# Linear-Time Ranking of Permutations

Martin Mareš[⋆] and Milan Straka

Department of Applied Mathematics
and
Institute of Theoretical Computer sciences (ITI)
Charles University
Malostranské nám. 25, 118 00 Praha, Czech Republic
{mares,fox}@kam.mff.cuni.cz

**Abstract.** A *lexicographic ranking function* for the set of all permutations of $n$ ordered symbols translates permutations to their ranks in the lexicographic order of all permutations. This is frequently used for indexing data structures by permutations. We present algorithms for computing both the ranking function and its inverse using $O(n)$ arithmetic operations.

## 1 Introduction

A permutation of order $n$ is a bijection of an $n$-element set $X$ onto itself. For convenience, we will always assume that $X = [n] = \{1, \ldots, n\}$, so the permutations become ordered $n$-tuples containing each number $1, \ldots, n$ exactly once.

Many applications ask for arrays indexed by permutations. Among other uses documented in [1], this is handy when searching for Hamilton cycles in Cayley graphs [2, 3] or when exploring state spaces of combinatorial puzzles like the Loyd's Fifteen [4]. To accomplish that, a *ranking function* is usually employed, which translates a permutation $\pi$ to a unique number $R(\pi) \in \{0, \ldots, n! - 1\}$. The inverse of the ranking function $R^{-1}(i)$ is also frequently used and it is called the *unranking function.*

Each ranking function corresponds to a linear order on the set of all permutations (it returns the number of permutations which are strictly less than the given one). The traditional approach to the ranking problem is to fix lexicographic order and construct the appropriate ranking and unranking functions. In fact, an arbitrary order suffices in many cases, but the lexicographic ranking has the additional advantage of a nice structure allowing additional operations on permutations to be performed directly on their ranks.

Naïve implementations of lexicographic ranking require time $\Theta(n^2)$ in the worst case [5, 6]. This can be easily improved to $O(n \log n)$ by using either a binary search tree to calculate inversions, or by a divide-and-conquer technique or by clever use of modular arithmetic (all three algorithms are described in [7]). Myrvold and Ruskey [8] mention further improvements to $O(n \log n / \log \log n)$ by using the data structures of Dietz [9].

⋆ Supported by grant 1M0021620808 of the Czech Ministry of Education.

Linear time complexity was reached also by Myrvold and Ruskey [8] by employing a different order, which is defined locally by the history of the data structure — in fact, they introduce a linear-time unranking algorithm first and then they derive an inverse algorithm without describing the order explicitly. However, they leave the problem of lexicographic ranking open.

In this paper, we present a linear-time algorithm for the lexicographic order. It is based on an observation that once we assume that our computation model is capable of performing basic arithmetics on numbers of the order of magnitude of the resulting rank, we can use such integers to represent fairly rich data structures working in constant time. This approach has been pioneered by Fredman and Willard [10, 11], who presented Fusion trees and Atomic heaps working in various RAM models. Our structures are built on similar principles, but in a simpler setting, therefore we even can avoid random access arrays and our algorithms work in almost all models of computation, relying only on the usual set of arithmetic and logical operations on the appropriately large integers.

We also extend our algorithm to ranking and unranking of $k$-permutations, i.e., ordered $k$-tuples of distinct elements drawn from $[n]$.

## 2 Ranking Permutations

Permutations have a simple recursive structure: if we fix the first element $\pi(1)$ of a permutation $\pi$ on $[n] = \{1, \dots, n\}$, the elements $\pi(2), \dots, \pi(n)$ form a permutation on $\{1, \dots, \pi(1)-1, \pi(1)+1, \dots, n\}$. The lexicographic order of $\pi$ and $\pi'$ is then determined by $\pi(1)$ and $\pi'(1)$ and only if these elements are equal, it is decided by lexicographic comparison of permutations $(\pi(2), \dots, \pi(n))$ and $(\pi'(2), \dots, \pi'(n))$. Therefore the rank of $\pi$ is $(\pi(1)-1) \cdot (n-1)!$ plus the rank of $(\pi(2), \dots, \pi(n))$.

This gives a reduction of (un)ranking of permutations on $[n]$ to (un)ranking of permutations on a $(n-1)$-element set, which suggests a straightforward algorithm, but unfortunately this set is different from $[n-1]$ and it even depends on the value of $\pi(1)$. We could renumber the elements to get $[n-1]$, but it would require linear time per iteration. Instead, we generalize the problem to permutations on subsets of $[n]$. Therefore for a permutation $\pi$ on $A \subseteq [n]$ we have:

$$R((\pi(1), \dots, \pi(m)), A) = r(\pi(1), A) \cdot (m-1)! + R((\pi(2), \dots, \pi(m)), A \setminus \{\pi(1)\}),$$

where $r(x, A)$ is a ranking function on elements of $A$.

This recurrence leads to the following algorithms for ranking and unranking. Here $\pi[i, \dots, n]$ denotes the array containing the permutation and $A$ is a data structure representing the subset of $[n]$ on which is $\pi$ defined. This structure supports ranking, unranking and deletion of individual elements.

**function** rank$(\pi, i, n, A)$ { return the rank of a permutation on $A$ }
  **if** $i \geq n$ **then return** $0$

$a \leftarrow r(\pi[i], A)$
$A \leftarrow A \setminus \{\pi[i]\}$
$b \leftarrow \mathrm{rank}(\pi, i+1, n, A)$
**return** $a \cdot (n-i)! + b$
**end**

**procedure** $\mathrm{unrank}(j, \pi, i, n, A)$ { construct the $j$-th permutation on $A$ }
**if** $i > n$ **then return**
$\pi[i] \leftarrow r^{-1}(\lfloor j/(n-i)! \rfloor, A)$
$A \leftarrow A \setminus \{\pi[i]\}$
$\mathrm{unrank}(j \bmod (n-i)!, \pi, i+1, n, A)$
**end**

If we precalculate the factorials and assume that each operation on the data structure $A$ takes time at most $t(n)$, both algorithms run in time $O(n \cdot t(n))$ and their correctness follows from the discussion above.

A trivial implementation of the data structure by an array yields $t(n) = O(n)$. Using a binary search tree instead gives $t(n) = O(\log n)$. The data structure of Dietz [9] improves it to $t(n) = O(\log n / \log \log n)$. In fact, all these variants are equivalent to the classical algorithms based on inversion vectors, because at the time of processing $\pi(i)$, the value of $r(\pi(i), A)$ is exactly the number of elements forming inversions with $\pi(i)$.

If we relax the requirements on the data structure to allow ordering of elements dependent on the history of the structure (i.e., on the sequence of deletes performed so far), we can implement all three operations in time $O(1)$. We store the values in an array and we also keep an inverse permutation. Ranking is done by direct indexing, unranking by indexing of the inverse permutation and deleting by swapping the element with the last element. We can observe that although it no longer gives the lexicographic order, the unranking function is still the inverse of the ranking function (the sequence of deletes from $A$ is the same in both functions), so this leads to $O(n)$ time ranking and unranking in a non-lexicographic order. This order is the same as the one used by Myrvold and Ruskey in [8].

However, for our purposes we need to keep the same order on $A$ over the whole course of the algorithm.

## 3 Word-encoded Sets

We will describe a data structure for the subsets of $[n]$ supporting insertion, deletion, ranking and unranking in constant time per operation in the worst case.

First, let us observe that whatever our computation model is, it must allow operations with integers up to $n!-1$, because the ranks of permutations can reach such large numbers. In accordance with common practice, we will assume that the usual set of arithmetic and logical operations is available and that they work in constant time on integers of that size, i.e., on $\lceil \log_2(n!) \rceil = \Omega(n \log n)$ bits.

Furthermore, multiple-precision arithmetics on numbers which fit in a constant number of machine words can be emulated with a constant number of operations on these words, so we can also assume existence of constant-time operations on arbitrary $O(n \log n)$-bit numbers. This gives us an opportunity for using the integers to encode data structures.

Let us denote $b = \lceil \log_2 n \rceil$ the number of bits needed to represent a single element of $[n]$. We will store the whole subset $A \subseteq [n]$, $k = |A|$, as a bit vector $\mathbf{a}$ consisting of $k$ fields of size $b$, each field containing a single element of $A$. The fields will be maintained in increasing order of values and an additional zero bit will be kept between adjacent fields and also after the highest field. The vector is $k(b+1) = O(n \log n)$ bits long, so it fits in $O(1)$ integers. We will describe how to perform the data structure operations using arithmetic and logical operations on these integers.

*Unranking* is just extraction of the $r$-th field of the vector. It can be accomplished by shifting the representation of $\mathbf{a}$ by $r \cdot (b+1)$ bits to the right and ANDing it with $2^b - 1$, which masks out the high-order bits.

*Insertion* can be reduced to ranking: once we know the position in the vector the new element should land at, i.e., its rank, we can employ bit shifts, ANDs and ORs to shift apart the existing fields and put the new element to the right place.

*Deletion* can be done in a similar way. The rank gives us the position of the field we want to delete and we again use bit operations to move the other fields in parallel.

*Ranking* of an element $x$ is the only non-trivial operation. We prepare a vector $\mathbf{c}$, which has all $k$ fields set to $x$ (this can be done in a single multiplication by an appropriate constant) and the separator bits between them set to ones. Observe that if we subtract $\mathbf{a}$ from $\mathbf{c}$, the separator bits change to zeroes exactly at the places where $a[i] > c[i] = x$ (and they absorb the carries, so the fields do not interfere with each other). Therefore the desired rank is the number of remaining ones in the separator bits minus 1.

Let us observe that if $\mathbf{z}$ is an encoding of a vector with separator zeroes, then $\mathbf{z} \bmod (2^{b+1} - 1)$ is the sum of all fields of the vector modulo $2^{b+1} - 1$. This works because $\mathbf{z} = \sum_i z[i] \cdot 2^{(b+1)i}$ and $2^{(b+1)i} \bmod (2^{b+1} - 1) = 1$ for every $i$. Hence the separator bits in $\mathbf{c}$ can be summed by masking out all non-separator bits, shifting $\mathbf{c}$ to the right to transform the separator bits to field values and calculating $\mathbf{c} \bmod (2^{b+1} - 1)$.

All four operations work in constant time. The auxiliary constants for bit patterns we needed can be precalculated in linear time at the start of the algorithm (we could even calculate them in constant time, as $2^0 + 2^k + 2^{2k} + \ldots + 2^{\ell k} = (2^{(\ell+1)k} - 1)/(2^k - 1)$, but it is an unnecessary complication). With this data structure, the ranking and unranking algorithms achieve linear time complexity.

## 4 Ranking *k*-permutations

Our (un)ranking algorithms can be used for $k$-permutations as well if we just stop earlier and divide everything by $(n-k)!$. Unfortunately, the ranks of $k$-permutations can be much smaller, so we can no longer rely on the data structure fitting in a constant number of integers.

We do a minor side step by remembering the complement of $A$ instead, that is the set of elements we have already seen. We will call it $G$ (and the corresponding vector $\mathbf{g}$), because they form gaps in $A$. Let us prove that $\Omega(k \log n)$ bits are needed to store the result, which is enough space to represent the whole $\mathbf{g}$.

**Lemma 1.** *The number of $k$-permutations on $[n]$ is $2^{\Omega(k \log n)}$.*

*Proof.* There are $n^{\underline{k}} = n(n-1)\ldots(n-k+1)$ such $k$-permutations. If $k \leq n/2$, then every term in the product is at least $n/2$, so $\log_2 n^{\underline{k}} \geq k(\log_2 n - 1)$. If $k \geq n/2$, then $n^{\underline{k}} \geq n^{\underline{n/2}}$ and $\log_2 n^{\underline{k}} \geq (n/2)(\log_2 n - 1) \geq (k/4)(\log_2 n - 1)$. $\square$

Deletes in $A$ now become inserts in $G$. The rank of $x$ in $A$ is just $x$ minus the rank of the largest element of $G$ which is smaller or equal to $x$. This is easy to get, since when we ask our data structure for a rank of an element $x$ outside the set, we get exactly the number elements of the set smaller than $x$.

The only operation we cannot translate directly is unranking in $A$. To achieve it, we will maintain another vector $\mathbf{b}$ such that $b[i] = g[i] - i$ (the number of elements in $A$ less than $g[i]$). Now, if we want to find the $r$-th element of $A$, we find the largest $i$ such that $b[i] \leq r$ (the rank of $r$ in $\mathbf{b}$ in the same sense as above) and we return $g[i] + r - b[i]$. Also, whenever a new element is inserted into $\mathbf{g}$, we can shift the fields of $\mathbf{b}$ accordingly and decrease all higher fields by one in parallel by a single subtraction.

We have replaced all operations on $A$ by the corresponding operations on the modified data structure, each of which works again in constant time. This gives us a linear-time algorithm for the $k$-permutations, too.

## 5 Concluding Remarks

We have shown linear-time algorithms for ranking and unranking of permutations and $k$-permutations on integers, closing a frequently encountered open question.

Our algorithms work in linear time in a fairly broad set of computation models. Even if we take the complexity of operations on numbers in account, we have proven that the number of arithmetic operations for determining the inversion vector is bounded by the number of those required to calculate the rank from the inversion vector, contrary to previous intuition.

The technique we have demonstrated should give efficient algorithms for ranking of various other classes of combinatorial objects, if the number of such objects is high enough to ensure word size suitable for our data structures. A good example would be the set of labelled trees.

## References


1. Critani, F., Dall'Aglio, M., Di Biase, G.: Ranking and unranking permutations with applications. In: Innovation in Mathematics: Proceedings of Second International Mathematica Symposium. (1997) 99–106
2. Ruskey, F., Jiang, M., Weston, A.: The Hamiltonicity of directed-Cayley graphs (or: A tale of backtracking). Discrete Appl. Math **57** (1995) 75–83
3. Ruskey, F., Savage, C.: Hamilton Cycles that Extend Transposition Matchings in Cayley Graphs of Sn. SIAM Journal on Discrete Mathematics **6**(1) (1993) 152–166
4. Slocum, J., D., S.: The 15 Puzzle Book. The Slocum Puzzle Foundation, Beverly Hills, CA, USA (2006)
5. Liebehenschel, J.: Ranking and Unranking of Lexicographically Ordered Words: An Average-Case Analysis. Journal of Automata, Languages and Combinatorics **2**(4) (1997) 227–268
6. Reingold, E.: Combinatorial Algorithms: Theory and Practice. Prentice Hall College Div (1977)
7. Knuth, D.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA (1998)
8. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79**(6) (2001) 281–284
9. Dietz, P.: Optimal Algorithms for List Indexing and Subset Rank. Springer-Verlag London, UK (1989)
10. Fredman, M., Willard, D.: Surpassing the information theoretic bound with fusion trees. Journal of Computer and System Sciences **47**(3) (1993) 424–436
11. Fredman, M., Willard, D.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. Journal of Computer and System Sciences **48**(3) (1994) 533–551