 9. In this case, the experiment hints a faster running time, possibly $O(n^3)$. We suspect that this speed up might come from the fact we only need one table level, $s_2(n, 1)$, when $m = 1$. However, for the case when $m = 1$, one can use Sattolo's algorithm [5] which requires only linear time.

## Acknowledgements

We would like to thank the referees for their comments and suggestions on the manuscript. This research was partially funded by the Thailand Research Fund, grant number RSA-6180074.

## References

[1] Pierre Rémond de Montmort. *Essay [sic] d'analyse sur les jeux de hazard*. Jacques Quillau, 1713.

[2] Conrado Martínez, Alois Panholzer, and Helmut Prodinger. Generating random derangements. In *2008 Proceedings of the Fifth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 234–240. SIAM, 2008.

[3] Kenji Mikawa and Ken Tanaka. Linear-time generation of uniform random derangements encoded in cycle notation. *Discrete Applied Mathematics*, 217:722–728, 2017.

[4] J Ricardo G Mendonça. Efficient generation of random derangements with the expected distribution of cycle lengths. *Computational and Applied Mathematics*, 39(3):1–15, 2020.

[5] Sandra Sattolo. An algorithm to generate a random cyclic permutation. *Information processing letters*, 22(6):315–317, 1986.

[6] Philippe Flajolet and Michele Soria. Gaussian limiting distributions for the number of components in combinatorial structures. *Journal of Combinatorial Theory, Series A*, 53(2):165–182, 1990.

[7] Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[8] Kenji Mikawa and Ken Tanaka. Efficient linear-time ranking and unranking of derangements. *Information Processing Letters*, 179:106288, 2023.

[9] Louis Comtet. *Advanced Combinatorics: The art of finite and infinite expansions*. Springer Science & Business Media, 2012.

[10] John Riordan. *An introduction to combinatorial analysis*. Princeton University Press, 2014.

[11] W.J. Ewens. The sampling theory of selectively neutral alleles. *Theoretical Population Biology*, 3(1):87–112, 1972.

[12] Poly H. da Silva, Arash Jamshidpey, and Simon Tavar. The feller coupling for random derangements. *Stochastic Processes and their Applications*, 150:1139–1164, 2022.

[13] Nattawut Phetmak. Random derangement with fixed number of cycles. In *The 23rd Thailand-Japan Conference on Discrete and Computational Geometry, Graphs, and Games (JCDCGGG'20+1)*, pages 34–35, 2021.

[14] Henry Bottomley. Entry A060540 in The On-Line Encyclopedia of Integer Sequences. http://oeis.org/A060540. Accessed: 2021-12-21.

[15] Charalambos A Charalambides. *Enumerative combinatorics*. CRC Press, 2002.
[16] Albert Nijenhuis and Herbert S Wilf. *Combinatorial algorithms: for computers and calculators*. Elsevier, 2014.
[17] Jörg Arndt. *Matters Computational: ideas, algorithms, source code*. Springer Science & Business Media, 2010.
[18] David Harvey and Joris Van Der Hoeven. Integer multiplication in time o (n log n). *Annals of Mathematics*, 193(2):563–617, 2021.
[19] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.