ELSEVIER

Information Processing Letters 79 (2001) 281–284

Information Processing Letters

www.elsevier.com/locate/ipl

Ranking and unranking permutations in linear time

Wendy Myrvold [1], Frank Ruskey [*,2]

Department of Computer Science, University of Victoria, Victoria, B.C. V8W 3P6, Canada

Received 17 April 2000; received in revised form 31 October 2000
Communicated by F.Y.L. Chin

Abstract

A *ranking function* for the permutations on n symbols assigns a unique integer in the range $[0, n! - 1]$ to each of the $n!$ permutations. The corresponding *unranking function* is the inverse: given an integer between 0 and $n! - 1$, the value of the function is the permutation having this rank. We present simple ranking and unranking algorithms for permutations that can be computed using $O(n)$ arithmetic operations. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Permutation; Ranking; Unranking; Algorithms; Combinatorial problems

A permutation of order n is an arrangement of n symbols. For convenience when applying modular arithmetic, this paper considers permutations of $\{0, 1, 2, \ldots, n-1\}$. The set of all permutations over $\{0, 1, 2, \ldots, n-1\}$ is denoted by S_n.

There are many applications that call for an array indexed by the permutations in S_n [2]. One example is the development of programs that search for Hamilton cycles in particular types of Cayley graphs [10,11]. To do such indexing, what is desired is a bijective *ranking* function r that takes as input a permutation π and produces $r(\pi)$, a number in the range $0, 1, \ldots, n! - 1$. The inverse of r is also often useful, and is called the *unranking* function.

The traditional approach to this problem is to first define an ordering of permutations and then find ranking and unranking functions relative to that ordering. For example, in lexicographic order, the rank of a permutation is simply the number of permutations that precede it in lexicographic order. Naive implementations of ranking and unranking functions for lexicographic order require $O(n^2)$ time [7,9].

Given a permutation $\pi = \pi_0\pi_1 \ldots \pi_{n-1}$, its *inversion vector* $v = v_0 v_1 \ldots v_{n-1}$ has v_i equal to the number of entries π_j such that $\pi_j > \pi_i$ and $j < i$. Hall (see [12, p. 203]) first observed that the inversion vector uniquely determines a permutation.

More sophisticated algorithms for ranking and unranking permutations in lexicographic order calculate the inversion vector as an intermediate step. The first step in ranking is to determine the inversion vector of a permutation. Unfortunately, naive implementations require $O(n^2)$ time and even the $O(n \log n)$ implementations using modular arithmetic [5, Ex. 6, p. 18] or mergesort [5, Ex. 21, p. 168] are too slow. The last step in unranking is to determine the permutation from its inversion vector. Again, the naive approach takes $O(n^2)$ time. A balanced binary search tree can be used to improve this to $O(n \log n)$. Using the complicated

* Corresponding author.
E-mail addresses: wendym@csr.uvic.ca (W. Myrvold), fruskey@csr.uvic.ca (F. Ruskey).

[1] Research supported in part by NSERC grant OGP0041927.
[2] Research supported in part by NSERC grant OGP0003379.

PII: S0020-0190(01)00141-7

data structure of Dietz [3] the running time can be reduced to $O((n \log n)/(n \log\log n))$, but we know of no implementations of this algorithm. Conversion between the inversion vector and the rank is straightforward and can be done in $O(n)$ arithmetic operations. So the bottleneck is the translation between a permutation and its inversion vector.

The whole problem of ranking permutations in lexicographic order seems inextricably intertwined with the problem of computing the number of inversions in a permutation, and it seems that a major breakthrough will be required to do that computation in linear time, if indeed it it possible at all. Our new algorithm achieves linear time by not insisting that the permutations are lexicographically ordered.

Other ranking algorithms for permutations have been published, for example, in the Steinhaus–Johnson–Trotter order, but these offer no running-time advantages over the lexicographic algorithm. See Reingold, Nievergelt, and Deo [9] or Kreher and Stinson [6] for a description of these algorithms.

Our approach to this problem differs from previous approaches in two important aspects. First, instead of selecting an ordering of the permutations and then finding the corresponding ranking and unranking algorithms, the ordering is defined by the unranking algorithm and it is not particularly easy to describe. The second difference is that the unranking algorithm is developed first and then the ranking algorithm is derived from it. Traditionally, ranking algorithms have been developed first, then the unranking algorithms. Furthermore, in all other cases that we know of, the unranking algorithm is more complicated than the ranking algorithm — but that is not the case here!

1. Ranking and unranking

In this section we present two slightly different approaches for ranking and unranking permutations. The first (*rank1* and *unrank1*) has simpler code. The second approach (*rank2* and *unrank2*) is included as it is easier to understand the ordering of the permutations according to their ranks.

Our inspiration is the standard algorithm [8,4,1] for generating a random permutation. The array $\pi[0..n-1]$ is initialized to the identity permutation (or some other permutation) and then the following loop is executed:

for $k := n-1, n-2, \ldots, 1$ **do**
  $swap(\pi[k], \pi[rand(k)])$;

where the call $rand(k)$ should produce a random integer in the range $0..k$.

This algorithm produces a permutation selected uniformly at random from amongst all permutations in S_n. Let $r_{n-1}, \ldots, r_1, r_0$ be the sequence of random elements produced by the algorithm, where $0 \leqslant r_i \leqslant i$. Since there are exactly $n(n-1)(n-2)\cdots 2 \cdot 1 = n!$ such sequences, each different sequence must produce a different permutation. Thus we should be able to unrank if we can take an integer r in the range $0..n!-1$ and turn it into a unique sequence of values $r_{n-1}, \ldots, r_1, r_0$, where $0 \leqslant r_i \leqslant i$. The details are given below.

To unrank a permutation we first initialize π to be the identity permutation: $\pi[i] := i$ for $i = 0, 1, \ldots, n-1$.

procedure *unrank1*(n, r, π)
  if $n > 0$ **then**
    $swap(\pi[n-1], \pi[r \bmod n])$;
    unrank1$(n-1, \lfloor r/n \rfloor, \pi)$;
  fi;
end {of *unrank1*};

It should be fairly obvious why this function works. We can use the argument alluded to above or argue directly as follows. We need only show that every permutation in S_n is a possible outcome for some $r \in \{0, 1, \ldots, n!-1\}$. Clearly, every possible value of $\pi[0..n-1]$ can appear in position $n-1$ after the interchange. After $\pi[n-1]$ is set it is never again modified. Further,

$$\{\lfloor r/n \rfloor : r \in \{0, 1, \ldots, n!-1\}\} = \{0, 1, \ldots, (n-1)!-1\},$$

so, inductively, we may assume that every possible permutation of $\pi[0..n-2]$ can occur.

To rank, first compute π^{-1}. This can be done in $O(n)$ operations by iterating

$$\pi^{-1}[\pi[i]] := i \quad \text{for } i = 0, 1, \ldots, n-1.$$

In the algorithm below, both π and π^{-1} are modified.

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

Fig. 1. Ranks of permutations for *rank1*, $n = 4$.

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

Fig. 2. Ranks of permutations for *rank2*, $n = 4$.

function *rank1*(n, π, π^{-1}) : integer;
  if $n = 1$ **then** RETURN(0) **fi**;
  $s := \pi[n-1]$;
  swap$(\pi[n-1], \pi[\pi^{-1}[n-1]])$;
  swap$(\pi^{-1}[s], \pi^{-1}[n-1])$;
  RETURN$(s + n \cdot$ *rank1*$(n-1, \pi, \pi^{-1}))$;
end {of *rank1*};

These algorithms obviously use $O(n)$ operations. The corresponding ranks for the permutations for $n = 4$ are as illustrated in Fig. 1.

We now present another unranking algorithm, different than the first, but based on the same underlying principle. In this algorithm the permutations occur in a different order; and order which is easier to describe than the order produced by the first algorithm. Before calling *unrank2*, initialize π to be the identity permutation; $\pi[i] := i$ for $i = 0, 1, \ldots, n-1$. Compute π^{-1} before calling *rank2*.

procedure *unrank2*(n, r, π)
  if $n > 0$ **then**
    $s := \lfloor r/(n-1)! \rfloor$;
    swap$(\pi[n-1], \pi[s])$;
    unrank2$(n-1, r \bmod (n-1)!, \pi)$;
  fi;
end {of *unrank2*};

function *rank2*(n, π, π^{-1}) : integer;
  if $n = 1$ **then** RETURN(0) **fi**;
  $s := \pi[n-1]$;
  swap$(\pi[n-1], \pi[\pi^{-1}[n-1]])$;
  swap$(\pi^{-1}[s], \pi^{-1}[n-1])$;
  RETURN$(s \cdot (n-1)! +$ *rank2*$(n-1, \pi, \pi^{-1}))$;
end {of *rank2*};

The order of generation for $n = 4$ is given in Fig. 2. In general this order may be described as follows. Let $\mathcal{L}_n$ denote the list of permutations of $0..n-1$. Let $\mathcal{L}_n^m$ denote the list $\mathcal{L}_n$, but with every occurrence of m replaced with n. Then

$$\mathcal{L}_{n+1} = \mathcal{L}_n^0 \cdot 0 \circ \mathcal{L}_n^1 \cdot 1 \circ \cdots \circ \mathcal{L}_n^{n-1} \cdot (n-1) \circ \mathcal{L}_n^n \cdot n.$$

For example, in the last column of Fig. 2 is the list $\mathcal{L}_3^3 \cdot 3 = \mathcal{L}_3 \cdot 3$. By $\circ$ we denote concatenation of lists, and the notation $\mathcal{L} \cdot x$ means to append the character x to the end of every permutation in the list $\mathcal{L}$.

2. Possible extensions

If the algorithm for generating random permutations is terminated at the kth step then positions $n-k..n-1$ hold a random k-permutation of $0, 1, \ldots, n-1$. Hence, our ranking and ranking algorithms are easily modified to do k-permutations of an n-set.

References

[1] G. de Balbine, Note on random permutations, Math. of Comput. 21 (1967) 710–712.
[2] F. Critani, M. Dall'Aglio, G. Di Biase, Ranking and unranking permutations with applications, in: Innovation in Mathematics (Rovaniemi, 1997), Comput. Mech., Southampton, 1997, pp. 99–106.
[3] P.F. Dietz, Optimal algorithms for list indexing and subset rank, in: Workshop on Algorithms and Data Structures (WADS), Lecture Notes in Comput. Sci., Vol. 382, Springer, Berlin, 1989, pp. 39–46.
[4] R. Durstenfeld, Algorithm 235: Random permutation, Comm. ACM (1964) 420.
[5] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd edn., Addison-Wesley, Reading, MA, 2000 (first published in 1973).
[6] D.L. Kreher, D.R. Stinson, Combinatorial Algorithms: Generation, Enumeration, and Search, CRC Press, Rockville, MD, 1999.

[7] J. Liebehenschel, Ranking and unranking of lexicographically ordered words: An average-case analysis, J. Automat. Languages Combinatorics 2 (1997) 227–268.
[8] L.E. Moses, R.V. Oakland, Tables of Random Permutations, Stanford University Press, Stanford, CA, 1963.
[9] E.M. Reingold, J. Nievergelt, N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, NJ, 1977.
[10] F. Ruskey, M. Jiang, A. Weston, The Hamiltonicity of directed σ-τ Cayley graphs (or: A tale of backtracking), Discrete Appl. Math. 57 (1) (1995) 75–83.
[11] F. Ruskey, C. Savage, Hamilton cycles that extend transposition matchings in Cayley graphs of S_n, SIAM J. Discrete Math. 6 (1) (1993) 152–166.
[12] C.B. Tompkins, Machine attacks on problems whose variables are permutations, in: Numerical Analysis, Proceedings of Symposia in Applied Mathematics, Vol. 6, American Mathematical Society, Providence, RI, 1956.