BA

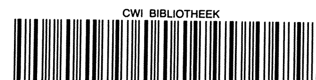
stichting
mathematisch
centrum

BA

AFDELING MATHEMATISCHE BESLISKUNDE BW 28/73 AUGUST

J.K. LENSTRA
RECURSIVE ALGORITHMS FOR ENUMERATING SUBSETS,
LATTICE-POINTS, COMBINATIONS AND PERMUTATIONS

2e boerhaavestraat 49 amsterdam



BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 05A05, 05A10, 90B35, 90C10

BW 28/73

Errata

page	line	
11	23	⋇ quence → sequence ⋇
16	21	⋇ , xi; → ; <u>real</u> xi; ⋇
17	19	⋇ variabele → variable ⋇
29	15	⋇ problem(x,n,-1) → problem(x,1,-1) ⋇
29	25	⋇ initiated → initialized ⋇

BW 28/73

Errata

page	line	
11	23	≭ quence → sequence ≭
16	21	≭ , xi; → ; real xi; ≭
17	19	≭ variabele → variable ≭
29	15	≭ problem(x,n,-1) → problem(x,1,-1) ≭
29	25	≭ initiated → initialized ≭

Abstract

Recursive algorithms for enumerating various types of combinatorial configurations are presented. We consider lexicographic as well as minimum-change methods. The algorithms are defined as ALGOL 60-procedures. Their correctness and their efficiency are discussed. Finally we indicate some applications in the field of mathematical programming.

Acknowledgements

The author gratefully acknowledges the valuable help and suggestions from Jack Alanen, Peter van Emde Boas and Hendrik Lenstra.

Contents

Abstract	1
Contents	3
1. Introduction	5
2. Subsets	6
3. Lattice-points	9
4. Combinations	11
5. Permutations	15
6. Computations	19
7. Applications	21
Appendix A	25
Appendix B	28
Appendix C	34
Literature	38

1. Introduction

In this report we present algorithms for enumerating various types of combinatorial configurations. We distinguish between *lexicographic* and *minimum-change* methods. Lexicographic methods generate the configurations in a "dictionary" order, while minimum-change methods produce a sequence in which successive configurations differ as little as possible. The latter methods have two important advantages. First, the entire sequence is generated efficiently, since each configuration is derived from its predecessor by a simple change. Secondly, in some applications each configuration has to be evaluated, and a minimum-change algorithm "may permit the value of the current arrangement to be obtained by a small correction to the immediately previous value, rather than *ab initio*" [15].

Our algorithms are defined as ALGOL 60-procedures. They are based on recursive procedures, they contain no labels, and after one call they generate the entire sequence of configurations. Each time a new configuration has been obtained, a call of a procedure 'problem' is made. Parameters of this procedure are the configuration, and, for minimum-change enumeration, the positions in which it differs from its predecessor. It has to be defined by the user to handle each configuration in the desired way.

This construction is unlike what has become usual in the literature. Most of the published procedures are organised in such a way that each call generates the next configuration in the sequence (see [2; 3; 4; 16; 19]; [1] is an exception). Then in each call it is necessary to recompute the point which has been reached in the sequence [15]. This is inherent to an iterative description of essentially recursive algorithms. A mechanism for reducing this kind of computations has recently been devised by Ehrlich [5]. To us, a recursive description seems more appropriate and more transparent.

The following four paragraphs discuss algorithms for the minimum-change enumeration of *subsets, lattice-points, combinations* and *permutations*, respectively. Appendix A contains four algorithms for lexicographic enumeration, appendix B four faster versions of our minimum-change methods, and appendix C three previously published minimum-change procedures. In § 6 we compare the running times of these fifteen algorithms on a computer. Some applications to *integer programming* and *scheduling problems* are indicated in § 7.

2. Subsets

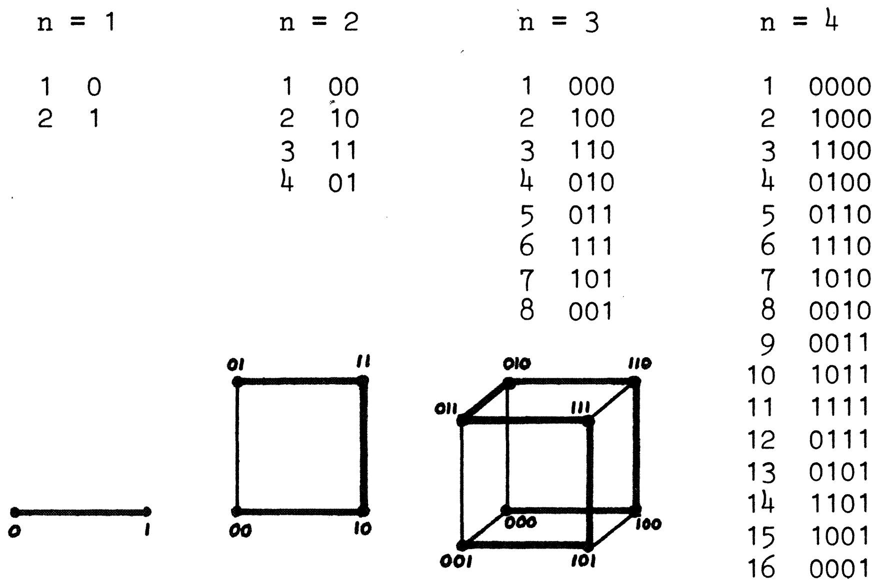
In this paragraph we discuss a method for the minimum-change enumeration of all subsets of a set, $S = \{e_1, e_2, \ldots, e_n\}$. A subset $X \subset S$ will be represented by an integer n-vector with components 0 and 1:

$$x[i] = 1 \quad \text{if} \quad e_i \in X,$$
$$x[i] = 0 \quad \text{if} \quad e_i \notin X.$$

These vectors correspond to the vertices of the n-dimensional cube. A hamiltonian path on this n-cube defines a minimum-change sequence of subsets in which *each subset is derived from its predecessor by adding or removing one element*. Such a sequence is called a *binary Gray code* [6; 7; 21].

The particular sequence which is generated by our algorithm is the *reflected* binary Gray code. For n elements ($n \geq 1$), it is produced in the following way. First, list the sequence for n-1 elements and add 0's as the n-th components. Secondly, list the (n-1)-sequence in reversed order, adding 1's as the n-th components. Obviously, the sequence for 0 elements consists only of the empty configuration.

As an illustration we present the n-cubes for $n = 1,2,3$ and the reflected binary Gray codes for $n = 1,2,3,4$.



n = 1		n = 2		n = 3		n = 4	
1	0	1	00	1	000	1	0000
2	1	2	10	2	100	2	1000
		3	11	3	110	3	1100
		4	01	4	010	4	0100
				5	011	5	0110
				6	111	6	1110
				7	101	7	1010
				8	001	8	0010
						9	0011
						10	1011
						11	1111
						12	0111
						13	0101
						14	1101
						15	1001
						16	0001

In the description, given above, we can replace "0" and "1" by "x_0[n]" and "1-x_0[n]" respectively, where x_0 denotes an arbitrary starting configuration. In this way a more general reflected binary Gray code is obtained. The last configuration in the sequence is adjacent to the first one, since they differ only in their n-th component. It follows that this Gray code constitutes a hamiltonian *circuit* on the n-cube.

If the rules are written down in a more formal way, the following algorithm for enumerating subsets results:

```
procedure brute force mc (problem,n,x); value n,x;
integer n; integer array x; procedure problem;
comment minimum-change enumeration of subsets;
begin
        procedure node(n); value n; integer n;
        begin   if n > 1 then node(n - 1);
                x[n]:= 1 - x[n]; problem(x,n);
                if n > 1 then node(n - 1)
        end;

        problem(x,0); node(n)
end brute force mc;
```

If x_0 and y_0 are adjacent vertices, differing in their last component, then a call

'brute force mc (problem,n,x_0)'

has the following effect:

- A hamiltonian path on the n-cube, starting from x_0 and ending in y_0, is traversed.
- In vertex x_0 a call 'problem(x_0,0)' is made.
- In each vertex x, reached by a change of the k-th component, a call 'problem(x,k)' is made.

The latter two assertions are clear from inspection. To prove the first one, it suffices to show that a call 'node(k)' accomplishes the following:

Starting from a configuration x, all x' for which

$$x' \neq x, \quad x'[l] = x[l] \quad \text{for} \quad k+1 \leq l \leq n$$

are reached, each exactly once, while no other vertices are reached. The

final vertex y is given by

$$y[k] = 1-x[k], \quad y[l] = x[l] \quad \text{for} \quad l \neq k.$$

The proof, which is by induction on k, is clear from the following diagram:

```
                          (x[1],.,x[k-2],   x[k-1],   x[k],x[k+1],.,x[n]) = x
                         {    :       :        :        |     |       |
           node(k-1)     {    v       v        v        v     v       v
                         {(x[1],.,x[k-2],1-x[k-1],   x[k],x[k+1],.,x[n])
node(k) {  x[k]:= 1-x[k]{(x[1],.,x[k-2],1-x[k-1],1-x[k],x[k+1],.,x[n])
                         {    :       :        :        |     |       |
           node(k-1)     {    v       v        v        v     v       v
                         {(x[1],.,x[k-2],   x[k-1],1-x[k],x[k+1],.,x[n]) = y
```

Here a broken arrow means that the component is changed; an unbroken arrow indicates that it remains unchanged.

3. Lattice-points

An n-dimensional lattice is defined by two integer n-vectors l and u. Its vertices are given by the integer n-vectors x for which

$$l[i] \le x[i] \le u[i] \quad \text{for} \quad 1 \le i \le n.$$

The n-cube is a lattice with $l[i] = 0$ and $u[i] = 1$ for $1 \le i \le n$. Correspondingly, an algorithm for the minimum-change enumeration of lattice-points is obtained as a straightforward generalization of 'brute force mc'. *Each vertex is derived from its predecessor by increasing or decreasing exactly one component by one.* However, not each lattice contains a hamiltonian circuit, as can be seen by taking $n = 1$, $l[1] < u[1]+1$ or $n = 2$, $l[i] = 0$, $u[i] = 2$ for $i = 1,2$. So the property that we can start in an arbitrary vertex has been lost.

As an illustration of this method we present two examples in which $n = 4$, $l[i] = 1$ and $u[i] = i, 5-i$ respectively.

i	1	2	3	4	
l[i]	1	1	1	1	
u[i]	1	2	3	4	
1	1	1	1	1	....
2	1	2	1	1	.+..
3	1	2	2	1	..+.
4	1	1	2	1	.-..
5	1	1	3	1	..+.
6	1	2	3	1	.+..
7	1	2	3	2	...+
8	1	1	3	2	.-..
9	1	1	2	2	..-.
10	1	2	2	2	.+..
11	1	2	1	2	..-.
12	1	1	1	2	.-..
13	1	1	1	3	...+
14	1	2	1	3	.+..
15	1	2	2	3	..+.
16	1	1	2	3	.-..
17	1	1	3	3	..+.
18	1	2	3	3	.+..
19	1	2	3	4	...+
20	1	1	3	4	.-..
21	1	1	2	4	..-.
22	1	2	2	4	.+..
23	1	2	1	4	..-.
24	1	1	1	4	.-..

i	1	2	3	4	
l[i]	1	1	1	1	
u[i]	4	3	2	1	
1	1	1	1	1	....
2	2	1	1	1	+...
3	3	1	1	1	+...
4	4	1	1	1	+...
5	4	2	1	1	.+..
6	3	2	1	1	-...
7	2	2	1	1	-...
8	1	2	1	1	-...
9	1	3	1	1	.+..
10	2	3	1	1	+...
11	3	3	1	1	+...
12	4	3	1	1	+...
13	4	3	2	1	..+.
14	3	3	2	1	-...
15	2	3	2	1	-...
16	1	3	2	1	-...
17	1	2	2	1	.-..
18	2	2	2	1	+...
19	3	2	2	1	+...
20	4	2	2	1	+...
21	4	1	2	1	.-..
22	3	1	2	1	-...
23	2	1	2	1	-...
24	1	1	2	1	-...

Our algorithm for generating lattice-points is presented below.

```
procedure brute force lp mc (problem,n,l,u); value n,l,u;
integer n; integer array l,u; procedure problem;
comment minimum-change enumeration of lattice-points;
begin
        procedure node(n); value n; integer n;
        begin   integer dn, ln, un;
                un:= u[n]; u[n]:= ln:= l[n];
                dn:= if ln < un then 1 else -1;
                if n > 1 then node(n - 1);
                for ln:= ln + dn step dn until un do
                begin   l[n]:= ln; problem(l,n,dn);
                        if n > 1 then node(n - 1)
                end
        end;

        problem(l,0,0); node(n)
end brute force lp mc;
```

One can check easily that a call

'brute force lp mc (problem,n,l,u)'

has the following effect:

- A hamiltonian path in the lattice, starting from l, is traversed.
- In vertex l a call 'problem(l,0,0)' is made.
- In each vertex x, reached from y with x[k] ≠ y[k], a call 'problem(x,k,x[k]-y[k])' is made.

4. Combinations

The algorithm, presented in § 2 and generalized in § 3, will now be used to derive a method for enumerating combinations.

A combination C of m out of n elements $e_1, e_2, \ldots, e_n$ is represented by a binary n-vector x:

$$x[i] = 1 \quad \text{if} \quad e_i \in C,$$
$$x[i] = 0 \quad \text{if} \quad e_i \notin C.$$

We define an undirected graph G(n,m) whose vertices are given by these vectors; (x,y) is an edge of G(n,m) iff x and y differ in exactly two components. A hamiltonian path in G(n,m) corresponds to a minimum-change sequence of combinations in which *each combination is derived from its predecessor by adding one element and removing one element.*

From the reflected binary Gray code with the empty set as starting configuration we take the subsequence consisting of those subsets which contain exactly m elements. We prove that this subsequence constitutes a hamiltonian path in G(n,m) from

$$x_0 = (\underbrace{1,\ldots,1,1}_{m},\underbrace{0,\ldots,0,0}_{n-m})$$

to

$$y_0 = (\underbrace{1,\ldots,1}_{m-1},\underbrace{0,0,\ldots,0}_{n-m},1)$$

(note that y_0 and x_0 are adjacent) if $1 \le m \le n-1$; if m = 0 or m = n the path clearly consists of only one vertex.

The proof proceeds by induction on n, the case n = 1 being obvious. For n > 1, $1 \le m \le n-1$, the sequence consists of two parts: first, the quence in G(n-1,m), with 0's added as the n-th components, and secondly, the sequence in G(n-1,m-1) in reversed order, with 1's added as the n-th components. By the induction hypothesis these two parts are hamiltonian paths which look like:

```
for m > 1:   1 ...m-2 m-1  m  m+1...n-2 n-1  n
            (1,...,1,  1,  1,  0,...,0,  0,  0)
            :                                :
           *(1,...,1,  1,  0,  0,...,0,  1,  0)
            (1,...,1,  0,  0,  0,...,0,  1,  1)
            :                                :
            (1,...,1,  1,  0,  0,...,0,  0,  1),
```

for m = 1:

```
   1   2 ...n-2 n-1  n
  (1,  0,...,0,  0,  0)
   :                 :
 *(0,  0,...,0,  1,  0)
  (0,  0,...,0,  0,  1).
```

Inspection shows that the transitions * are edges in G(n,m), so the total sequence is a hamiltonian path, as was to be proved.

As an illustration we present the reflected binary Gray code for n = 5 and its subsequences for $0 \le m \le 5$.

n = 5		m = 0	m = 1	m = 2	m = 3	m = 4	m = 5
1	00000	00000					
2	10000		10000				
3	11000			11000			
4	01000		01000				
5	01100			01100			
6	11100				11100		
7	10100			10100			
8	00100		00100				
9	00110			00110			
10	10110				10110		
11	11110					11110	
12	01110				01110		
13	01010			01010			
14	11010				11010		
15	10010			10010			
16	00010		00010				
17	00011			00011			
18	10011				10011		
19	11011					11011	
20	01011				01011		
21	01111					01111	
22	11111						11111
23	10111					10111	
24	00111				00111		
25	00101			00101			
26	10101				10101		
27	11101					11101	
28	01101				01101		
29	01001			01001			
30	11001				11001		
31	10001			10001			
32	00001		00001				

Combining the recursion scheme of 'brute force mc' (cf. § 2) and the results, presented above, we obtain the following algorithm for enumerating combinations:

```
procedure brute choose mc (problem,n,m); value n,m;
integer n,m; procedure problem;
comment minimum-change enumeration of combinations;
begin   integer k; integer array x[1:n];

        procedure over(n,m); value n,m; integer n,m;
        if n > m ∧ m > 0 then
        begin   integer xn, xk;
                xn:= x[n]; xk:= 1 - xn;
                over(n - 1,m - xn);
                k:= (if m = 1 then n else m) - 1;
                x[n]:= xk; x[k]:= xn;
                if xn = 0 then problem(x,n,k) else problem(x,k,n);
                over(n - 1,m - xk)
        end;

        for k:=     1 step 1 until m do x[k]:= 1;
        for k:= m + 1 step 1 until n do x[k]:= 0;
        problem(x,0,0); over(n,m)
end brute choose mc;
```

A call

'brute choose mc (problem,n,m)'

has the following effect:

- A hamiltonian path in G(n,m), starting from x_0 and ending in y_0, is traversed.
- In vertex x_0 a call 'problem(x_0,0,0)' is made.
- In each vertex x, reached by adding e_k and removing e_l, a call 'problem(x,k,l)' is made.

These assertions are proved along the same lines as those for 'brute force mc'. Note that in the body of 'over(n,m)' the components in positions n and (<u>if</u> m = 1 <u>then</u> n <u>else</u> m) - 1 are changed; this corresponds to the transitions * in the diagrams, given above.

Another minimum-change method for generating combinations has been proposed by Chase [4] and Ehrlich [5]. Still another method has been suggested by Wells [21,Ch.5.1,ex.7].

A minimum-change sequence for combinations of $m_1 \le m \le m_2$ out of n elements in which *each combination is derived from its predecessor by adding one element and/or removing one element*, is given by the subsequence of the reflected binary Gray code consisting of those subsets which contain $m_1 \le m \le m_2$ elements. The construction of a recursive algorithm for enumerating these configurations is left as a challenge to the reader.

5. Permutations

We next consider the minimum-change enumeration of all permutations of n different elements. An n-permutation is defined as an n-vector whose components are these elements in some order.

We define an undirected graph G(n) whose vertices are given by the n! n-permutations; (x,y) is an edge of G(n) iff x and y differ only in two neighbouring positions. A hamiltonian path in G(n) corresponds to a minimum-change sequence of permutations in which *each permutation is derived from its predecessor by transposing two elements in adjacent positions.*

Denoting the n elements by 1,2,...,n, we can construct such a sequence inductively as follows. For n = 1, it consists of the only 1-permutation. Let the sequence for (n-1)-permutations be given. Placing n at the right of the first (n-1)-permutation, we obtain the first n-permutation. The n-1 next ones are obtained by successively interchanging n with its left neighbour. After that, n is found at the left of the first (n-1)-permutation, which remained unchanged. Replacing this (n-1)-permutation by its successor in the (n-1)-sequence gives us the (n+1)-th n-permutation, and the n-1 next ones arise from successive transpositions of n with its right neighbour. Then n is found at the right of the second (n-1)-permutation, which now is replaced by the third one, and the process starts all over again.

As an illustration of this method we present the sequences and the graphs for n = 1,2,3,4. We note that G(4) is the edge graph of a solid truncated octahedron, replicas of which fill entire 3-space. Analogous statements hold for all n.

n = 1		n = 2		n = 3	
1	1	1	1 2	1	1 2 3
		2	2 1	2	1 3 2
				3	3 1 2
				4	3 2 1
				5	2 3 1
				6	2 1 3

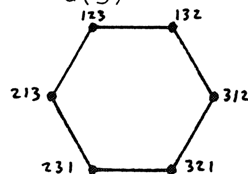
G(1)

1

G(2)

12 21

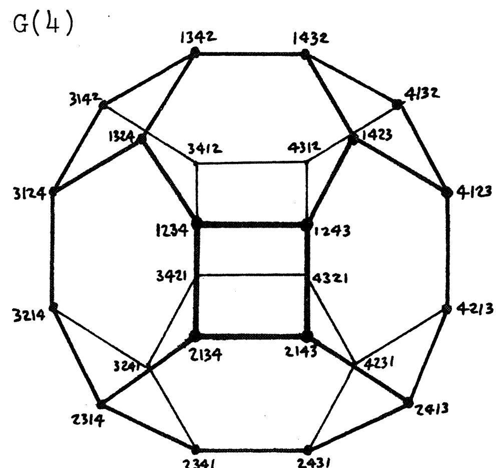
G(3)



n = 4

1	1 2 3 4	13	4 3 2 1
2	1 2 4 3	14	3 4 2 1
3	1 4 2 3	15	3 2 4 1
4	4 1 2 3	16	3 2 1 4
5	4 1 3 2	17	2 3 1 4
6	1 4 3 2	18	2 3 4 1
7	1 3 4 2	19	2 4 3 1
8	1 3 2 4	20	4 2 3 1
9	3 1 2 4	21	4 2 1 3
10	3 1 4 2	22	2 4 1 3
11	3 4 1 2	23	2 1 4 3
12	4 3 1 2	24	2 1 3 4

G(4)



The following algorithm generates the permutations in the order described above:

```
procedure brute permute mc (problem,n,x); value n,x;
integer n; array x; procedure problem;
comment minimum-change enumeration of permutations;
begin   real xk; integer k, l, q; integer array d[1:n];

        procedure node(i); value i; integer i;
        begin   integer di, ti, ui, xi;
                di:= d[i]; if di = 1 then
                begin   ti:= 1; ui:= i - 1; q:= q - 1
                end             else
                begin   ti:= i; ui:= 2
                end;
                xi:= x[q + ti];
                if i < n then node(i + 1);
                for ti:= ti step di until ui do
                begin   k:= q + ti; l:= k + di;
                        x[k]:= xk:= x[l]; x[l]:= xi;
                        if di = 1 then problem(x,k,xk,xi)
                                  else problem(x,l,xi,xk);
                        if i < n then node(i + 1)
                end;
                d[i]:= -di; if di = -1 then q:= q + 1
        end;

        for k:= 1 step 1 until n do d[k]:= -1; q:= 0;
        problem(x,0,0,0); if n ≥ 2 then node(2)
end brute permute mc;
```

If $\{x_0[1],\dots,x_0[n]\}$ is the n-set to be permuted, then a call

'brute permute mc (problem,n,x_0)'

has the following effect:

If n = 1, then a call 'problem(x_0,0,0,0)' is made, and else

- A hamiltonian path in G(n) is traversed, starting from x_0 and ending in y_0 such that

$$y_0[1] = x_0[2],\ y_0[2] = x_0[1],\ y_0[k] = x_0[k] \text{ for } 3 \le k \le n.$$

- In vertex x_0 a call 'problem(x_0,0,0,0)' is made.
- In each vertex x, reached by transposition of the elements in positions k and k+1, a call 'problem(x,k,x[k],x[k+1])' is made.

The latter two assertions are clear from inspection. The proof of the first one may be left to the reader. As a hint, we note that just before a call 'node(i)' and immediately after its execution, x, d and q satisfy the following conditions:

$\{j \mid i \le j \le n,\ d[j] = 1\}$ has exactly q elements, and if we index them such that $j_1 > j_2 > \dots > j_q$, then $x[k] = x_0[j_k]$ for $1 \le k \le q$, and if $\{j \mid i \le j \le n,\ d[j] = -1\} = \{j'_1,\dots,j'_r\}$, where $j'_1 > j'_2 > \dots > j'_r$, $r+q = n-i+1$, then $x[n+1-k] = x_0[j'_k]$ for $1 \le k \le r$.

Using the variabele q to determine the place of the transpositions is more efficient than keeping track of the inverse permutation for that purpose (cf. [5]).

Generation of permutation sequences has received much attention in the literature.

The algorithm for enumeration by adjacent transposition, presented above, was discovered independently by Trotter [19] and Johnson [9]. A different minimum-change method was found by Wells [20]; Boothroyd gives recursive [1] and iterative [2; 3] ALGOL 60-procedures for this algorithm.

Methods for generating permutations are surveyed by Lehmer [11], Ord-Smith [15] and Wells [21,Ch.5.2]. Ord-Smith [16] presents a time comparison between six algorithms, including three minimum-change procedures [19; 2; 3].

We make one final observation.

Let an undirected graph $H(n)$ on n vertices be given. Define an undirected graph $G_H(n)$ on the set of n-permutations, by drawing an edge between x and y iff x can be obtained from y by a single transposition of the elements in positions k and l, where (k,l) is an edge of H(n). One can prove the following:

> $G_H(n)$ contains a hamiltonian circuit if and only if H(n) contains a spanning tree.

The "only if"-part is obvious; the "if"-part follows by an inductive argument.

In the Johnson-Trotter algorithm, discussed above, the "transposition graph" H(n) is the tree which looks like:

The transposition graph of the Wells algorithm contains the above one properly.

6. Computations

In this paragraph we give computer results for the algorithms presented in this report. We include also results for three algorithms which have been published;
these are

- Chase's algorithm acm 382 [4], which implements a minimum-change method for generating combinations, different from ours (cf. § 4).
- Trotter's algorithm acm 115 [19;16] which generates permutations by adjacent transposition (cf. § 5). For a number of years it remained the fastest permutation procedure.
- Boothroyd's algorithm bcj 30 [3;16] which implements Wells' method for generating permutations by transposition [20](cf. § 5). Ord-Smith [16] found this procedure to be the fastest of six published permutation algorithms.

These algorithms have been modified slightly in order to make a fair comparison; see appendix C for details.

The procedures have been tested on the Electrologica X8-computer of the Mathematisch Centrum. When making time comparisons, we chose for the actual parameter, corresponding to the formal parameter 'problem', a procedure with an empty body; its declaration reads '<u>procedure</u> empty (x...);;' with the appropriate number of formal parameters.

The results are given in table 1. It is surprising that the Trotter algorithm is slower than our truly brute lexicographic method for generating permutations. As for minimum-change enumeration, it seems advantageous to use recursive algorithms. This may be explained by the discussion in § 1.

Very fast PL/1 procedures for generating various types of combinatorial configurations have been announced by Ehrlich [5]. It will be interesting to compare these "loopless" algorithms with our recursive ones, using the same programming language and the same computer.

ENUMERATION OF	seconds					configurations/second				
SUBSETS	brute force					brute force				
n	lex	mc	mc b			lex	mc	mc b		
14	26.8	19.5	18.5			611	839	887		
LATTICE-POINTS	brute force lp					brute force lp				
n l_i u_i	lex	mc	mc b			lex	mc	mc b		
14 0 1	46.7	28.7	26.3			351	571	623		
7 1 i	20.8	12.6	10.6			242	399	475		
7 1 8-i	7.4	4.9	4.4			682	1020	1145		
COMBINATIONS	brute choose			acm		brute choose			acm	
n m	lex	mc	mc b	382b		lex	mc	mc b	382b	
14 4	3.1	2.6	1.3	3.0		327	379	782	338	
14 7	10.1	9.0	5.3	9.7		341	380	643	356	
14 10	3.1	2.6	1.8	2.9		325	379	550	349	
14 ≥0,≤14	48.9	43.4	25.5	46.8						
PERMUTATIONS	brute permute			acm	bcj	brute permute			acm	bcj
n	lex	mc	mc b	115b	30b	lex	mc	mc b	115b	30b
8	97.6	58.9	51.4	101.0	75.6	413	685	785	399	534

Table 1 Computer results for 15 algorithms.

--- lex : lexicographic enumeration, see appendix A.

--- mc : minimum-change enumeration, see §§ 2-5.

--- mc b : faster minimum-change enumeration, see appendix B.

acm 382b : Chase [4], see appendix C.

acm 115b : Trotter [19;16], see appendix C.

bcj 30b : Boothroyd [3;16], see appendix C.

7. Applications

The enumeration algorithms may be applied to optimization problems in two ways. First, by generating and evaluating each feasible solution to a problem, one obtains an optimal solution. Secondly, one can try to improve upon a given solution by checking a limited set of local changes. If such a change in the solution proves to be advantageous, one starts anew, proceeding from the improved solution. A locally optimal solution has been obtained as soon as the entire set of changes has been enumerated unsuccessfully. While the former method yields optimal solutions to small problems only, the latter enables us to solve "real" problems in a suboptimal but often satisfactory way.

Some examples of each of these approaches are given below. In every case, the minimum-change character of the enumeration should be exploited (cf. § 1).

Our procedure 'brute force mc' can be used to enumerate the solutions to *0-1 programming problems*. Krol [10] reports that for small problems of this type, explicit enumeration surpasses several methods based on implicit enumeration. His use of a lexicographic method and his unability to describe it raised our interest in the present subject.

Similarly, 'brute force lp mc' can solve small *integer programming problems*.

By explicit enumeration of permutations one can solve *scheduling problems* P of the form

$$\min_x z_P(x)$$

where $x = (x[1],\dots,x[n])$ runs through all n! permutations. An example is the *quadratic assignment problem* (QAP) [14,Ch.8]:

$$z_{QAP}(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{x[i]x[j]}d_{ij}$$

where c and d are non-negative n×n-matrices. A special case of the QAP is obtained if we define

$$d_{ij} = 1 \quad \text{for} \quad i > j,$$
$$d_{ij} = 0 \quad \text{for} \quad i \le j.$$

It is called the *acyclic subgraph problem* (ASP) [12; 14,Ch.8.4.1]:

$$z_{ASP} = \sum_{i=1}^{n}\sum_{j=1}^{i-1} c_{x[i]x[j]}.$$

Another choice of d:

$$d_{i\ i+1} = 1 \quad \text{for} \quad 1 \le i \le n-1,$$
$$d_{n1} = 1,$$
$$d_{ij} = 0 \quad \text{otherwise},$$

leads to the well-known *travelling-salesman problem* (TSP) [14,Ch.6,8.4.2]:

$$z_{TSP}(x) = \sum_{i=1}^{n-1} c_{x[i]x[i+1]} + c_{x[n]x[1]}.$$

A *symmetric* TSP is characterized by $c_{ij} = c_{ji}$ for $1 \le i,j \le n$.

We define the *reflection* $\bar{x}$ and the *rotations* x_k of a permutation x by

$$\bar{x} = (x[n],x[n-1],\dots,x[2],x[1]),$$
$$x_k = (x[k+1],\dots,x[n],x[1],\dots,x[k]) \quad \text{for } 0 \le k \le n-1.$$

One easily proves

$$z_{ASP}(\bar{x}) = \sum_{i \ne j} c_{ij} - z_{ASP}(x),$$
$$z_{TSP}(\bar{x}) = z_{TSP}(x) \quad \text{for a symmetric TSP},$$
$$z_{TSP}(x_k) = z_{TSP}(x) \quad \text{for } 0 \le k \le n-1.$$

It follows that, when we attack a symmetric TSP or an ASP by brute force, it suffices to enumerate a *reflection-free* set of permutations (cf. [12, § 10]). Further, when solving a TSP, we can fix one of the components of x, say x[n], and permute only the elements x[1],...,x[n-1].

If the minimum-change algorithm, discussed in § 5, is used to generate all permutations of a given arrangement $x_0 = (x_0[1],\dots,x_0[n])$, then the elements $x_0[1]$ and $x_0[2]$ are transposed half-way. Let $x = (..x_0[1]..x_0[2].)$ be a permutation which is generated before this transposition. Its reflection $\bar{x} = (.x_0[2]..x_0[1]..)$ occurs in the second half of the enumeration.

It follows that the first n!/2 arrangements form a reflection-free set (cf. [18]).

In general, the n!/(m-1)! permutations in which the original order of the elements x[1],...,x[m-1] is preserved, can be enumerated using a simple modification of 'brute permute mc':

```
procedure brute permute m mc (problem,n,m,x);...;
begin   ...
        ...; if n ≥ m then node(m)
end brute permute m mc;
```

The above discussion shows that QAPs, ASPs and TSPs may be solved by calls of the form

'brute permute mc (qap,n,x)',
'brute permute m mc (asp,n,3,x)',
'brute permute m mc (tsp,n-1,<u>if</u> sym <u>then</u> 3 <u>else</u> 2,x)',

where 'qap', 'asp' and 'tsp' are procedures which compute the cost changes occurring in the QAP, ASP and TSP, respectively.

The n!/2 solutions which are checked in an ASP correspond to the hamiltonian paths in a complete directed graph. The (n-1)!/2 solutions to a symmetric TSP are the hamiltonian circuits in a complete undirected graph; they are called *rosary permutations* [8; 17; 18].

Finally, we indicate some examples of the second approach. These are sub-optimal methods, based on the enumeration of all combinations of m elements out of n, so we are dealing with applications of 'brute choose mc'. For practical purposes, it is advisable to construct special versions of 'brute choose mc' for fixed m, using a set of m nested <u>for</u>-loops. Further, in each application the calls of the procedure 'problem' should be replaced by its actual body.

A solution x to the ASP is said to be *relatively optimal* [12,§ 9] if

$$\begin{aligned} &\textstyle\sum_{i=j+1}^{k} (c_{x[j]x[i]} - c_{x[i]x[j]}) \geq 0 \\ &\textstyle\sum_{i=j}^{k-1} (c_{x[i]x[k]} - c_{x[k]x[i]}) \geq 0 \end{aligned} \qquad \text{for } 1 \leq j,k \leq n.$$

Such a solution can be constructed by enumeration of all pairs (j,k) with $1 \leq j,k \leq n$. This can be done efficiently with a special version of 'brute choose mc' for m = 2. In the phase of verification, when no further improvement is found, this method checks each element of the matrix c exactly once.

A solution x to the TSP is called *m-opt* if it is impossible to obtain a solution with smaller cost by replacing m of its links (x[i],x[i+1]) by a different set of m links [13; 14,Ch.6.6.2]. A 3-opt method, based on the m = 3-version of 'brute choose mc', turns out to be more efficient than the algorithm presented by Lin [13].

Similarly, suboptimal solutions to the QAP can be obtained by exchanging elements instead of links [14,Ch.8.3.2].

This approach might be applicable also to other types of complex optimization problems.

Appendix A Algorithms for lexicographic enumeration

A lexicographic enumeration method generates the configurations x in such a way that the number x[n] x[n-1] ... x[2] x[1] is increasing. Note that this is a binary number for subsets, a mixed-radix number for lattice-points, etc.

The reader will have no difficulty in fathoming the algorithms for lexicographic enumeration, presented in this appendix. They are even more simple than the minimum-change algorithms, and they are constructed in the same way. We indicate the following main differences:

- The array x in which the configuration is stored, is always declared within the procedurebody.
- At each level of recursion exactly one component of x is defined.
- The procedure 'problem' is called when the configuration has been completed, i.e. at the bottom of the recursion.
- The sole parameter of the procedure 'problem' is the array x. It is of no use to include here the positions in which x differs from the preceding configuration.

```
procedure brute force lex (problem,n); value n;
integer n; procedure problem;
comment lexicographic enumeration of subsets;
begin   integer array x[1:n];

        procedure node(n); value n; integer n;
        if n = 0 then problem(x) else
        for x[n]:= 0, 1 do node(n - 1);

        node(n)
end brute force lex;


procedure brute force lp lex (problem,n,l,u); value n;
integer n; integer array l,u; procedure problem;
comment lexicographic enumeration of lattice-points;
begin   integer array x[1:n];

        procedure node(n); value n; integer n;
        if n = 0 then problem(x) else
        begin   integer un, m;
                un:= u[n]; m:= n - 1;
                for x[n]:= l[n] step 1 until un do node(m)
        end;

        node(n)
end brute force lp lex;


procedure brute choose lex (problem,n,m); value n,m;
integer n,m; procedure problem;
comment lexicographic enumeration of combinations;
begin   integer array x[1:n];

        procedure over(n,m); value n,m; integer n,m;
        if m = 0 then bottom(n,0) else
        if m = n then bottom(n,1) else
        begin   x[n]:= 0; over(n - 1,m);
                x[n]:= 1; over(n - 1,m - 1)
        end;

        procedure bottom(n,d); value n,d; integer n,d;
        begin   for n:= n step -1 until 1 do x[n]:= d;
                problem(x)
        end;

        over(n,m)
end brute choose lex;
```

```
procedure brute permute lex (problem,n); value n;
integer n; procedure problem;
comment lexicographic enumeration of permutations;
begin   integer h; integer array x[1:n];

        procedure node(n); value n; integer n;
        if n = 1 then problem(x) else
        begin   integer k;
                node(n - 1);
                for k:= n - 1 step -1 until 1 do
                begin   h:= x[k]; x[k]:= x[n]; x[n]:= h;
                        node(n - 1)
                end;
                h:= x[n];
                for k:= n step -1 until 2 do x[k]:= x[k - 1];
                x[1]:= h
        end;

        for h:= n step -1 until 1 do x[h]:= n + 1 - h;
        node(n)
end brute permute lex;
```

Appendix B Faster algorithms for minimum-change enumeration

The following two considerations enable us to speed up the algorithms, discussed in §§ 2-5.

First, we note that each of these algorithms contains one recursive procedure which handles two types of changes simultaneously, e.g. increasing or decreasing the value of a component, or transposing an element and its left or right neighbour. This procedure can be split up in two procedures, each handling one type of change, and calling themselves and each other.

Secondly, one can obtain faster algorithms by explicitly writing out the deepest level of recursion. This clearly reduces the number of checks if the bottom of the recursion has been reached already. This device enables us also to deal separately with those elements which are involved in a considerable part of the changes. For example, in 'brute force mc' half of the changes occur in the first position, and in 'brute permute mc' the n-th element is transposed in $(n-1)/n$ of the cases. Also, in 'brute choose mc' the case $m = 1$ deserves a special treatment.

Following these lines, we can easily construct faster algorithms for minimum-change enumeration. They are presented below. Some minor differences are indicated if necessary. We emphasize the point that, in each case, the speeded-up algorithm is equivalent to the original one, in the sense that the same sequence of successive change positions is generated. Both methods can be applied for all $n \geq 1$.

```
procedure brute force mc b (problem,n); value n;
integer n; procedure problem;
comment minimum-change enumeration of subsets;
begin   integer k; integer array x[1:n];

        procedure rise(n); value n; integer n;
        if n = 1 then
        begin   x[1]:= 1; problem(x,1,1)
        end      else
        begin   rise(n - 1);
                x[n]:= 1; problem(x,n,1);
                fall(n - 1)
        end;

        procedure fall(n); value n; integer n;
        if n = 1 then
        begin   x[1]:= 0; problem(x,n,-1)
        end      else
        begin   rise(n - 1);
                x[n]:= 0; problem(x,n,-1);
                fall(n - 1)
        end;

        for k:= 1 step 1 until n do x[k]:= 0;
        problem(x,0,0); rise(n)
end brute force mc b;
```

We note two additional points of difference with 'brute force mc':

- The array x is declared within the procedure body, and initiated by $x[k]:= 0$, $1 \le k \le n$.
- The procedure'problem' has a third parameter, which equals 0 after initialization, and +1 (-1) after an element has been added (removed).

```
procedure brute force lp mc b (problem,n,l,u); value n;
integer n; integer array l,u; procedure problem;
comment minimum-change enumeration of lattice-points;
begin   integer k, x1, l1, u1;
        boolean array even[1:n]; integer array x[1:n];

        procedure rise(n); value n; integer n;
        if n = 1 then
        begin   for x1:= l1 + 1 step 1 until u1 do
                begin   x[1]:= x1; problem(x,1,1)
                end
        end      else
        begin   boolean rm; integer xn, un, m;
                un:= u[n]; m:= n - 1;
                rm:= true; rise(m);
                for xn:= l[n] + 1 step 1 until un do
                begin   x[n]:= xn; problem(x,n,1);
                        rm:= ¬rm; if rm then rise(m) else fall(m)
                end
        end;

        procedure fall(n); value n; integer n;
        if n = 1 then
        begin   for x1:= u1 - 1 step -1 until l1 do
                begin   x[1]:= x1; problem(x,1,-1)
                end
        end      else
        begin   boolean rm; integer xn, ln, m;
                ln:= l[n]; m:= n - 1;
                rm:= even[n]; if rm then rise(m) else fall(m);
                for xn:= u[n] - 1 step -1 until ln do
                begin   x[n]:= xn; problem(x,n,-1);
                        rm:= ¬rm; if rm then rise(m) else fall(m)
                end
        end;

        for k:= 2 step 1 until n do
        begin   x[k]:= l1:= l[k]; u1:= u[k] - l1;
                even[k]:= (u1:2) × 2 ≠ u1
        end;    x[1]:= l1:= l[1]; u1:= u[1];
        problem(x,0,0); rise(n)
end brute force lp mc b;
```

```
procedure brute choose mc b (problem,n,m); value n,m;
integer n,m; procedure problem;
comment minimum-change enumeration of combinations;
begin   integer k; integer array x[1:n];

        procedure over(n,m); value n,m; integer n,m;
        if n = m then else
        if m > 1 then
        begin   over(n - 1,m);
                x[n]:= 1; x[m - 1]:= 0; problem(x,n,m - 1);
                revo(n - 1,m - 1)
        end           else
        for m:= 2 step 1 until n do
        begin   x[m]:= 1; x[m - 1]:= 0; problem(x,m,m - 1)
        end;

        procedure revo(n,m); value n,m; integer n,m;
        if n = m then else
        if m > 1 then
        begin   over(n - 1,m - 1);
                x[n]:= 0; x[m - 1]:= 1; problem(x,m - 1,n);
                revo(n - 1,m)
        end           else
        for m:= n step -1 until 2 do
        begin   x[m]:= 0; x[m - 1]:= 1; problem(x,m - 1,m)
        end;

        for k:=     1 step 1 until m do x[k]:= 1;
        for k:= m + 1 step 1 until n do x[k]:= 0;
        problem(x,0,0); if m > 0 then over(n,m)
end brute choose mc b;
```

```
procedure brute permute mc b (problem,n); value n;
integer n; procedure problem;
comment minimum-change enumeration of permutations;
begin   integer h, k, l, q; integer array x[0:n];

        procedure regs(i); value i; integer i;
        if i = n then
        begin   q:= 0;
                for l:= 2 step 1 until i do
                begin   k:= l - 1; x[k]:= h:= x[l]; x[l]:= i;
                        problem(x,k,h,i)
                end
        end      else
        begin   boolean rj; integer ti, j;
                q:= q - 1;
                j:= i + 1;
                rj:= x[q] = j; if rj then regs(j) else linx(j);
                for ti:= 2 step 1 until i do
                begin   l:= q + ti;
                        k:= l - 1; x[k]:= h:= x[l]; x[l]:= i;
                        problem(x,k,h,i);
                        rj:= ¬rj; if rj then regs(j) else linx(j)
                end
        end;

        procedure linx(i); value i; integer i;
        if i = n then
        begin   for l:= i step -1 until 2 do
                begin   k:= l - 1; x[l]:= h:= x[k]; x[k]:= i;
                        problem(x,k,i,h)
                end;
                q:= 1
        end      else
        begin   boolean rj; integer ti, j;
                j:= i + 1;
                rj:= x[q] = j; if rj then regs(j) else linx(j);
                for ti:= i step -1 until 2 do
                begin   l:= q + ti;
                        k:= l - 1; x[l]:= h:= x[k]; x[k]:= i;
                        problem(x,k,i,h);
                        rj:= ¬rj; if rj then regs(j) else linx(j)
                end;
                q:= q + 1
        end;

        for k:= 0 step 1 until n do x[k]:= k; q:= 0;
        problem(x,0,0,0); if n ≥ 2 then linx(2)
end brute permute mc b;
```

There is one additional point of difference with 'brute permute mc':

- The array x is declared within the procedurebody, and initialized by $x[k] := k$, $1 \le k \le n$.

This is exploited by noting that, in 'regs(i)' or 'linx(i)' the element i+1 is waiting at the left iff $x[q] = i+1$, where q, as usually, equals the number of elements j ($j > i$) waiting at the left. So we dispense with the array d which indicated the type (direction) of the paths.

Appendix C The algorithms of Chase, Trotter and Boothroyd

This appendix contains the text of the three previously published algorithms for which computer results are given in § 6.

Originally, these procedures are organized as is usual in the literature, i.e. each call generates the next configuration in the sequence. We define new procedures each of which contains the declaration and a series of calls of the original procedure. Of course, just before these successive calls it is the right moment to initialize the necessary auxiliary variables and arrays.

The procedures, obtained in this way, generate all configurations after each call and are comparable to our algorithms. Inspection will reveal some additional minor modifications.

```
procedure acm 382b (problem,n,m); value n,m;
integer n,m; procedure problem;
comment chase, c.acm 13(1970)368;
begin   boolean busy; integer x, y; integer array b[0:n], p[0:n + 1];

        procedure twiddle(x,y); integer x,y;
        begin   integer i, j, k;
                j:= 0;
        l1:     j:= j + 1; if p[j] ≤ 0 then goto l1;
                if p[j - 1] = 0 then
                begin   for i:= j - 1 step -1 until 2 do p[i]:= -1;
                        p[j]:= 0; p[1]:= x:= 1; y:= j; goto l4
                end;
                if j > 1 then p[j - 1]:= 0;
        l2:     j:= j + 1; if p[j] > 0 then goto l2;
                i:= k:= j - 1;
        l3:     i:= i + 1; if p[i] = 0 then begin p[i]:= -1; goto l3 end;
                if p[i] = -1 then
                begin p[i]:= p[k]; x:= i; y:= k; p[k]:= -1; goto l4 end;
                if i = p[0] then begin busy:= false; goto l4 end;
                p[j]:= p[i]; p[i]:= 0; x:= j; y:= i;
        l4:
        end twiddle;

        y:= n - m;
        for x:= 1 step 1 until y do b[x]:= p[x]:= 0;
        for x:= y + 1 step 1 until n do begin b[x]:= 1; p[x]:= x - y end;
        y:= n + 1;
        p[0]:= y; p[y]:= -2; if m = 0 then p[1]:= 1;
        x:= y:= 0; busy:= true;
next:   b[x]:= 1; b[y]:= 0;
        problem(b,x,y); twiddle(x,y); if busy then goto next
end acm 382b;
```

```
procedure acm 115b (problem,n,x); value n,x;
integer n; array x; procedure problem;
comment trotter, c.acm 5(1962)434-435.
        ord-smith, comput.j. 14(1971)136-139;
begin   boolean busy; real s, t; integer q; integer array p, d[2:10];

        procedure perm(x,n); value n; integer n; array x;
        begin   integer k;
                k:= 0;
        index:  p[n]:= q:= p[n] + d[n];
                if q = n then begin d[n]:= -1; goto loop end;
                if q ≠ 0 then goto transpose;
                d[n]:= 1; k:= k + 1;
        loop:   if n > 2 then begin n:= n - 1; goto index end;
                q:= 1; busy:= false;
        transpose: q:= q + k; k:= q + 1;
                t:= x[q]; x[q]:= s:= x[k]; x[k]:= t
        end perm;

        for q:= 2 step 1 until n do begin p[q]:= 0; d[q]:= 1 end;
        q:= 0; s:= t:= 0; busy:= true;
next:   problem(x,q,s,t); perm(x,n); if busy then goto next
end acm 115b;
```

```
procedure bcj 30b (problem,n,x); value n,x;
integer n; array x; procedure problem;
comment boothroyd, comput.j. 10(1967)311.
        ord-smith, comput.j. 14(1971)136-139;
begin   boolean busy; real x1, x2, x3, x4; integer i;
        integer array d[5:10];

        procedure perm(x,n); value n; integer n; array x;
        begin   real xk; integer j, k, kless1, dk;
                switch s:= s1,s2,s1,s2,s1,s3,s1,s2,s1,s2,s1,s3,
                           s1,s2,s1,s2,s1,s4,s1,s2,s1,s2,s1,s5;
                switch ss:= ss1,ss2,ss3,ss4;
                i:= i + 1; goto s[i];
        s1:     xk:= x1; x1:= x[1]:= x2; x2:= x[2]:= xk; goto exit;
        s2:     xk:= x2; x2:= x[2]:= x3; x3:= x[3]:= xk; goto exit;
        s3:     xk:= x3; x3:= x[3]:= x4; x4:= x[4]:= xk; goto exit;
        s4:     xk:= x4; x4:= x[4]:= x1; x1:= x[1]:= xk; goto exit;
        s5:     kless1:= 4; k:= 5; i:= 0;
        count:  dk:= d[k]; if dk ≠ kless1 then goto swap;
                d[k]:= 0; if k ≠ n then
                begin kless1:= k; k:= k + 1; goto count end;
                busy:= false; goto exit;
        swap:   dk:= d[k]:= dk + 1; if dk > 2 then
                begin if k - k : 2 × 2 = 0 then kless1:= k - dk end;
                xk:= x[k]; x[k]:= x[kless1]; x[kless1]:= xk;
                goto if kless1 < 4 then ss[kless1] else exit;
        ss1:    x1:= xk; goto exit;
        ss2:    x2:= xk; goto exit;
        ss3:    x3:= xk; goto exit;
        ss4:    x4:= xk;
exit:   end perm;

        for i:= 5 step 1 until n do d[i]:= 0;
        i:= 0; x1:= x[1]; x2:= x[2]; x3:= x[3]; x4:= x[4]; busy:= true;
next:   problem(x); perm(x,n); if busy then goto next
end bcj 30b;
```

Literature

1. J.BOOTHROYD, Algorithm 6, Perm, *Comput.Bull.* 9(1965)104.
2. J.BOOTHROYD, Algorithm 29, Permutation of the elements of a vector, *Comput.J.* 10(1967)311.
3. J.BOOTHROYD, Algorithm 30, Fast permutation of the elements of a vector, *Comput.J.* 10(1967)311-312.
4. P.J.CHASE, Algorithm 382, Combinations of *M* out of *N* objects, *Comm.ACM* 13(1970)368.
5. G.EHRLICH, Loopless algorithms for generating permutations, combinations, and other combinatorial configurations, *J.ACM* 20(1973)500-513.
6. M.GARDNER, The curious properties of the Gray code and how it can be used to solve puzzles, *Sci.Amer.* 227.2(August 1972)106-109.
7. E.N.GILBERT, Gray codes and paths on the *n*-cube, *Bell System Techn.J.* 37(1958)815-826.
8. K.HARADA, Generation of rosary permutations expressed in hamiltonian circuits, *Comm.ACM* 14(1971)373-379.
9. S.M.JOHNSON, Generation of permutations by adjacent transposition, *Math.Comp.* 17(1963)282-285.
10. G.KROL, *Het gemillimeterde hoofd,* 133-138, Amsterdam,1967.
11. D.H.LEHMER, The machine tools of combinatorics, in: E.F.BECKENBACH(ed.), *Applied combinatorial mathematics,* New York,1964,5-31.
12. H.W.LENSTRA,JR., The acyclic subgraph problem, Report BW 26/73, Mathematisch Centrum, Amsterdam, 1973.
13. S.LIN, Computer solutions of the traveling salesman problem, *Bell System Tech.J.* 44(1965)2245-2269.
14. H.MÜLLER-MERBACH, *Optimale Reihenfolgen,* Berlin etc.,1970.
15. R.J.ORD-SMITH, Generation of permutation sequences: Part 1, *Comput.J.* 13(1970)152-155.
16. R.J.ORD-SMITH, Generation of permutation sequences: Part 2, *Comput.J.* 14(1971)136-139.
17. R.C.READ, A note on the generation of rosary permutations, *Comm.ACM* 15(1972)775.
18. M.K.ROY, Reflection-free permutations, rosary permutations, and adjacent transposition algorithms, *Comm.ACM* 16(1973)312-313.
19. H.F.TROTTER, Algorithm 115, Perm, *Comm.ACM* 5(1962)434-435.
20. M.B.WELLS, Generation of permutations by transposition, *Math.Comp.* 15 (1961)192-195.
21. M.B.WELLS, *Elements of combinatorial computing,* Oxford etc.,1971.