```
{
Solution for RACE
-------- --- ----

The data structure for a course is as follows:

CONST Max_Arrows = 100;
VAR number_of_points,number_of_arrows : INTEGER;
    Arrows : ARRAY [0..Max_Arrows-1,0..1] OF INTEGER;

The number of the start point is 0,
the number of the finish point is number_of_points-1.
The i-th arrow (0 <= i <= number_of_arrows-1) goes from the point with
number Arrows[i,0] to the point with number Arrows[i,1].

The data structure for the solution of the task is as follows:

CONST Max_points = 50;
TYPE Point_array = ARRAY [0..Max_points-1] OF BOOLEAN;
VAR number_unavoidable_points,number_splitting_points : INTEGER;
    unavoidable,splitting : Point_array;

number_unavoidable_points is the number of unavoidable points
(apart from the start point and the finish point).
number_splitting_points is the number of splitting points.
unavoidable[N] is TRUE iff the point with number N is an unavoidable point.
splitting[N] is TRUE iff the point with number N is a splitting point.

The main body of the program is:

BEGIN
  initialisation;
  read_input;
  compute_results;
  write_output;
  finalisation;
END.

The procedure initialisation initialises the variables mentioned above
and file variables for input and output.
The procedure read_input reads the input; the variables of the data
structure for the course receive their final value.
The procedure compute_results computes the results; the variables of
the data structure for the solution of the task receive their final value.
The procedure write_output writes the output to the appropriate file.
The procedure finalisation closes the files.

We will only consider the procedure compute_results from here, as the
implementation of the other procedures is straightforward.

The procedure compute_results determines for each point (except start
point and finish point) whether it is an unavoidable point, and if so,
whether it is a splitting point (Note that each splitting point is an
unavoidable point as well).
To determine whether current_point is an unavoidable point, determine
the set S of all points which can be reached from the start point via a
path which does not contain current_point. Obviously current_point is
```

an unavoidable point iff the finish point is not in S.
S is determined by calling the procedure find_reachable with current_point as argument. The result is stored in the Point_array reachable. After the call find_reachable (current_point) the value of reachable[N] is TRUE iff point N can be reached from the start point via a path which does not contain current_point. By definition, reachable [current_point] is FALSE.
Careful analysis now shows that current_point is a splitting point iff there is no arrow from a point P with reachable[P] = FALSE to a point Q with reachable[Q] = TRUE. This is checked by the function is_splitting.

Below the complete code is given:
}

```
PROGRAM RACE;

CONST Max_points = 50;
      Max_Arrows = 100;

TYPE Point_array = ARRAY [0..Max_points-1] OF BOOLEAN;

VAR input_file,output_file : text;
    current_arrow,current_point,number_of_points,number_of_arrows,
    number_unavoidable_points,number_splitting_points : INTEGER;
    Arrows : ARRAY [0..Max_Arrows-1,0..1] OF INTEGER;
    unavoidable,splitting : Point_array;

PROCEDURE initialisation;
BEGIN
  Assign(input_file,'input.txt');
  Reset(input_file);
  Assign(output_file,'output.txt');
  Rewrite(output_file);
  number_of_points:=1;
  number_of_arrows:=0;
  number_unavoidable_points:=0;
  number_splitting_points:=0;
  FOR current_point:=0 TO Max_points-1 DO
    BEGIN
      splitting [current_point]:=FALSE ;
      unavoidable [current_point]:=FALSE
    END;
END;

PROCEDURE read_input;
VAR num : INTEGER;
BEGIN
  read(input_file,num);
  WHILE NOT (num = -1) DO
  BEGIN
    IF num = -2
    THEN INC (number_of_points)
    ELSE BEGIN
           Arrows [number_of_arrows][0] := number_of_points-1;
           Arrows [number_of_arrows][1] := num;
           INC (number_of_arrows)
         END;
    read(input_file,num)
```

```
  END;
END;

PROCEDURE write_output;
BEGIN
  Write(output_file,number_unavoidable_points);
  FOR current_point:=1 TO number_of_points-2 DO
    IF unavoidable[current_point]
    THEN write(output_file,' ',current_point);
  Writeln(output_file);
  Write (output_file,number_splitting_points);
  FOR current_point:=1 TO number_of_points-2 DO
    IF splitting[current_point]
    THEN write(output_file,' ',current_point);
END;

PROCEDURE finalisation;
BEGIN
  Close(input_file);
  Close(output_file);
END;

PROCEDURE compute_results;

VAR reachable : Point_array;

  PROCEDURE find_reachable (current_point:INTEGER);
  VAR point:INTEGER;
      ready:BOOLEAN;
  BEGIN
    FOR point:=1 TO number_of_points - 1 DO
      reachable[point]:=FALSE;
    reachable[0]:=TRUE;
    ready:=FALSE;
    WHILE NOT ready DO
    BEGIN ready:=TRUE;
          FOR current_arrow:=0 TO number_of_arrows-1 DO
            IF reachable [Arrows[current_arrow,0]] AND
               NOT reachable [Arrows[current_arrow,1]] AND
               (Arrows[current_arrow,1]<>current_point)
            THEN BEGIN reachable[Arrows[current_arrow,1]]:=TRUE;
                       ready:=FALSE
                 END;
    END
  END;

  FUNCTION is_splitting:BOOLEAN;
  VAR current_arrow:INTEGER;
      OK:BOOLEAN;
  BEGIN
    current_arrow:=0;
    OK:=TRUE;
    WHILE (current_arrow<number_of_arrows) AND OK DO
    BEGIN
      OK:=reachable [Arrows[current_arrow,0]] OR
          NOT (reachable [Arrows [current_arrow,1]]);
      INC(current_arrow)
```

```
    END;
    is_splitting:=OK
  END;

BEGIN
  FOR current_point:=1 TO number_of_points-2 DO
    BEGIN
      find_reachable (current_point);
      IF NOT reachable[number_of_points-1]
      THEN BEGIN
             unavoidable [current_point] := TRUE;
             INC (number_unavoidable_points);
             IF is_splitting
             THEN BEGIN
                    splitting[current_point]:=TRUE;
                    INC (number_splitting_points)
                  END
           END
    END;
END;

BEGIN
  initialisation;
  read_input;
  compute_results;
  write_output;
  finalisation;
END.


{
Background
----------

The background for the task RACE is to be found in the field of
software structure metrics. In software structure metrics,
flowgraphs are assigned to software entities (like programs,
modules, procedures, expressions or statements). Flowgraphs
can be hierarchically decomposed into prime flowgraphs. The
metric values for a software entity are defined inductively
over the decomposition tree of the corresponding flowgraph.

A flowgraph is a directed graph with the same properties as a
well formed course in the task RACE:
One of the nodes is marked as the start node, one of the nodes
is marked as the stop node and:
- each node can be reached from the start node
- from each node the stop node can be reached
- the stop node has outdegree 0.

There are two ways of combining two flowgraphs into a new
flowgraph: sequencing and nesting.
Sequencing two flowgraphs means identifying the stop node
of the first flowgraph with the start node of the second.
Nesting of flowgraph F on flowgraph G means replacing an edge
in G (whose source must have outdegree one) by F.
```

Decomposition of a flowgraph means finding smaller flowgraphs
from which the flowgraph can be obtained by sequencing and/or
nesting.
Subtask B of RACE amounts to finding all decompositions of a
flowgraph as a sequence of two smaller flowgraphs.

The algorithms for the decomposition of flowgraphs, which are
used in software metrication tools, all start with finding all
post-dominator pairs in the flowgraph. A pair (a,b) of nodes is
a post-dominator pair if all paths from a to the stop node
contain b. Then b is called a post-dominator of a.

If only sequential decompostions of a flowgraph are sought,
only the post-dominators of the start node are needed.
To find the post-dominators of the start node is exactly
subtask A of RACE.

Pim van der Broek
Scientific Committee IOI'95
}