Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 979–982
DOI: 10.15439/2023F5568
ISSN 2300-5963 ACSIS, Vol. 35

A short note on computing permutations

Pawel Gburzynski
Department of Computer Engineering
Vistula University
ul. Stokłosy 3, 02-787 Warsaw, Poland
ORCID: 0000-0002-1844-6110

Janusz Zalewski
Department of Informatics
Ignacy Mościcki State Professional College
ul. Narutowicza 9, 06-400 Ciechanów, Poland
ORCID: 0000-0002-2823-0153

***Abstract*—We discuss an algorithm for generating all permutations of numbers between 1 and N. The algorithm is short and efficient, yet its behavior is not obvious from the code, mostly owing to the recursion. The discussion touches upon a few interesting methodological issues and brings in an educational case study in recursion.**

***Index Terms*—algorithms, recursion, permutations, algorithm analysis**

I. Introduction

NOVEL programs for generating permutations are not in big demand today, as the issue is deemed to have been "settled" by the opus of D. E. Knuth [1]. The algorithm we are about to present has been known to us since 1980, although it has never been published, except for a brief mention in [2]. It comes with a story which is best told in a narrative less formal than demanded by 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 a bit of 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 a 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 terminal and see what happens. That incident left us with a feeling that has persisted to this day, that some important questions deserve a thought.

We introduce the algorithm as a case study in algorithm design. First, the nature of recursion employed in it is nontrivial and educational, as it involves both categories of data (global, local) in a manner that makes them both relevant to the workings of the algorithm. The recursion is essential to the program's dynamics, e.g., in contrast to artificial examples where it can be trivially eliminated (like in calculating the Fibonacci function). While other recursive (and also optimal) algorithms for generating permutations are known [3], they (as most of their non-recursive relatives) assume element swapping as the basic operation. This way, the problem immediately receives an algebraic flavor and becomes that of transforming a given permutation into another permutation in such a way that all permutations are eventually mentioned in the transformation sequence. This is not necessarily the most natural expression of the problem from the viewpoint of a programmer. Our algorithm, in contrast, simply *generates* all permutations by making sure that all the elements are eventually *permuted*, i.e., each of them appears in all the possible slots, while giving all the other elements every possible chance to do the same.

Second, the basic variant of the algorithm, while being relatively easy to explain and instructional from the viewpoint of its correctness proof, is suboptimal from the viewpoint of performance. With some creativity, the algorithm can be improved such that its complexity matches the best (possible) solutions to the problem. The improved version appears more complicated (if introduced alone, it would have been considerably more difficult to analyze); however, owing to its descent from the basic variant, its analysis (both in term of correctness and performance) can be naturally carried over from the easier case. Then, we show how the algorithm can be transformed into a function that, instead of generating all the permutations in response to its zero-level call, can be invoked multiple times to yield consecutive permutations on individual demand. Overall, we believe it amounts to a case for beauty in programming, as per the views expressed in [4].

II. The basic algorithm

The algorithm has the form of a recursive procedure listed in Figure 1. It 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.

```
1  procedure F ( );
2      var i : integer;
3  begin
4      if k > N then
5          ready ( )
6      else
7          for i := 1 to N do
8              if A [i] = 0 then
9                  A [i] := k; k := k + 1;
10                 F ( );
11                 k := k – 1; A [i] := 0;
12             end if
13         end for
14     end if
15 end;
```

Fig. 1. The basic variant

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 in Figure 1) *generates* all permutations, e.g., as opposed 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.

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 remaining topic to be addressed was recursion. The problem had to be stated as briefly as possible, 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 invocation. In this respect, the two global variables k and A (N can be treated as 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 transparent.

III. Correctness

Formal correctness proofs of our algorithm have been the topic of several studies in Algorithmic Logic [6]. Here, for the sake of brevity, we shall confine ourselves to informal arguments. Our goal is to convince the reader that the algorithm terminates and in fact generates all permutations of the numbers from 1 to N, with each permutation appearing exactly once.

The following snippet 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 contains 1.

The global variable k is only modified in lines 9 and 11. It is incremented just before the internal (recursive) invocation of $F()$ and brought back to the previous value when the procedure returns. As it starts from 1, before the first (outer) call to $F()$ is made, it can be viewed as the counter of the recursive levels going from 1 until $N + 1$. Note that the last level ($N + 1$) is special: the function uses it to present its result, which action is represented by the call to *ready()*. In lines $7 - 13$, the procedure executes a loop going through all elements of A. Those elements that contain nonzero values are skipped, and the same value of k is consecutively inserted into the remaining positions in the array. Then, for every possible insertion of k, the procedure is called recursively with k incremented by one. Everything is undone when the recursive invocation of $F()$ returns. Looking globally, we see that the procedure inserts 1 in all places in A (initially, when $k = 1$, all of them are empty), then, for every configuration from the previous level, inserts 2 into all the remaining positions, and so on, all the way until N. This is how the problem of generating all permutations of the numbers from 1 to N is in fact defined. The procedure just literally fulfills this prescription; thus, in a sense, it can be viewed as the most natural (naive) solution to the problem.

IV. The time cost

Is our naive solution practical? To answer this question, we should gauge it against the best known algorithms for generating permutations which are known as loopless (or loop-free) ones [5], [7]. This term is somewhat unfortunate (no algorithm generating permutations can be truly loop-free) and refers to the constant average cost per permutation. In other words, the cost of generating all $N!$ permutations should be bounded by $c \times N!$ where c is some constant. Besides, a useful procedure should be able to generate *a* permutation when asked for it, i.e., one permutation at a time [1], [5], [7]–[10], as opposed to producing them all in response to a single invocation.

Let us start by calculating how many times $F()$ is called to produce all $N!$ permutations. We shall ignore the last-level call (for $k = N + 1$) because its is special; its sole purpose is to present a ready permutation available in A. Denoting the

number of (nontrivial) calls of the procedure by $U(N)$, we have:

$$\begin{array}{lcl} U(N) & = & N \times (1 + U(N-1)) \\ U(1) & = & 1 \end{array} \quad (1)$$

One can easily show by induction that:

$$U(N) = N! \times \sum_{i=1}^{N} \frac{1}{i!} \quad (2)$$

which means that:

$$U(n) < (e-1)N! \quad \text{and} \quad \lim_{N\to\infty} \frac{U(N)}{N!} = e \quad (3)$$

V. The asymptotically optimal variant

The number of invocations of *F()* needed to solve the problem for a given N is of order $N!$ with the factor $c < e-1$. If we could prevent the *for* loop from iterating over the nonzero entries in A, which simply have to be skipped and ignored, and make it proceed directly to the next free entry on every turn, we would bring the complexity of our algorithm down to $O(N!)$. To accomplish that, in addition to the original array A, we introduce another array acting as a representation of the list of free entries in A available at the current level. The new set of global declarations becomes this:

```
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] := 0;
...
G ( );
...
```

The new variant of the procedure shown in Figure 2 is named $G()$. We claim that it 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:

$$T(N) = cN! \quad (4)$$

To see this 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. 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, but the total number of instructions associated with every invocation of $G()$ is now constant.

```
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

VI. One permutation at a time

The algorithm is inherently recursive which a practical programmer may see as a disadvantage. One would prefer a function that could be called from an external program each time a new permutation is needed. [5], [7]. Of course, as any recursive procedure, the algorithm can be reprogrammed in a non-recursive manner, but one can argue that the recursive form is its essential feature.

Modern programming languages and environments offer tools which make the adaptation of our algorithm to a practical usage natural and easy while retaining its essentially recursive form. These tools, under the name of coroutines, originated historically with Simula 67 [11], becoming useful features of many contemporary platforms and being available in several guises offering handy shortcuts for typical applications.

Figure 3 presents a modern-flavor coroutine-like variant of our algorithm implemented in Python. The implementation consists of a Python function *perm()*, providing the actual callable generator, and its helper function *advance()* taking care of the recursive part. The semantics of the *yield* operation [12] consist in suspending the execution of the current function and returning to its caller in such a way that on a subsequent invocation of the same function its execution will continue from the point of the last interruption. The operation *yield from* f_2 carried out by a function f_1 invokes the specified function f_2 and, when that function returns via *yield*, carries

```
def advance ( ):
    global A, X, N, k;
    if k > N :
        yield A
    else:
        b = 0;
        while X [b] <= N :
            d = X [b]; A [d] = k; X [b] = X [d]
            k = k + 1;
            yield from advance ( )
            k = k – 1;
            X [b] = d; b = X [b]

def perm (n):
    global A, X, N, k
    A = { }; X = { }; N = n
    for i in range (N+1):
        X [i] = i + 1
    k = 1
    yield from advance ( )
```

Fig. 3. A coroutine-style implementation in Python

over the effect to its original caller. Consequently, when the original caller calls f_1 again, it will continue within f_2 from the place where f_2 last yielded.

We can easily see that *advance()* is basically a straightforward rewrite of *G()* in Python, except that: 1) the function yields with the value of array A whenever *G()* would produce a complete new permutation; 2) the yield has to be carried over recursively, so the recursive call is appropriately replaced with a *yield from*.

The following code illustrates the usage of the generator:

```
...
while 1:
    n = int (input ("Enter n: "))
    for p in perm (n):
        print (p)
...
```

In a serious project, the generator would be encapsulated into a structure isolating the namespace of its global variables.

VII. 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. Our discussions, involving students as well as experts, have raised these questions:

1) Why can the designer of a few-line program (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, so the right and correct ideas can materialize and find their way into the code 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: "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. It can also be viewed as the most straightforward plain-language specification of the problem and, at the same time, rather precisely explains the programmer's intention. According to the paradigm of literate programming [13], it should thus be incorporated into the procedure's code and become its integral component. Viewed in this light, $F()$ merely follows its simple specification to the letter. Considering that its efficiency is not worse than that of the most refined solutions known in the area, our algorithm should probably be viewed as the most natural solution to the problem of generating all permutations.

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming)*. Addison-Wesley Professional, 2005.

[2] L. Banachowski, A. Kreczmar, and W. Rytter, *Analysis of Algorithms and Data Structures*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[3] P. Flajolet and R. Sedgewick, *Analytic Combinatorics*. Cambridge University Press, 2009.

[4] A. Oram and G. Wilson, *Beautiful code: Leading programmers explain how they think*. O'Reilly Media, Inc, 2007.

[5] G. Ehrlich, "Loopless algorithms for generating permutations, combinations, and other combinatorial configurations," *Journal of the ACM*, vol. 20, no. 3, pp. 500–513, 1973.

[6] G. Mirkowska and A. Salwicki, *Algorithmic Logic*. PWN, Warszawa, 1987. [Online]. Available: http://lem12.uksw.edu.pl/wiki/Algorithmic_Logic

[7] N. Dershowitz, "A simplified loop-free algorithm for generating permutations," *BIT Numerical Mathematics*, vol. 15, no. 2, pp. 158–164, 1975.

[8] C. W. Ko and F. Ruskey, "Generating permulations of a bag by interchanges," *Information Processing Letters*, vol. 41, no. 5, pp. 263–269, 1992.

[9] D. R. van Baronaigien and F. Ruskey, "Generating permutations with given ups and downs," *Discrete Applied Mathematics*, vol. 36, no. 1, pp. 57–65, 1992.

[10] S. Effler and F. Ruskey, "A CAT algorithm for generating permutations with a fixed number of inversions," *Information Processing Letters*, vol. 86, no. 2, pp. 107–112, 2003.

[11] O.-J. Dahl and K. Nygaard, "Simula," in *Encyclopedia of Computer Science*. Wiley, 2003, pp. 1576–1578.

[12] D. Beazley and B. K. Jones, *Python cookbook: Recipes for mastering Python 3*. "O'Reilly Media, Inc.", 2013.

[13] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.