41. ročník matematické olympiády na středních školách

Kategorie PCategory P

In: Andrej Blaho (editor); Leo Boček (editor); Karel Horák (editor); Jozef Moravčík (editor); Václav Sedláček (editor); Jaromír Šimša (editor); Pavel Töpfer (editor): 41. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 1991/1992. 33. mezinárodní matematická olympiáda. 4. mezinárodní olympiáda v informatice. (Slovak). Praha: Jednota českých matematiků a fyziků, 1997. pp. 79–117.

Persistent URL: http://dml.cz/dmlcz/404961

Kategorie P

Texty úloh

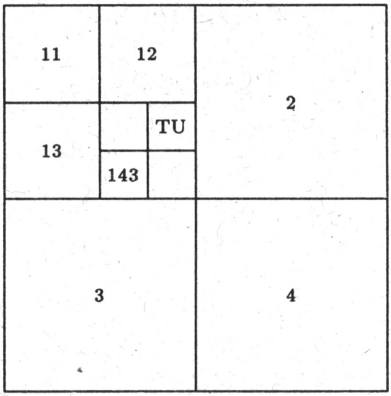
P – I – 1

Permutáciou čísel $1, 2, \ldots, N$ (krátko *permutáciou*) nazývame prosté zobrazenie množiny $\{1, 2, \ldots, N\}$ na seba. Permutácia P priradí každému číslu z intervalu $\langle 1\,..\,N\rangle$ nejaké číslo z intervalu $\langle 1\,..\,N\rangle$. Skladaním permutácií dostávame znova permutácie. Dvojité zloženie permutácie P je permutácia D s vlastnosťou: $D[i] = P[P[i]]$ pre každé i z intervalu $\langle 1\,..\,N\rangle$. Permutácia čísel $1, 2, \ldots, N$ sa v počítači reprezentuje dátovou štruktúrou typu pole, pričom i-ty prvok poľa obsahuje funkčnú hodnotu $P[i]$.

Úloha: Napíšte a zdôvodnite program, ktorý pre zadané celé číslo $N > 0$ a pole $P[1\,..\,N]$, reprezentujúce permutáciu čísel $1, 2, \ldots, N$, vypočíta jej dvojité zloženie. Výsledná permutácia musí byť po ukončení výpočtu uložená v poli P a program nesmie používať iné pole (resp. inú väčšiu dátovú štruktúru).

P – I – 2

So špeciálnym grafickým výstupom sa pracuje tak, že jeho štvorcová obrazovka je rozdelená na štvrtiny označené 1, 2, 3 a 4 (ľavá horná má číslo 1, pravá horná 2, ľavá dolná 3 a pravá dolná 4), každá zo štvrtín je zase rozdelená na štvrtiny a tak ďalej. Takto získané štvrtinky sa označujú kódom (tj. konečnou postupnosťou obsahujúcou číslice 1, 2, 3, 4) a nazývajú sa typizované štvorce. Pri komunikácii s programom sa kódy zapisujú vo forme prirodzených čísel v desiatkovej sústave (zmysel majú samozrejme len čísla obsahujúce iba cifry 1, 2, 3, 4). Celá obrazovka má kód 0. Napríklad štvorec 1, 4, 2 vidno na obrázku s nápisom „TU“ a jeho kód je číslo 142. Útvar je množina typizovaných štvorcov, reprezentovaná množinou ich kódov.



Úloha: Napíšte a dokážte čo najefektívnejší algoritmus, ktorý pre zadané $N > 0$ a pole $K[1..N]$ celých čísel — kódov typizovaných štvorcov zistí, či z nich vytvorený útvar je typizovaný štvorec.

P – I – 3

Máme daný nasledovný program v Pascale:

```
program MOPI3;
    var k, x, pocvel, pocmal: integer;
begin
    write('K: '); readln(k);
    {————————————————→} pocvel := 0; pocmal := 0;
    x := 0;
    while k > 1 do
        begin
        {————————————————→} pocvel := pocvel + 1;
        while (k mod 2) = 0 do
            begin
            x := x + 1; k := k div 2
            end;
        k := k − 1;
        while x > 0 do
            begin
            {————————————————→} pocmal := pocmal + 1;
            k := k * 2 + 1; x := x − 1
            end
        end;
```

writeln('Pocet velkych prechodov bol: ', *pocvel*);
writeln('Pocet malych prechodov bol: ', *pocmal*)
end.

Premenné *pocvel* a *pocmal* iba počítajú počty prechodov cez vonkajší cyklus a cez jeden z vnútorných.

Úloha: Zistite (a zdôvodnite svoje tvrdenie), pre ktoré čísla k je počet veľkých prechodov rovný počtu malých.

P – I – 4

Na vstupe máme slovo v tvare w_1 # w_2, kde reťazce w_1, w_2 sú neprázdne a obsahujú iba znaky '0' a '1'. Tieto reťazce reprezentujú čísla v dvojkovej sústave, pričom vyššie rády sú zapísané neskôr. Napr. vstup '1011#011' reprezentuje dvojicu čísel 1101 a 110 zapísaných v dvojkovej sústave.

a) Napíšte program vo Froscale, ktorý vypočíta súčet čísel w_1 a w_2 a vypíše ho v dvojkovom zápise v obrátenom poradí cifier (tj. tak isto, ako sú tie čísla zapísané na vstupe). Napríklad pre vstup v tvare '001#0111' by mal vypísať '01001'.
b) Napíšte program vo Froscale, ktorý vypočíta súčin čísel w_1 a w_2 a vypíše ho v dvojkovom zápise v obrátenom poradí cifier. Napríklad pre vstup '01#1001101' by mal vypísať '01001101'.
c) Popíšte (neformálne), ako by vyzerali programy riešené v časti a) a b) v prípade, že by reťazce w_1, w_2 obsahovali znaky '0' až '9' a reprezentovali by čísla v desiatkovej sústave.

Učebný text k 4. príkladu: Froscal — Frontový Pascal

Jazyk bez procedúr, premenných, syntaxou pripomínajúci Pascal (aj so skokmi) a s jedinou pamäťovou štruktúrou — frontom — to je **Froscal**. Front je dátová štruktúra s premenlivou veľkosťou (nie nepodobná zásobníku), u ktorej sa vkladá na jeden koniec a vyberá sa z druhého. Je známa rovnaká organizácia pamäti pod názvom FIFO — first in first out. Presnejšie: Froscal je Pascal bez premenných (a teda aj bez priradenia), s **while** a **repeat** cyklami, príkazmi **case** a **if** a skokom **goto**. Má dve fiktívne premenné INP a TOP typu char a skrytú premennú front znakov. INP je práve čítaný znak zo vstupu a TOP je znak na začiatku frontu. Ďalšie príkazy: NEXT načíta nový znak zo vstupu do fiktívnej premennej INP, PUT(znak) resp. PUT(premenná) uloží znak resp. hodnotu premennej INP alebo TOP na koniec frontu, GET vyhodí prvý prvok z frontu. Príkaz **write** má Pascalovskú definíciu, tj. môže sa vypísať znak, premenná, reťazec alebo postupnosť takýchto výrazov. Špeciálnym znakom na vstupe

a vo fronte je znak '?', ktorý na vstupe znamená presiahnutie vstupného slova a vo fronte znamená prázdny front. Ak INP = '?' (resp. TOP = '?'), potom príkaz NEXT (resp. GET) nič nespraví. Ani PUT('?') nič nevykoná.

Program pracuje tak, že na vstupe má slovo nad dopredu presne určenou abecedou (podmnožinou vypísateľných znakov) a na výstupe (**write**) môže vypísať riešenie problému. Testovať korektnosť vstupnej abecedy nie je nutné; dokonca ak je špecifikovaný formát vstupu, tak ho netreba testovať. Do frontu ukladať, testovať a vypisovať možno ľubovoľné vypísateľné znaky. Na začiatku práce programu je front prázdny (tj. TOP = '?').

Orientačná gramatika jazyka Froscal:

⟨program⟩ = **program** ⟨meno⟩; [⟨dekl. skokov⟩] ⟨telo programu⟩
⟨dekl. skokov⟩ = **label** ⟨skok⟩[, ⟨skok⟩]*;
⟨skok⟩ = ⟨celé číslo⟩
⟨telo programu⟩ = **begin** ⟨príkazy⟩ **end.**
⟨príkazy⟩ = ⟨nič⟩ | ⟨príkaz⟩[; ⟨príkazy⟩]
⟨príkaz⟩ = [⟨skok⟩:]*⟨elem. príkaz⟩
⟨elem. príkaz⟩ = **begin** ⟨príkazy⟩ **end**
| **if** ⟨podm⟩ **then** ⟨príkaz⟩[**else** ⟨príkaz⟩]
| **case** ⟨výraz⟩ **of**
(⟨selektor⟩: ⟨príkaz⟩;)*
[**else** ⟨príkaz⟩]
end
| **while** ⟨podm⟩ **do** ⟨príkaz⟩
| **repeat** ⟨príkazy⟩ **until** ⟨podm⟩
| **goto** ⟨skok⟩
| **write**(⟨write výraz⟩[, ⟨write výraz⟩]*)
| PUT(⟨výraz⟩)
| GET
| NEXT
⟨podm⟩ = ⟨relácia⟩
| (⟨podm⟩) and (⟨podm⟩)
| (⟨podm⟩) or (⟨podm⟩)
| not(⟨podm⟩)
⟨relácia⟩ = ⟨výraz⟩⟨relačný znak⟩⟨výraz⟩
⟨relačný znak⟩=
= | < | <= | > | >= | <> | in

⟨výraz⟩ = INP | TOP | ⟨znak konšt.⟩
| ⟨interval znakov⟩ | ⟨množina znakov⟩
⟨write výraz⟩ = INP | TOP | ⟨znak konšt.⟩ | ⟨reťazec⟩
⟨poznámka⟩ = {⟨text bez '}'⟩}
poznámka môže byť kdekoľvek rovnako ako v Pascale.

Príklad: Program, ktorý má na vstupe slovo nad abecedou ['a', 'b', 'c', 'd'] v tvare $w_1 \# w_2$. Treba otestovať, či sa $w_1 = w_2$.

```
program TEST;
label 13;
begin
    {front síce prázdniť netreba, ale na precvičenie}
    while TOP<>'?' do GET;
    while not((INP = '#') or (INP = '?')) do
        begin {ešte sme vo w1, treba uložiť}
            PUT(INP); NEXT
        end;
    if INP<>'#' then goto 13; {slovo má tvar w1 bez '#'}
    NEXT;
    while (INP<>'?') and (TOP = INP) do
        begin {ďalší znak sa rovná, ideme ďalej}
            GET; NEXT
        end;
    if ((INP = TOP) and (INP = '?')) then write('ANO')
    else 13: write('NIE')
end.
```

P – II – 1

Permutáciou čísel $1, 2, \ldots, N$ (krátko permutáciou) nazývame prosté zobrazenie množiny $\{1, 2, \ldots, N\}$ na seba. Permutácia P priradí každému číslu $i \in \langle 1..N \rangle$ nejaké číslo $P[i] \in \langle 1..N \rangle$. Skladaním permutácií dostávame znova permutácie. *K-te zloženie* permutácie P je permutácia P^K s vlastnosťou:

$$P^K[i] = \underbrace{P[P[\ldots P[i]]]}_{K-\text{krát}} \qquad \forall i \in \langle 1..N \rangle.$$

Permutácia čísel $1, 2, \ldots, N$ sa v počítači reprezentuje dátovou štruktúrou typu pole, pričom i-ty prvok poľa obsahuje funkčnú hodnotu $P[i]$.

Úloha: Napíšte a zdôvodnite čo najefektívnejší program (dôraz je na rýchlosti), ktorý pre zadané celé číslo $N > 0$ a pole $P[1..N]$, reprezentujúce permutáciu čísel $1, 2, \ldots, N$, a celé číslo K **podstatne väčšie** než N vypočíta jej K-te zloženie. Výsledná permutácia musí byť po ukončení výpočtu uložená v poli P.

P – II – 2

Uvažujme špeciálny grafický výstup opísaný v úlohe P–I–2.

Úloha: Napíšte a zdôvodnite program, ktorý pre dané dve celé čísla $S_1, S_2 \geqq 0$, reprezentujúce dva typizované štvorčeky (dané svojím kódom), zistí, či ležia pri sebe, tj. či sa dotýkajú svojimi stranami zvonka.

Príklad: Štvorčeky 3 a 143 ležia pri sebe, ale 143 a 1423 neležia pri sebe, rovnako ani 14 a 143 neležia pri sebe.

P – II – 3

Napíšte a dokážte čo najefektívnejší (dôraz je na rýchlosti) program, ktorý pre zadaný reťazec R dĺžky N nájde také dva rôzne znaky `Z1` a `Z2`, pri ktorých vykoná nasledovný algoritmus maximálny počet výmen.

```
EsteHladaj:=true;
while EsteHladaj do
  begin
    EsteHladaj:=false;
    for i:=1 to N-1 do
      if (R[i]=Z1) and (R[i+1]=Z2) then
        begin
          { výmena }
          R[i]:=Z2;
          R[i+1]:=Z1;
          EsteHladaj:=true
        end;
  end;
```

Znaky v reťazci R môžete uvažovať z intervalu `'A'..'J'`.

Poznámka: Dĺžka zadaného reťazca N môže byť veľmi veľká.

P – II – 4

Na vstupe je neprázdne slovo w obsahujúce iba znaky `'a'`. Navrhnite a popíšte program vo Froscale, ktorý vypočíta dolnú celú časť dvojkového logaritmu dĺžky slova w a vypíše ju v dvojkovom zápise v obrátenom poradí cifier.

Poznámka: Príklad sa dá riešiť aj bez použitia príkazu **goto**.

P – III – 1

Permutáciou čísel $1, 2, \ldots, N$ (krátko permutáciou) nazývame prosté zobrazenie množiny $\{1, 2, \ldots, N\}$ na seba. Permutácia P priradí každému číslu $i \in \langle 1..N \rangle$ nejaké číslo $P[i] \in \langle 1..N \rangle$. Skladaním permutácií dostávame znova permutácie. *Dvojité zloženie* permutácie P je permutácia P^2 s vlastnosťou:

$$P^2[i] = P[P[i]] \qquad \forall i \in \langle 1..N \rangle.$$

Permutácia čísel $1, 2, \ldots, N$ sa v počítači reprezentuje dátovou štruktúrou typu pole, pričom i-ty prvok poľa obsahuje funkčnú hodnotu $P[i]$.

Úloha: Napíšte a zdôvodnite čo najefektívnejší program (dôraz je na rýchlosti), ktorý pre zadané celé číslo $N > 0$ a pole $P_2[1..N]$, reprezentujúce permutáciu čísel $1, 2, \ldots, N$, vypočíta *počet* takých permutácií P, ktorých druhé zloženie je zadaná permutácia P_2, tj. pre ktoré platí $P^2 = P_2$.

P – III – 2

Uvažujme špeciálny grafický výstup opísaný v úlohe P–I–2.

Niekedy treba typizovaný štvorec presunúť niekam inam. Aký bude jeho kód po jeho presunutí o K jeho šírok doprava a L jeho šírok nahor?

Úloha: Napíšte a zdôvodnite program, ktorý pre dané celé číslo $S \geqq 0$, reprezentujúce typizovaný štvorec (daný svojim kódom), a nezáporné celé čísla $K, L \geqq 0$ vypočíta a vypíše kód štvorca S po presunutí o K jeho šírok doprava a L jeho šírok nahor, alebo ohlási `MIMO OBRAZOVKY`, ak výsledný štvorec sa nachádza mimo obrazovky zariadenia.

P – III – 3

Máme daný nasledovný program v Pascale:

```
program MOPIII3;
 var k,x,poc:integer;
     q:0..9;
begin
 write('K:');readln(k);
 { ---------------------------> } poc:=0;
 while k>0 do { hlavný cyklus }
  begin
   { ---------------------------> } poc:=poc+1;
```

```
q:=0;x:=0;
while (k mod 10)=q do { cyklus A}
 begin
  x:=x*10+q; k:=k div 10;q:=9-q
 end;
if q=9 then if k>0 then k:=(k+1)*10+9
                   else begin k:=0; x:=0; poc:=0 end
       else begin k:=k-1; x:=x*10 end;
while x>0 do { cyklus B }
 begin
  k:= k*10 + x mod 10; x:= x div 10
 end;
end;
writeln('Pocet prechodov bol:',poc)
end.
```

Úloha: Napíšte a dokážte program, ktorý pre zadané P vypočíta taký vstup K, aby výsledný počet prechodov hlavným cyklom *poc* bol rovný zadanému P.

P – III – 4

Na vstupe je slovo v tvare $w_1 * w_2$, pričom (neprázdne) slová w_1, w_2 obsahujú iba znaky `'a'` až `'z'` a dĺžka slova w_1 je *omnoho menšia* než dĺžka slova w_2.

Úloha: Napíšte a popíšte program vo Froscale, ktorý vypočíta *počet výskytov* slova w_1 v slove w_2 a vypíše ho v dvojkovom tvare v obrátenom poradí cifier.

Napríklad pre vstup `'aba*abababababababababababiuababab'` by mal vypísať postupnosť znakov `'0011'`, reprezentujúcu číslo 12 (dvojkovo 1100_2).

P – I – 1

Každú permutáciu P môžeme rozložiť na niekoľko navzájom disjunktných cyklov, $P = C_1 C_2 \dots C_m$. Každý cyklus tvoria nejaké prvky množiny $\{1, \dots, N\}$, označme preto $C_i = (a_{i,1} a_{i,2} \dots a_{i,l_i})$. Číslo l_i nazveme dĺžkou cyklu C_i.

Samotná permutácia (zobrazenie P) vznikne zložením jednotlivých cyklov. Podobne dvojité zloženie permutácie vznikne zložením dvojitých zložení jednotlivých cyklov; $D = P^2 = C_1^2 C_2^2 \dots C_m^2$. Stačí preto prejsť každým cyklom, „prečíslovať ho", tj. zabezpečiť aby $P'[i] = P[P[i]]$ (P' je stav poľa P po skončení programu, $P'[i] = D[i]$) a úloha je vyriešená.

Cyklus C_i tvorený prvkami $a_{i,1}, a_{i,2}, \dots, a_{i,l_i}$ môžeme prečíslovať postupnosťou priradení

$$P'[a_{i,1}] := P[P[a_{i,1}]]$$
$$P'[a_{i,2}] := P[P[a_{i,2}]]$$
$$\dots\dots\dots$$

Ale v okamihu vykonania prvého priradenia je hodnota $P[a_{i,1}]$ prepísaná novou hodnotou $P'[a_{i,1}]$, preto si treba pôvodnú hodnotu predchádzajúceho prvku pamätať, aby sme vedeli určiť $a_{i,2}$ ($= P[a_{i,1}]$). Takisto narazíme na problém pri prečíslovávaní posledného prvku cyklu. Totiž

$$P'[a_{i,l_i}] := P[P[a_{i,l_i}]] = P[a_{i,1}],$$

ale táto hodnota je opäť prepísaná (pamätáme si len jeden predchádzajúci prvok). Tento problém sa dá odstrániť uchovaním hodnoty prvého prvku cyklu.

Potrebujeme teda zabezpečiť dve veci: nájsť cyklus, ktorý sme zatiaľ neprečíslovali, a prečíslovať ho.

Jeden spôsob, ako nájsť ešte neprečíslovaný cyklus, je takýto:

Nech všetky prvky, ktoré sú už prečíslované, majú záporné znamienko; teda nech $P'[x] = -P[P[x]]$. Potom sa začiatok cyklu nájde ľahko ako prvý výskyt kladného čísla:

```
novy:=1;
repeat
  while (novy<=N) and (p[novy]<0) do
    begin
      p[novy]:=-p[novy];
```

```
                    prvok p[novy] už je prečíslovaný;
                    môžeme mu obnoviť kladné znamienko,
                    pretože ho už nebudeme testovať
      inc(novy)
    end;
  if novy<=N then
    precisluj(novy)        precisluj musí súčasne
                           nastavovať znamienko
until novy>N;
```

Úsek programu, ktorý zabezpečuje prečíslovanie jedného cyklu, môže vyzerať napr. takto:

```
procedure precisluj (zac:integer);
var i,j,pa:integer;
begin
  i:=zac;                zac je začiatok cyklu
  pa:=p[zac];            do pa uchováme P[prvý prvok cyklu]
  while p[i]<>zac do     podmienka P[i]=zac je splnená, iba
                         ak už je spracovaný posledný prvok
    begin                cyklu
      j:=p[i];           uchováme P[i]
      p[i]:=-p[p[i]];     ---> toto je vlastné priradenie;
                               súčasne nastavujeme znamienko
      i:=j               a posunieme sa na ďalší prvok cyklu
    end;
  p[i]:=-pa              ešte nastav posledný, opäť na záporné
end;
```

Pretože maximálna dĺžka cyklu je N, je aj zložitosť prečíslovania jedného cyklu $O(N)$. Lenže pri prečíslovávaní cyklu sa pristupuje *iba* k prvkom tohoto cyklu, teda pri prečíslovaní všetkých cyklov spolu sa vykoná $O(N)$ operácií. Časová zložitosť *celého* prečíslovania je teda lineárna vzhľadom k N. Pri hľadaní začiatku cyklu popísaný algoritmus testuje každý prvok práve raz (a práve raz sa mu otočí znamienko), preto je celková časová zložitosť tohoto riešenia úlohy $O(N)$.

Otáčanie znamienka je tu použité ako trochu nečistý trik, pretože sa takto vlastne simuluje pole logických hodnôt, a pomocné polia sú zakázané. Vychádzajúc z minulého ročníka MO-P, kde rozbor jedného príkladu považoval pomocné logické pole a „znamienkovanie" za rovnocenné metódy z hľadiska pamäťovej náročnosti, nemalo by to vlastne byť korektné riešenie. Faktom však ostáva, že predkladané riešenie skutočne pole v pravom slova zmysle nepoužíva, dosahuje však pomerne výhodnú časovú zložitosť. Pozor! Ak by sme hľadali začiatok cyklu vždy znova od prvého prvku, zvyšuje sa tým časovú zložitosť na $O(N^2)$.

Predvedieme teraz riešenie, ktoré sa dôsledne snaží nepoužívať metódy stotožniteľné s použitím poľa (majme napr. obmedzenie, že všetky prvky poľa P môžu nadobúdať iba hodnoty z intervalu $\langle 1..N \rangle$).

Na prvý pohľad sa zdá byť správna takáto metóda hľadania začiatku cyklu: Považujme za začiatky cyklov iba ich najľavejšie prvky. Pre každý prvok permutácie teda najprv predpokladáme, že v ňom cyklus začína. Prejdeme postupne celý cyklus, a ak sme pritom prešli prvok s indexom menším ako predpokladaný začiatok, práve testovaný prvok už nemôže byť začiatkom cyklu a treba skúsiť ďalší prvok:

```
for a:=1 to n do        skúšaj postupne všetky prvky
  begin
    i:=a;
    repeat
      moze:=(i>=a);     ak sme vpravo od a, ešte stále môže byť
      i:=p[i]           posunieme sa po permutácii
    until (a=i) or not moze;  celý cyklus alebo a nie je zač.
    if moze then        ak sme našli začiatok,
      precisluj2(a)     môžeme prečíslovať
  end;
```

(Pozn. `precisluj2` je ekvivalent procedúry `precisluj` s tým rozdielom, že neotáča znamienka.)

Pri tomto spôsobe vyhľadávania začiatku prechádzame pre každý prvok jeho cyklus (nie nutne celý, pretože hľadanie končí hneď pri prvom zlom prvku v cykle, ale vo všeobecnosti je počet prejdených prvkov v jednom cykle porovnateľný s N). Keďže prvkov je N, časová zložitosť je $O(N^2)$.

Popísaný spôsob má však výrazný nedostatok — rieši úlohu chybne. Ak totiž v permutácii P existuje cyklus párnej dĺžky, tak po dvojnásobnom zložení sa rozpadne na dva cykly polovičnej dĺžky. Ak potom pri hľadaní začiatku ďalšieho cyklu narazíme na najľavejší prvok patriaci druhej časti rozpadnutého cyklu (všetky prvky tejto druhej časti ležia vpravo od už nájdeného začiatku pôvodného veľkého cyklu), považujeme ho za začiatok nového, dosiaľ neprečíslovaného cyklu a aplikujeme naňho prečíslovanie. Tým sa však poruší pôvodné prečíslovanie, ktorým tento cyklus polovičnej dĺžky vlastne vznikol, čo v konečnom dôsledku poruší celé riešenie.

Celý postup teda zlyhá na druhej časti rozpadnutého veľkého cyklu. Stačí však za začiatok cyklu považovať nie najľavejší, ale najpravejší prvok cyklu. Tým sa zabezpečí, že žiaden z prvkov, ktoré sa práve prečíslujú,

sa už nebude testovať na to, či je začiatkom cyklu. Teda ak sa aj cyklus rozpadne, už to nevadí. Upravená časť programu:

```
for a:=1 to n do        skúšaj postupne všetky prvky
  begin
    i:=a;
    repeat
      moze:=(i<=a);     ak sme pred a, ešte stále môže byť
      i:=p[i]           posunieme sa po permutácii
    until (a=i) or not moze;  celý cyklus alebo a nie je zač.
    if moze then        ak sme našli začiatok,
      precisluj2(a)     môžeme prečíslovať
  end;
```

(Rovnako správne je hľadať najľavejší prvok, ale postupovať od konca poľa k začiatku.)

Pre takto upravený program platí všetko, čo sme povedali o predchádzajúcom (zložitosť), ale rieši úlohu správne a bez pomocného poľa.

Kompletný výpis jedného z riešení:

```
program P_I_1;
var p : array [1..100] of integer;
    n,a,i : integer;
    moze : boolean;
procedure precisluj (a:integer);
var i,j,pa:integer;
begin
  i:=a;
  pa:=p[a];
  while p[i]<>a do
    begin
      j:=p[i];
      p[i]:=p[p[i]];
      i:=j
    end;
  p[i]:=pa
end;
begin
  write('N: '); read(n);
  write('P: ');
  for i:=1 to n do read(p[i]);
  for a:=1 to n do
    begin
      moze:=true;
      i:=a;
      repeat
        moze:=(i<=a);
        i:=p[i]
```

```
      until (a=i) or not moze;
      if moze then
        precisluj(a)
    end;
  write('D: ');
  for i:=1 to n do write(p[i],' ');
  writeln
end.
```

P – I – 2

Nech A, B sú kódy dvoch typizovaných štvorcov. Štvorec B leží v štvorci A (je pokrytý štvorcom A) práve vtedy, keď $\exists k \geqq 0$: $A = B \; div \; 10^k$, teda kód A je prefixom kódu B. Takúto situáciu označíme $A = otec(B)$.

Štvorec K sa dá pokryť (štvorcami $K_1 \ldots K_n$), ak sa kód K vyskytuje medzi kódmi $K_1 \ldots K_n$, alebo ak sa medzi týmito kódmi vyskytujú súčasne $10 \cdot K + 1$, $10 \cdot K + 2$, $10 \cdot K + 3$, $10 \cdot K + 4$.

(1) Nech $K_x = otec(K_y)$. Je zrejmé, že kód K_y neovplyvní, či výsledný útvar bude typizovaný štvorec, alebo nie, lebo štvorec K_y je úplne pokrytý štvorcom K_x. Kód K_y teda nemusíme brať do úvahy.

(2) Nech K_x, K_y, K_z, K_v sú 4 *rôzne* kódy, pre ktoré platí:

$$K = K_x \; div \; 10 = K_y \; div \; 10 = K_z \; div \; 10 = K_v \; div \; 10.$$

Potom tieto štvorce tvoria spolu typizovaný štvorec K (o 1 „rád“ väčší).

Pravidlá (1) aj (2) redukujú počet kódov, ktoré treba brať do úvahy. Pravidlo (1) určuje, ktoré kódy sú zbytočné a pravidlo (2) skladá 4 štvorce do jedného. Prakticky všetky možné riešenia spočívajú v aplikovaní týchto pravidiel na vstupné kódy.

Tvrdenie: Ak $K_1 \ldots K_N$ sú kódy, ktoré ešte treba spracovať, $N > 1$ a na kódy $K_1 \ldots K_N$ sa nedá aplikovať žiadne z uvedených pravidiel, útvar nie je typizovaný štvorec.

DÔKAZ sporom: Predpokladajme, že výsledný útvar bude typizovaný štvorec, nech je jeho kód T. Nech $K_1 \ldots K_N$ sú kódy, ktoré ostali po aplikovaní pravidiel (1) a (2) a nech sa už žiadne z týchto pravidiel nedá použiť.Potom výsledný útvar je typizovaný štvorec práve vtedy, keď sa T dá pokryť štvorcami $K_1 \ldots K_n$. To znamená, že

a) $\exists i$, $1 \leqq i \leqq N$: $K_i = T$. Keďže ale T pokrýva všetky štvorce, platí $\forall j$: $K_i = otec(K_j)$ ($a = otec(a)$ platí). To znamená, že sa dá použiť pravidlo (1), čo je spor s predpokladom.

b) $\exists x, y, z, v$: $K_x = 10 \cdot T + 1$, $K_y = 10 \cdot T + 2$, $K_z = 10 \cdot T + 3$, $K_v = 10 \cdot T + 4$. Z toho vyplýva, že sa dá aplikovať pravidlo (2), čo je spor s predpokladom.

Rozdiel v riešeniach spočíva hlavne v rozdielnom spôsobe aplikácie pravidiel. Najjednoduchší algoritmus je nasledovný:

```
opakuj
  skús aplikovať (1)
  skús aplikovať (2)
pokiaľ sa ešte dá aplikovať
ak ostal jediný kód, tak  Je typizovaný štvorec
                     inak Nie je typizovaný štvorec
```

`skús aplikovať (1)` je zložitosti N^2 (prehľadávanie dvojíc čísel)
`skús aplikovať (2)` je zložitosti N^4 (prehľadávanie štvoríc čísel)

Hlavný cyklus prebehne rádovo N-krát (pravidlo (1) zmenší N o 1 a pravidlo (2) o tri (jeden kód nahradí štyri)). Celková zložitosť takéhoto algoritmu teda môže byť podľa individuálnych vylepšení $O(N^4)$ a viac.

Výrazné zlepšenie zložitosti sa dá dosiahnuť *usporiadaním* poľa kódov. Nech pre kódy $K_1 \ldots K_n$ platí:

a) $\forall x < y < z$: $K_x = otec(K_z) \Rightarrow K_x = otec(K_y)$, teda kódy štvorcov pokrytých štvorcom K_x tvoria súvislý úsek hneď za kódom K_x, a zároveň

b) $\forall x < y < z < v \Big(\exists K$: $K_x = 10 \cdot K + 1$; $K_y = 10 \cdot K + 2$; $K_z = 10 \cdot K + 3$; $K_v = 10 \cdot K + 4 \Rightarrow$

$$\forall w; x \leqq w \leqq v\colon (K_w = otec(K_x) \vee K_w = otec(K_y) \vee K_w = otec(K_z) \vee K_w = otec(K_v))\Big)$$

potom by sa operácie z predchádzajúceho programu `skús aplikovať (1)`, `skús aplikovať (2)` dali previesť lineárne vzhľadom na N aj s úpravou poľa.

Usporiadanie môže vyzerať napríklad takto: Nech $A = a_1 \ldots a_{n_A}$, $B = b_1 \ldots b_{n_B}$. Potom

$$\begin{aligned} a_1 < b_1 &\Rightarrow A < B \\ a_1 > b_1 &\Rightarrow A > B \\ a_1 = b_1 &\Rightarrow a_2 < b_2 \Rightarrow A < B \\ &\qquad\; a_2 > b_2 \Rightarrow A > B \\ &\qquad\; a_2 = b_2 \Rightarrow a_3 < b_3 \ldots \end{aligned}$$

pričom $a_x = 0$ pre $x > n_A$, podobne pre B. Toto usporiadanie vyhovuje požiadavkám.

Vidíme, že sa jedná o normálne usporiadanie podľa veľkosti kódov $K_1 \ldots K_N$, keby sme ich všetky doplnili sprava nulami na rovnaký počet cifier. Nech maximálny počet cifier je L. Potom zarovnanie na rovnaké dĺžky kódov je zložitosti $L \cdot N$. Triedenie možno použiť Quick-sort, Dobosiewics-sort (zložitosť $N \cdot \log N$) alebo takzvaný Radix-sort zložitosti $L \cdot N$.

Toto triedenie využíva skutočnosť, že čísla $K_1 \ldots K_N$ vieme jedným priechodom utriediť podľa jednej z cifier. Ak použijeme pomocné pole, dá sa dokonca zabezpečiť, aby pri triedení zachovával poradie kódov s rovnakou cifrou, podľa ktorej sa triedi, tzn. ak K_i, K_j sú kódy, zhodujú sa v cifre, podľa ktorej sa triedi a $i < j$, potom aj po roztriedení bude K_i *pred* K_j. Ak teda čísla roztriedime podľa poslednej (L-tej), predposlednej ($L-1$-vej), ... 2., 1. cifry, budú utriedené podľa veľkosti. Nech x-tá cifra $= \left[\frac{K_i}{10^{L-x}}\right] \bmod 10$,

$$A = a_1 \ldots a_{n_A}, 0, 0 \ldots \qquad (L \text{ cifier}),$$
$$B = b_1 \ldots b_{n_B}, 0, 0 \ldots \qquad (L \text{ cifier}).$$

Nech $(\exists r\colon a_1 = b_1 \wedge a_2 = b_2 \wedge \ldots \wedge a_r = b_r \wedge a_{r+1} < b_{r+1}) \Leftrightarrow A < B$.

Po triedení $(r+1)$-vou cifrou musí byť A pred B; po triedení r, $r-1$, $r-2$, ..., 1. cifrou sa poradie A, B musí zachovať, čiže skutočne $A < B$, takže po pretriedení bude A pred B.

Triedenie jednou cifrou je jeden priechod poľa; triedenie podľa L cifier má zložitosť $L \cdot N$.

Opäť existuje viac spôsobov ako zistiť, či takto utriedené pole je typizovaný štvorec. Ideálne by bolo, keby zložitosť tejto operácie nebola väčšia ako zložitosť triedenia.

Nech sa na $K_1 \ldots K_N$ nedá aplikovať (1) a dá sa (2). Môže nastať situácia, že by po aplikovaní (2) sa dalo aplikovať (1)? Nech $\exists x, y, z, v, K$: $K_x = 10 \cdot K + 1$, $K_y = 10 \cdot K + 2$, $K_z = 10 \cdot K + 3$, $K_v = 10 \cdot K + 4$. Teda K je predchodca K_x, K_y, K_z, K_v. Je jasné, že K_h také, že by $K = otec(K_h)$, ale ani jeden z $K_x \ldots K_v$ nie sú predchodcom K_h môže existovať len vtedy, ak $K_h = K$. Potom by sa ale dal aplikovať (1), lebo $K = otec(K_x) \ldots$, čo je spor s predpokladom. To znamená, že pravidlo (1) stačí aplikovať iba raz.

Majme utriedené pole $K_1 \ldots K_N$, aplikujme naň pravidlo (1). Ak by zvyšné kódy tvorili typizovaný štvorec, tak by sa dalo opakovane aplikovať pravidlo (2), kým by nezostal jediný štvorec.

Všimnime si prvok K_1:

a) buď K_1 je kódom výsledného typizovaného štvorca, potom ale N musí byť 1.

b) K_1 sa bude podľa pravidla (2) skladať s ďalšími kódmi; potom ale vďaka usporiadaniu sa musí $K_1 \bmod 10$ rovnať 1.

Algoritmus: opakovane, pokiaľ $N > 1$ a $K_1 \bmod 10 = 1$, skúšame poskladať kód K_1 *div* 10.

Celé zistenie, či množina kódov je typizovaný štvorec, alebo nie, je zložitosti $L \cdot N$: každý kód sa použije na spájanie do väčšieho štvorca práve raz; vďaka usporiadaniu sa štvorce využívajú porade.

P – I – 3

Správne riešenie je, že $PocVel = K - 1$ a počet priechodov malým a veľkým cyklom sa rovná vtedy, keď $K = 2^i$, $i \in \mathbb{N}$.

Aby sme mohli dokázať toto tvrdenie, musíme najprv dokázať niekoľko pomocných tvrdení:

Majme daný nasledovný algoritmus:

vstupná hodnota: `K[0]`

```
for i:=1 to x do
  K[i]:=2*K[i-1]+1;
```

výstupná hodnota `K[x]`

Dokážme, že $K_x = K_0 \cdot 2^x + (2^x - 1)$.

$$K_x = ((\ldots(\overbrace{(K_0 \cdot 2 + 1)}^{K_1} \cdot 2 + 1) \cdot 2 + 1) \ldots) \cdot 2 + 1),$$

čo je v podstate princíp Hornerovej schémy na výpočet hodnoty polynómu $p(y) = K_0 \cdot y^x + 1 \cdot y^{x-1} + 1 \cdot y^{x-2} + \ldots + 1 \cdot x + 1$ v bode 2. Hodnota polynómu bude

$$K_x = K_0 \cdot 2^x + (2^{x-1} + 2^{x-2} + \ldots + 2 + 1) = K_0 \cdot 2^x + M,$$

kde $M = 2^{x-1} + 2^{x-2} + \ldots + 2 + 1$. M je súčet geometrickej postupnosti: $a_0 = 2^{x-1}$; $n = x$; $q = \frac{1}{2}$,

$$M = a_0 \cdot \frac{q^n - 1}{q - 1} = 2^{x-1} \cdot \frac{1 - \frac{1}{2^x}}{\frac{1}{2}} = 2^x \cdot \frac{2^x - 1}{2^x} = 2^x - 1,$$

teda $K_x = K_0 \cdot 2^x + 2^x - 1$, čo bolo treba dokázať.

(1) Dokážme teraz, že hodnota K sa zmenší v každom priechode vonkajším cyklom o jedna, z čoho vyplýva, že hodnota *PocVel* bude $K - 1$:

```
Program Mopi3;
  Var K,X,PocVel,PocMal:Integer;

Begin
  Write('K:'); Readln(K);
  PocVel:=0; PocMal:=0;
  X:=0;
  While K>1 Do
    Begin
      PocVel:=PocVel + 1;      K = m.2^j; m mod 2 = 1; taký
                               zápis musí existovať, lebo K > 1
      While (K Mod 2)=0 Do     bude splnené, až keď K = m,
                               lebo K sa jedine delí dvoma
        Begin
          X:=X+1; K:=K Div 2   K = m.2^(j-1), m.2^(j-2) ... m.2^0
                               K_X = m.2^(j-X)
                               ak K = m.2^j1, tak X = j - j1
        End;                   K = m; X = j
      K:=K-1;                  K = m - 1
      While X>0 Do             opakuj X-krát
        Begin
          PocMal:=PocMal + 1; zväčši PocMal o 1 ⇒
                               na konci PocMal = PocMal + X
          K:=K*2+1; X:=X-1
        End                    viď hore: K_X = (m-1).2^x + 2^x - 1 =
                               = m.2^x - 1 = K - 1
                               Vo veľkom cykle sa K zmenší o 1.
                               K teda nadobúda hodnoty
                               K, K-1, K-2, ... 3, 2.
                               Vo veľkom cykle sa zároveň PocVel
                               zväčší o 1 ⇒ PocVel = K - 1
    End;
  Writeln('Pocet velkych prechodov bol:',PocVel);
  Writeln('Pocet malych  prechodov bol:',PocMal)
End.
```

(2) Hodnotu *PocMal* určíme nasledovne: Vieme, že veľký cyklus prebehne $K-1$ krát s hodnotami K: $K, K-1, K-2, \ldots, 3, 2, 1$. Ďalej vieme, že *PocMal* sa v každom cykle zväčší o i, kde $K = m \cdot 2^i$, $m \bmod 2 = 1$. Teda pre K deliteľné 2^i sa inkrementuje o i, ak 2^i je maximálna mocnina 2, ktorá delí K. Medzi číslami od 1 po K je deliteľných 2^i práve $\left[\frac{K}{2^i}\right]$, to znamená, že pri práve $\left[\frac{K}{2^i}\right]$ hodnotách premennej K sa *PocMal* zväčší o i. Celková hodnota *PocMal* je teda

$$PocMal = \sum_{i=1}^{L} i \cdot \left[\frac{K}{2^i}\right], \qquad 2^L \leqq K < 2^{L+1} \quad (L = [\log_2 K]).$$

V tomto súčte sme ale niektoré hodnoty zarátali viackrát, lebo hodnota K, ktorá je deliteľná 2^i $(i > 1)$, je deliteľná aj $2^{i-1}, \ldots, 2^1$. Nech $1 \leqq j < i$. Potom každé 2^{i-j}-te číslo deliteľné 2^j je deliteľné aj 2^i. Teda správna hodnota *PocMal* bude

$$\begin{aligned} PocMal &= \sum_{i=1}^{L} \left(i \cdot \left[\frac{K}{2^i}\right] - \sum_{j=1}^{i-1} \left[\frac{\left[\frac{K}{2^{i-j}}\right]}{2^j}\right]\right) = \\ &= \sum_{i=1}^{L} \left(i \cdot \left[\frac{K}{2^i}\right] - \sum_{j=1}^{i-1} \left[\frac{K}{2^i}\right]\right) = \\ &= \sum_{i=1}^{L} \left[\frac{K}{2^i}\right]. \end{aligned}$$

Tento výraz lepšie pochopíme nasledovne: Ak 2^i je najväčšia mocnina 2, ktorá delí K, tak číslo i je vlastne počet núl na konci čísla K zapísaného v dvojkovej sústave.

Poslednou časťou dôkazu je dokázať, že $\left[\frac{K}{2}\right] + \left[\frac{K}{4}\right] + \left[\frac{K}{8}\right] + \ldots + \left[\frac{K}{2^L}\right] \leqq K - 1$ a že rovnosť nastáva práve vtedy, keď $K = 2^L$.

Nech

$$\begin{aligned} K &= 2^{A_1} + 2^{A_2} + 2^{A_3} + \ldots + 2^{A_r}, \\ A_1 &= LA_1 > A_2 > A_3 > \ldots > A_r \end{aligned}$$

Potom

$$\left[\frac{K}{2}\right] = 2^{A_1-1} + 2^{A_2-1} + 2^{A_3-1} + \ldots + 2^{A_r-1} \tag{1}$$

$$\left[\frac{K}{4}\right] = 2^{A_1-2} + 2^{A_2-2} + 2^{A_3-2} + \ldots + 2^{A_r-2} \tag{2}$$

$$\ldots\ldots\ldots\ldots\ldots$$

$$\left[\frac{K}{2^{A_r}}\right] = 2^{A_1-A_r} + 2^{A_2-A_r} + 2^{A_3-A_r} + \ldots + 1 \qquad (A_r)$$

$$\ldots\ldots\ldots\ldots\ldots$$

$$\left[\frac{K}{2^{A_{r-1}}}\right] = 2^{A_1-A_{r-1}} + 2^{A_2-A_{r-1}} + 2^{A_3-A_{r-1}} + \ldots + 0 \qquad (A_{r-1})$$

$$\ldots\ldots\ldots\ldots\ldots$$

$$\left[\frac{K}{2^{A_1}}\right] = 1 + 0 + 0 + \ldots + 0 \qquad (A_1)$$

Spolu $2^{A_1} - 1 + 2^{A_2} - 1 + 2^{A_3} - 1 + \ldots + 2^{A_r} - 1 =$
$= K - r,$

ale $r = 1$ len pre $K = 2^L$, čo bolo treba dokázať.

2. spôsob.

$$PocMal = \sum_{i=1}^{L} \left[\frac{K}{2^i}\right] \leqq \sum_{i=1}^{L} \frac{K}{2^i} = K \cdot \sum_{i=1}^{L} 2^{-i} = K - \frac{K}{2^L},$$

L bolo ale definované tak, že $2^L \leqq K$, teda $PocMal \leqq K - 1$. Rovnosť nastáva práve pre $K = 2^L$. Vtedy

$$\sum_{i=1}^{L} \left[\frac{K}{2^i}\right] = \sum_{i=1}^{L} \frac{K}{2^i},$$

a teda $K - \frac{K}{2^L} = K - \frac{2^L}{2^L} = K - 1$, čo bolo treba dokázať.

P – I – 4

Nech A, B sú čísla, ktorým zodpovedajú dvojkové zápisy

$$A = a_m 2^m + a_{m-1} 2^{m-1} + \ldots + a_0 2^0,$$
$$B = b_n 2^n + b_{n-1} 2^{n-1} + \ldots + b_0 2^0,$$

a nech w_1, w_2 sú zápisy čísel A, B v dvojkovej sústave v obrátenom poradí cifier, teda $w_1 = a_0 a_1 \dots a_m$, $w_2 = b_0 b_1 \dots b_n$.

Súčet C čísel A a B môžeme určiť nasledujúcim postupom:

Vyhodnotením výrazu $a_0 + b_0$ získame prvú cifru c_0 zápisu súčtu v obrátenom poradí cifier a prvý prenos p_0 do vyššieho rádu, pretože $a_0 + b_0 = c_0 + 2p_0$. Podobne $a_1 + b_1 + p_0 = c_1 + 2p_1$. Takto môžeme vyjadriť všetky cifry súčtu. Dôležité pritom je, že na získanie c_i potrebujeme poznať iba a_i, b_i a p_{i-1}. Tieto tri vstupné hodnoty môžu byť určené znakom na začiatku frontu (a_i), na vstupe (b_i) a stavom programu (p_{i-1}). Keďže všetky tri môžu nadobúdať iba hodnoty 0 alebo 1, potrebujeme na správne rozlíšenie dva rôzne stavy programu (stav s prenosom 0 a stav s prenosom 1). Z toho už priamo vyplýva návrh štruktúry programu:

(1) Načítame slovo w_1 reprezentujúce číslo A do frontu.
(2) Výpočet začneme v stave s prenosom 0.
(3) Prečítame a_i (z frontu) a b_i (zo vstupu). Vyhodnotíme $a_i + b_i + p_{i-1}$. Párnosť výsledku udáva i-tu cifru výsledku c_i. Podľa toho, či nastal prenos, pokračujeme v príslušnom stave.
(4) Krok 3 opakujeme, kým sme neprečítali obe čísla, teda $\max(m, n)$-krát, pričom za „chýbajúce cifry" a_i (b_j), ak $i > m$ ($j > n$), použijeme 0.

Cifry čísla A čítame dvakrát (prvýkrát pri kopírovaní vstupu na front, druhýkrát pri samotnom výpočte), cifry čísla B iba raz; časová zložitosť popísaného algoritmu je teda lineárna vzhľadom na dĺžku vstupného slova $w_1 * w_2$.

Jeden z možných zápisov popísaného algoritmu v jazyku Froscal:

```
  writeln;                    načítanie slova w1 do frontu
  while INP<>'*' do begin PUT(INP); NEXT end;
  NEXT                        preskočenie oddeľovača *
0:                            stav bez prenosu
  while INP<>'?' do
    case INP of
      '0': begin
             NEXT;
             if TOP='1' then write('1') else write('0');
             GET
           end;
      '1': begin
             NEXT;
             if TOP='1' then begin write('0'); GET; goto 1 end;
```

```
            write('1');
            GET
          end
      end;
  goto 3;
1:                              stav s prenosom
  while INP<>'?' do
    case INP of
      '0': begin
             NEXT;
             if TOP='1' then write('1')
             else begin write('1'); GET; goto 0 end;
             GET
           end;
      '1': begin
             NEXT;
             if TOP='1' then write('1') else write('0');
             GET
           end
    end;
  while TOP='1' do
    begin
      write('0');
      GET
    end;
  write('1');
  GET;
3:
  while TOP<>'?' do
    begin
      write(TOP);
      GET
    end;
end.
```

Základná myšlienka počítania súčinu dvoch čísel zapísaných v dvojkovej sústave spočíva v spočítavaní mocnín jedného z činiteľov. Nech A, B sú čísla vyjadrené rovnako ako v časti a), nech S je ich súčin. Platí

$$\begin{aligned} S = A \cdot B = B \cdot A = \\ = (b_n 2^n + b_{n-1} 2^{n-1} + \ldots + b_0 2^0) \cdot A = \\ = b_n (2^n A) + b_{n-1} (2^{n-1} A) + \ldots + b_0 (2^0 A). \end{aligned}$$

Čísla b_i nadobúdajú hodnoty 1 alebo 0, určujú teda, či príslušný člen $2^i A$ do súčinu pripočítame alebo nie. Stačí teda postupne generovať čísla A,

$2A, 2^2A, \ldots, 2^nA$, po každom kroku prečítať príslušnú cifru čísla B, a ak je nenulová, pripočítať násobok A k súčinu. Číslo B teda netreba nikde uchovávať; každá jeho cifra prislúcha práve jednému násobku čísla A. Ak tieto násobky vieme generovať v potrebnom poradí (neskôr ukážeme, že áno), potom treba každé b_i použiť práve raz, môžu sa preto čítať priamo zo vstupu. Číslo A však treba uchovať na front. Spolu s týmto číslom je potrebné pamätať si vo fronte aj medzivýsledok sčitovania. Je vhodné ukladať tieto dve čísla tak, že pri sebe stoja vždy cifry rovnakého rádu, teda mať na fronte postupnosť $a_0 s_0 a_1 s_1 \ldots a_m s_m$, kde s_i sú cifry medzivýsledku. Takýto tvar uloženia čísel prináša výhodu, že pri sčitovaní násobku čísla A a medzivýsledku stačí prejsť frontom iba raz; takisto zdvojnásobenie čísla A (čo je vlastne posunutie cifier a_i o jedno miesto vpravo) vyžaduje iba jeden priechod frontom. Na vypočítanie celého súčinu je preto potrebných rádovo $m \cdot n$ základných frontových operácií.

Popíšme teraz bližšie samotnú implementáciu násobenia vo Froscale: Najskôr načítame číslo A do frontu a inicializujeme medzisúčet.

```
repeat
  PUT(INP); PUT('0'); NEXT
until INP='*';
NEXT;
```

Na fronte je teraz $a_0 0 a_1 0 a_2 0 \ldots a_m 0$. Tu začína samotné násobenie. Ak sa v čísle B vyskytuje cifra 1, ktorá zodpovedá práve sa vyskytujúcemu násobku čísla A na fronte, treba tento násobok pripočítať k dosiahnutému medzisúčtu. Algoritmus sčítania je totožný s riešením úlohy a) s tou výnimkou, že obe čísla sú teraz uložené na fronte.

```
repeat
  if INP='1' then
    begin
      PUT('*');
0:    if TOP='*' then goto 2;
      PUT(TOP);
      case TOP of
        '0': begin GET; PUT(TOP); GET; goto 0 end;
        '1': begin
               GET;
               case TOP of
                 '0': begin PUT('1'); GET; goto 0 end;
                 '1': begin PUT('0'); GET; goto 1 end
               end
             end
      end;
```

```
1:      if TOP='*' then begin PUT('0'); PUT('1'); goto 2 end;
        PUT(TOP);
        case TOP of
          '0': begin
                 GET;
                 case TOP of
                   '0': begin PUT('1'); GET; goto 0 end;
                   '1': begin PUT('0'); GET; goto 1 end
                 end
               end;
          '1': begin GET; PUT(TOP); GET; goto 1 end
        end;
2:      GET
      end;
```

Vytvoríme ďalší násobok čísla A. Znamená to, že súčasný násobok potrebujeme prenásobiť dvoma, čiže posunúť o jednu cifru vpravo vzhľadom na medzisúčet. To je ale to isté, ako keď posunieme medzisúčet o jednu cifru vľavo. Najnižšia cifra medzisúčtu, ktorá by takto zanikla, už patrí hľadanému súčinu, pretože všetky ďalšie pripočítavané násobky čísla A majú v dvojkovom zápise na konci dostatočný počet núl. Môžeme preto túto cifru vypísať.

```
    PUT('*');
    case TOP of
      '0': begin GET; write(TOP); GET; goto 10 end;
      '1': begin GET; write(TOP); GET; goto 11 end
    end;
10: if TOP='*' then goto 20;
    PUT('0');
    goto 19;
11: if TOP='*' then begin PUT('1'); PUT('0'); goto 20 end;
    PUT('1');
19: case TOP of
      '0': begin GET; PUT(TOP); GET; goto 10 end;
      '1': begin GET; PUT(TOP); GET; goto 11 end
    end;
20: GET;
```

Z pôvodného stavu frontu $a_0 s_i a_1 s_{i+1} \ldots a_{m-1} s_{i+m-1} a_m s_{i+m}$ sme teda získali $a_0 s_{i+1} a_1 s_{i+2} \ldots a_{m-1} s_{i+m} a_m 0$ a môžeme pokračovať v spracovávaní ďalšej cifry čísla B.

```
    NEXT
  until INP='?';
```

Prečítali sme obe čísla, spočítali sme ich súčin a vypísali sme jeho

menej významné cifry. Najvyššie cifry ešte ostali na fronte; treba ich vypísať. Ak je najvýznamnejšia cifra súčinu nula, nevypisujeme ju.

```
GET;
while TOP in ['0','1'] do
  case TOP of
    '0': begin GET; if TOP<>'?' then write('0'); GET end;
    '1': begin GET; GET; write('1') end
  end;
writeln
```

Ak by sme čísla na fronte neuchovávali „pomiešané", teda cifry rovnakého rádu pri sebe, ale najprv jedno číslo a potom druhé číslo — zložitosť by vzrástla na $m^2 \cdot n$.

Pri riešení časti c) si treba uvedomiť, ako sa zmenia počiatočné podmienky, ktoré sme položili pre naše riešenia a) a b).

Pre úlohu a) musíme namiesto štyroch kombinácií vstupných cifier uvažovať o 100 kombináciách. Znamená to, že každý príkaz case bude mať desať vetiev. Sčítavame vždy tri desiatkové číslice. Dve z nich patria zadaným číslam, ich súčet je najviac 18. V prvom kroku môžeme dostať prenos najviac 1, v druhom kroku teda max. súčet 19, čiže zase prenos 1. Je zrejmé, že prenos 2 nenastane nikdy, stačia preto dva stavy na vyjadrenie prenosu.

V úlohe b) sa rovnako rozšíri blok zabezpečujúci sčítanie. Všetky ostatné časti programu sa zmenia do tej miery, že namiesto dvoch cifier musíme rozlišovať cifier desať. Predpoklady riešenia zmena základu číselnej sústavy nemení, preto myšlienka riešenia zostáva rovnaká.

P – II – 1

Permutáciu P môžeme rozložiť na niekoľko navzájom disjunktných cyklov, $P = C_1 C_2 \ldots C_m$. Každý cyklus tvoria nejaké prvky množiny $\{1, \ldots, N\}$, označme preto $C_i = (a_{i,1} a_{i,2} \ldots a_{i,l_i})$. Číslo l_i nazveme dĺžkou cyklu C_i.

Cykly sú navzájom disjunktné, takže každý prvok $x \in \{1, \ldots, N\}$ patrí práve do jedného cyklu. Preto ak zoberieme ľubovoľné číslo e_i pre každý cyklus C_i, tak x patrí práve do jedného zo zložení $C_i^{e_i}$.

Nie je ťažké ukázať, že pre cyklus C_i dĺžky l_i a pre každé x spomedzi prvkov cyklu $a_{i,1}, \ldots, a_{i,l_i}$ platí

$$C_i^u[x] = C_i^v[x] = l_i \mid (u - v).$$

Ďalej ak $P = C_1 C_2 \ldots C_m$, tak $P^K = C_1^K C_2^K \ldots C_m^K$. Pretože K je oveľa väčšie ako N (a teda aj ako l_i), je výhodnejšie zapísať ako $P = C_1^{e_1} C_2^{e_2} \ldots C_m^{e_m}$, kde $l_i \mid (K - e_i)$. Čím menšie budú čísla e_i, tým menej operácií budeme potrebovať na výpočet $C_i^{e_i}$, a tým bude náš algoritmus efektívnejší.

Zoberme $e_i = K \bmod l_i$. Potom $C_i^{e_i}[x] = C_i^K[x]$ pre každé zmysluplné x. Budeme preto počítať e_i-te zloženia cyklov C_i pre každé $i \in \{1, \ldots, m\}$. Zjednotenie týchto „čiastočných zložení" bude K-te zloženie permutácie P.

Na vypočítanie $C_i^{e_i}$ si stačí uchovať pôvodné hodnoty prvkov poľa P prislúchajúce danému cyklu C_i v pomocnom poli (H) a potom ich z tohoto poľa späť ukladať do poľa P „posunuté o e_i", teda $P^K[x] = H[(x + e_i - 1) \bmod l_i + 1]$ (na ukladanie výsledku môžeme použiť opäť pole P, lebo cykly sú navzájom disjunktné).

Takéto riešenie spracuje každý prvok každého cyklu práve raz, preto je jeho časová zložitosť lineárna vzhľadom na N. Pamäťová zložitosť je taktiež lineárna, pretože v pomocnom poli potrebujeme uchovávať naraz vždy iba jeden cyklus s maximálnou dĺžkou N. Na vyhľadanie začiatku ďalšieho cyklu je použitá „znamienkovacia" metóda, teda prvky už spracovaných cyklov dostanú dočasne záporné znamienko, aby sa odlíšili od dosiaľ nespracovaných.

```
program Permutacie;
type pole = array [1..100] of integer;
var p       : pole;
    k,n     : integer;
    i,prvy  : integer;

procedure cyklus (prvy:integer);
var h             : pole;
    i,dalsi,d,s : integer;
begin
  i:=prvy;
  d:=0;
  repeat
    inc(d);
    h[d]:=p[i];
    i:=p[i]
  until i=prvy;
  s:=k mod d;
  i:=prvy;
  repeat
    dalsi:=p[i];
```

```
    p[i]:=-h[s];
    i:=dalsi;
    s:=(s mod d)+1
  until i=prvy
end;

begin
  write('Zadaj N: '); read(n);
  write('Zadaj P: '); for i:=1 to n do read(p[i]);
  write('Zadaj K: '); read(k);
  for prvy:=1 to n do
    if p[prvy]>0 then cyklus(prvy);
  for i:=1 to n do p[i]:=-p[i];
  write('P^',k,':');
  for i:=1 to n do write(' ',p[i]); writeln
end.
```

P – II – 2

Najvýznamnejšie cifry kódu budeme zrejme spracovávať ako prvé, preto je vhodné obrátiť poradie cifier v kódoch $s1$ aj $s2$. Pôvodne prvá cifra sa potom ľahko zistí ako kód mod 10.

Kým sa kódy vo svojich cifrách (teraz už od konca) zhodujú, ležia oba zadané štvorce v tom istom typizovanom štvorci (nie je podstatné, v ktorom), a nemôžeme preto začať rozhodovať o susedstve. Zhodné cifry preto odstránime. Ak jeden zo zvyšných kódov je 0 (a ten pokrýva najmenší typizovaný štvorec, v ktorom sa oba zadané štvorce nachádzajú), nemôžu sa už dotýkať zvonku; takisto je tomu v prípade, že ležia v protiľahlých kvadrantoch (1 a 4 alebo 2 a 3, vždy teda súčet 5). Tieto triviálne prípady preto vylúčime.

Teraz môžu nastať dva prípady: štvorce budú mať spoločnú buď vodorovnú hranu (ležia v kvadrantoch 1 a 3, resp. 2 a 4), alebo zvislú hranu (1 a 2, resp. 3 a 4). Na vzájomnom poradí štvorcov pritom nezáleží, preto bez ujmy na všeobecnosti riešenia nech kód $s1$ prislúcha štvorcu ležiacemu v kvadrante s nižším číslom (teda „ľavejšiemu", resp. „vrchnejšiemu" z oboch štvorcov).

Spomínané dva prípady rozlíšime vyhodnotením rozdielu posledných cifier zostávajúcich kódov. Pre každú z oboch možností existuje odteraz jednoznačný algoritmus na zistenie susedstva. Rozoberieme podrobnejšie iba prípad susedstva zvislou hranou; druhý prípad je obdobný:

Štvorec $s1$ má poslednú cifru kódu 1 a štvorec $s2$ dvojku (alebo 3 a 4) — pozor! tu už záleží na poradí týchto cifier. Potom jediné prípustné

kombinácie cifier na ďalšom mieste sú 2 a 1 alebo 4 a 3, inak sa susedstvo poruší. (Kód 0 je ekvivalentný všetkým kódom, teda napr. aj kombinácia 2, 0 je prípustná.)

V programe je definované pole *Test*. Jeho prvým indexom je druh susedstva (vodorovne/zvislo), druhý index je posledná cifra kódu $s1$ a tretí index posledná cifra kódu $s2$. Toto pole predstavuje tabuľku stavov nejakého konečného automatu a je definované takto:

0 – oba kódy sú celé spracované a nezistilo sa, že štvorce nesusedia, tedy štvorce susedia.

1 – dosiaľ spracované časti kódov zodpovedali susediacim dvojiciam štvorcov. Tento fakt susedstvo nepopiera, ale ani nepotvrdzuje, preto treba testovať ďalšie cifry.

2 – štvorce nesusedia

```
program Stvorce;
const Test : array [0..1,0..4,0..4] of 0..2 =
 (((0,1,1,2,2),(2,2,2,2,2),(2,2,2,2,2),(1,1,2,2,2),(1,2,1,1,1)),
 ((0,1,2,1,2),(2,2,2,2,2),(1,1,2,2,2),(2,2,2,2,2),(1,2,2,1,2)));
var S1,S2,S1kon,S2kon,Pom,T : integer;

function Prevrat (S:integer) : integer;
var p : integer;
begin
  p:=0;
  while S>0 do
    begin
      p:=10*p+S mod 10;
      S:=S div 10
    end;
  Prevrat:=p
end;

begin
  write('Zadaj S1: '); readln(S1);
  write('Zadaj S2: '); readln(S2);
  S1:=Prevrat(S1);
  S2:=Prevrat(S2);
  while (S1 mod 10) = (S2 mod 10) do
    begin
      S1:=S1 div 10;
      S2:=S2 div 10
    end;
  S1kon:=S1 mod 10;
  S2kon:=S2 mod 10;
  if (S1kon=0) or (S2kon=0) or (S1kon+S2kon = 5) then T:=2
  { Keď jeden leží v druhom alebo sú v uhlopriečnych štvorcoch.}
```

```
  else
    begin
      if S1kon>S2kon then
        begin Pom:=S1; S1:=S2; S2:=Pom end;
      Pom:=abs(S1kon-S2kon) mod 2;
      repeat
        S1:=S1 div 10;
        S2:=S2 div 10;
        T:=Test[Pom, S1 mod 10, S2 mod 10];
      until T<>1
    end;
  if T=0 then writeln('Lezia pri sebe!')
         else writeln('Nelezia pri sebe!')
end.
```

P – II – 3

Pojmom *lokálny počet inverzií* znakov Z_1 a Z_2 v reťazci R rozumieme počet takých výskytov dvojíc Z_1 a Z_2 v reťazci R, že medzi nimi sa nenachádzajú iné znaky ako Z_1 a Z_2. Ľahko sa nahliadne, že hľadané znaky zo zadania sú práve také dva znaky, ktoré majú maximálny lokálny počet inverzií.

Skúsme si teraz predstaviť, že máme vytvorenú tabuľku (dvojrozmernú) lokálnych inverzií pre nejaký začiatočný úsek reťazca R. Pridaním ďalšieho znaku (`inp`) môžu nastať tri prípady:

(1) Nový znak je zhodný s posledným (`posl2`) a nech pred ním existuje znak rôzny od neho (`posl1`). Vtedy stačí opraviť tabuľku inverzií iba na mieste [`posl1,inp`] pričítaním počtu výskytov znaku `posl1` na konci prerušovaných nanajvýš výskytmi znaku `inp`.

(2) Nový znak je zhodný s predposledným znakom `posl1`. Vtedy stačí na mieste [`posl2,inp`] pričítať počet výskytov `posl2` (posledného znaku) prerušovaných nanajvýš znakom `posl1`.

(3) Nový znak je rôzny od posledného (`posl2`) aj od `posl1`. Vtedy stačí tabuľku inverzií poopraviť na mieste [`posl2,inp`] o veľkosť posledného súvislého úseku znakov `posl2`.

Z uvedeného vidíme, že by bolo vhodné si viesť priebežne hodnoty posledných dvoch rôznych znakov `posl1`, `posl2`, ich počty posledných výskytov prerušovaných len navzájom (`poc1`, `poc2`), a veľkosť posledného súvislého úseku posledného znaku (`poc2posl`). Po prepísaní do programu dostávame lineárny algoritmus s poľom konštantnej veľkosti.

Hľadanie maximálnej hodnoty sa dá robiť priebežne. Osobitne treba ešte vyriešiť inicializáciu a začiatok, lebo v rozbore sa predpokladá, že už máme aspoň dva rôzne znaky spracované.

```
program Znaky;
 var inp,posl1,posl2,max1,max2 : char;
     poc1,poc2,poc2posl,pom    : integer;
     inver : array['A'..'J','A'..'J'] of integer;

begin
  for max1:='A' to 'J' do
    for max2:='A' to 'J' do
      inver[max1,max2]:=0;                    {inicializácia}
  max1:='A'; max2:='A';
  read(posl1); poc1:=0;                       {prvé načítanie}
  repeat inc(poc1); read(posl2) until posl2<>posl1;
  poc2:=1; poc2posl:=1;
  while not(eoln) do                          {hlavný cyklus}
    begin
      read(inp);
      if inp=posl2 then                       {prípad (1)}
        begin inc(poc2); inc(poc2posl) end
      else if inp=posl1 then                  {prípad (2)}
        begin
          posl1:=posl2; posl2:=inp;
          pom:=poc2; poc2:=poc1+1; poc1:=pom;
          poc2posl:=1;
        end
      else                                    {iný znak --- (3)}
        begin
          posl1:=posl2; poc1:=poc2posl;
          posl2:=inp; poc2:=1; poc2posl:=1;
        end;
      pom:=inver[posl1,posl2]+poc1;           {zvýšime počet }
      inver[posl1,posl2]:=pom;                { inverzií.    }
      if pom>inver[max1,max2] then            {je maximálny ?}
        begin max1:=posl1; max2:=posl2 end;
    end;
  writeln('Z1=',max1,'  Z2=',max2)
end.
```

P – II – 4

Dolná celá časť dvojkového logaritmu čísla je rovná počtu cifier v zápise tohoto čísla v dvojkovej sústave mínus 1. Tento počet cifier môžeme zistiť tak, že číslo celočíselne delíme dvoma. Počet delení, potrebný na to, aby

sme získali výsledok nula, sa rovná počtu cifier. Potrebujeme ale počet cifier mínus jedna, delenie preto skončíme, ak je výsledok menší ako 2.

Pre zadaný tvar vstupného slova, keď počet znakov 'a' udáva hodnotu čísla, sa delenie dvoma realizuje ľahko tak, že pre každú dvojicu po sebe idúcich znakov ('aa') do medzivýsledku pridáme iba jeden znak 'a'; tým získame číslo reprezentované polovičným počtom znakov, teda v našom zápise s polovičnou hodnotou. Súčasne zvyšujeme počítadlo, ktoré udáva počet vykonaných delení. Ak dosiahneme pri delení výsledok 0 alebo 1, náš výpočet končí. Stav počítadla potom udáva dolnú celú časť dvojkového logaritmu zadaného čísla.

```
program Logaritmus;
begin
  repeat PUT(INP); NEXT until INP='?';
  PUT('*'); PUT('0'); PUT('*');
  repeat          {redukuj počet 'a' na polovicu a zvýš čítač}
    GET;
    if TOP<>'*' then                    {aspoň 2 znaky}
      begin
        GET; PUT('a');
        while TOP<>'*' do               {každú celú}
          begin                         {dvojicu 'aa'}
            GET;                        {zmeň na 'a'}
            if TOP<>'*' then begin GET; PUT('a') end
          end;
        GET; PUT('*');
        while TOP='1' do begin GET; PUT('0') end; {zvýš čítač}
        PUT('1'); if TOP<>'*' then GET;
        while TOP<>'*' do begin PUT(TOP); GET end;
        GET; PUT('*')                   {čítač zvýšený}
      end
  until TOP='*';
  GET; repeat write(TOP); GET until TOP='*'
end.
```

Časová zložitosť tohoto riešenia je $O(2N+\log_2 N \cdot \log_2 \log_2 N)$, pamäťová zložitosť (maximálna veľkosť frontu) je $N+\log_2 \log_2 N$; N je dĺžka slova w.

P – III – 1

Pri riešení je dôležité si uvedomiť, čo sa s permutáciou deje, ak ju umocňujeme (teda vytvárame jej druhú mocninu), konkrétne nás zaujíma, čo sa deje s jej jednotlivými cyklami. Vieme, že cyklus nepárnej dĺžky bude po umocnení opäť cyklom (s rovnakou dĺžkou). Ale každý cyklus párnej

dĺžky sa po umocnení rozpadne na dva cykly polovičnej dĺžky. Z toho môžeme s určitosťou tvrdiť, že ak sa v druhej mocnine vyskytuje cyklus párnej dĺžky, vznikol rozpadnutím cyklu dvojnásobnej dĺžky v pôvodnej permutácii. Ak sa v druhej mocnine vyskytuje cyklus nepárnej dĺžky, mohol vzniknúť z cyklu rovnakej dĺžky, alebo rozpadom cyklu dĺžky dvojnásobnej.

Spočítajme v druhej mocnine počty cyklov jednotlivých dĺžok. V ďalšom texte budeme L označovať dĺžku cyklu a $K(L)$ počet cyklov tejto dĺžky. Zrejme cyklom dvoch rozdielnych dĺžok môžeme prisúdiť nezávislé spôsoby vzniku, a teda pri stanovovaní počtu druhých odmocnín, teda počtu permutácií, z ktorých po ich umocnení vznikla zadaná permutácia, musíme vyčísliť súčin počtov spôsobov vzniku skupín cyklov jednotlivých dĺžok. Ak označíme $S(L)$ počet spôsobov, ktorými mohli vzniknúť cykly dĺžky L, môžeme písať

$$\text{počet druhých odmocnín} = \prod S(L),$$

pričom násobíme cez všetky L, pre ktoré $K(L) > 0$.

Našou úlohou ostáva stanoviť $S(L)$ pre jednotlivé L. Predchádzajúce úvahy vedú k rozdeleniu na dva prípady podľa párnosti L:

(a) V prípade, že $L = 2m + 1$, teda L je nepárne, stanovíme $S(L)$ takto:
- jedna možnosť je, že všetky cykly vznikli zobrazením z cyklov rovnakej dĺžky (teda ani jeden nevznikol rozpadom dlhšieho cyklu) — príspevok do $S(L)$ je 1
- druhá možnosť je, že práve dva z $K(L)$ cyklov vznikli rozpadom cyklu dĺžky $2L$; týchto možností je $\binom{K(L)}{2}$. Lenže rozpadom nejakého cyklu V (dĺžky $2L$) vzniknú rovnaké cykly W_1 a W_2, ako rozpadom cyklu V', ktorý sa od V líši tým, že má cyklicky posunuté prvky na všetkých párnych (alebo všetkých nepárnych, čo je to isté) miestach; napr. z cyklu $V = 1 \to 2 \to 3 \to 4 \to 1$ vzniknú cykly $W_1 = 1 \to 3 \to 1$ a $W_2 = 2 \to 4 \to 2$, rovnako ako z cyklu $V' = 1 \to 4 \to 3 \to 2 \to 1$. Možných vzájomných posunutí cyklov W_1 a W_2 (teda možných cyklov V) je L, teda celkový príspevok bude $L\binom{K(L)}{2}$
- podobne mohli práve dve dvojice cyklov vzniknúť rozpadom cyklov dĺžky $2L$; analogickým odvodením získame príspevok $L^2 \cdot \frac{1}{2}\binom{K(L)}{2}\binom{K(L)-2}{2}$, atď.

Sčítaním týchto príspevkov dostávame $S(L)$:

$$S(L) = 1 + L\frac{\binom{K(L)}{2}}{1!} + L^2\frac{\binom{K(L)}{2}\binom{K(L)-2}{2}}{2!} + \\ + L^3\frac{\binom{K(L)}{2}\binom{K(L)-2}{2}\binom{K(L)-4}{2}}{3!} + \dots$$

pričom sčítame po prvý nulový sčítanec.

(b) V prípade, že $L = 2m$, teda L je párne, museli všetky cykly vzniknúť rozpadom cyklov dĺžky $2L$ (okrem iného to znamená, že $K(L)$ je párne, ale to je pre nás skoro nevyužiteľné). Potom $S(L)$ je počet spôsobov, ktorými môžeme „popárovať" $K(L)$ cyklov do dvojíc, krát $L^{\frac{1}{2}K(L)}$ (každá z $\frac{1}{2}K(L)$ dvojíc mohla vzniknúť z L rôznych cyklov; pozri úvahu v (a). Počet spôsobov vytvorenia dvojíc je $\big(K(L)-1\big)\big(K(L)-3\big)\dots 1$ (k prvému prvku môžeme vybrať druhý z $K(L)-1$ prvkov, ostalo $K(L)-2$ prvkov; k ďalšiemu prvému je to $\big(K(L)-2\big)-1 = K(L)-3$ spôsobov atď.). Teda

$$S(L) = \big(K(L)-1\big)\big(K(L)-3\big)\cdot\dots\cdot 1\cdot L^{\frac{1}{2}K(L)}$$

P – III – 2

Ak si uvedomíme, že dva typizované štvorce s kódmi dĺžky D a $D+1$ majú dĺžky svojich strán v pomere $2:1$, je prirodzené uvažovať o reprezentácii súradníc štvorca v dvojkovej sústave. Nech kód S má D cifier. Potom jemu zodpovedajúci štvorec je reprezentovaný dvoma súradnicami X a Y, ktoré majú v zápise v dvojkovej sústave D cifier. Priraďme týmto zápisom cifry nasledujúcim spôsobom (nech C_i je i-tá cifra čísla C):

$$\begin{aligned} X_i = 1 &\Leftrightarrow S_i = 2 \text{ alebo } S_i = 4, \\ X_i = 0 &\Leftrightarrow S_i = 1 \text{ alebo } S_i = 3, \\ Y_i = 1 &\Leftrightarrow S_i = 3 \text{ alebo } S_i = 4, \\ Y_i = 0 &\Leftrightarrow S_i = 1 \text{ alebo } S_i = 2, \quad 1 \leqq i \leqq D \end{aligned}$$

Je zrejmé, že ak rozdelíme celú obrazovku na typizované štvorce rovnakej veľkosti zhodné so štvorcom s kódom S, tak súradnice X (resp. Y) zodpovedajúce týmto štvorcom stúpajú v smere vpravo (resp. dolu). Potom posunutie štvorca s kódom S o K jeho dĺžok vpravo znamená pripočítanie čísla K k X-ovej súradnici tohoto štvorca. Podobne posunutie

o L dĺžok hore znamená odčítanie čísla L od Y-ovej súradnice. Po tejto úprave získame kód nového štvorca spätným prevodom zo súradníc X a Y na kód S:

$$S_i = 1 \Leftrightarrow X_i = 0 \text{ a } Y_i = 0,$$
$$S_i = 2 \Leftrightarrow X_i = 1 \text{ a } Y_i = 0,$$
$$S_i = 3 \Leftrightarrow X_i = 0 \text{ a } Y_i = 1,$$
$$S_i = 4 \Leftrightarrow X_i = 1 \text{ a } Y_i = 1, \quad 1 \leqq i \leqq D,$$

čo sa dá prehľadnejšie zapísať ako $S_i = 1 + X_i + 2Y_i$.

```
x:=0;
y:=0;
rad:=1;
while s>0 do
  begin
    last:=s mod 10;
    s:=s div 10;
    if (last=3) or (last=4) then y:=y+rad;
    if (last=2) or (last=4) then x:=x+rad;
    rad:=rad*2
  end;
x:=x+k;
y:=y-l;
if (x<0) or (y<0) or (x>=rad) or (y>=rad) then
  writeln('MIMO OBRAZOVKY')
else
  begin
    write('Po presunuti: ');
    repeat
      rad:=rad div 2;
      last:=1;
      if x>=rad then begin last:=last+1; x:=x-rad end;
      if y>=rad then begin last:=last+2; y:=y-rad end;
      write(last)
    until rad=1;
    writeln;
  end
```

P – III – 3

Pozrime sa, ako vyzerá dekrementovanie čísla v mínus-desiatkovej sústave.

Nech $A = (a_n a_{n-1} \ldots a_1 a_0)_{-10}$.
Ak $a_0 > 0$, potom $A - 1 = \overline{a_n a_{n-1} \ldots a_1 (a_0 - 1)}$.

Ak $a_0 = 0$ a $a_1 < 9$, potom $A - 1 = \overline{a_n a_{n-1} \ldots (a_1 + 1)9}$.
Ak $a_0 = 0$, $a_1 = 9$ a $a_2 > 0$, potom $A - 1 = \overline{a_n a_{n-1} \ldots (a_2 - 1)09}$.
...

Nech všeobecný algoritmus vyzerá nasledovne:

```
A) j:=0;
   while a[j] = 9*(j mod 2) do j:=j+1;
B) if j mod 2 = 1 then inc(a[j])
                  else dec(a[j])
C) for i:=j-1 downto 0 do a[i]:=9-a[i];
```

Ukážme si, že uvedený algoritmus robí presne to isté, čiže zmenšuje číslo v (-10)-sústave.

(1) Nech $A = \overline{a_n \ldots a_j 9090 \ldots 90}$, tzn. $j \bmod 2 = 0$. Potom $A = a_n \cdot (-10)^n + \ldots + a_j \cdot 10^j - 9 \cdot 10^{j-1} - 9 \cdot 10^{j-3} - \ldots - 90$. Uvedený algoritmus transformuje A na A' v tvare $A' = \overline{a_n \ldots (a_j - 1)09 \ldots 09}$, tzn. $A' = a_n \cdot (-10)^n + \ldots + (a_j - 1) \cdot 10^j + 9 \cdot 10^{j-2} + 9 \cdot 10^{j-4} + \ldots + 9$. Tvrdíme, že $A - A' = 1$:

$$\begin{aligned}
&(a_n \cdot (-10)^n + \ldots + a_j \cdot 10^j - 9 \cdot 10^{j-1} - 9 \cdot 10^{j-3} - \ldots - 90) - \\
&- (a_n \cdot (-10)^n + \ldots + a_j \cdot 10^j - 10^j + 9 \cdot 10^{j-2} + 9 \cdot 10^{j-4} + \\
&+ \ldots + 9) = \\
&= 10^j - 9 \cdot 10^{j-1} - 9 \cdot 10^{j-2} - \ldots - 90 - 9 = 1
\end{aligned}$$

(2) Nech $A = \overline{a_n \ldots a_j 090 \ldots 90}$, tzn. $j \bmod 2 = 1$. Potom $A = a_n \cdot (-10)^n + \ldots - a_j \cdot 10^j - 9 \cdot 10^{j-2} - 9 \cdot 10^{j-4} - \ldots - 90$. Uvedený algoritmus transformuje A na $A' = \overline{a_n \ldots (a_j + 1)909 \ldots 09}$, tzn. $A' = a_n \cdot (-10)^n + \ldots - (a_j + 1) \cdot 10^j + 9 \cdot 10^{j-1} + 9 \cdot 10^{j-3} + \ldots + 9$. Tvrdíme, že $A - A' = 1$:

$$\begin{aligned}
&(a_n \cdot (-10)^n + \ldots - a_j \cdot 10^j - 9 \cdot 10^{j-2} - 9 \cdot 10^{j-4} - \ldots - 90) - \\
&- (a_n \cdot (-10)^n + \ldots - a_j \cdot 10^j - 10^j + 9 \cdot 10^{j-1} + 9 \cdot 10^{j-3} + \\
&+ \ldots + 9) = \\
&= 10^j - 9 \cdot 10^{j-1} - 9 \cdot 10^{j-2} - \ldots - 90 - 9 = 1
\end{aligned}$$

Ukážme si teraz, že daný program vykoná v jednom cykle presne túto operáciu (zmenšenie čísla v (-10)-sústave o 1).

Cyklus A) zjavne vykonáva časť A) nášho algoritmu, pričom si ešte vytvára pomocnú premennú X: $X_{j-1} = a_0$, $X_{j-2} = a_1$, ..., $X_0 = a_{j-1}$. Keďže očakávame $a_0 = 0$, potom $X_{j-1} = 0$, tedy X má $j - 1$ cifier.

Premenná q predstavuje výraz $9 \cdot (j \bmod 2)$. Ak teda po skončení cyklu je $q = 9$, očakávali sme $a_j = 9$, teda $j \bmod 2 = 1$, podobne $q = 0$, a teda $j \bmod 2 = 0$.

Keď po skončení cyklu platí:

$q = 9$ a $k = 0$, $a_j = 0$; $a_{j-1} = 0$; $a_{j-2} = 9$; ...; znamená to, že A má zápornú hodnotu, nemôžeme ho zmenšovať, lebo by cyklus neskončil.

$q = 9$ a $k > 0$, $9 > a_j \geqq 0$. V tomto prípade sa vykonajú priradenia: $a_j = a_j + 1; a_{j-1} = 9$ (treba si uvedomiť, že v tomto prípade $x_0 = 0$), následne sa vykoná cyklus B), ktorý vykoná priradenia: $a_{j-2} = x_0$, $a_{j-3} = x_1$, ..., $a_0 = x_{j-2}$. Ak sa a_{j-1} sa rovnalo 0, teraz má hodnotu 9. Ak sa a_{j-2} rovnalo 9, teraz má hodnotu 0 (x_0). Ak sa a_0 rovnalo 0, teraz má hodnotu 9 (x_{j-2}).

$q = 0$ ($j \bmod 2 = 0$), platí $x_0 = 9$ ($a_{j-1} = 9$, $a_{j-2} = 0$, ...). V tomto prípade sa vykonajú priradenia: $a = a - 1$; $x_{j-1} = x_{j-2}$, $x_{j-2} = x_{j-3}$, ..., $x_1 = x_0$, $x_0 = 0$ (teraz bude $x_0 = 0$, $x_1 = 9$, ..., $x_{j-1} = 9$) $a_{j-1} = x_0$, $a_{j-2} = x_1$, ..., $a_0 = x_{j-1}$.

Opäť sa môžeme ľahko presvedčiť, že cifry na pozíciách $j - 1$ až 0 sa „zinvertovali". Iste ste si všimli, že predpokladáme priechod aj cyklom A), aj cyklom B). Vyjasnime si teda tieto krajné prípady:

(1) Počet priebehov prvým cyklom bol 0, tzn., že $a_0 \neq 0$. $A = Y + a_0$, teda $A - 1 = Y + (a_0 - 1)$. Presne takéto priradenie sa vykoná v podmienkovej časti, cyklus B) tiež neprebehne, výsledok je teda správny.

(2) Nech $a_0 = 0$, $a_1 \neq 9$. $A = Y - a_1 \cdot 10 + 0$, teda $A - 1 = Y - (a_1 + 1) \cdot 10 + 9$. Opäť sa presne takéto priradenie vykoná v podmienkovej časti, čiže aj v tomto prípade je výsledok správny.

Z uvedeného vyplýva, že uvedený program v cykle zmenšuje čísla v mínus-desiatkovej sústave o 1. Cyklus sa vykonáva, pokiaľ $K > 0$; *poc* určuje počet priechodov cyklu. Ak vstupné K je číslo v mínus-desiatkovej sústave, *poc* bude po skončení cyklu hodnota K (číslo v desiatkovej sústave). Ak chceme, aby výstup programu bol p ($poc = p$), musí byť vstupné K číslo p v mínus-desiatkovej sústave. Ťažisko úlohy je teda napísať program, ktorý vstupné p prevedie do mínus-desiatkovej sústavy.

Riešenie: Vytvorme polynóm $KK(-10)$, ktorého hodnota by bola p, nasledovne ($kk_0, kk_1, \ldots, kk_m$ sú koeficienty polynómu KK):

Pre všetky cifry p (v desiatkovej sústave):

Ak cifra p_i by v (-10)-sústave zodpovedala kladnej mocnine -10, tzn. $i \bmod 2 = 0$, potom vykonáme $kk_i = kk_i + p_i$.

Ak cifra p_i by v (-10)-sústave zodpovedala zápornej mocnine -10, tzn. $i \bmod 2 = 1$, potom túto cifru rozpíšeme ako $kk_{i+1} = kk_{i+1} + 1$, $kk_i = kk_i + 10 - p_i$, napr. $(20)_{10} = (180)_{-10}$.

Takto vytvorený polynóm má hodnotu p. Ak by sme chceli z koeficientov polynómu zostaviť číslo v mínus-desiatkovej sústave, treba, aby pre všetky i platilo $0 \leqq kk_i \leqq 9$.

Treba teda vymyslieť pravidlo, ktoré by zmenilo koeficienty polynómu tak, aby bola zabezpečená podmienka, ale nezmenilo by jeho hodnotu. Keďže koeficienty kk nemôžu byť záporné, stačí nám uvažovať koeficienty väčšie ako 9.

Nech a_j je posledný koeficient väčší ako 9 (pre všetky $i > j$ platí $a_i < 10$). Potom sa hodnota polynómu nezmení, ak vykonáme:

$kk_{j+1} = kk_{j+1} - 1$, $kk_j = kk_j \bmod 10$ $(c \cdot (-10)^x = (c-10) \cdot (-10)^x - (-10)^{x+1})$. Ak je ale $kk_{j+1} = 0$, dostali by sme záporný koeficient, čo predpokladáme, že sa nestane. Vtedy priradenia trochu obmeníme: $kk_j = kk_j \bmod 10$, $kk_{j+1} = 9kk_{j+2} = kk_{j+2} + 1$ $(c \cdot (-10)^j = (c-10) \cdot (-10)^j + 9 \cdot (-10)^{j+1} + (-10)^{j+2})$ $((-10) \cdot (-10)^{j+1} + 9 \cdot (-10)^{j+1} = -(-10)^{j+1} = 10 \cdot (-10)^j)$.

Použitím tohoto pravidla na každý koeficient polynómu väčší ako 9 dostaneme polynóm, ktorého koeficienty sú nezáporné a menšie ako 10, dá sa jednoduchým použitím koeficientov ako cifier dostať číslo v mínus-desiatkovej sústave, ktorého hodnota je p.

Je jasné, že pôvodný polynóm môže mať $l + 1$ koeficientov, kde l je počet cifier p v (10) sústave. Aplikovanie pravidla môže počet koeficientov zväčšiť maximálne o 2, teda maximálny počet cifier je $l + 3$, teda algoritmus je konečný.

```
const MAXCIF = 15;
var kk:array[0..MAXCIF] of integer;
    p,i,j:integer;
begin
  writeln;
  write('Zadaj P :'); readln(p);
  for i:=0 to MAXCIF do kk[i]:=0;
  i:=0;
  while p>0 do
    begin
      if odd(i) then
        begin
          kk[i]:=kk[i]+10-(p mod 10);
          kk[i+1]:=kk[i+1]+1
        end
```

```
      else
        kk[i]:=kk[i] + (p mod 10);
      inc(i); p:=p div 10
    end;
  for j:= 0 to i do
    if kk[j]>9 then
      begin
        if kk[j+1]>0 then dec(kk[j+1])
        else
          begin
            kk[j+1]:=9;
            kk[j+2]:=kk[j+2]+1;
            if j+2>i then i:=j+2;
          end;
        kk[j]:=kk[j]-10
      end;
  k:=0;
  for j:= i downto 0 do k:=10*k+kk[j];
  writeln('Vstupna hodnota musi byt ',k)
end.
```

P – III – 4

Základná myšlienka riešenia úlohy je založená na tom, že počas čítania slova w_2 si budeme v slove w_1 pamätať, pokiaľ až sa dosiaľ načítaná časť w_2 zhoduje so začiatkom w_1.

Označme M dĺžku slova w_1, N dĺžku slova w_2. Keďže M je omnoho menšie ako N, môžeme ho považovať za konštantu.

Pre náš algoritmus je potrebné uchovať slovo w_1 vo fronte. Zvoľme nasledujúci tvar: Slovo w_1 uložíme do fronty a za každý znak, ktorý je posledným znakom dosiaľ zhodnej časti w_1 a w_2, uložme špeciálny znak `'.'`. Okrem toho budeme potrebovať mať uložené počítadlo výskytov w_1 vo w_2. Odlíšme toto počítadlo od dosiaľ použitej abecedy `'a'..'z'`, `'.'` použitím znakov `'0'`, `'1'`.

Zaveďme nasledujúce označenie: `Z` nech je ľubovoľný znak frontu z abecedy `'a'..'z'` (potom postupnosť `Z.` označuje znak frontu, ktorý je zatiaľ posledným zhodným znakom medzi w_1 a časťou w_2), `Q` nech je práve spracúvaný znak slova w_2, `C` nech je symbolické označenie počítadla výskytov, `D` nech je označenie počítadla reprezentujúceho hodnotu o 1 väčšiu ako `C`.

Vykonávanie algoritmu môžeme zapísať nasledujúcimi pravidlami:

(0) Z → Z., ak Z=Q a Z je prvý znak frontu (potenciálny začiatok výskytu w_1 vo w_2)
(1) .Z → Z., ak Z=Q (zhoduje sa aj ďalší znak)
(2) .Z → Z, ak Z≠Q (ďalší znak sa nezhoduje ⇒ nenašli sme podslovo)
(3) .C → D (celé w_1 sa vyskytovalo vo w_2 ⇒ zarátame výskyt)

Po prepísaní do Froscalu máme riešenie:

```
repeat                          okopírujeme w1 do frontu
  PUT(INP);
  NEXT
until INP='*';
NEXT;
PUT('0');                       zatiaľ bolo 0 výskytov
```

Nášmu označeniu zodpovedajú Q=INP, Z=TOP (lebo k inému znaku frontu ako k TOP nemáme prístup).

```
repeat
  PUT('*');                          zarážka proti zacykleniu
  PUT(TOP);
  if INP=TOP then PUT('.');          pravidlo (0)
  GET;


  while TOP<>'*' do
    begin
      if TOP='.' then                pravidlá (1,2) - začiatok
        begin
          GET;
          if TOP<>INP then begin PUT(TOP); GET end  (2)
          else
           begin
             PUT(TOP); GET;
             if (TOP='0') or (TOP='1') or (TOP='*') then
               begin                                  (2,3)
                while TOP='1' do begin PUT('0'); GET end;
                PUT('1');
                if TOP<>'*' then GET;
                while TOP<>'*' do begin PUT(TOP); GET end;
               end
             else PUT('.')                              (2)
           end
        end
      else begin PUT(TOP); GET end; ak sa nedá aplikovať
    end;          ani jedno pravidlo, zober ďalší znak
  GET;
  NEXT;                                          ďalšie Q
```

```
until INP='?';
```

Nakoniec vypíšeme výsledok v požadovanom tvare:

```
repeat
  if (TOP='0') or (TOP='1') then write(TOP);
  GET
until TOP='?'
```

Stanovenie zložitosti riešenia:

Vo fronte je uchovaných max. $L = 2M + 1 + \log V$ znakov (V je počet výskytov, $V \leqq N$). Keďže M môžeme považovať za konštantu, priestorová náročnosť navrhovaného algoritmu je logaritmická vzhľadom k dĺžke vstupného slova. Pre každý znak slova w_2 spracujeme každý znak frontu, celková časová zložitosť je teda $O(NL) = O(N \log V) = O(N \log N)$.