Noname manuscript No.
(will be inserted by the editor)

A new algorithm for generating permutations without swapping elements

Paweł Gburzyński* · **Andrzej Salwicki**

Received: date / Accepted: date

Abstract We introduce an algorithm for generating all permutations of numbers between 1 and N. The algorithm is short, elegant, and no less efficient than the best known (and possible) solutions to the problem, and, to the best of our knowledge, it is also original: our exploration of the literature has failed to locate a published algorithm similar to our solution. The algorithm's behavior is not obvious from the code, yet its analysis (which makes for an interesting case study in recursion) indicates that it can be viewed as a natural solution to the problem. We present a proof of the algorithm's correctness and derive its complexity. The discussion naturally leads to two variants of the algorithm: one being easier to explain and comprehend, the other exhibiting optimal asymptotic cost.

Keywords Permutations · Permutation generation · Loopless algorithms · Recursive algorithms · Analysis of algorithms

1 Introduction

Algorithms in combinatorics, including algorithms for generating permutations of numbers or other objects, have been extensively studied (one could say exhaustively explored) in the literature. The opus of D.E. Knuth [9] elaborates at length on the importance of the problem and provides the authoritative

P. Gburzyński, *Corresponding Author
Department of Computing Science
University of Alberta
Edmonton, Alberta, CANADA T6G 2E8
E-mail: pawelg@ualberta.ca

A. Salwicki
Faculty of Mathematics and Natural Sciences
Cardinal Stefan Wyszynski University
ul. Wóycickiego 1/3, 01-938 Warsaw, POLAND
E-mail: salwicki@gmail.com

reference with a plethora of solutions, including ones for many interesting special cases. The algorithm that we are about to present has been known to us for some time, although, to the best of our knowledge, it has never been published. In particular, none of the solutions listed in [9] comes close to our algorithm, despite its being simple, natural, elegant and educational, and, at the same time, performing no worse than the best published algorithms.

The most entertaining way of introducing the algorithm would be to hide its purpose and ask the reader to guess it from the code. This does not seem possible in a research paper (so the story is bound to lose some of its suspense), but this is in fact how the algorithm was first presented, i.e., as a puzzle.[1] The difficulty of seeing its purpose from the code has been anecdotal among our professional friends and it seems to well illustrate the difference between the mindset of an algorithm designer and that of its (uninformed) analyzer. The designer may see a *natural* path towards transforming a problem formulation into a code, but the "naturalness" of that path may be difficult to arrive at by the viewer of the code, even though it can be demonstrated formally with the proper analysis.

The algorithm is recursive and, owing to the non-trivial application of the recursion, its correctness, including the halting property, is not obvious. The formal correctness proof that we present in this paper is important because one often hears that the elegance of functional programming tends to be marred with difficulties of proving termination properties of such programs [6]. In addition to proving the algorithm's correctness we also estimate its cost.

2 The algorithm

The algorithm has the form of a recursive procedure. For the basic variant introduced in this section, the procedure is named F (see Figure 1) and operates 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 effective size is 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 because it is convenient to have the array indexed from 1.

Before F is invoked for the first time (externally), N is set to the requisite parameter (and henceforth appears as a constant), the array A is initialized to zeros (for the indices from 1 to N), and k is set to 1.

Each call to ready in F 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) *generates* all permutations, e.g., as opposed

[1] While it may be too late to try to baffle the reader with a spoiled whodunit story, we nonetheless suggest them to view the code in Figure 1 pretending that the name of the culprit has not been revealed.

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

Fig. 1 The basic variant

to returning them one by one on subsequent invocations [5]. Function ready should be viewed as the *consumer* of the output produced by the algorithm. It is convenient to start the presentation with a variant where the consumer is intertwined with the procedure. Later we shall show how the two can be disentangled.

3 Correctness

The following snippet illustrates the way to invoke F including 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 prove 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 proceed through a series of formal observations. Note that F interprets three global variables (A, k, and N) and modifies two of them (A and k). These are not desirable characteristics of a "mathematical" function, but they

are essential for the case at hand. Formally, the declaration of F introduces a new (atomic) statement, i.e., an invocation of F. Before embracing this new statement as a programming construct, one would like to make sure that the statement terminates, i.e., does not lead to an infinite loop, especially when, as in our case, the statement appears within its own implementation [12].

Lemma 2 *The value of k is an invariant of the loop Ψ; in particular, it is not affected by the call to* F *made within the loop. Using the lingo of algorithmic logic [11], we express this formally as:*

$$(k = x) \Longrightarrow \{\mathsf{F}\,()\}(k = x) \tag{1}$$

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 Ψ is a bounded loop whose body is executed exactly N times (regardless of whether the if statement Γ succeeds or fails). Γ 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 the single statement ready. Thus, the purely "syntactic" bound on the possible number of invocations of ready (assuming that Γ succeeds on every possible occasion), before the procedure returns from its external call, is N^N. This implies that the external call of F always terminates.
◇

Let $z_{A,N}$ denote the current (calculated) number of zeroes in A. Recall that Δ denotes the body of procedure F (see Figure 1). From the viewpoint of formal analysis, an invocation of F can be replaced by the procedure's body Δ with the local variable i uniquely renamed [11]. Consider the value of $z_{A,N}$ at the beginning and at the end of Δ, i.e., at the invocation and at the return of F.

Lemma 4 *Before and after every execution of Δ, we have $z_{A,N} = N - k + 1$. In particular, the value of $z_{A,N}$ is preserved by every invocation of* F*. More formally:*

$$(z_{A,N} = N - k + 1) \Longrightarrow \{\Delta\}\,(z_{A,N} = N - k + 1) \tag{2}$$

Proof These are the two places (programs) within the procedure where the values of k and A are modified at all:

$$\Theta\text{:}\ \left|\begin{array}{l}\mathsf{A[\,i\,]:=k\,;}\\ \mathsf{k:=k+1;}\end{array}\right. \qquad \text{and} \qquad \Omega\text{:}\ \left|\begin{array}{l}\mathsf{k:=k-1;}\\ \mathsf{A[\,i\,]:=0\,;}\end{array}\right.$$

We shall show by induction that $z_{A,N} = N - k + 1$ 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 ($z_{A,N} = N$), because this is how the array has been initialized. So $z_{A,N} = N - k + 1$ holds before the very first execution of Θ. Suppose then that F has been called for some $k = q$, $1 \le q \le N$, and $z_{A,N} = N - q + 1$. Θ is only executed if $A[i] = 0$, and (by Corollary 1) k is never zero; thus we have:

$$(k = q \wedge z_{A,N} = N - q + 1) \Longrightarrow \{\Theta\}\,(k = q + 1 \wedge z_{A,N} = N - q) \qquad (3)$$

This is because Θ sets to nonzero one element of A that has been zero so far and increments the value of k by 1. So the equality is preserved by Θ. To prove it for Ω, we start from $k = N + 1$ and $z_{A,N} = 0$ (which has been shown above to hold after N iterations of Θ), i.e.,

$$(k = 1, z_{A,N} = N) \Longrightarrow \{\Theta\}^N\,(k = N + 1, z_{A,N} = 0) \qquad (4)$$

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

$$(k = q, z_{A,N} = N - q + 1) \Longrightarrow \{\Omega\}\,(k = q - 1, z_{A,N} = N - q + 2) \qquad (5)$$

which proves the equality for Ω all the way down to $k = 1$, thus proving the lemma.
◇

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* ready *is about to be called, i.e., the $k > N$ condition is satisfied, k is equal to $N + 1$, $z_A = 0$, and the array A (indices 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 *when the $k > N$ condition holds. The value of i_k gives the position of the value k in the permutation presented in A to* ready*. This follows from confronting the assignment to $A[i]$ in Θ with Lemma 4 and Corollary 2. Consequently, all permutations seen by* ready *are different.*

Lemma 5 *Consecutive invocations of* ready *see in A all the permutations of numbers between 1 and N.*

Proof Suppose otherwise, i.e., some permutation never appears in A over all invocations of ready, 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 this very permutation is going to 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 Φ 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 if statement Γ 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 Θ 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 else part Ψ 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 if condition for Γ 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 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 [5,3] where the average 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 [9,5,3,10,1,4]. Then an algorithm is deemed loopless if its invocation to generate one (next) permutation has a constant average cost independent of N.

Let us start by analyzing the complexity of the original (basic) version of our algorithm, as presented in Section 2.

Lemma 6 *Let $U(N)$ denote the total number of invocations of* F *for a given N. $U(N)$ is asymptotically bounded from above by $eN!$, i.e.,*

$$U(n) < eN! \quad \textit{and} \quad \lim_{N\to\infty} \frac{U(N)}{N!} = e \tag{6}$$

Proof In the light of Lemma 4 from Section 3, F is called $N-k+1$ times at level k. Thus, ignoring the first (external) invocation we have:

$$\begin{aligned} U(N) &= N \times (1 + U(N-1)) \\ U(1) &= 1 \end{aligned} \tag{7}$$

which implies that:

$$U(N) = N! \times \sum_{i=0}^{N-1} \frac{1}{i!} \tag{8}$$

whence follows the lemma.
◇

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 Ψ. 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 issuing 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:

$$U_f = N! \times \sum_{i=1}^{N} \frac{1}{i!} < (e-1)N! \tag{9}$$

which can be deemed to capture the number of "essential" invocations of F needed to solve the problem.

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 Ψ iterates N times at every level $k \leq N$ formally requiring $O(N)$ operations at every invocation of the procedure 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 9) by N. This is because U_f counts all the invocations of F that cause the loop to be entered, and when entered the loop always iterates exactly N times. So we have:

$$U_l = U_f \times N < (e-1)NN! \tag{10}$$

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

One can 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 representation of the list. The new set of global declarations becomes:

```
var N,k:integer; A:array[1..N] of integer;
    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 unoccupied slots in A. The initialization/invocation sequence is replaced with this code:

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

We have renamed the new variant of the procedure as G. Note that the array A need not be initialized because its occupancy data is now maintained by X. The size of X is $N+1$. The role of $X[i]$, 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. As before, k is initialized to 1. The new procedure is listed in Figure 2.

```
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[b]:=X[d];
       k:=k+1;
       G();
       k:=k-1;
       X[b]:=d; b:=X[b];
     end while
   end if
end;
```

Fig. 2 The "loopless" variant

Theorem 7 *Procedure* G *listed in Figure 2 generates all permutations of values* $1, \ldots, N$ *in an asymptotically constant number of steps per permutation, i.e., its time complexity is bounded from above by:*

$$T(N) = cN! \tag{11}$$

Proof Let us understand the essence of the transformation of F (as introduced in Section 2) into G. Notice that initially the values in X describe the straightforward succession of indices 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 all 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, equivalent to zeroing the corresponding element of A in F.

This implies that G carries out the same series of nonzero insertions into A as its previous version, thus accomplishing the same feat, but the total number of instructions associated with every invocation of G is now constant. From Lemma 6 and Equation 9, the total number of invocations of G per permutation, ignoring the last (special) call, is asymptotically approaching $e - 1$. Thus, the total cost per permutation approaches $e - 1$ times the fixed cost of the sequence of statements enveloping the recursive call to G.
◇

In addition to reducing the cost, the transformation 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 put until the numbers $k+1, \ldots, N$ have been exhaustively permuted over all the remaining positions. 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 critical 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 some may see as a disadvantage. In most practical applications one would prefer to have a "loopless" algorithm [5,3] 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.

The day can be saved by resorting to coroutines and turning the procedure into a *producer* providing data to an external *consumer* (Figure 3). The requisite tools have been available in many programming languages and environments, beginning historically with Simula 67 [2], becoming a standard feature of contemporary platforms and available in several guises offering handy

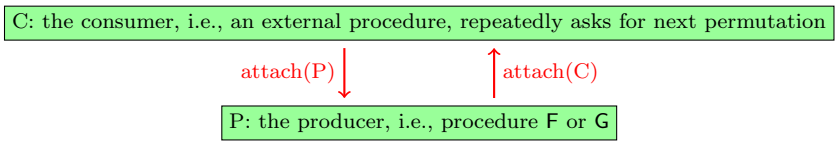

Fig. 3 The producer/consumer model

shortcuts for typical applications. For example, in Python, the procedure can be turned into a *generator* retaining its recursive form by applying a special type of the return statement known as yield [7]. The modification does not affect the practical efficiency of the implementation.

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. The algorithm has been used by us and some of our colleagues in a number of case studies and analyses illustrating the power and features of formal systems of inferring about programs. Seen from this angle, its interesting properties stem from the original inspiration: to devise a good educational example of recursion. For that one needs at least one local variable and at least one global variable (both of them essential), 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. Our discussions, involving students as well as experts, have raised these questions:

1. Why can the designer of a few-line program (devised 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 and find their way into the code, but in a manner that will make them transparent, so they can be seen and comprehended when the code is scrutinized?

The design of procedure F began 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." This sentence seems to explain everything there is to see about the algorithm, even

though it is inadequate as a proof of its correctness. It can also be viewed as the most straightforward plain-language specification of the problem and, at the same time, rather precisely reveals the programmer's intention. According to the paradigm of literate programming [8], it should thus be incorporated into the procedure's code and become its integral component. With a slightly more-informed look at procedure F, we see that it merely follows its simple specification to the letter. Thus, considering that its efficiency is not worse than that of the most refined solutions known in the area (some of which are considerably more difficult to explain), our algorithm should probably be viewed as the most natural solution to the problem.

References

1. van Baronaigien, D.R., Ruskey, F.: Generating permutations with given ups and downs. Discrete Applied Mathematics **36**(1), 57–65 (1992)
2. Dahl, O.J., Nygaard, K.: Simula. In: Encyclopedia of Computer Science, pp. 1576–1578. Wiley (2003)
3. Dershowitz, N.: A simplified loop-free algorithm for generating permutations. BIT Numerical Mathematics **15**(2), 158–164 (1975)
4. Effler, S., Ruskey, F.: A CAT algorithm for generating permutations with a fixed number of inversions. Information Processing Letters **86**(2), 107–112 (2003)
5. Ehrlich, G.: Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. Journal of the ACM **20**(3), 500–513 (1973)
6. Feferman, S.: Logics for termination and correctness of functional programs. In: Logic from Computer Science, pp. 95–127. Springer (1992)
7. Gorelick, M., Ozsvald, I.: High Performance Python: Practical Performant Programming for Humans. O'Reilly Media (2014)
8. Knuth, D.E.: Literate programming. The Computer Journal **27**(2), 97–111 (1984)
9. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming). Addison-Wesley Professional (2005)
10. Ko, C.W., Ruskey, F.: Generating permulations of a bag by interchanges. Information Processing Letters **41**(5), 263–269 (1992)
11. Mirkowska, G., Salwicki, A.: Algorithmic Logic. PWN, Warszawa (1987). URL http://lem12.uksw.edu.pl/wiki/Algorithmic_Logic
12. Wand, M.: Induction, Recursion and Programming. Elsevier Science Inc. (1980)