10. Various operations concerned with permutations.

10A. Cycle form of permutation (omitting unit cycles).

\[
\text{set} = \{ j, 1 < j < n \}; \quad \text{cycs} = \text{nl}; \quad \text{while set ne nl} \quad \text{beg from set};
\]

\[
\text{cycl} = \text{nl};
\]

\[
\text{cycl}(1) = \text{beg}; \quad \text{while perm(beg) \in set} \quad \text{beg} = \text{perm(beg)};
\]

\[
\text{cycl}(\#\text{cycl}+1) = \text{beg};
\]

\[
\text{set} = \text{set less beg}; \quad \text{end while}; \quad \text{if } \#\text{cycl} > 1 \quad \text{then cycl in cycs};
\]

\[
\text{end while};
\]

10B. Same, including unit cycles. Replace final if by: cycl in cycs;

10C. Standardized cycle form of a permutation.

\[
\text{cycs} = \{ \text{minfirst } c, c \in \text{cycs} \}; \quad (\forall c \in \text{cycs}) \quad \text{place}(c) =
\]

\[
\#\{ d \in \text{cycs} \mid d(1) > c(1) \};
\]

\[
\text{cyco} = \{ \langle \text{place}(c), c \rangle, c \in \text{cycs} \}; \quad \text{define } \text{minfirst } c; \quad k=1;
\]

\[
(2 < \forall j < \#c) \quad k = \text{if } c(j) < c(k) \quad \text{then } j \quad \text{else } k;
\]

\[
\text{return } \{ \langle \text{if } j \ge k \quad \text{then } j+1-k \quad \text{else } n+1-k+j, c(j) \rangle, 1 < \forall j < \#c \};
\]

\[
\text{end minfirst};
\]

10D. Integer sequence representing a permutation. Form cycle
form cycs, including unit permutation, then standardized cycle
form from this. Then order these in decreasing order of first
elements. Parentheses can be dropped, and reconstructed by
finding

\[
is = \text{nl}; \quad (1 < \forall n < \#\text{cyco}) \quad \text{cy} = \text{cyco}(n); \quad (1 < \forall m < \#\text{cy}) \quad \text{is} = (\#\text{is} + 1) =
\]

\[
\text{cy}(m); \quad \text{end } \forall n;
\]

And the inverse of this:

\[
\text{cyco} = \text{nl}; \quad \text{cyc} = \text{nl}; \quad \text{least} = \text{is}(1); \quad (1 < \forall n < \#is) \quad \text{if } \text{is}(n) < \text{least} \quad \text{then}
\]

\[
\text{cyco}(\#\text{cyco} + 1) = \text{cyc}; \quad \text{cyc} = \text{nl}; \quad \text{min} = \text{is}(n); \quad \text{cyc}(1) = \text{is}(n);\n\]

\[
\text{else } \text{cyc}(\#\text{cyc} + 1) = \text{is}(n); \quad \text{end if}; \quad \text{end } \forall n;
\]
10E. "In place" inversion of a permutation. Permutation given by $\text{perm}(n)$

todo = $\{ n_1 \} , 1 \leq n_1 \leq n_n$ ; (while todo ne nl) start from todo;
now = start; next = perm(start);
(while perm(now) $\in$ todo) $\langle$next, now, perm(next)$\rangle$
$\langle$perm(next), next, now$\rangle$; todo = todo less now;;
perm(start) = now; end while todo;

10F. Multiplication of permutations in cycle form. Second Knuth algorithm. This exploits the fact that (abcd...ef) restricted to the set of letters appearing is product

$$(f \rightarrow \star) \ (e \rightarrow f) \ (d \rightarrow e) \ldots \ (a \rightarrow b) \ (\star \rightarrow a),$$

and that

$$(x \rightarrow y) \ \text{map} = \text{map} \text{ means } \text{map}(x) = \text{map}(y), \text{ with all other values unchanged.}$$

map = nl; (#cycs $\geq$ $\forall n \geq 1$ \#cycs(n) $\geq$ 1)c = cycs(n);
mapstar = if map(c(1)) ne $\land$ then map(c(1)) else c(1);
(1 $\leq \forall i < \#c$) map(c(1)) = if map(c(i+1)) ne $\land$ then

map(c(i+1)) else c(i+1);;
map(c(#c)) = mapstar; end $\forall n$;

10G. Multiplication of permutations in cycle form. First Knuth algorithm. This keeps set which might be leftmost occurrence of some variable.

cyco = nl; list = nl; tag = nl; (1 $\leq \forall n \leq \#cycs$)
(1 $\leq \forall m \leq \#cycs(n)$) $\langle$list(#list+1), tag(#list+1)$\rangle$ =

$\langle$cycs(m), $\star$$\rangle$;
$\langle$list(#list+1), tag(#list+1)$\rangle$ = $\langle$cycs(1), $\star$$\rangle$; end $\forall n$;
(while 1 ≤ j ≤ #list | n tag(j))

⟨start, current, tag(j)⟩ = ⟨list(j), list(j+1), t⟩;
cyc = ⟨1, list(j)⟩; do buildcyc(start, current, j);
if (#cyc > 1) then cyco(#cyco+1) = cyc;; end while;

block buildcyc(start, current, j); [newelt:];
(while j+1 < #list | list(k) eq current and tag(k) eq f)
⟨j, current, tag(k)⟩ = ⟨k, list(k+1), t⟩;;
if current ne start then cyc(#cyc+1) = current; j = 1; go to newelt;
end if; end buildcyc;

10H. Inversion of a permutation, Boothroyd algorithm. This brief but surprising algorithm works as follows. Let c be a circular permutation, which we may think of as shifting elements arranged in a circle. Let s be the set of these elements. Initially, set f=c, and mark each element of s as a "head". Then, process the elements p of s, in any random order, as follows. Find the first element g=fk(p) which is marked as a head; remove the head mark of r=f(g); and redefine f(g) as f(r) and f(r) as p. To follow the action of this algorithm, divide the set s into a set of runs, each run consisting of a sequence g, g(p), g1(a), ... gk(a) beginning with an element marked as a head, and continuing up to but not
including the next element marked as a head. Then note inductively that

i. The first element of a run is marked, all other elements are unmarked (by definition);

ii. All elements of a run but its last have already been processed;

iii. For all but the first element of a run, \( f(p) \) is the previous element; for the first element of a run, \( f(p) \) is the first element of the next succeeding run (or is \( p \) itself, if only one run remains.)

All these remarks hold initially, with runs of unit length. Since by ii every unprocessed element is the tail \( q \) of a run, our algorithm always finds \( p \); causes the head \( r \) of the next run to point to \( q \), removes the mark on \( r \), and causes \( p \) to point to the head of the next run but 1, thereby joining two runs and preserving the conditions i, ii, and iii. Since this process works for every cyclic permutation, it works for any union of cyclic permutations, i.e., any permutation. In SETL, we assume a set \( s \) and a mapping \( f \). Then

\[ \text{heads} = s; \ (\forall p \in s) \; q = p; \ (\text{while } n \ q \in \text{heads}) \ q = f(q); \ ; r = f(q); \ \langle f(r), f(q) \rangle = \langle p, f(r) \rangle; \ \text{heads} = \text{heads less } r; \ \text{end } \forall p; \]

11. Algorithm for matching problem. Given a multivalued map \( f \) on a set \( s \), one may ask if there exists a 1-1 \( g \) such that \( g(x) \subseteq f(x) \). The necessary and sufficient condition is that \( \# f[t] > \# t \) for each subset \( t \) of \( s \). To prove this sufficient, call \( t \) thin if


\#f[t] = \#t = c(t); otherwise thick. If \( t_1 \) and \( t_2 \) are thin, then

\[
\#f[t_1 \cap t_2] = c(t_1) + c(t_2) - \#\left( f[t_1] \cap f[t_2] \right)
\]

\[
< c(t_1) + c(t_2) - c(t_1 \cap t_2),
\]

so that (by additivity of \( c \)) \( t_1 \cap t_2 \) is thin also, and

\[
\#\left( f[t_1] \cap f[t_2] \right) \leq c(t_1 \cap t_2).
\]

Hence there exists a minimal thin set, \( t \), and letting \( x \in t \) correspond to some chosen \( y \in f[x] \), we have condition satisfied for \( s \less x, f[s] \less y \), since for \( t_1 \) not thin there can be no trouble, and for \( t_1 \) thin, \( y \in f[t_1] \) implies \( x \in t_1 \). More generally, \( c \) can be any positive additive function, and we can require that a map \( g \) with all \( q(x) \)'s disjoint and

\[
\#g(x) \geq c(x)
\]

be found. Argument here reduces \( c(x) \) by 1 whenever \( y \) is chosen. Thus algorithmic procedure hinges on finding a minimal thin set. We take \( c(x) = [t:y \in x]k(y) \)

It is convenient to use the following function

define pwr(x,n); return \( \{ y \in pow(x) \mid \#y = n \} \); end pwr;

which will probably be available as a primitive.

g1 = match (s, f[s], f,k); define match (s, im, f,k);

t = minthin (s, im, f,k); x = \exists t; y = \exists f[x] \cap im;

if \( k(x) \geq 1 \) then \( k(x) = k(x) - 1; \)
else \( g_1 = match (t, f[t] \less y \cap im, f,k) \);

\( g_2 = match (s \cap t, f[s] \cap f[t], f,k); \)

\( return \ g_1 \cup g_2 \ with \ \langle x,y \rangle; \); end match;

define minthin (s, im, f,k); (1 \leq n \leq \#s,y \in pwr(s,n))

\( \begin{align*}
kk & = [t:z \in y]k(z); \quad xx = \{ x \in f[y] \mid x \in im \}; \quad \text{if} \ kk \geq \#xx \text{ then}
\end{align*} \)

print "necessary condition violated"; exit;
if \( k \) \( \equiv \) \#\( \text{xx} \), then return \( y \); end \( \forall n \); print "necessary condition violated"; exit; end \( \text{minthin} \);

11A. A slightly more efficient \( \text{minthin} \) algorithm if \( k = 1 \).

define \( \text{minthin}(s, \text{im}, f) \); \( n = 1 \); (while \( n \leq \#s \))

\[
\text{minim} = \#\text{im} + 1; (\forall y \in \text{pwr}(s, n)) \text{ xx } = \{ x \in f[y] / x \in \text{im} \};
\]

if \( \#\text{xx} \lt n \) then print "necessary condition violated"; exit;;

if \( \#\text{xx} \equiv n \) then return \( y \);

if \( \#\text{xx} \lt \text{minim} \) then \( \text{minset} = \{ \text{xx} \} \); \( \text{minim} = \#\text{xx} \);
else if \( \#\text{xx} \equiv \text{minim} \) then \( \text{xx} \) in \( \text{minset} \); end if; end \( \forall y \);

\[
(\forall \text{xx} \in \text{minset}) \text{ fill } = \{ x \in s / (\forall z \in f\{x\} \mid n \ z \in \text{im} \text{ or } z \in \text{xx}) \};
\]

if \( \#\text{fill} \equiv \text{minim} \) then return \( \text{fill} \); end \( \forall \text{xx} \); \( n = \text{minim} + 1 \); end while;

return \( s \); end \( \text{minthin} \);

12. **Cantor's Diagonalizer:** Given \( s \) and multi-valued map \( f: s \rightarrow s \), produces set which is not element

\[
\text{cantset} = \{ x \in s / n \ x \in f\{x\} \};
\]

13. **Power set generator.** On successive calls, get successive elements of \( \text{pwr}(\text{set}, n) \). When \( \text{set} = \text{nl} \), process resets.

define \( \text{nexpow}(\text{set}, \text{subs}, n) \); initially flag = 0; if flag \( \equiv 1 \)

then go to advance;;

if \( n \gt \#\text{set} \) then \( \text{subs} = \text{nl} \); return;; flag = 1; ordset = \text{nl};

\( s = \text{set} \);

(while \( s \neq \text{nl} \) elt from \( s \); ordset(\#ordset + 1) = elt;; last = \#set;

\( \text{img} = \{ \langle i, i \rangle \}, 1 < i < n \}; \) go to ret;
advance: if set = \texttt{nl} then go to drop;; if \texttt{img(n)} \texttt{lt} last then \texttt{m=} \texttt{n};
go to found;; if \texttt{n} > \lceil \frac{m}{2} \rceil \geq 1 \texttt{ } \texttt{img(m)} \texttt{lt} (\texttt{img(m+1)}-1) then go to found;
drop: flag = 0; \texttt{subs = \emptyset} ; return;
found: \texttt{img(m)} = \texttt{img(m)}+1; \texttt{im = img(m)}; (m < \forall k \leq n) \texttt{im = im+1};
\texttt{img(k) = im};
ret: \texttt{subs = \{ordset(img(i))}, 1 \leq i \leq n\}; return; end nexpow;

define nexmap(map, froms, into); initially flag = 0; if flag eq 1 then go to advance;;
if froms eq \texttt{nl} then \texttt{map = nl}; flag = 1; return;; flag = 1;
ordfrom = \texttt{nl}; ordto = \texttt{nl}; s = froms;
(while s ne \texttt{nl}) elt from s; ordfrom(#ordfrom+1) = elt;; s = into;
next = \texttt{nl}; elt from s; first = elt; (while s ne \texttt{nl}) eltx from s;
next (elt) = eltx; elt = eltx; end while; last = elt;
\texttt{img} = \{ordfrom(i), first\}, 1 \leq i \leq \#froms\}; go to ret;
advance: if set = \texttt{nl} then go to drop;
(\#froms \geq \forall j \geq 1) if \texttt{img(j)} ne last then \texttt{m = j}; go to found;;
drop: flag = 0; \texttt{map = \emptyset}; return;
found: \texttt{img(ordfrom(m)) = next(img(ordfrom(m)))};
(m < \forall k \leq \#froms) \texttt{img(ordfrom(k)) = first};
ret: \texttt{map = img}; return; end nextmap;