Algorithms

J. G. HERRIOT, Editor

ALGORITHM 305
SYMMETRIC POLYNOMIALS [C1]

P. Bratley and J. K. S. McKay (Recd. 23 Sept. 1966, 15 Feb. 1967 and 10 Mar. 1967)
Department of Computer Science, University of Edinburgh, Edinburgh, Scotland

```
real procedure express(b, unit, n);  value n;  integer n;
integer array b;  array unit;
```

comment *express* expresses the symmetric sum $\sum x_{i_1}^{b_1} x_{i_2}^{b_2} \cdots x_{i_n}^{b_n}$ over n variables as a sum of determinants in the unitary symmetric functions $\sum x_{i_1} x_{i_2} x_{i_3} \cdots x_{i_r}$. The non-negative exponents b_i $(i = 1, \cdots, n)$ are assumed to be in $b[1{:}n]$ on entry to *express*. (The elements of this array are altered by the procedure.) The symmetric sum is first expressed in terms of Schur functions which are then evaluated as determinants in the unitary symmetric functions. The Schur functions are generated in the local array $c[1{:}i]$ with the sign in the local integer *sig*. The unitary functions of degree $r = 1, \cdots, n$ should be in *unit*$[1{:}n]$ on entry to *express*.

This procedure may be used to determine the coefficients of a polynomial with roots the kth (k a positive integer) powers of the roots of a given monic polynomial. Use is made of the procedures *determinant* [Algorithm 224, *Comm. ACM 12* (Apr. 1964), 243)] and *perm* [Algorithm 306, *Comm. ACM 10* (July 1967), 450]

References:

1. Littlewood, D. E. *The Theory of Group Characters*. Clarendon Press, Oxford, England 1958, 2nd ed., Ch. 6.
2. McKay, J. K. S. On the representation of symmetric polynomials. *Comm. ACM 10* (July 1967), 428–429;

```
begin integer array c, d[1:n];
  integer sig, p, q, i, j;  Boolean finish;  real sigma;
  procedure sort (x, c, n);  value n;  integer c, n;
    integer array x;
  comment  sorts the integer array x[1:n] into descending order.
    c is set to ±1 according to whether the number of transpositions made is even or odd;
  begin integer i, j, k;
    c := 1;
L4: i := 1;  k := 0;  j := x[1];
L1: i := i + 1;  if i > n then go to L3;
    if x[i] ≤ j then
      begin x[i−1] := j;  j := x[i] end
    else begin x[i−1] := x[i];  k := 1;  c := −c end;
    go to L1;
L3: x[n] := j;  if k ≠ 0 then go to L4
  end sort;
  procedure conjugate(p, long1, q, long2);  value long1;
    integer array p, q;  integer long1, long2;
  comment  conjugate forms in q[1:long2] the partition conjugate to that in p[1:long1];
  begin
    integer r, i, j;
    long2 := 0;
    for r := long1 step −1 until 1 do
    begin i := if r = long1 then p[r] else p[r] − p[r+1];
      for j := 1 step 1 until i do
        begin long2 := long2 + 1;  q[long2] := r end
    end
  end conjugate;
  finish := true;  sigma := 0;
  sort (b, sig, n);
  if b[1] = 0 then begin sigma := 1;  go to L99 end;
L3: perm (b, n, finish);
  if finish then go to L99;
  for i := 1 step 1 until n do
  begin c[i] := b[i] + n − i;
    for j := 1 step 1 until i − 1 do
    if c[i] = c[j] then go to L3
  end;
  sort (c, sig, n);
  for i := 1 step 1 until n do
  begin c[i] := c[i] + i − n;
    if c[i] = 0 then
      begin i := i − 1;  go to L7 end
  end;
  i := n;
  comment  each Schur function and its sign are to be found in
    c[1:i] and sig respectively;
L7: conjugate (c, i, d, q);
  begin
    array x[1:q, 1:q];
    for i := 1 step 1 until q do
    for j := 1 step 1 until q do
    begin p := d[i] − i + j;
      x[i, j] := if p < 0 ∨ p > n then 0 else
      if p = 0 then 1 else unit[p]
    end;
    sigma := sigma + sig × determinant (x, q)
  end;
  go to L3;
L99: express := sigma
end express
```

ALGORITHM 306
PERMUTATIONS WITH REPETITIONS [G6]

P. Bratley (Recd. 23 Sept. 1966 and 15 Feb. 1967)
Department of Computer Science, University of Edinburgh, Edinburgh Scotland

```
procedure perm(a, n, last);  value n;  integer n;
  integer array a;  Boolean last;
comment  a[1:n] is an integer array. Initially the elements of
  a[1:n] must be arranged in descending order and last must be
  set true. If the elements of a are not initially in descending
  order the effect of the procedure is undefined. Successive calls of
  perm generate in a all permutations of its elements in reverse
  lexicographical order.
    last is set false if the procedure has generated a new permutation, but if the procedure is entered after all the permutations
```

have been generated, *last* will be set **true**. Neither *a* nor *n* should be altered between successive calls of the procedure;

```
begin integer i, p, q, r;
  own integer m;  own integer array b[1:n];
  if ¬ last then go to L12;  last := false;
  for i := 1 step 1 until n do b[i] := a[i];
  p := b[n];
  for i := n step −1 until 1 do
    if p ≠ b[i] then
      begin m := i;  go to L99 end;
  m := 0;  go to L99;
L12:  if m = 0 then go to L10;
  p := b[m];  q := m;  r := 0;
L9:  i := n;
L4:  if a[i] = p then go to L2;
  if a[i] < p then r := i;
L5:  i := i − 1;  go to L4;
L2:  a[i] := b[n] − 1;  if r = 0 then go to L8;
L1:  a[r] := p;  q := q + 1;
L3:  r := r + 1;  if r > n then go to L11 else if a[r] > p
  then go to L3;
L11:  if b[q] = p then go to L1;  r := 0;
L6:  r := r + 1;  if a[r] ≥ p then go to L6;
  a[r] := b[q];  if q = n then go to L7;
  q := q + 1;  go to L6;
L7:  last := false;  go to L99;
L8:  q := q − 1;  if q = 0 then go to L10;
  if b[q] = p then go to L5;
  p := b[q];  go to L9;
L10:  last := true;
L99:
end perm
```

ALGORITHM 307
SYMMETRIC GROUP CHARACTERS [A1]

J. K. S. McKay (Recd. 23 Sept. 1966, 15 Feb. 1967, and 10 Mar. 1967)
Department of Computer Science, University of Edinburgh, Edinburgh, Scotland

```
integer procedure character (n, rep, longr, class, longc, first);
  value n, rep, longr, class, longc;
  integer n, longr, longc;  Boolean first;
  integer array rep, class;
comment character produces the irreducible character of the
  symmetric group corresponding to the partitions of the repre-
  sentation and the class of the group S_n stored with parts in
  descending order in arrays rep[1:longr] and class[1:longc], re-
  spectively. Both arrays are preserved. The method is similar
  to that described by Bivins et al. [1]. Comét describes a later
  method.
    On first entry to character, first should be set true in order to
  initialize the own array p[0:n, 0:n]. This single initialization is
  sufficient for all symmetric groups of degree less than or equal
  to n. character is intended for computing individual characters.
  If a substantial part of the character table is required it is sug-
  gested that procedure generate [Algorithm 263, Comm. ACM
  8 (Aug. 1965), 493)] be used to produce the partitions prior to
  use of character. If this is done, then the own array p should be
  replaced by a suitable global array, and first should be set false
  to avoid unwanted initialization. character uses procedures set,
  generate, and place [Algorithms 262, 263, 264, Comm. ACM 8
  (Aug. 1965), 493].
```

REFERENCES:

1. BIVINS, R. L., METROPOLIS, N., STEIN, P. R., and WELLS, M. B. Characters of the symmetric groups of degree 15 and 16. *MTAC 8* (1954), 212–216.
2. LITTLEWOOD, D. E. *The Theory of Group Characters*. Clarendon Press. Oxford, England 1958, 2d ed., Ch. 5.
3. COMÉT, S. Improved methods to calculate the characters of the symmetric group. *MTAC 14* (1960), 104–117.;

```
begin
  integer procedure degree (n, rep, length);  value n, length;
    integer n, length;  integer array rep;
  comment degree gives the degree of the representation of the
    symmetric group on n symbols defined by the partition
    rep[1:length] with parts in descending order;
  begin
  own integer array p[0:n, 0:n];
    integer array q[1:length];  integer i, j, deg;
    integer procedure fac(n);  value n;  integer n;
    fac := if n = 1 then 1 else n × fac(n−1);
    for i := 1 step 1 until length do
      q[i] := rep[i] + length − i;
    deg := fac(n);
    for i := 1 step 1 until length do
    for j := i + 1 step 1 until length do
      deg := deg × (q[i]−q[j]);
    for i := 1 step 1 until length do
      deg := deg ÷ fac(q[i]);
    degree := deg
  end degree;
  if first then
    begin set (p, n);  first := false end;
  begin
    integer array pr[1:n], r[0:1, 0:p[n, n]−1];
    integer length, m, t, old, new, index, i, char, k, coeff, u, pos,
      j1, j2;
    m := longc;
    new := n;
    index := 1;
    for i := 0 step 1 until p[n, n] − 1 do
      r[index, i] := 0;
    r[index, place(p, n, rep)] := 1;
    for t := 1 step 1 until m do
    begin if class[t] = 1 then go to identity;
      index := 1 − index;  old := new;  new := new − class[t];
      for i := 0 step 1 until p[new, new] − 1 do
        r[index, i] := 0;
      for u := p[old, old] − 1 step − 1 until 0 do
      begin if r[1 − index, u] = 0 then go to B;
        generate (p, old, u, pr, length);
        k := length;  j1 := 1;
G:      j2 := j1;  coeff := r[1−index, u];
        for i := 1 step 1 until k do rep[i] := pr[i];
        if rep[1] = old then go to H;
        rep[j2] := rep[j2] − class[t];
        if rep[j2] + k − j2 < 0 then go to B;
E:      if rep[j2] ≥ if (j2 = k then 0 else rep[j2+1]) then go to F;
        if rep[j2+1]= rep[j2] + 1 then go to J;
        i := rep[j2+1];  rep[j2+1] := rep[j2] + 1;
        rep[j2] := i − 1;  coeff:= − 1 coeff;  j2 := j2 + 1;
        go to E;
H:      rep[1] := rep[1] − class[t];
F:      pos := place(p, new, rep);
        r[index, pos] := r[index, pos] + coeff;
J:      j1 := j1 + 1;  if j1 ≤ k then go to G;
B:
      end
    end;
```

```
A: char := r[index, 0];  go to Z;
identity:  char := 0;
    for u := p[new,new] − 1 step − 1 until 0 do
    begin if r[index, u] = 0 then go to BB;
      generate(p, new, u, pr, length);
      char := char + r[index, u] × degree (new, pr, length);
BB:
    end;
Z:  character := char
  end
end character
```

ALGORITHM 308
GENERATION OF PERMUTATIONS IN PSEUDO-LEXICOGRAPHIC ORDER [G6]

R. J. Ord-Smith (Recd. 11 Nov. 1966, 1 Dec. 1966, 28 Dec. 1966 and 27 Mar. 1967)
Computing Laboratory, University of Bradford, England

Lexicographic generation has the advantage of producing an order easily followed by the user, but its real value in certain combinatorial applications is that a $(k-1)$-th intransitive subgroup of permutations is generated before the kth element is moved. By not insisting on strict lexicographic generation, though preserving the latter property, an enormous reduction in the total number of transpositions is obtained. The total number of transpositions in this algorithm can be shown to tend asymptotically to (sinh 1) $n!$ which is less than in Algorithm 86 [J. E. L. Peck and G. F. Schrack, Permute, *Comm. ACM 5* (Apr. 1962), 208] and almost as good as Algorithm 115 [H. F. Trotter, Perm, *Comm. ACM 5* (Aug. 1962), 434]. The algorithm offers a further useful facility. Like several others it uses a nonlocal Boolean variable called *first*, which may be assigned the value **true**, to initialize generation. On procedure call this is set **false** and remains so until it is again set **true** when complete generation of permutations has been achieved. At any subsequent call after initializing generation of permutations of degree n, one may set parameter $n = n'$ where $n' \leq n$. Further calls with this value may continue until the completion of the subgroup of degree $(n' - 1)$ when *first* will be set **true**. The process can be continued by resetting *first* **false** and calling with a larger value of n. This gives the user complete control over the main attribute which lexicographic order offers. There is no restriction on the elements permuted. Table I gives results obtained for *ECONOPERM*. Times given in seconds are for an ICT 1905 computer. The algorithm has also been tested successfully on IBM 7094, Elliott 503 and STC Stantec computers. t_n is the time for complete generation of $n!$ permutations. r_n has the usual definition $r_n = t_n/(n \cdot t_{n-1})$.

TABLE I

Algorithm	t_6	t_7	t_8	r_6	r_7	r_8	*Number of transpositions*
ECONOPERM	0.85	6.2	50.6	—	1.04	1.02	→ 1.175$n!$

```
procedure ECONOPERM (x, n);  value n;  integer n;
  array x;
begin own integer array q[2:n];
  comment  own dynamic arrays are not often implemented.
    The upper bound will then have to be given explicitly;
  integer k, l, m;  real t;
  l := 1;  k := 2;
  if first then
    begin first := false;  go to label end;
  comment  the above is the initialization process;
loop:  if q[k] = k then
  begin if k < n then
  begin k := k + 1;  go to loop end
  else begin first := true;  go to finish end
end;
n := k − 1;
comment  note n called by value;
label:  for m := 2 step 1 until n do q[m] := 1;
  comment  after the initialization the for statement sets all
    elements of q array to 1. Otherwise only the first k−2 elements
    are reset 1;
  q[k] := q[k] + 1;
transpose:  t := x[l];  x[l] := x[k];  x[k] := t;
  l := l + 1;  k := k − 1;
  if l < k then go to transpose;
  comment  when k < 4 only one transposition occurs. On final
    exit when first is reset true, no transposition occurs at all;
finish:
end of procedure ECONOPERM
```

REMARKS ON:

ALGORITHM 87 [G6]
PERMUTATION GENERATOR
[John R. Howell, *Comm. ACM 5* (Apr. 1962), 209]

ALGORITHM 102 [G6]
PERMUTATION IN LEXICOGRAPHICAL ORDER
[G. F. Schrak and M. Shimrat, *Comm. ACM 5* (June (1962), 346]

ALGORITHM 130 [G6]
PERMUTE
[Lt. B. C. Eaves, *Comm. ACM 5* (Nov. 1962), 551]

ALGORITHM 202 [G6]
GENERATION OF PERMUTATIONS IN LEXICOGRAPHICAL ORDER
[Mok-Kong Shen, *Comm. ACM 6* (Sept. 1963), 517]

R. J. Ord-Smith (Recd. 11 Nov. 1966, 28 Dec. 1966 and 17 Mar. 1967)
Computing Laboratory, University of Bradford, England

A comparison of the published algorithms which seek to generate successive permutations in lexicographic order shows that Algorithm 202 is the most efficient. Since, however, it is more than twice as slow as transposition Algorithm 115 [H. F. Trotter, Perm, *Comm. ACM 5* (Aug. 1962), 434], there appears to be room for improvement. Theoretically a "best" lexicographic algorithm should be about one and a half times slower than Algorithm 115. See Algorithm 308 [R. J. Ord-Smith, Generation of Permutations in Pseudo-Lexicographic Order, *Comm. ACM 10* (July 1967), 452] which is twice as fast as Algorithm 202.

ALGORITHM 87 is very slow.

ALGORITHM 102 shows a marked improvement.

ALGORITHM 130 does not appear to have been certified before. We find that, certainly for some forms of vector to be permuted, the algorithm can fail. The reason is as follows.

At execution of $A[f] := r$; on line prior to that labeled *schell*, f has not necessarily been assigned a value. f has a value if, and only if, the Boolean expression $B[k] > 0 \wedge B[k] < B[m]$ **is true** for at least one of the relevant values of k. In particular when matrix A is set up by $A[i] := i$; for each i the Boolean expression above is **false** on the first call.

ALGORITHM 202 is the best and fastest algorithm of the exicographic set so far published.

A collected comparison of these algorithms is given in Table I. t_n is the time for complete generation of $n!$ permutations. Times are scaled relative to t_8 for Algorithm 202, which is set at 100. Tests were made on an ICT 1905 computer. The actual time t_8 for Algorithm 202 on this machine *was* 100 seconds. r_n has the usual definition $r_n = t_n/(n \cdot t_{n-1})$.

TABLE I

Algorithm	t_6	t_7	t_8	r_6	r_7	r_8
87	118	—	—	—	—	—
102	2.1	15.5	135	1.03	1.08	1.1
130	—	—	—	—	—	—
202	1.7	12.4	100	1.00	1.00	1.00

CERTIFICATION OF:
ALGORITHM 258 [H]
TRANSPORT
[G. Bayer, *Comm. ACM 8* (June 1965), 381]
ALGORITHM 293 [H]
TRANSPORTATION PROBLEM
[G. Bayer, *Comm. ACM 9* (Dec. 1966), 869]

LEE S. SIMS (Recd. 21 Feb. 1967 and 17 Mar. 1967)
Kates, Peat, Marwick & Co., Toronto, Ont., Canada

Both of these algorithms were coded in Extended ALGOL 60 and tested on a Burroughs B5500. Three problems were solved correctly, one of them being of medium size (55 × 167). On this larger problem *transp*1 was found to be about twice as fast as *transport*.

In coding and debugging *transp*1 three apparent errors were found. In the right-hand column on page 870, after line 27 which is

$i := listu[u]$; $nlvi := nlv[i]$;

a line is missing. This line should read

for $s := (i-1) \times n + 1$ **step** 1 **until** *nlvi* **do**

Also in the right-hand column, the line

*s*4: ;

should be inserted ahead of line −12, which begins

comment Step 4. A column j with $b[j]$ has been labeled, $b[j]$

On page 871, in the left-hand column, line −22 which reads

for $s := 1$ **step** 1 **until** n **do**

should read

for $s := l$ **step** 1 **until** n **do**

CERTIFICATION OF ALGORITHM 285 [H]
THE MUTUAL PRIMAL-DUAL METHOD
[Thomas J. Aird, *Comm. ACM 9* (May 1966), 326]

H. SPÄTH (Recd. 13 Feb. 1967)
Institut für Neutronenphysik und Reaktortechnik, Kernforschungszentrum, Karlsruhe, Germany

The procedure *Linearprogram* has been translated into FORTRAN II and successfully run on the IBM 7074 Computer. The following corrections had been made (the first two are merely typographical errors).

1. P. 328, left column, 1 line after label *B*3:
reads:
if $A[row[k-1, i], col[k, 0]] >$ **then**
should read:
if $A[row[k-1, i], col[k, 0]] > 0$ **then**

2. P. 328, left column, 1 line after label *B*4:
reads:
if $A[row[k-1, i], col[k, 0]] >$ **then**
should read:
if $A[row[k-1, i], col[k, 0]] > 0$ **then**

3. P. 328, right column, after the end of the procedure *pickapivot* and before the label *NEXTPIVOT* there must be inserted the statement
col[0, 0] := 0;

Otherwise *col*[0, 0] has no assigned value when the procedure *subschema* is entered for the first time.

REMARK ON ALGORITHM 301 [S20]
AIRY FUNCTION [Gillian Bond and M.L.V. Pitteway, *Comm. ACM 10* (May 1967), 291]

M.L.V. Pitteway (Recd. 19 May 1967)
Brunel University, ACTON, W.3., England

The initial minus sign has been omitted from the line immediately following the line

end calculation of derivatives;

The statement should read

$p := -(rtmdx/xi) \times (2 \times A[2] + 4 \times A[4] + 6 \times A[6] + 8 \times A[8] + 10 \times A[10])$;