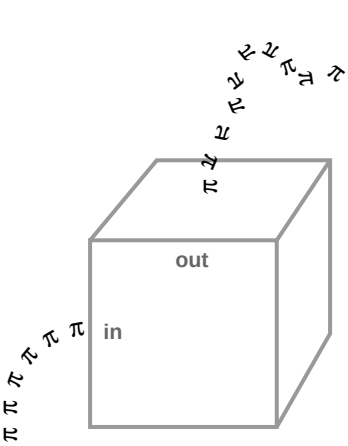

Computing permutation encodings

K. Rustan M. Leino[0]

27 May 1994

Computer Science, Mail code 256-80, California Institute of Technology, Pasadena, CA 91125, U.S.A.

rustan@cs.caltech.edu

Abstract: We consider some encodings of permutations of the first N natural numbers, discuss some relations among them and how one can be computed from others. We show a short proof of an existing efficient algorithm for one encoding, and present two new efficient algorithms for encoding permutations. One of these algorithms is constructed as the inverse of an existing algorithm for decoding, making it the first efficient permutation encoding algorithm obtained that way.

0 Overview

0.0 Permutation encodings

Let A be a permutation of the first N natural numbers. For any i such that $0 \le i < N$, we let $A.i$ denote element i of the permutation. As the elements of A are unique, we have, for all i, j such that $0 \le i \le j < N$,

$$(A.i = A.j) \;=\; (i = j) \qquad . \tag{0}$$

For the rest of this section, i will implicitly be universally quantified over $0 \le i < N$.

Consider an encoding B of A, defined as

$$B.i \;=\; \langle\, \#\, j \mid 0 \le j < i \;\triangleright\; A.j < A.i \,\rangle \qquad . \tag{1}$$

The right-hand side is a quantified expression, where $\mid$ and $\triangleright$ separate the dummy, the range of the dummy, and the term of the expression. The formula expresses that $B.i$ equals the number of times $A.j < A.i$ is true as j ranges over $0 \le j < i$. Thinking of the $A.i$'s as being arranged from left to right, (1) states that $B.i$ is the number of values smaller than $A.i$ to the left of $A.i$ in the permutation, that is, among the leftmost i numbers in the permutation.

In [1] and [2], B is called the *code* of A. A variation C of B is

$$C.i \;=\; \langle\, \#\, j \mid 0 \le j < i \;\triangleright\; A.j > A.i \,\rangle \qquad . \tag{2}$$

Due to (0), a relationship between B and C is

$$B.i + C.i \;=\; i \qquad , \tag{3}$$

from which we see that one can compute B from C or C from B in time $\mathrm{O}(N)$.

[0] Supported in part by a research grant from the Digital Equipment Corporation Systems Research Center.

Two other encodings of A are D and E, defined as

$$D.i \;=\; \langle\, \#j,k \mid A.k = i \wedge 0 \le j < k \;\triangleright\; A.j < A.k \,\rangle \quad \text{, and} \tag{4}$$

$$E.i \;=\; \langle\, \#j,k \mid A.k = i \wedge 0 \le j < k \;\triangleright\; A.j > A.k \,\rangle \quad . \tag{5}$$

Stated in words, $D.i$ is the number of times $A.j < A.k$ is true as j and k range over

$$A.k = i \wedge 0 \le j < k \quad . \tag{6}$$

Note that because A is a permutation, there is exactly one k satisfying (6). E is called the *inversion table* of A (*cf.* [4]).

By considering D applied to $A.i$ instead of i, we get

$$\begin{array}{ll} & D.(A.i) \\ = & \quad \{\ (4) \text{ — def. of } D \text{, with } i := A.i \ \} \\ & \langle\, \#j,k \mid A.k = A.i \ \wedge\ 0 \le j < k \;\triangleright\; A.j < A.k \,\rangle \\ = & \quad \{\ (0), \text{ with } i,j := k,i \ \} \\ & \langle\, \#j,k \mid k = i \ \wedge\ 0 \le j < k \;\triangleright\; A.j < A.k \,\rangle \\ = & \quad \{\ \text{one-point rule} \ \} \\ & \langle\, \#j \mid 0 \le j < i \;\triangleright\; A.j < A.i \,\rangle \\ = & \quad \{\ (1) \text{ — def. of } B \ \} \\ & B.i \qquad , \end{array} \tag{7}$$

and a similar calculation reveals

$$E.(A.i) \;=\; \langle\, \#j \mid 0 \le j < i \;\triangleright\; A.j > A.i \,\rangle \;=\; C.i \quad . \tag{8}$$

From (7) and (8), we conclude it is possible to, given A, compute B from D, C from E, D from B, or E from C in time $\mathrm{O}(N)$.

A relationship between A, D, and E, established by (7), (8), and (3), is

$$D.(A.i) + E.(A.i) \;=\; i \quad , \tag{9}$$

from which we can see that, given A, one can compute D from E or E from D in time $\mathrm{O}(N)$. Furthermore, (9) lets us infer that, given D and E, one can compute A in time $\mathrm{O}(N)$.

In Section 1, we describe a linear-time linear-space algorithm for computing A from B and D, and one for computing A from C and E.

To summarize, given one of B and C, one can compute the other in linear time, and given *two* of A, B, D, and E, one can compute any other in linear time.

0.1 Algorithms

In [1] and [2, Chapter 21], we find an *in situ* program that computes B from A in time $\mathrm{O}(N^2)$. This program has the nice property of being *invertible*. That is, without writing a new program from scratch, there is a way to run this program "backwards". The backward rendering of the program, which then can be written as a program in its own right, is called the *inverse* of the forward rendering. The inverse of the program in [1, 2] therefore gives an *in situ* algorithm for computing A from B in time $\mathrm{O}(N^2)$.

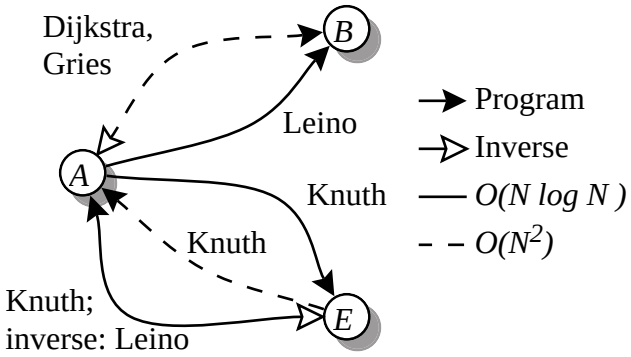

Figure 0: Algorithms for encoding and decoding permutations

We present a faster program for computing B from A in Section 2. This program runs in time $O(N \log N)$ using linear storage. Although invertible, the inverse does not compute A from B. The reason is that the forward program computes, in addition to B, two auxiliary arrays. So, the inverse program takes as input B and the auxiliary arrays. The problem is that the auxiliary arrays cannot efficiently be computed directly from B, so the inverse program is not of much use (see Section 2.4).

In [4, Section 5.1.1], Knuth shows a linear space, time $O(N \log N)$ algorithm for computing E from A (Ex. 6). We show a short proof of this algorithm in Section 3. Using this algorithm and our observed relations between A, B, C, and E, it is possible to compute B from A in time $O(N \log N)$ in the following way:

compute E from A in time $O(N \log N)$;
compute C from A and E in time $O(N)$;
compute B from C in time $O(N)$.

This method is asymptotically as good as, but differs from, that presented in Section 2.

For the other direction, *i.e.* computing A from E, Knuth not only gives an $O(N^2)$ algorithm (Ex. 4), but also a linear space, time $O(N \log N)$ algorithm (Ex. 5).

Although Knuth shows one efficient algorithm that computes E from A, and one that computes A from E, these algorithms are quite different. Regarding the inverse of his programs, Knuth mentions nothing. It is hard to see how the former of these algorithms can be inverted. The latter, however, we can invert, as will be our concern in Section 4.

Figure 0 summarizes the individual algorithms to encode and decode permutations, and their running times. Each $O(N \log N)$ algorithm uses linear space.

0.2 Outline of paper

The rest of the paper is organized in the following way. Section 1 demonstrates that A can be obtained in linear time from B and D, or from C and E. Section 2 develops a new and efficient algorithm for computing the code, that is, computing B from A. Section 3 contains a short proof of an existing

efficient algorithm for computing the inversion table of a permutation. An efficient algorithm, due to Knuth, for decoding an inversion table is presented in Section 4. It is then shown that this algorithm can be inverted, yielding a new efficient algorithm for computing the inversion table of a permutation. Finally, Section 5 offers some concluding remarks.

1 Decoding a permutation from two of its encodings

In this section, we show two algorithms: one computes A from B and D, the other computes A from C and E.

We recall from (7) that $B.i = D.(A.i)$ for any i. Consequently, we have, for any m,

$$\{\, i \mid B.i = m \triangleright\ i \,\} = \{\, i \mid D.(A.i) = m \triangleright\ i \,\} \quad . \tag{10}$$

This equation exhibits two set constructors, defined as follows. Let $f.i$ be any function of i, and $R.i$ a boolean function of i. Then, $\{\, i \mid R.i \triangleright\ f.i \,\}$ is the set of all $f.i$'s obtained when i ranges over the values prescribed by $R.i$. So, (10) states that the set of i's for which $B.i = m$, equals the set of i's for which $D.(A.i) = m$.

We may rewrite equation (10) as follows.

$$\begin{array}{ll} & \{\, i \mid B.i = m \triangleright\ i \,\} \ = \ \{\, i \mid D.(A.i) = m \triangleright\ i \,\} \qquad (11) \\ = & \quad \{\ \text{apply } A \text{ to the terms } \} \\ & \{\, i \mid B.i = m \triangleright\ A.i \,\} \ = \ \{\, i \mid D.(A.i) = m \triangleright\ A.i \,\} \\ = & \quad \{\ \text{rename dummy } i := A.i \text{, since } A \text{ has an inverse } \} \\ & \{\, i \mid B.i = m \triangleright\ A.i \,\} \ = \ \{\, i \mid D.i = m \triangleright\ i \,\} \quad . \qquad (12) \end{array}$$

As functions of m, we let $S.m$ denote the set on either side of (11) and $T.m$ the set on either side of (12). That $|S.m| = |T.m|$ follows from the hints in the calculation above.

We let $A.(S.m)$ be the set $S.m$ in which A has been applied to every element. With that, we can rewrite (12) as

$$A.(S.m) \ = \ T.m \quad . \tag{13}$$

For $S.m = \{i\}$ and $T.m = \{k\}$, we have $\{A.i\} = \{k\}$, that is, $A.i = k$, which gives us a way to reconstruct $A.i$ given $S.m$ and $T.m$. Now, that is only whenever $|S.m| = 1$, but the idea inspires our algorithm.

We continue our exposition together with an example. Consider the following values for A, B, D where $N = 9$.

		0	1	2	3	4	5	6	7	8
A	=	4	8	0	7	1	5	3	6	2
B	=	0	1	0	2	1	3	2	5	2
D	=	0	1	2	2	0	3	5	2	1

For $m = 2$, we find

$$\begin{array}{lll} S.2 & = & \{3, 6, 8\} \\ T.2 & = & \{2, 3, 7\} \quad . \end{array}$$

```
forall m in 0.., N loop S.m, T.m := ε, ε end;
for i := 0 to N − 1 loop S.(B.i), T.(D.i) := S.(B.i)++i, i++T.(D.i) end;
forall m in 0.., N loop
    forall i in 0.., |S.m| loop A.(S.m.i) := T.m.i end;
end
```

Figure 1: An algorithm for computing A given B and D

Hence, from (13) we know

$$\{A.3, A.6, A.8\} = \{2, 3, 7\} \quad , \tag{14}$$

but which value goes with which index? We note, for any i, k,

$$\begin{array}{ll} & \mathit{true} \\ = & \quad \{ \text{ for } i < k \ \wedge \ A.i < A.k \ , \ B.k \text{ counts at least } A.i \text{ and everything counted by } B.i \ \} \\ & i < k \ \wedge \ A.i < A.k \ \Rightarrow \ B.i < B.k \\ = & \quad \{ \text{ calculus } \} \\ & i < k \ \wedge \ B.i \geq B.k \ \Rightarrow \ A.i \geq A.k \\ = & \quad \{ \ i \neq k \ \Rightarrow \ A.i \neq A.k \ \} \\ & i < k \ \wedge \ B.i \geq B.k \ \Rightarrow \ A.i > A.k \\ \Rightarrow & \quad \{ \text{ strengthen antecedent } \} \\ & i < k \ \wedge \ B.i = B.k \ \Rightarrow \ A.i > A.k \quad . \end{array} \tag{15}$$

Our solution is now clear: We treat $S.m$ and $T.m$ as ordered lists, $S.m$ in increasing order and $T.m$ in decreasing order. Indexing these lists by an i in $0 \leq i < |S.m|$, we can formulate a stronger version of (13) as

$$A.(S.m.i) = T.m.i \quad .$$

Applying this to our example (14), we get

$$A.3 = 7 \qquad A.6 = 3 \qquad A.8 = 2 \quad .$$

We write our algorithm in Figure 1. We use the notation $m..,n$ to mean the $n-m$ integers beginning at m, that is, $m, m+1, \ldots, n-1$. (This notation is a variation of the one used in [3].) We also use $+\!\!+$ as the operator that concatenates two lists. An empty list is denoted ε, and we do not distinguish between an integer and a list containing one integer.

The iterations of the **forall** loops can be done sequentially in any order, or can be done in parallel, whereas the **for** loop, as written, should be done sequentially. The algorithm uses linear space and runs in time $\mathrm{O}(N)$.

Computing A from C and E can be done in a similar way. The difference from the above is that (15) becomes

$$i < k \ \wedge \ C.i = B.k \ \Rightarrow \ A.i < A.k \quad .$$

Hence, an algorithm for computing A from C and E is the one in Figure 1 with the assignment in the **for** loop replaced by

$$S.(C.i),\ T.(E.i) := i \mathbin{+\!\!+} S.(C.i),\ i \mathbin{+\!\!+} T.(E.i) \qquad .$$

2 Computing the code

In this section, we develop an algorithm for computing the code of a given array. In fact, our algorithm will be slightly more general, because rather than requiring the input to be a permutation of the first N natural numbers, we allow it to be any array of N distinct integers. The algorithm will run in $O(N \log N)$ time using linear space.

2.0 Specification

For any array a and integers m and n (with $m \leq n$), we let $a[m..,n]$ denote the subarray of the $n-m$ elements of a indexed by consecutive integers starting at m. (Note that $a[n]$ is not an element of that subarray. We let the application of the brackets after a bind stronger than function application. So, for example, $f.a[m..,n]$ means $f.(a[m..,n])$, not $(f.a)[m..,n]$. The elements in the subarray are referenced using the same indices as those in the whole array. We will not make any distinction between arrays and subarrays, and will refer to either as arrays. To refer to the element indexed by an appropriate i in an array a, we write $a.i$. We define equality between two arrays $a[m..,m+l]$ and $b[k..,k+l]$ of the same size as

$$\langle\, \forall\, i \mid 0 \leq i < l \,\triangleright\, a.(m+i) = b.(k+i) \,\rangle \qquad .$$

For any array $a[m..,n]$, we define $code.a[m..,n]$ as an array $b[m..,n]$ where, for each i,

$$b.i = \langle\, \#\, j \mid m \leq j < i \,\triangleright\, a.j < a.i \,\rangle \qquad . \tag{16}$$

Moreover, we let $distinct.a[m..,n]$ denote that $a[m..,n]$ is an array of distinct integers.

We define our problem specification as writing a program that computes b according to

$$\begin{array}{ll} \text{Pre}: & 0 \leq N \wedge distinct.a[0..,N] \\ \text{Post}: & b[0..,N] = code.a[0..,N] \end{array} \tag{17}$$

where a and b are global arrays. Under initial condition $a = A$, our algorithm then establishes $b = B$, where A and B are those from Section 0.

So as to make our specifications easier to read, we will follow the convention that any array element not mentioned in a postcondition or invariant is unchanged by the procedure or loop, respectively. Moreover, array a will always remain unchanged, and so will any **in** parameters.

In our program notation, we will allow local variables to be declared at any time. The scope of such a variable begins at its declaration and ends at the end of the current block. For purposes of counting how much space is being used, we will use that a local variable exists only within its scope. We will always count space in terms of number of integers, and pledge not to pull any bitpacking stunts.

A special kind of local variable is declared using **const** $k : P$. This construct declares a local variable k, whose initial value satisfies condition P. After its declaration, k is read-only.

In the following sections, we develop the program and enhance its efficiency in stages.

```
procedure Code(in m, in n) =                    { Specification: (18) }
begin
  if n − m = 0 → skip
  [] n − m = 1 → c.m := 0
  [] n − m ≥ 2 → const h : h = (m + n) div 2;
                 Code(m, h); Code(h, n);
                 var k; k := h;
                 { Invariant: ⟨∀ i | m ≤ i < k ▷ b.i = ⟨# j | m ≤ j < i ▷ a.j < a.i ⟩ ⟩ ∧
                              ⟨∀ i | k ≤ i < n ▷ b.i = ⟨# j | h ≤ j < i ▷ a.j < a.i ⟩ ⟩ ∧
                              h ≤ k ≤ n }
                 do k ≠ n → b.k, k := b.k + ⟨# j | m ≤ j < h ▷ a.j < a.k ⟩, k + 1 od
  fi
end     .
```

Figure 2: A divide-and-conquer algorithm for computing the code

2.1 A divide-and-conquer algorithm

The large range of j in (16) is an obstacle when attempting to write an efficient solution to the problem. We will attempt to reduce this range by means of a divide-and-conquer algorithm $Code$ with parameters m and n, according to the specification

$$\begin{array}{ll} \text{Pre}: & m \le n \wedge distinct.a[m..,n] \\ \text{Post}: & b[m..,n] = code.a[m..,n] \quad . \end{array} \tag{18}$$

The program statement that solves the entire problem (18) is then $Code(0, N)$.

With two base cases and one recursive step, we write our program as in Figure 2. We annotate our programs with the crucial ingredients of a correctness proof, but, for brevity, we omit the proofs themselves. Termination will always be obvious.

The recursion depth of this algorithm is $O(\log(n-m))$. The quantified expression in the loop takes $O(h-m)$ steps to compute using a naïve implementation, and the loop iterates $n-h$ times. The total time needed for the loop is then $O((n-m)^2)$, so $Code(0, N)$ has time complexity $O(N^2)$. We also need space proportional to the recursion depth.

Our next objective is to eliminate the unfortunate quantified expression. In the following sections, we will maintain the overall structure of the above algorithm, but will attempt to reduce the time required for the recursive step.

2.2 Adding a sorted permutation of a

The previous version of our program used a lot of time computing the quantified expression

$$\langle \# j \mid m \le j < h \triangleright a.j < a.k \rangle \quad ,$$

since it did so using a simple loop. We can speed this up if we have a sorted permutation of $a[m..,h]$, denoted by the array

$$sort.a[m..,h]$$

whose element indices are in the range $m..,h$. Then we can apply a binary search to the sorted array in order to find the value of the quantified expression. For this purpose, we introduce a new global array s, and modify the specification of $Code$ accordingly to

$$\begin{array}{ll} \text{Pre}: & m \leq n \wedge distinct.a[m..,n] \\ \text{Post}: & b[m..,n] = code.a[m..,n] \wedge \\ & s[m..,n] = sort.a[m..,n] \quad . \end{array} \tag{19}$$

We let min (max) applied to any array $a[m..,n]$ denote the minimum (maximum) element in $a[m..,n]$, or ∞ ($-\infty$) if $m = n$.

Updating the base cases of our program is trivial. Updating the recursive step can be done using a merge, whose loop invariant is

$$\begin{array}{l} ss[m..,k] = (sort.a[m..,n])[m..,k] \wedge \\ s[m..,h] = sort.a[m..,h] \wedge s[h..,n] = sort.a[h..,n] \wedge \\ max.ss[m..,k] = max.s[m..,x] \textbf{ max } max.s[h..,y] < min.s[x..,h] \textbf{ min } min.s[y..,n] \wedge \\ m \leq x \leq h \leq y \leq n \wedge m \leq k \leq n \quad . \end{array} \tag{20}$$

We write the next approximation of our program as in Figure 3.

Let us first examine our new space requirements. The new global array s requires linear space, and so does local array ss. We observe that at most one ss array exists at any one time. Hence, we are within linear space, irrespective of the recursion depth.

Looking at the time needed, we know the quantified expression can be replaced by a binary search. The binary search is performed once for each of the $n-h$ iterations, requiring a total of $O((n-h)\log(h-m))$ steps. The merge is faster, needing only $O(n-m)$ steps. The total time required to compute $Code(0,N)$ is thus $O(N \log^2 N)$.

We realize at this time that we already have achieved writing an algorithm that is faster than $O(N^2)$ using linear space. However, we think it unfortunate that a binary search is needed to update each value in the upper half of b. Hence, we strive for an even faster algorithm.

2.3 Adding a mapping from s to a

Arrays s and a contain permutations of the same elements. Since the elements of s are sorted, it is easy to compute the code of s. We can make use of this when computing the code of a only if we know how the elements of s correspond to those in a. We will invent a mechanism for just that in this next and final approximation of our program.

We introduce a new global array t, satisfying, for each of its indices i,

$$a.(t.i) = s.i \quad . \tag{21}$$

```
procedure Code(in m, in n) =                    { Specification: (19) }
begin
  if n − m = 0 → skip
  [] n − m = 1 → b.m, s.m := 0, a.m
  [] n − m ≥ 2 → const h : h = (m + n) div 2;
                 Code(m, h); Code(h, n);
                 var k; k := h;
                 do k ≠ n → b.k, k := b.k + ⟨ # j | m ≤ j < h ▷ a.j < a.k ⟩, k + 1 od;
                 var x, y, ss[m.., n]; k, x, y := m, m, h;
                 { Invariant: (20) }
                 do k ≠ n →
                   if y = n ∨ (x ≠ h ∧ s.x < s.y) →   {s.x = min.s[x.., h] min min.s[y.., n]}
                        x, k, ss.k := x + 1, k + 1, s.x
                    [] x = h ∨ (y ≠ n ∧ s.y < s.x) →   {s.y = min.s[x.., h] min min.s[y.., n]}
                        y, k, ss.k := y + 1, k + 1, s.y
                   fi
                 od;
                 s[m.., n] := ss[m.., n]
  fi
end     .
```

Figure 3: Program with merge

```
var x, y, ss[m.., n], tt[m.., n];
k, x, y := m, m, h;
{ Invariant: (20) ∧  ⟨∀ i | m ≤ i < k ▷ m ≤ tt.i < n ∧ a.(tt.i) = ss.i ⟩ ∧
                     ⟨∀ i | m ≤ i < h ▷ m ≤ t.i < h ∧ a.(t.i) = s.i ⟩ ∧
                     ⟨∀ i | h ≤ i < n ▷ h ≤ t.i < n ∧ a.(t.i) = s.i ⟩ }
do k ≠ n →  if y = n ∨ (x ≠ h ∧ s.x < s.y) → x, k, ss.k, tt.k := x + 1, k + 1, s.x, t.x
            [] x = h ∨ (y ≠ n ∧ s.y < s.x) → y, k, ss.k, tt.k := y + 1, k + 1, s.y, t.y
            fi
od;
s[m.., n], t[m.., n] := ss[m.., n], tt[m.., n]      .
```

Figure 4: Merge extended to also calculate t

As was the case when adding s, updating the base cases is trivial. For the recursive step, we need to permute t in the same way we permute s. This alters our merge as given in Figure 4, after which *Code* satisfies the specification

$$\begin{array}{ll} \text{Pre :} & m \leq n \wedge \mathit{distinct}.a[m..,n] \\ \text{Post :} & c[m..,n] = \mathit{code}.a[m..,n] \wedge s[m..,n] = \mathit{sort}.a[m..,n] \wedge \\ & \langle\, \forall\, i \mid m \leq i < n \;\triangleright\; m \leq t.i < n \wedge a.(t.i) = s.i \,\rangle \quad . \end{array} \tag{22}$$

We would like to update the upper half of b in this loop as well. In particular, we want the loop to also implement

$$\begin{array}{ll} \text{Pre :} & \langle\, \forall\, i \mid m \leq i < h \;\triangleright\; b.i = \langle\, \#\, j \mid m \leq j < i \;\triangleright\; a.j < a.i \,\rangle \,\rangle \wedge \\ & \langle\, \forall\, i \mid h \leq i < n \;\triangleright\; b.i = \langle\, \#\, j \mid h \leq j < i \;\triangleright\; a.j < a.i \,\rangle \,\rangle \\ \text{Post :} & \langle\, \forall\, i \mid m \leq i < n \;\triangleright\; b.i = \langle\, \#\, j \mid m \leq j < i \;\triangleright\; a.j < a.i \,\rangle \,\rangle \quad . \end{array}$$

Since $t[m..,h]$ is a permutation of the numbers in the range $m..,h$, and $t[h..,n]$ is a permutation of the numbers in $h..,n$, we can restate the above conditions as

$$\begin{array}{ll} \text{Pre :} & \langle\, \forall\, i \mid m \leq i < h \;\triangleright\; b.(t.i) = \langle\, \#\, j \mid m \leq j < t.i \;\triangleright\; a.j < a.(t.i) \,\rangle \,\rangle \wedge \\ & \langle\, \forall\, i \mid h \leq i < n \;\triangleright\; b.(t.i) = \langle\, \#\, j \mid h \leq j < t.i \;\triangleright\; a.j < a.(t.i) \,\rangle \,\rangle \\ \text{Post :} & \langle\, \forall\, i \mid m \leq i < n \;\triangleright\; b.(t.i) = \langle\, \#\, j \mid m \leq j < t.i \;\triangleright\; a.j < a.(t.i) \,\rangle \,\rangle \quad . \end{array}$$

For the loop invariant, we can replace a constant, *viz.*, h, by a variable in the range from h to n. We have a variable just like that, *viz.*, y. Thus, we will add to the merge loop invariant

$$\begin{array}{l} \langle\, \forall\, i \mid m \leq i < y \;\triangleright\; b.(t.i) = \langle\, \#\, j \mid m \leq j < t.i \;\triangleright\; a.j < a.(t.i) \,\rangle \,\rangle \wedge \\ \langle\, \forall\, i \mid y \leq i < n \;\triangleright\; b.(t.i) = \langle\, \#\, j \mid h \leq j < t.i \;\triangleright\; a.j < a.(t.i) \,\rangle \,\rangle \quad , \end{array}$$

which frees us from the need of the first loop.

One way to satisfy this invariant is to use a statement like

$$b.(t.y) := b.(t.y) + \langle\, \#\, j \mid m \leq j < h \;\triangleright\; a.j < a.(t.y) \,\rangle$$

in the second alternative of the conditional inside the loop. We see the unwanted quantified expression crop up once more, but this time we are armed to eliminate it. We know the value of the quantified expression is at most $h - m$. The simplest expression we can think of that has a good chance of "being right" is $x - m$. We calculate, under the new invariant and the guards leading to the execution of the second alternative of the conditional (see Figure 3),

$$\begin{array}{ll} & x - m = \langle\, \#\, j \mid m \leq j < h \;\triangleright\; a.j < a.(t.y) \,\rangle \\ = & \quad \{\ (21) \text{ — property of } t \ \} \\ & x - m = \langle\, \#\, j \mid m \leq j < h \;\triangleright\; a.j < s.y \,\rangle \\ = & \quad \{\ s[m..,h] = \mathit{sort}.a[m..,h] \text{, so } s[m..,h] \text{ is a permutation of } a[m..,h] \ \} \\ & x - m = \langle\, \#\, j \mid m \leq j < h \;\triangleright\; s.j < s.y \,\rangle \\ = & \quad \{\ \text{split range, since } m \leq x \leq h \ \} \\ & x - m = \langle\, \#\, j \mid m \leq j < x \;\triangleright\; s.j < s.y \,\rangle + \langle\, \#\, j \mid x \leq j < h \;\triangleright\; s.j < s.y \,\rangle \\ \Leftarrow & \quad \{\ \text{all/none of the terms being } \mathit{true} \ \} \\ & x - m = \langle\, \#\, j \mid m \leq j < x \;\triangleright\; s.j < s.y \,\rangle \wedge 0 = \langle\, \#\, j \mid x \leq j < h \;\triangleright\; s.j < s.y \,\rangle \\ = & \quad \{\ \#\ \text{and}\ \forall \ \} \end{array}$$

```
procedure Code(in m, in n) =                    { Specification: (22) }
begin
  if n − m = 0 → skip
  [] n − m = 1 → b.m, s.m, t.m := 0, a.m, m
  [] n − m ≥ 2 → const h : h = (m + n) div 2;
                 Code(m, h); Code(h, n);
                 var k, x, y, ss[m.., n], tt[m.., n];
                 k, x, y := m, m, h;
                 do k ≠ n →
                   if y = n ∨ (x ≠ h ∧ s.x < s.y) → x, k, ss.k, tt.k := x + 1, k + 1, s.x, t.x
                   [] x = h ∨ (y ≠ n ∧ s.y < s.x) → y, k, ss.k, tt.k, b.(t.y) :=
                                                      y + 1, k + 1, s.y, t.y, b.(t.y) + x − m
                   fi
                 od;
                 s[m.., n], t[m.., n] := ss[m.., n], tt[m.., n]
  fi
end     .
```

Figure 5: The final program

$$
\begin{array}{ll}
 & \langle \forall\, j \mid m \le j < x \,\triangleright\, s.j < s.y \,\rangle \;\wedge\; \langle \forall\, j \mid x \le j < h \,\triangleright\, s.y \le s.j \,\rangle \\
= & \quad \{\quad \mathbf{max}, \mathbf{min} \quad \} \\
 & max.s[m.., x] < s.y \;\wedge\; s.y \le min.s[x.., h] \\
= & \quad \{\quad s.y = min.s[x.., h] \;\mathbf{min}\; min.s[y.., n]\,, \text{ twice} \quad \} \\
 & max.s[m.., x] < min.s[x.., h] \;\mathbf{min}\; min.s[y.., n] \\
\Leftarrow & \quad \{\quad \mathbf{max} \quad \} \\
 & max.s[m.., x] \;\mathbf{max}\; max.s[h.., y] < min.s[x.., h] \;\mathbf{min}\; min.s[y.., n] \\
= & \quad \{\quad \text{Invariant} \quad \} \\
 & true \qquad .
\end{array}
$$

We write our final program in Figure 5. This program uses only $O(N \log N)$ time with $O(N)$ space!

As a little optimization, if a is not needed at the end of this computation, we can eliminate array b by storing $b[m.., n]$ in $a[m.., n]$. This is because the only reference to an element of a occurs in one of the base cases.

2.4 Inverting the program

In this section, we touch on the rules for program inversion (see [1, 2, 0]) and show how our final program from Figure 5 can be inverted. We do so by explaining how to invert an assignment statement, sequential composition, **if** statements, and **do** statements. By inverting our program, we hope to arrive at a

program that computes A from B.

We begin with the assignment statement. Consider, for example, the statement that increments a variable n by 1, $n := n + 1$. The inverse of it is the statement that decreases the same variable by 1, $n := n - 1$.

So what about a statement like $n := 3$? We cannot invert this statement by itself, because there is no information about the previous value of n. However, the program

$$\{n = 5\}\ n := 3 \quad ,$$

that is, the assignment $n := 3$ known to start in a state where $n = 5$, can be inverted. Its inverse is

$$\{n = 3\}\ n := 5 \quad .$$

The point of inverting a program is often to reverse the rôles of input and output. If out is a variable that is part of the output of a program, we can view a statement $out := x$ as $write(out, x)$, that is, writing x to stream out. Thus, the inverse of such a statement is $read(out, x)$, that is, reading x from stream out. Rather than the $read$ statement, we may use $x := out$. Similarly, if in is a variable that is part of the input of a program, we invert $x := in$ as $in := x$.

So much for assignment. Sequential composition is easy: For statements S and T,

$$(S; T)^{-1} = T^{-1}; S^{-1} \quad .$$

Running an **if** statement in reverse requires that it be known which alternative to take. So, to invert an **if** statement like

$$\textbf{if } B0 \rightarrow S0 \ [\!] \ B1 \rightarrow S1 \textbf{ fi} \quad ,$$

we need to find two mutually exclusive conditions $C0$ and $C1$ such that one holds after execution of $S0$ and the other after execution of $S1$. We restrict our attention to inverting only deterministic programs, so we assume $B0$ and $B1$ are mutually exclusive, too. With that, we state the rule for inverting an **if** statement as

$$(\textbf{if } B0 \rightarrow S0\{C0\} \ [\!] \ B1 \rightarrow S1\{C1\} \textbf{ fi})^{-1} = \textbf{if } C0 \rightarrow S0^{-1}\{B0\} \ [\!] \ C1 \rightarrow S1^{-1}\{B1\} \textbf{ fi} \quad .$$

Similar to the **if** statement is the **do** loop. Here, too, there is a choice of whether or not to iterate the loop backwards once more. For this, we find a condition C that holds after every execution of the loop body and that does not hold prior to entering the loop. The rule for inverting a loop can then be stated as

$$\begin{array}{l} \quad (\{\neg C\}\ \textbf{do } B \rightarrow S\{C\}\ \textbf{od}\ \{\neg B\})^{-1} \\ = \\ \quad \{\neg B\}\ \textbf{do } C \rightarrow S^{-1}\{B\}\ \textbf{od}\ \{\neg C\} \end{array} \quad .$$

Now that we have presented what program inversion is all about, we return to the task of inverting our final program from Figure 5. The following conditions show that the program is indeed invertible.

Outer **if** alternatives:	$n - m = 0$	$n - m = 1$	$n - m \geq 2$
Before and after **do** loop:	$m = k$	$m \neq k$	
Inner **if** alternatives:	$m \leq tt.(k - 1)$	$tt.(k - 1) < m$	

```
{a = A}
forall k in 0..,N loop e.k := 0 end;
for k := ⌊log N⌋ downto0 loop
  forall s in 0..⌊N/2^(k+1)⌋ loop x.s := 0 end;
  var j; j := 0;
  do j ≠ N →
    const r,s : r = ⌊a.j/2^k⌋ mod 2 ∧ s = ⌊a.j/2^(k+1)⌋ ;
    if r = 0 → e.(a.j) := e.(a.j) + x.s [] r = 1 → x.s := x.s + 1 fi;
    j := j + 1
  od
end
{e = E}
```

Figure 6: Knuth's algorithm for computing the inversion table

The inverse computes array a given its code b and the arrays s and t. It is very reasonable that s be part of the input to a program that computes an array from its code, because the code only contains ordering information, not the array numbers themselves. The programs in [1, 2] require that the given array consists of a permutation of the first N natural numbers; hence, s is implied, and can be omitted. Our program, on the other hand, allows a to be any array of distinct integers.

The array t, however, is generally not available when wanting to compute an array from its code. In fact, by letting t into the picture, a can be computed directly from s and t without using b at all, as is seen from (21).

So, although it exists, the inverse of $Code$ is a program that is probably not very useful in practice.

3 A proof of Knuth's encoding algorithm

In this section, we present a proof of an algorithm that computes E from A. The algorithm, shown in Figure 6, is from Section 5.1.1, Ex. 6 of [4].

For any j in the range $0..,N$, we have $a.j < N$, and thus the constant s satisfies $s \leq \lfloor N/2^{k+1} \rfloor$. Hence, every $x.s$ used in the inner loop has been set to 0 in the **forall** s loop, in the same **for** k iteration. We then see that the only information kept between iterations of the outer loop is e. As the only operation on e within the inner loop is an increment, the order in which the iterations of the outer loop are performed does not affect the program output (despite that Knuth specifies the iterations to be done in decreasing order of k).

In order to understand the program, we write an invariant for the inner loop. For convenience, we let $bit.k.y$ denote $\lfloor y/2^k \rfloor$ **mod** 2, for natural k and y. We then write the invariant as

$$\begin{array}{l} 0 \leq j \leq N \wedge \\ \langle\, \forall\, s \mid 0 \leq s \leq \lfloor n/2^{k+1} \rfloor \;\triangleright\; x.s \;=\; \langle\, \#\, t \mid 0 \leq t < j \;\triangleright\; \lfloor a.t/2^{k+1} \rfloor = s \wedge bit.k.(a.t) = 1 \,\rangle \,\rangle \end{array} \quad . \tag{23}$$

We leave it to the reader to verify this invariant.

So, at the time $e.(a.j)$ is incremented by $x.s$,

$$(23)\ \wedge\ r = bit.k.(a.j) = 0\ \wedge\ s = \lfloor a.j/2^{k+1} \rfloor$$

holds. Therefore, the program computes each $e.(a.j)$ to be

$$\langle\, \Sigma\, k \mid 0 \le k \le \lfloor \log N \rfloor \wedge bit.k.(a.j) = 0 \,\triangleright \\ \langle\, \#\, t \mid 0 \le t < j \,\triangleright\, \lfloor a.t/2^{k+1} \rfloor = \lfloor a.j/2^{k+1} \rfloor \wedge bit.k.(a.t) = 1 \,\rangle\,\rangle \qquad .$$

We calculate,

$$\begin{array}{ll} & e.(a.j) \\ = & \quad \{\ \text{observations made above}\ \} \\ & \langle\, \Sigma\, k \mid 0 \le k \le \lfloor \log N \rfloor \ \wedge\ bit.k.(a.j) = 0 \,\triangleright \\ & \quad \langle\, \#\, t \mid 0 \le t < j \,\triangleright\, \lfloor a.t/2^{k+1} \rfloor = \lfloor a.j/2^{k+1} \rfloor \ \wedge\ bit.k.(a.t) = 1 \,\rangle\,\rangle \\ = & \quad \{\ \text{term is 0 for } k > \lfloor \log N \rfloor, \text{ because } a.t < N \text{ and therefore } bit.k.(a.t) = 0\ \} \\ & \langle\, \Sigma\, k \mid bit.k.(a.j) = 0 \,\triangleright\, \langle\, \#\, t \mid 0 \le t < j \,\triangleright\, \lfloor a.t/2^{k+1} \rfloor = \lfloor a.j/2^{k+1} \rfloor \ \wedge\ bit.k.(a.t) = 1 \,\rangle\,\rangle \\ = & \quad \{\ \text{nesting}\ \} \\ & \langle\, \Sigma\, t \mid 0 \le t < j \,\triangleright\, \langle\, \#\, k \mid bit.k.(a.j) = 0 \,\triangleright\, \lfloor a.t/2^{k+1} \rfloor = \lfloor a.j/2^{k+1} \rfloor \ \wedge\ bit.k.(a.t) = 1 \,\rangle\,\rangle \\ = & \quad \{\ \text{for each } t \text{ and } j, \text{ at most one } k \text{ satisfies the range and term}\ \} \\ & \langle\, \#\, t \mid 0 \le t < j \,\triangleright\, \langle\, \exists\, k \mid bit.k.(a.j) = 0 \,\triangleright\, \lfloor a.t/2^{k+1} \rfloor = \lfloor a.j/2^{k+1} \rfloor \ \wedge\ bit.k.(a.t) = 1 \,\rangle\,\rangle \\ = & \quad \{\ \text{lexicographical bit-order}\ \} \\ & \langle\, \#\, t \mid 0 \le t < j \,\triangleright\, a.t > a.j \,\rangle \qquad , \end{array}$$

which shows the correctness of the program.

4 Inverting an algorithm that decodes an inversion table

In this section, we present Knuth's fast algorithm for decoding an inversion table, or, to use the notation introduced in Section 0, computing A from E. Using the rules for program inversion from Section 2.4, we then invert Knuth's algorithm, arriving at a new efficient algorithm for computing E from A.

We consider strings of pairs of natural numbers, and use the following notational conventions.

ε	the empty string
juxtaposition	string catenation
$[a, A]$	a pair consisting of the natural numbers a and A
Greek letters, x, y, z	strings

So, for example, $[a, A][b, B][c, C]$ is a string of length 3, and $[a, A]\alpha$ is a string that begins with the pair $[a, A]$. We define binary operator $\circ$, binding weaker than catenation, on strings as follows.

$$\varepsilon \circ \alpha = \alpha = \alpha \circ \varepsilon$$

$$[a, A]\alpha \circ [b, B]\beta = \begin{cases} [a, A](\alpha \circ [b - a, B]\beta) & \text{if } a \le b \\ [b, B]([a - b - 1, A]\alpha \circ \beta) & \text{if } b < a \end{cases}$$

```
procedure circ(in α, in β, out σ) =
{ Post:  σ = α ∘ β  }
begin
   var z, x, y;
   z, x, y := ε, α, β;
   { Invariant:  z(x ∘ y) = α ∘ β  }
   do x ≠ ε ∧ y ≠ ε →
      const a, A, γ : x = [a, A]γ;
      const b, B, δ : y = [b, B]δ;
      if a ≤ b → z, x, y := z[a, A], γ, [b − a, B]δ
      [] b < a → z, x, y := z[b, B], [a − b − 1, A]γ, δ
      fi
   od;     {Invariant ∧ (x = ε ∨ y = ε)}
   if x = ε → z, y := z y, ε
   [] y = ε → z, x := z x, ε
   fi;       {x = y = ε ∧ z = α ∘ β}
   σ := z
end
```

Figure 7: Program that computes $\alpha \circ \beta$

This definition reminds of a merge, in that the recursive step selects either the first pair from $[a, A]\alpha$ or the first pair from $[b, B]\beta$, after which $\circ$ is applied to a smaller problem. The time needed to compute $\alpha \circ \beta$ is then linear in the length of $\alpha\beta$. We display this algorithm, called *circ*, in Figure 7.

Knuth leaves to the reader, and so do we, to verify that $\circ$ is associative and that

$$[E.0, 0] \circ [E.1, 1] \circ \cdots \circ [E.(N-1), N-1] \;=\; [0, A.0][0, A.1] \cdots [0, A.(N-1)] \qquad , \tag{24}$$

where A and E are those from Section 0. Because $\circ$ is associative, we can compute the $\circ$ operators of (LHS-24) in any order. In particular, we can compute (RHS-24) from (LHS-24) by breaking (LHS-24) up into two halves, computing each of the halves, and then computing the middle $\circ$ operator. This is the description of a divide-and-conquer algorithm, which then runs in time $\mathrm{O}(N \log N)$. We show the algorithm, named *Decode*, in Figure 8. Its specification is

$$\begin{array}{ll} \text{Pre :} & m \leq n \\ \text{Post :} & aa = ee[m] \circ ee[m+1] \circ \cdots \circ ee[n-1] \qquad . \end{array} \tag{25}$$

When writing specifications, we will use the conventions introduced in Section 2.0. The statement that solves the entire problem is then $Decode(0, N, E', A')$, where E' and A' are E and A modified to fit the left- and right-hand sides of (24).

Constructing the inverse of *Decode*, call it *Encode*, is easy. Since m and n are not modified, the guards of *Encode* are the same as those of *Decode*. Instead of two recursive calls to *Decode* followed

```
procedure Decode(
      in m, in n,
      in ee : array m, n.., of string,
      out aa : string) =
{ Specification: (25) }
begin
   if n − m = 0 → aa := ε
   [] n − m = 1 → aa := ee[m]
   [] n − m ≥ 2 →
      const p : p = (m + n) div 2;
      var a0, a1 : string;
      Decode(m, p, ee[m.., p], a0);
      Decode(p, n, ee[p.., n], a1);
      circ(a0, a1, aa)
   fi
end
```

Figure 8: Program that decodes an inversion table

by a call to *circ*, the inverse program has a call to the inverse of *circ* followed by two recursive calls to *Encode*.

So what about the inverse of *circ*? This program, call it *cric*, needs to decompose a given string σ into two strings α and β such that $\sigma = \alpha \circ \beta$. The problem is, however, that such α and β are not, in general, uniquely determined. Indeed, inspecting the first **if** statement of *circ*, we find that there is no information available after the **if** statement that allows us to conclude which alternative was executed.

To find the solution, we return to the definition of $\circ$ and (24). From the former, we see that the left-hand sides of the pairs are modified in the recursive step, but the right-hand sides are not. Also, regardless of the order in which the $\circ$ operations are carried out when computing (LHS-24), the relative positioning of the operands of $\circ$ is the same. Moreover, the right-components of (24) are arranged in ascending order.

The solution now stares us in the face. Our observations show that every right-component of $ee[m..,p]$ is less than every right-component of $ee[p..,n]$ in *Decode*. In fact, every right-component of $ee[m..,p]$ is less than p, and every right-component of $ee[p..,n]$ is at least p. By passing this p to *circ*, there is enough information in *circ* for it to be inverted.

More precisely, let $R.\alpha$ be the string of right-components of α, that is, α projected onto its right-components. We define *ascend*, for any string α, as

$$ascend.\alpha \;=\; R.\alpha \text{ is strictly ascending} \quad .$$

Finally, we define $\alpha \sqsubset_p \beta$, as

$$\langle \forall\, A \mid A \in R.\alpha \;\triangleright\; A < p \,\rangle \wedge \langle \forall\, B \mid B \in R.\beta \;\triangleright\; p \leq B \,\rangle \quad .$$

We now add the precondition

$$ascend.(ee[m]\ ee[m+1]\ \cdots\ ee[n-1]) \wedge \langle\, \forall\, i \mid m \le i < n \,\triangleright\, first.(R.ee[i]) = i \,\rangle$$

to *Decode*. We leave it to the reader to verify that this condition holds for (LHS-24) and for each recursive call. Adding to *circ* new parameter **in** p, we prescribe its new precondition as $\alpha \sqsubset_p \beta$, and accordingly modify the call of *circ* from *Decode* to

$$circ(a0, a1, aa, p) \quad .$$

Next, we extend the loop invariant in *circ* with

$$\begin{array}{l} x \sqsubset_p y \wedge (x \text{ is suffix of } \alpha) \wedge (y \text{ is suffix of } \beta) \\ (z \neq \varepsilon \wedge x = \varepsilon \Rightarrow last.(R.z) < p) \wedge \\ (z \neq \varepsilon \wedge y = \varepsilon \Rightarrow p \le last.(R.z)) \quad . \end{array}$$

If $\alpha = \beta = \varepsilon$ holds initially in *circ*, then the loop will not iterate, and thus both guards of the second **if** statement will be true, but we only want to invert deterministic statements. As $m < p < n$ holds in *Decode*, both $a0$ and $a1$ will have positive length in the call to *circ*. Realizing the opportunity, we thus add

$$\alpha \neq \varepsilon \wedge \beta \neq \varepsilon$$

as another precondition of *circ*. Using this, the invariant, and the negation of the guards, we conclude that

$$z \neq \varepsilon \wedge (x = \varepsilon) \neq (y = \varepsilon)$$

is a precondition of the second **if** statement.

Now, the following annotations reveals that the **if** and **do** statements of *circ* can be inverted.

Before and after **do** loop:	$z = \varepsilon$	$z \neq \varepsilon$
Alternatives of first **if** statement:	$z \neq \varepsilon \wedge y \neq \varepsilon \wedge last.(R.z) < p$	
	$z \neq \varepsilon \wedge x \neq \varepsilon \wedge p \le last.(R.z)$	
Alternatives of second **if** statement:	$p \le last.(R.z)$	$last.(R.z) < p$

We now only have one task left: inverting the assignment statements. These are all straightforward, except the ones in the second **if** statement. We show one alternative; the other is symmetric. Our task is to invert

$$\begin{array}{l} \{z \neq \varepsilon \wedge x \neq \varepsilon \wedge y = \varepsilon \wedge p \le last.(R.z)\} \\ z, x := z\ x, \varepsilon \\ \{last.(R.z) < p\} \quad . \end{array}$$

(The last conjunct of the precondition comes from the loop invariant.) The inverse program then moves pairs from the tail of z, putting them on x, until $p \le last.(R.z)$ holds. This can be done using a loop.

We are now done, and show the inverse programs, *Encode* and *cric*, in Figures 9 and 10. With that, we have shown a new linear space, time $O(N \log N)$ algorithm for computing E from A. This algorithm, however, distinguishes itself from the others mentioned, as being efficient *and* having been obtained as the inverse of another algorithm.

```
procedure Encode(
      in m, in n,
      out ee : array m, n.., of string,
      in aa : string) =
begin
   if n − m = 0 → skip
   [] n − m = 1 → ee[m] := aa
   [] n − m ≥ 2 →
      const p : p = (m + n) div 2;
      var a0, a1 : string;
      cric(a0, a1, aa, p)
      Encode(p, n, ee[p.., n], a1);
      Encode(m, p, ee[m.., p], a0);
   fi
end
```

Figure 9: Program for computing an inversion table

```
procedure cric(out α, out β, in σ, in p) =
begin
   var z, x, y;
   z, x, y := σ, ε, ε;
   if p ≤ last.(R.z) → do p ≤ last.(R.z) → const c, C, θ : z = θ[c, C]; z, y := θ, [c, C]y od
   [] last.(R.z) < p → do last.(R.z) < p → const c, C, θ : z = θ[c, C]; z, y := θ, [c, C]x od
   fi;
   do z ≠ ε →
      const c, C, θ : z = θ[c, C];
      if C < p → const b, B, δ : y = [b, B]δ;
                 z, x, y := θ, [c, C]x, [b + c, B]δ
      [] p ≤ C → const a, A, γ : x = [a, A]γ;
                 z, x, y := θ, [a + c + 1, A]γ, [c, C]y
      fi
   od;
   α, β := x, y
end
```

Figure 10: Program that decomposes $\alpha \circ \beta$

5 Conclusion

We showed 4 ways in which permutations can be encoded and how these encodings are related. In particular, we showed that one can compute between B and C in linear time, and given two of A, B, D, and E, one can compute any other in linear time.

We gave an overview of previously known algorithms for computing to and from two of these encodings, the *code* and the *inversion table*. Some of these algorithms have time complexity $O(N^2)$, whereas the efficient ones run in time $O(N \log N)$. We presented a proof for one of the existing efficient algorithms.

We also presented two new efficient algorithms, one for computing the code and one for computing the inversion table. Both algorithms have running time $O(N \log N)$ and use linear space. The concept of *program inversion* has been applied to several problems, including computing the code of a permutation, which was the example application used in the introduction of the concept [1]. We obtained one of our algorithms as an inverse of an existing algorithm, hence introducing the first *efficient* algorithm obtained in that way for a permutation encoding algorithm.

Acknowledgements

The two algorithms in Section 1 were developed jointly with Paolo A.G. Sivilotti. I am grateful for the comments on the programs, proofs, and presentation provided by Mani Chandy, Robert Harley, Peter Hofstee, Rajit Manohar, Adam Rifkin, Paul Sivilotti, and Jan van de Snepscheut (who also introduced me to the problem).

References

[0] W. Chen and J.T. Udding. Program inversion: More than fun! *Science of Computer Programming*, 15:1–13, 1990.

[1] E.W. Dijkstra. *Program Inversion.* EWD 671, Technological University Eindhoven, 1978. Published in *Selected Writings on Computing: a Personal Perspective.* Springer-Verlag, 1982.

[2] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[3] E.C.R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.

[4] D.E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and searching.* Addison-Wesley, 1973.