On a certain recursive algorithm for generating permutations

Paweł Gburzyński\textsuperscript{a}, Andrzej Salwicki\textsuperscript{b}

\textsuperscript{a}Department of Computing Science, University of Alberta, Edmonton, Alberta, CANADA T6G 2E8
\textsuperscript{b}Faculty of Mathematics and Natural Sciences, Cardinal Stefan Wyszynski University, ul. Wóycickiego 1/3, 01-938 Warsaw, POLAND

Abstract

We introduce an algorithm for generating all permutations of numbers between 1 and \(N\) without swapping elements. To the best of our knowledge, the algorithm has never been published, even though it has been known to us for some time. The algorithm is short and efficient, yet its behavior is not obvious from the code, mostly owing to the recursion. While the problem, as such, may not be an exciting research topic any more, we believe that the algorithm touches upon a few interesting issues, and, for one thing, provides an educational case study in recursion.

Keywords: Permutations, permutation generation, loopless algorithms, recursive algorithms, analysis of algorithms

1. Introduction

New algorithms for generating permutations are probably not in demand today, as the issue seems to have been exhaustively discussed in the literature (as usual, the opus of D.E. Knuth [1] provides the authoritative reference). The algorithm that we are about to present has been known to us since ca. 1980, although, to the best of our knowledge, it has never been published (in particular, it is not mentioned in [1]). It comes with a story which is best told in a narrative less formal than a research paper, because announcing upfront the algorithm’s purpose removes from the yarn the essential element of suspense.

When the algorithm was first introduced to an audience of students in an introductory programming course, it caused some confusion. During an exam, the students were asked to guess what the program was doing, explain its flow control, and describe the output produced, i.e., tell the ordering of the resulting sequence of permutations. The era of portable communication/computing gadgets (so nightmarish from the viewpoint of the contemporary examiner) was still far ahead, so the students were left to their own “devices.” To the one of us who devised the exam and the question the problem seemed non-trivial but well within the grasp of a university student in computer science who had acquired the understanding of recursion in programming. But it was in fact a disaster. When two local and accomplished faculty experts in algorithm design and analysis were subsequently shown the question, their reflex, after a brief deliberation, was to run to the computer room and see what happens. Things have settled eventually, but the incident left us with a feeling that has persisted to this day, that some questions deserve a study. This has stimulated a recent effort to describe and analyze the algorithm’s properties with the formal tools of algorithmic logic [2] which is the topic of a separate publication (in preparation). In the present note we introduce the algorithm from the practical angle (say, for the record) and briefly discuss some of its properties.

2. The algorithm

The algorithm has the form of a recursive procedure operating on three global variables:
var N, k : integer; A : array [1..N] of integer;

The array will contain consecutive permutations generated by the algorithm and its size should be N. We want to express the algorithm in a simple, generally understood programming language such that the code is (almost) immediately runnable. It makes sense to use a Pascal lookalike (say, instead of C) because it is convenient to have the array indexed from 1. Strictly speaking, the above declaration is incorrect because the size of A is determined by a variable. We want to say that the array should be at least as long as N and its effective size is precisely N. Any size not less than N will do in a real-life program.

Before the procedure is invoked, N is set to the requisite parameter (and henceforth appears as a constant), the array A is initialized to zeros (for the relevant indexes from 1 to N), and k is set to 1. The procedure is listed below:

```
procedure F();
var i : integer;
begin
  ∆: if k>N then
      ready();
  else
    Ψ: for i:=1 to N do
      Φ: if A[i]=0 then
          Γ: A[i]:=k;
          k:=k+1;
          F();
          k:=k−1;
          A[i]:=0;
      end if
    end for
  end if
end
```

Each call to ready marks the moment when A contains a new permutation (which can be printed out or otherwise used). Thus, the algorithm (in its boilerplate variant listed above) actually generates all permutations, e.g., as opposed to returning them one by one on subsequent invocations.

Before looking into the algorithm’s behavior, let us reflect on the author’s inspiration. An exam was being devised, most of the questions had been written down, and the one topic remaining to be addressed was recursion. The problem had to be stated briefly and it had to touch upon all the essential aspects of data from the viewpoint of a recursive algorithm. Thus, we needed at least one local variable and at least one global variable (both of them relevant), and (of course) a non-trivial recursive call. In this respect, the two global variables k and A (N is effectively a constant), and the local variable i nicely fit the bill. Consequently, one advantage of our algorithm is that it provides an educational case study in recursive programming, even if its practical significance is not evident.

### 3. Correctness

The following sequence illustrates the way to invoke F as to account for the required initialization:

```
... readln(N);
for k:=N downto 1 do A[k]:=0;
... F();
... 
```

Note that the loop setting the array elements to zero has the side effect of initializing k to 1. Thus, when the procedure is called from the outside (as opposed to its recursive invocation within itself), A is filled with zeros and k (the global variable) is equal 1. We shall show the following:
Theorem 1. When $F$ is invoked with $N \geq 1$, $k = 1$, and $A[i] = 0$, for $i = 1, \ldots, N$, then the procedure will eventually stop, ready will have been called exactly $N!$ times, and each time it is called $A$ will contain a different permutation of numbers from $1$ to $N$.

We prove the Theorem through a series of observations. The proof assumes that the semantics of some statements, notably a function/procedure call, is obvious.

Lemma 2. The value of $k$ is an invariant of the loop $\Psi$; in particular, it is not affected by the call to $F$ made within the loop. Using the lingo of algorithmic logic [2], we express this formally as:

$$ (k = x) \implies (\forall i)(i = x) $$

where $x$ is an extra variable.

Proof. Indeed, the only two statements where $k$ is modified encapsulate the single call of $F$ within itself, and the second statement undoes the effect of the first one.

This lets us draw the following:

Corollary 1. The global variable $k$ indexes the levels of the recursive invocations of $F$. Note that $k$ is initialized to 1, incremented by 1 before every call to $F$, and decremented by 1 after every return. In consequence, the maximum value reached by $k$ (and also the maximum recursion level) is $N+1$, because no more calls happen when $k$ has reached $N+1$.

Lemma 3. The algorithm terminates for every $N > 0$.

Proof. Notice that $\Psi$ is a bounded loop whose body is executed at most $N$ times. The body includes a single recursive call, and, by Corollary 1, the maximum level of recursion is $N+1$, with the topmost level $N+1$ used to execute a single statement (invoke ready). Thus, the purely “syntactic” bound on the possible number of invocations of ready, before the procedure returns from its external call, is $N^{N+1}$. This implies that the external call always returns.

Lemma 4. Let $z_A$ denote the number of elements $A[i]$, $i = 1, \ldots, N$, that are equal zero. On every execution of the procedure’s body $\Delta$, at the beginning and at the return, $z_A = N-k+1$. In particular, $z_A$ is an invariant of the procedure. Formally:

$$ (z_A = N-k+1) \implies (\Delta) (z_A = N-k+1) $$

Proof. These are the two places (programs) within the subroutine where the values of $k$ and $A$ are modified:

$$ \Theta: \begin{cases} A[i] := k; \\ k := k+1; \end{cases} \text{ and } \Omega: \begin{cases} k := k-1; \\ A[i] := k; \end{cases} $$

We shall show by induction that the invariant $z_A = N-k+1$ holds before and after either of those modifications. Immediately after $F$ has been called for the first time, with $k = 1$, all $N$ elements of $A$ are equal zero ($z_A = N$), because this is how the array has been initialized. So the invariant holds before the very first execution of $\Theta$. Suppose then that $F$ has been called for some $k = q$, $1 \leq q \leq N$, and $z_A = N-q+1$. $\Theta$ is only executed if $A[i] = 0$, and (by Corollary 1) $k$ is never zero; thus we have:

$$ (k = q \land z_A = N-q+1) \implies (\Theta) (k = q+1 \land z_A = N-q) $$

This is because $\Theta$ sets to nonzero one element of $A$ that has been zero so far and increments the value of $k$ by 1. So the invariant holds for all the cases of $\Theta$.

To prove the invariant for $\Omega$, we start from $k = N+1$ and $z_A = 0$ (which has been shown above to hold after $N$ iterations of $\Theta$), i.e.,

$$ (k = 1, z_A = N) \implies (\Theta)^N (k = N+1, z_A = 0) $$

and suppose that $k = q$, $1 < q \leq N+1$, $F$ returns from a recursive call, and we are about to execute $\Omega$. We have:

$$ (k = q, z_A = N-q+1) \implies (\Omega) (k = q-1, z_A = N-q+2) $$

which proves the invariant for $\Omega$ all the way down to $k = 1$, thus proving the theorem.
Corollary 2. From Corollary 1 and Lemma 4 we immediately infer another invariant of the loop Ψ: the \( k - 1 \) nonzero elements in \( A \) are all different and cover values from 1 to \( k - 1 \).

Corollary 3. Every time \( \text{ready} \) is about to be called, i.e., the \( \text{if} \) \( k > N \) condition is satisfied, \( k \) is equal to \( N + 1 \), \( z_A = 0 \), and the array \( A \) (indexes 1 through \( N \)) contains a permutation of numbers from 1 to \( N \). This is a straightforward conclusion from Corollary 2.

Corollary 4. Let \( i_1, \ldots, i_N \) be the values of the local copies of variable \( i \) corresponding to the levels 1, \ldots, \( N \) of the stacked invocations of \( F \). This follows from confronting the assignment to \( A[i] \) in \( \Theta \) with Lemma 4 and Corollary 2. Consequently, all permutations seen by \( \text{ready} \) are different.

Lemma 5. Consecutive invocations of \( \text{ready} \) see in \( A \) all the permutations of numbers between 1 and \( N \).

Proof. Suppose otherwise, i.e., some permutation is skipped, and denote the missing permutation by \( \{ p_j \} \) where \( p_j \in \{ 1, \ldots, N \} \), \( j \in \{ 1, \ldots, N \} \), and \( p_j \neq p_k \) for \( j \neq k \). We shall show by induction on \( k \) how the permutation is going to be produced by the algorithm (i.e., materialize in \( A \)) thus contradicting the assumption.

Let \( J(m) \) denote the index of the element \( m \in \{ 1, \ldots, N \} \) in the permutation \( \{ p_j \} \), i.e., \( p_{J(m)} = m \). The loop body \( \Phi \) is executed once for every element in \( A \). When \( k = 1 \), then \( z_A = N \) by Lemma 3, i.e., all elements in \( A \) are zeros. Thus the body of the \( \text{if} \) statement \( \Gamma \) will execute for every possible value of \( i \) between 1 and \( N \), and it will also execute for \( i = J(1) \). Consequently, there is an iteration of the loop where 1 is assigned to its position in \( \{ p_j \} \), i.e., \( p_{J(1)} = 1 \). This is accomplished with the assignment to \( A[i] \) in \( \Theta \) when \( k = 1 \). Following that assignment, the procedure will be invoked recursively with the value of \( k \) incremented by 1.

Suppose now that the procedure executes the \( \text{else} \) program \( \Psi \) when \( 1 < k \leq N \). By the inductive assumption, by Lemma 4, and by Corollaries 2 and 4, we have in \( A \) a partial permutation \( \{ q_j \} \), \( j = 1, \ldots, k - 1 \), such that \( q_j = p_j \) for \( j = 1, \ldots, k - 1 \). By Lemma 4, \( z_A = N - k + 1 \) which means that \( A \) contains exactly \( N - z_A = k - 1 \) nonzero elements at positions \( i_j, j = 1, \ldots, k - 1, i_j = J(j) \), and \( A[i_j] = j \). The remaining positions in \( A \) are zeros, in particular \( A[J(k)] = 0 \) because \( J(k) \neq J(l) \) for \( 1 \leq l < k \). Thus, the \( \text{if} \) condition for \( \Gamma \) will succeed for \( i = J(k) \), and the partial permutation \( \{ q_j \} \), \( j = 1, \ldots, k - 1 \) will be extended into \( \{ q_j \} \), \( j = 1, \ldots, k \), such that \( q_j = p_j \) for \( j = 1, \ldots, k \). This completes the proof of the Lemma.

Proof of Theorem 1. The proof follows directly from Lemmas 3 and 5.

4. The cost

The only practically interesting class of algorithms for generating permutations are the so-called loopless (or loop-free) ones [3, 4] where the cost per permutation is constant (does not depend on \( N \)). The term “loopless” is somewhat confusing: one cannot generate all permutations of \( N \) elements without a loop of some sort, and it relates to the standard (practical) approach to generating permutations where, following some initialization, a function is called each time a new permutation is needed [1, 3, 4, 5, 6, 7]. Then an algorithm is deemed loopless if its invocation to generate one (next) permutation has a constant average cost independent of \( N \). Before explaining how our algorithm can be organized after this fashion, let us determine the complexity of the original version, as presented in Section 2.

We begin by calculating the total number of invocations of \( F \) for a given \( N \), which we will denote by \( U(N) \). In the light of Lemma 2 from Section 3, the procedure is called \( N - k + 1 \) times at level \( k \). Thus, ignoring the first (external) invocation, it is called \( N \) times at level 1 (with \( k = 2 \)), and, for each of those calls, \( N - 1 \) times at level 2 (\( k = 3 \)), and so on, down to 1 time at level \( N \) (\( k = N + 1 \)) for every call at level \( N - 1 \). So we have:

\[
\begin{align*}
U(N) &= N \times (1 + U(N - 1)) \\
U(1) &= 1
\end{align*}
\]  

(6)

yielding:

\[
U(N) = N! \times \left\lfloor \frac{1}{e} \right\rfloor < eN!
\]

(7)
which shows that the average number of invocations of $F$ per permutation is bounded by $e$. If we insist on including the first (external) invocation of $F$ in the count, the summation index should be extended to $N$. It makes sense, not only as an exercise in splitting hairs, because the structure (function) of that call is the same as for any other (recursive) call with $k \leq N$, which is to say that the procedure runs through the loop. By the same token, we may prefer to exclude from the calculations the last call per permutation, made at level $N$ ($k = N + 1$), whose sole function is to “present” the permutation (the loop is not entered) and which can be easily eliminated (by checking $k$ before the recursive call). This will have the effect of subtracting $N!$ from the formula (or doing the summation from $i = 1$ instead of $i = 0$). In the end we get this expression:

$$U_f = N! \times \sum_{i=1}^{N} \frac{1}{i!} < (e - 1)N!$$  \hspace{1cm} (8)

The way the procedure has been programmed in Section 2 does not make the average number of actual operations per permutation bounded by a constant. This is because the loop iterates $N$ times at every level $k \leq N$ formally requiring $O(N)$ operations at every invocation of the subroutine except for the last one (when $k = N + 1$). To obtain the total number of turns of the loop required to produce all the permutations we should simply multiply $U_f$ (Equation 8) by $N$. This is because $U_f$ counts all the invocations of $F$ that cause the loop to be entered, and when entered it always runs $N$ times. So we have:

$$U_l = U_f \times N < (e - 1)NN!$$  \hspace{1cm} (9)

which shows that the cost per permutation is of order $N$.

One can, however, reprogram the procedure taking advantage of a list to skip over the filled entries in the array without having to traverse them in a loop. For that, in addition to the original array $A$, we introduce another array acting as a simple representation of the list:

```plaintext
var X:array[0..N] of integer;
```

The role of $A$ is now reduced to storing the permutation being constructed by the procedure, while $X$ keeps track of the free slots in $A$. The initialization/invocation sequence now looks like this:

```plaintext
... readln(N); for k:=N+1 downto 1 do X[k-1]=k; ... G(); ...
```

We rename the new variant of the procedure as $G$. Note that the array $A$ need not be initialized because the requisite information is now maintained by $X$. The size of $X$ is $N + 1$. Its role, for $i = 1, \ldots, N$, is to point to the next unused element in $A$ with respect to the element number $i$. $X[0]$ is the head of the list, and $X[N]$ contains a sentinel value $(N + 1)$ which does not represent a valid index in $A$. The procedure is now reprogrammed as follows:

```plaintext
procedure G(); var b,d:integer; begin if k>N then ready(); else begin b = 0; while X[b]<=N do begin d=X[b]; A[d]:=k; X[d]=X[b]; k:=k+1; G(); k:=k-1;
```
\[ X[b] := d; \]
\[ b := X[b]; \]
\end{while}
\end{if}
\end{end}

The modification are easily explained with reference to \( F \) in Section 2. Initially, the values in \( X \) describe the straightforward succession of indexes in \( A \) where the head points to element 1, every subsequent element of \( X \), for \( i = 1, \ldots, N-1 \) points to the next element \((i+1)\), and the last element contains a special value \((N+1)\) indicating that the list ends there. Thus, immediately after the initialization when \( k = 1 \) traversing the array through the links in \( X \) will amount to going through all its elements in exactly the same order as with the straightforward loop in \( F \). When a value is inserted into \( A \), the corresponding index in \( X \) is replaced with its successor, which has the effect of removing the index from the list for the subsequent recursive calls (formally equivalent to making the entry in \( A \) nonzero in \( F \)). The index is restored upon return from the recursive call, similar to zeroing the corresponding element of \( A \) in \( F \).

One can easily see that the new procedure carries out the same series of nonzero insertions into \( A \) as its previous version, thus accomplishing the same feat, but the time cost corresponding to a single (recursive) invocation of \( G \) is now constant. Thus, the time cost of the algorithm is:

\[ T(N) = cN! \]  

(10)

where \( c \) is a constant bounded by (and asymptotically approaching) \( e - 1 \) times the cost of the sequence of statements enveloping the recursive call to \( G \) in line 13.

In addition to reducing the cost, the re-implementation of \( F \) into \( G \) brings in one extra feature. Note that the order in which \( F \) and \( G \) generate the permutations explores the positions of higher values first. More formally, if the numbers \( 1, \ldots, k \) have been assigned to some positions in \( A \), they will stay on those positions until the remaining numbers \( k + 1, \ldots, N \) have been exhaustively permuted. Thus, for example, for \( N = 3 \), the presented permutation order is: 123, 132, 213, 312, 231, 321. Note that there is a single statement in \( G \), \[ A[d] := k; \] where a value is assigned to an element of \( A \). Both \( d \) and \( k \) are between 1 and \( N \). If \( A[j] \), for \( j = 1, \ldots, N \), is a permutation of numbers from 1 to \( N \), then \( B[A[j]] = j \) is another (inverse) permutation of the same numbers. The new permutation uses the value of \( k \) (originally interpreted as the item to be inserted into the permutation sequence) as the index of the value \( j \) (originally interpreted as the index). Thus, we can replace the statement with \[ A[k] := d; \] (swapping the two variables) to make the permutations appear in the lexicographic order, i.e., 123, 132, 213, 231, 312, 321.

5. The “loopless” variant

The algorithm is inherently recursive, with its output materializing at the topmost level, which can be seen as a disadvantage. In most practical applications, one would prefer to have a “loopless” algorithm \([3, 4]\) in the form of a function called from an external program each time a new permutation is needed. The algorithm can be reprogrammed into a non-recursive variant and tweaked to provide a function producing consecutive permutations on consecutive calls, but one can convincingly argue that the recursive form is the algorithm’s essential feature.

\[ C: \text{the consumer, i.e., an external procedure, repeatedly asks for next permutation} \]
\[ P: \text{the producer, i.e., procedure } F \text{ or } G \]

\begin{center}
\begin{tikzcd}
\text{attach}(P) & \text{attach}(C) \\
\end{tikzcd}
\end{center}

\text{Figure 1. The producer/consumer model}

The day can be saved by resorting to coroutines and turning the procedure into a \textit{producer} providing data to an external \textit{consumer} (Figure 1). The requisite tools have been available in many programming languages and environments, beginning historically with Simula67 \([8]\), becoming a standard feature of contemporary platforms and available
in several guises providing handy shortcuts for typical applications. For example, in Python [9], the procedure can be turned into a generator retaining its recursive form by applying a special type of return statement. The modification does not affect the efficiency of the implementation to a perceptible degree.

6. Final comments

One intriguing feature of our algorithm is the apparent difficulty to see its function at first sight and the wrong intuitions that it tends to connote for a first-time viewer, if presented without the spoiler. While this is a feature shared by many recursive algorithms, the present case of recursion is very simple: there is just a single recursive call, and the procedure is parameter-less. Owing to its anecdotal history, the algorithm has been used by us and some of our colleagues in a number of educational case studies and analyses illustrating the power and features of formal systems of inferring about programs. Those discussions, involving students as well as experts, have raised a few questions:

1. Why can the designer of a program (even a simple, half-a-page, self-contained fragment of code devised for educational purposes and with no malicious intentions) see things much clearer than a competent reader subsequently looking at the same piece?
2. How to best convey the “obvious” idea behind the design that, ideally, should be present there, in the very code, plain for everyone to see?
3. How to prevent misunderstandings and misrepresentations of the ideas implanted into programs by their designers? In other words, how to ensure that programs are correct?
4. How to think about programs when designing them, so the right and correct ideas can materialize, but also when reading (studying) them, so those ideas can be extracted from the code and comprehended?

The design of procedure \( F \) begun with a simple narrative that materialized in the programmer’s head: “I am going to generate all permutations of the values from 1 to \( N \) by inserting 1 into all possible places, and then, for every such insertion, inserting 2 into all places that still remain unoccupied, and so on, continuing doing so until all the values have been inserted.” It seems to explain everything there is to see about the algorithm, even though it is inadequate as a formal proof of its correctness. It can also be viewed as the most straightforward plain-language specification of the problem and, at the same time, it rather precisely explains the programmer’s intention. According to the paradigm of literate programming [10], it should thus be incorporated into the procedure’s code and become its integral component.

With this slightly more-informed look at procedure \( F \), we see that it merely follows this specification to the letter. Thus, considering that its efficiency is not worse than that of the most advanced solutions known in the area (some of which are considerably more difficult to explain), it seems that our algorithm should be viewed as the most natural solution to the problem.

References