Opgave A: Deadlock Detectie

Nederlands Kampioenschap Programmeren
voor Universiteitsteams

Invoerbestand : `A.IN`
Uitvoerbestand: `A.UIT`
Programma : `A.PAS` of `A.C`

In een computercentrum staat een leuke mainframe, waar maximaal 25 gebruikers tegelijkertijd hun proces op kunnen draaien. Om de kosten zo laag mogelijk te houden moeten deze gebruikers een aantal resources met elkaar delen. Hierbij moet je denken aan printers, modems, enzovoorts. Er zijn nooit meer dan 20 verschillende soorten resources en van elke soort kunnen er maximaal maar 100 zijn.

Dit gaat echter herhaaldelijk mis, omdat verschillende processen op elkaars resources staan te wachten zonder dit van elkaar te weten (deadlock). Om deze situatie, waarin een proces eigenlijk indirect op zichzelf staat te wachten, te herkennen moet er een programma geschreven worden, die aan de hand van het aanvragen en weer loslaten van resources door de processen controleert of er een deadlock ontstaat.

Bij het aanvragen en loslaten (elk proces maximaal 50 keer) geldt een aantal regels:

1. Als een aanvraag geheel gehonoreerd kan worden, dan krijgt dit proces zijn resources en gaat verder.
2. Als een aanvraag gedeeltelijk gehonoreerd kan worden, dan krijgt dit proces zoveel mogelijk resources en gaat vervolgens wachten op de rest.
3. Als een aanvraag helemaal niet gehonoreerd kan worden, dan gaat dit proces wachten tot de resources wel beschikbaar zijn.
4. Als een proces resources weer vrijgeeft, dan worden deze resources wanneer mogelijk verdeeld over de wachtende processen. Hierbij hebben de processen die het langst staan te wachten voorrang op de andere processen. Dit herverdelen gaat door tot de resources weer op zijn of tot er geen wachtende processen meer zijn die de beschikbare resources willen hebben.

De vraag is om een programma te schrijven dat bij een gegeven invoer van een bepaald aantal runs bepaalt of er tijdens de executie van de processen een deadlock ontstaat, hoe laat deze ontstaat, door welk proces dit veroorzaakt wordt en welke processen er momenteel betrokken zijn bij de deadlock (welke processen allemaal indirect op zichzelf staan te wachten). Als er geen deadlock ontstaat willen we graag weten hoe laat al de processen afgelopen zijn.

In de invoerfile staan de volgende regels:

```
          <#runs : r>
          <#processen run 1: n1> <#soorten resources : m1>
          <#resources1> <#resources2> . . . <#resourcesm1>
          <#aanvragen/loslatingen proces 1: p1>
          <a of l> <tijd> <#resources1> . . . <#resourcesm1>
p1 keer
          <a of l> <tijd> <#resources1> . . . <#resourcesm1>
          <#aanvragen/loslatingen proces 2: p2>
          <a of l> <tijd> <#resources1> . . . <#resourcesm1>
p2 keer
          <a of l> <tijd> <#resources1> . . . <#resourcesm1>
n1 keer
          <#aanvragen/loslatingen proces n1: p(n1)>
          <a of l> <tijd> <#resources1> . . . <#resourcesm1>
p(n1) keer
          <a of l> <tijd> <#resources1> . . . <#resourcesm1>
r keer
          <#processen run r: nr> <#soorten resources : mr>
          <#resources1> <#resources2> . . . <#resourcesmr>
```

```
          <#aanvragen/loslatingen proces 1: p1>
          <a of 1> <tijd> <#resources1> . . . <#resourcesmr>
p1 keer
          <a of 1> <tijd> <#resources1> . . . <#resourcesmr>
          <#aanvragen/loslatingen proces 2: p2>
          <a of 1> <tijd> <#resources1> . . . <#resourcesmr>
p2 keer
          <a of 1> <tijd> <#resources1> . . . <#resourcesmr>
n1 keer
          <#aanvragen/loslatingen proces n1: p(n1)>
          <a of 1> <tijd> <#resources1> . . . <#resourcesmr>
p(n1) keer
          <a of 1> <tijd> <#resources1> . . . <#resourcesmr>
```

Deze invoerfile is zo opgebouwd dat een proces nooit meer resources vrijgeeft als dat hij in zijn bezit heeft. Verder kan een proces van een resource wel meer aanvragen dan er maximaal van die resource aanwezig is, dit resulteert in een deadlock waar alleen dat proces bij betrokken is. De laatste aanvraag/loslating is altijd gelijk aan het loslaten van al de resources, die het proces nog in zijn bezit heeft. Met `<a of 1>` wordt bedoeld dat als er een `a` staat de genoemde resources worden aangevraagd, staat er een `1` dan worden de genoemde resources vrijgegeven. Verder is `<tijd>` de tijd in rekenstappen die verstreken is sinds de vorige aanvraag/loslating helemaal afgehandeld is. De totale tijd dat een run duurt komt niet boven *maxint* uit.

De uitvoerfile ziet er als volgt uit:

```
<run>: <tekst>
..
<run>: <tekst>
```

Waarbij `<tekst>` is:

```
alle machines klaar op tijdstip <tijdstip>
```

of

```
deadlock op tijdstip <tijdstip>
veroorzaakt door proces <procesno veroorzaker>
betrokken zijn: <proces> <proces> ... <proces>
```

Al naar gelang de uitkomst van je programma.

In het eerste geval, is er tijdens de run geen deadlock ontstaan. In het tweede geval is er wel een deadlock ontstaan en wel op tijdstip `<tijdstip>`. Het veroorzakende proces is dat proces dat door zijn nu afgehandelde aanvraag/loslating een deadlock doet ontstaan. De betrokken processen zijn die processen, die op tijdstip `<tijdstip>` in een deadlock zijn geraakt.

Voorbeeld in- en uitvoer:

A.IN

```
2
3  2
1  2
3
a  0  0  1
a  1  1  0
l  2  1  1
3
a  0  1  0
a  2  0  1
l  1  1  1
2
a  0  0  1
l  3  0  1
2  2
1  1
3
a  0  1  0
a  1  0  1
l  1  1  1
3
a  0  0  1
a  1  1  0
l  1  1  1
```

A.UIT

```
1: alle machines klaar op tijdstip 6
2: deadlock op tijdstip 1
veroorzaakt door proces 2
betrokken zijn: 1 2
```

Einde van opgave A

Let op: Er hoeft alleen naar de linker mouwen gekeken te worden. Als die eenmaal vaststaan, zijn door cyclisch verschuiven altijd kleuren voor de middenstukken en rechtermouwen. (bv 1,2,2 / 2,3,3 / 3,4,4 / 4,1,1)

Opgave B: Jassen

Nederlands Kampioenschap Programmeren
voor Universiteitsteams

$n \leq 100$

Invoerbestand : `B.IN`
Uitvoerbestand: `B.UIT`
Programma : `B.PAS` of `B.C`

Een groep van N mensen gaat jassen kopen met afritsbare mouwen. Deze jassen zijn in zeer veel verschillende kleuren leverbaar. De mouwen van de jassen zijn verwisselbaar. Hierdoor kan een grote verscheidenheid aan jassen gemaakt worden (bijvoorbeeld: een rode linker mouw, een blauw middenstuk en een groene rechtermouw). Deze mensen willen nu jassen kopen, en wel zo dat (1) vrienden onderling een verschillende kleur linkermouw, rechtermouw en middenstuk hebben. En (2) iedere persoon afzonderlijk een jas heeft waarvan de linker- en rechtermouw een andere kleur heeft. Er is een NxN matrix M, waarin vrienden worden aangegeven door een 1 en niet-vrienden door een 0.

Opgave:
Bouw een algoritme, dat als invoer een getal N en een matrix M accepteert en berekent wat het minimaal aantal kleuren is waarmee volstaan kan worden.

Voorbeeld:
Er zijn vier personen, persoon a, b, c en d. De personen a en c zijn bevriend, hetzelfde geldt voor de personen a en d, b en c en b en d. Invoer is hier dus het getal $N(=4)$ en de matrix M:

$$M = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} & 0 & 1 & 1 \\ 0 & & 1 & 1 \\ 1 & 1 & & 0 \\ 1 & 1 & 0 & \end{pmatrix} \end{array}$$

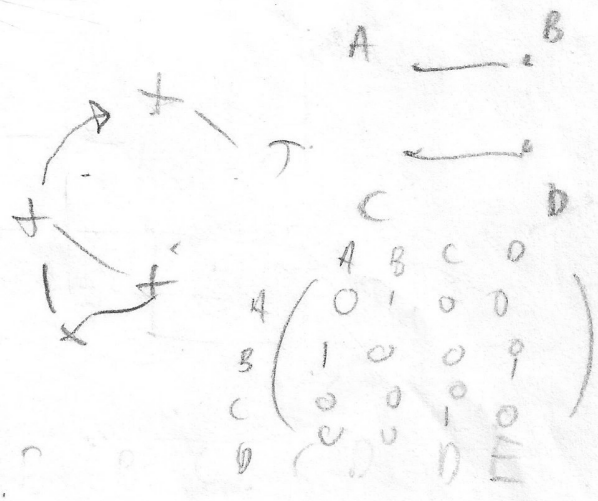
Er kan nu worden volstaan met twee kleuren zeg x en y. Een oplossing is: (linkermouw, middenstuk, rechtermouw) $a : (x, x, y)$ $b : (x, x, y)$ $c : (y, y, x)$ en $d : (y, y, x)$

De invoer ziet er als volgt uit:

```
T (#testen)
dimensie bij test 1 (=N)
v12 v13   ...   v1N
v21 v23 v24 ... v2N
    ...
vN1 vN2 ... vN(N-1)
dimensie bij test 2
...
dimensie bij test T
```

De uitvoer hoort als volgt te zijn:

```
k1 (#kleuren bij test 1)
k2
k3
..
kT
```



Voorbeeld in- en uitvoer:

B.IN

```
2
4
  0 1 1
0   1 1
1 1   0
1 1 0
5
 0 1 1 0
0  0 0 1
1 0  1 1
1 0 1  0
0 1 1 0
```

B.UIT

```
2
3
```

Let erop dat de er overal spaties staan tussen de getallen. Ook op de diagonaal en achter de laatste regel.

Einde van opgave B

Opgave C: Bootjes

Nederlands Kampioenschap Programmeren
voor Universiteitsteams

Invoerbestand : `C.IN`
Uitvoerbestand: `C.UIT`
Programma : `C.PAS` of `C.C`

In een of andere grootste havenstad van Nederland moet een binnenkomend schip achtereenvolgens de volgende handelingen verrichten (tussen de haakjes staat de duur van de handeling in minuten):

1. Melden (5)
2. *Wachten* (???)
3. Slepen (80)
4. *Wachten* (???)
5. Douane (30)
6. *Wachten* (???)
7. Legen (600)

Hierbij betekent een ??? dat er een onbekende tijd gewacht moet worden op respectievelijk een sleepboot, douanier of leeghaal ploeg. Er zijn $a(a > 0)$ sleepboten, $b(b > 0)$ douaniers en $c(c > 0)$ leeghaalploegen die met één boot tegelijk bezig kunnen zijn. Het melden kan met willekeurig veel boten tegelijk gebeuren. Als een sleepboot/douanier/leeghaalploeg klaar is met z'n taak, kunnen ze meteen (instantaan) met een volgend schip aan de slag. Op deze manier kunnen de schepen semi-parallel verwerkt worden. De schepen worden geholpen in de volgorde van aankomst (bij gelijke tijden mag wilekeurig gekozen worden).

In de invoerfile staan een regel met $x(x > 0)$, een integer die het aantal runs aangeeft, gevolgd door x maal een input in hetvolgende formaat: een regel met a, een met b en een met c, gevolgd door een regel met n $(n > 0)$ wat het aantal schepen aangeeft waarachter een n-tal aankomsttijden van n verschillende schepen (in minuten vanaf $t = 0$, aankomsttijden zijn $>= 0$).

Gevraagd: het tijdstip waarop het laatste schip klaar is en het totaal aantal gewachtte minuten van alle schepen bij elkaar.

Voorbeeld in- en uitvoer:

C.IN

```
2
2
2
2
2
23
312
1
1
1
2
226
227
```

C.UIT

```
Run:1
Laatste:1027
Wachttijd:0
Run:2
Laatste:1541
Wachttijd:599
```

Einde van opgave C

levien
Acmg2nk

Opgave D: Nullen tellen

Nederlands Kampioenschap Programmeren
voor Universiteitsteams

Invoerbestand : D.IN
Uitvoerbestand: D.UIT
Programma : D.PAS of D.C

Bovenaan in de invoerfile staat een getal $n (n > 0)$, het aantal runs. Hieronder staan n getallen m ($1 <= m <= 25000$). Bepaal voor elk van deze getallen m op hoeveel nullen (0'en) het getal $m!$, uitgeschreven in het 12-tallig stelsel, eindigt. In de uitvoerfile komt op regel n het aantal nullen in de 12-tallige ontwikkeling van m_n.

Voorbeeld in- en uitvoer:

D.IN
3
6
24
180

D.UIT
2
10
88

Einde van opgave D

Opgave E: Blokkenwereld

Nederlands Kampioenschap Programmeren
voor Universiteitsteams

Invoerbestand : `E.IN`
Uitvoerbestand: `E.UIT`
Programma : `E.PAS` of `E.C`

Een populair onderwerp is tegenwoordig robotica. Het gaat hier vaak om neurale netwerken die zijn gekoppeld aan een robotarm en een videocamera. Het netwerk moet blokken leren herkennen en deze verplaatsen binnen de kamer. In onze kamer staan drie tafels. Op de eerste tafel staat een stapel van $n(n > 0)$ blokken, $(V_1 \ldots V_n)$. Deze blokken kunnen **K**ubussen, **C**ilinders, **P**iramides of **B**ollen ($V_i \in \{K, C, P, B\}$) zijn. Het is de bedoeling deze blokken te verplaatsen van de eerste tafel naar de derde tafel. De robotarm kan maar 1 voorwerp tegelijk verplaatsen. De kunstmatige intelligentie in de robotarm zorgt ervoor dat hij niet een voorwerp uit het midden van de stapel probeert te pakken. Hij mag gebruik maken van de tweede tafel om de blokken tijdelijk op te stapelen.

Het beeldherkenningsgedeelte is evenals het robotarm-besturingsgedeelte al door ons uitgeprogrammeerd. Het is de bedoeling om een algoritme te verzinnen dat het minimum aantal verplaatsingen, t, bepaalt dat nodig is om de blokken van de eerste tafel naar de derde tafel te verplaatsen. Helaas zijn niet alle blokken even groot; de blokken bovenop de stapel zijn kleiner dan de blokken onderop. Grote blokken kunnen niet op kleinere blokken gestapeld worden. Bovendien kunnen niet alle blokvormen op elkaar gestapeld worden.

Voor het stapelen van de blokken geldt nog een voorwaarde: We definiëren de functie Op(a,b), die zegt of a op b gestapeld kan worden:

$$\begin{aligned}\mathrm{Op}(V_i, V_j) \Leftrightarrow\ & i < j \wedge \quad (\quad (V_j = K \wedge V_i \in \{K, C, P, B\}) \quad \vee \\ & \qquad\qquad (V_j = C \wedge V_i \in \{K, C, B\}) \quad \vee \\ & \qquad\qquad (V_j = B \wedge V_i \in \{C\}) \quad \vee \\ & \qquad\qquad (V_j = P \wedge V_i \in \{B\}) \\ & \qquad).\end{aligned}$$

In het invoerbestand staan op achtereenvolgende regels het aantal runs, $r(> 0)$. Vervolgens voor elke run het aantal blokken $n(n > 0)$ en de blokken V_i, op elke regel 1, beginnende bij V_1 aflopend tot V_n. Het eerste blok is dus de bovenste op de stapel. Er worden geen onmogelijke stapels in het invoerbestand gedefinieerd.

In het uitvoerbestand moet voor elke run telkens op een nieuwe regel komen het getal t, het minimum aantal verplaatsingen. Als het niet mogelijk is de stapel in zijn geheel te verplaatsen, moet dat gemeld worden met de string "`Niet verplaatsbaar`".

Voorbeeld in- en uitvoer:

E.IN
2
3
B
P
K
5
P
K
C
C
C

E.UIT
7
Niet verplaatsbaar

Einde van opgave E

Opgave F: Layout

Nederlands Kampioenschap Programmeren
voor Universiteitsteams

Invoerbestand : `F.IN`
Uitvoerbestand: `F.UIT`
Programma : `F.PAS` of `F.C`

Ieder jaar wordt er aan het begin van het universiteitsjaar een poging gewaagd eerstejaars de kunst van het programmeren bij te brengen. Over het algemeen wordt dit gedaan in *Pascal*, maar ook zijn geruchten gehoord over *Elan1*, *Modula* of zelfs *Scheme*.

De basisvormen van programmeren kunnen vrij snel, redelijk worden bijgebracht. Ook de stappen in de richting van functionele decompositie en data-abstractie worden nog door studenten geaccepteerd.

De acceptatiegraad daalt echter tot onder het vriespunt als er meer subjectieve zaken aan de orde komen. Het verbod op het gebruik van globale variabelen in procedures en functies wordt nog wel eens geslikt (als het weerwoord "Maar het werkt zo toch ook!" wordt genegeerd). Het ultieme twistpunt is en blijft toch altijd de layout van programma's. Menig maal wordt dan door de praktikant opgemerkt dat het aanpassen van de layout volgens de gevraagde regels dom werk en tijdsverlies is, dat evengoed door de computer gedaan kan worden.

Gevraagd:
Schrijf een programma dat een bepaalde *Pascal*-achtige broncode controleert op layout en de aanwezige layout-fouten corrigeert. Hierbij dienen de hieronder beschreven structuurregels te worden toegepast.

1. **Inspringen**

   Inspringen is alleen toegestaan op grond van structuurregels 2 tot en met 10. Als er geen reden is om meer of minder in te springen, dient de zojuist hiervoor gebruikte linkerkantmarge te worden aangehouden. De inspringdiepte is drie spaties. Controleer het programma op nodeloos inspringen (en terugspringen).

2. **BEGIN - END**

   `BEGIN - END` wordt gebruikt om een blok van meer dan één instructie aan te geven. Alleen bij het aangeven van de body van programma's, procedures en functies mag het blok instructies uit minder dan twee instructies bestaan. `BEGIN` met bijbehorende `END` dienen altijd recht onder elkaar te worden geplaatst. Na een `BEGIN` wordt er verder gegaan op de volgende regel, waarbij de eerstvolgende instructie drie spaties verder naar rechts geplaatst dient te worden. Controleer op het juist inspringen, of `BEGIN`s en `END`s recht onder elkaar staan en of een blok inderdaad meer dan één instructie omvat (tenzij het de body betreft van een functie, procedure of programma). Merk op dat de `END` van de body van een `PROGRAM` gevolgd wordt door een ".".

```
PROCEDURE Hallo;
BEGIN
   WriteLn('Hallo');
   WriteLn('Allemaal')
END;
```

3. **IF <conditie> THEN - ELSE**

   De `IF` staat recht boven de bijbehorende `ELSE`. Direct na een `IF`-statement volgt op dezelfde

regel respectievelijk de te testen conditie en het woord THEN. Op de volgende regel komt de THEN-tak. Als er een ELSE-tak bestaat wordt deze op de regel direct onder het woord ELSE begonnen. Als de THEN-tak of ELSE-tak uit meer dan één instructie bestaat, wordt er recht onder de THEN resp. ELSE begonnen met BEGIN. Hierna kan regel 1 gevolgd worden. Als de THEN- tak of ELSE-tak uit één instructie bestaat wordt deze drie spaties meer naar rechts geplaatst dan de IF resp. ELSE.

```
FUNCTION Fac(n:Word):Word;
BEGIN
   IF n = 0 THEN
      Fac := 1
   ELSE
      Fac := n*Fac(n-1)
END;
```

4. **WHILE <conditie> DO**

Gevraagd wordt WHILE <conditie> DO op één regel te houden. Hierna dient te worden ingesprongen als het één instructie betreft. Betreft het meerdere instructies, dan volgt op de volgende regel direct onder het woord WHILE het woord BEGIN dat het begin van het blok instructies aangeeft.

```
...
WHILE (NOT stop) AND (aantal_landen < max_land) DO
BEGIN
   ...
   ...
END;
...
```

5. **FOR <voorwaarde> DO**

Bij de FOR-lus dient men hetzelfde te werk te gaan als bij de WHILE-lus. Op één regel staat het woord FOR, gevolgd door de naam van een teller, het toekenningsteken :=, de startwaarde, het woord TO of DOWNTO, de eindwaarde en tot slot het woord DO. Vervolgens dient men op de volgende regel in te springen, tenzij het een blok betreft (zie regel 4).

```
...
FOR teller := 1 TO 10 DO
   WriteLn(teller:2,' x 7 = ',teller*7:2);
...
```

6. **REPEAT - UNTIL <conditie>**

De REPEAT-UNTIL-lus wijkt in zoverre af van de andere lussen doordat het duidelijk uit twee gescheiden delen bestaat. Eerst komt het woord REPEAT, waarna op de volgende regel ingesprongen wordt en de body van de lus volgt. Uitzonderlijk hierbij is het feit dat er geen BEGIN-END nodig is. Op deze overbodige BEGIN hoeft niet gelet te worden in het programma. Na het blok volgt recht onder het woord REPEAT het woord UNTIL met het bijbehorende stopcriterium op dezelfde regel.

```
...
REPEAT
   Read(character)
UNTIL (character='J') OR (character='N');
...
```

7. **PROGRAM/PROCEDURE/FUNCTION**

Het keyword PROGRAM, PROCEDURE of FUNCTION begint altijd op de eerste kolom van een

regel. Na het key-word volgt de rest van de header. De gehele header dient op één regel te staan. Na de header volgt de declaratie van o.a. constanten, types en variabelen. Hiervoor wordt verwezen naar regel 8. Bij het PROGRAM keyword moet een lege regel volgen. Op de FUNCTION en PROCEDURE keywords volgt geen lege regel, maar na het bijbehorende BEGIN - END blok dient wel een lege regel te komen.

```
PROGRAM DatabaseControl;

USES
   Crt;
CONST
   BufferGrootte = 4096;
TYPE
   PersoonType =
   RECORD
      naam   : ...
      straat : ...
      ...
   END;
   DataBaseType = File OF PersoonType;
VAR
   database : DataBaseType;
   recpointer : LongInt;
BEGIN
   ...
   ...
END;
```

8. **USES/CONST/TYPE/VAR**

   Deze keywords kunnen voorkomen als deel van de programma- of een subprogrammadefinitie. De keywords staan alleen op een regel en na de declaratie dient te worden ingesprongen met drie spaties. Een voorbeeld staat bij 7.

9. **RECORD-END**

   Een RECORD-definitie wordt gebruikt om een groep gegevens, die logisch bij elkaar horen te koppelen. Het wordt gebruikt in de typedeclaratie, na de type-identifier volgt een = waarna direct onder de type-identifier het woord RECORD geplaatst wordt. Hierna wordt ingesprongen en de recorddeclaratie wordt afgesloten met een END recht onder het woord RECORD. Een voorbeeld wordt gegeven bij regel 6.

10. **Overbodige ";"**

    Het teken ";" wordt gebruikt om het einde van een instructie aan te geven. Gevallen waarin meer dan één ";" echter elkaar staan (direct of gescheiden door spaties of lege regels) hoeven dus niet voor te komen. In het te maken programma wordt verwacht dat deze gevallen eveneens gecorrigeerd worden. Let op! Alleen meerdere ";" achter elkaar hoeven dus maar verwijderd te worden, één enkele ";" achter een statement mag altijd blijven staan.

11. **Overbodige witregels**

    Op witregels in het invoer bestand moet niet worden gelet. In het uitvoer bestand komen er alleen witregels na de regel met PROGRAM, en na de body van een FUNCTION of een PROCEDURE.

Er kunnen natuurlijk veel meer regels aan deze worden toegevoegd. Voorbeelden hiervan zijn voorschrijvende regels voor het gebruik van hoofdletters en kleine letters. Deze vallen echter buiten

het bestek van deze opgave.

De invoer (F.IN) bestaat uit een PASCAL-programma, welke op basis van deze regels verbeterd dient te worden en weggeschreven naar een file F.UIT

Voorbeeld in- en uitvoer

F.IN

```
PROGRAM OpgaveFTest; VAR Teller : INTEGER;
PROCEDURE Decr(VAR x: INTEGER); BEGIN
   x := x-1;;; END;
BEGIN Teller := 3;;
      WHILE Teller > 0 DO
   Decr(Teller);;;


      IF Teller = 0 THEN WriteLn('Pipco!')
    ELSE WriteLn('ERNSTIGE FOUT, BEL BORLAND!');
    END.
```

F.UIT

```
PROGRAM OpgaveFTest;

VAR
   Teller : INTEGER;
PROCEDURE Decr(VAR x: INTEGER);
BEGIN
   x := x-1;
END;

BEGIN
   Teller := 3;
   WHILE Teller > 0 DO
      Decr(Teller);
   IF Teller = 0 THEN
      WriteLn('Pipco!')
   ELSE
      WriteLn('ERNSTIGE FOUT, BEL BORLAND!');
END.
```

Einde van opgave F

Opgave G: Transputer

Nederlands Kampioenschap Programmeren
voor Universiteitsteams

Invoerbestand : `G.IN`
: `G.SRT`
Uitvoerbestand: `G.UIT`
Programma : `G.PAS` of `G.C`

Tegenwoordig worden transputers veel gebruikt voor de implementatie van parallele algoritmen. Iemand heeft een demonstratie model gebouwd dat een reeks getallen sorteert. Het model bestaat uit een aantal, $N(> 0)$ transputers, $T_1 \ldots T_N$. Het gedrag van elke transputer wordt beschreven door het volgende pseudo-programma:

T_i

```
VAR a, b :integer;
REPEAT receive a from T(i-1);
IF a < b THEN send a to T(i+1)
             ELSE BEGIN send b to T(i+1);
                        b:= a
                  END
UNTIL switchoff;
```

De transputer T_1 krijgt zijn invoer niet van T_0, maar van de invoer file `G.IN` (een tekstfile), transputer T_N stuurt zijn uitvoer naar de file `G.SRT` (ook een tekstfile). Voor iedere test run kan de operator, vanaf de console het getal N wijzigen naar een getal tussen 1 en 256. Een speciale truuk in de hardware zorgt ervoor dat de transputers $T_1 \ldots T_N$ geïnitialiseerd worden met de eerste N getallen van de invoer reeks. Een andere truuk zorgt ervoor dat de laatste N uit de transputers worden gehaald. Op deze manier kunnen de gebruikers zich concentreren op de files `G.IN` en `G.SRT` en hoeven ze zich niet druk te maken over implementatiedetails. De invoerreeks is altijd minstens zo lang als het aantal transputers.

Bij kleine files werkte alles prima, maar voor langere files ging het sorteren niet perfect. In eerste instantie werd de schuld gegeven aan spanningsverschillen op het electriciteitsnet, maar toen bleek dat de fouten reproduceerbaar waren moest de fout in de software worden gezocht. Een slimme jongeman ontdekte dat hij in sommige gevallen het aantal transputers kon bepalen door de files `G.IN` en `G.SRT` te vergelijken. In de overige gevallen kon hij in ieder geval het minimum aantal bepalen.

Zijn jullie net zo slim als hij?

Hieronder staan twee voorbeelden. De invoer en uitvoer reeksen staan van links naar rechts, en dus is het het eenvoudigste om de transputers ook te nummeren van links naar rechts.

In	T_1	T_2	T_3	*Uit*
4 2 1 4 3 5	?	?	?	
. .	. .	. .	. .	. .
4 2 1	4	3	5	
4 2	4	3	5	1
4	4	3	5	2 1
	4	4	5	3 2 1
. .	. .	. .	. .	. .
	?	?	?	5 4 4 3 2 1

In	T_1	T_2	T_3	*Uit*
3 2 1 4 4 3 5	?	?	?	
. .	. .	. .	. .	. .
3 2 1 4	4	3	5	
3 2 1	4	4	5	3
3 2	4	4	5	1 3
3	4	4	5	2 1 3
. .	. .	. .	. .	. .
	?	?	?	5 4 4 3 2 1 3

In het eerste voorbeeld zijn minimaal drie transputers nodig om de gewenste uitvoer te krijgen, maar bij een groter aantal zou er geen verschil zijn. In het tweede voorbeeld zijn er *precies* drie transputers nodig.

Invoer:
De invoer van uw programma bestaat uit twee tekstbestanden. Het bestand `G.IN` bevat een aantal van (niet-lege) rijtjes van integers. Elke integer staat op een aparte regel. De rijtjes worden gescheiden door een lege regel, het laatste rijtje wordt niet gevolgd door een lege regel.

Het bestand `G.SRT` heeft precies hetzelfde formaat, het bevat de rijtjes die geproduceerd worden door het transputer-netwerk vanuit de corresponderende invoerrijtjes.

Uitvoer:
Het uitvoerbestand van uw programma is een tekstbestand dat op aparte regels het minimum nummer N geeft dat nodig is om vanuit elk rijtje in `G.IN` het corresponderende rijtje in `G.SRT` te maken.

Voorbeeld in- en uitvoer

G.IN
4
3
1
4
4
3
1
4

G.SRT
1
3
4
4
3
1
4
4

G.UIT
2
1

Einde van opgave G