Efficient Algorithms to Rank and Unrank Permutations in Lexicographic Order

Blai Bonet

Departamento de Computación
Universidad Simón Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

Abstract

We present uniform and non-uniform algorithms to rank and unrank permutations in lexicographic order. The uniform algorithms run in $O(n \log n)$ time and outperform Knuth's ranking algorithm in all the experiments, and also the linear-time non-lexicographic algorithm of Myrvold-Ruskey for permutations up to size 128. The non-uniform algorithms generalize Korf-Schultze's linear time algorithm yet require much less space.

Introduction

In an increasing number of different applications, most of them related to heuristic search and combinatorial optimization (Korf & Schultze 2005; Ruskey, Jiang, & Weston 1995; Critani, Dall'Aglio, & Biase 1997), there is the need to rank a given permutation over n elements into an integer between 0 and $n! - 1$, and also to unrank such integer into a permutation. In others cases, one is interested in a particular subset of elements, and given a permutation over n elements, rank/unrank the positions of these elements (Culberson & Schaeffer 1998; Korf & Felner 2002; Felner, Korf, & Hanan 2004). A ranking function is called lexicographic if it maps a permutation and its lexicographically next permutation into consecutive integers.

There are well-known algorithms for these tasks that perform $O(n)$ arithmetic operations but not in lexicographic order (Myrvold & Ruskey 2001). For lexicographic ranking/unranking, there are $O(n \log n)$ algorithms that rely on modular arithmetic based on inversion tables (Knuth 1973, Ex. 6, p. 19), and also unrank algorithms based on binary search. Myrvold and Ruskey (Myrvold & Ruskey 2001) mention that using a data structure of Dietz (1989), the number of operations can be reduce to $O(n \log n / \log \log n)$ but the algorithm is rather complicated and difficult to implement. All these algorithms are uniform in the sense that they work for permutations of any size without the need to pre-process or store additional information (advice).

Non-uniform linear-time algorithms for ranking and unranking in lexicographic order were presented in (Korf & Schultze 2005), but these need to pre-calculate and store an advice of exponential size. Applications that need to perform a large number of rank/unrank operations of small size, e.g. $n = 16$, can amortize the time to calculate the advice.

The main contribution of this paper is the development of novel uniform algorithms for ranking and unranking of permutations in lexicographic order that perform $O(n \log n)$ arithmetic operations and utilize $O(n)$ space. The algorithms are very simple and easy to implement. We also perform an empirical comparison against the linear-time algorithm of Myrvold and Ruskey, and the algorithm of Knuth in order to account for the hidden constant factors. As it will be seen, the new algorithms dominate Knuth's algorithm, are faster than Myrvold and Ruskey's for small n, and fall a bit short of the latter for n up to size 1,024.

We also present a generalization of Korf-Schultze's non-uniform linear-time ranking algorithm that permits to reduce the size of the advice.

The paper is organized as follows. The next two sections are devoted to the uniform and non-uniform algorithms. Then, we present some empirical results, and conclude.

Uniform Algorithms

Permutations of size n are assumed to be over integers $\{0, \ldots, n - 1\}$, and denoted by $\pi = \pi_0 \ldots \pi_{n-1}$. For example, $\pi = 25714603$ denotes the permutation $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 5 & 7 & 1 & 4 & 6 & 0 & 3 \end{pmatrix}$ in which 0 is mapped into 2, 1 is mapped into 5, and so on. The lexicographic ranking of π is defined as

$$r(\pi) = d_0 \cdot (n-1)! + d_1 \cdot (n-2)! + \cdots + d_{n-2} \cdot 1! + d_{n-1} \cdot 0! \tag{1}$$

where d_i is the relative position of element i with respect to the elements $j < i$. For $\pi = 25714603$, $d_0 = 2$ since 0 is goes into position 2, $d_1 = 4$ since 1 goes into position 4 once 0 is fixed at position 2, and so on. In general, d_i equals π_i minus the number of elements $j > i$ that are to the left of element i. Each d_i ranges over $\{0, \ldots, n - 1 - i\}$ independently of the others. The vector $(d_0, \ldots, d_{n-1})$ is known as the factorial representation of $r(\pi)$ as they are the digits in a factorial-base system and also as the *inversion table* for π (Knuth 1973).

Another equivalent expression for $r(\pi)$ is

$$r(\pi) = (\cdots (d_0 \cdot (n-1) + d_1) \cdot (n-2) + \cdots + d_{n-2}) \cdot 1 + d_{n-1}. \tag{2}$$

Both expressions, (1) and (2), are closely related to the way tuples over Cartesian products are ranked into unique inte-
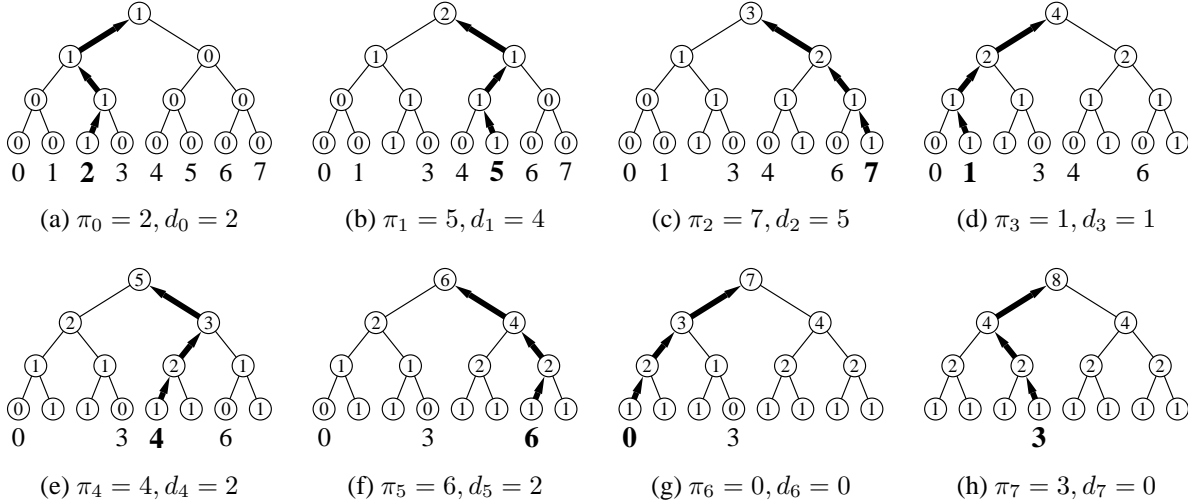
Figure 1: the ranking algorithm applied to permutation $\pi = 25714603$. Panels (a)–(h) show the paths transversed, the final configuration of the tree after transversing each path, and the calculated value for each d_i.

gers. The pair $(x, y) \in X \times Y$ is ranked into $x|Y| + y$ as for each x there are $|Y|$ possibilities for y. For the triplet $(x, y, z) \in X \times Y \times Z$, we can either think of it as the pair $(x, (y, z))$ and rank it into $x|Y||Z| + (y|Z| + z)$, or think of it as the pair $((x, y), z)$ and rank it into $(x|Y| + y)|Z| + z$. Both expression are equivalent. As each factorial digit d_i ranges over $n - i$ possible values, the vector of factorial digits is ranked into (1) (or equivalently (2)). Once the vector of factorial digits is known, the rank of the permutation is obtained in linear time. Therefore, the difficult task is to compute the digits efficiently.

Ranking

The naive approach to compute the factorial digits is to scan the permutation from left to right counting the number of elements to the left of i that are less than i, and then setting d_i to π_i minus such count. This method requires linear time per digit which results in an $O(n^2)$ algorithm.

However, if at the time of computing d_i, $\pi_i > \lceil n/2 \rceil$ and the number of elements less than i in positions 0 to $\lceil n/2 \rceil$ is stored in a counter, then only positions between $\lceil n/2 \rceil + 1$ and $\pi_i - 1$ must be scanned. If we apply this principle recursively, dividing by half successively the interval $[0, n - 1]$, the time to compute d_i is reduced to $O(\log n)$. This is the underlying principle in the algorithm that is explained next.

Let us assume for the moment that $n = 2^k$ is a power of 2. In order to store the required counts, we make use of a complete binary tree of height k whose nodes store the counts, and where the n leaves are associated with the elements of the permutation. Initially, all stored values equal zero.

To calculate d_i, the algorithm transverses the tree in a *bottom-up* manner. Starting at the leaf associated with π_i and by using a counter initialized to the value π_i, the algorithm moves up along the path that leads to the root performing two operations at each node: 1) if the current node is the right child of its parent then subtract from the counter the

Input: permutation π and array T of size $2^{1+\lceil \log n \rceil} - 1$
Output: rank of π
begin

    $k := \lceil \log n \rceil$
    $rank := 0$
    for $i = 1$ **to** $2^{1+k} - 1$ **do** $T[i] := 0$
    for $i = 1$ **to** n **do**
        $ctr := \pi[i]$
        $node := 2^k + \pi[i]$
        for $j = 1$ **to** k **do**
            if $node$ *is odd* **then** $ctr := ctr - T[(node \gg 1) \ll 1]$
            $T[node] := T[node] + 1$
            $node := node \gg 1$
        $T[node] := T[node] + 1$
        $rank := rank \cdot (n + 1 - i) + ctr$

    return $rank$
end

Figure 2: $O(n \log n)$ uniform lexicographic ranking.

value stored at the left child of the parent, and 2) increase the value stored at the current node. At the end of the journey, the value of the counter is the value of d_i.

For example, for $\pi = 25714603$, the path transversed for d_0 starts at the leaf associated with $\pi_0 = 2$ (the third leaf from the left). Since all initial stored values equal zero then $d_0 = 2$. The resulting tree and the path transversed for d_0 are shown in Fig. 1(a). For d_1, the counter is initialized to $\pi_1 = 5$ and the path starts at the leaf associated to 5. This leaf is a right child but its left sibling has a stored value of 0 so the counter does not change. The next node is a left child and so the counter is not decremented. Finally, just before moving to the root, the node is a right child and the left sibling has a stored value of 1. This value is decremented from the counter that becomes 4 which is the final value of d_1. Panels (a)–(h) in Fig. 1 show all paths transversed by the
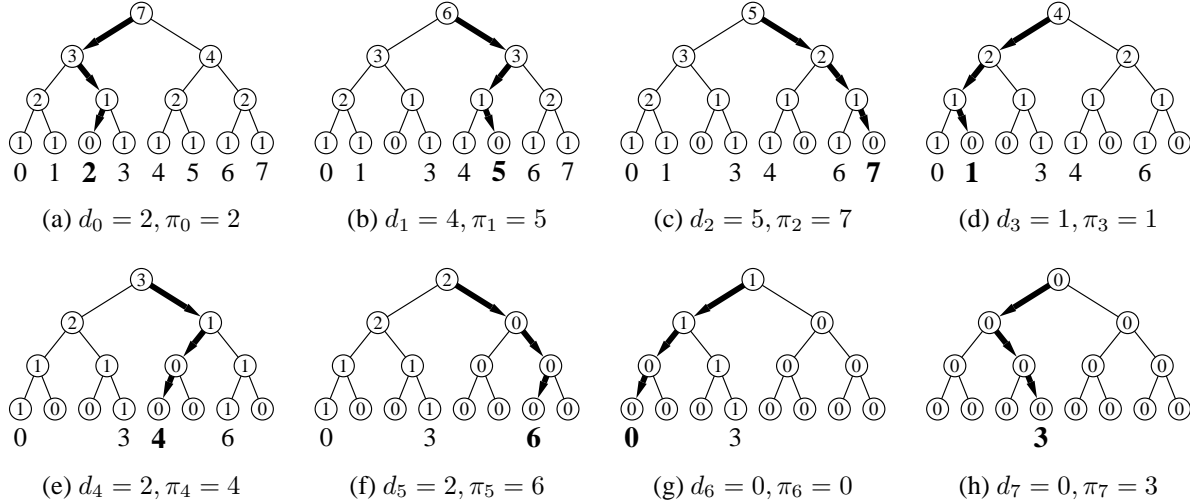
Figure 3: the unranking algorithm applied to the factorial digits 24512200. Panels (a)–(h) show the paths tranversed, the final configuration of the tree after transversing each path, and the calculated value for each π_i.

algorithm and the calculated values.

The tree used to compute the rank can be stored layer-by-layer in an array of size $2n - 1$ indexed from 1 to $2n - 1$, as done with heaps (Cormen, Leiserson, & Rivest 1990).[1] In this array, the left and right children of node i are the nodes $2i$ and $2i + 1$ respectively, the parent of node i is the node $i \div 2$, and all leaves appear ordered from left to right in positions n through $2n - 1$. A node i is a left or right child whether i is even or odd. The pseudo-code of the ranking algorithm is depicted in Fig. 2.[2]

Unranking

Unranking is the inverse process: given a rank r obtain the permutation π such that $r(\pi) = r$. In this case, the factorial digits are obtained in linear time from r using successive integer divisions and mod operations. The value d_i gives the relative position of element i in the permutation once the elements $j < i$ had been placed. In the example, the position of 0 is 2, the relative position of 1 is 4 and given that position 2 is occupied then 1 goes into position 5, and so on.

A naive algorithm scans the permutation being constructed at each stage in order to place element i given its relative position d_i which results in a $O(n^2)$ algorithm. As before, the time can be reduced to $O(n \log n)$ by using a complete binary tree of height $\lceil \log n \rceil$. This time, the stored values for nodes at depth i are initialized to 2^{k-i}.

Once the d_i's are calculated, the algorithm transverses the tree in a *top-down* manner for each d_i. Starting at the root, the algorithm chooses the next node as the left or right child whether d_i is less than the stored value at the left child. If the next node is the right child, then the value of the left child is

---

[1] For general values of n, the tree is of height $\lceil \log n \rceil$ which occupies $O(n)$ space.

[2] '$n \ll 1$' refers to a left shift of 1 position and similar for '$n \gg 1$' and '$n \div m$' to the integer division of n by m.

Input: factorial digits d, array π, and array T of size $2^{1+\lceil \log n \rceil} - 1$
Output: permutation π
begin
  $k := \lceil \log n \rceil$
  for $i = 0$ **to** k **do**
    for $j = 1$ **to** 2^i **do** $T[2^i + j - 1] := 1 \ll (k - i)$
  for $i = 1$ **to** n **do**
    $digit := d[i]$
    $node := 1$
    for $j = 1$ **to** k **do**
      $T[node] := T[node] - 1$
      $node := node \ll 1$
      if $digit \geq T[node]$ **then**
        $digit := digit - T[node]$
        $node := node + 1$
    $T[node] := 0$
    $\pi[i] \leftarrow node - 2^k$
end

Figure 4: $O(n \log n)$ uniform lexicographic unranking.

subtracted from d_i. With each movement, the stored counts along the nodes in the transversed path are decremented. Finally, upon reaching a leaf, the value π_i is set to the position associated to the leaf. Panels (a)–(h) in Fig. 3 show the paths transversed for the example, and Fig. 4 the pseudo-code for the unranking algorithm.

The reader should note that the final configuration for the unranking algorithm equals the initial configuration for the ranking algorithm and vice versa, and thus efficient implementations can exploit this fact too.

Non-Uniform Algorithms

Non-uniform algorithms solves tasks efficiently for fixed values of parameters utilizing pre-computed information

(also called advice). Typically, the computation and/or size of the advice may be large, but such investment cab be effectively amortized if a large number of computations are performed.

In the case of ranking and unranking permutations, a non-uniform algorithm only works for permutations of fixed size. If the size of the permutation changes, another advice must be generated before running the algorithm again.

The idea behind Korf-Schultze's algorithm is to use a table T_r to store for each integer of n bits, the number of bits equal to 1 in its binary representation. The table is used as follows.

Recall that to rank a permutation π, we need to compute the d_i's, and that d_i equals π_i minus the number of elements to the left of π_i that are less than it. If, as we compute each d_i's, we set to 1 the π_i-th bit in an integer N of n bits, then it is not hard to see that

$$d_i = \pi_i - T_r[N \gg (n-i)].$$

Indeed, in our example, we have

i	π_i	N before	T_r	d_i	N after
0	2	0000 0000	0	2	0000 0100
1	5	0000 0100	1	4	0010 0100
2	7	0010 0100	2	5	1010 0100
3	1	1010 0100	0	1	1010 0110
4	4	1010 0110	2	2	1011 0110
5	6	1011 0110	4	2	1111 0110
6	0	1111 0110	0	0	1111 0111
7	3	1111 0111	3	0	1111 1111

where the third column refers to integer N before the update at iteration i, and the last column refers to N after the update.

The table T_r, which is precomputed and stored in memory, allows us to compute any number of rankings in linear time. The size of T_r is clearly $2^n \log n$ bits. For $n = 16$, we need to store $65,536$ entries each requiring at most 4 bits for a total of 256K bytes, and for $n = 25$ the table contains $33,554,432$ entries each requiring at most 5 bits for a total size of 20M bytes which is a considerable amount of memory.

Ranking

The idea is very simple, instead of the big table T_r, store *a smaller table T_r^** of size $2^m \log m$ bits that indexes an integer N of m bits into the number of bits equal 1 in N. Given such table, the ranking algorithm needs to access the table *multiple times* depending on m. Thus, m should be treated as a parameter that tradeoffs the size of the advice versus the running time. Before explaining how to exploit this trade off in practice, let us explain the ranking algorithm.

In our example, consider the case $m = 4$. We have a table T_r^* that maps integers of 4 bits into the number of bits equal to 1 in each such integer. Since the number of bits equal to 1 in 8 bits, equals the sum of the number of bits equals to 1 in each 4-bit half, we have

$$T_r[N] = T_r^*[N \& mask] + T_r^*[(N \gg 4) \& mask],$$

where $mask$ is the 4-bit integers with all bits equal to 1.

Θ	$m = \log n$	$m = n^\epsilon$	$m = n/\log n$
$T(n)$	$n^2/\log n$	$n^{2-\epsilon}$	$n \log n$
$A(n)$	$n \log \log n$	$2^{n^\epsilon} \log n$	$2^{n/\log n} \log n$

Table 1: Asymtotic growth rates (Θ) for time and space for the three choices of m in the non-uniform ranking algorithm.

In general, we have the relation

$$T_r[N] = \sum_{k=0}^{\lceil n/m \rceil} T_r^*[(N \gg mk) \& mask],$$

where $mask$ is the m-bit integers with all bits equal to 1. For the case of $n = 25$ and $m = 5$, above expression requires 5 access (per digit d_i) to the table T_r^* whose size is $2^5 \lceil \log 5 \rceil = 96$ bits.

Tradeoffs

The parameter m has influence on both the running time and the stored size. A large m, implies more size but less computation time, while a small m implies less size but more computation time. The extremes values of m are easy to calculate yet deserve attention. For $m = n$, we obtain Korf-Schultze's algorithm with an advice of size $A(n) = 2^n \log n$ and linear running time. For $m = 1$, we obtain the naive algorithm with constant space and quadratic running time.

If m is constant, the quadratic running time is reduced by a constant factor but only need to store a constant amount of space $\Theta(2^m \log m)$. If $m = n/c$, for constant c, we get a linear time algorithm reducing the amount of space to $A(n) = \Theta(2^{n/c} \log n)$ by an exponential factor; this case is a good choice when n is small.

For large n, we consider the parameter m as a function of n. Of special interest are the three cases: $m = \log n$, $m = n^\epsilon$, for $\epsilon > 0$, and $m = n/\log n$. Table 1 shows the obtained exact asymptotic growth rates for the running times and advice sizes for each choice of m.

In the second case, $A(n)$ is dominated by c^n for every $c > 1$, i.e. $A(n)/c^n \to 0$, and thus the algorithm uses sub-exponential space. For instance, if $m = \sqrt{n}$, then $T(n) = \Theta(n^{3/2})$ and $A(n) = \Theta(2^{\sqrt{n}} \log n)$. In the last case, $A(n)$ is also sub-exponential but dominates $2^{n^\epsilon} \log n$. Finally, observe that none of these three cases is superior to the uniform algorithm that runs in time $O(n \log n)$ and uses $O(n)$ space.

Unranking

To unrank in linear time, consider a table T_u that maps relative positions $0 \le p < n$ and integers of n bits into positions. The entry $T_u[p, N]$ indicates the position of the pth 0-bit in N. Then, $\pi_{T_u[d_i, N_i]} = i$ where $N_0 \doteq 0$ and $N_{i+1} \doteq N_i | (1 \ll \pi_i)$. The size of table T_u is clearly $\Theta(n 2^n)$.

As before, we can reduce the space needed by using a smaller table T_u^* of size $\Theta(m 2^m)$. The running time is this case becomes $T(n) = \Theta(n^2/m)$ and the space $A(n) =$

$\Theta(m2^m)$. The table T_u^* maps a position $0 \le p < m$ and m-bit integer N into the position of the pth zero in N if there is such zero or to m otherwise.

In our example, for $m = 4$, T_u^* maps positions $0 \le p < 4$ and 4-bit integers:

T_u^*	0000	0001	0010	0011	$\ldots$	1110	1111
0	0	1	0	2	$\ldots$	0	4
1	1	2	2	3	$\ldots$	4	4
2	2	3	3	4	$\ldots$	4	4
3	3	4	4	4	$\ldots$	4	4

Let us define the function $\widehat{T}_u(p, N)$ as

$$\widehat{T}_u(p, N) \doteq \left\{ \begin{array}{ll} 0 & \text{if } p < 0, \\ T_u^*[p, N] & \text{if } p \ge 0. \end{array} \right.$$

The relation between the table T_u and the function $\widehat{T}_u$ is

$$T_u[p, N] = \widehat{T}_u(p_0, N \& mask) + \widehat{T}_u(p_1, (N \gg 4) \& mask)$$

where $p_0 \doteq p$ and $p_1 \doteq p_0 - m + T_r(N \& mask)$. For example,

$$\begin{aligned}
T_u[2, 0000\,0000] &= \widehat{T}_u(2, 0000) + \widehat{T}_u(-2, 0000) \\
&= 2 + 0 = 2 \implies \pi_0 = 2 \\
T_u[4, 0000\,0100] &= \widehat{T}_u(4, 0100) + \widehat{T}_u(1, 0000) \\
&= 4 + 1 = 5 \implies \pi_1 = 5 \\
T_u[5, 0010\,0100] &= \widehat{T}_u(5, 0100) + \widehat{T}_u(2, 0010) \\
&= 4 + 3 = 7 \implies \pi_2 = 7 \\
T_u[1, 1010\,0100] &= \widehat{T}_u(1, 0100) + \widehat{T}_u(-2, 1010) \\
&= 1 + 0 = 1 \implies \pi_3 = 1 \\
T_u[2, 1010\,0110] &= \widehat{T}_u(2, 0110) + \widehat{T}_u(0, 1010) \\
&= 4 + 0 = 4 \implies \pi_4 = 4 \\
T_u[2, 1011\,0110] &= \widehat{T}_u(2, 0110) + \widehat{T}_u(0, 1011) \\
&= 4 + 2 = 6 \implies \pi_5 = 6 \\
T_u[0, 1111\,0110] &= \widehat{T}_u(0, 0110) + \widehat{T}_u(-1, 1111) \\
&= 0 + 0 = 0 \implies \pi_6 = 0 \\
T_u[0, 1111\,0111] &= \widehat{T}_u(0, 0111) + \widehat{T}_u(-1, 1111) \\
&= 3 + 0 = 3 \implies \pi_7 = 3
\end{aligned}$$

In general, we have

$$T_u[p, N] = \sum_{k=0}^{\lceil n/m \rceil} \widehat{T}_u(p_k, (N \gg mk) \& mask)$$

where $p_0 \doteq p$ and $p_{i+1} \doteq p_i - m + T_r[(N \gg ik) \& mask]$. The total required size is $\Theta(2^m m \log m)$ bits and the over-all running time for unranking becomes $\Theta(n^2/m)$ since to compute each permutation element we need to perform n/m table lookups. The tradeoffs are similar to the ranking case.

Empirical Results

A fair comparison should be made between algorithms in the same class, i.e. either uniform or non-uniform. *We chose to make an empirical comparison between uniform algorithms over permutations of different sizes.* Although, the algorithm of Myrvold-Ruskey runs in linear time and is non-lexicographic, we decided to compare our algorithms to it and to Knuth's ranking algorithm; Knuth does not say how to unrank permutations, though.

We implemented the ranking algorithm proposed here, an iterative version of Myrvold-Ruskey's algorithm, and Knuth's algorithm. These algorithms were run on permutations of sizes 4 to 1,024, and to avoid multiple-precision arithmetic (which is the same for all algorithms once the factorial digits are computed), we modified the algorithms in order to just compute the factorial digits and not the rank.

In a first experiment, we compared our algorithms against Myrvold-Ruskey's and Knuth's over 'small' random permutations of sizes 4, 8, 12, ..., 24. For each size, 10 million ranking operations were performed on random permutations of the given size. The time reported is the cumulative time of the 10 million rankings.

Results are depicted in Fig. 5(a). The first observation is that Knuth's algorithm is clearly outperformed by the two others. As noted in (Myrvold & Ruskey 2001), this is due to the excessive use of modular arithmetic. The second observation is that our algorithm is the fastest for these sizes even though Myrvold-Ruskey's is a linear-time algorithm. This suggests that the constant factors hidden in the asymptotics is much bigger for Myrvold-Ruskey's than for our algorithm. Indeed, the ratio of running times (our/MR's) are 0.83, 0.81, 0.94, 0.83, 0.93, 0.92 (a ratio < 1 means that our algorithm runs faster than Myrvold-Ruskey's). As it can be seen, the ratios tend to increase with the sizes as expected.

A second experiment compared the algorithms over 'big' permutations of sizes 32, 34, 36, ..., 64, 128, ..., 1,024. As in the first experiment, Knuth's algorithm is clearly dominated by the other two. As shown, our algorithm runs faster than Myrvold-Ruskey's til permutations of size 128, the point in which the two curves cross each other, and then Myrvold-Ruskey's dominates. This is not surprising given that our algorithm runs in $O(n \log n)$ time and the other in linear time, yet recall that ours is lexicographic. However, even for the size 1,024, the ratio of running times of our algorithm over Myrvold-Ruskey's is only 1.10.

Finally, we did one experiment to compare the new ranking algorithm with Korf and Schultze's algorithms for permutations of size 16. As before, we only measure the time to compute the factorial digits for 10 million random permutations. Our algorithm took 9.80 seconds while Korf-Schultze's 9.70 seconds demonstrating that the latter is a bit better than the former (yet this small advantage decreases even more if the time to calculate the rank from the factorial digits is accounted for).

Summary

We have presented novel and simple $O(n \log n)$ uniform algorithms to rank and unrank permutations in lexicographic order. The algorithms can be easily modified to rank/unrank subset of elements in permutations, and to compute the inversion table for a given permutation. The new algorithms
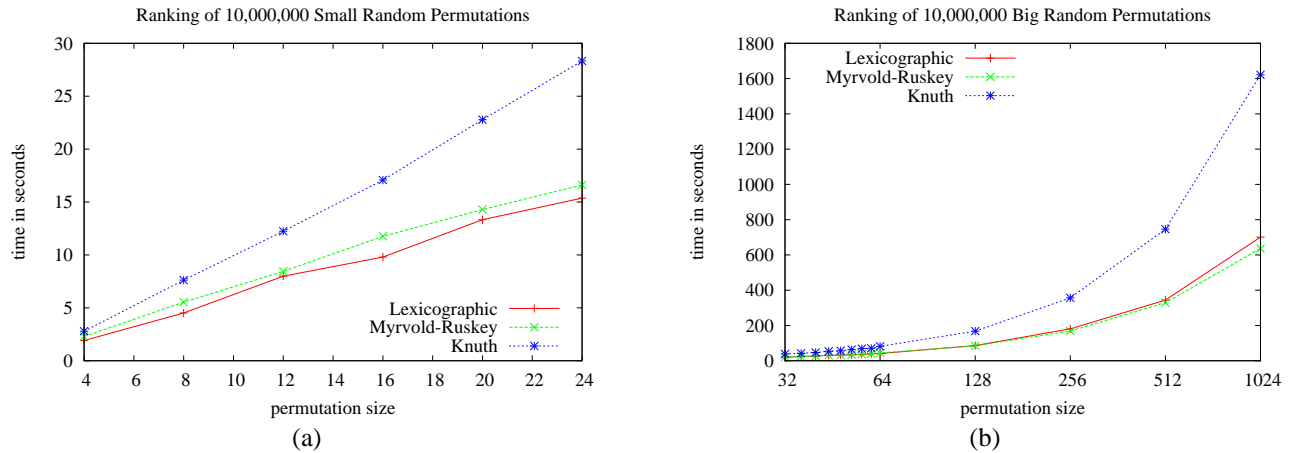
Figure 5: Experimental results for ranking random permutations of different size. Each point is the accumulated time that result of ranking 10 million random permutation of the given size.

are competitive against the linear-time algorithms in (Myrvold & Ruskey 2001) and faster than the $O(n \log n)$ algorithm in (Knuth 1973) at least for size up to 1,024.

We also presented novel non-uniform algorithms to rank and unrank permutations in lexicographic order that generalize the Korf-Schultze's algorithm (Korf & Schultze 2005). These algorithms are parametrized in m. For constant $m = n$, the algorithms reduce to the linear-time algorithms of Korf-Schultze yet they require an advice of exponential size $\Theta(n2^n)$. For $m = n/c$, the algorithms are still linear-time yet the size of the advice reduces to $\Theta(n2^{n/c}$, that is still exponential but can be a good choice when the size of the permutation is small and a great number of rank/unrank operations need to be performed. The resulting asymptotic behaviors in time and space are analyzed for other interesting values of the parameter m.

In the future, we would like to develop incremental version of the new algorithms as some applications in heuristic search will benefit from such.

References

Cormen, T.; Leiserson, C.; and Rivest, R. 1990. *Introduction to Algorithms*. MIT Press.

Critani, F.; Dall'Aglio, M.; and Biase, G. D. 1997. Ranking and unranking permutations with applications. *Innovation in Mathematics* 15(3):99–106.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

ietz, P. F. 1989. Optimal algorithms for list indexing and subset ranking. In *Proc. of Workshop on Algorithms and Data Structures*, 39–46. Springer LNCS 382.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Knuth, D. E. 1973. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley.

Korf, R., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134:9–22.

Korf, R., and Schultze, P. 2005. Large-scale, parallel breadth-first search. In Veloso, M., and Kambhampati, S., eds., *Proc. 20th National Conf. on Artificial Intelligence*, 1380–1385. Pittsburgh, PA: AAAI Press / MIT Press.

Myrvold, W. J., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79(6):281–284.

Ruskey, F.; Jiang, M.; and Weston, A. 1995. The Hamiltonicity of directed σ-τ Cayley graphs (or: A tale of backtracking). *Discrete Appl. Math.* 57(1):75–83.