A CAT Algorithm for Generating Permutations with a Fixed Number of Inversions

Scott Effler Frank Ruskey [*]

Department of Computer Science, University of Victoria, Canada

Abstract

We develop a constant amortized time (CAT) algorithm for generating permutations with a given number of inversions. We also develop an algorithm for the generation of permutations with given index.

Key words: Permutation, exhaustive listing, inversion, index.

1 Introduction

Among all statistics of permutations, the number of inversions are of paramount importance to computer scientists because of their connection with sorting and searching algorithms. Furthermore, they have interesting combinatorial properties and are intensely studied in the mathematical literature (e.g., Margolis [7], Clark [3]). Both the computational and mathematical properties of permutations with a given number of inversions are nicely summarized by Knuth [4].

However, we know of no published algorithms for generating all n-permutations with a given number of inversions (or with given index). There are many published algorithms for generating various types of permutations and many of these run in constant time per permutation, in an amortized sense. For example, there are algorithms for derangements [1], involutions [10], up-down permutations [2], and linear extensions of posets [9],[5].

[*] Research supported in part by NSERC grant OGP0003379.
Email addresses: `seffler@csc.uvic.ca` (Scott Effler), `fruskey@cs.uvic.ca` (Frank Ruskey).

An algorithm runs in constant amortized time (CAT) if the amount of computation, after a small amount of preprocessing, is proportional to the number of objects that are generated [8].

An *inversion* of a permutation π is a pair (π_i,π_j) where $i < j$ and $\pi_i > \pi_j$ [4]. Let $I(\pi)$ denote the number of inversions in π. Clearly, $0 \leq I(\pi) \leq \binom{n}{2}$, with the extremes occurring for $12\cdots n$ and $n\cdots 21$. Following Knuth [4] we use $I_n(k)$ to denote the number of permutations of $[n]$ with k inversions. By $[n]$ we denote the set $\{1, 2, \ldots, n\}$. This paper presents a CAT algorithm for generating permutations of $[n]$ with a given number of inversions. The algorithm is outlined in Section 2.

The *index* of a permutation π is the sum of indices j such that $\pi_j > \pi_{j+1}$, denoted $J(\pi)$ [4]. A theorem of MacMahon [8] establishes a bijection between classes of permutations counted by the two statistics.

Theorem 1.1 *The number of permutations of $[n]$ with k inversions is the same as the number of permutations of $[n]$ with index k; i.e., both sets have cardinality $I_n(k)$.*

Even though there is a bijection between two sets, a fast algorithm for generating one set will not necessarily imply the existence of a fast algorithm for generating the other set. An algorithm for generating permutations with fixed index k that experimentally appears to be CAT in the range $2 < k < \binom{n}{2} - 2$ is given in Section 3.

2 Generating Permutations with a Given Number of Inversions

In this section, we develop a CAT algorithm for generating all permutations $\pi \in \mathbb{S}_n$ where $I(\pi)$ is equal to some given value k.

2.1 Algorithm

The algorithm works from right to left in the one line notation of the permutation by placing elements one at a time. The key to the algorithm is the following lemma.

Lemma 2.1 *Given natural numbers n, k with $0 \leq k \leq \binom{n}{2}$, and $x \in [n]$, there is a permutation $\pi \in \mathbb{S}_n$ of the form $\pi = \pi_1\pi_2\cdots\pi_{n-1}x$ such that $I(\pi) = k$ if*

```
gen( $n$, $k$ )
  (1) **if** $n = 0 \wedge k = 0$ **then** output $\pi_1\pi_2\cdots\pi_N$
  (2) **for each** $x \in [N] \setminus \{\pi_{n+1}, \ldots, \pi_N\}$
        (a) **if** $n - rank(x) \leq k \leq \binom{n-1}{2} + n - rank(x)$ **then**
              (i) $\pi_n := x$
              (ii) gen( $n-1$, $k - n + rank(x)$ )
```

Fig. 1. Generating Permutations With Given Number of Inversions

and only if

$$n - x \leq k \leq \binom{n-1}{2} + n - x. \tag{1}$$

Proof. We must show both directions of the lemma.
$\Rightarrow$: Suppose $\pi = \pi_1 \cdots \pi_{n-1} x$ and $I(\pi) = k$. The number of inversions in $\pi_1 \cdots \pi_{n-1}$ satisfies $0 \leq I(\pi_1 \cdots \pi_{n-1}) \leq \binom{n-1}{2}$. There are exactly $n - x$ inversions of the form (π_i, x). Thus, we have $n - x \leq I(\pi) \leq \binom{n-1}{2} + n - x$.
$\Leftarrow$: Suppose the inequality in the lemma is satisfied for n, k, and x. There is a permutation $\pi_1\pi_2\cdots\pi_{n-1}$ of $[n] \setminus \{x\}$ such that $I(\pi_1\pi_2\cdots\pi_{n-1}) = k - n + x$ since $0 \leq k - n + x \leq \binom{n-1}{2}$ by (1). But then $\pi_1\pi_2\cdots\pi_{n-1}x \in \mathbb{S}_n$ and $I(\pi_1\pi_2\cdots\pi_{n-1}x) = k$. □

We only recurse if it is possible to extend the partial permutation to a permutation with the given number of inversions. That is to say, no branch in the computation tree is a dead-end. Such algorithms are said [8] to be BEST (backtracking ensuring success at terminals). BEST algorithms are often fast in practice and are generally easier to analyze.

Our algorithm generates permutations of $[N]$. Define $rank(x)$ as the position of x in the ordered list $X = [N] \setminus \{\pi_{n+1}, \ldots, \pi_N\}$ of remaining elements. Notice that the $n - rank(x)$ is the number of inversions created by placing element x at the current position, n. The method is outlined in Figure 1.

Lemma 2.2 *Given that $\pi_{n+1} \cdots \pi_N$ is an $(N-n)$-permutation of $\{1, 2, \ldots, N\}$, the call gen(n, k) generates all permutations $\pi_1\pi_2\cdots\pi_N$ of $\{1, 2, \ldots, N\}$ such that $I(\pi_1 \cdots \pi_n) = k$.*

Proof. Our proof is by induction on n. Since the empty permutation has no involutions, the lemma is true when n=0.
There are precisely $n - rank(\pi_n)$ inversions of the form (π_i, π_n). By Lemma 2.1 we know that π_n can be any value such that $n - rank(x) \leq k \leq \binom{n-1}{2} + n - rank(x)$. Inductively, for each possible value of $x = \pi_n$, the recursive call *gen*$(n-1, k-n+rank(\pi_n))$ generates all permutations for which $I(\pi_1 \cdots \pi_{n-1}) = k - n + rank(\pi_n)$. Together with the $n - rank(\pi_n)$ inversions created by element π_n, we have generated all n-permutations with k inversions. □

Corollary 2.1 *The call gen(N, K) generates all permutations $\pi_1\pi_2\cdots\pi_N$ of $\{1, 2, \ldots, N\}$ such that $I(\pi_1\pi_2\cdots\pi_n) = K$.*

The following result is important in determining the order to traverse the list X of remaining elements of the permutation.

Lemma 2.3 *If $a_1 < a_2 < \cdots < a_n$ where $\{a_1, a_2, \ldots, a_n\} = [N]\backslash\{\pi_{n+1}, \ldots, \pi_N\}$ then those a_i that satisfy the condition in lemma 2.1 form a contiguous subsequence $a_i, a_{i+1}, \ldots, a_j$ where $i = 1$ or $j = n$.*

Proof. It is clear from (1) that the values occur in a contiguous subsequence. We will show, under the assumption that $0 \le k \le \binom{n}{2}$, that (1) is satisfied for $x = 1$ or $x = n$ (or both), which will suffice to prove the lemma.

So suppose that (1) fails for $x = n$. Then either $k < 0$ or $k > \binom{n-1}{2}$. Since the former cannot occur we have

$$\binom{n-1}{2} < k \le \binom{n}{2}.$$

Hence

$$n - 1 \le \binom{n-1}{2} < k \le \binom{n}{2} = \binom{n-1}{2} + n - 1,$$

which is (1) when $x = 1$. □

If $0 < k \le \binom{n-1}{2}$ then we traverse the list in reverse order from the largest to the smallest element. Otherwise, we traverse the list in order from the smallest to the largest element. In either case, if the test in step 2(a) of the algorithm of Figure 1 fails, we stop traversing the list. Implemented in this way, the algorithm has the crucial property that its running time is proportional to the total number of nodes in its computation tree. In order to maintain the list we use the technique dubbed "dancing links" by Knuth [6].

2.2 Example

As an example, consider generating all permutations of length 4 with 2 inversions. The five such permutations are $\{3124, 2314, 2143, 1423, 1342\}$. The computation tree is given in Figure 2. The list of remaining permutation elements at each node is given in square brackets followed by the current values of n and k, along with the current partial permutation.

Consider the leftmost child of the root as an example. We already know that any child permutation ends in a 4 and has two inversions to the left of the current position. The next step is to find a value for the third position. This

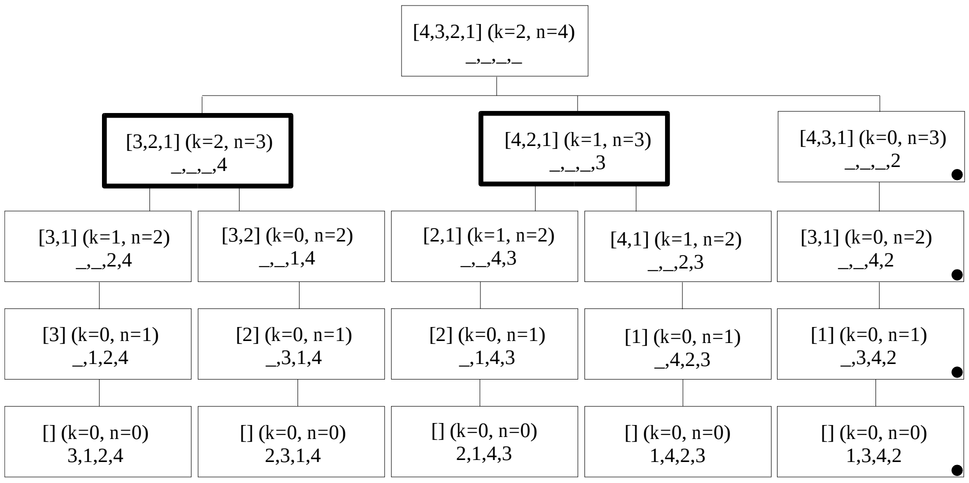

Fig. 2. Example of the Algorithm of Figure 1

position cannot be 3 since this leaves only 1 and 2, which can only produce one inversion. Therefore, this node has two children, as shown in the computation tree. This process is continued recursively until all positions are filled in.

2.3 Path Elimination

We notice that there may be many successive nodes in the computation tree with only one child. This leads to inefficiency since, for example, to generate the one permutation with $k = 0$ takes time $\Theta(n)$, and to generate the $n-1$ permutations with $k = 1$ takes time $\Theta(n^2)$. To combat this problem, we will apply a path elimination technique (PET) from [8] and stop the recursion early. This will allow us to eliminate most, but not all of the nodes in the recursive computation tree that have one child. Details of this process follow.

First, if $k = 0$, there is always just one valid permutation of X; the one with all elements ascending. We do not eliminate such paths, unless they arise from a $k = 1$ node higher in the tree. If $k = 1$, we can easily generate all valid permutations of X in constant amortized time by starting with all elements in ascending order and swapping one pair of adjacent elements at each step. For example, if $n = 4$ and $X = [4]$, the permutations are $\{2134, 1324, 1243\}$. So, if $k = 1$, we stop the recursion and generate the $n-1$ permutations directly in time $\Theta(n)$.

Similarly, if $k = \binom{n}{2}$, then there is exactly one valid permutation; the the one with all elements descending. To eliminate many of these 1-paths we notice that if $k = \binom{n}{2} - 1$ we can easily generate all valid permutations in constant amortized time by starting with all elements in descending order and swapping one pair of adjacent elements at each step. So, if $k = \binom{n}{2} - 1$ we again stop the recursion and generate the $n-1$ permutations directly in time $\Theta(n)$.

In Figure 2 the two nodes with darker borders are the roots of subtrees with $k = 1$ and $k = \binom{n}{2}$ (with $n = 3$). The subtrees at these nodes are replaced with non-recursive calls. One 1-path remains and it is indicated by the nodes with the black dots in their lower right corners. The cost of computing this path is assigned to the subtree rooted at the highlighted $k = 1$ node.

Now, we state and prove the main theorem of this paper. The argument that our algorithm runs in constant amortized time is dependent upon the list traversal and path elimination techniques discussed earlier.

Theorem 2.1 *The algorithm of Figure 1 with path elimination techniques generates permutations with a given number of inversions in constant amortized time.*

Proof. By our earlier discussion the computation time of the algorithm is proportional to the number of nodes in its recursive computation tree. If call with $k = 1$ or $k = \binom{n}{2} - 1$ occurs, then the $n-1$ permutations are generated in $\Theta(n)$ time by a separate non-recursive routine and we regard the corresponding subtrees as being replaced by a single node in the recursive computation tree. Now every root of a 1-path in the computation tree has $k = 0$ or $k = \binom{n}{2}$. A root r of a 1-path with $k = 0$ can occur only once amongst its siblings, and it must have a sibling q with $k = 1$. The $\Theta(n)$ cost of generating the one path is charged to the $n - 1$ leaves of the subtree represented by q. A root of a 1-path with $k = \binom{n}{2}$ is handled in an analogous manner. A subtree can only be charged once by this scheme (unless $n = 3$ in which case it can be charged twice). If $1 < k < \binom{n}{2} - 1$, then an internal node of the computation tree has at least two children. Thus the total number of such internal nodes is less than the number of leaves. □

3 Generating Permutations with Given Index

In this section, we are concerned with the generation of all π such that $J(\pi)$ is equal to some given value. The algorithm for this problem has a similar flavor to the algorithm of Figure 1 for generating permutations with a given number of inversions, but there are significant and subtle differences. For fixed k in the range $2 < k < \binom{n}{2} - 2$ the algorithm appears to be CAT in the sense that the amortized time per permutation is observed to be decreasing for large enough n.

```
$gen(n,k)$
 (1) if $n = 0 \wedge k = 0$ then output $\pi_1\pi_2\cdots\pi_N$
 (2) for each $x \in [N] \setminus \{\pi_{n+1},\ldots,\pi_N\}$
     (a) if $x > \pi_{n+1}$ then $k' := k - n$ else $k' := k$;
     (b) if $n - rank(x) \le k' \le \binom{n}{2} - rank(x) + 1$ then
         (i) $\pi_n := x$;
         (ii) $gen(n-1,k')$;
```

Fig. 3. Generating Permutations With Given Index (assumes $\pi_{N+1} = N + 1$).

3.1 Algorithm

As was the case for inversions, this algorithm works from right to left. We assume that we are generating permutations of $[N]$. In order to treat all positions in a uniform manner we will assume that $\pi_{N+1} = N + 1$. Some simple results will help us construct an algorithm in much the same way as was done for inversions.

Consider a permutation of $[n]$ of the form $\pi_1\pi_2\cdots\pi_{n-1}x$. For x fixed in position n, the permutation of smallest index is

$$(x{+}1)(x{+}2)\cdots(x+(n{-}x))\ 12\cdots(x{-}1)\ x$$

and the permutation of greatest index is

$$(x{-}1)\cdots 21\ (x{+}(n{-}x))\cdots(x{+}2)(x{+}1)\ x.$$

The index of those permutations is $n - x$ and $\binom{n}{2} - x + 1$, respectively. Based on this observation and reasoning similar to that used to prove Lemma 1, we are led to the following lemma.

Lemma 3.1 *Given a triple (n,x,k) of integers, there is a permutation π of the form $\pi = \pi_1\cdots\pi_{n-1}x$ such that $J(\pi) = k$ if and only if*

$$n - x \le k \le \binom{n}{2} - x + 1. \qquad (2)$$

The algorithm of Figure 3 generates all n-permutations with index k. It is based on Lemma 3.1, except that we need to use $rank(x)$ instead of x and if $x > \pi_{n+1}$ then the index target needs to be adjusted by n.

Lemma 3.2 *Given that $\pi_{n+1}\cdots\pi_N$ is an $(N-n)$-permutation of $[N]$, the call $gen(n,k)$ will generate all permutations $\pi_1\pi_2\cdots\pi_N$ of $[N]$ such that $J(\pi_1\pi_2\cdots\pi_n) = k$.*

Proof. We will prove this result by induction on n. Since the empty permutation has index 0. the lemma is true when n=0.

If $n = N$ or $\pi_n < \pi_{n+1}$ then π_n does not contribute to the index. By Lemma 3.1 we know that π_n can be any value such that $n - rank(x) \leq k \leq \binom{n}{2} - rank(x) + 1$. For each possible value of $x = \pi_n$, we recurse with call $gen(n-1, k)$. By our inductive assumption this call generates all permutations such that $J(\pi_1 \cdots \pi_{n-1}) = k$.

If $\pi_n > \pi_{n+1}$ then π_n contributes n to the index. By Lemma 3.1 we know that π_n can be any value such that $n - rank(x) \leq k - n \leq \binom{n}{2} - rank(x) + 1$. For each possible value of $x = \pi_n$, we recurse with call $gen(n-1, k-n)$. By our inductive assumption this call generates all permutations such that $J(\pi_1 \cdots \pi_{n-1}) = k - n$. Together with the n added to the index by π_n, we have generated all n-permutations with index k. □

Corollary 3.1 *The call $gen(N, K)$ will generate all permutations $\pi_1 \pi_2 \cdots \pi_N$ of $\{1, 2, \ldots, N\}$ such that $J(\pi_1 \cdots \pi_n) = K$.*

A similar result to Lemma 2.3 is necessary to determine which order to traverse the list $X = [N] \setminus \{\pi_{n+1}, \ldots, \pi_N\}$ of remaining elements of the permutation.

Lemma 3.3 *If $a_1 < \cdots < a_{p-1} < \pi_{n+1} < a_p < \cdots < a_n$ where $\{a_1, a_2, \ldots, a_n\} = [N] \setminus \{\pi_{n+1}, \ldots, \pi_N\}$ then,*

- *for $1 \leq i < p$, those a_i that satisfy the condition in Lemma 3.1 for (n, a_i, k) form a contiguous subsequence $a_i, a_{i+1}, \ldots, a_j$ where $i = 1$ or $j = p - 1$, and*
- *for $p \leq i < n$, those a_i that satisfy the condition in Lemma 3.1 for $(n, a_i, k - n)$ form a contiguous subsequence $a_i, a_{i+1}, \ldots, a_j$ where $i = p$ or $j = n$.*

Proof. It is clear from (2) that the values occur in a contiguous subsequence.

For $1 \leq i < p$, we will show that if (2) is satisfied for a_i then it is satisfied for $a_{i+1}, \ldots, a_{p-1}$ or $a_1, \ldots, a_{i-1}$ (or both), which will suffice to prove the first half of the lemma. If (2) is satisfied for a_i then we have

$$n - rank(a_i) \leq k \leq \binom{n}{2} - rank(a_i) + 1.$$

If $n - rank(a_i) \neq k$ then $x = rank(a_j)$ satisfies (2) for $1 \leq j < i$. If $\binom{n}{2} - rank(a_i) + 1 \neq k$ then $x = rank(a_j)$ satisfies (2) for $i < j < p$.

For $p \leq i \leq n$, the argument is identical except k is replaced by $k - n$. □

At Step (2) of the algorithm, we will traverse the list X up to four times. Once from largest to smallest, once from smallest to largest and in both directions starting from the largest element a_{p-1} less than π_{n+1}. Our data structure allows us to identify a_{p-1} in the list in constant time. In all of these cases, if the test of step 2(b) of the algorithm of Figure 3 fails, we stop traversing the list. Also note that we must keep track of where each traversal ends, making sure that we do not recurse on the same element twice. Implemented in this way, the algorithm has the crucial property that its running time is proportional to the total number of nodes in its computation tree.

Similar improvements can be made to this algorithm as were made to the inversion generation algorithm so as to produce a constant amortized time algorithm, but only if $2 < k < \binom{n}{2} - 2$, and this has only been observed experimentally. To be precise, for $n > 4$, the ratio of recursive calls to permutations generated is strictly decreasing. To get this CAT behavior we must use the path elimination techniques described in the following paragraphs.

Let $\{a_1, a_2, \ldots, a_n\} = [N] \setminus \{\pi_{n+1}, \ldots, \pi_N\}$ where $a_1 < a_2 < \cdots < a_n$. If $k = 0$ and $a_n < \pi_{n+1}$ then there is only one valid permutation; the one with all elements ascending. We do not eliminate such paths, unless they arise from a $k = 1$ node, where $a_n < \pi_{n+1}$, higher in the computation tree. If $k = 1$ and $a_n < \pi_{n+1}$, we can easily generate all valid permutations in constant amortized time by starting with the identity and doing swaps $(\pi_1, \pi_2), (\pi_1, \pi_3), \ldots, (\pi_1, \pi_n)$, printing each intermediate permutation.

Similarly, if $k = \binom{n}{2}$ and $a_n < \pi_{n+1}$, then there is exactly one valid permutation; the one with all elements descending. To eliminate many of these 1-paths we notice that if $k = \binom{n}{2} - 1$ and $a_n < \pi_{n+1}$ we can easily generate all valid permutations in constant amortized time by starting with all elements in descending order and swapping as done above for $k = 1$.

4 Final Remarks

The algorithms described in this paper are used in the second author's "Combinatorial Object Server" at `httpd://www.theory.csc.uvic.ca/~cos/` in the permutations section. Java implementations of the algorithms can also be downloaded there.

References

[1] Selim G. Akl, *A New Algorithm for Generating Derangements*, BIT 20 (1980) 2–7.

[2] Bruce Bauslaugh and Frank Ruskey, *Generating Alternating Permutations Lexicographically*, BIT, 30 (1990) 17-26.

[3] L.A. Clark, *An Asymptotic Expansion for the Number of Permutations with a Certain Number of Inversions*, Electronic Journal of Combinatorics, 7 (2000) #R50.

[4] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching.* Second Edition, Addison-Wesley, 1998.

[5] Donald E. Knuth, *The Art of Computer Programming, Volume 4: Prefascicle 2B*, January 2002.

[6] Donald E. Knuth, *Dancing Links*, Millennial Perspectives in Computer Science, (Houndmills, Basingstoke, Hampshire: Palgrave, 2000), 187-214.

[7] B.H. Margolis, *Permutations with inversions*, Journal of Integer Sequences, 4 (2001) Article 01.2.4.

[8] Frank Ruskey, *Combinatorial Generation.* Working version of book in progress, 1995–2002

[9] G. Pruesse and F. Ruskey, *Generating Linear Extensions Fast*, SIAM J. Computing, 23 (1994) 373–386.

[10] D. Roelants van Baronaigien, *Constant time generation of involutions*, Proceedings of the Twenty-third Southeastern International Conference on Combinatorics, Graph Theory, and Computing (Boca Raton, FL, 1992). Congr. Numer. 90 (1992), 87–96.