A Formal Model and an Algorithm for Generating the Permutations of a Multiset

VINCENZO DE FLORIO and GEERT DECONINCK
Electrical Engineering Department – ESAT
Katholieke Universiteit Leuven,
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee
BELGIUM

Abstract: – This paper may be considered as a mathematical divertissement as well as a didactical tool for undergraduate students in a universitary course on algorithms and computation. The well-known problem of generating the permutations of a multiset of marks is considered. We define a formal model and an abstract machine (an extended Turing machine). Then we write an algorithm to compute on that machine the successor of a given permutation in the lexicographically ordered set of permutations of a multiset. Within the model we analyze the algorithm, prove its correctness, and show that the algorithm solves the above problem. Then we describe a slight modification of the algorithm and we analyze in which cases it may result in an improvement of execution times.

Key-Words: – Computational combinatorics, permutations, Turing machines, algorithms, number theory.

1 Introduction

The problem of generating the permutations of a multiset of marks is considered. First some mathematical entities and properties are defined within a formal model. Then an abstract machine is introduced and an algorithm is shown which computes those entities on the abstract machine. Within this framework the algorithm is analyzed and proved to be correct. Finally it is shown that the algorithm computes the successor of a given permutation in the ordered set of lexicographically ordered permutations.
The structure of this work is as follows: the formal model, the machine and the algorithm are described in Section 2. Section 3 deals with the link between the algorithm and the process of generating the permutations of a multiset in lexicographical order. Some conclusions are drawn in Section 4.

2 A machine and an algorithm

2.1 Prologue—some definitions

Let n be an integer, $n > 2$, and let $A_n = \{a_0, a_1, \ldots, a_{n-1}\}$ be a set of "marks". Let I_n be set $\{0, 1, \ldots, n-1\}$. Let furthermore m be an integer, $0 < m \leq n$.

Definition 1 *Set $A = \{a_0, a_1, \ldots, a_{m-1}\}$ be called "the Alphabet".*

Let $I_m = \{0, 1, \ldots, m-1\}$. Let us consider multiset

$$M = \{\underbrace{a_0, \ldots, a_0}_{c_0}, \underbrace{a_1, \ldots, a_1}_{c_1}, \cdots, \underbrace{a_{m-1}, \ldots, a_{m-1}}_{c_{m-1}}\}$$

such that $\sum_{i \in I_m} c_i = n$ and $\forall i \in I_m : c_i > 0$.

Definition 2 *Any ordered arrangement of the elements of a multiset M be called* M-permutation *and represented as p^M or, where M is evident from the context, as p. Let $p^M[i]$ (or $p[i]$) denote the $(i+1)^{\text{th}}$ element of p. Let us call $\mathcal{P}^M$ (or simply $\mathcal{P}$ when unambiguous) the set of the M-permutations.*

Let $o : A_n \to I_n$ be a bijection defined so that $\forall i \in I_n : o(a_i) = i$. Clearly o induces a total order on the marks (and, *a fortiori*, on the elements of the Alphabet). Furthermore, through o, any M-permutation may be interpreted as an n-digit, base-m number [1].

Definition 3 *Given multiset M as above, arrangement*

$$\underbrace{a_0 \cdot \cdots \cdot a_0}_{c_0} \cdot \underbrace{a_1 \cdot \cdots \cdot a_1}_{c_1} \cdots \underbrace{a_{m-1} \cdot \cdots \cdot a_{m-1}}_{c_{m-1}}$$

is called the zero permutation *of M, or briefly its* zero, *and is denoted as p_0^M or simply as p_0 when this can be done unambiguously. For the sake of brevity, operator "$\cdot$", concatenating any two marks, will be omitted in the following.*

Definition 4 *Given multiset M as above, arrangement*

$$\underbrace{a_{m-1} \ldots a_{m-1}}_{c_{m-1}} \underbrace{a_{m-2} \ldots a_{m-2}}_{c_{m-2}} \cdots \underbrace{a_0 \ldots a_0}_{c_0}$$

is called the last permutation *of M and is denoted as p_∞^M.*

Definition 5 *Given any two M-permutations, p_1 and p_2:*

- *p_1 is said to be* equal *to p_2 iff $\forall i \in I_n : p_1[i] = p_2[i]$. Let $p_1 = p_2$ represent this property.*

- *p_1 and p_2 are said to be* different *iff $\neg(p_1 = p_2)$, or $p_1 \neq p_2$ for brevity.*

- *$p_1 < p_2$ iff $\exists k \in I_n \ni' (\forall j < k : p_1[j] = p_2[j]) \wedge (p_1[k] < p_2[k])$.*

Definition 6 *Let $p \in \mathcal{P}^M$. An* inversion *is a couple of contiguous marks of p, $p[i]$ and $p[i+1]$, $i \in \{0, \ldots, n-2\}$, such that $p[i] < p[i+1]$. If no such couple can be found in p, then p is said to be* inversion-free *(INF). If at least one such couple does exist, then p is said to be* non-inversion-free *(NIF).*

Theorem 1 *For any M, p_∞^M is the only possible INF-permutation.*

PROOF By construction, p_∞^M is INF. *Ab absurdo,* let us suppose there exists a permutation $p \neq p_\infty^M$ which is INF. Then either $p < p_\infty^M$ or vice-versa. In both cases we reach a contradiction by Def. 5. . ■

2.2 A machine

Let us consider $\mathcal{M}$, a generalized Turing machine [6] with two tapes, T_1 and T_2, and three heads, H_{1l}, H_{1r}, and H_2. Heads H_{1l} and H_{1r} are "jointed", *i.e.*, capable of reading or writing any two consecutive squares of T_1 at a time. H_2 operates on T_2. Let us call S_1 and S_2 resp. the set of squares of T_1 and the set of squares of T_2.

Definition 7 *Be s a square; square s' then represents the square which immediately follows s. For any $t \in \mathbb{N}$ let $s\overbrace{''\cdots'}^{t}$ represent square $\overbrace{((\ldots(}^{t} s \overbrace{')' \ldots)'}^{t}$.*

Definition 8 *Let $z_1 \in S_1$. Bijection $\pi_1 : S_1 \to \mathbb{Z}$, such that*

$$\forall s \in S_1 : \begin{cases} \pi_1(s) = 0 & \text{if } s = z_1, \\ \pi_1(s) = t & \text{if } s \neq z_1 \text{ and } \exists t \in \mathbb{N} \ni' \\ & (s = z_1 \overbrace{''\cdots'}^{t}), \\ \pi_1(s) = -t & \text{if } s \neq z_1 \text{ and } \exists t \in \mathbb{N} \ni' \\ & (z_1 = s \overbrace{''\cdots'}^{t}), \end{cases}$$

is called the relative distance from z_1 in S_1. *Following the same procedure, let $\pi_2 : S_2 \to \mathbb{Z}$ be the relative distance from a fixed $z_2 \in S_2$.*

From Def. 2, any M-permutation p can be coded on Tape T_1 of $\mathcal{M}$ by writing, $\forall i \in I_n$, mark $p[i]$ onto square $\pi_1^{-1}(i)$, *i.e.*, onto square $z_1 \overbrace{''\cdots'}^{i}$.

Machine $\mathcal{M}$ can execute read, write and head-movement actions. These basic actions can be described via the following symbolic notation:

Let i be an integer and H_x and H_y be heads respectively operating on Tape T_x and Tape T_y (possibly the same). Let s_x be the square under Head H_x and s_y the square under H_y. Let us call π_x and π_y the relative distances respectively for T_x and T_y (possibly equal). Let r be a function converting a square into the mark currently coded onto that square. Then let:

(H_x) represent the output of a read of square s_x, available as $o(r(s_x)) \in I_m$, and let

H_x represent the position of Head H_x, available as $\pi_x(s_x) \in \mathbb{Z}$.

The infix, "overloaded" (in the sense of [5]) operator "←" be used to represent both writings of squares and absolute movements of heads as described below:

$(H_x) \leftarrow i$ writes in s_x mark $o^{-1}(i \bmod n)$,

$(H_x) \leftarrow (H_y)$ overwrites s_x with the mark written in s_y,

$(H_x) \leftarrow H_y$ overwrites s_x with mark $o^{-1}(\pi_y(s_y) \bmod n)$,

$H_x \leftarrow i$ moves Head H_x over square $\pi_x^{-1}(i)$,

$H_x \leftarrow (H_y)$ moves Head H_x over square $\pi_x^{-1}(o(r(s_y)))$,

$H_x \leftarrow H_y$ moves Head H_x over square $\pi_x^{-1}(\pi_y(s_y))$.

Postfix operators "++" and "--", borrowed from the grammar of the C programming language [2], are used to represent incremental / decremental writings and relative, single-square movements as follows:

(H_x)++ increments, modulo n, the number coded in the square under Head H_x, *i.e.*, $\forall i \in \{0, \ldots, n-2\}$, mark a_i is promoted to mark a_{i+1}, and mark a_{n-1} is demoted to mark a_0.

(H_x)-- decrements, modulo n, the number coded in the square under Head H_x, *i.e.*, $\forall i \in \{1, \ldots, n-1\}$, mark a_i is demoted to mark a_{i-1}, and mark a_0 is promoted to mark a_{n-1}.

H_x++ moves Head H_x on the square immediately following s_x, $\pi_x^{-1}(\pi_x(s_x)+1)$.

H_x-- moves Head H_x on the square immediately preceding s_x, $\pi_x^{-1}(\pi_x(s_x)-1)$.

The instruction set of $\mathcal{M}$ includes **if** and **while** statements so to modify the flow of control depending on arithmetical and logical expressions. The bracket statements **do** and **od** can be used for grouping the argument of **if** and **while** statements as well as a whole instruction table, which is declared via the **proc** statement. Keyword **call** invokes an instruction table. Control returns to the caller as soon as the callee executes instruction **return**. Let us furthermore suppose there exists a symbolic assembler such that it is possible to define symbolic constants by means of the pseudo-instruction DEFINE. Comments are also supported—string "//" denotes the beginning of a comment which continues up to the end of the line.
Finally, let us suppose that each of the above instructions, including each evaluation of a Boolean or arithmetical atom, lasts the same amount of time.

2.3 An algorithm for machine $\mathcal{M}$

By means of the above just sketched agreements it is possible to compose the following computational procedure:

proc SUCC
DEFINE NIF $:= 0$ // p^M *has found to be NIF.*
DEFINE INF $:= 1$ // p^M *has found to be INF.*
do

// *Phase 1: Heads H_{1l} and H_{1r} are moved on the*
// *last couple of squares of the encoding of p^M onto Tape 1.*
// *H_{1r} is on the last such square, H_{1l} on the last but one.*

$H_{1r} \leftarrow n-1$

while $(H_{1l}) \geq (H_{1r}) \ \wedge \ H_{1l} \geq 0$
do
H_{1r}--
$H_2 \leftarrow (H_{1r})$
(H_2)++
od

// *Phase 3: Case $p^M = p^M_\infty$.*

if $H_{1l} < 0$
do
$H_{1r} \leftarrow n$ // *Moves H_{1r} beyond coding of p^M…*
$(H_{1r}) \leftarrow$ INF // *…stores the INF flag…*
return // *…and stops.*
od

// *Phase 4: Case $p^M \neq p^M_\infty$:*
// *an inversion,* i.e., $(H_{1l}) < (H_{1r})$ *has been found.*

$H_2 \leftarrow (H_{1l})$

(H_2)++ // (H_{1l}) *is accounted on Tape 2.*

// *Phase 5: Looks on Tape 2 for the least previously*
// *recorded digit which is greater than* (H_{1l}).

```
H_2++
while (H_2) = 0    // While there are empty entries,
do // skip them.
    H_2++
od
```

// *Phase 6: Swaps* (H_{1l}) *and* (H_2).

```
(H_1l) ← H_2
(H_2)--
H_1l++
```

// *Phase 7: Piles up the digits as recorded on Tape 2.*

```
H_2 ← 0                    // Rewinds Head H_2.
while H_1l < n
do
    if (H_2) ≠ 0
        do
            (H_1l) ← H_2
            H_1l++
            (H_2)--
        od
    else
        do
            H_2++
        od
od
```

// *Phase 8: Writes the exit status and returns.*

```
                // H_1l is already on o^{-1}(n).
    (H_1l) ← NIF // Stores the NIF flag...
    return      // ... and stops.
od.
```

A number of properties of this computational procedure are now to be unraveled.

Theorem 2 *Let us suppose that a* p^M *is coded on Tape 1 of* $\mathcal{M}$ *and an* n*-digit zero is coded onto Tape 2. Then procedure* SUCC *always terminates, and it does in linear time at most.*

PROOF We need to show that Phases 2, 5, and 7 stop in linear time. Phase 2 stops either

1. when $(H_{1l}) < (H_{1r})$, *i.e.*, in the presence of an inversion, or

2. when Head H_{1l} goes out of the "left" boundary of the coding of p.

Both heads of Tape 1 are shifted leftward of one square by executing action H_{1r}--, so Condition 2 is going to be met after at most n cycles. No other movement in Tape 1 is commanded within that cycle. The two other actions in that cycle count on Tape 2 the occurrences of visited marks of Tape 1.

Obviously Condition 1 implies that p is NIF. The heads of Tape 1 lay onto the inversion when Phase 2 is exited in this case. Condition 2 implies that p is INF, *i.e.*, $p = p^M_\infty$. Phase 3 takes care of this possibility—the nature of p is recorded on $\pi_1^{-1}(n)$ and the procedure stops.

If Phase 4 is being executed then we are in the case of Condition 1. This Phase accounts (H_{1l}) as well on Tape 2. When Phase 4 terminates, H_2 lays onto square $s = \pi_2^{-1}(o(\,(H_{1l})\,))$. As p is NIF, and as $((H_{1l}), (H_{1r}))$ represents an inversion, there is at least one square of Tape 2, say t, such that both the following conditions hold:

1. t is a successor of H_2,

2. t holds a mark m such that $m > (H_{1l})$.

This proves that Phase 5 terminates after at most $n-1$ iterations.

At this point Head H_2 lays on the first non-zero square on the "right" of square s. As such, it represents the least mark accounted on T_2 which is greater than (H_{1l}). Phase 6 overwrites such mark onto square (H_{1l}), "de-accounts" it from T_2, and shifts the heads of T_1 one square to the "right".

Phase 7 first rewinds Head H_2. The loop then moves H_2 through Tape 2 looking for non-zero squares. For any such square, the corresponding relative distance, cast to a mark, overwrites the square Head H_{1l} lays onto. On each of these overwritings H_{1l} is shifted and square H_2 is decremented. This goes on while H_{1l} lays on the encoding squares. Head H_2 is only moved when (H_2) holds $a_0 = o^{-1}(0)$. This Phase then writes on Tape 1 all and only the marks visited during the hunt for an inversion, *i.e.*, Tape 1 contains an arrangement, other than the original one, of the same multiset M: another M-permutation. Furthermore the maximum duration of this Phase is $n-2$ iterations, for at most $n-2$ marks differ from a_0. ■

After Theorem 2 computational procedure SUCC is found to be an Algorithm [3]. Note also that when SUCC stops, Tape 2's encoding is restored to its original value—n-digit number zero.

Theorem 2 allows to show that, if $p \neq p^M_\infty$, then it is always possible to factorize p as follows:

Theorem 3 (LaR Factorization) *If $p^M \neq p^M_\infty$ then $\exists L \subset M, R \subset M, a_i \in M$, such that:*

1. $p^M = p^L a_i \, p^R_\infty$.
2. $R \neq \emptyset \Rightarrow a_i < \max\{a_j \mid a_j \in R\}$.

PROOF Let us represent p^M on the tape of a Turing machine and instruct the machine so that it scans the permutation right-to-left, halting at the first couple of contiguous symbols which is *not* an inversion, or at the left of its leftmost character—this is possible due to Theorem 2. The Head at first stands onto the rightmost character of p^M. At the end of processing time the Head of the machine might

- have moved one position leftward. In this case, take $R = \emptyset$, a_i the rightmost symbol of p^M, and $L = \mathcal{C}_M\{a_i\}$ (*i.e.*, the complementary set of $\{a_i\}$ with respect to M).
- be laying somewhere else within the permutation *i.e.*, the Head's total number of shifts were more than 1 and less than n. In this case, let a_i be the symbol the Head stands on; then let L and R be the two substrings respectively on the left and on the right of a_i (L may also be empty).

It will not be possible to find the Head on the left of the leftmost character of the permutation, because this would mean that no inversion had been found. In this case p^M would be equal p^M_∞, contradicting the hypothesis. ■

Definition 9 *Let p be a permutation of a multiset M. Then there are two distinct cases:*

$p \neq p^M_\infty$: *In this case let us consider p's LaR factorization, $p^L a_i \, p^R_\infty$ for some L, a_i, and R. Be $k = \min_R\{j \mid a_j < a_i\}$, and $\overline{R} = \{a_i\} \cup \mathcal{C}_R\{a_k\}$; then*

$$p' = p^L a_k \, p^{\overline{R}}_0$$

is defined as the successor permutation *for permutation p.*

$p = p^M_\infty$: *In this case we say that p' is undefined, or that $p' = \Lambda$.*

Note $p' = \text{SUCC}(p)$, *i.e.*, Algorithm SUCC computes the just defined successor of permutation p: if at the end of computing time square $\pi_1^{-1}(n)$ holds INF, then $p' = \Lambda$, otherwise the coding of p' can be found in the coding squares of Tape 1.

Theorem 4 *Let $p \in \mathcal{P}^M, p \neq p^M_\infty$. Then the following two conditions hold:*

1. $p < p'$.
2. $\nexists q \in \mathcal{P}^M \ni' \; p < q < p'$.

PROOF Condition *1* follows directly from Theorem 3 and Def. 9. Condition *2* follows by observing that p and p' share the same left substring p^L and start differing on the mark directly following that substring. Let us call a and b these characters resp. in p and p'. By construction, b is the least available character that is greater than a. ■

3 The nature of Algorithm SUCC

Definition 10 *Let p and q be any two permutations of a given multiset M. Then p is said to precede q by Algorithm SUCC iff:*

$$\exists z \in \mathbb{N}^\star \ni' p^{\overbrace{\prime\prime\cdots\prime}^{z}} = q.$$

Let us denote this property as $p \prec q$.

Let us consider the following Algorithm:

```
proc ITER
DEFINE NIF := 0, INF := 1
do
    H_1l ← n
    (H_1l) ← NIF
    while (H_1l) ≠ INF
    do
        call SUCC          // Invokes Algorithm SUCC.
        H_1l ← n
    od
    return
od.
```

Obviously Algorithm ITER takes as input an M-permutation and produces all successors of that permutation, until the last permutation is reached.

Definition 11 *Let p_0 and p_∞ be the zero and last permutations of a given multiset M. Via Algorithm* ITER *it is possible to consider set $P = \{p_0, p_0', p_0'', \ldots, p_\infty\}$, the set of all outputs of that Algorithm. Note that P is linearly ordered by the relation "$\prec$". Let us call P* the output set.

It is now possible to show that the process of determining a zero permutation and then generating all permutations, successor by successor, by means of Algorithms SUCC and ITER, is equivalent to the process of generating in lexicographical order all permutations of a string with multiple occurrences of the same characters in it [4]:

Theorem 5 *Given a multiset M, let $P(n)$ be the number of different arrangements that can be observed starting from p_0 and going up to p_∞ recursively applying the successor operator (computable via algorithm* SUCC*)* i.e.,

$$P(n) = z + 1 \quad \Leftrightarrow \quad p_0 \overbrace{''\cdots'}^{z} = p_\infty.$$

Then $P(n) = \left(c_0, c_1, \overset{n}{\ldots}, c_{m-1}\right).$

PROOF The proof follows by induction over n. It is left as an exercise to the reader. ■

Theorem 5 is the formal proof that algorithm SUCC does generate each and every permutation of a multiset. In other words, the output set coincides with set $\mathcal{P}$ and Algorithm ITER computes the generating process of $\mathcal{P}$. Recursive relation $p' = \text{SUCC}(p)$ is a concise representation of such process. Furthermore, relation "$\prec$" coincides with relation "$<$".

4 Conclusions

We formally showed that Algorithm SUCC computes the successor of a permutation of a multiset M, and that the process which develops permutations via successive calls of SUCC generates each and every permutation of M in lexicographical order.

Acknowledgement This project is partly supported by the IST-2000-25434 Project "DepAuDE". Geert Deconinck is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (FWO).

References

[1] George E. Andrews, *Number Theory,* W. B. Saunders Co., Philadelphia, 1971.

[2] Brian W. Kernighan and Dennis M. Ritchie, "*The C Programming Language,*" 2nd edition, Prentice-Hall, Englewood Cliffs, N.J., 1988.

[3] Donald E. Knuth, *The Art of Computer Programming,* Vol.1 ("Fundamental Algorithms"), 2nd edition, (Addison-Wesley, Reading, MA, 1973.

[4] E. S. Page and L. B. Wilson, *An Introduction to Computational Combinatorics,* Cambridge University Press, Cambridge, 1979.

[5] Bjarne Stroustrup, "*The C++ Programming Language,*" 2nd edition, Addison-Wesley, Reading, MA, 1995.

[6] Alan M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," Proc. London Math. Soc. Vol.42, 1936, pp. 230–265.