

# Generating Random Permutations

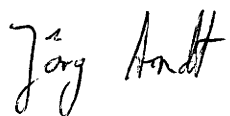
Jörg Arndt

30 October 2009  
(revised 10 March 2010)

A thesis submitted for the degree of Doctor of Philosophy  
of the Australian National University

# Declaration

The work in this thesis is my own except where otherwise stated.

A handwritten signature in black ink, reading "Jörg Arndt". The signature is written in a cursive style with a large, stylized 'J' and 'A'.

Jörg Arndt

# Contents

<b>Acknowledgments</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Notation, terminology, and conventions</b>	<b>3</b>
2.1 Basic definitions and notation . . . . .	3
2.2 Representation in a computer . . . . .	4
2.3 The cycle type . . . . .	5
2.4 Stirling cycle numbers . . . . .	5
2.5 Transpositions and parity . . . . .	6
2.6 Uniform projections . . . . .	7
2.7 Mixed radix and factorial numbers . . . . .	7
2.8 Ranking and unranking . . . . .	8
2.9 Symbol for 'precedes' . . . . .	8
2.10 Cyclic shifts . . . . .	9
2.11 Specifications and floating-point model . . . . .	9
2.12 Symbols, auxiliary routines, and further remarks . . . . .	9
<b>3 All permutations and cyclic permutations</b>	<b>11</b>
3.1 Arbitrary permutations . . . . .	11
3.2 Permutations of a multiset . . . . .	12
3.3 Prefix of a permutation . . . . .	13
3.4 Permutation modulo cyclic shifts . . . . .	13
3.5 Cyclic permutations . . . . .	13
3.6 Permutations with given parity . . . . .	16
3.7 Permutations modulo reversal . . . . .	16
<b>4 Putting elements into cycles</b>	<b>19</b>
4.1 Conversion between array form and CCF . . . . .	19
4.2 Prescribed elements in one cycle . . . . .	21
4.3 Prescribed elements in distinct cycles . . . . .	22
4.4 Prescribed sets in distinct cycles . . . . .	22
<b>5 Given cycle type</b>	<b>25</b>
5.1 Maintenance of the set . . . . .	25

5.2	One cycle of prescribed length . . . . .	25
5.3	Permutations with given cycle type . . . . .	27
<b>6</b>	<b>Number of cycles</b>	<b>31</b>
6.1	Bijection via swaps . . . . .	31
6.2	A property of the bijection . . . . .	34
6.3	Exactly 2 cycles . . . . .	35
6.4	Even or odd number of cycles . . . . .	37
6.5	Exactly $m$ cycles . . . . .	38
<b>7</b>	<b>Inversions</b>	<b>43</b>
7.1	Inversion table . . . . .	43
7.2	Fast conversion to and from the inversion table . . . . .	45
7.3	Inversions modulo $m$ . . . . .	47
7.4	Fixed number of inversions (open problem) . . . . .	48
<b>8</b>	<b>Connected permutations</b>	<b>49</b>
8.1	Asymptotics . . . . .	49
8.2	The rejection method . . . . .	50
8.3	Improved algorithm . . . . .	51
8.4	Implementation . . . . .	52
<b>9</b>	<b>Involutions</b>	<b>55</b>
9.1	The number of involutions . . . . .	55
9.2	Branching probabilities . . . . .	56
9.3	Practical algorithm . . . . .	57
9.4	Forward variant . . . . .	59
<b>10</b>	<b>Cycle spectrum</b>	<b>63</b>
10.1	Prescribed maximal cycle spectrum . . . . .	63
10.2	Outline of the algorithm . . . . .	64
10.3	Algorithm . . . . .	65
10.4	Implementation . . . . .	66
<b>11</b>	<b>Derangements</b>	<b>69</b>
11.1	The number of derangements . . . . .	69
11.2	Branching probabilities . . . . .	69
11.3	Algorithm . . . . .	71
11.4	Comparison with rejection method . . . . .	72
11.5	Length of cycles at least $m$ . . . . .	73
<b>A</b>	<b>Bit-array</b>	<b>77</b>
<b>B</b>	<b>Left-right array</b>	<b>81</b>
	<b>Bibliography</b>	<b>87</b>

# Acknowledgments

First and foremost I'd like to thank Richard Brent, my supervisor, for giving me the opportunity to do my research in such an inspiring environment. It has been a great pleasure to work with you.

Thanks go to Adam Rennie for reading through the whole text and suggesting various improvements in presentation. I much appreciate that.

The two anonymous examiners gave very detailed and helpful comments on the submitted version of this thesis, thanks for that.

With Paul Leopardi I had a great many discussions which I truly enjoyed, thanks man.

Lance Gurney, Greg Stephenson, and Charles Baker were crucial whenever it came to discussions regarding matters 42. I appreciated that a lot. Thanks guys.

Thanks go to Edith Parzefall for enduring many discussions about Gray codes.

Last but not least, I am indebted to the Australian taxpayer. You provided much of my budget which has been converted into the algorithms presented here. I do thank you all.

# Abstract

We give algorithms for generating unbiased random permutations of certain types, such as

- Permutations with prescribed parity.
- Permutations modulo cyclic shifts.
- Permutations modulo reversal.
- Permutations with prescribed elements in one cycle.
- Permutations with prescribed elements in distinct cycles.
- Permutations with prescribed sets of elements in distinct cycles.
- Permutations with prescribed cycle type.
- Permutations with even or odd number of cycles.
- Permutations with prescribed number of cycles.
- Permutations with  $r$  inversions modulo  $m$  where  $2 \leq m \leq n$  (where  $n$  is the length of the permutation).
- Connected (indecomposable) permutations.
- Self-inverse permutations (involutions).
- Permutations with prescribed maximal cycle spectrum.
- Permutations with all cycles of minimum length  $m$ .

All algorithms appear to be new and many have optimal (linear) complexity. Implementations are given which can easily be translated into the programming language of the reader's choice.

# Chapter 1

## Introduction

Algorithms for the generation of all combinatorial objects are known for most of the standard types such as combinations, integer partitions, or permutations (for example, see [23], [19], or [2]). These algorithms are often used for either testing purposes or as building blocks for the generation of structures based on the respective combinatorial objects.

The fast growth of the number of objects, however, usually makes the actual generation of all objects infeasible in practice. Therefore algorithms for unbiased random generation are desirable. Such algorithms for random subsets, combinations, compositions, permutations, integer partitions, set partitions, and unlabeled rooted trees are given in [19].

In this thesis we restrict our attention to the random generation of special types of permutations, such as involutions.

In the following the year of publication is given in parentheses. An algorithm for the unbiased generation of a random permutation by a computer was given by Durstenfeld (1964) [7]. Note that Knuth [13, alg. P, sect. 3.4.2] attributes the algorithm to Fisher and Yates (1938) [9]. The problem of generating random permutations in external memory is treated in [11] (2008). An algorithm for cyclic permutations was given by Sattolo (1986) [25].

Given the relative ease by which these algorithms are found one may suspect that algorithms for the random generation of special types of permutations such as involutions or permutations with minimum cycles length would be known. However, as of the time of this writing only two such algorithms can be found in the literature: Diaconis et.al. (2001) [6, prop. 2.3, p. 201] give an algorithm for permutations where an element can be displaced by at most one unit (Fibonacci permutations) and Martínez et.al. (2008) [15] for permutations without fixed points (derangements).

The lack of such algorithms is even more striking as statistical properties of combinatorial objects is an area of active research and there even is a book on statistics of permutations by Bóna (2004) [3].

The main reason for this appears to be the fact that the algorithms for permutations and cyclic permutations have an unusually simple structure. For most of the algorithms presented we need to keep track of a set of elements whose processing has not yet finished (see section 5.1 on page 25), some branching probabilities have to be computed (see section 9.2 on page 56 for the case of involutions), and the specific

constructions in the (recursive or iterative) steps have to be identified (see section 11.3 on page 71 for derangements).

The computation of each probability needs to be done in  $O(1)$  time to obtain the best complexity. Indeed all of the important algorithms presented have linear complexity which is optimal. These can be expected to appear in all software packages dealing with combinatorial structures, such as computer algebra systems.



## Chapter 2

# Notation, terminology, and conventions

### 2.1 Basic definitions and notation

The group  $\mathcal{S}_n$  of all bijections from the set of  $n$  elements to itself is called the *symmetric group*. A *permutation*  $P$  is an element of  $\mathcal{S}_n$ . In the description of the algorithms and for the implementations we take the set of  $n$  elements as either  $\{0, 1, \dots, n-1\}$  or  $\{1, 2, \dots, n\}$ . A standard form to write down a particular permutation is the *two-line notation*

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & \dots & n-1 \\ p(0) & p(1) & p(2) & p(3) & \dots & p(n-1) \end{pmatrix} \quad (2.1)$$

The permutation that fixes all elements is the *identical permutation* (or *identity*)  $1_n$ , the unit in  $\mathcal{S}_n$ . The group multiplication of  $\mathcal{S}_n$  is given by the *composition* of permutations as follows: For  $P$  and  $Q$  in  $\mathcal{S}_n$  define  $R = QP$  via

$$r(i) := q(p(i)) \quad (2.2)$$

The *inverse*  $P^{-1}$  of  $P \in \mathcal{S}_n$  is the permutation such that  $PP^{-1} = P^{-1}P = 1_n$ . The inverse is computed by swapping the upper and lower row in the two-line notation, followed by sorting with respect to the upper line. For example, the inverse of

$$P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 4 & 3 & 2 & 0 & 1 & 9 & 8 & 5 & 6 & 7 \end{pmatrix} \quad (2.3)$$

is

$$P^{-1} = \begin{pmatrix} 4 & 3 & 2 & 0 & 1 & 9 & 8 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 4 & 2 & 1 & 0 & 7 & 8 & 9 & 6 & 5 \end{pmatrix} \quad (2.4)$$

We write  $P^j$  ( $j \in \mathbb{Z}$ ) for the  $j$ -fold composition of  $P$  with itself, where  $P^0$  is the identity and  $P^j = (P^{-1})^{-j}$  for  $j < 0$ .

Every permutation  $P$  is the union of disjoint cycles (because  $p^{k_u}(u) = u$  for some  $k_u$ , where  $k_u$  is the length of the cycle containing  $u$ ). The *cycle notation* of  $P$  is

$$(u, p(u), p^2(u), \dots, p^{k_u-1}(u)) (v, p(v), p^2(v), \dots, p^{k_v-1}(v)) \dots \quad (2.5)$$

For example, the permutation  $P \in \mathcal{S}_{10}$  whose cycle notation is

$$(0, 4, 1, 3) (2) (5, 9, 7) (6, 8) \quad (2.6)$$

has the two-line notation given as relation (2.3). There is some degree of freedom in the cycle notation. Firstly, cyclic shifts (notationally!) of a cycle do not change it, for example,  $(1, 2, 3)$ ,  $(2, 3, 1)$ , and  $(3, 1, 2)$  all denote the same cycle. Secondly, the order of the cycles is irrelevant.

There are four forms of normalization that are used in practice: starting with smallest element, ending with smallest, starting with largest, or ending with largest. To make the notation unique, we start each cycle with the smallest element it contains, and order the cycles by their first elements. If a cycle is written (or represented in memory) such that it starts with its smallest element, we say it is in *normalized form*.

A cycle of length 1 is called a *fixed point* of a permutation. We call a cycle of length  $k$  a *k-cycle*.

## 2.2 Representation in a computer

To represent a permutation in computer memory we mostly use the *array notation* (or *one-line notation*), obtained by omitting the top line in the two-line notation:

$$\{p(0), p(1), p(2), \dots, p(n-1)\} \quad (2.7)$$

To represent the cycle form of a permutation without extra memory (needed to mark the cycle ends), the following convention can be used: write each cycle so that it ends with its smallest element (normalization) and sort the cycles with respect to their last elements. We call this form the *canonical cycle form* (CCF). For example,

$$(4, 1, 3, 0) (2) (9, 7, 5) (8, 6) \quad (2.8)$$

is the CCF of the permutation in (2.3). The array (spaces to highlight cycle ends)

$$[4, 1, 3, 0, \quad 2, \quad 9, 7, 5, \quad 8, 6] \quad (2.9)$$

would be the computer representation of this CCF. The same data can also be interpreted as array notation of the (in general different) permutation

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 4 & 1 & 3 & 0 & 2 & 9 & 7 & 5 & 8 & 6 \end{pmatrix} \quad (2.10)$$

The bijection via the two interpretations will sometimes be useful in itself.

## 2.3 The cycle type

The *cycle type* (or simply *type*) of a permutation  $\in \mathcal{S}_n$  is the sequence  $C = [c_1, c_2, \dots, c_n]$  where  $c_k$  is the number of cycles of length  $k$ . We necessarily have

$$n = 1c_1 + 2c_2 + \dots + nc_n = \sum_{k=1}^n kc_k \quad (2.11)$$

So  $C$  corresponds to an integer partition of  $n$ . For the number  $m$  of cycles we have

$$m = \sum_{k=1}^n c_k \quad (2.12)$$

The number  $Z_{n,C}$  of permutations of  $n$  elements with type  $C$  equals [3, p. 80]

$$Z_{n,C} = \frac{n!}{c_1!c_2!c_3! \dots c_n! 1^{c_1} 2^{c_2} 3^{c_3} \dots n^{c_n}} = \frac{n!}{\prod_{k=1}^n c_k! k^{c_k}} \quad (2.13)$$

The exponential generating function  $\exp(L(z))$  where

$$L(z) = \sum_{k=1}^{\infty} \frac{t_k z^k}{k} \quad (2.14a)$$

gives detailed information about all cycle types [24, p. 68]:

$$\exp(L(z)) = \sum_{n=0}^{\infty} \left[ \sum_C (Z_{n,C} \prod t_k^{c_k}) \right] \frac{z^n}{n!} \quad (2.14b)$$

The exponent of  $t_k$  indicates how many cycles of length  $k$  are present in the given cycle type. For example, the coefficient of  $z^4$  is

$$(1t_1^4 + 6t_2t_1^2 + 8t_3t_1 + 3t_2^2 + 6t_4) / 24 \quad (2.15)$$

That is, there is one permutation  $\in \mathcal{S}_4$  with four fixed points, six with two fixed points and one 2-cycle, eight with one fixed point and one 3-cycle, and so on.

## 2.4 Stirling cycle numbers

The number of permutations of  $n$  elements into  $m$  cycles is given by the (unsigned) *Stirling numbers of the first kind* (or *Stirling cycle numbers*)  $s(n, m)$ . The first few are shown in figure 2.1. We have  $s(1, 1) = 1$  and [4, p. 214]

$$s(n, m) = s(n-1, m-1) + (n-1)s(n-1, m) \quad (2.16)$$

n:	total	m=	1	2	3	4	5	6	7	8	9
1:	1	1									
2:	2	1	1								
3:	6	2	3	1							
4:	24	6	11	6	1						
5:	120	24	50	35	10	1					
6:	720	120	274	225	85	15	1				
7:	5040	720	1764	1624	735	175	21	1			
8:	40320	5040	13068	13132	6769	1960	322	28	1		
9:	362880	40320	109584	118124	67284	22449	4536	546	36	1	

Figure 2.1: Stirling numbers of the first kind  $s(n, m)$  (Stirling cycle numbers).

for  $n > 0$  and  $m > 0$ , and  $s(n, 0) = s(0, m) = 0$  except for  $s(0, 0) = 1$ . We have

$$\sum_{m=1}^n s(n, m) x^m = x^{\bar{n}} \quad \text{where} \quad (2.17a)$$

$$x^{\bar{n}} = \prod_{m=0}^{n-1} (x + m) \quad (2.17b)$$

A generating function for the Stirling cycle numbers is given by

$$\sum_{n=0}^{\infty} e^{\bar{n}} x^n = \sum_{n=0}^{\infty} \left( \sum_{m=0}^n s(n, m) e^m \right) x^n \quad (2.18a)$$

$$\begin{aligned} &= 1 + ex + (e + e^2)x^2 + (2e + 3e^2 + e^3)x^3 + \\ &\quad + (6e + 11e^2 + 6e^3 + e^4)x^4 + \\ &\quad + (24e + 50e^2 + 35e^3 + 10e^4 + e^5)x^5 + \dots \end{aligned} \quad (2.18b)$$

## 2.5 Transpositions and parity

A *transposition* is a permutation  $\in \mathcal{S}_n$  that consists of a 2-cycle and  $n - 2$  fixed points. The minimal number of transpositions whose composition is a  $k$ -cycle is  $k - 1$ . Write  $\text{tr}(P)$  for the minimal number of transpositions whose composition is a permutation  $P \in \mathcal{S}_n$  of cycle type  $C$ . We have

$$\text{tr}(P) = \sum_{k=1}^n c_k (k - 1) \quad (2.19)$$

For a permutation  $P$  with  $m$  cycles we have (use  $\text{tr}(P) = \sum_{k=1}^n (c_k k - c_k)$ , relations (2.11) and (2.12))

$$\text{tr}(P) = n - m \quad (2.20)$$

The (nontrivial) homomorphism from  $\mathcal{S}_n$  into the group of the two elements 0 and 1 with addition modulo 2 defines the *parity* of a permutation. The parity is 0 or 1

if  $\text{tr}(P)$  is even or odd, respectively. An equivalent notion is that of the *sign* of a permutation, which is  $+1$  or  $-1$  in the respective cases (using the homomorphism into the group with elements  $+1$  and  $-1$  with multiplication).

The subgroup  $\mathcal{A}_n \in \mathcal{S}_n$  of permutations with even parity is called the *alternating group*. Permutations  $P \in \mathcal{A}_n$  are called *even permutations* ( $\text{tr}(P)$  is even). There are  $n!/2$  such permutations. Permutations  $P \notin \mathcal{A}_n$  are called *odd permutations*.

## 2.6 Uniform projections

A *projection* is a map  $f$  such that  $f^2 = f$ . We call a projection *uniform* if the number of preimages is the same for every image. For example, let  $k \leq n$  and  $f$  be the map that sorts the first  $k$  elements. Then  $f$  is uniform because every image has  $k!$  preimages. The number of preimages of each image of a uniform projection will be called the *quotient* of the projection.

If we have an algorithm for the unbiased random generation of elements  $a \in A$  and a uniform projection  $f$  onto  $B \subset A$  then an unbiased random element  $b \in B$  can be computed as  $f(a)$ .

## 2.7 Mixed radix and factorial numbers

	rising fact	falling fact
0:	[ . . . ]	[ . . . ]
1:	[ 1 . . ]	[ 1 . . ]
2:	[ . 1 . ]	[ 2 . . ]
3:	[ 1 1 . ]	[ 3 . . ]
4:	[ . 2 . ]	[ . 1 . ]
5:	[ 1 2 . ]	[ 1 1 . ]
6:	[ . . 1 ]	[ 2 1 . ]
7:	[ 1 . 1 ]	[ 3 1 . ]
8:	[ . 1 1 ]	[ . 2 . ]
9:	[ 1 1 1 ]	[ 1 2 . ]
10:	[ . 2 1 ]	[ 2 2 . ]
11:	[ 1 2 1 ]	[ 3 2 . ]
12:	[ . . 2 ]	[ . . 1 ]
13:	[ 1 . 2 ]	[ 1 . 1 ]
14:	[ . 1 2 ]	[ 2 . 1 ]
15:	[ 1 1 2 ]	[ 3 . 1 ]
16:	[ . 2 2 ]	[ . 1 1 ]
17:	[ 1 2 2 ]	[ 1 1 1 ]
18:	[ . . 3 ]	[ 2 1 1 ]
19:	[ 1 . 3 ]	[ 3 1 1 ]
20:	[ . 1 3 ]	[ . 2 1 ]
21:	[ 1 1 3 ]	[ 1 2 1 ]
22:	[ . 2 3 ]	[ 2 2 1 ]
23:	[ 1 2 3 ]	[ 3 2 1 ]

Figure 2.2: Nonnegative integers  $< 24$  expressed in rising (left) and falling (right) factorial base. Dots denote zeros. The least significant digits appear on the left.

The *mixed radix* representation  $A = [a_0, a_1, a_2, \dots, a_{n-1}]$  of an integer  $x$  with respect

to a radix vector  $M = [m_0, m_1, m_2, \dots, m_{n-1}]$  is given by

$$x = \sum_{k=0}^{n-1} a_k \prod_{j=0}^{k-1} m_j \quad (2.21)$$

where  $0 \leq a_j < m_j$  (and  $0 \leq x < \prod_{j=0}^{n-1} m_j$ , so that  $n$  digits suffice). Note that  $m_{n-1}$  is not used in the relation but it determines the range of values of  $a_{n-1}$ . For  $M = [r, r, r, \dots, r]$  the relation reduces to the radix- $r$  representation:

$$x = \sum_{k=0}^{n-1} a_k r^k \quad (2.22)$$

We will use the *rising factorial* base  $M = [2, 3, 4, \dots]$  and the *falling factorial* base  $M = [\dots, 4, 3, 2]$ . If the order of the radices is irrelevant we will just talk about the *factorial base*. Figure 2.2 shows the factorial representation of the integers  $< 24$ .

These *factorial numbers* with  $n - 1$  digits can represent all integers  $k$  where  $0 \leq k < n!$  which highlights their importance for permutations. Several bijections between factorial numbers and permutations are known. We will use two of them.

## 2.8 Ranking and unranking

Let  $L$  be the list of combinatorial objects in some (fixed) order. A *ranking* algorithm takes a combinatorial object as input and computes the index (rank) of the object in the list.

For example, the rightmost column of figure 2.2 shows the (length-4) permutations in lexicographic order (input), and the leftmost column gives the ranks of the permutations (output).

An *unranking* algorithm does the converse, its input is the rank and its output is the corresponding combinatorial object.

## 2.9 Symbol for ‘precedes’

Let  $W$  be a word of distinct letters containing both letters  $u$  and  $v$ . We write

$$u \prec v \quad (2.23)$$

if  $u$  precedes  $v$ . All words where  $u \prec v$  have the form  $[AuBvC]$  where  $A$ ,  $B$ , and  $C$  denote words of arbitrary length (including the empty word).

In a permutation in array form,  $u \prec v$  if  $u = p(k)$ ,  $v = p(j)$ , and  $k < j$ .

## 2.10 Cyclic shifts

A *cyclic shift* by  $k$  positions of a vector

$$[a_0, a_1, \dots, a_{n-1}] \quad (2.24)$$

is the vector

$$[a_{0-k}, a_{1-k}, \dots, a_{n-1-k}] \quad (2.25)$$

where subscripts are to be taken modulo  $n$ . For example, the cyclic shift by one position (*cyclic right shift*) of  $[a, b, c, d]$  is  $[d, a, b, c]$ .

## 2.11 Specifications and floating-point model

All algorithms sample uniformly (unbiased), except where stated otherwise (e.g. Algorithm 6.6 on page 34). All algorithms generate permutations in array form except where stated otherwise (e.g. Algorithm 4.5 on page 23).

Many of the algorithms given here work with floating-point numbers (C-type `double`) to represent real numbers. Due to the imprecision and roundoff-errors the achieved uniformity is not perfect (even if the random numbers are assumed to be perfect). For all practical purposes the uniformity should suffice, and floating-point numbers of higher (but fixed) precision can be used for even better uniformity.

With sufficient floating-point precision the limiting factor is rather the periodicity of the pseudorandom number generator (PRNG). Even for PRNGs with very long periods only a subset of all possible permutations (of sufficient length) is possibly generated. This problem can be mitigated by using a source of entropy to prevent periodicity.

Perfect uniformity requires both exact arithmetic (e.g. rationals) and a perfect source of random numbers. With exact arithmetic, however, the complexity of the algorithms will in general be worse because of computations with rapidly growing numbers. Another approach to perfect uniformity is the use of interval arithmetic, see [5]. No attempt for perfect uniformity is made in this thesis.

## 2.12 Symbols, auxiliary routines, and further remarks

Most implementations are given in the C++ language. Only a minimum of features beyond plain C are used: essentially only templates and (for the auxiliary routines in appendix A on page 77 and B on page 81) simple classes. For implementations where the details might obscure the underlying algorithm, the GP language [21] is used. A short introduction into the GP language is given in [2, app. C].

In the C++ implementations the type `long unsigned int` is abbreviated as `ulong`. They are part of the FXT library [1].

In the algorithms the symbol  $\mathbf{Z}(k)$  stands for a uniform random integer  $\in \{0, 1, \dots, k-1\}$ , the corresponding C++ function is `rand_idx(k)`, the GP function is `random(k)`.

The symbol  $\mathbf{R}()$  stands for a uniform random real  $t$  where  $0 \leq t < 1$ , the corresponding C++ function is `rand01()`.

The floating-point type used in C++ implementations is the type `double`, the IEEE double precision number having a 53-bit significand (mantissa).

The symbol `:=` is used for assignment in the algorithms and comments appear in parenthesis, for example:

1. Set  $k' := k - 1$  (there are  $k'$  elements left to process).



## Chapter 3

# All permutations and cyclic permutations

We recall the known algorithms for generating random permutations and random cyclic permutations. We also give some algorithms that can be obtained from these via uniform projections, such as algorithms for permutations with prescribed parity, permutations modulo cyclic shift, and permutations modulo reversal.

### 3.1 Arbitrary permutations

A random permutation  $P = [p(0), p(1), \dots, p(n-1)] \in \mathcal{S}_n$  can be computed as follows:

**Algorithm 3.1 (PermS).** *Generate a random permutation  $P \in \mathcal{S}_n$ .*

1. Set  $S := \{0, 1, \dots, n-1\}$ .
2. For  $k := 0, 1, \dots, n-1$ , do:  
     Select a random element  $s \in S$  and remove it from  $S$ . Set  $p(k) := s$ .

The first element ( $p(0)$ ) is uniformly sampled from all  $n$  elements  $\in S$ , the second from the  $n-1$  elements  $\neq p(0)$ , and so on. There are  $n!$  different ways the algorithm can proceed, and there are  $n!$  permutations altogether.  $\square$

A practical variant is the following iterative algorithm:

**Algorithm 3.2 (Perm1).** *Generate a random permutation  $P \in \mathcal{S}_n$ .*

1. Set  $P := 1_n$  (identical permutation).
2. For  $k := n, n-1, \dots, 2$  do:  
     Set  $i := \mathbf{Z}(k)$  (random  $i \in \{0, 1, \dots, k-1\}$ ). Swap  $p(i)$  with  $p(k-1)$ .

This form is due to Durstenfeld [7], it is usually called *Fisher-Yates shuffle*, sometimes also *Knuth shuffle* see [13, alg. P, sect. 3.4.2]. An alternative version of the method is (again see [13, alg. P, sect. 3.4.2])

**Algorithm 3.3 (Perm0).** *Generate a random permutation  $P \in \mathcal{S}_n$ .*

1. Set  $P := 1_n$ .
2. For  $k := 1, 2, \dots, n-1$  do:  
     Set  $i := \mathbf{Z}(k+1)$  ( $i \in \{0, 1, \dots, k\}$ ). Swap  $p(i)$  with  $p(k)$ .

With both methods we can omit the initialization of  $P$  as long as  $P$  contains data corresponding to a valid permutation on entry.

If the cycle form of a random permutation is required, simply interpret the data as canonical cycle form as defined in section 2.2.

To randomly permute an array, one could generate a random permutation and apply it to the array. However, there is a more direct way.

**Algorithm 3.4 (Perm1F).** *Apply a random permutation to the elements in  $F = [f(0), f(1), \dots, f(n-1)]$ .*

1. For  $k := n, n-1, \dots, 2$  do:  
     Set  $i := \mathbf{Z}(k)$  ( $i \in \{0, 1, \dots, k-1\}$ ). Swap  $f(i)$  with  $f(k-1)$ .

An implementation is

```

1  template <typename Type>
2  void random_permute(Type *f, ulong n)
3  // Randomly permute the elements of f[].
4  {
5      for (ulong k=n; k>1; --k)
6          {
7              const ulong i = rand_idx(k);
8              swap2(f[k-1], f[i]);
9          }
10 }
```

For the routine corresponding to Algorithm 3.3, replace the loop by

```

1      for (ulong k=1; k<n; ++k)
2      {
3          const ulong i = rand_idx(k+1);
4          swap2(f[k], f[i]);
5      }
```

A Routine for applying a permutation in-place to an array  $F$  (i.e. avoiding a temporary copy of  $F$ ) is given in [2, sect. 2.4].

## 3.2 Permutations of a multiset

The algorithms can also be used to generate a unbiased random permutation of a multiset, as ignoring the order between certain elements corresponds to a uniform projection. Consider a multiset with  $k$  kinds of elements and  $m_j$  elements of kind  $j$  where  $1 \leq j \leq k$ . In any permutation the  $m_j$  elements of kind  $j$  occupy  $m_j$  distinct positions. If these elements were distinguishable there would be  $m_j!$  ways to arrange them. As we cannot distinguish the elements of one type, there are  $m_j!$  permutations that correspond to each permutation of the multiset. The same is true for all kinds of elements, so there are  $\frac{(m_1+m_2+\dots+m_k)!}{m_1!m_2!\dots m_k!}$  permutations of the multiset, which is well known.

Another way to see this is by the following (unnecessarily complicated) algorithm:  
**Algorithm 3.5 (MSetPerm).** *Generate a random permutation of a multiset with  $k$  kinds of elements and  $m_j$  elements of kind  $j$  where  $1 \leq j \leq k$ .*

1. Set  $n := \sum_{j=1}^k m_j$ . Generate a random permutation  $\in \mathcal{S}_n$ .

2. Replace the  $m_1$  smallest elements with kind 0, the  $m_2$  smallest of the remaining elements with kind 1, and so on.

The last step is a uniform projection with quotient  $m_1! m_2! \cdots m_k!$ .  $\square$

### 3.3 Prefix of a permutation

If in a random permutation of a length- $n$  vector  $F$  only the first  $m$  elements (the  $m$ -prefix of  $F$ ) are of interest, then  $O(m)$  operations suffice [22]:

**Algorithm 3.6** (PrefM). *Compute  $m$ -prefix where  $1 \leq m \leq n - 1$  of a random permutation of the elements in  $F = \{f(0), f(1), \dots, f(n - 1)\}$ .*

1. For  $k := 0, 1, \dots, m - 1$  do:  
 Set  $i := k + \mathbf{Z}(n - k)$  ( $i \in \{k, k + 1, \dots, n - 1\}$ ). Swap  $f(i)$  with  $f(k)$ .

The element  $f(0)$  is randomly selected from all  $n$  elements,  $f(1)$  is selected from all  $n - 1$  remaining elements, and so on.  $\square$

If  $m = n - 1$  we simply obtain a random permutation of all elements.

### 3.4 Permutation modulo cyclic shifts

If we consider all cyclic shifts (see section 2.10 on page 9) of a permutation as equivalent, we can use the convention of always having the largest element last. Then a random permutation (modulo the implied equivalence) in array form is obtained by randomly permuting all but the last element.

**Algorithm 3.7** (PermCS). *Generate a random permutation modulo cyclic shifts.*

1. Set  $P := 1_n$ .
2. For  $k := n - 1, n - 2, \dots, 2$  do:  
 Set  $i := \mathbf{Z}(k)$  ( $i \in \{0, 1, \dots, k - 1\}$ ). Swap  $p(i)$  with  $p(k - 1)$ .

The algorithm is obtained by omitting the first step ( $k = n$ ) in the loop of Algorithm 3.2.

### 3.5 Cyclic permutations

A permutation  $P \in \mathcal{S}_n$  that consists of a single cycle of length  $n$  is called a *cyclic permutation*. Generating a random cyclic permutation is no harder than generating an arbitrary permutation, we just have to make sure that a swap happens in every step:

**Algorithm 3.8** (CyclicS). *Generate a random cyclic permutation.*

1. Set  $S := \{1, 2, \dots, n - 1\}$ . Set  $C := (0)$  (a length-1 cycle).

2. Until  $S$  is empty, do:
  - Select a random element  $s \in S$  and remove it from  $S$ .
  - Append  $s$  to the cycle  $C$ .

The algorithm is usually given in this form:

**Algorithm 3.9** (Cyclic1). *Generate a random cyclic permutation.*

1. Set  $P := 1_n$ .
2. For  $k := n - 1, n - 2, \dots, 1$  do:
  - Set  $i := \mathbf{Z}(k)$  ( $i \in \{0, 1, \dots, k - 1\}$ ). Swap  $p(i)$  with  $p(k)$ .

This variant is called *Sattolo's algorithm*, for a proof of its validity see [25].

We now give a slightly different form which is easy to prove correct. The basic ingredient is the following: Let  $p(i)$  and  $p(k)$  be elements of different cycles of a permutation  $P$  of lengths  $l_i$  and  $l_k$ , respectively. Let  $T$  be the transposition of the positions  $i$  and  $k$ . Then  $Q := TP$  has a cycle of length  $l_i + l_k$  that contains both  $p(i)$  and  $p(k)$ . That is, a swap of two elements in different cycles joins the cycles.

**Algorithm 3.10** (Cyclic0). *Generate a random cyclic permutation.*

1. Set  $P := 1_n$ .
2. For  $k := 1, 2, \dots, n - 1$  do:
  - Set  $i := \mathbf{Z}(k)$  ( $i \in \{0, 1, \dots, k - 1\}$ ). Swap  $p(i)$  with  $p(k)$ .

Only cyclic permutations are generated: Step  $k$  in the loop joins  $p(k) = k$  (a fixed point) with the length- $k$  cycle given by  $p(0), p(1), \dots, p(k - 1)$ . Thereby after step  $k$  the permutation contains a single cycle consisting of the first  $k + 1$  elements  $0, 1, \dots, k$ . All cyclic permutation are generated: At step  $k$  the random number  $\mathbf{Z}(k)$  can have  $k$  different values, each creating a different  $(k + 1)$ -cycle. So there are  $1 \cdot 2 \cdot 3 \cdots (n - 1) = (n - 1)!$  different permutations the method can generate, and there are  $(n - 1)!$  cyclic permutations altogether.  $\square$

The following implementation corresponds to Algorithm 3.10, it permutes the elements of an array  $F$ :

```

1  template <typename Type>
2  void random_permute_cyclic(Type *f, ulong n)
3  // Permute the elements of f by a random cyclic permutation.
4  {
5      for (ulong k=1; k<n; ++k)
6          {
7              const ulong i = rand_idx(k);
8              swap2(f[k], f[i]);
9          }
10 }
```

The routine corresponding to Algorithm 3.9 is obtained by replacing the loop with

```

1      for (ulong k=n-1; k>0; --k)
2      {
3          const ulong i = rand_idx(k);
4          swap2(f[k], f[i]);
5      }
```

To compute a cyclic permutation, use

```

1  inline void random_cyclic_permutation(ulong *f, ulong n)
```

```

2 // Create a random permutation that is cyclic.
3 {
4     for (ulong k=0; k<n; ++k) f[k] = k;
5     random_permute_cyclic(f, n);
6 }

```

The normalized cycle form of a random cyclic permutation can of course be computed by randomly permuting all elements except the first (or the last, as in Algorithm 3.7). If the cycle is not required to be normalized, then also a random permutation of all elements can be used.

The following algorithm applies a cyclic permutation to a subset of  $F$ .

**Algorithm 3.11** (Cyclic-Positions1). *Apply a random cyclic permutation to those  $n_p \geq 2$  elements in  $F = [f(0), f(1), \dots, f(n-1)]$  whose positions are given in  $P_s = [p_s(0), p_s(1), \dots, p_s(n_p-1)]$ .*

1. Set  $n_r := n_p$  (number of remaining elements).
2. Set  $i_0 := 0$  (cycle leader, no need to take random position).
3. Set  $n_r := n_r - 1$ . If  $n_r = 0$  then return.
4. Set  $i_1 := \mathbf{Z}(n_r)$  ( $0 \leq i_1 < n_r$ ). Swap  $f(p_s(i_0))$  with  $f(p_s(i_1))$  (extend cycle).
5. Swap  $p_s(i_0)$  with  $p_s(n_r)$  (remove  $p_s(i_0)$  from set).
6. Set  $i_0 := i_1$  and go to step 3.

The removal of elements from the set can be avoided as follows.

**Algorithm 3.12** (Cyclic-Positions0). *Apply a random cyclic permutation to those  $n_p \geq 2$  elements in  $F = [f(0), f(1), \dots, f(n-1)]$  whose positions are given in  $P_s = [p_s(0), p_s(1), \dots, p_s(n_p-1)]$ .*

1. Set  $k := 0$ .
2. Set  $k := k + 1$ . If  $k = n_p$  then return.
3. Set  $i := \mathbf{Z}(k)$  ( $0 \leq i < k$ ). Swap  $f(p_s(k))$  with  $f(p_s(i))$  (extend cycle).
4. Go to step 2.

An implementation is

```

1 template <typename Type>
2 void random_permute_positions_cyclic(Type *f, ulong np, const ulong *ps)
3 // Randomly permute the np elements f[ps[0]], f[ps[1]], ..., f[ps[np-1]]
4 // by a cyclic permutation.
5 {
6     for (ulong k=1; k<np; ++k)
7     {
8         const ulong i = rand_idx(k);
9         swap2(f[ps[k]], f[ps[i]]);
10    }
11 }

```

For the arbitrary and cyclic permutations we gave versions of the algorithms (Algorithm 3.1 and Algorithm 3.8) that involve manipulations of a set. However, the practical versions did not need these operations, so Algorithm 3.1 and Algorithm

3.8 might appear unnecessarily complicated. We will see that for other types of permutations we usually *do* need to keep track of the set of unprocessed elements.

### 3.6 Permutations with given parity

To generate a permutation with prescribed parity we modify Algorithm 3.3 to keep track of the parity of the permutation generated: whenever two distinct elements are swapped the parity changes. If the generated permutation does not have the right parity we swap the first two elements of it.

**Algorithm 3.13** (PermPar). *Generate a random permutation  $P \in \mathcal{S}_n$  ( $n \geq 2$ ) with prescribed parity  $e$ .*

1. Set  $P := 1_n$ . Set  $g := 0$  (parity of the generated permutation).
2. For  $k := 1, 2, \dots, n - 1$  do:  
     Set  $i := \mathbf{Z}(k + 1)$  ( $i \in \{0, 1, \dots, k\}$ ).  
     If  $i \neq k$  then swap  $p(i)$  with  $p(k)$  and set  $g := 1 - g$  (parity changed).
3. If  $g \neq e$  then swap  $p(0)$  with  $p(1)$  (fix parity via a transposition  $T$ ).

The last step is a uniform projection with quotient 2, it corresponds to a multiplication (by a transposition) in the group  $\mathcal{S}_n$ . The transposition  $T$  in the last step is odd. We either obtain one of the  $n!/2$  permutations with required parity  $e$ , and return it, or the permutation has the wrong parity and composition with  $T$  changes the parity to  $e$ . As a map between the even and odd permutations,  $T$  is a bijection.  $\square$

The algorithm can be used to generate a permutation where the parity of the number  $m$  of cycles is prescribed. With  $t$  for the number of transpositions we have  $n = t + m$ , by relation (2.20). If  $n$  is even then  $m$  and  $t$  are simultaneously even or odd, else  $m$  is even if and only if  $t$  is odd.

There seems to be no easy way to generalize the method to generate permutations satisfying other modulo conditions for their number of transpositions, as the distributions are nonuniform.

### 3.7 Permutations modulo reversal

A natural dual of Algorithm 3.13 is the following.

**Algorithm 3.14** (PermRev). *Generate a random permutation  $P \in \mathcal{S}_n$  ( $n \geq 2$ ) where  $0 \prec 1$ .*

1. Generate a random permutation  $P$ .
2. Find  $i_0$  and  $i_1$  such that  $p(i_0) = 0$  and  $p(i_1) = 1$ , respectively.
3. If  $i_0 > i_1$  then swap  $p(i_0)$  with  $p(i_1)$  (fix order).

The last step is a uniform projection with quotient 2.  $\square$

An implementation is

```
1  inline void random_ord01_permutation(ulong *p, ulong n)
2  // Random permutation such that elements 0 and 1 are in order.
3  {
4      random_permutation(p, n);
5      ulong t = 0;
6      while ( p[t]>1 ) ++t;
7      if ( p[t]==0 ) return; // already in correct order
8      p[t] = 0;
9      do { ++t; } while ( p[t]!=0 );
10     p[t] = 1;
11 }
```





## Chapter 4

# Putting elements into cycles

We give algorithms for generating random permutations with conditions on which elements lie in the same or in distinct cycles. The central ingredients are the bijection between array form and canonical cycle form, and uniform projections by sorting certain elements.

### 4.1 Conversion between array form and CCF

For the following algorithms we need a routine for the conversion to the canonical cycle form (CCF). The variables  $l_c$  and  $n_c$  are scalars and respectively correspond to `lc` and `nc` in the implementation.

**Algorithm 4.1** (Perm2CCF). *Convert permutation  $P \in S_n$  given in array form to canonical cycle form (written to length- $n$  array  $C$ ).*

1. Set  $B := [0, 0, \dots, 0]$  (length  $n$ , tag array for marking elements as processed).
2. Set  $m := 0$  (minimum value of unprocessed elements).
3. Set  $j := 0$  (position in  $C$ ).
4. Set  $l_c := m$  (next cycle;  $m$  is last element in cycle to be processed).
5. Set  $n_c := p(l_c)$  (next element in cycle).
6. Set  $C(j) := n_c$  and  $j := j + 1$ .
7. Set  $l_c := n_c$  and  $B(l_c) := 1$  (mark  $l_c$  as processed).
8. If  $l_c \neq m$  then go to step 5.
9. (Find smallest unprocessed element:)  
     for  $i := m + 1 \dots n - 1$  do: if  $B(i) = 0$  then set  $m := i$  and go to step 4.

The time complexity is  $O(n)$  which is optimal.

The implementation uses a bit-array (described in appendix A on page 77):

```
void
perm2ccf(const ulong *p, ulong n, ulong *c, bitarray *tb/**=0*/)
// Convert permutation in p[] (array form) into
// canonical cycle form (CCF), written to c[].
{
    bitarray * b = tb;
```

```

if ( 0==tb ) b = new bitarray(n);
b->clear_all();
ulong m = 0; // minimum of unprocessed elements (==cycle end)
ulong j = 0; // position in c[]
while ( m!=n )
{
    ulong lc = m; // last in cycle
    do
    {
        ulong nc = p[lc]; // next in cycle
        c[j] = nc;
        ++j;
        lc = nc;
        b->set(lc); // mark as processed
    }
    while ( lc!=m );
    m = b->next_clear(m+1);
}
if ( 0==tb ) delete b;
}

```

The algorithm for the other direction also has complexity  $O(n)$ :

**Algorithm 4.2 (CCF2Perm).** *Convert permutation  $\in \mathcal{S}_n$  given in canonical cycle form (as length- $n$  array  $C$ ) to array form (written to array  $P$ ).*

1. Set  $B := [0, 0, \dots, 0]$  (length  $n$ , tag array for marking elements as processed).
2. Set  $m := 0$  (minimum value of unprocessed elements).
3. Set  $j := 0$  (position in  $C$ ).
4. Set  $l_c := m$  (next cycle;  $m$  is last element in cycle to be processed).
5. Set  $n_c := C(j)$  (next element in cycle) and  $j := j + 1$ .
6. Set  $P(l_c) := n_c$ .
7. Set  $l_c := n_c$  and  $B(l_c) := 1$  (mark  $l_c$  as processed).
8. If  $l_c \neq m$  then go to step 5.
9. (Find smallest unprocessed element:)
  - for  $j := m + 1, \dots, n - 1$  do: if  $B(j) = 0$  then set  $m := j$  and go to step 4.

An implementation is

```

1 void
2 ccf2perm(const ulong *c, ulong n, ulong *p, bitarray *tb/**=0*/)
3 // Convert permutation in canonical cycle form (CCF) in c[] into
4 // array form, written to p[].
5 {
6     bitarray * b = tb;
7     if ( 0==tb ) b = new bitarray(n);
8     b->clear_all();
9
10    ulong m = 0; // minimum of unprocessed elements (==cycle end)
11    ulong j = 0; // position in c[]
12    while ( j!=n )
13    {
14        ulong lc = m; // last in cycle
15        do
16        {
17            ulong nc = c[j]; // next in cycle

```

```

18         ++j;
19         p[lc] = nc;
20         lc = nc;
21         b->set(lc); // mark as processed
22     }
23     while ( lc!=m ); // until cycle end
24     m = b->next_clear(m+1); // ==n if no clear bit present
25 }
26
27 if ( 0==tb ) delete b;
28 }

```

## 4.2 Prescribed elements in one cycle

An easy extension of Algorithm 3.14 is to let all elements  $\in \{1, \dots, k-1\}$  precede 0.

**Algorithm 4.3** (InOneCycle). *Generate a random permutation  $P \in \mathcal{S}_n$  such that  $1, 2, \dots, k-1$  all precede 0.*

1. Generate a random permutation  $P$ .
2. Find  $i_0$  and  $i_l$  such that  $p(i_0) = 0$  and  $i_l$  is the position of the last element  $\in \{0, 1, \dots, k-1\}$ .
3. Swap  $p(i_0)$  with  $p(i_l)$ .

Step 3 is a uniform projection with quotient  $k$ . □

Interpreted as canonical cycle form,  $P$  has a cycle containing all elements  $0, 1, \dots, k-1$  (all elements preceding 0 are in the same cycle as 0 and we guaranteed that  $1, 2, \dots, k-1$  all precede 0).

An implementation is

```

1  inline void random_lastk_permutation(ulong *p, ulong n, ulong k)
2  // Random permutation such that 0 appears as last of the k smallest elements.
3  // Must have k<=n.
4  {
5      random_permutation(p, n);
6      if ( k<=1 ) return;
7
8      ulong p0=0, pl=0; // position of 0, and last (in k smallest elements)
9      for (ulong t=0, j=0; j<k; ++t)
10     {
11         if ( p[t]<k )
12         {
13             pl = t; // update position of last
14             if ( p[t]==0 ) { p0 = t; } // record position of 0
15             ++j; // j out of k smallest found
16         }
17     }
18     // here t is the position of the last of the k smallest elements
19     swap2( p[p0], p[pl] );
20 }

```

### 4.3 Prescribed elements in distinct cycles

Another extension of Algorithm 3.14 is to fix the order of the elements  $0, 1, \dots, k-1$ . **Algorithm 4.4** (InDistinctCycles). *Generate a random permutation  $P \in S_n$  such that  $0 < 1 < 2 < \dots < k-1$ .*

1. Generate a random permutation  $P$ .
2. Set  $e := 0$  and  $j := 0$ .
3. While  $e < k$  do: (fix order of elements  $0, 1, \dots, k-1$ )
  - (a) If  $p(j) < k$  then (an element  $< k$  was found)
    - set  $p(j) := e$  (write next element in the sequence  $0, 1, \dots, k-1$ )
    - and set  $e := e + 1$  (next in sequence).
  - (b) Set  $j := j + 1$ .

Step 3 is a uniform projection with quotient  $k!$ . □

The time complexity is  $O(n)$ . Interpreted as canonical cycle form,  $P$  contains  $k$  distinct cycles, each containing exactly one element  $< k$ . An implementation is

```

1  inline void random_ordk_permutation(ulong *p, ulong n, ulong k)
2  // Random permutation such that the k smallest elements are in order.
3  // Must have k<=n.
4  {
5      random_permutation(p, n);
6      for (ulong j=0,e=0; e<k; ++j) if ( p[j]<k ) { p[j]=e; ++e; }
7  }
```

### 4.4 Prescribed sets in distinct cycles

	CCF	cycles
1:	[ 1 2 0 4 3 5 ]	(1, 2, 0) (4, 3) (5)
2:	[ 1 2 0 4 5 3 ]	(1, 2, 0) (4, 5, 3)
3:	[ 1 2 0 5 4 3 ]	(1, 2, 0) (5, 4, 3)
4:	[ 1 2 5 0 4 3 ]	(1, 2, 5, 0) (4, 3)
5:	[ 1 5 2 0 4 3 ]	(1, 5, 2, 0) (4, 3)
6:	[ 2 1 0 4 3 5 ]	(2, 1, 0) (4, 3) (5)
7:	[ 2 1 0 4 5 3 ]	(2, 1, 0) (4, 5, 3)
8:	[ 2 1 0 5 4 3 ]	(2, 1, 0) (5, 4, 3)
9:	[ 2 1 5 0 4 3 ]	(2, 1, 5, 0) (4, 3)
10:	[ 2 5 1 0 4 3 ]	(2, 5, 1, 0) (4, 3)
11:	[ 5 1 2 0 4 3 ]	(5, 1, 2, 0) (4, 3)
12:	[ 5 2 1 0 4 3 ]	(5, 2, 1, 0) (4, 3)

**Figure 4.1:** All permutations of length 6 elements where the elements 0, 1, and 2 lie in the same cycle  $C_0$  and the elements 3 and 4 lie in the same cycle  $C_1 \neq C_0$ , in canonical cycle form (left) and cycle form (right).

Let  $D = [d(0), d(1), \dots, d(u-1)]$ . We give an algorithm to generate a random permutation where  $d(0)$  prescribed elements are in a cycle  $C_0$ ,  $d(1)$  prescribed elements are in a cycle  $C_1 \neq C_0$ , and so on ( $u$  distinct cycles). Without loss of generality we



```

3                                     ulong *tv=0)
4 // sdc := Sets into Distinct Cycles.
5 //
6 // Let NN={0,1,...,n-1},
7 // S0 be the set of the d[0] smallest elements of NN,
8 // S1 be the set of the d[1] smallest elements of NN \ S0
9 // S2 be the set of the d[2] smallest elements of NN \ { S0 union S1 }
10 // and so on.
11 // Let m0 = min(S0), m1 = min(S1) etc.,
12 // write a<<b for "a precedes b".
13 //
14 // Generate random permutation such that
15 // x << m0 for all elements (!=m0) of S0,
16 // m0 << x << m1 for all elements (!=m1) of S1,
17 // m1 << x << m2 for all elements (!=m2) of S2,
18 // and so on.
19 //
20 // As canonical cycle form (CCF):
21 // The elements of S0 are in one cycle C0,
22 // the elements of S1 are in one cycle C1!=C0,
23 // the elements of S2 are in one cycle C2!=C1, C2!=C0,
24 // and so on.
25 {
26     ulong w = 0; // number of elements specified via d[]
27     for (ulong k=0; k<nd; ++k) w += d[k];
28
29     ulong *v = tv;
30     if ( tv==0 ) v = new ulong[w];
31     for (ulong k=0; k<w; ++k) v[k] = k;
32
33     ulong e = 0, j = 0;
34     while ( e<nd )
35     {
36         const ulong de = d[e];
37         swap2( v[j], v[j+de-1] ); // cycle end
38         if ( de>2 ) random_permute( v+j, de-1 ); // make cycle random
39         j += de; ++e;
40     }
41
42     random_permutation(p, n);
43
44     e = 0; j = 0;
45     while ( e<w ) // sort w smallest elements in p[] according to v[]
46     {
47         if ( p[j]<w ) { p[j]=v[e]; ++e; }
48         ++j;
49     }
50
51     if ( tv==0 ) delete [] v;
52 }

```

## Chapter 5

# Given cycle type

We introduce the techniques for maintaining the set of unprocessed elements and for creating random cycles. These are used in many of the algorithms that follow in this thesis.

### 5.1 Maintenance of the set

For most of the algorithms we need to keep track of a set  $S$  of the positions of the elements not processed so far.

The following two operations should be efficient: unbiased random selection of an element  $s \in S$  and removal of an element  $s$  from the set.

We represent the set  $S$  as an array  $R = [R(0), R(1), R(2), \dots]$ , initially containing all positions. During processing the elements still  $\in S$  will be in positions  $0, 1, \dots, r-1$  where  $r = |S|$ .

To randomly select an element  $s$ , set  $s := R_i$  where  $i = \mathbf{Z}(r) \in \{0, 1, \dots, r-1\}$ . To remove the element  $s = R(i)$ , swap  $R(i)$  with  $R(r-1)$  and set  $r := r-1$ . Both operations are obviously  $O(1)$  which is optimal. This technique is given in [8].

The order of the elements is in general lost during processing. This, however, does not interfere with the operations just described.

### 5.2 One cycle of prescribed length

We first give an auxiliary algorithm to create a cycle. It is used in the algorithms that appear in this chapter.

The subscripted variables  $k_0$  and  $k_p$  are scalars. They respectively correspond to the variables  $k_0$  and  $k_p$  in the implementation.

**Algorithm 5.1** (Cycle). *Let  $P$  be a permutation with  $r \geq 1$  fixed points. Let  $R = [R(0), R(1), R(2), \dots]$  be an array whose first  $r$  elements contain the positions of the unprocessed elements of  $P$ . Generate a random length- $c$  cycle ( $1 \leq c \leq r$ ) whose elements are a subset of the unprocessed elements in  $P$ .*

1. Select cycle leader from unprocessed elements:

- (a) Set  $i := Z(r)$ , set  $k_0 := R(i)$  ( $k_0$  is the cycle leader).
- (b) Set  $r := r - 1$  and swap  $R(i)$  with  $R(r)$  (remove  $R(i)$  from set).
2. Set  $k_p := k_0$  (predecessor in cycle).
3. Set  $c := c - 1$  (number of elements to process).
4. While  $c \neq 0$  repeat: (append random element to cycle)
  - (a) Set  $i := Z(r)$ , set  $k := R(i)$  ( $k$  will be appended to the cycle).
  - (b) Set  $r := r - 1$  and swap  $R(i)$  with  $R(r)$  (remove  $R(i)$  from set).
  - (c) Set  $p(k_p) := p(k)$  ( $= k$ , append to cycle).
  - (d) Set  $k_p := k$  (update predecessor).
  - (e) Set  $c := c - 1$ .
5. Set  $p(k_p) := p(k_0)$  ( $= k_0$ , close cycle).

That a length- $c$  cycle is created is clear from construction. Each cycle is generated equally often: the set of elements it contains is sampled uniformly and the order of these  $c$  elements is a random permutation of them. Thereby any particular cycle can be generated in  $c!/(c-1)! = c$  ways.  $\square$

Note that when  $c = 1$  (a length-1 cycle is a fixed point) an element of  $R$  is removed but  $P$  is not modified.

The algorithm is written in a way (steps 4c and 5) that allows for easy modification to a slightly more general routine as follows. The variable `nr` corresponds to  $r$  in the implementation:

```

1  template <typename Type>
2  inline ulong random_cycle(Type *f, ulong cl, ulong *R, ulong nr)
3  // Permute a random subset (of size cl)
4  // of those elements in f whose positions are given in
5  // R[0], ..., R[nr-1] by a random cycle of size cl.
6  // Must have nr >= cl and cl != 0.
7  {
8      if ( cl==1 ) // just remove a random position from R[]
9      {
10         const ulong i = rand_idx(nr);
11         --nr; swap2( R[nr], R[i] ); // remove position from set
12     }
13     else // cl >= 2
14     {
15         const ulong i0 = rand_idx(nr);
16         const ulong k0 = R[i0]; // position of cycle leader
17         const Type f0 = f[k0]; // cycle leader
18         --cl;
19         --nr; swap2( R[nr], R[i0] ); // remove position from set
20
21         ulong kp = k0; // position of predecessor in cycle
22         do // create cycle
23         {
24             const ulong i = rand_idx(nr);
25             const ulong k = R[i]; // random available position
26             f[kp] = f[k]; // move element
27             --nr; swap2( R[nr], R[i] ); // remove position from set
28             kp = k; // update predecessor
29         }

```



```

30         while ( --c1 );
31
32         f[kp] = f0; // close cycle
33     }
34
35     return nr;
36 }

```

We give an example for creating a 3-cycle in the array  $f=[A,B,C,D,E]$  where none of the elements have been processed so far. The random numbers shall be chosen as 1, 1(again), and 0:

```

      R[]          f[]
[0,1,2,3,4; ]    [A,B,C,D,E] (start)

```

Choose cycle leader:

```

[0,4,2,3; 1]    i==1 (R[1]==1 swapped out), f0==B==f[1] is cycle leader
  x             x

```

Loop step 1:

```

[0,3,2; 4,1]    i==1 (R[1]==4 swapped out), now replace f[1] with E==f[4]
  x           x    [A,E,C,D,E]

```

Loop step 2:

```

[2,3; 0,4,1]    i==0 (R[0]==0 swapped out), now replace f[4] with A==f[0]
  x           x    [A,E,C,D,A]

```

Close cycle: replace  $f[0]==A$  with leader  $f0==B$   

```

      [B,E,C,D,A] (result)
      x

```

The cycle created is  $(A, B, E)$ . The positions of the unprocessed elements  $C$  and  $D$  are now the first two elements in the array  $R[]$ .

### 5.3 Permutations with given cycle type

To generate a random permutation of prescribed cycle type we start with the identical permutation and successively create cycles of the required lengths.

**Algorithm 5.2** (CycleType). *Generate a random permutation  $P \in S_n$  with prescribed cycle type  $C = [c_1, c_2, \dots, c_n]$ .*

1. Set  $P := 1_n$  (identical permutation).
2. Set  $R := [0, 1, \dots, n-1]$ , set  $r := n$  (initialize set).
3. For  $j := 1, 2, \dots, n$  do:  
 Create  $c_j$  cycles of length  $j$  in  $P$  by using Algorithm 5.1  $c_j$  times.

The algorithm has complexity  $O(n)$ .

The algorithm could be slightly simplified because  $p(k) = k$  whenever  $p(k)$  is a fixed point (as required). However, as given, it can be used to randomly permute arbitrary data as follows.

**Algorithm 5.3** (CycleTypeF). *Apply a random permutation with prescribed cycle type  $C = [c_1, c_2, \dots, c_n]$  to the elements in  $F = [f(0), f(1), \dots, f(n-1)]$ .*

1. Set  $R := [0, 1, \dots, n-1]$ , set  $r := n$  (initialize set).

2. For  $j := 1, 2, \dots, n$  do:

    Create  $c_j$  cycles of length  $j$  in  $F$  by using Algorithm 5.1  $c_j$  times.

An implementation is

```

1  template <typename Type>
2  inline void
3  random_permute_cycle_type(Type *f, ulong n,
4                          const ulong *c,
5                          ulong *tr=0)
6  // Permute the elements of f by a random permutation of
7  // prescribed cycle type.
8  // The permutation will have c[k] cycles of length k+1.
9  // Must have s <= n where s := sum(k=0, n-1, c[k]).
10 // If s < n then the permutation will have n-s fixed points.
11 // Complexity O(n).
12 {
13     ulong *r = tr;
14     if ( tr==0 ) r = new ulong[n];
15     for (ulong k=0; k<n; ++k) r[k] = k; // initialize set
16     ulong nr = n; // number of elements available
17     // available positions are r[0], ..., r[nr-1]
18
19     for (ulong k=0; k<n; ++k)
20     {
21         ulong nc = c[k]; // number of cycles of length k+1;
22         if ( nc==0 ) continue; // no cycles of this length
23         const ulong cl = k+1; // cycle length
24         do
25         {
26             nr = random_cycle(f, cl, r, nr);
27         }
28         while ( --nc );
29     }
30
31     if ( tr==0 ) delete [] r;
32 }

```

A permutation of given cycle type is computed by permuting  $F = [0, 1, 2, \dots, n-1]$ :

```

1  inline void random_cycle_type_permutation(ulong *p, ulong n,
2                                          const ulong *c,
3                                          ulong *tr=0)
4  {
5      for (ulong k=0; k<n; ++k) p[k] = k;
6      random_permute_cycle_type(p, n, c, tr);
7  }

```

With only a small number of distinct cycle lengths  $l_k$  there is a more compact way to specify the cycle type as a partition

$$n = \sum_{k=1}^N m_k l_k \quad (5.1)$$

The two arrays  $M = [m_1, m_2, \dots, m_N]$  and  $L = [l_1, l_2, \dots, l_N]$  are used in the following algorithm to specify the partition.

**Algorithm 5.4** (CycleTypePart). *Generate a random permutation  $P \in S_n$  with cycle type given as partition  $n = \sum m_k l_k$  specified by the arrays  $M$  and  $L$ : create  $m_1$  cycles of length  $l_1$ ,  $m_2$  cycles of length  $l_2$ , and so on.*

1. Set  $P := 1_n$  (identical permutation).

2. Set  $R := [0, 1, \dots, n - 1]$ , set  $r := n$  (initialize set).
3. For  $j := 1, 2, \dots, N$  do:  
 Create  $m_j$  cycles of length  $l_j$  in  $P$  by using Algorithm 5.1  $m_j$  times.

An implementation is

```

1  template <typename Type>
2  inline void random_permute_cycle_type(Type *f, ulong n,
3                                     const ulong *m, ulong nm,
4                                     const ulong *len,
5                                     ulong *tr=0)
6  // Permute the elements of f by a random permutation of
7  // prescribed cycle type given as partition:
8  // The permutation will have m[k] cycles of length len[k]
9  // where k = 0,1,...,nm-1.
10 // Must have s <= n where s := sum(k=0, nm-1, m[k]*len[k]).
11 // If s < n then the permutation will have n-s fixed points.
12 {
13     ulong *r = tr;
14     if ( tr==0 ) r = new ulong[n];
15     for (ulong k=0; k<n; ++k) r[k] = k; // initialize set
16     ulong nr = n; // number of elements available
17     // available positions are r[0], ..., r[nr-1]
18
19     for (ulong k=0; k<nm; ++k)
20     {
21         ulong nc = m[k]; // number of cycles of length len[k];
22         if ( nc==0 ) continue; // no cycles of this length
23         const ulong cl = len[k]; // cycle length
24         do
25         {
26             nr = random_cycle(f, cl, r, nr);
27         }
28         while ( --nc );
29     }
30     if ( tr==0 ) delete [] r;
31 }
32

```



## Chapter 6

# Number of cycles

We give algorithms to generate random permutations with conditions on the number of cycles.

### 6.1 Bijection via swaps

Let the vector  $F = [f(1), f(2), \dots, f(n-1)]$  be a mixed radix number in rising factorial base, that is,  $0 \leq f(j) \leq j$  (there are  $j+1$  choices for the digit  $f(j)$ ). The following algorithm generates a permutation of  $n$  elements from a given  $F$  with  $n-1$  elements.

**Algorithm 6.1** (RFact2PermSwp). *Convert the mixed radix number in rising factorial base given as vector  $F$  into a permutation  $P \in \mathcal{S}_n$ :*

1. Set  $P := 1_n$ .
2. For  $k := 1, 2, \dots, n-1$  do:  
    Set  $i := f(k)$  ( $i \in \{0, 1, \dots, k\}$ ). Swap  $p(k-i)$  with  $p(k)$ .

The algorithm is obviously  $O(n)$ . We could have used a swap of  $p(i)$  with  $p(k)$  in the last step, but in the form given the all-zeros word in  $F$  corresponds to the identical permutation.

Each of the  $2 \cdot 3 \cdot 4 \cdots n = n!$  numbers  $F$  corresponds to a unique permutation. To see this, we generate a random permutation in two steps:

**Algorithm 6.2** (PermSwaps). *Generate a random permutation  $P \in \mathcal{S}_n$ :*

1. For  $k := 1, 2, \dots, n-1$  do: Set  $i := \mathbf{Z}(k+1)$  ( $i \in \{0, 1, \dots, k\}$ ), set  $f(k) := i$ .
2. Convert  $F$  into a permutation via Algorithm 6.1.

The vector  $F = [f(1), f(2), \dots, f(n-1)]$  created in step 1 is a random mixed radix number in rising factorial base.

We can improve Algorithm 6.1 as follows. In step 2 we have  $p(k) = k$ , so the swap can be replaced by the assignments  $p(k) := p(i)$  and  $p(i) := k$ . Then step 1 can be replaced by  $p(0) := 0$ , saving  $O(n)$  writes to memory. Now the arrays  $F$  and  $P$  can share the same storage, as follows.

**Algorithm 6.3** (RFact2PermSwpX). Convert the mixed radix number in rising factorial base, given in  $x(1), x(2), \dots, x(n-1)$  into a permutation  $P \in S_n$  written to  $x(0), x(1), x(2), \dots, x(n-1)$ :

1. Set  $x(0) := 0$ .
2. For  $k := 1, 2, \dots, n-1$  do:  
Set  $i := x(k)$  ( $i \in \{0, 1, \dots, k\}$ ). Set  $x(k) := x(i)$  and  $x(i) := k$ .

An implementation of Algorithm 6.1 is

```

1 void
2 rfact2perm_swp_l2r(const ulong *fc, ulong n, ulong *x)
3 {
4     for (ulong k=0; k<n; ++k) x[k] = k;
5     for (ulong k=1; k<n; ++k)
6     {
7         ulong i = fc[k-1]; // 0<=i<=k
8         swap2( x[k-i], x[k] );
9     }
10 }
```

The routine proceeds from left to right (thus the suffix `_l2r`): at each step of the loop only elements of the prefix  $x[0], x[1], \dots, x[k]$  are (in general) reordered.

	fact. num.	left to right	right to left
0:	[ . . . ]	[ . 1 2 3 ] (0) (1) (2) (3)	[ . 1 2 3 ] (0) (1) (2) (3)
1:	[ 1 . . ]	[ 1 . 2 3 ] (0, 1) (2) (3)	[ . 1 3 2 ] (0) (1) (2, 3)
2:	[ . 1 . ]	[ . 2 1 3 ] (0) (1, 2) (3)	[ . 2 1 3 ] (0) (1, 2) (3)
3:	[ 1 1 . ]	[ 1 2 . 3 ] (0, 1, 2) (3)	[ . 3 1 2 ] (0) (1, 3, 2)
4:	[ . 2 . ]	[ 2 1 . 3 ] (0, 2) (1) (3)	[ . 3 2 1 ] (0) (1, 3) (2)
5:	[ 1 2 . ]	[ 2 . 1 3 ] (0, 2, 1) (3)	[ . 2 3 1 ] (0) (1, 2, 3)
6:	[ . . 1 ]	[ . 1 3 2 ] (0) (1) (2, 3)	[ 1 . 2 3 ] (0, 1) (2) (3)
7:	[ 1 . 1 ]	[ 1 . 3 2 ] (0, 1) (2, 3)	[ 1 . 3 2 ] (0, 1) (2, 3)
8:	[ . 1 1 ]	[ . 2 3 1 ] (0) (1, 2, 3)	[ 2 . 1 3 ] (0, 2, 1) (3)
9:	[ 1 1 1 ]	[ 1 2 3 . ] (0, 1, 2, 3)	[ 3 . 1 2 ] (0, 3, 2, 1)
10:	[ . 2 1 ]	[ 2 1 3 . ] (0, 2, 3) (1)	[ 3 . 2 1 ] (0, 3, 1) (2)
11:	[ 1 2 1 ]	[ 2 . 3 1 ] (0, 2, 3, 1)	[ 2 . 3 1 ] (0, 2, 3, 1)
12:	[ . . 2 ]	[ . 3 2 1 ] (0) (1, 3) (2)	[ 2 1 . 3 ] (0, 2) (1) (3)
13:	[ 1 . 2 ]	[ 1 3 2 . ] (0, 1, 3) (2)	[ 3 1 . 2 ] (0, 3, 2) (1)
14:	[ . 1 2 ]	[ . 3 1 2 ] (0) (1, 3, 2)	[ 1 2 . 3 ] (0, 1, 2) (3)
15:	[ 1 1 2 ]	[ 1 3 . 2 ] (0, 1, 3, 2)	[ 1 3 . 2 ] (0, 1, 3, 2)
16:	[ . 2 2 ]	[ 2 3 . 1 ] (0, 2) (1, 3)	[ 2 3 . 1 ] (0, 2) (1, 3)
17:	[ 1 2 2 ]	[ 2 3 1 . ] (0, 2, 1, 3)	[ 3 2 . 1 ] (0, 3, 1, 2)
18:	[ . . 3 ]	[ 3 1 2 . ] (0, 3) (1) (2)	[ 3 1 2 . ] (0, 3) (1) (2)
19:	[ 1 . 3 ]	[ 3 . 2 1 ] (0, 3, 1) (2)	[ 2 1 3 . ] (0, 2, 3) (1)
20:	[ . 1 3 ]	[ 3 2 1 . ] (0, 3) (1, 2)	[ 3 2 1 . ] (0, 3) (1, 2)
21:	[ 1 1 3 ]	[ 3 2 . 1 ] (0, 3, 1, 2)	[ 2 3 1 . ] (0, 2, 1, 3)
22:	[ . 2 3 ]	[ 3 1 . 2 ] (0, 3, 2) (1)	[ 1 3 2 . ] (0, 1, 3) (2)
23:	[ 1 2 3 ]	[ 3 . 1 2 ] (0, 3, 2, 1)	[ 1 2 3 . ] (0, 1, 2, 3)

**Figure 6.1:** All 3-digit factorial numbers with rising factorial base in lexicographic order and the permutations (in both array and cycle form) obtained by processing from left and right. Dots denote zeros.

We now give both directions of a bijection that is obtained by processing from right to left (suffixes are reordered). Figure 6.1 shows the permutations obtained by processing in either direction. For example, in the second entry in the permutation computed by processing from left to right the first two elements are swapped (prefix) whereas in the permutation computed by processing from right to left the last two

elements are swapped (suffix). Note the permutations in each line have the same cycle type.

The following algorithm generates a permutation of  $n$  elements from a given  $F$  with  $n - 1$  elements, processing suffixes.

**Algorithm 6.4** (RFact2PermSwp-R2L). *Convert the mixed radix number in rising factorial base given as vector  $F$  into a permutation  $P \in \mathcal{S}_n$ :*

1. Set  $P := 1_n$ .
2. For  $k := 1, 2, \dots, n - 2$  do:  
     Set  $i := f(k)$  ( $i \in \{0, 1, \dots, k\}$ ). Set  $j = n - 2 - k$ . Swap  $p(j)$  with  $p(j + i)$ .

Again the algorithm is  $O(n)$ . An implementation is

```

1 void
2 rfact2perm_swp(const ulong *fc, ulong n, ulong *x)
3 {
4     for (ulong k=0; k<n; ++k) x[k] = k;
5     for (ulong k=0, j=n-2; k<n-1; ++k, --j)
6     {
7         ulong i = fc[k];
8         swap2( x[j], x[j+i] );
9     }
10 }
```

The routine for the conversion of a permutation to the corresponding factorial number uses the inverse permutation (the algorithm is given in [16]).

```

1 void
2 perm2rfact_swp(const ulong *x, ulong n, ulong *fc)
3 // Convert permutation in x[0,...,n-1] into
4 // the (n-1) digit (swaps) factorial representation in fc[0,...,n-2].
5 // We have: fc[0]<2, fc[1]<3, ..., fc[n-2]<n (rising radices)
6 {
7     ulong t[n];
8     for (ulong k=0; k<n; ++k) t[k] = x[k];
9     ulong ti[n]; // inverse permutation
10    for (ulong k=0; k<n; ++k) ti[t[k]] = k;
11
12    for (ulong k=0; k<n-1; ++k)
13    {
14        ulong j = ti[k]; // location of element k, j>=k
15        fc[n-2-k] = j - k;
16        ulong tk = t[k]; // >=k
17        ti[tk] = j;
18        t[j] = tk;
19    }
20 }
```

Let  $F$  be a factorial number and  $P$  the corresponding permutation (with respect to Algorithm 6.4). The permutation  $P^{-1}$  can be obtained directly, by reversing the order in the loop (and noting that a swap is its own inverse).

```

void
rfact2invperm_swp(const ulong *fc, ulong n, ulong *x)
// Generate inverse permutation wrt. rfact2perm_swp().
{
    for (ulong k=0; k<n; ++k) x[k] = k;
    if ( n<=1 ) return;
    ulong k = n-2, j=0;
    do
    {
        ulong i = fc[k];
```

```

        swap2( x[j], x[j+i] );
        ++j;
    }
    while ( k-- );
}

```

The explicit computation of the permutation can be avoided if only its action is of interest.

```

1  template <typename Type>
2  void rfact2perm_swp_apply(const ulong *fc, ulong n, Type *x)
3  // Permute x by permutation corresponding to rfact2perm_swp().
4  {
5      for (ulong k=0,j=n-2; k<n-1; ++k,--j)
6          {
7              ulong i = fc[k];
8              swap2( x[j], x[j+i] );
9          }
10 }

```

## 6.2 A property of the bijection

We now fix  $n$  and consider integers  $N$  where  $0 \leq N < n!$ . The factorial representation  $F$  of integers  $N$  where  $0 \leq N < n!$  has at most  $n - 1$  nonzero digits. In what follows we let  $F$  refer to the first  $n - 1$  digits (ignoring infinitely many zeros to the right). See figure 2.2 on page 7 for an example with  $n = 4$  (and  $n - 1 = 3$  digits).

Let  $Z(F)$  be the number of zeros in  $F$  and  $P \in S_n$  the corresponding permutation. We have

**Lemma 6.5.** *The permutation  $P$  consists of  $Z(F) + 1$  disjoint cycles.*

Consider step 2 of Algorithm 6.1. Before the swap we have  $p(j) = j$  for  $j \geq k$  (only fixed points at the right). Now if  $f(k) \neq 0$  then the element  $k$  is joined to one of the cycles at the left. If  $f(k) = 0$  then the element  $k$  lies in its own cycle (initially a fixed point). The cycle containing  $k$  is never joined to any cycle so far generated as the algorithm proceeds: each element  $j > k$  is either joined to the cycle containing  $k$  or to a cycle not containing  $k$ .  $\square$

Note that Algorithm 3.10 on page 14 is obtained by disallowing zeros in  $F$ , a cyclic permutation consists of one cycle so  $Z(F) = 0$ .

Now a permutation into exactly  $m$  cycles can be generated as follows.

**Algorithm 6.6 (NumCycles-BIASED).** *Generate a random permutation  $P \in S_n$  (where  $n \geq m$ ) with exactly  $m$  cycles. Non-uniform(!) sampling.*

1. For  $k := 1, 2, \dots, n - 2$  set  $f(k) := 1 + Z(k)$  (random factorial number  $F$  without zero digits).
2. Choose a random subset of  $m - 1$  elements  $U \subseteq \{0, 1, \dots, n - 2\}$ .
3. For all  $u \in U$  set  $f(u) := 0$  (now  $F$  has  $m - 1$  zeros).
4. Convert  $F$  into a permutation, using Algorithm 6.1.



The distribution of the generated permutations is uniform only for  $m = 1$  (cyclic permutations) and  $m = n$  (identity). To remove the bias, the choice in step 2 has to be made according to the statistics of the permutations.

The simplest cases are  $m = n - 1$  (one transposition, which we omit) and  $m = 2$  which will be treated next.

### 6.3 Exactly 2 cycles

Let  $s(n, m)$  be the Stirling cycle numbers (see section 2.4 on page 5). There are  $s(n, 2)$  permutations with exactly two cycles. In the next algorithm we will generate a factorial number  $F$  with  $n - 1$  digits, all except one nonzero. There are  $(n - 1)!/k$  such  $F$  with digit  $k$  zero. So we find

$$s(n, 2) = (n - 1)! \sum_{k=1}^{n-1} \frac{1}{k} = (n - 1)! \left[ \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-1} \right] \quad (6.1)$$

In the algorithm we use the harmonic numbers  $H_k = \sum_{j=1}^k 1/j$  as (unnormalized, cumulative) probabilities.

The identity can also be obtained from relation (2.16) on page 5 (with  $m = 2$ ):

$$s(n, 2) = s(n - 1, 1) + (n - 1) s(n - 1, 2) \quad (6.2a)$$

$$= (n - 2)! + (n - 1) s(n - 1, 2) \quad (6.2b)$$

Here we used  $s(n, 1) = (n - 1)!$ . So we find

$$s(n, 2) = (n - 1)! \left[ \frac{1}{n-1} + \frac{1}{(n-2)!} s(n-1, 2) \right] \quad (6.2c)$$

Repeated use of this relation gives the desired result:

$$= (n - 1)! \left[ \frac{1}{n-1} + \frac{1}{(n-2)!} (n-2)! \left[ \frac{1}{n-2} + \frac{1}{(n-3)!} s(n-2, 2) \right] \right] \quad (6.2d)$$

$$= (n - 1)! \left[ \frac{1}{n-1} + \frac{1}{n-2} + \frac{1}{(n-3)!} s(n-2, 2) \right] \quad (6.2e)$$

and so on.

In the following algorithm we assume that an array of cumulative probabilities  $b(k) = H_{k+1}$  has been precomputed.

**Algorithm 6.7** (TwoCycles). *Generate a permutation  $P \in \mathcal{S}_n$  (where  $n \geq 1$ ) with exactly 2 cycles.*

1. Set  $t := b(n - 2) \cdot \mathbf{R}()$  (random real number where  $0 \leq t < H_{n-1}$ ).
2. Determine smallest index  $s$  such that  $b(s) \geq t$  (using binary search).
3. For  $j := 1, 2, \dots, n - 1$  set  $f(j) := 1 + \mathbf{Z}(j)$  (factorial number  $F$  without zeros).  
Set  $f(s) := 0$ .

4. Convert  $F$  into a permutation, using Algorithm 6.1.

We now give an algorithm for the precomputation of the cumulative probabilities.

**Algorithm 6.8** (Harmonic). Set  $b(k) = H_{k+1}$  for  $k \in \{0, 1, \dots, n-1\}$ .

1. Set  $h := 1.0$  and  $j := 1.0$  (floating-point types).
2. For  $k := 0, 1, \dots, n-1$  do
  - (a) Set  $b(k) := h$  ( $= H_{k+1}$ );
  - (b) Set  $j := j + 1.0$ , then set  $h := h + 1.0/j$  (update  $h$ ).

This can be implemented as

```

1  inline void init_harmonic(double *b, ulong n)
2  // Set b[k] = sum( j=1, k+1, 1/j )
3  {
4      double h = 1.0, j = 1.0;
5      for (ulong k=0; k<n; ++k) { b[k]=h; j+=1.0; h+=1.0/j; }
6  }
```

In the main routine for Algorithm 6.7 the digits  $f(k)$  are generated as we proceed (instead of being written to an array). Also the array to be permuted is supplied, a permutation is obtained by setting it to the identical permutation before the call.

```

1  template <typename Type>
2  inline void random_permute_2cycles(Type *f, ulong n,
3                                     double *tb=0, bool bi=false)
4  // Permute the elements of f by a random permutation
5  // consisting of exactly two cycles (must have n>1).
6  // Set bi:=true to signal that the sums tb[k] = sum( j=1, k+1, 1/j )
7  // have been precomputed (via init_harmonic()).
8  {
9      if ( n<=2 ) return;
10     double *b = tb;
11     if ( tb==0 ) { b = new double[n]; bi=false; }
12     if ( !bi ) init_harmonic(b, n);
13     const double hn = b[n-2];
14     const double t = rnd01() * hn;
15     const ulong s = bsearch_geq(b, n-1, t) + 1; // 1<s<n
16     for (ulong k=1; k<s; ++k)
17     {
18         const ulong i = rand_idx(k);
19         swap2(f[k], f[i]);
20     }
21     // skipping index s here
22     for (ulong k=s+1; k<n; ++k)
23     {
24         const ulong i = rand_idx(k);
25         swap2(f[k], f[i]);
26     }
27     if ( tb==0 ) delete [] b;
28 }
29
30
31
32
```

The binary search routine used is

```

1  template <typename Type>
2  ulong bsearch_geq(const Type *f, ulong n, const Type v)
3  // Return index of first element in f[] that is >= v
4  // Return n if there is no such element.
5  // f[] must be sorted in ascending order.
```

```

6 // Must have n!=0
7 {
8     ulong nlo=0, nhi=n-1;
9     while ( nlo != nhi )
10    {
11        ulong t = (nhi+nlo)/2;
12
13        if ( f[t] < v ) nlo = t + 1;
14        else           nhi = t;
15    }
16
17    if ( f[nhi]>=v ) return nhi;
18    else           return n;
19 }

```

We note a generalization of relation (6.1) that is found by counting mixed radix numbers with  $m - 1$  zeros:

$$s(n, m) = (n - 1)! \sum_{0 < l_1 < l_2 < \dots < l_{m-1} < n} \frac{1}{l_1 l_2 \dots l_{m-1}} \quad (6.3)$$

Compare to the following sum over compositions of  $n$  into at most  $m$  parts [3, ex. 18, p. 116], here the sum is over integer partitions:

$$s(n, m) = \frac{n!}{m!} \sum_{l_1 + l_2 + \dots + l_m = n} \frac{1}{l_1 l_2 \dots l_m} \quad (6.4)$$

## 6.4 Even or odd number of cycles

In the following algorithm we toggle the value of  $f(0)$  if necessary to keep the parity of the number of cycles even or odd according to the value of  $r$ :

**Algorithm 6.9** (CyclesPar). *Generate a permutation  $P \in \mathcal{S}_n$  (where  $n \geq 2$ ) such that the number of cycles modulo 2 equals  $r$ .*

1. For  $j := 1, 2, \dots, n - 1$  set  $f(j) := \mathbf{Z}(j + 1)$  (random factorial number).
2. Set  $z := 1$ . For  $j := 0, 1, \dots, n - 2$  if  $f(j) = 0$  set  $z := 1 - z$ .
3. (Here  $z$  is the number of cycles modulo 2).
4. If  $z \neq r$  then set  $f(0) := 1 - f(0)$  (adjust number of cycles).
5. Using Algorithm 6.1 convert  $F$  to a permutation  $P$ .

The adjustment in step 4 is a uniform projection with quotient 2. □

The following routine permutes the elements of the array  $x$ :

```

1 template <typename Type>
2 void random_permute_ncm2(Type *x, ulong n, ulong r, ulong *tf=0)
3 // Apply random permutation with number of cycles == r mod 2
4 // ncm2 := Number of Cycles Modulo 2
5 // Must have n>=2.
6 {
7     ulong *f = tf;
8     if ( tf==0 ) f = new ulong[n];
9
10    for (ulong j=0; j<n-1; ++j) f[j] = rand_idx(j+2);

```

```

11
12     ulong z = 1; // number cycles in factorial number
13     for (ulong j=0; j<n-1; ++j) z += (f[j] == 0);
14     z &= 1; // mod 2
15     if ( z!=r ) f[0] = 1 - f[0]; // adjust num cycles mod 2
16
17     rfact2perm_swp_apply(f, n, x);
18
19     if ( tf==0 ) delete [] f;
20 }

```

A permutation can be computed as follows:

```

1 void
2 random_ncm2_permutation(ulong *p, ulong n, ulong r, ulong *tf=0)
3 // Generate a random permutation with number of cycles == r mod 2
4 {
5     for (ulong k=0; k<n; ++k) p[k] = k; // identity
6     random_permute_ncm2(p, n, r, tf);
7 }

```

## 6.5 Exactly $m$ cycles

The recurrence for the Stirling cycle numbers (relation (2.16) on page 5)

$$s(n, m) = s(n-1, m-1) + (n-1)s(n-1, m) \quad (6.5)$$

can be interpreted as follows: a length- $n$  permutation into  $m$  cycles (term  $s(n, m)$ ) can be obtained from a permutation of length  $n-1$  in two ways. Consider the greatest element of the permutation. It is either a fixed point and the remaining elements are a permutation into  $m-1$  cycles (term  $s(n-1, m-1)$ ), or it is inserted into a cycle in a permutation with  $m$  cycles (and there are  $n-1$  ways to do this, term  $(n-1)s(n-1, m)$ ).

We generate a random permutation in two steps. The first step computes a *recipe*  $R = [r_1, r_2, r_3, \dots, r_m]$  where  $r_1 = 1$ ,  $r_j < r_{j+1}$  for all  $j$ , and  $r_m \leq n$ . The second step converts the recipe into a permutation: the entries  $r_j$  are permutation sizes where a fixed point has to be added. The algorithm for generating a recipe works for  $N \geq 1$  and  $1 \leq M \leq N$ .

**Algorithm 6.10** (NumCyclesRecipe). *Generate a recipe  $R = [r_1, r_2, r_3, \dots, r_M]$  for a random permutation  $P \in S_N$  with exactly  $M$  cycles.*

1. Set  $m := M$  (cycles in remaining permutation) and  $c = m$  (write position in recipe).
2. For  $n := N, N-1, \dots, 1$  do:
  - (a) If  $m = 1$  then set  $r_1 := 1$  and terminate this loop (one cycle).
  - (b) Set  $r := Z(a)$  where  $a := s(n, m)$ .
  - (c) If  $r < s(n-1, m-1)$  then (a fixed point will be added):  
set  $r_c := n$ ,  $c := c-1$ , and  $m := m-1$ .
3. Return  $R = [r_1, r_2, r_3, \dots, r_m]$ .

As given the algorithm has a complexity worse than  $O(n)$  because the numbers  $s(n, m)$  cannot be stored in space  $O(1)$ . Using instead a table of the probabilities  $b(n, m) = s(n-1, m-1)/s(n, m)$ , stored as floating-point numbers, would give linear complexity. Then steps 2b and 2c would be replaced by

Set  $r := \mathbf{R}()$  (random real number  $0 \leq r < 1$ ).

If  $r < b(n, m)$  then (add fixed point, same assignments as before).

The following algorithm converts a recipe into a permutation.

**Algorithm 6.11** (Recipe2Perm). *Convert a recipe  $R = [r_1, r_2, r_3, \dots, r_M]$  into a random permutation  $P \in \mathcal{S}_N$  with exactly  $M$  cycles.*

1. Set  $P = [p(1), p(2), \dots, p(n)] := [1, 2, \dots, n]$  (one-based).
2. Set  $n := 1$ .
3. For  $a := 1, 2, \dots, m$  do:
  - (a) For  $j := n, n+1, \dots, r_a$  do: (join elements to cycles)  
set  $i := 1 + \mathbf{Z}(j-1)$  and swap  $p(i)$  with  $p(j)$ .
  - (b) (add a fixed point for  $j = r_a$ , this is a no-op).
  - (c) Set  $n := r_a + 1$ .
4. For  $j := n, n+1, \dots, N$  do: (join remaining elements to cycles)  
set  $i := 1 + \mathbf{Z}(j-1)$  and swap  $p(i)$  with  $p(j)$ .

The algorithm has complexity  $O(n)$ .

The following routines are written in the GP scripting language [21]. The implementation uses a precomputed table of the Stirling cycle numbers  $s(n, m)$  where  $1 \leq n \leq N$  and  $1 \leq m \leq M$ . We use an auxiliary routine for updating a vector of cycle numbers  $s(n-1, m)$  (where  $1 \leq m \leq M$ ) to numbers  $s(n, m)$ .

```

1  st1_next(s, n1)=
2  \\ Let s be a vector of the Stirling cycle numbers
3  \\ s(n,1), s(n,2), ..., s(n,m) and n1=n-1.
4  \\ Return the vector of the cycle numbers
5  \\ s(n+1,1), s(n+1,2), ..., s(n+1,m).
6  {
7      local(k, sn);
8      k = length(s);
9      sn = vector(k);
10     sn[1] = n1*s[1];
11     for (m=2, k, sn[m] = s[m-1] + n1*s[m]; );
12     return( sn );
13 }
```

The routine for the computation of the table  $T$  is

```

1  st1_tab(N, M)=
2  \\ Compute table T of Stirling cycle numbers
3  \\ T[n][m] == s(n,m) for 1<=n<=N and 1<=m<=M.
4  {
5      local(s, T);
6      s = vector(M);
7      s[1] = 1;
8      T = vector(N);
9      T[1] = s;
10     for (n=2, N,
```

```

11     s = st1_next(s, n-1);
12     T[n] = s;
13     );
14     return( T );
15 }

```

The routine for computing a recipe is

```

1  recipe_num_cyc(N, M)=
2  \\ Generate a recipe for a random length-n permutation into m cycles.
3  \\ Uses global variable T, the table of Stirling cycle numbers.
4  {
5      local( R, ct, m );
6      R = vector(M); \\ recipe
7      ct = M; \\ write position in R
8      m = M; \\ cycles in remaining permutation
9      forstep(n = N, 1, -1,
10
11         if ( m==1, \\ only one cycle left
12             R[ct]=1; \\ ==R[1]
13             ct--1;
14             break()
15         );
16
17         st0 = (n-1) * T[n-1][m]; \\ probability of joining to cycle
18         st1 = T[n-1][m-1]; \\ probability of adding a fixed point
19         rd = random( st0 + st1 ); \\ 0 <= rd < st0+st1
20         if ( rd < st1,
21             \\ THEN add fixed point at step n
22             m -= 1;
23             R[ct] = n;
24             ct -= 1;
25             \\ ELSE connect to any cycle
26         );
27     );
28 };
29
30 return( R );
31 }

```

The following routine converts a recipe to a random permutation:

```

1  recipe2perm(R, N)=
2  \\ Convert recipe to a length-n permutation.
3  {
4      local(P, n, t);
5      P = vector(N,j,j); \\ permutation
6      n = 1; \\ position in permutation
7      for (a=1, length(R),
8          for (j=n, R[a]-1,
9              i = 1 + random(j-1); \\ one-based arrays
10             t=P[i]; P[i]=P[j]; P[j]=t; \\ join to cycle (swap)
11         );
12         \\ and add fixed point (no-op)
13         n = R[a] + 1;
14     );
15
16     for (j=n, N, \\ (elements left for processing)
17         i = 1 + random(j-1); \\ one-based arrays
18         t=P[i]; P[i]=P[j]; P[j]=t; \\ join to cycle (swap)
19     );
20
21     return(P);
22 }

```

It should be noted that the recipes are in general not uniformly distributed, because the recipes correspond to different numbers of permutations. For example, with  $n = 4$  and  $m = 3$  there are three recipes whose relative frequencies are  $3 : 2 : 1$ .

With the generation of 60,000 such permutations we obtain the following statistics:

num	permutation	recipe
10107	[1, 2, 4, 3]	[1, 2, 3]
10001	[1, 4, 3, 2]	[1, 2, 3]
9955	[4, 2, 3, 1]	[1, 2, 3]
9961	[1, 3, 2, 4]	[1, 2, 4]
10071	[3, 2, 1, 4]	[1, 2, 4]
9905	[2, 1, 3, 4]	[1, 3, 4]





## Chapter 7

# Inversions

An *inversion* of a permutation  $P = [p(0), p(1), \dots, p(n-1)]$  is a pair of indices  $k$  and  $j$  where  $k < j$  and  $p(j) < p(k)$ . We call such an inversion a *right inversion* at  $k$ , or a *left inversion* at  $j$ . Inversions are important for the analysis of sorting algorithms, see [14]. We give algorithms to generate random permutations with conditions on the number of inversions.

### 7.1 Inversion table

	permutation	num. inv.	right inv.	left inv.
0:	[ . 1 2 3 ]	0	[ . . . ]	[ . . . ]
1:	[ . 1 3 2 ]	1	[ . . 1 ]	[ . . 1 ]
2:	[ . 2 1 3 ]	1	[ . 1 . ]	[ . 1 . ]
3:	[ . 2 3 1 ]	2	[ . 1 1 ]	[ . . 2 ]
4:	[ . 3 1 2 ]	2	[ . 2 . ]	[ . 1 1 ]
5:	[ . 3 2 1 ]	3	[ . 2 1 ]	[ . 1 2 ]
6:	[ 1 . 2 3 ]	1	[ 1 . . ]	[ 1 . . ]
7:	[ 1 . 3 2 ]	2	[ 1 . 1 ]	[ 1 . 1 ]
8:	[ 1 2 . 3 ]	2	[ 1 1 . ]	[ . 2 . ]
9:	[ 1 2 3 . ]	3	[ 1 1 1 ]	[ . . 3 ]
10:	[ 1 3 . 2 ]	3	[ 1 2 . ]	[ . 2 1 ]
11:	[ 1 3 2 . ]	4	[ 1 2 1 ]	[ . 1 3 ]
12:	[ 2 . 1 3 ]	2	[ 2 . . ]	[ 1 1 . ]
13:	[ 2 . 3 1 ]	3	[ 2 . 1 ]	[ 1 . 2 ]
14:	[ 2 1 . 3 ]	3	[ 2 1 . ]	[ 1 2 . ]
15:	[ 2 1 3 . ]	4	[ 2 1 1 ]	[ 1 . 3 ]
16:	[ 2 3 . 1 ]	4	[ 2 2 . ]	[ . 2 2 ]
17:	[ 2 3 1 . ]	5	[ 2 2 1 ]	[ . 2 3 ]
18:	[ 3 . 1 2 ]	3	[ 3 . . ]	[ 1 1 1 ]
19:	[ 3 . 2 1 ]	4	[ 3 . 1 ]	[ 1 1 2 ]
20:	[ 3 1 . 2 ]	4	[ 3 1 . ]	[ 1 2 1 ]
21:	[ 3 1 2 . ]	5	[ 3 1 1 ]	[ 1 1 3 ]
22:	[ 3 2 . 1 ]	5	[ 3 2 . ]	[ 1 2 2 ]
23:	[ 3 2 1 . ]	6	[ 3 2 1 ]	[ 1 2 3 ]

Figure 7.1: All permutations  $\in \mathcal{S}_4$  in lexicographic order, their number of inversions, and their left and right inversion tables. Dots denote zeros.

The (right) *inversion table*  $I = [i_0, i_1, \dots, i_{n-2}]$  of a permutation is computed by setting  $i_k$  to the number of right inversions at  $k$ . Note we omit the element  $i_{n-1} = 0$ . We have  $0 \leq i_k < n - k$  by definition, so the inversion table is a mixed radix number in falling factorial base.

For example, the permutation  $P = [3, 0, 1, 4, 2]$  has the inversion table  $I = [3, 0, 0, 1]$ : three elements less than the first element (3) lie to the right of it, no elements less

than the second (0) or third (1) elements lie right to them, and one element less than 4 lies right of it.

We could similarly define a left inversion table, figure 7.1 shows both types of inversion tables for all permutations  $\in S_4$ .

The obvious algorithm for computing the inversion table is to count, for each element  $p(k)$ , the number of elements  $p(j)$  right to it ( $j > k$ ) that are smaller than  $p(k)$ . An implementation is

```

1 void perm2ffact(const ulong *x, ulong n, ulong *ff)
2 // Convert permutation in x[0,...,n-1] into
3 // the (n-1) digit falling factorial representation in ff[0,...,n-2].
4 // We have: ff[0]<n, ff[1]<n-1, ..., ff[n-2]<2 (falling radices)
5 {
6     for (ulong k=0; k<n-1; ++k)
7     {
8         ulong xk = x[k];
9         ulong i = 0;
10        for (ulong j=k+1; j<n; ++j) if ( x[j]<xk ) ++i;
11        ff[k] = i;
12    }
13 }
```

The complexity is  $O(n^2)$ , also for the other direction. We first give the algorithm.

**Algorithm 7.1** (FFact2Perm-Rot). *Convert the right inversion table  $F$  into the corresponding permutation  $P \in S_n$ .*

1. For  $j := 0, 1, \dots, n-1$  do: set  $p(j) := j$  (identical permutation).
2. For  $k := 0, 1, 2, \dots, n-2$  do: (rotations)
  - (a) Set  $s := f(k)$ .
  - (b) Cyclically rotate  $[p(k), p(k+1), \dots, p(k+s)]$  ( $s+1$  elements) by one position to the right (no-op if  $s = 0$ ).

As an example we compute the length-5 permutation corresponding to the inversion table  $[3, 0, 0, 1]$ :

```

[0, 1, 2, 3, 4] (start, identity)
  > > > >
[3, 0, 1, 2, 4] (rotate 4=3+1 elements, starting from k=0)
  > > >
[3, 0, 1, 2, 4] (rotate 1=0+1 elements, starting from k=1, no-op)
  > > >
[3, 0, 1, 2, 4] (rotate 1=0+1 elements, starting from k=2, no-op)
  > >
[3, 0, 1, 4, 2] (rotate 2=1+1 elements, starting from k=3)
```

The underlying trick is that the rotation starting at position  $k$  leaves the suffix starting at position  $k+1$  in ascending order. The following right shifts by one position move the last element of the shifted range to its front, leaving  $s$  smaller elements (all from the shifted part) to the right of it.

An implementation is

```

1 void
2 ffact2perm(const ulong *ff, ulong n, ulong *x)
3 // Convert the (n-1) digit falling factorial representation
4 // in ff[0,...,n-2] into a permutation in x[0,...,n-1].
5 // into permutation in x[0,...,n-1]
6 // Must have: ff[0]<n, ff[1]<n-1, ..., ff[n-2]<2 (falling radices)
```

```

7  {
8    for (ulong k=0; k<n; ++k) x[k] = k;
9    for (ulong k=0; k<n-1; ++k)
10   {
11     ulong f = ff[k];
12     if ( f ) rotate_right1(x+k, f+1);
13   }
14 }
```

The routine `rotate_right1(x+k, f+1)` applies a cyclic shift of  $f+1$  elements of the array  $X$  to the right. The part  $x(k), x(k+1), \dots, x(k+f), x(k+f+1)$  is changed into  $x(k+f+1), x(k), x(k+1), \dots, x(k+f)$ .

Initially the elements are in order. The cyclic shift at step  $k$  moves an element to the left of  $f(k)$  greater elements. The ascending order of all elements to the right of position  $k$  is preserved. Thus the routine computes a permutation with  $f(k)$  (right) inversions at position  $k$ , as required.

The method has a certain similarity with the set maintenance technique of section 5.1 on page 25. Here we put removed elements to the left, and the rotation leaves all elements right of the element just removed in ascending order.

Let  $I$  be the inversion table of a permutation  $P \in \mathcal{S}_n$ , we define

$$\text{inv}(P) := \sum_{k=0}^{n-2} i_k \quad (7.1)$$

The quantity  $\text{inv}(P)$  gives the number of inversions in  $P$ .

Inversion tables appear in Rothe's contribution "Ueber Permutationen, in Beziehung auf die Stellen ihrer Elemente." to [12, pp. 263ff]. References to earlier work are given [17, chap. 4, p. 92], the earliest being to Cramer (1750).

## 7.2 Fast conversion to and from the inversion table

We now give routines to convert between inversion tables and permutations that have complexity  $O(n \log n)$ . While it appears to be well-known that these conversions are possible, no explicit solution seems to be available in the published literature.

The following  $O(n^2)$  algorithm is our starting point.

**Algorithm 7.2** (FFact2Perm). *Convert the right inversion table  $F$  into the corresponding permutation  $P \in \mathcal{S}_n$ .*

1. For  $j := 0, 1, \dots, n-1$  do: set  $b(j) := 0$  (tag-array  $B$ :  $b(j) = 1$  if element  $j$  is used, otherwise  $b(j) = 0$ ).
2. For  $k := 0, 1, 2, \dots, n-2$  do: (insert  $f(k)$ -th unused element, complexity  $O(n)$ )
  - (a) Set  $c := f(k) + 1$ .
  - (b) For  $j := 0, 1, \dots, n-1$  do: (find  $f(k)$ -th unused element)
    - If  $b(j) = 0$  then set  $c := c - 1$  (element  $j$  unused).
    - If  $c = 0$  then set  $p(k) := j$ ,  $b(j) := 1$ , and terminate this loop.

3. For  $j := 0, 1, \dots, n-1$  do: (find remaining element)
  - if  $b(j) = 0$  then set  $p(n-1) := j$  and terminate this loop.

The last step is separated because the inversion table omits an implicit trailing zero.

The searches for the unused elements are  $O(n)$  as given but can be made  $O(\log n)$  if the LR-array described in appendix B on page 81 is used.

**Algorithm 7.3 (FFact2Perm-LR).** Convert the right inversion table  $F$  into the corresponding permutation  $P \in S_n$ .

1. For  $k := 0, 1, 2, \dots, n-2$  do: (insert  $f(k)$ -th unused element)
  - (a) Find  $j$  the  $f(k)$ -th unused element (where  $f(k) = 0$  corresponds to the first element) and mark this position as used (using an LR-array these operations are  $O(\log n)$ ).
  - (b) Set  $p(k) := j$ .
2. Find  $j$ , the remaining unused element, and set  $p(n-1) := j$ .

The implementation using an LR-array is

```

1 void
2 ffact2perm(const ulong *ff, ulong n, ulong *x, left_right_array &LR)
3 {
4     LR.free_all();
5     for (ulong k=0; k<n-1; ++k)
6     {
7         ulong i = LR.get_free_idx_chg( ff[k] );
8         x[k] = i;
9     }
10    ulong i = LR.get_free_idx_chg( 0 );
11    x[n-1] = i;
12 }
```

The LR-array passed as an extra argument has to be of size  $n$ . The routine for the fast computation of the inversion table is

```

1 void
2 perm2ffact(const ulong *x, ulong n, ulong *ff, left_right_array &LR)
3 {
4     LR.set_all();
5     for (ulong k=0; k<n-1; ++k)
6     {
7         // i := number of Set positions Left of x[k], Excluding x[k].
8         ulong i = LR.num_SLE( x[k] );
9         LR.get_set_idx_chg( i );
10        ff[k] = i;
11    }
12 }
```

For the computation of the number of inversions the array holding the factorial number  $F$  can obviously be avoided:

```

1 ulong
2 count_inversions(const ulong *f, ulong n, left_right_array *tLR)
3 {
4     left_right_array *LR = tLR;
5     if ( tLR==0 ) LR = new left_right_array(n);
6
7     ulong ct = 0;
8     LR->set_all();
9     for (ulong k=0; k<n-1; ++k)
10    {
11        ulong i = LR->num_SLE( f[k] );
```

```

12         LR->get_set_idx_chg( i );
13         ct += i;
14     }
15
16     if ( tLR==0 ) delete LR;
17     return ct;
18 }

```

No method better than  $O(n \log n)$  for computing  $\text{inv}(P)$  (or even the inversion table) appears to be known.

### 7.3 Inversions modulo $m$

In the inversion table  $I = [i_0, i_1, \dots, i_{n-2}]$  of a permutation  $P \in \mathcal{S}_n$  we have  $0 \leq i_k < n - k$ , so there are  $n - k$  possible values of  $i_k$ . To generate a random permutation with a prescribed number  $r$  of inversions modulo  $m$  where  $2 \leq m \leq n$ , we generate a random inversion table, adjust  $i_{n-m}$  according to the requirement, and convert into a permutation.

**Algorithm 7.4** (InvModM). *Generate a random permutation  $P \in \mathcal{S}_n$  such that  $\text{inv}(P) \equiv r \pmod m$  where  $2 \leq m \leq n$ .*

1. For  $j := 0, 1, \dots, n - 2$  set  $i_j := \mathbf{Z}(n - j)$  (random inversion table  $I$ ).
2. Set  $i := \sum_{j=0}^{n-2} i_j$  ( $i = \text{inv}(P)$ ) and  $m_i := i \pmod m$ .
3. If  $m_i = r$  then go to step 8.
4. Set  $d := i_{n-m}$  (value of digit to adjust).
5. Set  $d := d + r - m_i$  (adjust value, may be out of range).
6. If  $d < 0$  then set  $d := d + m$ . If  $d \geq m$  then set  $d := d - m$ . (Now  $0 \leq d < m$ ).
7. Set  $i_{n-m} := d$  (write back to inversion table, now  $\sum_{j=0}^{n-2} i_j \equiv r \pmod m$ ).
8. Convert  $I$  into a permutation  $P$ .

The adjustment of  $i_{n-m}$  is a uniform projection with quotient  $m$ . □

The last step is  $O(n \log n)$  (if the fast conversion technique is used) and so is the whole algorithm.

An implementation is

```

1  inline void random_inv_mod_m_permutation(ulong *p, ulong n,
2                                           ulong r, ulong m,
3                                           ulong *tff=0)
4  // Create random permutation p[] such that (i%m)==r
5  // where i is the number of inversions.
6  // Must have: 2 <= m <= n and 0 <= r < m.
7  {
8      ulong *ff = tff;
9      if ( tff==0 ) ff = new ulong[n-1];
10
11     // Create random factorial number:
12     for (ulong rx=n, j=0; rx>1; --rx, ++j) ff[j] = rand_idx(rx);
13
14     ulong i = 0; // number of inversions (sum of digits):
15     for (ulong j=0; j<n; ++j) i += ff[j];

```

```

16
17     ulong mi = i % m; // i mod m of given permutation
18     if ( mi != r ) // need to adjust digit
19     {
20         const ulong ps = n - m; // position of digit to adjust
21         ulong d = ff[ps]; // value of digit
22         d += (r-mi);
23         if ( (long)d < 0 ) d += m;
24         if ( d >= m ) d -= m;
25         ff[ps] = d;
26     }
27     ffact2perm(ff, n, p); // may replace by O(n*log(n)) routine
28
29     if ( tff==0 ) delete [] ff;
30
31 }

```

Here we use the  $O(n^2)$  method for conversion, replace with fast method to obtain an  $O(n \log n)$  routine.

## 7.4 Fixed number of inversions (open problem)

No efficient algorithm for the unbiased generation of a random permutation with a prescribed number of inversions could be found.

An obvious approach is to generate a random composition into  $n - 1$  parts where the  $j$ -th part is  $\leq j$  ( $1 \leq j \leq n - 1$ ). The generation of a random composition without the restriction on the size of the parts is straightforward (see [19, chap. 6, p. 52]). However, no efficient method for imposing the size restriction could be found.

The distribution of the number of inversions in a random permutation is studied in [20], where an algorithm for the generation of arbitrarily distributed permutations is given.

## Chapter 8

# Connected permutations

A *proper prefix* of a sequence  $A = [A_0, A_1, \dots, A_{n-1}]$  is a sequence  $[A_0, A_1, \dots, A_k]$  where  $0 \leq k < n - 1$ . A permutation  $P \in \mathcal{S}_n$  that does not map any proper prefix to itself is called *connected* (or *indecomposable*). For example, the permutation  $P = [2, 0, 1, 4, 3] \in \mathcal{S}_5$  is not connected because it maps the prefix  $[0, 1, 2]$  to itself. Also a permutation that maps any proper suffix to itself cannot be connected, for example, the permutation just given maps the suffix  $[3, 4]$  to itself, so it must map the prefix  $[0, 1, 2]$  to itself.

### 8.1 Asymptotics

The sequence of the numbers  $C_n$  of indecomposable permutations starts as

$n:$	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	...
$C_n:$	1,	1,	3,	13,	71,	461,	3447,	29093,	273343,	2829325,	31998903,	...
$n!:$	1,	2,	6,	24,	120,	720,	5040,	40320,	362880,	3628800,	39916800,	...

This is entry A003319 in [26]. We have the following recurrence:

$$C_n = n! - \sum_{k=1}^{n-1} k! C_{n-k} \quad (8.1)$$

We now show that almost all permutations  $\in \mathcal{S}_n$  are connected for large  $n$ .

Let  $P \in \mathcal{S}_n$  where  $n$  is large. We write  $a, b, c$  for the three smallest elements  $(0, 1, 2)$  in  $P$ , and  $x, y, z$  for the three largest  $(n-3, n-2, n-1)$ . Also write  $C(n)$  for a connected permutation of length  $n$ , write  $[u, v, w, C(n-5), r, s]$  for the set of all permutations with prefix  $[u, v, w]$ , suffix  $[r, s]$  and where  $C(n-5)$  is a connected permutation of the remaining elements.

Every permutation can be written uniquely in the form  $[A, C(n-k), Z]$  for some  $k$  where  $A$  is a permutation of the first  $j$  elements ( $0 \leq j \leq k$ ) and  $Z$  is a permutation of the last  $k-j$  elements, as long as  $n-k \geq 2$ : A permutation  $P \in \mathcal{S}_n$  can either be connected (i.e.  $P \in [C(n)]$ ), or of the form  $[a, C(n-1)]$  or  $[C(n-1), z]$  (i.e.  $P \in \{[a, C(n-1)] \cup [C(n-1), z]\}$ ), or of the form  $[b, a, C(n-2)]$  or  $[C(n-2), z, y]$  or  $[a, C(n-2), z]$  or  $[a, b, C(n-2)]$  or  $[C(n-2), y, z]$ , and so on.

Write  $C_n$  for the number of connected permutations of length  $n$ . Counting gives the following asymptotic expansion:

$$n! \sim \sum_{k \geq 0} a_k C_{n-k} \quad \text{where} \quad a_k = \sum_{j=0}^k j! (k-j)! \quad (8.2)$$

$$= C_n + 2C_{n-1} + 5C_{n-2} + 16C_{n-3} + 64C_{n-4} + 312C_{n-5} + \dots \quad (8.3)$$

The sequence of coefficients  $a_k$  is entry A003149 in [26].

We make the following Ansatz for the asymptotic expansion of  $C_n$ :

$$C_n \sim n! \left( \frac{b_0}{1} + \frac{b_1}{n} + \frac{b_2}{n^2} + \frac{b_3}{n^3} + \frac{b_4}{n^4} + \dots \right) \quad (8.4)$$

To solve for the coefficients  $b_j$  we recursively evaluate  $C_n$  up to  $N = 10$  terms. The following is a script in the GP language.

```

1  a(n)=sum(k=0,n, k!*(n-k)!) \\ seq. A003149
2  C(n,N)= \\ N terms of C_n
3  { return( 1 - sum(k=1, N, a(k) * C(n-k, N-k) / prod(j=0,k-1,n-j) ) ); }
4  N=10
5  t = C(n,N) \\ expression (for C_n/n!) in n
6  t = subst(t, n, 1/e) \\ expression in e = 1/n
7  t = taylor(t, e) \\ series in e = 1/n
8  \\ == 1 - 2*e - e^2 - 5*e^3 - 32*e^4 - 253*e^5 -
9  Vec(t)
10 \\ == [1, -2, -1, -5, -32, -253, -2381, -25912, -319339, ...]
```

Thus we have

$$C_n \sim n! \left( \frac{1}{1} - \frac{2}{n} - \frac{5}{n^2} - \frac{32}{n^3} - \frac{253}{n^4} - \dots \right) \quad (8.5)$$

The probability that a random permutation is not connected is  $2/n + O(1/n^2)$  for  $n$  large. We have not found (8.5) in the literature, so it may be new.

## 8.2 The rejection method

An algorithm for generating a random object  $x$  where  $x \in U \subset V$  ( $U$  is the subset of all objects in  $V$  that have a required property) is as follows [18].

**Algorithm 8.1** (Reject). *Generate a (uniform) random object  $x \in U \subset V$ .*

1. Generate a random object  $x \in V$ .
2. If  $x \in U$  then return it ( $x$  is of the required type).
3. Go to step 1.

We will call this algorithm the *rejection method*.

The method is efficient if generating a random object  $x \in V$  is fast, the test whether  $x \in U$  is fast, and  $|V|/|U|$  (the expected number of iterations) is not too large.

To test for connectedness, we compute the maxima  $m_k$  of all proper prefixes  $[p(0), p(1), \dots, p(k)]$  where  $k \leq n - 2$ . If  $m_k \leq k$  then all elements in the prefix are  $\leq k$ ,



so the prefix is mapped to itself and the permutation is not connected. The test can be done in one linear scan, so it has complexity  $O(n)$ .

**Algorithm 8.2** (TestConnected). *Return whether a permutation  $P \in \mathcal{S}_n$  is connected.*

1. Set  $m := -\infty$  (maximum of the prefix seen so far).
2. For  $k := 0, 1, \dots, n - 2$  (for all proper prefixes), do:
  - (a) If  $p(k) > m$  then set  $m := p(k)$  (update max).
  - (b) If  $m \leq k$  then return false (prefix mapped to itself,  $P$  not connected).
3. Return true ( $P$  is connected).

By the above, the following algorithm runs in expected time  $n(1 + 2/n + O(1/n^2)) = n + 2 + O(1/n)$  for  $n$  large.

**Algorithm 8.3** (Connected1). *Generate a random connected permutation  $P \in \mathcal{S}_n$ .*

1. Generate a random permutation  $P$ .
2. If  $P$  is not connected (Algorithm 8.2) then goto step 1.
3. Return  $P$ .

### 8.3 Improved algorithm

The two forms of non-connected permutations that lead to the term  $2/n$  in the asymptotic expansion are  $[a, C(n-1)]$  and  $[C(n-1), z]$ . Avoiding both of these forms in the random permutation  $P$  reduces the probability that  $P$  is not connected to  $O(1/n^2)$ . The five forms leading to the quadratic term are  $[b, a, C(n-2)]$ ,  $[C(n-2), z, y]$ ,  $[a, C(n-2), z]$ ,  $[a, b, C(n-2)]$ , and  $[C(n-2), y, z]$ . As the last three are excluded by avoiding the prefix  $a$  and suffix  $z$ , the probability that  $P$  is not connected is  $2/n^2 + O(1/n^3)$ .

Thus the following algorithm runs in expected time  $n(1 + O(1/n^2)) = n + O(1/n)$  for  $n$  large.

**Algorithm 8.4** (Connected2). *Generate a random connected permutation  $P \in \mathcal{S}_n$ .*

1. If  $n \leq 3$  then (small cases)
  - (a) Set  $P := 1_n$ .
  - (b) If  $n \leq 1$  then return  $P$  (trivial cases).
  - (c) Swap  $p(0)$  with  $p(n-1)$ .
  - (d) If  $n = 2$  then return  $P$  ( $P = [1, 0]$ ).
  - (e) (Here  $P = [2, 1, 0]$ )
  - (f) Set  $i := \mathbf{Z}(3)$  ( $i \in \{0, 1, 2\}$ ). Swap  $p(1)$  with  $p(i)$ .
  - (g) Return  $P$  ( $i = 0 \implies [1, 2, 0]$ ,  $i = 1 \implies [2, 1, 0]$ ,  $i = 2 \implies [2, 0, 1]$ ).

2. Set  $P := 1_n$  (here  $n > 3$ ).
3. Set  $i_0 := 1 + Z(n-1)$  ( $i_0 \in \{1, 2, \dots, n-1\}$ ).
4. Swap  $p(0)$  with  $p(i_0)$  (now  $p(0) \neq 0$ ).
5. Set  $i_1 := 1 + Z(n-1)$  (for swap with  $p(1)$ , which will be moved to last position).
6. Swap  $p(1)$  with  $p(i_1)$ .
7. If  $p(1) = n-1$  then (last element would be greatest, try again)
  - (a) Swap  $p(1)$  with  $p(i_1)$ . Swap  $p(0)$  with  $p(i_0)$  (undo swaps, note reversed order).
  - (b) Goto step 3 (probability  $\approx 1/n$ , but work only  $O(1)$ ).
8. Swap  $p(1)$  with  $p(n-1)$  (move to last position).
9. (Here  $p(0) \neq 0$  and  $p(n-1) \neq n-1$ ).
10. Randomly permute second to second last element ( $[p(1), \dots, p(n-2)]$ ).
11. If  $P$  is not connected (use Algorithm 8.2 for testing) then goto step 2 (probability  $\approx 2/n^2$ , work  $O(n)$ ).
12. Return  $P$ .

## 8.4 Implementation

The code for the cases  $n \leq 3$  is

```

1  inline void random_connected_permutation(ulong *f, ulong n)
2  // Create a random connected (indecomposable) permutation.
3  {
4      if ( n<=3 )
5      {
6          for (ulong k=0; k<n; ++k) f[k] = k;
7          if ( n<2 ) return; // [] or [0]
8          swap2(f[0], f[n-1]);
9          if ( n==2 ) return; // [1,0]
10         // here: [2,1,0]
11         const ulong i = rand_idx(3);
12         swap2(f[i], f[i]);
13         // i = 0 ==> [1,2,0]
14         // i = 1 ==> [2,1,0]
15         // i = 2 ==> [2,0,1]
16         return;
17     }
18 }

```

The optimized version of the main part is (Algorithm 8.4)

```

1      // will repeat with probability 2/n^2:
2      do
3      {
4          for (ulong k=0; k<n; ++k) f[k] = k;
5
6          while ( 1 )
7          {
8              const ulong i0 = 1 + rand_idx(n-1); // first element must move
9              const ulong i1 = 1 + rand_idx(n-1); // f[i1] will be last element

```

```

10     swap2( f[0], f[i0] );
11     swap2( f[1], f[i1] );
12     if ( f[1]==n-1 ) // undo swap and repeat (here: f[0]!=0)
13     {
14         swap2( f[1], f[i1] );
15         swap2( f[0], f[i0] );
16         continue; // probability 1/n but work only O(1)
17     }
18     else break;
19 }
20
21 swap2(f[1], f[n-1]); // move f[1] to last
22 // here: f[0] != 0 and f[n-1] != n-1
23 random_permute(f+1, n-2); // permute 2nd ... 2nd last element
24 }
25 while ( ! is_connected(f, n) );
26 }

```

Without the optimization the main part of the routine is (Algorithm 8.3)

```

1 // will repeat with probability 2/n:
2 for (ulong k=0; k<n; ++k) f[k] = k;
3 do { random_permute(f, n); } while ( ! is_connected(f, n) );
4 }

```

The optimized routine is faster for small  $n$ : the performance ratio is about 1.35 for  $n = 5$ , 1.17 for  $n = 10$ , 1.10 for  $n = 20$ , and close to 1 for  $n > 100$ .



## Chapter 9

# Involutions

A permutation which is its own inverse is called an *involution*. Involutions contain only fixed points and cycles of length 2.

### 9.1 The number of involutions

Let  $I(n)$  denote the number of involutions of size  $n$ . The sequence of numbers  $I(n)$  is entry A000085 of [26]. It starts as follows:

n: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...  
 $I(n)$ : 1, 1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, ...

array form	cycles
[ 0 1 2 3 4 ]	(0) (1) (2) (3) (4)
[ 0 1 2 4 3 ]	(0) (1) (2) (3, 4)
[ 0 1 3 2 4 ]	(0) (1) (2, 3) (4)
[ 0 1 4 3 2 ]	(0) (1) (2, 4) (3)
[ 0 2 1 3 4 ]	(0) (1, 2) (3) (4)
[ 0 2 1 4 3 ]	(0) (1, 2) (3, 4)
[ 0 3 2 1 4 ]	(0) (1, 3) (2) (4)
[ 0 3 4 1 2 ]	(0) (1, 3) (2, 4)
[ 0 4 2 3 1 ]	(0) (1, 4) (2) (3)
[ 0 4 3 2 1 ]	(0) (1, 4) (2, 3)
[ 1 0 2 3 4 ]	(0, 1) (2) (3) (4)
[ 1 0 2 4 3 ]	(0, 1) (2) (3, 4)
[ 1 0 3 2 4 ]	(0, 1) (2, 3) (4)
[ 1 0 4 3 2 ]	(0, 1) (2, 4) (3)
[ 2 1 0 3 4 ]	(0, 2) (1) (3) (4)
[ 2 1 0 4 3 ]	(0, 2) (1) (3, 4)
[ 2 3 0 1 4 ]	(0, 2) (1, 3) (4)
[ 2 4 0 3 1 ]	(0, 2) (1, 4) (3)
[ 3 1 2 0 4 ]	(0, 3) (1) (2) (4)
[ 3 1 4 0 2 ]	(0, 3) (1) (2, 4)
[ 3 2 1 0 4 ]	(0, 3) (1, 2) (4)
[ 3 4 2 0 1 ]	(0, 3) (1, 4) (2)
[ 4 1 2 3 0 ]	(0, 4) (1) (2) (3)
[ 4 1 3 2 0 ]	(0, 4) (1) (2, 3)
[ 4 2 1 3 0 ]	(0, 4) (1, 2) (3)
[ 4 3 2 1 0 ]	(0, 4) (1, 3) (2)

Figure 9.1: All 26 involutions  $\in \mathcal{S}_5$  in lexicographic order.

We have  $I(0) = 1$ ,  $I(1) = 1$ , and, for  $n > 1$ , the recursion [24, pp. 85-86]

$$I(n) = I(n-1) + (n-1)I(n-2) \quad (9.1)$$

Pick any element  $x$  of an involution  $\in \mathcal{S}_n$ . If it is a fixed point, the remaining elements are an involution  $\in \mathcal{S}_{n-1}$ . If it lies in a 2-cycle with some other element  $y$  (there are  $n-1$  choices for  $y$ ), then remaining elements are an involution  $\in \mathcal{S}_{n-2}$ .  $\square$

Figure 9.1 illustrates this for the  $I(5) = 26$  involutions of 5 elements. There are  $I(n-1) = I(4) = 10$  involutions that have  $x = 0$  as a fixed point. The elements  $\neq x$  form an involution in each of these. The remaining involutions contain a cycle  $(x, y) = (0, y)$  with  $(n-1) = 4$  choices for  $y$ . For each choice of  $y$  the elements  $\neq x$  and  $\neq y$  form an involution of size  $(n-2) = 3$ , there are  $I(3) = 4$  such involutions.

## 9.2 Branching probabilities

To generate a random involution, we put each element either into a fixed point or into a 2-cycle. To obtain a uniform distribution, we need to compute (for every  $n$ ) the probability that the element under consideration is a fixed point and branch accordingly.

We can interpret relation (9.1) as follows. Among the  $I(n)$  involutions  $\in \mathcal{S}_n$  there are  $I(n-1)$  such that the element  $x$  under consideration is a fixed point and the remaining  $(n-1)I(n-2)$  involutions have  $x$  in a 2-cycle. A first version of the algorithm can now be given.

**Algorithm 9.1** (Involutions). *Generate a random involution  $\in \mathcal{S}_n$ .*

1. Set  $P := 1_n$ . Set  $S := \{0, 1, \dots, n-1\}$ .
2. If  $n \leq 1$  exit.
3. Choose the smallest  $x \in S$  (cycle leader) and remove  $x$  from  $S$ .
4. Set  $t := \mathbf{Z}(I(n))$  and  $f := I(n-1)$ . Set  $n := n-1$ .
5. If  $t < f$  then goto step 2 (fixed point, nothing to do).
6. Randomly choose  $y \in S$  ( $n-1$  choices) and remove  $y$  from  $S$ .
7. Set  $n := n-1$  and swap  $x$  with  $y$  (create a 2-cycle). Goto step 2.

In step 3 the word ‘smallest’ is used to make the choice well-defined. In fact any element  $\in S$  can be used.

The algorithm is obviously correct, but it is not practical. It has complexity  $O(n)$  only if the quantities  $I(n)$  can be computed in time  $O(1)$ . Relation (9.1) allows to update a pair  $(I(n), I(n-1)) \mapsto (I(n+1), I(n))$  with one multiplication and addition. Using integer variables leads to large numbers requiring more than constant time for these operations. With floating-point variables these operations are  $O(1)$  but an overflow will occur already for moderately large  $n$ . For example, with IEEE double precision floats  $I(295) \approx 2.80309 \cdot 10^{307}$  is the largest representable  $I(n)$ .

We note that Algorithm 9.1 is a special case of a technique that uses unranking (see section 2.8 on page 8). Let  $C(n)$  be the number of size- $n$  combinatorial objects of a certain type satisfying a recurrence relation of the form

$$C(n) = u(n)C(n-1) + v(n)C(n-2) + w(n)C(n-3) + \dots \quad (9.2)$$

where the coefficient  $u(n)$ ,  $v(n)$ ,  $w(n)$ ,  $\dots$  are nonnegative. For example, relation 9.1 is obtained by setting  $u(n) = 1$ ,  $v(n) = (n-1)$ , and  $w(n) = \dots = 0$ . We call the objects corresponding to the term  $u(n)C(n-1)$  objects of the first type, those corresponding to  $v(n)C(n-2)$  objects of the second type, and so on. An algorithm for unranking can be given as follows.

**Algorithm 9.2 (Unrank).** *Generate the  $r$ -th ( $0 \leq r < C(n)$ ) combinatorial object.*

1. Set  $U = u(n)C(n-1)$ ,  $V = v(n)C(n-2)$ ,  $W = w(n)C(n-3)$ , etc.
2. If  $r < U$  then return the  $r$ -th object of the first type of size  $n-1$  (recursion).
3. If  $r < U + V$  then return the  $(r - U)$ -th object of the second type of size  $n-2$  (recursion).
4. If  $r < U + V + W$  then return the  $(r - U - V)$ -th object of the third type of size  $n-3$  (recursion).
5. (And so on).

A random unbiased object can be generated by calling this algorithm with parameter  $r := \mathbf{Z}(C(n))$  ( $0 \leq r < C(n)$ ).

### 9.3 Practical algorithm

The key to an efficient algorithm is to use normalized probabilities. Dividing relation (9.1) by  $I(n)$  gives

$$1 = \frac{I(n-1)}{I(n)} + \frac{(n-1)I(n-2)}{I(n)} + \dots \quad (9.3)$$

The terms on the right side are the probabilities for a fixed point and for a 2-cycle, respectively. We only need one quantity. Define  $b(n) = I(n-1)/I(n)$ , we have  $0 \leq b(n) < 1$ . The update  $b(n) \mapsto b(n+1)$  is

$$b(n+1) = \frac{1}{1 + nb(n)} \quad (9.4)$$

In the algorithm we need the sequence  $b(n)$ ,  $b(n-1)$ ,  $b(n-2)$ , etc., that is the other direction:

$$b(n) = \frac{1}{n} \left( \frac{1}{b(n+1)} - 1 \right) \quad (9.5)$$

We could compute  $b(n)$  (via (9.4), in time  $O(n)$ ) before the start of the algorithm, and use (9.5) in the course of the algorithm. However, (9.5) is numerically unstable.

Therefore we will use an array of precomputed values  $b(k)$  for  $2 \leq k \leq n$ . If many involutions need to be generated, this also has the advantage that the (usually expensive) divisions in the computation are replaced by a simple read from memory. The recursion used in the following auxiliary algorithm is numerically stable.

**Algorithm 9.3** (AuxInvBranches). *Precompute array  $B = [b(2), b(3), \dots, b(n)]$  of probabilities for a fixed point.*

1. Set  $r := 0.5$  (floating-point type).
2. For  $k := 2, 3, 4, \dots, n$  do:
  - (a) Set  $b(k) := r$ .
  - (b) Set  $r := 1/(1 + (k + 1)r)$  (update  $b(k) \mapsto b(k + 1)$ ).

The implementation uses an auxiliary routine for updating the probabilities:

```

1  inline void next_involution_branch_ratio(double &rat, double &n1)
2  // R(n) = 1 / ( 1 + (n-1) * R(n-1) )
3  // R(n+1) = 1 / ( 1 + n * R(n) )
4  {
5      n1 += 1.0;
6      rat = 1.0/( 1.0 + n1*rat );
7  }
8
9  inline void init_involution_branch_ratios(double *b, ulong n)
10 {
11     b[0] = 1.0;
12     double rat = 0.5, n1 = 1.0;
13     for (ulong k=1; k<n; ++k)
14     {
15         b[k] = rat;
16         next_involution_branch_ratio(rat, n1);
17     }
18 }
```

In the main algorithm the set operations are to be done as described in section 5.1 on page 25. The function  $R()$  returns an unbiased random real number  $0 \leq x < 1$  of floating-point type.

**Algorithm 9.4** (InvolutionsPrac). *Generate a random involution  $\in S_n$ .*

1. Precompute the array  $B = [b(2), b(3), \dots, b(n)]$  of branching probabilities with Algorithm 9.3.
2. Set  $P := 1_n$ . Set  $S := \{0, 1, \dots, n - 1\}$ .
3. If  $n \leq 1$  exit.
4. Choose any  $x \in S$  (cycle leader) and remove  $x$  from  $S$ .
5. Set  $t := R()$  and  $f := b(n)$ . Set  $n := n - 1$ .
6. If  $t < f$  then goto step 3 (fixed point, nothing to do).
7. Randomly choose  $y \in S$  ( $n - 1$  choices) and remove  $y$  from  $S$ .
8. Set  $n := n - 1$  and swap  $x$  with  $y$  (create a 2-cycle). Goto step 3.

The algorithm has complexity  $O(n)$  which is optimal.



The main routine can use preallocated workspaces for the set  $S$  (variable `tr`) and the array of probabilities (variable `tb`). The first element chosen in each step is the last (in the array representation) of the set  $S$ , with this choice the swap for removing the element can be avoided.

```

1  template <typename Type>
2  inline void random_permute_self_inverse(Type *f, ulong n,
3                                         ulong *tr=0,
4                                         double *tb=0, bool bi=false)
5  // Permute the elements of f by a random involution.
6  // Set bi:=true to signal that the branch probabilities in tb[]
7  // have been precomputed (via init_involution_branch_ratios()).
8  {
9      ulong *r = tr;
10     if ( tr==0 ) r = new ulong[n];
11     for (ulong k=0; k<n; ++k) r[k] = k;
12     ulong nr = n; // number of elements available
13     // available positions are r[0], ..., r[nr-1]
14
15     double *b = tb;
16     if ( tb==0 ) { b = new double[n]; bi=false; }
17     if ( !bi ) init_involution_branch_ratios(b, n);
18
19     while ( nr>=2 )
20     {
21         const ulong x1 = nr-1; // choose last element
22         const ulong r1 = r[x1]; // available position
23         // remove from set:
24         --nr; // no swap needed if x1==last
25
26         const double rat = b[nr]; // probability to choose fixed point
27
28         const double t = rnd01(); // 0 <= t < 1
29         if ( t > rat ) // 2-cycle
30         {
31             const ulong x2 = rand_idx(nr);
32             const ulong r2 = r[x2]; // random available position != r1
33             --nr; swap2(r[x2], r[nr]); // remove from set
34             swap2( f[r1], f[r2] ); // create a 2-cycle
35         }
36         // else fixed point, nothing to do
37     }
38
39     if ( tr==0 ) delete [] r;
40     if ( tb==0 ) delete [] b;
41 }

```

## 9.4 Forward variant

We give an algorithm which avoids the array of branch ratios. The sequence of branching probabilities has to be computed each time which can be a disadvantage if many involutions are to be generated.

Fix  $n$  and let  $T_j$  ( $0 \leq j \leq \lfloor n/2 \rfloor$ ) be the number of involutions  $\in \mathcal{S}_n$  with  $j$  2-cycles. We proceed in two steps. In the first step we randomly select  $0 \leq j \leq \lfloor n/2 \rfloor$  such that the probability for obtaining  $j$  is  $T_j/I_n$  where  $I_n = \sum_{j=0}^{\lfloor n/2 \rfloor} T_j$  is the number of involutions  $\in \mathcal{S}_n$ . In the second step we generate a random involution with  $j$  2-cycles.

In the following auxiliary algorithm the abbreviation NTC is used for “Number of

2-cycles”.

**Algorithm 9.5** (Num2Cycles). *Compute the number of 2-cycles for a random involution  $\in \mathcal{S}_n$ .*

1. Set  $t_1 := 0$  and  $t_0 := 0$ . (NTC in permutations of 0 and 1 elements).
2. Set  $r := 0.5$  (probability for a fixed point for  $n = 2$ ).
3. for  $k := 2, 3, \dots, n$  do:
  - (a) (Here  $t_1$  and  $t_0$  are the NTC for  $k - 2$  and  $k - 1$ , respectively).
  - (b) Set  $t := \mathbf{R}()$ .
  - (c) If  $t > r$  then set  $(t_1, t_0) := (t_0, 1 + t_1)$  (one more 2-cycle), else set  $(t_1, t_0) := (t_0, t_0)$  (same number of 2-cycles).
  - (d) (Here  $t_1$  and  $t_0$  are the NTC for  $k - 1$  and  $k$ , respectively).
  - (e) Set  $r := 1/(1 + (k + 1)r)$  (update probability for  $k + 1$ ).
4. Return  $t_0$  (we have  $0 \leq t_0$  and  $2t_0 \leq n$ ).

The implementation is straightforward:

```

1  inline ulong rand_num_2cycles_involution(ulong n)
2  // Return number of 2-cycles for a random involution of n elements.
3  {
4      ulong t0 = 0, t1 = 0;
5      double rat = 0.5, n1 = 1.0;
6      for (ulong k=2; k<=n; ++k)
7          {
8              const double t = rnd01(); // 0 <= t < 1
9              if ( t > rat ) // 2-cycle
10                 {
11                     ulong v = 1 + t1; // == 1 + second-last
12                     t1 = t0;
13                     t0 = v;
14                 }
15                 else { t1 = t0; } // fixed point
16                 next_involution_branch_ratio(rat, n1);
17             }
18         }
19         return t0;
20     }
21 }
```

Now the main algorithm is almost trivial:

**Algorithm 9.6** (InvolutionsPrac1). *Permute the elements of  $F = [f(0), f(1), \dots, f(n - 1)]$  with a random involution  $\in \mathcal{S}_n$ .*

1. Compute  $j$  with Algorithm 9.5 (number of 2-cycles).
2. Set  $R := [0, 1, \dots, n - 1]$ , set  $r := n$  (initialize set).
3. For  $k := 1, 2, \dots, j$  do:
  - (a) Call Algorithm 5.1 to swap two of the unprocessed elements of  $F$  (Algorithm 5.1 modifies the array  $B$ : the positions of the swapped elements are removed).
  - (b) Set  $r := r - 2$  (adjust number of remaining arguments).

The implementation uses the routine for creating a random cycle given as Algorithm 5.1 on page 25:

```

1  template <typename Type>
2  inline void random_permute_self_inverse1(Type *f, ulong n, ulong *tr=0)
3  // Permute the elements of f by a random involution.
4  // This routine avoids the array of branch probabilities.
5  {
6      ulong *r = tr;
7      if ( tr==0 ) r = new ulong[n];
8      for (ulong k=0; k<n; ++k) r[k] = k; // initialize set
9      ulong nr = n; // number of elements available
10     // available positions are r[0], ..., r[nr-1]
11
12     ulong n2c = rand_num_2cycles_involution(n);
13     while ( n2c-- ) nr = random_cycle(f, 2, r, nr);
14
15     if ( tr==0 ) delete [] r;
16 }

```

A random involution is obtained with  $F = I_n$ :

```

1  inline void random_self_inverse_permutation1(ulong *f, ulong n, ulong *tr=0)
2  // Create a random involution.
3  {
4      for (ulong k=0; k<n; ++k) f[k] = k;
5      random_permute_self_inverse1(f, n, tr);
6  }

```

The routine implementing Algorithm 9.6 is slower than the routine for Algorithm 9.4. Permuting 100,000 words (of 64 bits) 1,000 times takes (using an AMD64 CPU with 2.2GHz) respectively 16.7 and 10.2 seconds.



## Chapter 10

# Cycle spectrum

We call  $P$  a permutation with *maximal cycle spectrum*  $L$  if all cycles of  $p$  have a length  $\lambda \in L$ . For example, involutions have maximal cycle spectrum  $L = \{1, 2\}$  and derangements of length  $n$  have  $L = \{2, 3, 4, \dots, n\}$ . We give algorithms to generate random permutations with prescribed maximal cycle spectrum  $L$  (that is, permutations where only cycles of length  $\lambda \in L$  are allowed).

### 10.1 Prescribed maximal cycle spectrum

The recurrence (9.1) on page 56 can be generalized for permutations where only cycles of certain lengths are allowed. Let  $L = \{l_1, l_2, \dots, l_u\}$  be the set of allowed cycle lengths. Let  $P_L(n)$  be the number of permutations of  $n$  elements with maximal cycle spectrum  $L$ . We have  $P_L(0) = 1$ ,  $P_L(n) = 0$  for  $n < 0$ , and for  $n \geq 1$  the recurrence relation (by repeated application of relation (2.16) on page 5)

$$P_L(n) = \sum_{l \in L} F(n-1, l-1) P_L(n-l) \quad \text{where} \quad (10.1a)$$

$$F(n, e) := (n)(n-1)(n-2) \dots (n-(e-1)) \quad (10.1b)$$

and  $F(n, e) = 1$  for  $e \leq 0$ . The exponential generating function (EGF) for permutations with maximal cycle spectrum  $L$  is [27, example 5.2.10, p. 18-19]

$$\sum_{n=0}^{\infty} P_L(n) \frac{x^n}{n!} = \exp \left( \sum_{l \in L} \frac{x^l}{l} \right) \quad (10.2)$$

For example, if only cycles of length 1 or 3 are allowed ( $L = \{1, 3\}$ ), the recurrence is

$$P_L(n) = P_L(n-1) + (n-1)(n-2)P_L(n-3) \quad (10.3)$$

The sequence of numbers of these permutations (whose order divides 3) is entry A001470 of [26]:

n: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...  
 P\_L(n): 1, 1, 1, 3, 9, 21, 81, 351, 1233, 5769, 31041, ...

We can compute the sequence via the EGF as follows:

```
? Vec( serlaplace( exp( x^3/3+x^1/1 +0(x^11) ) ) )
[1, 1, 1, 3, 9, 21, 81, 351, 1233, 5769, 31041]
```

## 10.2 Outline of the algorithm

We use a generalization of the algorithm described in section 9.4 on page 59. A permutation of length  $n$  is generated in two steps: first an integer partition of  $n$  into parts  $\in L$  is created, then a permutation with the cycle type given by the partition is generated. The latter is given as Algorithm 5.4 on page 28 so we only discuss the generation of the partitions.

At each step a partition of  $n$  is generated by adding some  $l \in L$  to a partition of  $n - l$ . A partition of  $n$  into parts  $\in L$  (with multiplicities  $m_k \geq 0$ )

$$n = \sum_{k=1}^u m_k l_k \quad (10.4)$$

is stored as a vector

$$M(n) := [m_1, m_2, \dots, m_u] \quad (10.5)$$

We further assume that  $l_1 < l_2 < \dots < l_u$  so that  $l_u$  is the greatest cycle length allowed. The list of the last  $l_u$  partitions has to be maintained as any of those may be chosen to create the next partition. The size of each partition is  $O(u)$  so the list is of size  $O(u l_u)$ .

The choices for partitions to be used for creating the next partition is in general not amongst all  $l_u$  in the list. We call  $n$  *admissible* (with respect to  $L$ ) if  $P_L(n) \neq 0$ . A value  $n$  is admissible if and only if a partition of  $n$  into parts  $\in L$  exists.

Set  $g := 0$  if  $L$  is empty,  $g := l_1$  if  $L$  contains just one element, and otherwise set  $g := \gcd(l \in L)$ . If  $g \neq 1$  then only values of  $n$  which are multiples of  $g$  are admissible. If  $g = 1$  there may still occur values of  $n$  that are not admissible, for example with  $L = \{3, 5\}$  the sequence of numbers of partitions starts as

```
? Vec( 1/((1-x^3)*(1-x^5)) +0(x^22) )
[1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 2, 2]
```

so the values 1, 2, 4, and 7 are not admissible. The values in the sequence are zero if and only if the number of permutations is zero:

```
? Vec( serlaplace( exp( x^3/3+x^5/5 +0(x^22) ) ) )
[1, 0, 0, 2, 0, 24, 40, 0, 2688, 2240, 72576, 443520, ...]
```

For generating a partition of  $n$  we need a list  $H$  (for ‘history’) of pairs  $H(k) = (P_L(k), M(k))$  for  $n-1 \leq k \leq n-l_u$  where  $M(k)$  is defined by (10.5). If  $P_L(n-l) = 0$  for all  $l \in L$  then we mark  $n$  as inadmissible (that is, we set  $H(n) = (0, *)$  where the second component is immaterial). Otherwise we choose  $k \in L$  according to the probabilities obtained by dividing identity (10.1a) by the term on its left

$$1 = \sum_{l \in L} \frac{P_L(n-l) F(n-l, l-1)}{P_L(n)} \quad (10.6)$$

Then we set  $H(n) = (P_L(n), M(n))$  where

$$M(n) = M(k) + E_k \quad (10.7)$$

and  $E_k$  is the unit vector with a single one at the position of  $m_k$ .

Using the probabilities of relation (10.6) introduces a bias in the partitions according to the number of permutations of cycle type corresponding to the partitions. The non-uniformity in the distribution of the partitions is such that the permutations are uniformly distributed.

### 10.3 Algorithm

In the following algorithm we use the cardinalities from relation (10.1a) (instead of the probabilities from (10.6)).

**Algorithm 10.1** (CycSpecPart). *Generate an integer partition  $M$  of  $N$  into parts  $\in L = \{l_1, l_2, \dots, l_u\}$ , biased such that the permutation whose cycle type corresponds to  $M$  is uniformly chosen from all permutations with maximal cycle spectrum  $L$ . Return the pair  $(P_L(N), M(N))$  where  $P_L(N)$  is the number of length- $N$  permutations with maximal cycle spectrum  $L$  and  $M(N)$  is the generated partition.*

1. Set  $E$  to the length- $u$  vector of zeros (empty partition).
2. If  $n < 0$  return  $(0, E)$  (negative  $N$  are not admissible).
3. Set  $u := |L|$ .
4. Set  $H(1) := (1, E)$  (top entry in history, corresponding to  $n = 0$ ).
5. For  $k := 2, 3, \dots, l_u$  set  $H(k) := (0, E)$  (history entries for  $n < 0$ ).
6. Set  $n := 1$  (loop variable).
7. If  $n < N$  return  $H(1)$ .
8. (here  $H(k) = (P_L(n - k), M(n - k))$ ).
9. For  $j := 1, 2, \dots, u$  do: (create vector  $[p(1), p(2), \dots, p(u)]$  of cardinalities  
 set  $b := l_j$  (offset in history),  
 set  $x := P_{n-b}$  (this is the first entry in the pair  $H(b)$ ),  
 set  $p(u) := x F(n - 1, b - 1)$  (for probability to choose  $j$  in update).
10. Set  $P_L(n) := \sum_{j=1}^u p(j)$ .
11. If  $P_L(n) = 0$  then set  $H(0) := (0, E)$  and go to step 18 ( $n$  is not admissible).
12. Set  $r := 1 + \mathbf{Z}(P_L(n))$  (we have  $1 \leq r \leq P_L(n)$ ).
13. Set  $c := 0$  (auxiliary cumulative sum for choice).
14. For  $j := 1, 2, \dots, u$  do: (determine choice  $k$ )  
 set  $c := c + p(j)$ , if  $c \geq r$  then set  $k := j$  and terminate this loop.
15. Set  $b := l_k$  (will update with entry  $H(b)$ ).
16. Set  $M_0 := M(b)$  (second entry in  $H(b)$ ), increase the  $k$ -th component by 1 (update partition: using one more part of size  $l_k$ ).

17. Set  $H(0) := (P_L(n), M_0)$ .
18. For  $j := l_u, l_u - 1, \dots, 3, 2$  set  $H(j) := H(j - 1)$  (update history).
19. Set  $H(1) := H(0)$  (entry for  $n$ ).
20. Set  $n := n + 1$  and go to step 7.

The algorithm for the permutation is

**Algorithm 10.2** (CycSpec). *Generate a length- $n$  permutation with maximal cycle spectrum  $L = \{l_1, l_2, \dots, l_u\}$ .*

1. Compute a pair  $H(n) = (P_L(n), M(n))$  using algorithm Algorithm 10.1.
2. If  $P_L(n) = 0$  then signal error and exit.
3. Use Algorithm 5.4 on page 28 with arguments  $n$ ,  $M(n)$ , and  $L$  to generate the permutation with cycle type described by  $M(n)$  and  $L$ .

Assume  $n$  is admissible with respect to  $L$ . The algorithm generates only permutations with cycle type  $L$ , by construction. All such permutations are generated uniformly as guaranteed by relation (10.1a).  $\square$

## 10.4 Implementation

We use the GP language [21] in order to keep the implementation readable. The following routine is used to compute  $F(n, e)$ :

```

1  ffactpow(n,e)=
2  {
3      local(f=1);
4      for (k=0, e-1, f*=(n-k) );
5      return( f );
6  }
```

The routine takes the arguments  $N = n$ ,  $L = L$ , and  $VB \in \{0, 1\}$  to determine whether intermediate quantities are printed in the process. Figure 10.1 shows the output with the generation of a partition into parts  $\in L = \{3, 5\}$  for a length-8 permutation with cycle spectrum  $L$ . The output is done in the lines of the form

```
if ( VB, print(...) );
```

The initialization part of the routine is

```

1  gen_part(L,N, VB=0)=
2  {
3      local(u,lu);  \\ number of allowed parts, greatest part
4      local(H, H0);  \\ history, new entry in history
5      local(E);     \\ aux: empty partition
6      local(pp, pln);  \\ probabilities (as cardinalities)
7      local(cp);   \\ aux: cumulative sums with choice
8      local(k, lk);  \\ choice from history
9
10     u=length(L);
11     E = vector(u);  \\ empty partition
12
13     if ( N<0, return( [0,E] ) );  \\ negative N is not admissible
14
15     lu = L[u];     \\ number of entries in history (greatest part in L)
16     H=vector(lu);  \\ pairs (as vectors) as history
```



```

? L={3,5} \\ allowed parts
? gen_part(L,8,1) \\ generate length-8 permutation
[1, " H=", [[1, [0, 0]], [0, [0, 0]], [0, [0, 0]], [0, [0, 0]], [0, [0, 0]]]
[1, " pp=", [0, 0], " pln=", 0]
[1, " no choice available"]
[1, " new=", [0, [0, 0]]]
[2, " H=", [[0, [0, 0]], [1, [0, 0]], [0, [0, 0]], [0, [0, 0]], [0, [0, 0]]]
[2, " pp=", [0, 0], " pln=", 0]
[2, " no choice available"]
[2, " new=", [0, [0, 0]]]
[3, " H=", [[0, [0, 0]], [0, [0, 0]], [1, [0, 0]], [0, [0, 0]], [0, [0, 0]]]
[3, " pp=", [2, 0], " pln=", 2]
[3, " rnd=", 1, " ==> choice: k=", 1, " lk=", 3]
[3, " new=", [2, [1, 0]]]
[4, " H=", [[2, [1, 0]], [0, [0, 0]], [0, [0, 0]], [1, [0, 0]], [0, [0, 0]]]
[4, " pp=", [0, 0], " pln=", 0]
[4, " no choice available"]
[4, " new=", [0, [0, 0]]]
[5, " H=", [[0, [0, 0]], [2, [1, 0]], [0, [0, 0]], [0, [0, 0]], [1, [0, 0]]]
[5, " pp=", [0, 24], " pln=", 24]
[5, " rnd=", 13, " ==> choice: k=", 2, " lk=", 5]
[5, " new=", [24, [0, 1]]]
[6, " H=", [[24, [0, 1]], [0, [0, 0]], [2, [1, 0]], [0, [0, 0]], [0, [0, 0]]]
[6, " pp=", [40, 0], " pln=", 40]
[6, " rnd=", 26, " ==> choice: k=", 1, " lk=", 3]
[6, " new=", [40, [2, 0]]]
[7, " H=", [[40, [2, 0]], [24, [0, 1]], [0, [0, 0]], [2, [1, 0]], [0, [0, 0]]]
[7, " pp=", [0, 0], " pln=", 0]
[7, " no choice available"]
[7, " new=", [0, [0, 0]]]
[8, " H=", [[0, [0, 0]], [40, [2, 0]], [24, [0, 1]], [0, [0, 0]], [2, [1, 0]]]
[8, " pp=", [1008, 1680], " pln=", 2688]
[8, " rnd=", 57, " ==> choice: k=", 1, " lk=", 3]
[8, " new=", [2688, [1, 1]]]
[2688, [1, 1]] \\ return 8==1*3+1*5, and there are 2688 such permutations
[1, 0, 0, 1, 0, 1, 1, 0, 1, 1, ... ] \\ numbers of partitions
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ... ] \\ == n
[1, 0, 0, 2, 0, 24, 40, 0, 2688, 2240, ... ] \\ number of permutations

```

Figure 10.1: Quantities with generating a partition for a length-8 permutation with maximal cycle spectrum  $L = \{3, 5\}$  (the cycle spectrum is indeed  $L$ ).

```

17
18 \\ H[k] contains partition n-k, we start with n==1:
19 for (k=2, lu, H[k]=[0,E] ); \\ not admissible (negative n)
20 H[1] = [1, E]; \\ admissible, empty partition (n==0)

```

The main part updates the history until the top entry corresponds to the required permutation length:

```

1 for (n=1, N,
2 if ( VB, printp([n, " H=", H]); );
3
4 \\ probabilities (as cardinalities) biased for permutations:
5 pp = vector(u,j, (H[L[j]][1]) * ffactpow(n-1,L[j]-1) );
6
7 \\ number of length-n permutations with cycle spectrum L:
8 pln = sum(j=1,u, pp[j]);
9
10 if ( VB, print([n, " pp=",pp, " pln=",pln]); );
11
12 if ( pln==0, \\ no admissible entry can be reached
13 H0=[0, E]; \\ ... so n is not admissible

```

```

14         if ( VB, print([n, " no choice available"]); );
15
16     , \\ else (make choice and update)
17
18         rnd = 1 + random(pln); \\ vectors are one-based
19         k = 0; \\ index of choice
20         cp = 0; \\ aux: cumulative sums of probabilities
21         for (j=1,u, cp+=pp[j]; if ( cp>=rnd, k=j; break(); ) );
22
23         lk = L[k]; \\ this far back in history
24         if ( VB, print([n, " rnd=",rnd,
25             " ==> choice: k=", k, " lk=", lk ])); );
26         H0=H[lk];
27         H0[1] = pln;
28         H0[2][k] += 1; \\ update partition
29     );
30
31     \\ move entries in history down (shift H):
32     forstep (j=lu, 2, -1, H[j]=H[j-1]);
33     \\ ... and replace top entry:
34     H[1] = H0; \\ this is the entry corresponding to n
35
36     if ( VB, print([n, " new=", H[1]]); );
37     if ( VB, print(); );
38 );
39
40 return( H[1] );
41 }

```

For better efficiency, the history should be kept in a data structure that allows for a less costly update, such as a ringbuffer. Still, even if probabilities (instead of cardinalities) are used and all arithmetic operations are assumed to be  $O(1)$ , the algorithm is  $O(n^2)$  in the general case.

It may be possible to replace probabilities by cumulative probabilities and use binary search instead of linear search, but we will not attempt to do this here.

If the weights for permutations are removed, we obtain a routine for unbiased random compositions (not partitions). For the corresponding routine replace (at the start of the main loop)

```

    \\ probabilities (as cardinalities) biased for permutations:
    pp = vector(u,j, (H[L[j]][1]) * ffactpow(n-1,L[j]-1) );

```

by

```

    \\ probabilities for unbiased compositions:
    pp = vector(u,j, nL[L[j]] );

```

The size  $lj$  chosen in each step has to be recorded, keeping the order. The list of these quantities is a composition of  $n$  into parts  $\in L$ .

## Chapter 11

# Derangements

A permutation with only cycles of length  $\geq 2$  does not contain any fixed point and is called a *derangement*. We give algorithms for random derangements and also for derangements with all cycles of length  $\geq m$ .

### 11.1 The number of derangements

Let  $D(n)$  be the number of derangements of  $n$  elements and  $e = \exp(1)$ . We have  $D(0) = 1$ ,  $D(1) = 0$ ,  $D(2) = 1$ , and for  $n \geq 2$  (see [4, p. 182] or [24, p. 74])

$$D(n) = (n-1)[D(n-1) + D(n-2)] \quad (11.1)$$

The sequence of numbers  $D(n)$  (entry A000166 in [26]) starts as

$n$ :	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	...
$D(n)$ :	0,	1,	2,	9,	44,	265,	1854,	14833,	133496,	1334961,	14684570,	...
$n!$ :	1,	2,	6,	24,	120,	720,	5040,	40320,	362880,	3628800,	39916800,	...

We will further use the following identities:

$$D(n) = n! \sum_{k=0}^n \frac{(-1)^k}{k!} \quad \text{for } n \geq 0 \quad (11.2a)$$

$$D(n) = \lfloor (n! + 1)/e \rfloor \quad \text{for } n \geq 1 \quad (11.2b)$$

### 11.2 Branching probabilities

In each step of the algorithm for a random derangement we select two different elements and swap them to join the cycles containing them. Then we either remove one or both of them from the set of available elements. Assume that  $n$  elements are to be processed. If both chosen elements are removed, the cycle is closed and it remains to generate a derangement of  $n-2$  elements. Otherwise the cycle containing the chosen elements stays open for further extension and it remains to generate a derangement of  $n-1$  elements.

If  $n$  elements are left, the probability of closing the cycle is

$$b(n) = \frac{(n-1)D(n-2)}{D(n)} \quad (11.3)$$

This can be seen by dividing relation (11.1) by  $D(n)$ :

$$1 = \frac{(n-1)D(n-1)}{D(n)} + \frac{(n-1)D(n-2)}{D(n)} \quad (11.4)$$

By relation (11.2b) we have  $b(n) \approx (n-1)(n-2)!/n! = 1/n$  for  $n$  large. Already for  $n > 30$  we have  $|b(n) - 1/n| < 2^{-106}$ . This is due to the rapid convergence (to  $1/e$ ) of the sum in relation (11.2a): we have  $b(n) = 1/n + O(1/n!)$ .

Therefore we precompute the branching probabilities only for  $n < 32$ .

**Algorithm 11.1** (DerangeBranches). *Precompute array  $B = [b(2), \dots, b(31)]$  of probabilities for closing the cycle.*

1. Set  $b(2) := 1.0$  (floating-point type).
2. Set  $d_0 := 1.0$ ,  $d_1 := 0.0$ , and  $n_1 := 1.0$ .
3. For  $n := 3, 4, \dots, 31$  do:
  - (a) Set  $d_2 := d_1$ ,  $d_1 := d_0$ ,  $n_1 := n_1 + 1.0$ , and  $d_0 := n_1(d_1 + d_2)$ .
  - (b) Set  $b(n) := n_1 d_2 / d_0 (= (n-1)D(n-2)/D(n))$ .

In the following implementation the variables  $dn0 = d_0$ ,  $dn1 = d_1$ , and  $dn2 = d_2$  are respectively the values  $D(n)$ ,  $D(n-1)$ , and  $D(n-2)$ , as floating-point numbers:

```

1 // number of precomputed branch ratios:
2 #define NUM_PBR 32 // OK for up to 106-bit significand
3
4 inline void init_derange_branch_ratios(double *b)
5 // Precompute branching probabilities for random derangements.
6 // n == 1, 2, 3, 4, 5, 6, 7, 8, ...
7 // b[] == 0, 1, 0, 0.3333, 0.1818, 0.1698, 0.1423, 0.1250, ...
8 // b[n-1] == (n-1) * D(n-2) / D(n)
9 {
10     b[0] = 0.0; // unused
11     b[1] = 1.0;
12     double dn0 = 1.0, dn1 = 0.0, n1 = 1.0;
13     for (ulong n=3; n<=NUM_PBR; ++n)
14     {
15         const double dn2 = dn1;
16         dn1 = dn0;
17         n1 += 1.0;
18         dn0 = n1*(dn1 + dn2);
19         b[n-1] = (n1) * dn2/dn0; // == (n-1) * D(n-2) / D(n)
20     }
21 }
```

For greater values of NUM\_PBR or with floating-point types with smaller range the variable  $dn0$  will overflow. This can be avoided if normalization is used. To do so, replace the body of the for-loop with the following statements:

```

1     double dn2 = dn1;
2     dn1 = 1.0; // ==dn0;
3     n1 += 1.0;
4     dn0 = n1*(dn1 + dn2);
5     dn1 /= dn0;
6     dn2 /= dn0;
7     // dn0 /= dn0; // i.e. dn0==1.0
8     b[n-1] = n1 * dn2; // == (n-1) * D(n-2) / D(n)
```

## 11.3 Algorithm

The main algorithm works for  $n \geq 2$  (there is no derangement of length 1). We assume that the array  $B$  of branching probabilities has been precomputed.

**Algorithm 11.2** (Derangement). *Apply a random derangement  $\in \mathcal{S}_n$  to  $F = [f(0), f(1), \dots, f(n-1)]$ .*

1. Set  $S := \{0, 1, \dots, n-1\}$  (initialize set).
2. Select an element  $r_1$  from  $S$ .
3. Select a random  $r_2 \neq r_1$  from  $S$ .
4. Swap  $f(r_1)$  with  $f(r_2)$  (join the cycles containing  $f(r_1)$  and  $f(r_2)$ ).
5. Set  $r := |S|$ . If  $r \geq 32$  then set  $p := 1.0/r$  else set  $p := b(r)$  (probability).
6. Set  $t := \mathbf{R}()$  (a random real number  $0 \leq t < 1$ ).
7. If  $t < p$  then remove both  $r_1$  and  $r_2$  from  $S$  (close cycle).
8. Otherwise (i.e.  $t \geq p$ ) remove  $r_1$  from  $S$  (leave cycle open).
9. If there are at least 2 elements in  $S$  then goto step 2.

If we use the method from section 5.1 on page 25 for maintaining the set  $S$  then the algorithm is  $O(n)$  which is optimal.

The method is (essentially) given in [15]. However, the paper leaves open how to compute the probabilities  $b(n)$  in  $O(1)$ , and the method for maintaining the set of available elements used there is slightly less efficient.

If many derangements are to be generated, the divisions in step 5 may be avoided with a precomputed look-up table of values  $1.0/r$ . No problems with nonlocal memory access should occur as the elements of the table are accessed in a sequential fashion.

The implementation uses the following function to determine the probability of closing the cycle:

```

1  inline double derange_branch_ratio(const double *b, ulong n)
2  // Return probability for closing cycle with n elements.
3  {
4      if ( n < NUM_PBR ) return b[n];
5      else               return 1.0/(double)n;
6  }
```

The routine for random derangement can use preallocated workspaces for the set  $S$  (variable  $tr$ ) and the array of probabilities (variable  $tb$ ). The first element chosen in each step is the last (in the array representation) of the set  $S$ , with this choice the swap with removing the element can be avoided.

```

1  template <typename Type>
2  inline void random_derange(Type *f, ulong n,
3                          ulong *tr=0,
4                          double *tb=0, bool bi=false)
5  // Permute the elements of f by a random derangement.
6  // Set bi=true to signal that the branch probabilities in tb[]
7  // have been precomputed (via init_derange_branch_ratios()).
```

```

8 // Must have n > 1.
9 {
10     ulong *r = tr;
11     if ( tr==0 ) r = new ulong[n];
12     for (ulong k=0; k<n; ++k) r[k] = k;
13     ulong nr = n; // number of elements available
14     // available positions are r[0], ..., r[nr-1]
15
16     double *b = tb;
17     if ( tb==0 ) { b = new double[NUM_PBR]; bi=false; }
18     if ( !bi ) init_derange_branch_ratios(b);
19
20     while ( nr>=2 )
21     {
22         const ulong x1 = nr-1; // last element
23         const ulong r1 = r[x1];
24
25         const ulong x2 = rand_idx(nr-1); // random element !=last
26         const ulong r2 = r[x2];
27
28         swap2( f[r1], f[r2] ); // join cycles containing f[r1] and f[r2]
29
30         // remove r[x1]=r1 from set:
31         --nr; // swap2(r[x1], r[nr]); // swap not needed if x1==last
32
33         const double rat = derange_branch_ratio(b, nr);
34         const double t = rnd01(); // 0 <= t < 1
35         if ( t < rat ) // close cycle
36         {
37             // remove r[x2]=r2 from set:
38             --nr; swap2(r[x2], r[nr]);
39         }
40         // else cycle stays open
41     }
42
43     if ( tr==0 ) delete [] r;
44     if ( tb==0 ) delete [] b;
45 }

```

## 11.4 Comparison with rejection method

The probability that a random permutation of length- $n$  is a derangement is  $\approx 1/e$  for large  $n$ , as can be seen from relation (11.2b). Therefore the rejection method described in section 8.2 on page 50 is a practical alternative to Algorithm 11.2.

For the comparison we use a routine that improves over the obvious algorithm by detecting fixed points early:

```

1 inline void random_derangement_rej(ulong *p, ulong n)
2 // Random derangement via rejection method.
3 {
4     if ( n<2 ) return; // avoid hang
5     for (ulong k=0; k<n; ++k) p[k] = k;
6
7     again:
8     for (ulong k=n; k>1; --k)
9     {
10         const ulong i = rand_idx(k);
11         swap2(p[k-1], p[i]);
12         // early detection of fixed point:
13         if ( (p[k-1]==k-1) ) goto again;
14     }
15
16     if ( (p[0]==0) ) goto again;

```

17 }

For timing, 100 thousand derangements of 1000 elements (fitting into the first level cache of the machine used) are generated. The rejection method needs 8.25 seconds, while the implementation of Algorithm 11.2 needs 7.76 seconds. If the table of branch ratios is computed for all  $n \leq 1000$  the time needed (again for Algorithm 11.2) is 7.28 seconds (as we avoided the divisions by precomputing the branch ratios).

While Algorithm 11.2 is not much faster on average, it has two advantages. The generation of one derangement is guaranteed to finish in linear time, and, more importantly, it can be generalized as follows.

## 11.5 Length of cycles at least $m$

Let  $D_m(n)$  be the number of length- $n$  permutations with all cycles of length  $\geq m$  (where  $m \geq 2$ ). We have  $D_m(0) = 1$ ,  $D_m(k) = 0$  for  $0 < k < m$ , and the recurrence

$$D_m(n) = (n-1)D_m(n-1) + F(n-1, m-1)D_m(n-m) \quad \text{where} \quad (11.5a)$$

$$F(n, m) := (n)(n-1)(n-2) \dots (n-(m-1)) \quad (11.5b)$$

and  $F(n, m) = 1$  for  $m \leq 0$  (compare to relation (10.1a) on page 63).

In each step of the algorithm we use the probabilities

$$1 = \frac{(n-1)D_m(n-1)}{D_m(n)} + \frac{F(n-1, m-1)D_m(n-m)}{D_m(n)} \quad (11.6)$$

to choose between two possible actions. With probability  $(n-1)D_m(n-1)/D_m(n)$  we select two distinct elements, swap them, and remove one of them from the set of available elements. With probability  $F(n-1, m-1)D_m(n-m)/D_m(n)$  (the complement of the former) we select  $m$  distinct elements, apply a random cyclic permutation of them to join all their (initially distinct) cycles and remove all of them from the set.

**Algorithm 11.3** (Der-M). *Apply a random permutation  $\in \mathcal{S}_n$  (where  $n \geq m$ ) with only cycles of length  $\geq m$  where  $m \geq 2$  to  $F = [f(0), f(1), \dots, f(n-1)]$ .*

1. Set  $S := \{0, 1, \dots, n-1\}$  (initialize set).
2. Set  $s := |S|$ . If  $s < 2$  then terminate.
3. Set  $p := (s-1)D_m(s-1)/D_m(s)$  (probability, read from precomputed array).
4. Select an element  $r_1$  and remove it from  $S$  (no need for random selection).
5. Set  $t := \mathbf{R}()$  (a random real number  $0 \leq t < 1$ ).
6. If  $t < p$  then do: (join two cycles)
  - (a) Select a random  $r_2 \neq r_1$  from  $S$ .
  - (b) Swap  $f(r_1)$  with  $f(r_2)$  (join the cycles containing  $f(r_1)$  and  $f(r_2)$ ).

7. If  $t \geq p$  then do: (join  $m$  cycles and remove resulting cycle from  $S$ )
  - (a) Randomly select  $m - 1$  distinct elements  $r_2, r_3, \dots, r_m$  and remove them from  $S$ .
  - (b) Using Algorithm 3.12 on page 15 apply a random cyclic permutation to  $f(r_1), f(r_2), \dots, f(r_m)$  (join the  $m$  cycles containing these elements).
8. Go to step 2.

The algorithm is  $O(n)$  which is optimal. The complexity does not depend on  $m$ .

We now give an implementation for the derangements with all cycles of length  $\geq 3$ . The sequence of numbers  $D_3(n)$  of such permutations starts as (see A038205 in [26]):

```

n: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
D3(n): 1, 0, 0, 2, 6, 24, 160, 1140, 8988, 80864, 809856, 8907480, ...

```

The recurrence (11.5a) specializes as

$$D_3(n) = (n-1)D_3(n-1) + (n-1)(n-2)D_3(n-3) \quad (11.7)$$

An array of precomputed probabilities will be used:

```

1 inline void init_derange3_branch_ratios(double *b, ulong N)
2 // Precompute branching probabilities for random derangements
3 // with all cycles of length >= 3.
4 // n == 2, 3, 4, 5, 6, 7, 8, 9
5 // 0, 0, 1, 1, 3/4, 16/19, 95/107, 321/361,
6 // b[] == [0, 0, 1, 1, 0.75, 0.8421, 0.8878, 0.8891,
7 // b[n-2] == (n-1) * D3(n-1) / D3(n) (i.e. offset=2)
8 {
9     b[0] = 0.0; b[1]=0.0;
10    double dn0 = 2.0, dn1 = 0.0, dn2 = 0.0;
11    double n1 = 2.0, n2 = 1.0;
12
13    for (ulong n=2; n<N; ++n)
14    {
15        const double dn3 = dn2;
16        dn2 = dn1;
17        dn1 = dn0;
18        n1 += 1.0; n2 += 1.0;
19        dn0 = n1*(dn1 + n2*dn3);
20        b[n] = (n1) * dn1/dn0; // == (n-1) * D3(n-1) / D3(n)
21    }
22 }

```

For  $n$  large normalization is needed, replace the loop body with the following statements:

```

1     double dn3 = dn2;
2     dn2 = dn1;
3     dn1 = 1.0; // ==dn0;
4     n1 += 1.0; n2 += 1.0;
5     dn0 = n1*(dn1 + n2*dn3);
6     dn1 /= dn0;
7     dn2 /= dn0;
8     dn3 /= dn0;
9     // dn0 /= dn0; // i.e. dn0==1.0
10    b[n] = n1 * dn1; // == (n-1) * D3(n-1) / D3(n)

```

We have  $\lim_{n \rightarrow \infty} D_3(n)/n! = \exp(-3/2)$  and the probabilities rapidly approach  $(n-1)/n$ . So we could use a technique of mixed lookup for small  $n$  and on-the-fly computation for large  $n$ . For the sake of simplicity we omit this.



The general expression for the limit is [28, rel. 5.2.9, p. 176]

$$\lim_{n \rightarrow \infty} D_m(n)/n! = \exp(-H_{m-1}) \quad (11.8)$$

where  $H_k = \sum_{j=1}^k 1/j$ . The exponential generating function is

$$\sum_{n=0}^{\infty} D_m(n) \frac{x^n}{n!} = \frac{\exp\left(-\sum_{k=1}^{m-1} \frac{x^k}{k}\right)}{1-x} \quad (11.9)$$

The main routine for  $m = 3$  is

```

1  template <typename Type>
2  inline void random_derange3(Type *f, ulong n,
3      ulong *tr=0,
4      double *tb=0, bool bi=false)
5  // Permute the elements of f by a random derangement
6  // with all cycles of length >= 3.
7  // Set bi:=true to signal that the branch probabilities in tb[]
8  // have been precomputed (via init_derange3_branch_ratios()).
9  // Must have n >= 3.
10 {
11     ulong *r = tr;
12     if ( tr==0 ) r = new ulong[n];
13     for (ulong k=0; k<n; ++k) r[k] = k;
14     ulong nr = n; // number of elements available
15     // available positions are r[0], ..., r[nr-1]
16     double *b = tb;
17     if ( tb==0 ) { b = new double[n]; bi=false; }
18     if ( !bi ) init_derange3_branch_ratios(b, n);
19     while ( nr>=2 )
20     {
21         const double rat = b[nr-2];
22         const double t = rnd01(); // 0 <= t < 1
23         const ulong x1 = nr-1; // last element
24         const ulong r1 = r[x1];
25         // remove r[x1]=r1 from set:
26         --nr; // swap2(r[x1], r[nr]); // (swap not needed if x1==last)
27
28         if ( t < rat ) // join two cycles, leave resulting cycle open
29         {
30             const ulong x2 = rand_idx(nr); // random element !=last
31             const ulong r2 = r[x2];
32             swap2( f[r1], f[r2] ); // join cycles containing f[r1] and f[r2]
33         }
34         else // connect cycles of 3 elements and remove all
35         {
36             const ulong x2 = rand_idx(nr); // random element !=last
37             const ulong r2 = r[x2];
38             --nr; swap2(r[x2], r[nr]); // remove r[x2]=r2 from set:
39
40             const ulong x3 = rand_idx(nr); // random element !=both
41             const ulong r3 = r[x3];
42             --nr; swap2(r[x3], r[nr]); // remove r[x3]=r3 from set:
43
44             // random cyclic permutation of all three elements:
45             swap2( f[r1], f[r3] ); // [c,b,a]
46             if ( rnd01() < 0.5 ) swap2( f[r1], f[r2] ); // [b,c,a]
47             else swap2( f[r2], f[r3] ); // [c,a,b]
48         }
49     }
50 }

```

```

53     }
54
55     if ( tr==0 ) delete [] r;
56     if ( tb==0 ) delete [] b;
57 }

```

In the routine for general  $m$  only the part for the joining  $m$  cycles (the else branch in the main loop) has to be adapted. We firstly remove  $m$  elements from the set. Their values do *not* need to be recorded as they end up in the  $m$  consecutive elements  $r[nr-m]$ ,  $r[nr-m+1]$ ,  $r[nr-m+1]$ , ...,  $r[nr-1]$ . The remaining task is to apply a cyclic permutation to  $f[r[nr-m]]$ ,  $f[r[nr-m+1]]$ ,  $f[r[nr-m+1]]$ , ...,  $f[r[nr-1]]$ . This is easily achieved with Algorithm 3.12 on page 15. Assuming one element has already been removed as in given implementation (first few lines in the while-loop), we can use the following code:

```

1         else // connect cycles of m elements and remove all
2         {
3             for (ulong j=1; j<m; ++j) // get m-1 random positions
4             {
5                 ulong x = rand_idx(nr);
6                 --nr; swap2(r[x], r[nr]);
7             }
8             random_permute_positions_cyclic(f, m, r+nr-m);
9         }
10

```

A recursive method for generating combinatorial objects of certain types that include permutations with all cycles of size  $\geq m$  is given in [5] (see also [10]). The algorithm given there has complexity  $O(n \log n)$ . Amusingly, the problem of generating a random derangement is posed as an exercise (without solution) in [23, ex. 19.a, p. 201].

## Appendix A

# Bit-array

We give the implementation of the bit-array used in chapter 4 on page 19. The described class offers several convenient methods that do not exist for the bit-fields of the C-language, such as the search for the next set bit. The routines here are also a guide for implementing the corresponding mechanisms in languages that do not have built-in Boolean arrays.

The value `BITS_PER_LONG` has to be set to the number of bits in a word of type unsigned long (abbreviated as `ulong`).

```

1  class bitarray
2  // Bit-array class mostly for use as memory saving array of Boolean values.
3  // Valid index is 0...nb-1 (as usual in C arrays).
4  {
5  public:
6      ulong *f_; // bit bucket
7      ulong n_; // number of bits
8      ulong nfw_; // number of words where all bits are used, may be zero
9      ulong mp_; // mask for partially used word if there is one, else zero
10     // (ones are at the positions of the _unused_ bits)
11     bool myfq_; // whether f[] was allocated by class

```

We keep the data public to avoid the necessity for various get and set methods. The names of the class variables end in an underscore, making them easily identifiable in the class methods.

```

1  private:
2      ulong ctor_core(ulong nbits)
3      {
4          n_ = nbits;
5
6          // nw: number of words (incl. partially used), nw>=1
7          ulong nw = n_ / BITS_PER_LONG; // number of words
8
9          // nbl: number of bits used in last (partially used) word, 0 if mw==mfw
10         ulong nbl = n_ - nw*BITS_PER_LONG; // number of bits used in last word
11         nfw_ = nw; // number of fully used words
12
13         if ( 0!=nbl ) // there is a partially used word
14         {
15             ++nw; // increase total number of words
16             mp_ = (~0UL) >> (BITS_PER_LONG-nbl); // correct mask for last word
17         }
18         else mp_ = 0UL;
19
20         return nw;
21     }
22

```

The constructor allocates memory by default. If the second argument is nonzero, it must point to an accessible memory range of sufficient size:

```

1 public:
2   bitarray(ulong nbits, ulong *f=0)
3     // nbits must be nonzero
4     {
5       ulong nw = ctor_core(nbits);
6       if ( f!=0 )
7       {
8         f_ = f;
9         myfq_ = false;
10      }
11      else
12      {
13        f_ = new ulong[nw];
14        myfq_ = true;
15      }
16    }
17
18    ~bitarray() { if ( myfq_ ) delete [] f_; }

```

The following auxiliary definitions are used to access the  $n$ -th bit in the array  $f_$ :

```

1 #define DIVMOD(n, d, bm) \
2   ulong d = n / BITS_PER_LONG; \
3   ulong bm = 1UL << (n % BITS_PER_LONG);
4
5 #define DIVMOD_TEST(n, d, bm) \
6   ulong d = n / BITS_PER_LONG; \
7   ulong bm = 1UL << (n % BITS_PER_LONG); \
8   ulong t = bm & f_[d];

```

The division and remnant computations will respectively be optimized to shifts and bit-wise ANDs by the compiler.

The methods for testing, setting, clearing, and changing one bit are

```

1   ulong test(ulong n) const
2     // Test whether n-th bit set.
3     {
4       DIVMOD_TEST(n, d, bm);
5       return t;
6     }
7
8   void set(ulong n)
9     // Set n-th bit.
10    {
11      DIVMOD(n, d, bm);
12      f_[d] |= bm;
13    }
14
15   void clear(ulong n)
16     // Clear n-th bit.
17     {
18      DIVMOD(n, d, bm);
19      f_[d] &= ~bm;
20    }
21
22   void change(ulong n)
23     // Toggle n-th bit.
24     {
25      DIVMOD(n, d, bm);
26      f_[d] ^= bm;
27    }
28
29   ulong test_set(ulong n)
30     // Test whether n-th bit is set and set it.
31     {
32      DIVMOD_TEST(n, d, bm);
33      f_[d] |= bm;
34      return t;

```

```

35     }
36
37     ulong test_clear(ulong n)
38     // Test whether n-th bit is set and clear it.
39     {
40         DIVMOD_TEST(n, d, bm);
41         f_[d] &= ~bm;
42         return t;
43     }
44
45     ulong test_change(ulong n)
46     // Test whether n-th bit is set and toggle it.
47     {
48         DIVMOD_TEST(n, d, bm);
49         f_[d] ^= bm;
50         return t;
51     }
52

```

The methods for setting, clearing, or testing all bits in the array are

```

1     void clear_all()
2     // Clear all bits.
3     {
4         for (ulong k=0; k<nfw_; ++k) f_[k] = 0;
5         if ( mp_ ) f_[nfw_] = 0;
6     }
7
8     void set_all()
9     // Set all bits.
10    {
11        for (ulong k=0; k<nfw_; ++k) f_[k] = ~OUL;
12        if ( mp_ ) f_[nfw_] = ~OUL;
13    }
14
15    bool all_set_q() const
16    // Return whether all bits are set.
17    {
18        for (ulong k=0; k<nfw_; ++k) if ( ~f_[k] ) return false;
19        if ( mp_ )
20        {
21            ulong z = f_[nfw_] & mp_;
22            if ( z!=mp_ ) return false;
23        }
24        return true;
25    }
26
27    ulong all_clear_q() const
28    // Return whether all bits are clear.
29    {
30        for (ulong k=0; k<nfw_; ++k) if ( f_[k] ) return false;
31        if ( mp_ )
32        {
33            ulong z = f_[nfw_] & mp_;
34            if ( z!=0 ) return false;
35        }
36        return true;
37    }
38

```

Methods for finding the next set or clear bit from a given index on are

```

1     ulong next_set(ulong n) const
2     // Return index of next set or value beyond end.
3     // Note: the given index n is included in the search
4     {
5         while ( (n<n_) && (!test(n)) ) ++n;
6         return n;
7     }
8

```

```
9     ulong next_clear(ulong n) const
10    // Return index of next clear or value beyond end.
11    // Note: the given index n is included in the search
12    {
13        while ( (n<n_) && (test(n)) ) ++n;
14        return n;
15    }
16 };
```

## Appendix B

### Left-right array

The *left-right array* (or *LR-array*) is a data structure to keep track of a range of indices  $0, \dots, n - 1$ . Every index can have two states, *free* or *set*. The LR-array implements the following operations in time  $O(\log n)$ : marking the  $k$ -th free index as set; marking the  $k$ -th set index as free; for the  $i$ -th (absolute) index, finding how many indices of the same type (free or set) are left (or right) to it (including or excluding  $i$ ).

The underlying algorithm is a binary search and similar functionality could be obtained using equivalent data structures such as binary trees. The LR-array is especially suited for the fast conversion between permutations and inversion tables, see section 7.2 on page 45. If performance is of utmost importance the optimizations suggested at the end of this section should be used. The implementation described in the following is taken from [2, sect. 4.7]. The data is

```

1  class left_right_array
2  {
3  public:
4      ulong *fl_; // Free indices Left (including current element)
5                  // in bsearch interval
6      bool *tg_; // tags: tg[i]==true if and only if index i is free
7      ulong n_; // total number of indices
8      ulong f_; // number of free indices

```

As with the bit-array, we keep the data public to avoid the necessity for various get and set methods. The arrays used have  $n$  elements:

```

1  public:
2      left_right_array(ulong n)
3      {
4          n_ = n;
5          fl_ = new ulong[n_];
6          tg_ = new bool[n_];
7          free_all();
8      }
9
10     ~left_right_array()
11     {
12         delete [] fl_;
13         delete [] tg_;
14     }
15
16     ulong num_free() const { return f_; }
17     ulong num_set() const { return n_ - f_; }

```

The initialization routine `free_all()` uses a variation of the binary search algorithm. The crucial observation is that the set of all intervals occurring with binary search is fixed if the size of the searched array is fixed. For any interval  $[i_0, i_1]$  the element `fl[t]` where  $t = \lfloor (i_0 + i_1)/2 \rfloor$  contains the number of free positions in  $[i_0, t]$ .

```

1 private:
2     void init_rec(ulong i0, ulong i1)
3         // Set elements of fl[0,...,n-2] according to empty array a[].
4         // The element fl[n-1] needs to be set to 1 afterwards.
5         // Work is O(n).
6     {
7         if ( (i1-i0)!=0 )
8         {
9             ulong t = (i1+i0)/2;
10            init_rec(i0, t);
11            init_rec(t+1, i1);
12        }
13        fl_[i1] = i1-i0+1; // number of elements in [i0, i0+1, ..., i1]
14    }
15
16 public:
17     void free_all()
18         // Mark all indices as free.
19     {
20         f_ = n_;
21         for (ulong j=0; j<n_; ++j) tg_[j] = true;
22         init_rec(0, n_-1);
23         fl_[n_-1] = 1;
24     }

```

Compare the method `init_rec()` to the following routine for binary search:

```

1 template <typename Type>
2 ulong bsearch(const Type *f, ulong n, const Type v)
3 // Return index of first element in f[] that equals v
4 // Return n if there is no such element.
5 // f[] must be sorted in ascending order.
6 // Must have n!=0
7 {
8     ulong i0=0, i1=n-1;
9     while ( i0 != i1 )
10    {
11        ulong t = (i1+i0)/2;
12        if ( f[t] < v ) i0 = t + 1;
13        else i1 = t;
14    }
15    if ( f[i1]==v ) return i1;
16    else return n; // element not found
17 }

```

The following method returns the  $k$ -th free index:

```

1     ulong get_free_idx(ulong k) const
2         // Return the k-th ( 0 <= k < num_free() ) free index.
3         // Return ~OUL if k is out of bounds.
4         // Work is O(log(n)).
5     {
6         if ( k >= num_free() ) return ~OUL;
7
8         ulong i0 = 0, i1 = n_-1;
9         while ( 1 )
10        {
11            ulong t = (i1+i0)/2;
12            if ( (fl_[t] == k+1) && (tg_[t]) ) return t;
13
14            if ( fl_[t] > k ) // left:
15            {
16                i1 = t;
17            }
18            else // right:
19            {
20                i0 = t+1; k-=fl_[t];
21            }
22        }

```



fl[] = 1 2 3 1 5 1 2 1 1	
a[] = * * * * * * * *	
----- first: -----	
fl[] = 0 1 2 1 4 1 2 1 1	
a[] = 1 * * * * * * * *	
----- last: -----	
fl[] = 0 1 2 1 4 1 2 1 0	
a[] = 1 * * * * * * * 2	
----- first: -----	
fl[] = 0 0 1 1 3 1 2 1 0	
a[] = 1 3 * * * * * * 2	
----- last: -----	
fl[] = 0 0 1 1 3 1 2 0 0	
a[] = 1 3 * * * * * 4 2	
----- first: -----	
fl[] = 0 0 0 1 2 1 2 0 0	
a[] = 1 3 5 * * * * 4 2	

(continued)

	----- last: -----
fl[] = 0 0 0 1 2 1 1 0 0	
a[] = 1 3 5 * * * 6 4 2	
	----- first: -----
fl[] = 0 0 0 0 1 1 1 0 0	
a[] = 1 3 5 7 * * 6 4 2	
	----- last: -----
fl[] = 0 0 0 0 1 0 0 0 0	
a[] = 1 3 5 7 * 8 6 4 2	
	----- first: -----
fl[] = 0 0 0 0 0 0 0 0 0	
a[] = 1 3 5 7 9 8 6 4 2	

Figure B.1: Alternately setting the first and last free position in an LR-array. Asterisks denote free positions, indices  $i$  where  $tg[i]$  is true.

23        }

The following method returns the  $k$ -th free index and marks the corresponding element as set. The necessary changes to the arrays `fl_` and `tg_` are made in the process of the computation:

```

1   ulong get_free_idx_chg(ulong k)
2   // Return the k-th ( 0 <= k < num_free() ) free index.
3   // Return ~OUL if k is out of bounds.
4   // Change the arrays and fl[] and tg[] reflecting
5   // that index i will be set afterwards.
6   // Work is O(log(n)).
7   {
8       if ( k >= num_free() ) return ~OUL;
9
10      --f_;
11
12      ulong i0 = 0, i1 = n_-1;
13      while ( 1 )
14      {
15          ulong t = (i1+i0)/2;
16
17          if ( (fl_[t] == k+1) && (tg_[t]) )
18          {
19              --fl_[t];
20              tg_[t] = false;
21              return t;
22          }
23
24          if ( fl_[t] > k ) // left:
25          {
26              --fl_[t];
27              i1 = t;
28          }
29          else // right:
30          {
31              i0 = t+1; k--=fl_[t];
32          }
33      }
34  }
```

For example, the following program sets alternately the first and last free position

until no free position is left:

```

1     ulong n = 9;
2     ulong *A = new ulong[n];
3     left_right_array LR(n);
4     LR.free_all();
5
6     // PRINT
7     for (ulong e=0; e<n; ++e)
8     {
9         ulong s = 0; // first free
10        if ( 0!=(e&1) ) s = LR.num_free()-1; // last free
11
12        ulong idx2 = LR.get_free_idx_chg(s);
13        A[idx2] = e+1;
14        // PRINT
15    }

```

Its output is shown in figure B.1. The method to free the  $k$ -th set position is

```

1     ulong get_set_idx_chg(ulong k)
2     // Return the k-th ( 0 <= k < num_set() ) set index.
3     // Return ~OUL if k is out of bounds.
4     // Change the arrays and fl[] and tg[] reflecting
5     // that index i will be freed afterwards.
6     // Work is O(log(n)).
7     {
8         if ( k >= num_set() ) return ~OUL;
9
10        ++f_;
11
12        ulong i0 = 0, i1 = n_-1;
13        while ( 1 )
14        {
15            ulong t = (i1+i0)/2;
16            // how many elements to the left are set:
17            ulong slt = t-i0+1 - fl_[t];
18
19            if ( (slt == k+1) && (tg_[t]==false) )
20            {
21                ++fl_[t];
22                tg_[t] = true;
23                return t;
24            }
25
26            if ( slt > k ) // left:
27            {
28                ++fl_[t];
29                i1 = t;
30            }
31            else // right:
32            {
33                i0 = t+1; k-=slt;
34            }
35        }
36    }

```

The following method returns the number of free indices left of  $i$  (and excluding  $i$ ):

```

1     ulong num_FLE(ulong i) const
2     // Return number of
3     // Free indices Left of (absolute) index i (Excluding i).
4     // Work is O(log(n)).
5     {
6         if ( i >= n_ ) { return ~OUL; } // out of bounds
7
8         ulong i0 = 0, i1 = n_-1;
9         ulong ns = i; // number of set element left to i (including i)
10        while ( 1 )
11        {

```

```

12         if ( i0==i1 ) break;
13
14         ulong t = (i1+i0)/2;
15         if ( i<=t ) // left:
16         {
17             i1 = t;
18         }
19         else // right:
20         {
21             ns -= fl_[t];
22             i0 = t+1;
23         }
24     }
25     return i-ns;
26 }
27

```

Based on this method are methods to determine the number of free/set indices to the left/right, including/excluding the given index. We omit the out-of-bounds clauses in the following:

```

1     ulong num_FLI(ulong i) const
2     // Return number of
3     // Free indices Left of (absolute) index i (Including i).
4     { return num_FLE(i) + tg_[i]; }
5
6     ulong num_FRE(ulong i) const
7     // Return number of
8     // Free indices Right of (absolute) index i (Excluding i).
9     { return num_free() - num_FLI(i); }
10
11    ulong num_FRI(ulong i) const
12    // Return number of
13    // Free indices Right of (absolute) index i (Including i).
14    { return num_free() - num_FLE(i); }
15
16    ulong num_SLE(ulong i) const
17    // Return number of
18    // Set indices Left of (absolute) index i (Excluding i).
19    { return i - num_FLE(i); }
20
21    ulong num_SLI(ulong i) const
22    // Return number of
23    // Set indices Left of (absolute) index i (Including i).
24    { return i - num_FLE(i) + !tg_[i]; }
25
26    ulong num_SRE(ulong i) const
27    // Return number of
28    // Set indices Right of (absolute) index i (Excluding i).
29    { return num_set() - num_SLI(i); }
30
31    ulong num_SRI(ulong i) const
32    // Return number of
33    // Set indices Right of (absolute) index i (Including i).
34    { return num_set() - i + num_FLE(i); }

```

## Optimizations

With large sizes (say, well beyond the size of the second-level memory cache) much of the time is spent waiting for the memory access. Using a structure corresponding to an  $m$ -ary tree (instead of the binary tree structure we used here) can drastically improve the performance. The data on each node should fill at least one cache line and be correspondingly aligned in memory.

Let  $B$  be the number of bits in a computer word (usually the C type `unsigned long`). At the terminals, a word can be used as a bit-array to indicate which of the  $B$  entries are occupied. A fast routine for determining the index of the  $i$ -th set bit of the word should be used [2, sect. 1.10].

To further improve memory locality, the tags (indicating whether a position is occupied, `tg_` in the class) should not be separated from the counts of the free positions to the left (`f1_` in the class). One way to do this is using one array where the least significant bit in each word represents the tag and all higher bits are used for the count.

# Bibliography

- [1] Jörg Arndt: **FXT, a library of algorithms**, (1996-2009). URL: <http://www.jjj.de/fxt/>. 9
- [2] Jörg Arndt: **Matters Computational**, to appear, (2009). URL: <http://www.jjj.de/fxt/#fxtbook>. 1, 9, 12, 81, 86
- [3] Miklós Bóna: **Combinatorics of Permutations**, Chapman & Hall/CRC, (2004). 1, 5, 37
- [4] Louis Comtet: **Advanced Combinatorics**, revised edition, D. Reidel Publishing, (1974). 5, 69
- [5] Alain Denise, Paul Zimmermann: **Uniform random generation of decomposable structures using floating-point arithmetic**, Theoretical Computer Science, vol.218, no.2, pp.219-232, (1999). 9, 76
- [6] Persi Diaconis, Ronald Graham, Susan P. Holmes: **Statistical problems involving permutations with restricted positions**, In: State of the Art in Probability and Statistics: Festschrift for Willem R. van Zwet, Papers from the symposium held at the University of Leiden, Leiden, March 23–26, 1999, pp.195-222, (2001). URL: <http://projecteuclid.org/euclid.lnms/1215090070>. 1
- [7] Richard Durstenfeld: **Algorithm 235: random permutation**, Communications of the ACM, vol.7, no.7, p.420, (July-1964). 1, 11
- [8] Jarmo Ernvall, Olli Nevalainen: **An Algorithm for Unbiased Random Sampling**, The Computer Journal, vol.25, no.1, pp.45-47, (1982). 25
- [9] R. A. Fisher, F. Yates: **Statistical Tables for Biological, Agricultural and Medical Research**, Oliver & Boyd, London, (1938). URL (6th edition): [http://digital.library.adelaide.edu.au/coll/special/fisher/stat\\_tab.pdf](http://digital.library.adelaide.edu.au/coll/special/fisher/stat_tab.pdf). 1
- [10] Philippe Flajolet, Paul Zimmermann, Bernard Van Cutsem: **A calculus for the random generation of labelled combinatorial structures**, Theoretical Computer Science, vol.132, no.1-2, pp.1-35, (September-1994). 76
- [11] Jens Gustedt: **Efficient sampling of random permutations**, Journal of Discrete Algorithms, vol.6, no.1, pp.125-139, (March-2008). URL: <http://hal.inria.fr/inria-00000900/en/> (preprint). 1
- [12] Carl Friedrich Hindenburg: **Sammlung combinatorisch-analytischer Abhandlungen**, Fleischer, Leipzig, (1800). URL: <http://books.google.com/books?id=YK0AAAAAMAAJ>. 45

- [13] Donald E. Knuth: **The Art of Computer Programming**, third edition, Volume 2: Seminumerical Algorithms, Addison-Wesley, (1997). 1, 11
- [14] Donald E. Knuth: **The Art of Computer Programming**, second edition, Volume 3: Sorting and Searching, Addison-Wesley, (1997). 43
- [15] Conrado Martínez, Alois Panholzer, Helmut Prodinger: **Generating random derangements**, Proceedings of the 10th ACM-SIAM Workshop on Algorithm Engineering and Experiments (ALENEX) and the 5th ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics (ANALCO), pp.234-240, (2008). URL: <http://www.siam.org/proceedings/analco/2008/analco08.php>. 1, 71
- [16] Wendy Myrvold, Frank Ruskey: **Ranking and unranking permutations in linear time**, Information Processing Letters, vol.79, pp.281-284, (2001). URL: <http://www.cs.uvic.ca/~ruskey/Publications/>. 33
- [17] Eugen Netto: **Lehrbuch der Combinatorik**, Teubner Verlag, (1901). URL: <http://www.archive.org/details/lehrbuchdercomb00nettgoog>. 45
- [18] John von Neumann: **Various techniques used in connection with random digits**, in *Monte Carlo Method*, Appl. Math. Series 12, US National Bureau of Standards, (1951), pp.36-38 (summary written by G. E. Forsythe); reprinted in *John von Neumann Collected Works* (ed. A. H. Taub), vol.5, Pergamon Press, New York, (1963), pp.768-770. 50
- [19] Albert Nijenhuis, Herbert S. Wilf: **Combinatorial Algorithms for Computers and Calculators**, Academic Press, second edition, (1978). URL: <http://www.math.upenn.edu/~wilf/website/CombAlgDownld.html>. 1, 48
- [20] B. J. Oommen, D. T. H. Ng : **On Generating Random Permutations with Arbitrary Distributions**, The Computer Journal, vol.33, no.4, pp.368-374, (1990). 48
- [21] The PARI Group (PARI): **PARI/GP**, version 2.3.4, (2008). URL: <http://pari.math.u-bordeaux.fr/>. 9, 39, 66
- [22] M. C. Pike: **Remark on algorithm 235 [G6]: random permutation**, Communications of the ACM, vol.8, no.7, p.445, (1965). 13
- [23] Edward M. Reingold, Jurg Nievergelt, Narsingh Deo: **Combinatorial algorithms – theory and practice**, Prentice-Hall, Englewood Cliffs, (1977). 1, 76
- [24] John Riordan: **An Introduction to Combinatorial Analysis**, Wiley, (1958). 5, 56, 69
- [25] Sandra Sattolo: **An algorithm to generate a random cyclic permutation**, Information Processing Letters, vol.22, no.6, pp.315-317, (1986). 1, 14
- [26] N. J. A. Sloane: **The On-Line Encyclopedia of Integer Sequences**, (2009). URL: <http://www.research.att.com/~njas/sequences/>. 49, 50, 55, 63, 69, 74
- [27] Richard P. Stanley: **Enumerative Combinatorics**, Cambridge University Press, vol.1, (1997), vol.2, (1999). List of errata at <http://math.mit.edu/~rstan/ec/>. 63
- [28] Herbert S. Wilf: **generatingfunctionology**, second edition, Academic Press, (1992). URL: <http://www.math.upenn.edu/~wilf/DownldGF.html>. 75